Scientific Discovery using Genetic Programming

Maarten Keijzer

LYNGBY 2001

IMM-PHD-xx

IMM

IMM
INFORMATICS AND MATHEMATICAL MODELLING

Technical University of Denmark
DK-2800 Lyngby – Denmark

May 1, 2002
M.K.

Scientific Discovery using Genetic Programming

Maarten Keijzer

LYNGBY 2001

IMM-PHD-xx

ISSN 0909–3192

Abstract

Genetic Programming is capable of automatically inducing symbolic computer programs on the basis of a set of examples or their performance in a simulation. Mathematical expressions are a well-defined subset of symbolic computer programs and are also suitable for optimization using the genetic programming paradigm. The induction of mathematical expressions based on data is called *symbolic regression*.

In this work, genetic programming is extended to not just fit the data i.e., get the numbers right, but also to get the dimensions right. For this *units of measurement* are used. The main contribution in this work can be summarized as:

> *The symbolic expressions produced by genetic programming can be made suitable for analysis and interpretation by using units of measurement to guide or restrict the search.*

To achieve this, the following has been accomplished:

- A standard genetic programming system is modified to be able to induce expressions that more-or-less abide type constraints. This system is used to implement a preferential bias towards dimensionally correct solutions.
- A novel genetic programming system is introduced that is able to induce expressions in languages that need context-sensitive constraints. It is demonstrated that this system can be used to implement a declarative bias towards
    1. the exclusion of certain syntactical constructs;
    2. the induction of expressions that use units of measurement;
    3. the induction of expressions that use matrix algebra;
    4. the induction of expressions that are numerically stable and correct.
- A case study using four real-world problems in the induction of dimensionally correct empirical equations on data using the two different methods is presented to illustrate the use and limitations of these methods in a framework of scientific discovery.

Resume (Abstract in Danish)

Genetisk programmering er i stand til at producere computer programmer, automatisk pa baggrund af eksempler pa programmernes virkning i en simulering. Da matematiske udtryk er en veldefineret delmangde af symbolske computer programmer og kan disse ogsa bestemmes under genetisk programmerings paradigmet. Empirisk bestemmelse af matematiske udtryk kaldes *symbolsk regression*.

I dette arbejde bliver genetisk programmering udvidet til, et varktoj der ikke bare "fitter data", men ogsa giver korrekte fysiske dimensioner. De vasentligste bidrag i dette arbejde opsummeres ved:

> *Symbolske udtryk, udledt ved hjalp af genetisk programmering kan gores tilgangelige for analyse og fortolkning, ved at lade dimensionsbetragtninger stotte eller begranse sogerummet.*

Dette er opnaet ved at

- Et standard genetisk programmerings-varktoj er blevet modificeret til at producerer udtryk som hovedsagligt er dimensionelt konsistente. Dette modificerede system er anvendt til at malrette genetisk sogning mod dimensionelt korrekte udtryk via sakaldt "preferential bias".
- Et nyt genetisk programmeringsvarktoj er blevet introduceret, som kan producere udtryk baseret pa kontekst-folsomme bibetingelser. Det er blevet demonstreret at dette system kan implementere malrettet sogning som via sakaldt "declarative bias" giver mulighed for at
  1. udelukke visse syntaktiske udtryk,
  2. producere udtryk baseret pa fysiske dimensioner,
  3. producere udtryk der involverer matrix algebra,
  4. producere udtryk som er numeriske stabile og korrekte,
- Der er endvidere udfort et empirisk studie der er baseret pa fire praktiske problemer og de to metoder, som involverer udtryk med korrekte fysiske dimensioner og derved illustrerer muligheder og begransninger indenfor automatisk data-analyse.

Preface

This thesis has been submitted in partial fulfilment for the degree of Doctor of Philosophy. The work documented in this thesis has been carried out both at DHI — Water & Environment and the Department for Mathematical Modelling, Section for Digital Signal Processing at the Technical University of Denmark. The work was supervised by Professor Lars Kai Hansen of the DTU and Dr. Vladan Babovic of DHI — Water & Environment.

During the Ph.D. study a number of conference papers and journal papers have been written.

Accepted Journal Papers and Book Chapters

- **Maarten Keijzer** and Vladan Babovic. Declarative and preferential bias in gp-based scientific discovery. *Genetic Programming and Evolvable Machines*, to appear 2002.
- Vladan Babovic and **Maarten Keijzer**. On the introduction of declarative bias in knowledge discovery computer systems. In P. Goodwin, editor. *New paradigms in river and estuarine management*. Kluwer, 2001.
- Vladan Babovic and **Maarten Keijzer**. Genetic programming as a model induction engine. *Journal of Hydroinformatics*, 2(1):35-61, 2000.
- Vladan Babovic and **Maarten Keijzer**. Forecasting of river discharges in the presence of chaos and noise. In J. Marsalek, editor, *Coping with Floods: Lessons Learned from Recent Experiences*, Kluwer, 1999.
- Vladan Babovic, Jean Philip Drecourt, **Maarten Keijzer** and Peter Friis Hansen. Modelling of water supply assets: a data mining approach. *Urban Water*, to appear 2002.

Conference Papers

- **Maarten Keijzer**, Vladan Babovic, Conor Ryan, Michael O'Neill, and Mike Cattolico. Adaptive logic programming. In Lee Spector et.al., eds, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, 2001.
- **Maarten Keijzer**, Conor Ryan, Michael O'Neill, Mike Cattolico, and Vladan Babovic. Ripple crossover in genetic programming. In Julian Miller et.al., *Genetic Programming, Proceedings of EuroGP*,2001

- **Maarten Keijzer** and Vladan Babovic. Genetic programming within a framework of computer-aided discovery of scientific knowledge. In Darell Whitley, et.al., *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, 2000.
- **Maarten Keijzer** and Vladan Babovic. Genetic programming, ensemble methods and the bias/variance tradeoff — introductory investigations. In Riccardo Poli et.al., *Genetic Programming, Proceedings of EuroGP'2000*, 2000.
- **Maarten Keijzer** and Vladan Babovic. Dimensionally aware genetic programming. In Wolfgang Banzhaf et al., *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, 1999.
- **Maarten Keijzer**, J.J. Merelo, G. Romero, M. Schoenauer. Evolving Objects: a general purpose evolutionary computation library In Pierre Collet, *EA-01, Evolution Artificielle, 5th International Conference on Evolutionary Algorithms*, 2001.
- **Maarten Keijzer** and Vladan Babovic. Error correction of a deterministic model in Venice lagoon by local linear models. In *Modelli complessi e metodi computatzionali intensivi per la stima e la previsione*, 1999.
- Michael O'Neill, Conor Ryan, **Maarten Keijzer** and Mike Cattolico. Crossover in Grammatical Evolution: The Search Continues. In Julian Miller et.al., *Genetic Programming, Proceedings of EuroGP*,2001.
- Kim Jørgensen, Berry Elfering, **Maarten Keijzer**, and Vladan Babovic. Analysis of long term morphological changes: A data mining approach. In *Proceedings of the International Conference on Coastal Engineering*, Australia, 2000.
- Vladan Babovic, **Maarten Keijzer**, and Magnus Stefansson. Optimal embedding using evolutionary algorithms. In *Proceedings of the Fourth International Conference on Hydroinformatics*, Iowa City, USA, 2000.
- Vladan Babovic, **Maarten Keijzer**, and Marek Bundzel. From global to local modelling: A case study in error correction of deterministic models. In *Proceedings of the Fourth International Conference on Hydroinformatics*, Iowa City, USA, 2000.
- Vladan Babovic, **Maarten Keijzer**, David R. Aquilera, and Joe Harrington. An evolutionary approach to knowledge induction: Genetic programming in hydraulic engineering. In *Proceedings of the World Water & Environmental Resources Congress*, 2001.
- Vladan Babovic and **Maarten Keijzer**. A Gaussian process model applied to the prediction of water levels in Venice lagoon. In *Proceedings of the XXIX Congress of the International Association for Hydraulic Research*,2001.
- Vladan Babovic and **Maarten Keijzer**. An evolutionary algorithm approach to the induction of differential equations. In *Proceedings of the Fourth International Conference on Hydroinformatics*, 2000.

- Vladan Babovic and **Maarten Keijzer**. Computer supported knowledge discovery — A case study in flow resistance induced by vegetation. In *Proceedings of the XXVIII Congress of the International Association for Hydraulic Research*, 1999.

- Vladan Babovic and **Maarten Keijzer**. Data to knowledge — the new scientific paradigm. In D. Savic and G. Walters, editors, *Water Industry Systems*, 1999.

Submitted Journal Papers

- **Maarten Keijzer** and Vladan Babovic. Knowledge fusion in data driven modeling. *Machine Learning*.

- Vladan Babovic and **Maarten Keijzer**. Rainfall runoff modelling based on genetic programming. *Nordic Hydrology*.

Acknowledgements

First and foremost I would like to thank Vladan, not only for convincing me to try to obtain a Ph.D. in Denmark by joining him in his Talent project, but also for his insistent enthusiasm and his many valuable contributions to this work. Although as a Ph.D. thesis, this work is necessarily authored by me alone, most of the views that are expressed in this work have been jointly developed.

Lars Kai and his group at the DTU have been very helpful. Although right from the start I've taken an almost diametrically opposite path from the group's research by concentrating on the use of symbolic expressions rather than 'sound' numerical procedure, these 'numerics' did have a profound influence on the work. I have learned a lot from the group.

Conor Ryan, Michael O'Neill and Mike Cattolico deserve mentioning for the many intense and considerably less intense discussions we held during the various conferences and workshops in the past three years. One of the tangible results of these discussions is the ALP system which is based on Michael and Conor's 'Grammatical Evolution' system. I hope we can continue to cooperate in the future.

The Ph.D. and this thesis were funded by the Danish Research Council under Talent Project 9800463 entitled "Data to Knowledge – D2K". This funding is greatly appreciated.

Deventer, May 1, 2002

Maarten Keijzer

Contents

Chapter 1

Introduction

> Physical concepts are free creations of the human mind, and are not, however it may seem, uniquely determined by the external world.
> -*Albert Einstein and Leopold Infeld*, 1938

The formation of modern science occurred approximately in the period between the late 15th and the late 18th century. The new foundations were based on the utilization of a *physical experiment* and the application of a *mathematical apparatus* in order to describe these experiments. The works of Brahe, Kepler, Newton, Leibniz, Euler and Lagrange personify this approach. Prior to these developments, scientific work primarily consisted of collecting the observables, or recording the 'readings of the book of nature itself'.

This scientific approach is traditionally characterized by two stages: a first one in which a set of observations of the physical system are collected, and a second one in which an inductive assertion about the behaviour of the system — a hypothesis — is generated. Observations present *specific knowledge*, whereas hypotheses represents a *generalization* of these data which *implies* or *describes* observations. One may argue that through this process of hypothesis generation, one fundamentally economizes thought, as more compact ways of describing observations are proposed.

Although this view of the dispassionate scientist observing facts and producing equations is popular, it is not all there is to say about the process of scientific discovery. In the years that lead to Kepler's famous laws of planetary motion, he introduced and abandoned various informal models of the solar-system. These models initially took the form of a collection of embedded spheres (Holland et al., 1986)(pp. 323-325). It was only when he abandoned the idea of planets moving in circular orbits around the sun and replaced it with ellipses that he was able to postulate his laws. Kepler is not unique in this; the process of the formulation of scientific law or theory usually takes place in the context of a mental model of the phenomenon under study: using the right concept to explain the equation provides additional justification for these equations. Finding a proper conceptualization of the problem is as much a feat of scientific discovery as the formulation of a mathematical description or explanation of a phenomenon.

Today, in the beginning of the 21st century, we are experiencing yet another change in the scientific process as just outlined. This latest scientific approach is one in which information technology is employed to assist the human analyst in the process of hypothesis generation. This computer-assisted analysis of large, multi-dimensional data sets is sometimes referred to as a process of *Data Mining and Knowledge Discovery*. The discipline aims at providing tools to facilitate the conversion of data into a number of forms that convey a better understanding of the process that generated or produced these data. These new models combined with the already available understanding of the physical processes — the theory — can result in an improved understanding and novel formulation of physical laws and an improved predictive capability.

One particular mode of data mining is that of *model induction*. Inferring models from data is an activity of deducing a closed-form explanation based solely on observations. These observations, however, always represent (and in principle only represent) a limited source of information. The question emerges how this, a limited flow of information from a physical system to the observer, can result in the formation of a model that is complete in the sense that it can account for the *entire* range of phenomena encountered within the physical system — and to even describe the data that are outside the range of previously encountered observations. The confidence in model performance can not be based on data alone, but might be achieved by grounding models in the domain so that appropriate semantic content is obtainable. These models can then be used to reinforce, inspire or abandon the scientists' view of the problem.

The overall goal of the approach is then to subtly change the process of scientific discovery. Rather than having the scientist 'read' the data, invent a conceptualization (an informal model) of the problem using this data in order to finally provide a formal expression that describes the conceptualization and thus the phenomenon, the scientific discovery process envisioned here removes the need for the scientist to work with only the raw data to inspire a conceptualization of the process under study. What is attempted here is to automatically generate expressions that use high level physical concepts — units of measurement — to provide an approximate, but interpretable view of the data. It is thought that such approximate expressions, once analyzed, can help the scientist in understanding the data better. Finally, once the conceptualization is trustworthy, a formal expression can be proposed that is either build out of (parts of) the automatically generated expressions or is build directly out of the informal model itself.

The prototypical cycle of observation, imagination, formalization and testing that is associated with scientific discovery is then extended to include an automated modelling step between observation and imagination. By providing tentative formalizations based on data and high-level physical concepts, the scientist is freed from examining measurements only: examining well-fitted, possibly meaningful expressions is thought to be an task that can inspire novel conceptualizations of the processes under study. As such an automated method is biased only to the available data and to these high-level concepts, it would be free to propose approximate solutions to the problem that are radically different from contemporary thought. Understanding how such a different approximation fits in the scientific framework might lead to an enhanced or maybe even different approach to describing the physical system.

In order to create such a system, we need model induction algorithms that produce models amenable to interpretation next to the ability to fit the data. The interpretation of these models should then provide the additional justification that is needed to use the model with more than just statistical confidence. Clearly, every model has its own syntax. The question is whether such syntax can capture the semantics of the system it attempts to model. Certain classes of model syntax may be inappropriate as a representation of a physical system. One may choose a model whose representation is complete, in the sense that a sufficiently large model can capture the data's properties to a degree of error that decreases with an increase in model size. Thus, one may decide to expand Taylor or Fourier series to a a degree that will decrease the error to a certain, arbitrarily given degree. However, completeness of the representation is not the issue. The issue is in providing an adequate representation amenable to interpretation.

The present work is an attempt to make the models produced by the technique of genetic programming more suitable to be used within a scientific discovery framework. It critically uses *units of measurement* as the apparatus to ground the models in the physical domain. Units of measurement have been chosen as they embody a formal system for manipulating physical concepts such as lengths, velocities, acceleration and forces. Manipulating numbers using arithmetic is then accompanied with manipulating units of measurement. The units of measurement are proposed to form a suitable set of high-level concepts to be used in scientific discovery. The resulting symbolic expressions produced by this system are fully dimensioned: the scientist working with the system can analyze and interpret the equations by translating the formal definition of the units back to the respective physical concepts.

In the scientific discovery process that is proposed here, the scientist still plays a pivotal role. Although the process of creating equations from data is done using automated means, the important process of interpretation, analysis and embedding the proposed hypotheses in an existing or new theory remains firmly in the hands of the scientist. The equations that are discovered form both an empirical formulation of the relationships in the data and a tentative proposal of the physical concepts that are manipulated by the formulation. It is thought that the scientist using these tentative proposals can more efficiently set up, test and refute models for the problem under study.

This work will focus on the definition and comparison of methods that incorporate units of measurement in the search. The thesis forwarded in this work can then be summarized as:

> *The symbolic expressions produced by genetic programming can be made suitable for analysis and interpretation by using units of measurement to guide or restrict the search.*

To examine this, several paths will be traversed. Two genetic programming systems will be developed: one that *guides* the search to (more-or-less) dimensionally correct expressions, the other that *restricts* the search to only those expressions that are dimensionally correct. The work is then organized as:

- **Chapter 2** gives a brief overview of genetic and evolutionary computation, in particular the technique of genetic programming. The concept of multi-objective optimization in the context of evolutionary search will be described.

Multi-objective optimization, in particular using the concepts of Pareto optimality, enables searching in a space where the trade-offs between the objectives are not known beforehand.

- **Chapter 3** introduces the standard form of inducing expressions using genetic programming. This is called *symbolic regression*. Here it will be argued that although genetic programming is capable of inducing mathematical (symbolic) expressions, interpretability is not a natural by-product of these equations.

- **Chapter 4** will lay some groundwork for the rest of the thesis. It will focus on what it means to induce an *empirical equation*, and will briefly describe two ways of incorporating knowledge about the units of measurement in the search.

- **Chapter 5** describes the technique called *Dimensionally Aware* genetic programming. Rather than abiding to the units of measurement at all cost, it implements a preference toward dimensionally correct equations. It balances the ability to fit the data with the ability to use the units in a correct way.

- **Chapter 6** introduces the system used for implementing *Dimensionally Correct* genetic programming. Due to the context-sensitivity of the constraints present in this system, the expressiveness of a Logic Programming language is used. The search strategy in this Logic Programming system is a genetic algorithm, its task is to optimize paths through the search tree defined by a Logic Program.

- **Chapter 7** applies this novel system in a series of experiments involving the exclusion of syntactical constructs, the use of interval arithmetic, the use of units of measurement and finally the induction of correct sentences in matrix algebra. These four experiments are used to highlight the versatility of the approach.

- Finally, **Chapter 8** compares the two approaches. On four real-world problems, the dimensionally aware approach will be contrasted with the dimensionally correct approach to model induction. The experiments will be conducted on the basis of quantative measures — ability to provide well-fitted correct equations — and on the quality, the interpretability of the expressions.

- **Chapter 9** concludes the thesis.

Chapter 2

Genetic Programming

2.1 Evolution at work: Genetic & Evolutionary Computation

In 'The origin of species', Charles Darwin (Darwin, 1859) introduced the principle of natural selection as a unifying view for the origin and further evolution of organisms in nature. Using similar principles the field of *Evolutionary Computation* tackles difficult problems by evolving approximate solutions inside a computer. Starting with a primordial diversity of random solutions, repeated selection and variation are applied to improve the quality of the solutions. The basic criteria for evolution to occur — be it *in vitro* as in biology or *in silico*, with computers — have been summarized by the biologist Maynard-Smith (Maynard-Smith, 1975) as:

- **Criterion of Fecundity** Variants leave a different number of offspring; specific variations have an effect on behaviour and behaviour has an effect on reproductive success;
- **Criterion of Heredity** Offspring are similar to their parents: the copying process maintains a high fidelity;
- **Criterion of Variability** Offspring are not exactly the same as their parent: the copying process is not perfect.

These criteria are necessary ingredients for evolution to occur and are used to solve problems by employing an *Evolutionary Algorithm*. Such an evolutionary algorithm operates on populations of candidate solutions, each solution is graded according to its performance and constitutes a basis to improve upon for future generations.

In its most basic form, an evolutionary algorithm works on a population of solutions, P, which is subject to the iteration:

$$P_{t+1} = v(s(P_t)) \tag{2.1}$$

where the functions s and v are called *selection* and *variation* operators respectively. Starting with a randomly generated population P_0, this algorithm is applied for many iterations, called *generations*. The selection function s implements the criterion of fecundity: it makes sure that solutions that have above average performance receive more copies in the next generation. The selection function is thus used to enforce the goal of the optimization process; getting the best solution possible. These selected solutions (copies) are subsequently processed by the variation operator v. The variation operator usually applies random, undirected changes, and is supposed to balance the heredity and variability criteria. Too much variation and the evolution will degrade to a random search, too little variation and the population of tentative solutions will evolve to a population of clones only.

The selection operator uses the performance of the solutions to give above average performing instances more copies in the next generation. An objective function needs to be defined that can calculate this performance. In the most simple case, this function returns a scalar value that calculates some *objective value*. Thus given some function that calculates the performance of an individual and a population size n, the selection function assigns copies for the next generation. It can do this through one of many ways.

- **Proportional Selection** Create n copies of individuals proportional to the performance of solutions, the variation operators will then be applied to this new population;
- **Truncation Selection** For a number $m < n$, select the best m individuals from the population, add $n - m$ randomly selected copies from these m best individuals to obtain a new population[1], the variation operators will be applied to these $n - m$ copies;
- **Tournament Selection** For some k, the tournament size, repeatedly select k individuals at random, and put the best of those k in the next generation after applying the variation operators until n copies are assigned.

The definition of this basic evolutionary algorithm is representation-free. It does not mention what form of solutions should be considered, and in effect, many representations are used in the field of evolutionary computation. The best known evolutionary algorithm is the *genetic algorithm* (Holland, 1980; Goldberg, 1989), that typically uses fixed length bitstrings as the representation of choice. Other, older, work involved finite state automaton (Fogel et al., 1966) and real valued vectors (Rechenberg, 1965; Schwefel, 1995). This work has evolved into the separate but related fields of *evolutionary programming* and *evolution strategies*. Currently, many problem-dependent representations are in use for practical applications.

Once the representation is chosen, variation operators need to be defined. The simplest of such operators is the mutation operator that makes a small randomized change to the representation: in the case of bitstring flipping one or several bits is a common operation; when using floating point values a small Gaussian change can

[1] This selection mechanism is usually defined in a slightly different way, where the variation operator enlarges the population and the selection operator reduces it; but the definition given here is equivalent with this. It is presented in this way to keep it in line with the abstract evolutionary algorithm in Equation 2.1.

be applied. Often also a recombination operator is defined: this is called crossover. The crossover operator recombines the information in two solutions to create one or two new solutions. It does this in a randomized fashion. Its purpose is to explore new combinations of parts of the solution, in the hope that this leads to a new level of performance.

Not any combination of representation, selection and variation makes sense however. Variation of solutions need to be correlated in some way with the performance of solutions. In its strongest form this means that a small change in the representation of the solution should be accompanied with a small change in the performance of the solution. Another correlation that is often hypothesized is the exploitation of building blocks in the problem. These are partial representations (schemata) whose worth in complete solutions are as independent as possible from the context they are used in. By recombining building blocks, new high-performing solutions might be obtained. The interplay between the performance of solutions, the representation of solutions and the variation of representations is a major research area in the field.

Below, *genetic programming* is described. With this method, the representations that are being evolved are computer programs that try to solve a specific problem. It is an attempt to perform *automatic programming* in the sense outlined by Samuel (Samuel, 1959), where computers are programmed by *telling them what to do, not how to do it*.

2.2 Standard Genetic Programming

Koza's monograph "Genetic Programming, on the programming of computers by natural selection" (Koza, 1992) marks the beginning of the field of genetic programming. It contains a wealth of examples where a basic genetic programming system was used to solve problems in various fields of artificial intelligence. The crucial insight in the book was the observation that many, if not most problems in artificial intelligence can be stated as:

> Given a problem X, find a computer program that solves X.

Together with a method to automatically find computer programs — genetic programming — this guideline was powerful enough to solve a wealth of problems taken from the artificial intelligence literature. Thus instead of using specialistic representations like neural networks, decision trees, horn clauses or frames, the genetic programming method tries to solve problems by relying on a single representation framework: that of computer programs[2].

Genetic programming as envisioned by Koza does not process computer programs in the same way as human programmers do. There's no file of statements written in ASCII, no pesky syntax with various special symbols like semi-columns that can be misplaced to produce a syntactically meaningless result, no myriad of data types that cannot be mixed. The standard single-typed genetic programming system operates using an abstraction of computer programs — an already parsed expression, typically

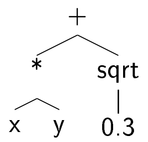
[2]The primitive functions and variables used inside the computer programs vary however from application to application

represented in a parse tree. The use of a parse tree representation in a genetic algorithm was pioneered by Cramer (Cramer, 1985).

There's nothing special about parse trees other than that it circumvents issues of a purely syntactical nature and suggest a few natural variation operators. If one were to try to optimize program snippets in C for instance, one could try to proceed by using a genetic algorithm using the ASCII character set. Say that by chance a function such as

```
double solution(double x, double y)
{
  return x * y + sqrt(0.3);
}
```

would evolve, and arbitrary variation operators are allowed, it is easy to see that the possibility of introducing syntactical errors is immense. Changing a single character to an arbitrary other character would in most cases result in a syntactic error. Genetic programming circumvents this problem by only considering the relevant part of the syntax in a computer friendly format — the parse tree. For all practical purposes the C-style function above can be described by the parse tree



where no information about the computation that is performed is lost, but a few syntactic issues are cleared up. The number of arguments for each function can be deduced from the number of children of a node and also issues of operator precedence are resolved. The parse tree thus represents an unambiguous way of computing the function. It is this property that is also employed by compilers. These generally use parse trees as an intermediate representation before generating machine code.

The parse tree also provides inspiration to the issue of variation. As a parse tree decomposes a computation into a hierarchy of subcomputations, varying these subcomputations at the various levels in the tree is a natural way of obtaining new programs. Section 2.2.3 will go into more detail on how to vary solutions in standard genetic programming.

2.2.1 The Primitives

A parse tree is composed of function symbols — the inner nodes — and terminal symbols — the leafs of the tree. Defining these function and terminal symbols is an inherently problem dependent issue. If the problem is one of regression, it would be natural to include the independent variables in the terminal set and let a variety of mathematical functions form the function set. If the problem is one of simulated robotics, various sensor information could be used as terminals or branching instruction. The output of the program or side-effecting functions could then be used as effectuators.

Finding a symbolic expression based on some data is a central problem in this work. Typical function and terminal sets that are used here involve simple mathematical functions, operating ultimately on the independent variables, the terminals. The most commonly used function set in this work is:

$$F = \{\text{plus/2, times/2, minus/2, divide/2, sqrt/1}\}$$

where the number behind the function name indicates the arity of the function. The terminal set consists of the independent variables and a special terminal: a real valued constant. In Koza's original setup such constants were initialized at random, but were not changed during the run. Here we will however use a special mutation operator for these constants. A terminal set involving independent variables x, y and randomly initialized constants will be denoted as:

$$T = \{x, y, \mathcal{R}\}$$

There are only vague guidelines for choosing a particular function and terminal set. In general one tries to find a suitable high-level set of functions accompanied by a set of terminals that are most descriptive for the problem at hand. There is inherently some arbitrariness in this selection. It is however quite accepted that very low-level functions are not very useful: although logically complete, finding a real-valued function using only the `nand` operator is considered to be a waste of time due to the enormous size of the parse trees one needs to even implement simple functions. The function and terminal set is usually chosen in such a way that different, powerful solutions can be implemented by relatively small parse trees. The function set defined above can already describe all rational functions of arbitrary degree, and the `sqrt` function allows fractional powers as well.

2.2.2 Initialization

Using the primitives, it is possible to generate well-formed parse trees. This can be done in several ways. One of the simplest is the `grow` method, where a primitive — be it a function or terminal — is selected uniformly at random, and as long as there are unresolved subtrees, the process is repeated. When a prespecified depth or size limit is reached only terminals are chosen. An example of this process is depicted in figure 2.1

Another method developed by Koza is the `full` method. Here function nodes are always chosen until the prespecified-specified depth limit is reached. At that point only terminal nodes are chosen. The tree in Figure 2.1 could equally well have been created by the `full` method if the depth limit was set at the low value of 3. Using the `grow` and `full` method each for 50% of the population is known as the `ramped-half-and-half` initialization method.

As the primitives are chosen uniformly from the available primitives, the expected size of the trees varies considerably with the sizes of the function and terminal set. In Section 6.2.1 the gambler's ruin model will be used to analyze the `grow` method. An overview of alternative tree initialization routines and an empirical comparison between those can be found in (Luke and Panait, 2001).

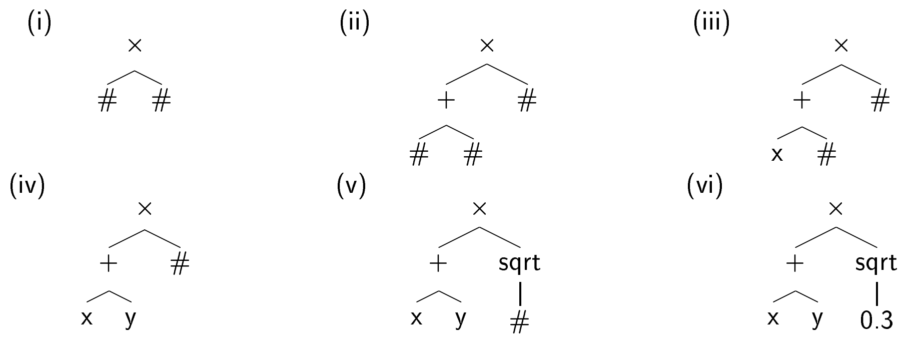


Figure 2.1: Creating a tree. Empty spots (denoted by #) are recursively filled in until the tree is completed.

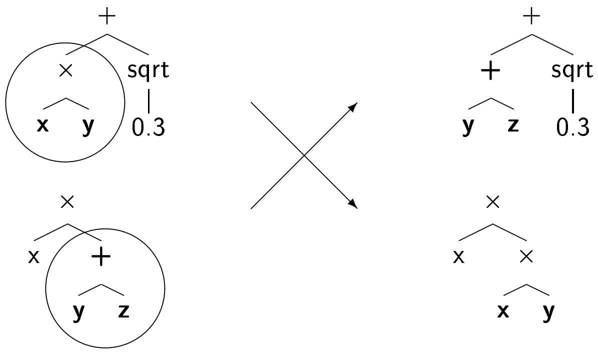


Figure 2.2: Example of a subtree crossover where the parents on the left produce the children on the right by exchanging the two circled subtrees.

2.2.3 Variation

Given a parse tree where the internal nodes represent the functions and the leafs the terminals, many variation operators can be defined. Two basic operators will be described here: subtree mutation and subtree crossover. These operators are very simple: subtree mutation replaces a randomly chosen subtree in a tree with a randomly generated subtree, while subtree crossover swaps two randomly chosen subtrees in the parents to create a new tree. Figure 2.2 gives an example of subtree crossover.

Choosing subtrees randomly from all available subtrees implies some bias towards selecting smaller subtrees. This comes naturally from the parse tree representation where subtrees can be composed of subtrees themselves. When randomly choosing subtrees from a tree composed of binary functions, slightly more than 50% of the subtrees are terminals; randomly choosing nodes in the tree will then in more than half of the cases result in choosing a terminal. As terminals are the smallest parts of trees, this results on average in an exchange of a minimal amount of information. To counter this, Koza (Koza, 1992) took the pragmatic approach of selecting an internal node (a function) 90% of the time and a terminal 10% of the time. Many

x	y	t	$(x+y) \times \sqrt{0.3}$	error
0.0	1.0	0.55	0.54772	0.0022774
0.2	0.6	0.45	0.48990	0.039900
...	...	...	...	...
0.9	0.1	0.6	0.54772	0.052277
				0.8321

Table 2.1: Example of evaluating a function induced by genetic programming on the available data, where t is the target variable.

other approaches have been defined however to implement some other distribution on the selection of subtrees (Langdon, 1999; Harries and Smith, 1997).

One special mutation operator is used here that selects a constant — if present — in the tree and changes its value a little — usually by adding a normally or Cauchy distributed number.

2.2.4 Measuring Performance and Wrapping

The main feedback to an evolutionary algorithm is the performance measure. The performance measure is used by the selection function to determine which programs receive more variants in the next generation. Often the performance measure is a single scalar value, but more than a single performance criterion can be used as well. One would then enter the area of evolutionary multi-objective optimization. A good performance measure for any evolutionary search methods gives an as fine-grained differentiation between competing solutions as possible, focusing on the eventual use of the program and avoiding giving false information. An example of evaluating a mathematical function on some data is given in Table 2.1.

In this work, two error measures are mainly used for reporting results. One is the root mean squared error (RMS error, or RMSE), defined as

$$\text{RMS}(y,t) = \sqrt{\frac{1}{(N-1)} \sum_{i}^{N} (y_i - t_i)^2}$$

using the symbols y and t as the model outputs and the target outputs on a data set of size N respectively. The RMS error can be used to obtain a performance measure stated in the same units as the target variable. Another measure that is used here is the *normalized* RMS error (NRMS), which is defined as:

$$\text{NRMS}(y,t) = \frac{\text{RMS}(y,t)}{\text{std}(t)}$$

where std is the standard deviation measure. The NRMS error measure scales the error in such a way that a prediction of the average in the target data has an NRMS error of 1.0.

In genetic programming, often the output of the programs is *wrapped*, that is, the output is changed in such a way that it can be used by the performance calculation. Here the use of wrapping and its influence on the performance measure will be illustrated in two problem domains: classification and regression. Two wrappers will be discussed that are capable of enlarging the solution space for genetic programming.

Wrappers for Classification Consider a problem in binary classification. Here the object of search is a program that classifies input cases as belonging to a certain class or not. One possibility of tackling such a problem is to only consider functions that return boolean values: if the program returns true for a certain input case it will be interpreted as a positive, otherwise a negative. This is the general approach when the inputs are boolean variables, but for input data that is real-valued a different approach is usually adopted. In that case, real valued functions are used and the output of the program is interpreted as a score: a real valued ordinal variable. Usually a fixed cutoff value is set: scores falling above the cutoff will be classified as positive, and negatives otherwise. The use of an arbitrary cutoff value to be able to interpret a real valued outcome as a binary classification is a first example of a wrapper function.

Wrappers can however vary in their ability to make optimization easy or difficult. In the case of the binary classification problem an arbitrary cutoff value can make optimization needlessly difficult. Evolving a classifier against a fixed cutoff value makes this value very important for the classifiers. This might hinder the search in unforeseen ways as it biases the search towards classifiers that discriminate optimally in the context of this arbitrary value. A better approach would be to calculate the optimal cutoff value for each classifier independently. Here the wrapper function would examine the full range of scores produced by the classifier and will find that cutoff value that produces the optimal discrimination between the positive and negative cases. This can be accomplished with a single pass through a score array and is usually feasible computationally.

As the scores produced by the classifiers are ordinal, the actual values are irrelevant, it is the relative order that matters. By not imposing an arbitrary cutoff point, but using the implied optimal cutoff after the evaluation, the classifiers are less constrained in the score range: in particular adding a constant value to the classifier will not change its performance. This increases the number of solutions to the problem and can thus help the evolutionary search in finding good classifiers.

Wrappers for Regression For regression a similar wrapper can be defined. Usually in regression problems, the object of search is an expression that minimizes some least squares error criterion. A straightforward approach would then be to use this error as the performance measure. It will thus constrain the search to expressions that are as close to the target values as possible. However, this will also constrain the search to expressions that have the proper slope and intercept that is present in the data: for example, an expression that produces outputs that have the same shape as the desired output but is structurally wrong with a certain constant value, will have the squared value of this constant added to its error for every case. However, using standard (fast) methods of linear regression on the outputs of the expression can identify such structurally different slopes and intercepts and scale the output of the expression to the appropriate range. This again makes the programs invariant against these transformations and increases the solution space. With a larger space of solutions, the search is more likely to find a good expression. Using linear regression, for any well-defined expression $f(\mathbf{x})$, it is possible to calculate a and b such that the squared error between the target values and the wrapped expression $a + b \cdot f(\mathbf{x})$ is minimal. This calculation is linear in the number of cases that are considered.

Even though the slope and intercept can be calculated linearly with the number of cases, it is possible to circumvent the use of such a wrapper entirely during the run. If one were to employ Pearson's squared correlation coefficient[3] as the performance measure, no slopes and intercepts need to be calculated during the run: the correlation measure already calculates a squared error equivalent, regardless of the slope and intercept. At the end of the run, the best expression can then be wrapped and used for making predictions.

An interesting side-effect of using a correlation coefficient is that it is undefined when the predictions are constant. Interestingly enough, in running genetic programming using a squared error measure some runs converge prematurely on an expression consisting of a constant only, which usually represents the average value of the target data. Using a correlation coefficient as the performance measure will identify such an expression and by giving it the worst possible performance value, such expressions are effectively culled.

These are two examples of using some knowledge about the performance measure to enlarge the solution space for genetic programming. Such tricks are not necessary for more standard regression and classification methods, as these usually solve the problems of arbitrary cutoffs, slopes and intercepts by making these explicit in the model architecture. For example, in artificial neural networks finding the proper intercept is accomplished by adding so-called bias nodes to the neural network: the gradient based search techniques will set the weights from these bias nodes to appropriate values. This section showed that for genetic programming a similar effect can be achieved at the output level by employing smart wrappers.

2.2.5 Auxiliary parameters and variables

A few auxiliary parameters and variables need to be set before running a genetic programming system. One of the most important of these is the population size. However, not much is known on the optimal or even minimal population size in genetic programming. Other parameters involve the rate of applying the variation operators, the exact way of performing selection and the maximum size or depth the trees are allowed to grow to.

2.3 Multi-Objective Optimization

Often, the quality of a solution can not be easily captured in a single number. For instance, when designing a power plant, both the cost of building a plant and

[3]This is the correlation coefficient found in statistical packages, defined as: $\left(\frac{\mathsf{cov}(x,y)}{\mathsf{std}(x)\mathsf{std}(y)}\right)^2$

the risk of the plant blowing up and taking countless lives needs to be minimized. These objectives are usually contradictory and very hard to balance at the outset of designing a plant. Building a plant that has a minimal risk involves implementing countless security measures, each costing money. Avoiding to implement any security measures at all will be very cheap, though the people living near the plant might not be happy with such an insecure plant in their vicinity. Without knowing the full distribution of designs that balance cost and risk, it is difficult if not impossible to judge which balance of objectives is optimal. This is the area multi-objective optimization applies to.

The simplest form of multi-objective optimization involves a weighting scheme, where the relative importance of the objectives are fixed at the outset. In the example, these weights are multiplied with the cost value and the risk value, and subsequently added together to obtain a single scalar value that judges a design. This process involves *a priori* assumptions on the relative worth of the objectives, and in the plant example would require an objective judgement about the cost of taking a human life. There will quite likely be some disagreement about this monetary value between the owners of the plant and the inhabitants of the neighbourhood. Without some knowledge about the trade-offs involved in building the plant i.e., thus without a set of designs that balance cost and risk, such a discussion would be made using a priori arguments only, quite likely not leading to any level of agreement between the parties involved.

If there is no agreement how to translate one objective into another objective, how does one measure the quality of a solution? This is where the concept of *Pareto dominance* can help. Instead of giving an absolute (scalar) judgement for a solution, a partial order is defined based on *dominance*. A solution is said to *dominate* another solution when it is better on one objective, and not worse on the other objectives. Thus a solution a dominates a solution b if and only if $\exists i : o_i(a) < o_i(b)$ and $\forall_{j \neq i} o_j(a) \leq o_j(b)$. This assumes without loss of generality that the objective functions $o_1, \ldots, o_m$ need to be minimized. A solution is said to be non-dominated if no solution can be found that dominates it.

The definition of the dominance relation gives rise to the definition of the *Pareto optimal set*, also called the *set of non-dominated solutions*. This set contains all solutions that balance the objectives in a unique and optimal way. An example of such a set is depicted in Figure 2.3. Since there is no single scalar judgement, this set usually contains a wealth of solutions. As there is no notion present of one objective being more important than another, the aim of multi-objective optimization is to induce this entire set. Picking a single solution from this set is then an *a posteriori* judgement, which can be done in terms of concrete solutions with concrete trade-offs, rather than in terms of possible weightings of objectives.

The question for multi-objective optimization is now how to find this Pareto optimal set. One approach would be again a weighted approach, where the weights are varied between runs and for each unique weighting scheme a solution is obtained. This would require many runs to estimate the Pareto set and the granularity of the weight changes needs to be estimated or assumed.

An evolutionary multi-objective approach avoids the granularity and multiple runs issues altogether by using the wealth of solutions present in the evolving population to find a balance *during* a single run of the algorithm. It thus tries to find and store the Pareto optimal set in the population. Many concrete algorithms to achieve

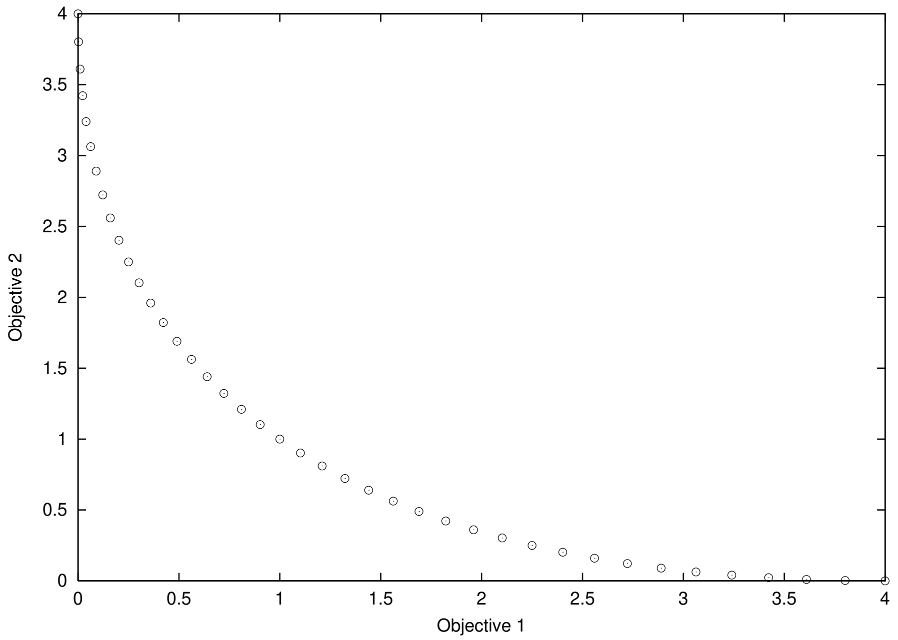


Figure 2.3: An estimate of the Pareto optimal set using two objectives. The objectives are: $o_1 = x^2$ and $o_2 = (x-2)^2$

this have been proposed, we will focus here on the non-dominated sorting genetic algorithm II (NSGA-II) (Deb et al., 2000).

A non-dominated sorting GA assigns ranks to solutions by first finding the set of non-dominated solutions in the current population. These are removed from the population and assigned rank 1. As these solutions are removed, a new so-called *front of non-dominated* solutions is now present in the remainder of the original population. This second front is extracted and assigned rank 2. This procedure is repeated until no more solutions are present in the population: each solution gets a rank according to the pass that is used to extract it. It was shown that this procedure can be implemented in $\mathcal{O}(n^2)$, where n is the population size (Deb et al., 2000).

All solutions in the population have now been assigned an integer rank. Without any further processing, this algorithm will not find a good estimate of the Pareto optimal set. A population consisting purely of clones is for instance a point of convergence. Repeated selection acting upon a finite population will ensure convergence to this point due to stochastic sampling effects. What is needed is a mechanism to spread out the population over the entire front.

All solutions taken from the same front have the same integer rank. The NSGA-II algorithm will break ties by calculating the uniqueness of a solution in the front, filling in the fractional part of the rank with this uniqueness value. The value is determined by calculating the distance in objective value space between a solution and its nearest neighbours. The uniqueness value is then calculated by sorting each front for each objective and calculating the distance between each solution and its two nearest neighbours. Solutions at the extremes get 'highest' distance.

Subsequently, for all objectives this distance value is summed, scaled to values between zero and one, and subtracted from the integer rank. In this way the integer value still denotes the rank of the individuals, while the fractional part is used as a tie-breaker with competing solutions of the same integer rank. Solutions with more unique trade-offs will have a better rank then solutions in more populated areas. The population can now be sorted on this rank and truncation selection can be used.

The sorting is performed for each objective and each front. The computational complexity of this procedure is at most $\mathcal{O}(n \log n)$. The overall complexity of this procedure thus remains at $\mathcal{O}(n^2)$.

The NSGA-II algorithm is very robust and makes it possible to perform an adequate search for a Pareto optimal set. It is used throughout this text (Chapters 5, 7 and 8) whenever a multi-objective problem is addressed.

2.4 Implementation Issues

Evaluating individual programs for their performance is in most non-trivial applications the most time-consuming task. Much effort has undergone into making this evaluation as fast as possible. Two main methods of representing parse trees in the C programming language are in use: a pointer tree implementation and a token string implementation. The pointer tree implementation has as its main advantage that coding manipulations on the trees is very natural and can be very fast; it has as a drawback however that memory management is non-trivial. If one ignores memory management, the time involved in allocating and de-allocating nodes can lead to sub-optimal performance. Because in this representation pointers to the children of a node needs to be kept, it also has a relatively high memory footprint. The token string representation on the other hand is very parsimonious because it only needs to keep an identifier to the node (a token) per element in the string. An arity function that returns the arity of a node given the identifier can be used to keep the string syntactically correct when applying the variation operators. String operations on modern computers are very fast and memory management is also less of an issue. However, it is quite a bit more cumbersome to keep a string syntactically correct, which makes this representation less suitable for rapid development.

For a review of pointer tree and token string implementations, the reader is referred to (Keith and Martin, 1994) that presents a comparative study of several implementations and the corresponding evaluation functions. The paper focuses on how to make traversing the tree as fast as possible as a typical evaluation function requires that the parse tree is traversed multiple times. Another interesting approach was pioneered by Handley (Handley, 1994), where subtree sharing was used to reduce evaluation time. It was experimentally shown that with using subtree sharing the amount of memory that needs to be used to store a population can be significantly less when compared to a string-based approach (Keijzer, 1996). This is not obvious as a (sub)tree based approach needs to store indices or pointers next to some function identification token, while a string based implementation only needs to store the token.

Vectorized Evaluation Here we will describe a method that in the case of a function and terminal set that do not have side-effects requires only a single pass through the tree, regardless of the amount of data points. This method is by no means new, in the numerical computation literature it is known as vectorized evaluation, and in effect this was used by Handley (Handley, 1994) to cache previously performed computations. It is reviewed here, as it can be used without regard to subtree sharing and caching and the bookkeeping necessary to implement these. It has to the best of the author's knowledge as such not been presented in the genetic programming literature before. It can be implemented in practically any tree-based implementation to speed up evaluation considerably.

Consider a classification or regression task where the function and terminal set are purely functional by nature: there are no side-effects when evaluating a function and there exists a large set of data points. The usual approach of evaluating a tree on a data point is to recursively go down the tree to evaluate a single case. Using recursion is however fairly slow compared to iteration. By vectorizing the evaluation, all cases will be evaluated for each node in the tree. The tree is then recursively traversed only once.

To illustrate the vectorized evaluation procedure, some C++ code is presented in Figure 2.4. It assumes that each subtree has a node identification number and a pointer to the children trees. It also assumes the existence of the auxiliary (global) functions `pop_container` and `push_container` that dispense and re-take preallocated containers from a (growing) stack and a function `get_variable_values` that returns a container with the values for a specific variable for all cases. The liberal use of the address operator can in C code be replaced by a pointer without any difference. It is used here to simplify the syntax.

This evaluation function will return a container containing the output for all data points. After the performance has been evaluated it can be re-used by using the `push_container` function. Copying the variable values is wasteful, especially considering that terminals are often the most numerous elements in the tree. This can be resolved by modifying the container storage functions to recognize the containers containing the variable values and subsequently refraining from using in-place calculations. This is not done here as this would make the implementation more involved then necessary. The vectorized evaluation presented here achieves its task: it can evaluate a tree on an entire dataset with a single recursive traversal through the tree. It does this at a cost of keeping a number of vectors proportional to the depth of the tree. Replacing recursion by iteration in this way is expected to speed up evaluation considerably on problems that use non side-effecting functions and a limited number of conditional branching instructions. As traversing the tree is done only once per evaluation, no special attention needs to be given to optimizing the tree traversal routine. Most notably, the `switch` statement in the routine is only executed once for every node in the tree regardless of the amount of data that is manipulated.

It is maybe interesting to note that, when used with for example the `bitset<size_t>` template class in the standard C++ library (Stroustrup, 1997)(pp. 492-496) as the container class, this procedure is equivalent with sub-machinecode genetic programming (Poli and Langdon, 1999). The `bitset` class implements optimized vectorized logical operations on bitstrings stored parsimoniously in integers, and there is then no need to manually implement evaluation and packing/unpacking procedures.

```
container& Tree::evaluate()
{
  switch(nodeId)
  {
    case plus :
    {
      container& c0 = child[0].evaluate();
      container& c1 = child[1].evaluate();

      // assuming a properly vectorized addition
      // function defined on the container class
      c0 += c1;

      // c1 is not needed anymore
      push_container(c1);

      return c0;
    }
    // other functions
    default : // assume this is a variable
    {
      const container& v = get_variable_values(nodeId);
      container& result = pop_container();
      result = v;   // copy
      return result;
    }
  }
}
```

Figure 2.4: C++ snippet for performing vectorized evaluation. It assumes a properly defined container class and a method of storing and retrieving a growing number of these containers. In this example, the container class needs to be able to perform vectorized evaluation, but this can also be done in the code itself.

2.5 Summary

This chapter presented a very short introduction in genetic programming. For a more thorough introduction into the subject of genetic programming, the reader is referred to Koza (Koza, 1992) and Banzhaf et al. (Banzhaf et al., 1998). The main focus in this chapter was in providing the bare essentials to understand the evolutionary computation approach in general and genetic programming in particular. The material describes a few concepts that will be used in subsequent chapters. A few tricks and tips have been described here that have been developed for practical applications employing genetic programming. These techniques involving wrapping and vectorized evaluation have never made it into a separate paper and the opportunity of writing this thesis was taken to give them an audience.

Chapter 3

Symbolic Regression

Although genetic programming can be used for various automatic programming tasks, this text will focus on the induction of mathematical expressions on data. This is called symbolic regression (Koza, 1992), to emphasize the fact that the object of search is a symbolic description of a model, not just a set of coefficients in a prespecified model. This is in sharp contrast with other methods of regression, including feedforward artificial neural networks, where a specific model is assumed and often only the complexity of this model can be varied.

The regression task can be specified with a set of input, independent, variables $\mathbf{x}$ and a desireded output, dependent variable, t. The object of search is then to approximate t using $\mathbf{x}$ and coefficients $\mathbf{w}$ such that:

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

where ϵ represents a noise term. With standard regression techniques the functional form f is prespecified. Using linear regression for example, f would be:

$$f(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + \ldots w_n x_n \tag{3.1}$$

Where the coefficients $\mathbf{w}$ are found using least square regression. In matrix form this would read:

$$f(\mathbf{x}, \mathbf{w}) = \mathbf{w}\mathbf{x}$$

where the *bias* coefficient w_0 has been ommited for reasons of clarity. The nonlinear technique of regressing a feedforward artificial neural network would introduce an auxillary *transfer* function g (usually a sigmoid) and would use the mapping:

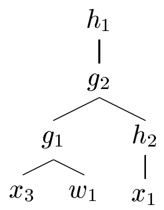
$$f(\mathbf{x}, \mathbf{w}) = \mathbf{w_o} \cdot g(\mathbf{w_h}\mathbf{x}) \tag{3.2}$$

Here the coefficients $\mathbf{w}$ are usually called weights: $\mathbf{w_h}$ are the weights from the input nodes to the hidden nodes and $\mathbf{w_o}$ are the weights from the hidden nodes to the output layer. Again bias weights for each layer in the neural network are ommited in the equation. Due to its functional form, calculating the error gradient for the weights of such an artificial neural network is straightforward and has linear complexity in the number of weights.

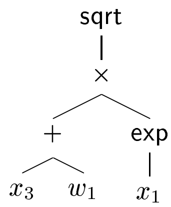
In contrast with these techniques, genetic programming applied to the task of symbolic regression does not use a prespecified functional form. It uses low-level primitive functions. These functions can be combined to specify the full function. Given a set of primitive functions taking one argument $h_1, \ldots, h_u$, and a set of functions taking two arguments $g_1, \ldots, g_b$, the overal functional form induced by genetic programming can take a variety of forms. The functions $\mathbf{h}$ and $\mathbf{g}$ are usually standard arithmetical functions such as addition, subtraction, multiplication and division but could also include trigonometric, logical, and transcendental functions. An example function could be:

$$f(\mathbf{x}, \mathbf{w}) = h_1(g_2(g_1(x_3, w_1), h_2(x_1)))$$

But any legal combination of functions and variables can be obtained. This particular function can be depicted in tree form as:



Filling in some concrete primitive functions for the abstract symbols h and g can lead to the tree:



Or as an expression

$$f(\mathbf{x}, \mathbf{w}) = \sqrt{(x_3 + w_1) \exp x_1}$$

The object of search is then a composition of the input variables, coefficients and primitive functions such that the error of the function with respect to the desired output is minimized. The shape and the size of the solution is not specified at the outset of the optimization (although typically a maximum size is given) and is another object of search. The number of coefficients to use and which value they take is another issue that is determined in the search process itself. The system is also free to exclude certain input variables from the equation, it can thus perform a form of dimensionality reduction. By the use of such primitive functions, genetic programming is in principle capable of expressing any functional form that use these functions: in particular given a sufficiently expressive function set, it is capable of expressing a linear relationship such as in Equation 3.1 or a non-linear relationship such as the artificial neural network in Equation 3.2.

Genetic programming is not the only system capable of inducing symbolic expressions on data. A well known computational work on the induction of equations on data is the program BACON (Langley et al., 1987). In contrast with genetic programming, the BACON system explores the search space of possible expressions using various heuristics. These take the form of numerical comparisons: if for instance two terms (variables or already induced expressions) appear to increase together, an expression will be considered that takes the ratio of the two terms. Similarly, when one term increases while the other decreases and does this in a non-linear way, the product between the terms will be considered.

The heuristics in the BACON system thus relate the numerical values between terms with the symbolic manipulations that will be considered. It will thus produce an expression where all the functions that are applied have this heuristic justification. This presupposes that any mathematical relationship between sets of data can be incrementally build using these heuristics. Furthermore, when the data is polluted by noise, concepts such as jointly increasing or decreasing values become difficult to measure. The application of the heuristics would then need further parameters that need to be set with regard to estimates of the noise.

With genetic programming, the possible transformations are not limited to some set of numerically motivated heuristics. As described in Chapter 2, the variation operators are randomized, while the overall performance of a complete expression is used as the guide to select expressions. The expressions that are induced in this way then do not neccessarily have to abide some internal structure that is incrementally justified. It is thus capable of performing 'creative' computations as long as that decreases the error. This has as a drawback that the expressions that are induced can become too creative, to the point that they are hard to understand.

A genetic programming system performing symbolic regression is thus required to find the shape of the equation, the composition of primitive functions, the use of input variables, the use and values of coefficients plus the complexity of this composition all in a single optimization pass. Furthermore, no gradient information is available about the composition of functions[1], nor numerical heuristics on the structure of the functions are employed. The only feedback the system receives is the overall performance of a given expression on the data given some error function.

This is a daunting task and the question can be asked why one would try to find an expression in such a way when alternatives such as artificial neural networks are available. If one is purely interested in approximating some data, the expressive power of genetic programming provides no immediate benefit over other methods. For example: even though with the proper set of primitive functions the space of artificial neural networks is only a subset of the expressions that can be induced by genetic programming, it has been shown that a feedforward artificial neural network of sufficient complexity can already approximate any mapping (Park and Sandberg, 1991). The question of which technique is more capable of optimizing some data can then not be resolved a priori using the expressiveness of the methods as the main argument. Any comparison would be empirical. It is then expected to find cases where neural networks outperform genetic programming and vice versa.

[1]Though gradient information can sometimes be used to optimize the coefficients (Topchy and Punch, 2001).

Even though the issue of expressiveness does not give an immediate benefit of genetic programming over for instance artificial neural networks in the context of its capability in approximating some data, this expressiveness *is* the main subject of this text. In contrast with neural networks, genetic programming is capable of providing answers in the *symbolic* language of mathematics, while artificial neural networks can neccessarily only provide answers in the form of sets of *numbers*, weights, valid in the context of a prespecified functional form (Equation 3.2).

This contrast between inducing symbolic expressions by genetic programming and matrices of numbers by regression becomes more pronounced when approximating the data is not the only object of search. In scientific discovery for example, obtaining some adequate fit is not enough. To fully incorporate some results in the body of scientific work, it is neccessary that some form of understanding about the expressions that are induced is achieved. The expressions thus need some further justification before they can be used as *models* of the phenomenon under study.

3.1 The Concentration of Suspended Sediment

As a red thread through this work, the problem of finding an expression that predicts and/or describes the concentration of suspended sediment near the bed of a stream is used. Not only is this problem accompanied with some extra-ordinary high quality data, it has been studied intensively by various researchers. This research has lead to an empirical equation for this process that can be used as a benchmark equation. Below, symbolic regression is used to obtain expressions that fit this data.

Background The bottom concentration of suspended sediment is a key parameter within the mechanics of sediment transport. Here the aim is to develop an empirical formulation for the bed concentration c_b, defined at an elevation of a few grain diameters from the bed. This seems to be more reasonable from a physical point of view then defining the reference concentration further from the bottom, since already a few diameters away from the bed the sediment particles are kept in suspension by the turbulence of the fluid rather than by grain-to-grain collisions, and should therefore be regarded as sediment in suspension.

It is normally accepted that the profile of suspended sediment concentration is well described by the Rouse (Rouse, 1939) distribution:

$$c = c_a \left(\frac{D-y}{y} \frac{a}{D-a} \right)^z \tag{3.3}$$

in which

$$z = \frac{w_s}{\kappa u_f} \tag{3.4}$$

In equation (3.3) c denotes volume concentration of suspended sediment; c_a denotes a reference concentration at a distance a above the bed; y denotes vertical coordinate, measured upward from the bottom; and D denotes water depth.

In equation (3.4) z denotes Rouse parameter; κ denotes von Kármán constant (≈ 0.40); u_f shear velocity; and w_s settling velocity of suspended sediment.

If the value c_a appearing in Equation 3.3 is known, the suspended load transport can be easily found as:

$$q_s = \int_a^D c(y)u(y)dy \tag{3.5}$$

where u denotes flow velocity; and D water depth. The integration of Equation 3.5 was performed by Einstein (Einstein, 1950) who assumed the concentration profile to be given by Equation 3.3 and a logarithmic variation of the velocity along the vertical.

The concentration profile $c(y)$ is usually calculated by accepting the diffusion concept for suspended sediment. In steady uniform flow, this leads to a balance between the downward settling of sediment due to gravity and the upward diffusion associated with turbulent fluctuations $i.e.$

$$w_s c + \epsilon_s \frac{\partial c}{\partial y} = 0 \tag{3.6}$$

where ϵ_s denotes diffusion coefficient for the suspended sediment, which is normally taken to be proportional to the eddy viscosity of the flow ϵ.

$$\epsilon_s = \beta\epsilon = \beta\kappa u_f y \left(1 - \frac{y}{D}\right) \tag{3.7}$$

where β denotes momentum correction factor. If the parabolic distribution of ϵ_s given by (3.7) is inserted in (3.6), the profile of suspended sediment concentration given by (3.3) can be obtained by direct integration. The Rouse number z is now described by:

$$z = \frac{w_s}{\kappa\beta u_f} \tag{3.8}$$

In the special case in which the reference level a in (3.3) is taken equal to the distance from the bed to the lower limit of the suspended sediment layer δ, the reference concentration c_a becomes equal to the bed concentration c_b.

One major problem with regard to the bed concentration c_b is the definition of the distance δ. Einstein (Einstein, 1950) suggested δ to be of the order of twice the grain size of the bed material d, and assumed the bed concentration to be proportional to the concentration of bed load particles. Further analysis and sheet-flow experiments showed that the thickness of sheet-flow later δ increases with the Shields parameter θ according to:

$$\delta_s = 10\theta d \tag{3.9}$$

where:

$$\theta = \frac{u_f^2}{(s-1)gd} \tag{3.10}$$

where d denotes median grain diameter (usually indicated as d_{50}); g acceleration of gravity; and s relative density of sediment.

Data A total number of 10 data sets were utilized in the determination of c_b (Guy et al., 1966). The experiments consisted of a number of alluvial channel tests with the aim to determine the effects of the grain size and of water temperature on the hydraulic and sediment transport variables.

The tests were performed in two different flumes: the larger one was 8 ft (2.44 m) wide, 2 ft (0.61 m) deep and 150 ft (45.72 m) long. Its slope could be adjusted between 0 and 0.015, and the water discharge between 0 and 22 cfs (0-0.613 m^3/s). The smaller flume was 2 ft (0.61 m) wide, 2.4 ft (0.76 m) deep and 60 ft (18.29 m) long. Its bottom slope could be varied between 0 and 0.10 and the water discharge between 0 and 8 cfs (0-0.227 m^3/s).

A different kind of sand was used for each set of the tests. The median size d varied between 0.19 and 0.93 mm, while the geometric standard deviation σ_g (defined by Equation 3.11) ranged from 1.25 to 2.07.

$$\sigma_g = 0.5\left[\frac{d_{84}}{d_{50}} + \frac{d_{50}}{d_{16}}\right] \tag{3.11}$$

where d_{50} denotes median particle size of the sediment; and d_{16} and d_{84} particle sizes for which 16% and 84% of the sediment is finer by weight.

The hydraulic conditions of the individual tests were adjusted by changing the discharge, the slope, or both, and the water and sediment were re-circulated until equilibrium conditions were reached. A significant drawback of these data sets is the limited range of water depth covered (from 0.06 to 0.41 m). Apart from that, the tests comprise a wide range of situations, both from the point of view of the hydraulic parameters as well as the bed materials used, the transport rates measured, and the bed forms present, making them very attractive for the derivation of an expression for the near bed concentration in pure current flow.

Table (3.1) summarizes the quantities used in the problem of determination of concentration of suspended sediment near bed. It is interesting to observe that only ν, w_s and d_{50} represent 'raw' observations. Shear velocities u_f and u'_f are calculated on the basis of raw observations as:

$$u_f = \sqrt{gDI} \tag{3.12}$$

$$u'_f = \sqrt{gD'I} \tag{3.13}$$

where I denotes water surface slope; and D' denotes the boundary thickness layer defined through:

$$\frac{v}{u'_f} = 6 + 2.5\ln\frac{D'}{k_N} \tag{3.14}$$

with v denoting mean flow velocity; and k_N bed roughness $\approx 2.5d$.

Human Proposed Relationship for Near-bed Concentration Generally, the near-bed concentration of suspended sediment c_b depends on: *(i)* the effective shear stress exerted on the bed by the flow τ'; *(ii)* the characteristics of the bed

variable	description	**uom**
ν	kinematic viscosity	m^2/s
w_s	settling velocity	m/s
d_{50}	median grain diameter	m
g	gravity acceleration	$9.81 m/s^2$
u_f	shear velocity	m/s
u'_f	shear velocity related to skin friction	m/s
c_b	concentration of sediment near the bed	dimensionless

Table 3.1: Units of measurement of the independent and the dependent variables for the problem of determining the concentration of sediment near the bed.

material (size d, density ρ_s); and *(iii)* the characteristics of the fluid (density ρ, kinematic viscosity ν). Application of dimensional analysis leads to the functional relationship

$$c_b = \left(\theta', \frac{[g(s-1)d]^{0.5}}{w}, \theta_c\right) \tag{3.15}$$

where θ_c denotes a critical value of Shields parameter for initiation of motion. It should be noted that, for a given bed material, the fall velocity w can be uniquely defined in terms of the kinematic viscosity ν (which in turns depends on water temperature) and of the grain size d, so that w in (3.15) can be effectively replaced by ν and T.

(Zyserman and Fredsøe, 1994) followed an approach initially adopted by (Garcia and Parker, 1991) for the selection of an expression for c_b, namely

$$c_b = \frac{Ax^n}{1 + \frac{Ax^n}{c_m}} \tag{3.16}$$

where A, c_m, and n are constants and x a suitable combination of the independent dimensionless parameters. The choice of the functional form (3.16) is driven by the fact that c_b becomes zero when x does as well as c_b converging to the limiting value c_m for high values of x.

The fitting (Zyserman and Fredsøe, 1994) yielded values $A = 0.331$, $c_m = 0.46$ and $n = 1.75$, resulting in

$$c_b = \frac{0.331(\theta' - 0.045)^{1.75}}{1 + \frac{0.331}{0.46}(\theta' - 0.045)^{1.75}} \tag{3.17}$$

The proposed relationship compares well to values of near-bed concentration obtained from independent data sets. It also provides an improved accuracy over similar expert-generated expressions and is universally regarded as the formulation describing the concentration of suspended sediment near bed.

3.2 Symbolic Regression on the Sediment Transport Problem

Straightforward application of symbolic regression The data in the sediment transport problem are accompanied with units of measurements that describe the various data fields. In standard regression, dimensioned variables cannot be used without any pre-processing. Usually one employs some form of pre-processing, be it applying Buckingham's Π-theorem (Buckingham, 1914), or creating an ad-hoc set of dimensionless values using the original data. It is also possible to scale the variables to unit variance, by calculating the standard deviation and divide the original measurement by this value. As the standard deviation of a measurement is stated in the same units as the measurement itself, this scaling will render the pre-processed data dimensionless. Any form of manipulation is subsequently formally allowed.

Here we use the data *as is* to be used for genetic programming. To meet the formal requirements (but not the intent) of modelling using units of measurement, it is simply assumed that the data is divided by a constant of magnitude one stated in the same units as the original data. This conveniently avoids the issues of dealing with units of measurements which will be tackled in later chapters.

After dividing the data in a training set and a test set, a genetic programming system is applied that tries to find an expression that fits the data. The language consist of the observations $\{\nu, w_s, d_{50}, u_f, u'_f\}$, and additionaly the gravity acceleration constant g set at $9.81m/s^2$. Also including arbitary constant values, the full terminal set is described by:

$$T = \{\nu, w_s, d_{50}, u_f, u'_f, g, \mathcal{R}\}$$

The function set is:

$$F = \{\text{plus/2, times/2, minus/2, divide/2, sqrt/1}\}$$

Using a standard genetic programming setup, using 50 independent runs, the best fitting expression on the training set was:

$$c_b \approx 0.284 \left(\left(u'_f - w_s\right)^3 \left(u'_f - g\right) \left(g + \frac{u'_f + u_f}{u_f - g}\right) g^{-5} {u_f}^{-1} \left(g + 13.0 \left(w_s + g^3 u'_f\, w_s {u_f}^{-1} \left(g + \frac{u'_f}{g}\right)^{-1}\right) {u_f}^{-1} \left(u'_f - 11.3 \frac{g u'_f}{\left(u'_f - w_s\right)^2} - g^2\right)^{-1} \left(g + \left(d_{50} + \sqrt{\frac{\left(u'_f - w_s\right)^2 w_s}{g^4}} + g\right) \left(2\,g + \frac{u'_f}{w_s} + u'_f - u_f - w_s + g^2\right)^{-1}\right)^{-1}\right)^{-1}\right)^{\frac{1}{2}}$$

A few observations can be made at this point. The solution produced by symbolic regression on the raw observations presented above is very complex. It manipulates the variables in a variety of intractable ways, using high order polynomials and repeated square roots. If we were to complicate things by adding trigonometric and transcendental functions, the genetic programming system would quite likely find a way to use this enhanced expressive power to obtain a better fit on the training data. It is quite likely that the net result would become an even more incomprehensible set of manipulations.

Another problem with this equation is that it does not use the units of measurement properly. No information about these units was given, and it is would have been quite unlikely to get a dimensionally correct result. However, the absence of the dimensions makes the equation even harder to comprehend.

Pre-processing An approach more in line with the practice in physics is to pre-process the variables to render them dimensionless. Applying the Π-theorem using the variable d_{50} and the constant g to perform the transformation, it is possible to reduce the number of independent variables by using the transformation:

$$
\begin{aligned}
\pi_1 &= w_s d_{50}^{-0.5} g^{-0.5} \\
\pi_2 &= u_f d_{50}^{-0.5} g^{-0.5} \\
\pi_3 &= u'_f d_{50}^{-0.5} g^{-0.5} \\
\pi_4 &= \nu d_{50}^{-1.5} g^{-0.5}
\end{aligned}
$$

As c_b is already dimensionless no pre-processing needs to be done on this variable. Running a genetic programming system using the π variables as input produced as the best expression:

$$
\begin{gathered}
c_b \approx \\
\pi_3 \Bigg(\pi_1 \left(3262\,\pi_3{}^{-1} - \pi_3\right) \frac{1}{\sqrt{\pi_3}} + \pi_2{}^3 \pi_4{}^{-1} \pi_3{}^{-1} \Bigg(\pi_2 + 34.2 - 1.09\,\pi_2{}^2 \pi_4 \frac{1}{\sqrt{\pi_3}} \\
\left(\pi_2 - \frac{\pi_3}{\pi_4}\right)^{-1} - \pi_4 + \pi_3 + \frac{\pi_2{}^2}{\sqrt{\pi_3}\,(2\,\pi_3 - \pi_1 + \pi_4)} \Bigg)^{-1} + \pi_3 + \pi_4\,\pi_3 \\
\left(\frac{\pi_2 \sqrt{\pi_3}}{\pi_4} - \frac{\sqrt{\pi_3}}{\pi_4} \right) \left(\pi_3 + \pi_2 \sqrt{\pi_3} \left(\pi_3 + \frac{\pi_3}{\pi_2\,\pi_4} \right)^{-1} \pi_4{}^{-2} \right) \\
\left(\pi_3 + 0.054\,\pi_2{}^{-2}\right)^{-1} \Bigg(57.1\,\pi_3{}^{-1} + 33.3 - 76.6\,(\pi_3 + \pi_1)^{-1} \\
\left(2\,\pi_4 - \frac{\pi_2}{\pi_3} \right)^{-1} \pi_4{}^{-1} \Bigg)^{-1} \Bigg(\frac{\pi_1\,(57.1 - \pi_3)}{\sqrt{\pi_3}} + \frac{\pi_2{}^3}{\pi_4\,\pi_3\,(\pi_2 + 0.54)} - \pi_2\,\pi_4{}^2 \\
\left(57.1\,\pi_1{}^{-1} + 33.3 - 93.2\,\frac{1}{\pi_3\,\pi_1} \right) \frac{1}{\sqrt{\pi_3}} (\pi_3 - \pi_4 + 8.81)^{-1} \Bigg)^{-1} \Bigg)^{-1}
\end{gathered}
$$

Datasets	Zysermann & Fredsøe	SR (Raw Values)	SR (Π-theorem)
train	0.051	0.038	0.035
test	0.047	0.047	0.054
train + test	0.049	0.041	0.046

Table 3.2: Comparison of the benchmark formulation produced by scientists and the formulations found by symbolic regression (SR) using the raw values or the pre-proccesed values. The error measure that is used is the Root Mean Squared Error (RMS). The benchmark formulation was induced using all data, the split in test and training set presented here was used in the symbolic regression experiment.

This equation is also extremely complex, quite possibly unneccesarily so. Although the equation is dimensionally correct, it is such by virtue of the inputs being dimensionless. All arithmetical operations are then allowed. To interpret this expression, an additional translation needs to be performed, where the variables are translated into their original definition.

In these two experiments it was not attempted to reduce the complexity of the equations. Genetic programming practioners often use some form of parsimony pressure to influence the size of the solutions (Zhang and Mühlenbein, 1994; Zhang and Mühlenbein, 1996; Iba et al., 1994; Zhang, 2000; de Jong et al., 2001). This might help somewhat, though the main problems of the two experiments — ignorance of units, or extensive pre-processing — will remain.

Even though is not clear what data is used to induce Equation 3.17, a comparison on the performance on the different sets used in the experiment is presented in Table 3.2. The performance of human induced equation, which uses Shield's parameter θ', comes from the fact that this particular variable is already highly correlated with the concentration. Performing a linear regression on θ' alone produces an expression that has a RMS error of 0.050, slightly worse than the performance of the benchmark equation 3.17.

3.3 Summary

Using symbolic regression alone does not seem to help much in providing interesting hypotheses in this domain. When using 'raw' observations, the resulting expression can become very complex very easily. No attention is given to the units of measurement, it merely presents a numerical recipe to map input numbers to a predictive value. This prediction might be good or bad, depending on some statistical estimate of the generalization error. The expressions that are produced might be symbolic, the language that is used seems to be alien: it does not give the user much information about the process underlying the data.

Relying on dimensionless values does not help either. It has the same problems with the complexity of the solutions. In effect, the dimensionless values help to further obfuscate the relationships found by their reliance on an extra translation step.

Although the ability to fit the data is reasonable, the symbolic regression runs do not add much to our understanding of the problem. It thus presents a similar black-box model similar to those produced by artificial neural networks. The only approach that seems to be feasible is to attempt to control the size of the solutions so that short solutions are produced. Analyzing such short solutions might be possible, though then still interpretation needs to be performed mainly using numerical arguments. This is not much different from interpreting a small artificial neural network.

In the case of the sediment transport problem, the Shield's parameters that were provided were highly correlated with the target variable, on their own they already provide a robust estimate. Often however, formulating such a parameter is exactly the object of search.

Chapter 4

Induction of Empirical Equations

Suppose that we are given the task to model an unknown or poorly understood system. In such situations a logical starting point is the design of measurement campaigns and the collection of data. One usually measures forcing variables (the ones that are outside the system) and simultaneously the response of the system in view of the change of the state of the system (state- or internal variables), and the change in corresponding output of the system (resulting functions). After enough data of sufficient quality are collected, one can attempt to identify the system. Then, three possible scenarios can occur (Kompare, 1995):

1. Nothing useful can be concluded from the observations. This can happen if the measuring campaign was poorly designed, or nor carried out over a sufficiently long period of time, or if relationships among variables simply do not exist. More measurements, or redesigned more elaborate observations are needed to improve the situation.
2. Sometimes we may end up with a statistical, black box model. With this category of models we will be able to predict the proper behaviour of the system, although we will not be able to characterize its intrinsic structure and behaviour. In other words, we will be able to say what the model does, but not how. In addition to this, we will not be able to guarantee the behaviour of such model in regions not covered by the data from which the model was constructed. This is due to the fact that the model covers only the relationships found within the given data.
3. In some cases we may be able to recognize patterns within the data and form from these patterns inference about basic processes in the observed system. After repeated measurements we should be able to develop a conceptual (mechanistic) model. Such a model is a so-called white box, or transparent model and we should be able to say what the model describes and how its performance is achieved. Due to the conceptual background of the model, we are much more certain that the model will represent reality. This also helps when using the data out of the range in which model was constructed.

Quite obviously, the likelihood of us being able to construct a conceptual model for an unknown or poorly understood system increases with both the quality and the quantity of the observed data. To make the most of some experimental data, it is generally desirable to express the relation between the variables in a symbolic form: an equation. In the view of the approximate nature of this functional relation, such an equation is described as empirical. No particular stigma should be attached to the name since many ultimately recognized chemical, physical and biological laws have started out as empirical equations.

Sciences devote particular attention to the development of a symbol system, such as a scheme of notation in mathematics, together with more refined representations of physical and conceptual processes in the form of equations in the corresponding symbols. Each equation can be regarded as a collection of signs, which constitutes a model of an object, process or event. Data, on the other hand, remain as mere data just to the extent that they remain a collection of signs that does not serve as a model.

From this point of view, the induction of an equation within a symbol system represents a means of better conveying the meaning or semantic content encapsulated in the data. Induction of an equation corresponds to finding a model. In this process the information content is very little changed, or even unchanged, but the meaning value is commonly increased immensely. Since it is just this increase in meaning value that justifies the activity of substituting equations for data, there is a great interest in processes that induce equations from data.

Chapter 3 showed that the use of symbolic regression as such does not provide this increase in meaning value directly. The interpretation of the symbolic expressions is hindered by the size but even more by the seemingly arbitrariness of the computations that are performed. The system that produces these expressions is only guided by numerical measures such as the error it makes. The computations that are performed inside the expressions are then tailored to get this error down in whatever way possible. With symbolic regression there is no explicit means to create expressions that can be interpreted by a scientist, and no means to justify the use of the expressions that transcends statistical statements about the performance. Without such means, interpretability of the expressions is coincidental. What is needed is a method that can help in the interpretation of the expression and the sub-steps that are taken in the computation. If the computation defined by the expression can be related to the physical process that it models, it is possible to consider it a hypothesis of the system, instead of a black box. The method to aid in this interpretation that is considered here is the use of units of measurement.

Throughout science, the units of measurement of observed phenomena are used to classify, combine and manipulate experimental data. Measurement is the practice of applying arithmetic to the study of quantitative relations. Every measurement is made on some scale. According to Stevens, to make a measurement is simply to make 'an assignment of numerals to things according to a rule — any rule' (Stevens, 1959). There is a close connection between the concept of a scale and the concept of an application of arithmetic. Units of measurement are the names of these scales. Simple unit names such as 'kilogram', 'second', 'degrees are used for fundamental and associative scales. Complex unit names, such as 'kg m s^{-1}' are used for derivative scales.

Common methods for finding equations based on data usually involve a dimensional analysis (which attempts to remove issues of scale) and subsequent curve-fitting by hand or automatic means. An example of this was given in Chapter 3. Here it is suggested that an approach in which the dimensions — physical units of measurement — of the data can be used as an additional source of information in order to help creating expressions, as well as checking their validity and usefulness. Rather than ignoring dimensions altogether, or proposing dimensionless formulae (*i.e.*, expressions based exclusively on dimensionless numbers), the objective is to create fully dimensioned formulae. It is postulated that such formulae can be easier interpreted. Then, if a fully dimensioned expression is obtained, it can provide a basis for the creation of a mechanistic, white box model.

Using units helps in transforming physical concepts into mathematical expressions. If a physical concept or physical manipulation is sensible and numerically analogous to some application of arithmetic, the substitution of the resulting equation to better describe the data is justified. The reverse is however not necessarily true: not every application of arithmetic on measurements is rooted in physical reality. For example: the addition of the lengths of two sticks can correspond with a proposal to combine the sticks to form one longer stick. The results produced by the addition then describe the length of the combined stick. In an experimental situation this might or might not be a sensible proposal. Dividing the two length of these sticks can have a variety of interpretations: it can correspond with the sine of the angle the two sticks make in the case the sticks are part of a triangle, but it can also be an operation to normalize the lengths in the case they are independent measurements. Without reference to the physical experiment that is described, it is impossible to determine the exact meaning of such an application of arithmetic. Relating a dimensioned formulation to the physical experiment is then the role of the human user. The dimensioned formulae are tentative proposals: the units of measurement that are manipulated by the formulae form a guide to their interpretation.

To achieve the goal of creating fully dimensioned formulae, genetic programming is used. One of the advantages of genetic programming over other methods for regression is the symbolic nature of the solutions that are produced. In the natural sciences for instance, a symbolic answer in a language the user understands, mathematics, provides a great benefit over methods that produce coefficients in a prespecified model. This is especially pronounced in empirical modelling of unknown phenomena where an underlying theoretical model does not exist. As was shown in Chapter 3, the solutions produced by genetic programming can not be interpreted at all times. The size of the solutions produced can hinder interpretation, while setting the size to low limits can hinder the search efficiency.

The goal of this approach is simple: to create a genetic programming system that produces equations that are easier to interpret by domain specialists. The system is thus applied to the area of *scientific discovery*. The GP-produced equations are supposed to form a set of hypotheses in and about the domain, stated in the symbolic language of equations. Rather than producing black-box solutions to problems, the aim is to provide statements where the units of measurement help in interpreting the expression and ultimately help in forming an enhanced view of the problem.

Operation	Type
Addition/Subtraction	$([x,y] \to [x,y] \to [x,y])$
Multiplication	$([x,y] \to [v,w] \to [x+v, y+w])$
Division	$([x,y] \to [v,w] \to [x-v, y-w])$
Square Root	$([x,y] \to [x/2, y/2])$
a^c	$([x,y] \to [0,0] \to [cx, cy])$
Transcendental Functions	$([0,0] \to [0,0])$

Table 4.1: The type system defined by the physical units of measurement. It defines constraints in the case of addition and subtraction where the units of the operands need to be the same, in the case of trigonometric, hyperbolic, exponential and many other functions the units of the operands need to be dimensionless. Multiplication, division and the square root function are always defined, but introduce arithmetical manipulations on the types. Finally, the power function a^c is only defined when the second operand is a constant, whose value will influence the output type. Here the actual value of the expression influences its type.

4.1 Units of Measurement as a Type System

Consider a variable v measured in units $L^x T^y M^z$ where L, T and M are the dimensions of length, time and mass respectively and x, y and z the corresponding exponents. When one of the exponents is unity and the other exponents zero, the unit of v is referred to as a *base unit*. When all exponents are zero the unit is called *dimensionless*. In all other cases we speak of *derived units*. Furthermore, vector notation for the units such that $\mathbf{u} = [x, y, z]$ is used to denote the vector of exponents. This vector of exponents contains all information necessary to make statements about the units of measurement of variable v.

For example: $\mathbf{u} = [1, -2, 1]$ defines a derived unit of force, whether it is measured in $kg\ m/sec^2$ or in $lbs\ ft/sec^2$. Although in this paper the SI units of measurement are used, other units such as for example *income per capita* can also be defined. For notational convenience a smaller system, consisting of two physical dimensions is used below (Table 4.1). This generalizes trivially to arbitrary numbers of dimensions, physical or otherwise.

The term *type system* is used to refer to the combination of type specifications of variables and constants together with the type specifications of the operators. The notation for this type system is borrowed from the typed λ-calculus, in which $(T \to U \to V)$ denotes a function that requires two arguments of type T and U and returns a value of type V. The types in the units of measurement (**uom**) system are then real valued vectors.

The constraints on the mathematical operators involved in **uom** problems are specified as follows: each operator can impose constraints on its operands (for instance equality requirement in the case of addition) or it can specify manipulations in order to produce the output type from the input types as for example in the case of multiplication. Several constraints and manipulations are defined within the **uom** system as specified in Table 4.1.

Along with the definition of the independent and dependent variables and possibly typed constants, this type system defines an uncountably infinite number of types, where any real-valued vector of the appropriate size is a data type in its own right. If all variables and constants are dimensionless, the language reduces to an untyped language. In this case, no manipulations can introduce non-zero exponents.

The existence of derived units makes this type system more complex than the type systems usually used in computer languages: these define only base types such as `float`, `int` and `string`, the only way to combine them is to put them in `structs` or `tuples`.

4.2 Language, Bias and Search

With defining the type system for the units of measurement, a *language* of expressions in this type system is defined. Several routes can be taken at this point. The most obvious is to implement this type system in genetic programming in such a way that the system will only induce expressions that are dimensionally correct. This is the area where *Strongly Typed Genetic Programming* is employed. Below a review of such systems is presented, together with some argumentation on how many of the existing systems are either not powerful enough to express the language of dimensionally correct expressions, or why they are not expected to perform well on this particular problem domain. This discussion leads to the formulation of a new strongly typed genetic programming system which is fully described in Chapter 6.

However, strictly abiding the constraints imposed by this type system might not be the most appropriate approach to create *useful* expressions i.e., workable hypotheses that provide maximum insight into the problem. Although 'getting the units right' has been hammered into many scientists and engineers, it is important to consider that with the automatic induction of expressions based on data and units of measurement it is not tried to induce scientific law from examples, or even to make a statement about cause and effect. The goal is to create a good performing expression that helps in the analysis of causes, an enhanced conceptualization of the problem, which might ultimately form the basis of a new empirical law. This law should be proposed by the domain expert however, as the hypothesis generation engine envisioned here can only provide expressions; not justifications.

Interpreting arithmetic applied to measurements is not clear cut. A mathematical operation can apply to many things which are difficult, if not impossible, to interpret as a realistic description of a physical process. Above, examples were given about the possible meanings a simple division of two length measurements can have: depending on the experimental layout of the measurements this division operator can indicate a measurement of the sine of an angle or a simple normalization. The same holds for other operations. Although the units of measurement provide some information about the use of the variables, applying arithmetic on those variables needs further justification that goes beyond formal means.

Modelling using units of measurement balances on the border between causative modelling and modelling by association. As any statistical textbook warns: correlation does not equal causation. Trying to reduce the error for some model is just a particular form of maximizing correlation between a model and a dependent variable. Limiting the independent variables to forcing variables and including the units of measurement in the search might include some causative element in the search. However, restricting the class of sentences that might be produced to those that abide all the restrictions in the measurements might miss out on important associations between measurements. Even more, a formally correct manipulation can be meaningless. Taking the arc tangent of a ratio of two weight measurements will formally produce an angular measurement, but there is no physical interpretation of the angle between two weights.

Because model induction is used in areas where no predictive theory exists [1], it can not be established that everything that is measured is relevant in exactly the form (units) that it is measured in. It might turn out that a certain measurement is associated with a related property of the problem, stated in different units. For example, a measurement in length units might be best used as if it was stated in surface units — simply because the measured value determines the value along one axis of a surface variable, while the value along the other axis is constant. Such a variable stated in length units would then be proportional to the unmeasured surface variable. By strongly abiding to the units in the problem, such a use of this variable will never be considered, possibly leading to the premature conclusion that the particular set of measurements is useless.

One particular solution to these kind of problems is to introduce constants stated in arbitrary units. Then it is possible to multiply the example variable measured in length units with a constant value stated in length units to obtain the desired measurement in surface units. This will however defeat the entire purpose of modelling using units of measurement: the units of any measurement can then be transformed to any other unit by multiplying it with an appropriate constant. We can even perform this as a pre-processing step, leading to a symbolic regression set up as presented in Chapter 3.

As a formal system, the use of units of measurement poses a few more problems. Given the existence of constants stated in arbitrary units it is easy to trivialize dimensional correctness by making liberal use of these constants. If arbitrary dimensioned constants are absent, it is quite likely that modelling will not succeed as experimentally collected data cannot vary or measure everything that is relevant. Any given measurement might be indicative for a range of units. A variable stated in length units might be proportional to a rectangular surface if the other axis of the rectangle is kept constant. Likewise, a set of length measurements can be proportional to a set of velocity measurements if all experimentation is performed using a fixed period of time. The details on the possible roles a single variable can take is determined by the physical setup of the experimentation and can be hard to exhaustively specify beforehand.

The use of a dimensionally (more or less) correct expression can however be great for the scientist. By balancing dimensions, an expression stated in some particular units can point to phenomena in the problem that are not obvious. By analyzing such formulae, the scientist might be able to update a mental picture of the phenomenon under study and gain new insight. This analysis needs to be performed with respect to the way the data is produced: the experimental setting. This setting is only present in a watered down form for the model induction engine, in the form of units of measurements.

Abiding rigorously to the units of measurement implements a *declarative bias*. As it

[1] If such theory did exist, we would not bother performing predictive modelling.

involves a type system, this particular form of bias is called *semantic bias* (Muggleton and Raedt, 1994). A declarative bias imposed on a language reduces the set of sentences that can be derived: it thus restricts the search. It is however not a priori guaranteed that this bias is justified in a set of empirically collected data. It was attempted to show that a declarative bias to using only the units in the experimental setup can be misleading in at least two ways: dimensionally correct expressions are not guaranteed to be meaningful, as not all formally allowed arithmetic operations will have a physical interpretation; and formally incorrect expressions can be meaningful when a measurement is associated with another phenomenon to which it is proportional.

To also investigate formally incorrect expressions, it might then be worthwhile to examine ways of *guiding* the search rather than restricting it. Changing the search to make it more likely that a certain class of sentences is produced is called implementing a *preferential bias* (Muggleton and Raedt, 1994). A system that uses a preferential bias towards dimensionally correct expressions is presented in Chapter 5. It is hypothesized that such a system — which allows errors in the units to occur — provides a more fertile ground for the ultimate goal of modelling using units of measurement: understanding the data better by analyzing expressions that fit the data well.

Apart from the possibility that formally incorrect expressions can provide additional information, the use of a preferential bias might also help simply as an enhancement of the search capabilities of the system — even when ultimately only correct expressions are considered.

4.3 Typing in Genetic Programming

Strongly typed genetic programming (Montana, 1995) was the first of many approaches that constrain the allowable programs in genetic programming by means of a type system. The purpose of a strongly typed system is to make only those programs well-formed that are type correct. It thus attempts to reduce the search space to the space of correctly typed programs. It thus introduces a *declarative bias* in the search space.

Tree Based GP Montana's work introduced the concept of a *generic function* in genetic programming. A generic function is well-defined over all or a well-defined subset of available types. As an example: in Table 4.1, the constraints on the **uom** type system are defined as generic functions.

Strongly typed genetic programming aims at initializing and maintaining a population consisting of correctly typed programs. The goal is to optimize the programs with respect to some objective function. When these programs are represented as trees, most effort is devoted to defining a suitable initialization routine and a strongly typed *subtree* crossover.

Several approaches have followed upon this work. The following are discussed here: context free grammars (point typing) (Gruau, 1996; Whigham, 1996a), parametric polymorphism (generic typing) (Yu and Clack, 1998), inheritance (subtyping)

(Haynes et al., 1996), and logic grammars (Wong and Leung, 1997). These approaches share the use of a tree representation in order to store the program along with the type information, together with the definition of variational operators that manipulate this tree representation. Most notably, a strongly typed subtree crossover is defined that exchanges parts of the programs while keeping the type information in the tree up to date and correct.

Developmental GP As an alternative to using trees as the representation, developmental approaches have been proposed (Banzhaf, 1994; O'Neill and Ryan, 2001). Here linear strings of bits or integers are maintained, that are mapped into an expression using some derivation process. Developmental GP makes a distinction between an untyped *genotype* — the string — and a typed *phenotype* — the derivation tree and ultimately the expression generated by the string.

In contrast with tree based approaches, the variational operators in developmental GP are simple and untyped: the string representation is directly manipulated and all issues concerning typing are handled in the derivation process. These systems are strongly typed as they only produce correctly typed expressions. The main difference with tree based approaches lies in the absence of strongly typed variation operators. String based systems can employ syntactic constraints such as context free grammars (CFGs), whose content is a disjunction of production rules. This implements a declarative bias in the form of a syntactic bias (Muggleton and Raedt, 1994). Recently however, grammar-based approaches have been successfully extended to use logic programs (Keijzer et al., 2001a) that can model context-sensitive information. Chapter 6 is devoted to the introduction of this logic programming based developmental system.

4.4 Expressiveness of Type Systems

A context free grammar (CFG) can implement only a limited type system, and there is no notion of *generic functions*. Because of this, a CFG needs a separate symbol for each type in the grammar. This is called point typing. As an example, consider the usual arithmetical function set and two terminals, `x` and `y`. The set of parse trees can be defined by the context free grammar:

Grammar 1 *A Lisp-style grammar using a single type:*

```
<expr>  ::=  x |
             y |
             (sqrt <expr>) |
             (+ <expr> <expr>) |
             (* <expr> <expr>) |
             (- <expr> <expr>) |
             (/ <expr> <expr>).
```

where the CFG symbol `<expr>` is of type `double`. The types and arity of the function set are hidden in the production rules for `<expr>`. An equivalent grammar treating T and F as terminal symbols which produces sentences in a more C-style language is:

Grammar 2 *A context-free grammar for symbolic regression:*

```
<expr>      ::=  <terminal> |
                 <mon_op>(<expr>) |
                 (<expr> <bin_op> <expr>).

<terminal>  ::=  x | y.

<mon_op>    ::=  sqrt.
<bin_op>    ::=  + | * | - | /.
```

The type system indirectly implemented by this grammar is defined as: `<expr>` and `<terminal>` are of type `double`, `<mon_ op>` is of type (`double` $\rightarrow$ `double`) and `<bin_ op>` is of type (`double` $\rightarrow$ `double` $\rightarrow$ `double`). By adding more symbols, other types can be incorporated in the grammar. Customarily, the `<terminal>` symbol is defined as a separate symbol from the `<expr>` symbol even though they have the same type.

There is no apparent benefit for preferring one grammar over the other. In the literature, Koza-style genetic programming uses (albeit implicitly) the first, while users of CFG based genetic programming seem to prefer the second (Whigham, 1996a; O'Neill and Ryan, 2001). Although the use of different grammars can result in a radically different performance, it is in general not possible to choose an optimal or even a good grammar in advance.

Although a context free grammar can be used to specify the syntax of an arbitrary computer language, the importance of such syntactical issues is very limited. In contrast with parsing, generating sentences in a specific computer language is trivial when the computation that needs to be performed is represented in an unambiguous format. A parse tree is such a format. If pure syntactical issues — such as where to put a semi-column — are hardly relevant, why are context free grammars in common use in genetic programming?

One reason for using context free grammars is to implement a type system that can be used to constrain the expressions that can be produced. When the function set is composed of — say — logical functions and arithmetical functions, a context free grammar can be used to make sure that boolean and real-valued types do not get mixed. The type system that can be implemented with a context free grammar alone is however severely limited as each type needs to be represented by a set of production rules.

Another potential benefit in using context free grammars was identified by Whigham as the possibility of changing the bias of a genetic programming system by changing the grammar, while leaving the language (the set of possible sentences that can be expressed) intact (Whigham, 1996b). As it is difficult to find a good grammar in advance, Whigham experimented with changing the grammar during optimization (Whigham, 1996a).

Context free grammars are not well suited to model the **uom** system. Since there is an uncountably infinite number of types present in the **uom** system, a full specification is impossible. There have been implementations for a subset of the **uom** system (Ratle and Sebag, 2000), where all integer units in the range $[-2, 2]$ have been modelled using the restricted function set of $\{+, *, -, /\}$. Since this gives 5 different types per dimension, a full specification of the grammar (excluding functions such as sqrt) for a problem stated in LTM requires $5^3 = 125$ different symbols, each having many rules (for example, the multiplication and division operators requiring up to 125 rules per type):

```
<exp_in_0_0_0> ::=
   (<exp_in_0_0_0> * <exp_in_0_0_0>) |
   (<exp_in_1_0_0> * <exp_in_-1_0_0>) |
   (<exp_in_2_0_0> * <exp_in_-2_0_0>) |
   ...
   [followed by 122 more definitions for multiplication].
```

Due to the tediousness of writing such a grammar by hand, (Ratle and Sebag, 2000) used a grammar generation routine. A more detailed discussion of the effectiveness of such a grammar in combination with a strongly typed subtree crossover is postponed until Section 4.5.

Typing through inheritance (subtyping) is more expressive than using CFGs. The approach is used in modern Object-Oriented languages and in its most basic form models an **is-a** relationship. Although existing, the support for generic functions is limited. For example suppose that a base type *object* is defined as well as two derived types *integer* and *float*. To define a generic list, in the subtyping approach one would need to create a type *object_ list*[2]. It is now possible to add *integers* and *floats* to the list through their ancestor *object*. When retrieving an element from this list however, one retrieves something of type *object*, rather then an *integer* or *float*. In general a runtime check must to be performed in order to determine the type of the *object*. If the goal is to create a list of integers, a whole new type *integer_ list* needs to be defined, duplicating the functionality of the *object_ list*. As the **uom** system requires generic functions where the actual type of the operands is required to calculate the output type, the subtyping approach seems not to be suitable.

Strongly typed GP through parametric polymorphism (generic typing) (Yu and Clack, 1998) is modelled on the basis of modern functional languages such as Haskell, that have their roots in the typed λ-calculus. It presents a complete type system where type variables and thus generic functions play a prominent role. For the previous example, given two types *integer* and *float*, it is possible to define a generic list by the type specification $[T]$, where T denotes a type variable, or parameter (hence the name parametric polymorphism). A function *cons* can easily be defined to be of the type $(T \to [T] \to [T])$, meaning that the first argument is of type T, the second argument a list of Ts and it returns a list of Ts. A unification procedure infers the types in the tree generation routine, and ensures that for example no *float* can be added to a list of *integers*. Parametric polymorphism enhanced

[2]In fact, the Java language gives such a list-of-objects in its standard library.

with type arithmetic is well suited for implementing a strongly typed version of the **uom** system. In this paper logic programming is used in order to implement this type system.

Similarly to context free grammars, logic grammars do not form a type-system *per se*, but rather a definition of a (computer) language. Logic grammars are more powerful than context free grammars since additional — semantic — information can be manipulated as well. A logic grammar is usually translated into a logic program that can parse and generate sentences in the language. Chapter 6 will introduce a system for performing strongly typed genetic programming on logic programs directly.

4.5 Typed Variation Operators

Most strongly typed genetic programming systems rely on strongly typed crossover operators that attempt to keep the expressions correct at all times by only swapping type-compatible sub-expression. Two a priori arguments in favour of relaxing the type constraint for the variational operators are identified here: a general argument involving ergodicity of the search space and a specific argument involving loss of diversity for subtree crossover.

4.5.1 Broken ergodicity

Strongly typed genetic programming systems vastly reduce the size of the search space (Montana, 1995) by excluding all but correctly typed formulations. For enumerative or random search this considerably accelerates the search process. However, for algorithms utilizing some notion of a neighbourhood, a reduced search space may be detrimental, especially when the resulting space is fragmented with respect to the neighbourhood function. In the case of simulated annealing for instance, an a priori requirement for global convergence is that the search space is ergodic: any point in the search space needs to be connected to any other point in a finite number of steps. When this condition is not met, the search method is said to suffer from *broken ergodicity* (Palmer, 1982). Broken ergodicity implies that the results strongly depend on the initial conditions. The apparent focus on 'proper' initialization in the genetic programming literature already points at the existence of a problem.

As an example of fragmentation, consider the **uom** system with strongly typed subtree crossover and mutation. For example, let us consider a node that adds two velocities $[m/s]+[m/s]$. Both of the velocities are calculated in subtrees below the addition node. Once these types are instantiated, subtree crossover and mutation are forced to treat the arguments of the addition as velocities. The variational operator cannot change one of the arguments to, say, a subtree stated in units of length as this would produce an inconsistent tree. There is therefore no path to incrementally transform the addition of velocities to an expression stated in other units, no matter how beneficial for the performance of the expression this might be. The system is always forced to treat the expression as being a statement in velocity units.

Using strongly typed variation operators, all changes to an expression (be it using crossover or mutation) are made in the context of the types present in the unchanged part of the expression. This has the potential to lead to a strong dependence on the initial population, where the typing structure of the best performing expression determines a template the rest of the optimization has to conform to. As this template is chosen relatively arbitrarily (it is chosen on the basis of a limited sample of randomly generated programs that might have undergone only a few rounds of selection and variation), this can have a strong impact on the ability to search well.

4.5.2 Loss of diversity

Another problem for tree-based genetic programming in the context of a type-system lies in the availability of genetic material (subtrees) to be recombined using strongly typed subtree crossover. Consider a system based on a tree-based strongly typed approach, using strongly typed variational operators working on a type-system containing T types (or a context free grammar with T non-terminal symbols). Since a strongly typed subtree crossover only swaps subtrees of the same type, the code base of subtrees present in the population is effectively partitioned into T different subspaces: one for every type. There is also only a limited capacity of types that can be present in any one individual due to limitations of size.

Although strongly typed subtree crossover is capable of creating new subtrees, it is usually not capable of creating new types: exchanging material of the same type will generally not change the type of the node above the exchange spot. An exception is the exponentiation rule (see Table 4.1) where the value of the constant influences the type. Exchanging constants with different values will change the type of the expression, potentially leading to a type error[3].

Due to the effects of repeated selection and subtree duplication by subtree crossover, it can be expected that number of *distinct* subtrees for any one type will be only a small fraction of all available subtrees (Keijzer, 1996). The mutation operators are then solely responsible for introducing new types in the population, leading to the expectation that subtree crossover will loose its effectiveness during the run.

4.6 Summary

This chapter lays some groundwork for the chapters that follow. The goal and method of this thesis is identified: making the equations produced by genetic programming more suitable for analysis and interpretation by the use of units of measurement as a type system.

The system of units of measurement is defined in the notation of the typed λ-calculus, where the units are described by a vector of exponents. These vectors form the types in this system. Because the number of types as such is uncountably infinite, simple type systems cannot cater for this level of expressiveness. It was

[3]When using context sensitive grammars, such errors cannot in principle all be checked syntactically, therefore (Wong and Leung, 1997) implemented a semantic validation procedure to check if all constraints are satisfied after a subtree crossover event.

shown that specifically a context-free grammar can not be used to specify all possible types in this system.

A brief review of strongly typed genetic programming is given, showing that the most common approach to ensure type correctness of expressions is by employing operators that keep all types correct at all times. Some arguments are given why strongly typed variation operators are not necessarily the optimal approach to ensure this type correctness. A brief hint is given that in developmental genetic programming systems untyped variation operators can be used. This will be explored in depth in Chapters 6 and 7, where a developmental genetic programming system is introduced that can implement the units of measurement type system in full generality, without the need for strongly typed variation operators.

The term *declarative bias* is associated with strongly typed genetic programming. In the units of measurement system, the language of all possible expressions is reduced to only those expressions that are dimensionally correct. Declarative bias is a method of introducing background knowledge from the problem domain into the search. By reducing the number of well-formed expressions, a search speedup is expected (Montana, 1995). Above it is argued however that such a speedup can only be expected when enumerative of random search are used; the introduction of declarative bias in search techniques that employ some form of neighbourhood function — such as genetic programming and simulated annealing — might hinder the search in unforeseen ways by breaking the ergodicity in the neighbourhood function defined by the search operators.

Chapter 5 will however introduce a method where the information about the units of measurement is not taken as an a priori reduction of allowed expressions. It will implement the units of measurement as a *preference* rather than a constraint. This will be called a *preferential bias*, where the search space is not reduced, but extra information about the amount of typing error is included in the performance evaluation. A multi-objective search is then used to find the optimal balance between the fit on the data and the type consistency of the proposed formulations.

Chapter 5

Dimensionally Aware Genetic Programming

Strong typing is not the only approach imaginable to obtain expressions that use units of measurement. Due to possible problems with ergodicity, but also because the particular set of units that are used in the experimental setup cannot be expected to present the best possible set, a less strict approach might be worth investigating. Such an approach is the coercion based approach introduced here.

Rather then insisting on keeping the expressions correctly typed at all cost, the approach relies on a set of *coercion* functions of arity 1, that map one type into some other type, any other type. Coercions in computer languages are often useful when the type system prevents otherwise sensible operations. Without coercions, it would for instance be hard to add an integer value to a floating point value; an operation that from a mathematical point of view should not pose any difficulties.

As coercion functions circumvent a type system, too many coercions make the code hard to read and interpret. With the coerced genetic programming approach the number of coercions is used as a second objective, coercions are thus allowed when they help in better solving the primary objective, while gratuitous coercions are punished when alternatives are present.

The main philosophy behind this approach is that while type correctness can help in creating *readable* and *interpretable* computer programs, a rigorous adherence to a specific type system might exclude the induction of well-performing expressions. By refusing to view the constraints imposed by a particular type system as hard constraints, it is expected that this leads to a more efficient search. Furthermore, a limited number of typing errors can be quite acceptable if it helps the performance.

Because it is not clear what the optimal balance between type correctness and performance is at the outset of the experimentation, a multi-objective strategy based on Pareto optimality is used. This has an advantage over penalty functions (Yu and Bentley, 1998) that no a priori choice has to be made about the balance between performance and type correctness. Due to the existence of coercion functions, incorrectly typed expressions are not viewed as illegal expressions, as they can still be executed. More importantly even, a run results in a front of non-dominated solutions that balance performance and type correctness in unique ways. Inspecting the exact balance achieved between type violations and performance can highlight problems in the problem definition and might lead to additional insight.

5.1 Coerced Genetic Programming

The name *Coerced Genetic Programming* is chosen as the general name for this approach. Rather then avoiding to generate incorrectly typed parse trees, the trees are repaired by inserting coercion functions. The application of a coercion function is associated with a *coercion error* that is supposed to model the 'badness' of the particular coercion that is applied.

The system thus depends on the definition of a complete set of coercion functions. When all types can be coerced into each other using such functions, the coercion approach to typing can be defined as follows:

1. Set up a basic (single-typed) genetic programming system using all functions and terminals, without the set of coercion functions. No effort is made to ensure that the types match, other than making sure that a function has as many arguments as its arity (*i.e.* all binary functions are of the generic form $(T \to U \to V)$.);
2. Before evaluation, repair the tree by recursively matching actual types. If and when a type violation occurs:
    - insert the appropriate coercion function;
    - add the associated coercion error to the total coercion error for the program;
3. evaluate the repaired tree;
4. return the evaluation result together with the total *coercion error*.

The *coercion error* of a program is used as a second objective in a multi-objective search. Thus, rather than constraining the search to type-correct formulations only, all expressions can be inferred. Type correctness is viewed as a soft constraint, and the search is *guided* rather than *forced* to abide these constraints. It thus implements a *preferential bias* towards correct solutions rather then the *declarative bias* used by strongly typed genetic programming. At the end of a run, typically a *Pareto front* of non-dominated solutions is delivered; it is up to the user to judge which balance between the ability to solve the problem and type correctness is the most appropriate for the problem at hand.

Applicability of the approach The coercion approach to typing is suitable for any language where the types have a more-or-less meaningful coercion into each other. The language of units of measurement possesses this, but also a language that mixes integers, floats and booleans have such 'natural' translations.

If these 'natural' coercion functions can not be defined however, a strongly typed approach might be worth investigating. An example of this would be a language that allows string manipulations together with numeric operators. It is not clear how to coerce a string into a number in a sensible way or vice versa.

The coercion approach is related to repair-based algorithms that are used in runtime typed genetic programming systems (Yu and Bentley, 1998). There is one crucial difference: repair based algorithms do not in general use the effort that is needed in repairing expressions to guide the search. There is therefore no selection pressure towards finding expressions that do not need repair. Coerced genetic programming does provide such pressure. It will tend to avoid expressions that need excessive amounts of repair.

5.1.1 Calculating the Coercion Error for the uom system

For the **uom** system a single coercion function that only passes its argument can be defined:

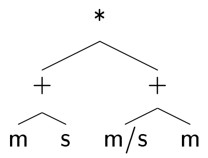
$$\begin{aligned} \text{coerce :} & \quad ([x,y] \rightarrow [u,v]) \\ \text{coercion-error} &= |u-x| + |v-y| \end{aligned}$$

which states that the coercion function can transform a type stated in a **uom** into a type within any other **uom** . This is equivalent to multiplying the input type with a constant of magnitude unity and **uom** $[u-x, v-y]$. The coercion error is accumulated through the expression and is used as an additional objective. In previous work (Keijzer and Babovic, 1999) the coercion error has also been called *goodness-of-dimension*. The goal of the coercion approach within the **uom** system is to find a trade-off between dimensionally correct formulations and well-fitted formulations. The coercion approach provides a graceful degradation when no correct formulations that fit the data well can be found.

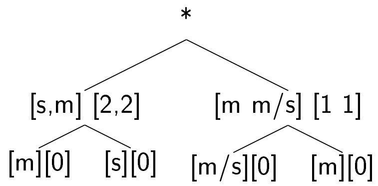
As this coercion function does not calculate anything — it simply returns the value of its arguments — no actual manipulations to the expression are necessary. While calculating the *coercion error*, the algorithm considers coercions only at the following nodes:

- At the root node: when the **uom** of the expression differs from the desired **uom**;
- At addition and subtraction nodes: when the two arguments differ, one argument is coerced into the **uom** of the other argument;
- At transcendental nodes: when the argument differs from the dimensionless **uom**, a coercion takes place.

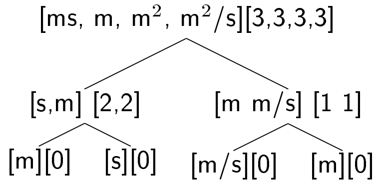
The algorithm for calculating the coercion error is recursively applied at all possible coercion points and selects that set of coercions that gives the smallest coercion error. The addition and subtraction nodes are constrained to coerce the **uom** of one argument into the other and not both to some third potentially more optimal **uom** . Binary functions such as multiplication and division propagate the constraints, and thus calculate the product of possible coercions. As an example: consider a tree using a length measurement, a time measurement and a velocity measurement in the following way.



This expression is clearly incorrect, it adds lengths to time measurements and lengths to velocities. To calculate the minimal coercion error, two arrays of values are maintained: one to store the coerced units of the tree, the second to store the coercion error. The terminals are initialized with arrays storing the input dimensions and a coercion error of 0. Calculating these arrays at the addition node will result in the following structure:



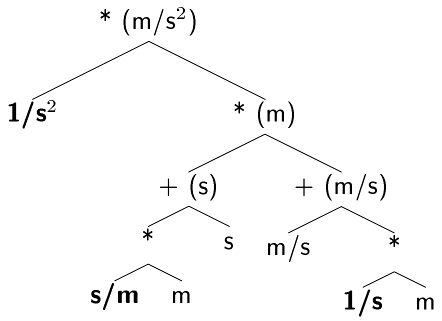
For the first addition branch, the information contained in the arrays [s,m][2,2], indicates that to obtain an expression stated in seconds, the first argument of the function (the length measurement) needs to be coerced. This coercion involves an effort (the coercion error) of 2: the length measurement should be multiplied with a constant stated in units of time per length to obtain a time measurement. The other option is to coerce the variable in unit time to a variable in units of length. To obtain such a length measurement, it needs to be multiplied with a constant in velocity units. For the second addition branch a similar calculation is made. Because there the arguments to the function only differ in the time dimension, the coercion error is 1. The multiplication node distributes the tentative outputs of the addition nodes by considering all possible output units. Because no coercion errors can be made at this point, it will simply add the associated coercion errors of its arguments. The calculation will then look like:



The possible outputs of this tree is thus one of [ms, m, m^2, m^2/s]. Suppose that the target **uom** for this problem is an acceleration. Coercing the arrays of possible outputs to an acceleration will result in:

$$[\mathrm{m/s^2},\ \mathrm{m/s^2},\ \mathrm{m/s^2},\ \mathrm{m/s^2}]\ [6,5,6,5]$$

Then, the minimal coercion error for this expression is the second or the fourth series of coercions, corresponding with a coercion error of 5. Considering the second series of coercions will produce the correct expression:



where the coercions that are (implicitly) applied are shown in boldface. Although in principle the use of binary functions such as multiplication and division can involve an exponential effort in calculating the coercion error of expressions, in practice (due to the pressure on minimizing coercions) the computation is feasible.

The coercion error is then used as a second objective that is to be minimized in the search. This coerced genetic programming system applied to problems involving units of measurement is called *Dimensionally Aware Genetic Programming* (DAGP). The name indicates that even though the system uses the dimensions in the data, it is only 'aware' of them, not forced to abide them at all cost. The primary motivation for the definition of this DAGP was the suspicion that for many practical problems not all relevant data would be measured and that this data is not always stated in the optimal units for inducing expressions (Section 4.2). Making the system aware of the **uom** in the problem description, rather than rigorously abiding them is thought to provide a more robust system than a strongly typed approach.

5.1.2 Wrapping

Often with genetic programming, methods can be devised that enlarge the solution space by making use of wrappers. Section 2.2.4 introduced wrappers for regression and classification. When inducing fully dimensioned empirical equations, another opportunity for wrapping the output arises.

In the system described above, the coercion error made at the output level (the root node of the tree) was included in the overall coercion error of the expression. An expression that is dimensionally consistent, but produces an output in the wrong units will thus have a non-zero coercion error. By relaxing this constraint to allow any output units, the solution space is again enlarged. The wrapper that will be used then takes the form of a multiplication by a constant, stated in such units that the overall output is stated in the correct units. Such dimensioned constants at the output level are part of standard scientific practice and produced normalizing constants such as Chezy's roughness coefficient (stated in $m^{0.5}/s^2$).

5.2 Example: Sediment Transport

The sediment transport problem was more fully described in Section 3.1. The terminal set is presented here again in Table 5.1, and the function set consists of the usual:

$$F = \{\text{plus/2,times/2,div/2,minus/2,sqrt/2}\}$$

Name	**uom**	description
ν	m^2/s	kinematic viscosity
u_f	m/s	sheer velocity
u'_f	m/s	sheer velocity related to skin friction
w_s	m/s	settling velocity
d_{50}	m	median grain diameter
g	$9.81m/s^2$	gravity acceleration
c_b	dimensionless	concentration of suspended sediment

Table 5.1: Dimensioned terminal set for the sediment transport problem.

Optimizing on the coercion error defined in Section 5.1.1, and on the normalized root mean squared error (NRMS), the typical result of a dimensionally aware genetic programming run is a front of non-dominated solutions that balance between accuracy on the training data and coercion error. Such a front is depicted in Figure 5.1. This figure is typical in that there exists a trade-off between the error on the data and the coercion error. This is not at all obvious as the doctrine of dimensional analysis in science seems to suggest that dimensionally incorrect formulations are expected to be wrong. These equations might be 'wrong' when looking at the dimensions only, but they do succeed in modelling the data well.

The reason for dimensionally incorrect expressions to evolve and have better accuracy than the dimensionally correct expressions has its origins in two separate reasons. Firstly, there are simply more dimensionally incorrect expression than dimensionally correct ones. The number of mathematical relations that can be modelled with an incorrect expression is larger then that of dimensionally correct expressions, as the space of incorrect expressions contains *all* arithmetical functions using the functions and variables. Secondly the data is collected and measured through empirical means: not all relevant phenomena can be measured in such a process and it is not guaranteed that the space of dimensionally correct models contains a solution.

With this trade-off explicitly modelled in the front of non-dominated solutions, the user's judgement enters the equation. The difference in error between the best dimensionally correct equation and the best equation fitted on the data in Figure 5.1 is sufficiently large to examine some other equations. In this particular case it would seem wise to also examine the formulations that have a coercion error close to 0.5 and a NRMS value close to 0.44, and contrast them with the dimensionally correct expression that evolved in the same run. A coercion error of 0.5 is fairly low: it can for instance be caused by an addition of a length measurement with a measurement stated in the square root of length as the only violation of the constraints. Whether this inconsistency weights up against the level of improvement is neccissarily a subjective choice.

Performing many independent runs results in a set of fronts of non-dominated solutions. From these again a front can be formed (see Figure 5.2). This is ultimately the set the user has to choose from. Also here a trade-off between performance

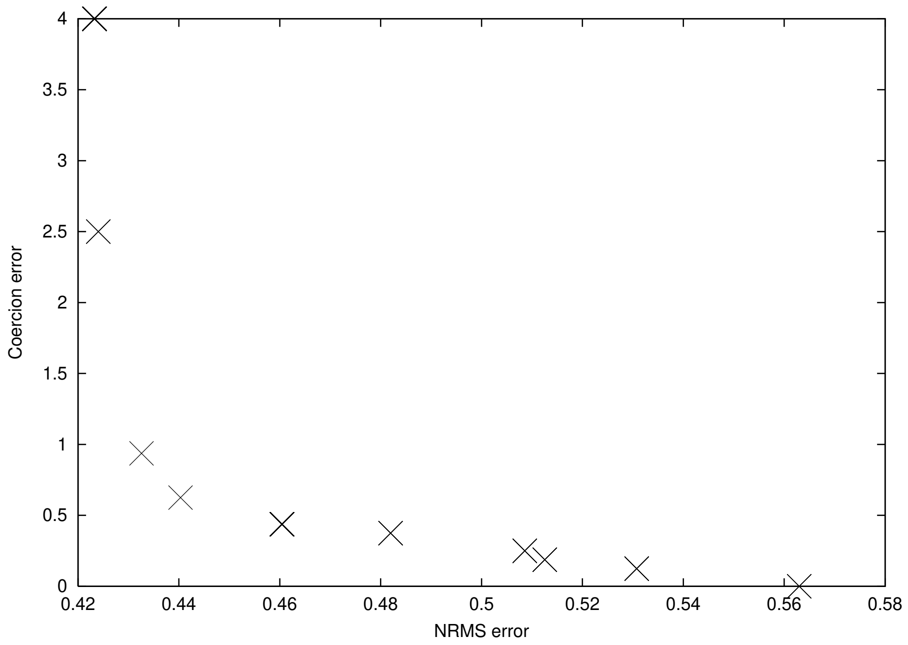


Figure 5.1: A typical front of non-dominated solutions produced in a single run of the dimensionally aware system. Almost invariably, a balance between goodness-of-fit and coercion error exists in the front of solutions.

on the data and dimensional correctness can be observed, careful examination of the formulae can provide additional insight into which units need to be violated to obtain a better fit on the data.

Interpretability Using a version of dimensionally aware genetic programming, in previous work (Keijzer and Babovic, 2000b) the following equation was induced for this problem.

$$c_b \approx 1.121 \times 10^{-5} \left(1 + 100 \frac{u'_f w_s}{g d_{50}}\right) \frac{u'_f - w_s}{u'_f + u_f}$$

This formula is dimensional correct, and furthermore it uses the most relevant physical properties in the relevant context. For example, the dimensionless term $\frac{u'_f w_s}{g d_{50}}$ is effectively a ratio of shear and gravitational forces. Shear forces are represented by u'_f, 'responsible' for elevating sediment particles into the stream, while the gravitational term $\frac{g d_{50}}{w_s}$ is 'responsible' for settling the particles. The remaining group $\frac{u'_f - w_s}{u'_f + u_f}$ is a ratio of resultant energy near the bed and of the total available energy in the flow transporting the particles.

The formula thus introduces two dimensionless terms, each being relevant to the problem. Three sources of information lead to the induction of the expression:

- the input output relation present in the data;

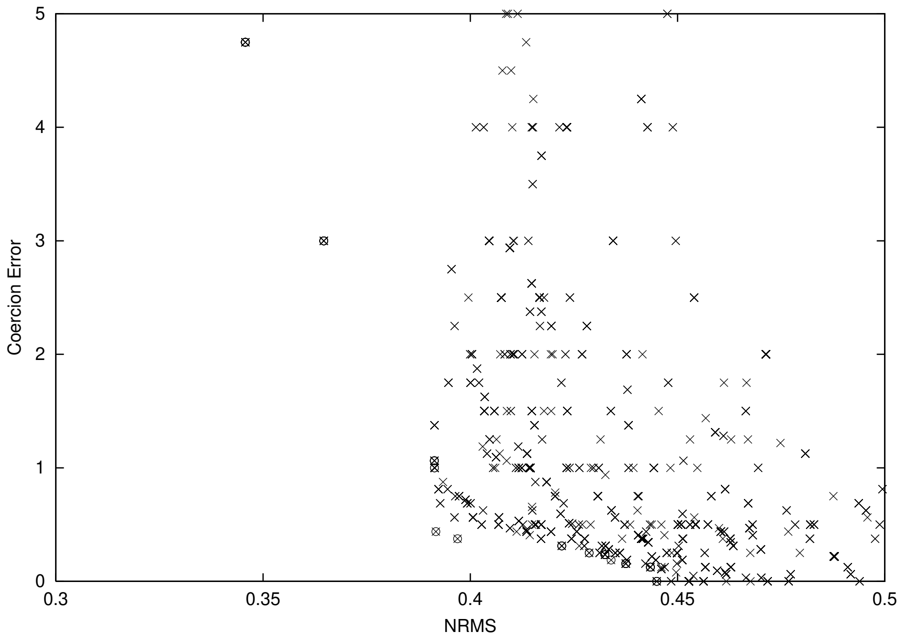


Figure 5.2: Overview of all fronts of non-dominated solutions produced by 50 independent runs of a dimensionally aware genetic programming system. The front of this set is depicted with additional circles.

- the units of measurement describing the dependent and independent variables in the problem;
- the user that selected this expression, symbolically manipulated it and tried to interpret it.

The first two sources are automated, while the third step critically depends on a user that tries to distill meaning out of the proposed relationship. The explicit use of the units of measurement helped in finding a link between the expression and the physical world. Several of such tentative relationships have been proposed above.

5.3 Summary

The method of using coercion rather than strong typing was first introduced in (Keijzer and Babovic, 1999), where it was applied to the problem of inducing expressions in the language of units of measurement. The particular combination of *typing as coercion* on problems involving units of measurement is called 'Dimensionally Aware GP', abbreviated to DAGP. The details of the calculation of the coercion error in (Keijzer and Babovic, 1999) are slightly different from those presented here.

Even though DAGP is capable of optimizing well, it has a drawback in that it relies on a multi-objective search strategy to balance fitting capability and coercion error. The computational complexity of this multi-objective search strategy (NSGA-II) is quadratic in the population size. For practical applications, this limits the population size that can be used. The specific calculation of the coercion error used here is in the worst case exponential in the depth of the tree. In practice upper bounds on the number of coercions that are maintained can be employed: this is not likely to have adverse effects, as a tree with a large number of potential coercions, will have a very large coercion error. The pressure on minimizing this coercion error helps in avoiding such large computations, but a maximum number can be easily set to cull expressions with an excessive amount of possible coercions.

To investigate the capabilities of DAGP in contrast with a dimensionally correct approach, an implementation of a strongly typed genetic programming is needed. As was shown in Section 4.4, systems that can handle only context-free constraints or that use typing through inheritance are not capable of expressing the language of units of measurement in full generality. A form of parametric polymorphism that allows explicit type calculations is needed.

To achieve this a new genetic programming system is defined. This is called an Adaptive Logic Programming system and will be introduced in Chapter 6. It is based on a developmental genetic programming system called Grammatical Evolution (O'Neill and Ryan, 2001), but is extended to handle arbitrary logic programs instead of context-free grammars only. Section 7.1.3 will contrast the untyped crossover used in ALP, with a typed crossover on problems involving units of measurement.

The experiment performed here is anecdotal in nature. It highlights some issues that arise when using the DAGP method of inducing dimensioned equations, in particular the Pareto front that is produced by this method. Although it is easier to use a system that produces a single best answer, it is thought that particularly in an exploratory endeavour such as scientific discovery the many alternatives that are explicitly delivered makes the system more useful for the scientist using it.

With the definition of this strongly typed genetic programming system in Chapter 6 and its scope in Chapter 7, it is then finally possible to compare symbolic regression, dimensionally aware genetic programming and strongly typed genetic programming on the problem of finding empirical equations on data. This comparison involving four real-world unsolved problems can be found in Chapter 8.

Chapter 6

An Adaptive Logic Programming System

Logic Programming makes a rigorous distinction between the declarative aspect of a computer program and the procedural part (Burke and Foxley, 1996). The declarative part defines the meaning of the program: the set of all facts that can be deduced. The procedural part aims at derives these facts.

The programming language Prolog is a concrete implementation for the Logic Programming paradigm, where the procedural aspect is implemented using a depth-first search-strategy through the rules (clauses) defined by a logic program (Sterling and Shapiro, 1994).

Due to its declarative nature, logic programming is very suitable for defining *computer* languages and constraints on them. Such a logic program then consists of a definition of all valid computer programs. In effect, the logic program defines both a parser and a generator for the language. The language can take the form of simple algebraic expressions; a robot steering language; constrained languages such as algebraic expression in the language of physical units of measurements and matrix algebra; as well as logic programs themselves. If the latter is the case, one normally speaks of Inductive Logic Programming (ILP) (Muggleton and Raedt, 1994).

When such a logic program is run using the Prolog search strategy, it will enumerate all possible computer programs in the domain defined by the logic program. When the search is for that particular computer program that performs best on some problem, and the number of possible programs is large, such an enumeration is not a viable search strategy.

An alternative for the depth-first search strategy of Prolog is examined here. A variable length genetic algorithm is used to specify the choice to make at each choice-point in the derivation of a query. This hybrid of a variable length genetic algorithm operating on logic programs is given the name Adaptive Logic Programming.

$$\underbrace{\overbrace{\text{pred(functor(Var1, Var2))}}^{\text{head}} : - \overbrace{\underbrace{\text{pred(Var1)}}_{\text{literal}} , \underbrace{\text{pred(Var2)}}_{\text{literal}}}^{\text{body}} .}_{\text{clause}}$$

Figure 6.1: The structure of a clause in a logic program.

6.1 Logic Programming

The basic construct in logics programs is a term. A *term* is a constant, a variable or a compound term. Constants denote particular elements such as integers, floating point values and atoms, while variables denote a single but unspecified element. The symbol for an atom can be any sequence of characters. It can be quoted to avoid confusion with other symbols (such as variables). Symbols for variables are distinguished by beginning with an uppercase letter or an underscore.

A *compound term* comprises a functor and a sequence of one or more terms called *arguments*. A functor is characterized by its *name*, which is an atom, and its *arity* or number of arguments. Constants are considered functors of arity 0. Syntactically functors have the form $f(t_1, t_2, \ldots, t_n)$ where the functor has the name f and is of arity n. The t_i's are the arguments. A functor f of arity n is denoted f/n. Functors with the same name but different arities are distinct. Terms are *ground* if they contain no variables; otherwise they are *non-ground*. *Goals* are atoms or compound terms, and are generally non-ground.

A logic program consists of clauses consisting of a head and a body. See Figure 6.1 for the structure of a clause. Clauses themselves can be thought of as compound terms in their own right, they are characterized by their principal functor `:-/2`. The head and the body are the two arguments for this functor. The terms occurring as principal functors in the body are called literals, to emphasize that they are literally used i.e., they are evaluated. A clause with an empty body is called a *fact*. The term *predicate* is reserved for a set of clauses that share the same functor (name and arity) in the head of the clause. Finally a *logic program* is defined as a set of such predicates.

With all the definitions and notational conventions in place, consider the logic program

```
sym(x).
sym(y).
sym(X + Y) :-  sym(X), sym(Y).
sym(X * Y) :-  sym(X), sym(Y).
```

which recursively defines the predicate `sym/1`. The derivation functor `:-/2` should be read as the implication sign $\leftarrow$. This program declares the complete and infinite set of legal expressions containing the atoms `x` and `y` and the functions of addition `+/2` and multiplication `*/2`. This logic program is equivalent with the context free grammar:

```
<sym> ::= x.
<sym> ::= y.
<sym> ::= <sym> + <sym>.
<sym> ::= <sym> * <sym>.
```

Logic programming has its roots in predicate logic. Clauses are universally quantified over the variables. The third clause in the program above can be translated in predicate logic as

$$\forall X, Y : \mathsf{sym}(X + Y) \leftarrow \mathsf{sym}(X) \wedge \mathsf{sym}(Y) \tag{6.1}$$

Thus: `X+Y` is a `sym` if `X` and `Y` are `syms`. This is the declarative reading of the clause. The procedural reading would be: to show that `sym(X+Y)` is valid, show that both `sym(X)` and `sym(Y)` are valid. In contrast with the program, a query consists of a term where the variables are existentially quantified. For example, the query

```
?- sym(X).
```

can be interpreted as the inquiry $\exists X$: $\mathsf{sym}(X)$ i.e., is there such an X? When running this query in Prolog it produces the following sequence of solutions:

```
X = x;
X = y;
X = x + x;
X = x + y;
X = x + (x + x);
X = x + (x + y);
X = x + (x + (x + x));
...
```

From this sequence, the general operation of the depth first clause selection in Prolog can be inferred. It first examines the first clause of the program: `sym(x).` Binding the variable `X` to the atom `x` gives the first instance of the sequence. A binding such as this is usually described in a *substitution* format: `[x/X]`. When the user asks for the next solution, the system backtracks: a flag gets set at this clause, the binding of `X` is undone and Prolog will examine the next clause: `sym(y).` This will result in the substitution `[y/X]`. Backtracking for a second time involves substituting `X` with `X1 + X2` (Prolog will provide fresh variables wherever a conflict might arise). This is denoted as: `[(X1 + X2)/X]`. The goal stack is updated with two new goals: `sym(X1)` and `sym(X2)`. The Prolog engine will now try to resolve these two goals, in the first instance resulting in the bindings `[x/X1]` and `[x/X2]`. The full set of bindings will then be: `[x/X2] [x/X1] [(X1 + X2)/X]` , which can be simplified to `[(x+x)/X]`. The return value will thus be `X = x + x`.

Extrapolating this sequence it is easy to see that without bounds on the depth or size of the derivation, the depth-first clause selection strategy employed in Prolog will never generate an expression that contains the multiplication character. If such limits are employed, Prolog will eventually generate such expressions, though this can take a long time.

A logic program as such does not define how to obtain solutions, it simply defines all possible expressions of this simple computer language. One of the many interesting features of logic programming is that there is no strict definition of input and output. The same program can be used both for generating expressions as well as for parsing expression. When running the query `?- sym(x + x * y)` in Prolog, the program will return with the answer `yes`, indicating that indeed, the expression `x + x * y` is a member of the set `sym`. The parsing and generating parts can be mixed: the query `sym(X + X*Y)` would enumerate all possible bindings for the variables `X` and `Y`.

Although in this example program input and output can be mixed, not all predicates can be written that way. This leads to the definition of the *mode* of the variables in a predicate. A variable occurring in the head of a clause can be *input*, *output*, or both. The mode is strictly speaking not a part of the Prolog language, though several variants have been defined to use the mode of predicates to produce more efficient code. In some logic programs that are used to generate sentences in Chapter 7, the notion of mode will be used to write more efficient programs.

Logic programming is a convenient paradigm for specifying languages and constraints. A predicate can have several arguments that can be used as *attributes*. These attributes can be used to constrain the search space. For example, the logic program and query

```
sym(x,1).
sym(y,1).
sym(X+Y,S) :-
   sym(X,S1), sym(Y,S2), S is S1+S2+1.
sym(X*Y,S) :-
   sym(X,S1), sym(Y,S2), S is S1+S2+1.

?-sym(X, S), S<5.
```

Program 6.1.1: Logic program defining a set of expressions together with their size.

specifies all expressions of size smaller than 5. The descriptive power of a logic program, makes it an ideal candidate for implementing attribute logic and constraint logic programming. It is this convenient representation of data, program structures and constraints that the genetic algorithm will try to exploit in this work.

Formally, a Logic Programming system is defined by Selected Literal Definite clause resolution (or SLD-resolution for short), and an *oracle* function that selects the next clause or the next literal. This *oracle* function is in Prolog implemented as:

- Select first clause
- Select first literal
- Backtrack on failure

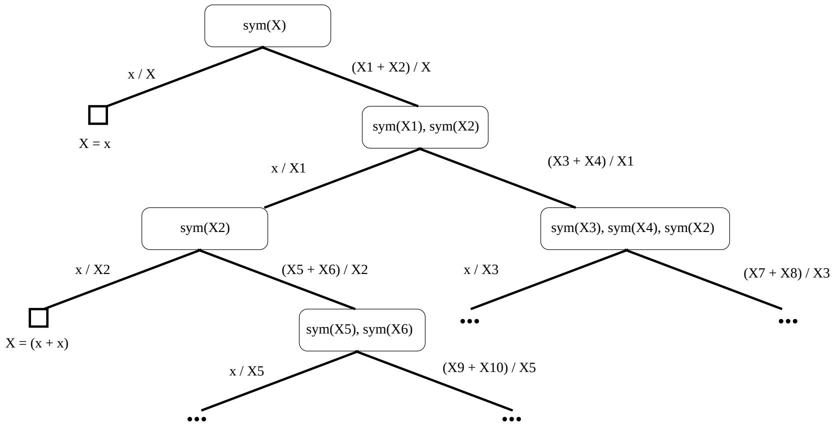


Figure 6.2: The search tree spanned by the logic program containing the clauses `sym(x)` and `sym(X+Y) :- sym(X), sym(Y)` Prolog will always choose the leftmost branch first.

Prolog thus tries to enumerate the entire domain with a depth first strategy, but it can get trapped in an infinite derivation. Figure 6.2 presents the search tree that is spanned by a simple logic program containing the fact x/0, and the function +/2. In this case all the solutions are present on the left side of the tree, thus Prolog can enumerate them. Changing the order of the clauses would transfer all the solutions to the right side of the tree, and without a depth limit, Prolog would not be able to derive a single instance of the set.

Due to the non-deterministic definition of the expressions, Logic Programming is a convenient paradigm to define constrained expressions. Parsing expressions and generating expressions can in principle be done with the same program. The goal of this approach is to generate expressions from some constrained set of computer programs. In particular the goal is to generate that computer program that performs best on some *objective function*. Prolog is capable of enumerating all computer programs given their definition in a logic program. When the number of possible solutions grows, this enumeration is not a viable search strategy, especially since the ordering of the clauses in the program determine the enumeration order. In the examples above, expressions containing a multiplication operator can only be generated after all valid expressions containing the other operators are generated.

To realistically search in the space of expressions defined by a set of predicates, the Adaptive Logic Programming system is introduced. It replaces the *first clause* rule in Prolog with a string of choices that represents an arbitrary path through the search tree. A variable length genetic algorithm is used to search this space of paths through the logic program.

6.2 An Adaptive Logic Programming System

Grammatical Evolution (O'Neill and Ryan, 2001) aims at inducing arbitrary computer programs based on a context-free specification of the language. It employs a variable length integer representation that specifies a sequence of choices made in

```
(0) sym(x).
(1) sym(y).
(2) sym(X + Y) :-  sym(X), sym(Y).
(3) sym(X * Y) :-  sym(X), sym(Y).
```

goal stack	substitutions	codon value
`?- sym(X).`		
`?- sym(X1), sym(X2).`	`[(X1 + X2)/X]`	2
`?- sym(X2).`	`[y/X1]`	1
`?- sym(X3), sym(X4).`	`[(X3 * X4)/X2]`	3
`?- sym(X4).`	`[x/X3]`	0
`?-`	`[y/X4]`	1

Table 6.1: Deriving a solution from a logic program by guiding the selection of clauses by a string of integers.

the context-free grammar. This sequence of choices represents a path through the context-free grammar and thus a sentence in the language the grammar defines. Due to the specific representation of a sequence of choices no type information needs to be maintained in the evolving strings. Furthermore, no custom mutation and crossover operators need to be designed: simple variable length string operators are used. In the GE-system, the choices are called *codons* to emphasize a biological analogy with triplets of nucleotides encoding a choice for a specific protein.

In the Adaptive Logic Programming system (ALP) we similarly use a sequence of codons as the base representation, but rather than choosing between the production rules of a context-free grammar, the codons are used to make a choice between the clauses in a logic program. The sequence of choices thus represents the clause-selection function operating together with SLD-resolution on the logic program. It defines a path through the search tree.

To give an example of the process, consider Program 6.1, and a sequence of choices $[2, 1, 3, 0, 1]$. The derivation of an instance is shown in Table 6.1. The initial query is `sym(X)`. By choosing clause 2 — the addition clause — two new goals are induced and a variable binding is made that introduces two new logic variables. At every step in the derivation, the first literal in the goal stack is selected. When a fact is selected, no new literals appear in the goal stack and a logic variable is bound to a ground term.

The result of this process is a list of variable substitutions:

`[(X1 + X2)/X][y/X1][(X3 * X4)/X2][x/X3][y/X4].`

which ultimately leads to the unification: `X = y + x*y`. This symbolic expression is produced in the form of a parse tree, not unlike the S-expressions used in LISP, and they can readily be accessed and evaluated[1]. The depth-first clause selection of Prolog is thus replaced by a guided selection where choices are drawn from the genotype. The genotype represents a path through the search tree (an example of

[1]The algebraic notation used to present these programs are supported by Prolog for notational convenience.

Selection	Prolog	Modification
Clause	First Found	**From Genotype**
Literal	**First Found**	From Genotype
On Failure	**Backtrack**	Restart

Table 6.2: Possible modifications to the selection function. The ALP system used here is identified in boldface.

such a search tree can be found in Figure 6.2). The first unresolved literal is chosen to be the first to derive. It is possible to replace this with guided selection as well, be it in the same string or in a separate string. Together with a choice whether to do backtracking or not, this leads to Table 6.2 which gives an overview of the parts of the Prolog engine that can be replaced. Table 6.2 thus defines a family of adaptive logic programming systems. Here we will focus on the system that corresponds with modified clause selection using backtracking. This was compared with a setup that did not employ backtracking, and it was shown that for more constrained programs, backtracking is indeed helpful (Keijzer et al., 2001a).

There are some practical problems associated with replacing literal selection. In many applications, a logic program consists of a mix of non-deterministic predicates (such as the `sym/1` and `sym/2` predicates above) and deterministic predicates (such as numerical assignment and comparison operators). The deterministic predicates often assume some variables to be bound to ground terms, evaluating them out of order would lead to runtime errors. The system studied here, which uses backtracking, encapsulates the Prolog language as a special case: the string containing an (infinite) number of zeros is equivalent with running the program through Prolog.

A logic program is thus used as a formal specification of a set of parse trees, the sequence of choices is used to steer the search process to derive a parse tree, and a small external program is used to evaluate the parse trees. See Figure 6.3 for the typical flow of information. The scope of the system are then logic programs where there is an abundance of solutions that satisfy the constraints, which are subsequently evaluated for performance on a problem domain. In some circumstances, when the objective function can be efficiently evaluated in Prolog, the external program is not necessary.

6.2.1 Representation and the Mapping Process

Like Grammatical Evolution, the ALP system studied here is a member of the family of Developmental GP systems (DGP) (Banzhaf et al., 1998)(pp. 250-255). In DGP a distinction is made between the representation the variation operators act upon and the computer program that is encoded by this representation. Using a biological analogy, the internal representation is called the *genotype* and the computer program the *phenotype*. The mapping process to go from a genotype to a phenotype is then seen as a developmental process, hence the name developmental GP. In the case of ALP, the developmental process that is used to derive a computer program takes itself place in the context of a full-fledged programming environment: the execution of a logic program.

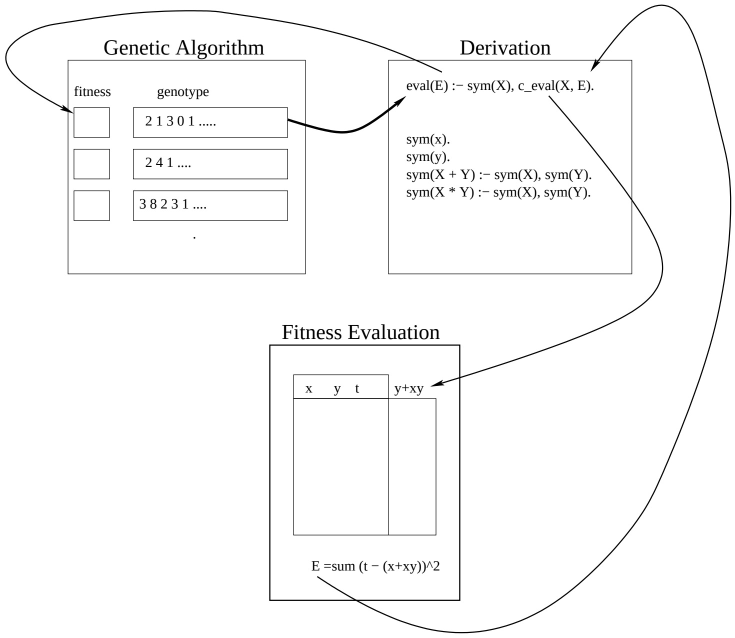


Figure 6.3: Overview of the ALP system: the sequence of choices is used in the derivation process to derive a specific instance for `sym(X)`, this instance is passed to the evaluation function. The calculated objective function value is returned to the genetic algorithm.

The representation that is used in the ALP system is a variable length string consisting of integers called codons — named such to emphasize another biological analogy — Given n predicates, each having the number of clauses $C = [c_1, c_2, \ldots, c_n]$, the value of the integers are restricted to lie in the range $[0, \prod_i^n c_i)$, the product appears for reasons given below. When there is a single predicate in the program, the integers decode simply to a clause in the predicate. As an example, consider the simple program:

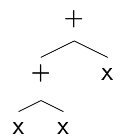
```
(0)  sym(x).
(1)  sym(X + Y) :- sym(X), sym(Y).
```

Because the number of clauses in this program equals 2 and there is only one predicate involved, the genotype is equivalent with a bitstring, where a 0 denotes the terminal `x` and 1 denotes the addition function. This program was already encountered above in Figure 6.2. It is now clear that the codons encode a choice for the path to take through the search tree: 0 encodes a choice for the left branch, 1 encodes a choice to the right. When using such a simple program that contains a single predicate without additional constraints, there is a one-to-one correspondence between the string of codons and the form of the resulting expression:

1	1	0	0	0	1	0
+	+	x	x	x	+	x

Table 6.3: Correspondence between a path through a logic program and the symbols that are induced.

the parse tree of the resulting program will then be:



This direct correspondence holds for any single-predicate program that does not introduce additional constraints on the expressions that are induced.

The two extra bits at the tail of the bitstring in Table 6.3 are unexpressed in the resulting program. They will be kept in the genotype as they might get expressed after a crossover or mutation event. It is quite possible that the program is not finished when there are no more choices left in the string. In that case, one of several mechanisms can be employed:

1. **destruction**: the string gets the worst possible performance value;
2. **repair**: the string is extended with random integers until it finds a solution;
3. **reuse**: the string gets wrapped and the reading restarts at the beginning until some maximum level of wrappings is reached (O'Neill and Ryan, 2001);
4. **Prolog**: the string is extended with an infinite number of zeros (i.e., the unfinished logic program is executed using Prolog).

From these choices the first one, destruction, is used here. Destructing the genotype will result in a *failure rate* for the algorithm. The destruction method is mainly chosen for its simplicity: the repair method would involve changing the genotype in the mapping process, and it would have the disadvantage that when considering more constrained programs a large runtime overhead can be induced. It has this in common with the Prolog method. This overhead occurs when the unfinished expression is located in a branch of the search tree that does not contain solutions, or only contains equations that are too long. Escaping such a branch involves examining all possible paths in that branch until the search depth is exceeded, backtracking would then be employed to find another branch.

The wrapping method has as a disadvantage that it is not guaranteed to finish. In particular, when using a simple program such as the one above, the wrapping method is guaranteed *not* to finish as the same unfinished tree would be used repeatedly.

There are thus three possible outcomes of the mapping process: *(i)* the string is completely mapped into an expression with no spare codons left, *(ii)* the string is mapped into an expression with spare codons left, or *(iii)* the string is mapped into a partially completed expression. The last outcome is considered a failure.

Properties of the encoding for simple programs.

Consider a simple (Koza-style) language consisting of t terminals $[x_1, \ldots, x_t]$, u unary functions $[h_1, \ldots, h_u]$, and b binary functions $[f_1, \ldots, f_b]$. The language of parse trees can then be modelled by the single predicate logic program:

```
expr(x1).
...
expr(xt)

expr(h1(X)) :-      expr(X).
...
expr(hu(X)) :-      expr(X).

expr(f1(X, Y)) :-   expr(X), expr(Y).
...
expr(fb(X, Y)) :-   expr(X), expr(Y).
```

Program 6.2.1: General program for performing 'Koza'-style genetic programming. Inclusion of higher arity functions is straightforward.

every codon in a string will now decode into a specific clause if it is used in the context of the query `?- expr(X).` Obviously, ternary and higher arity functions can be catered for as well.

As there is a one-to-one correspondence to an element in the string and the clause in the logic program it encodes for, the size of the goal stack during the generation of an expression can be written in terms of the codons of the string. Define the function:

$$\mathsf{b}(x) = \text{number of literals in the body of clause } x \text{ minus one}$$

thus:

$$\mathsf{b}(x) = \begin{cases} -1 & \text{if } x < t & : & \text{terminal} \\ 0 & \text{if } t \leq x < t + u & : & \text{unary} \\ 1 & \text{if } t + u \leq x & : & \text{binary} \end{cases} \tag{6.2}$$

Define the function g, the number of literals on the goal stack after k codons are used from string s as:

$$g(s,k) = g_0 + \sum_{i=1}^{k} \mathsf{b}(s_i) \tag{6.3}$$

Where g_0 is the initial state of the goal stack (usually 1). Then a string s of length N will decode to a complete expression when the following condition is satisfied:

$$\exists k \leq N : g(s,k) = 0 \tag{6.4}$$

The validity of this condition can be checked using the correspondence between the function g and the number of literals on the goal stack. Initially, there are g_0 literals on the goal stack. When a binary function is added, one literal is popped from the goal stack, and two are added. For a terminal, one literal is removed and for a unary function, one is removed and one is added. Finally, when there are no literals left i.e., the goal stack is empty, g equals zero and the query (and thus the expression) is finished.

In effect, using programs of this simple form, the string of integers is simply a *prefix* encoding of the parse trees under consideration, with the possibility of not fully specifying the complete parse tree.

Initialization and the Gambler's Ruin Model Given these logic programs, it is important to know what the probability is of generating a *legal* string, given that we randomly generate choices. This obviously depends on the proportion of variables, unary functions and binary functions that are present in the program. These are denoted here by t, u and b respectively, and the total number by $C = t + u + b$. Examining Equation 6.4 for string length 0, there is 1 unresolved literal, the original query. For each codon we add to the string, this number can increase, decrease or stay the same, depending on what value is drawn (Equation 6.2). If the number of unresolved literals drops to zero at any point, Equation 6.4 defines that the string will encode a legal expression regardless of the codons that appear after that point.

This situation is much like the situation of a gambler in a casino. The gambler has a starting fortune, and can place bets, that are won or lost. Given a fixed probability of winning a bet p and of losing a bet q, what is the probability that the gambler would be ruined? This is known as the gambler's ruin model.

Generating a string of random codons with a fixed probability is equivalent with successively placing bets; the pay-off function is given by Equation 6.2. A bet is lost when a terminal is drawn, while a bet is won when a binary function is drawn. Drawing a unary function does not change the fortune (i.e., the number of unresolved literals in the goal stack). The gambler starts with an initial fortune of $g_0 = 1$ unresolved literal, the initial goal. When the gambler runs out of literals, the fortune is lost and the gambler is ruined. In this case the gambler's ruin corresponds with the ALP system's gain: it successfully derived an expression.

When uniformly generating codons from the range $[0, C)$, the probabilities of winning respectively losing are: $p = b/C$ and $q = t/C$. It is well known that with a starting capital of g_0 and a stopping capital of T, the gambler goes home with capital T with probability

$$P_{\text{illegal}} = \begin{cases} \frac{(\frac{q}{p})^{g_0} - 1}{(\frac{q}{p})^{T} - 1} & \text{when } p \neq q \\ \frac{g_0}{T} & \text{when } p = q \end{cases} \tag{6.5}$$

In our case, the start capital is 1 unresolved literal, while the gambler never stops without any bounds on the size of the genotype. The probability of the tree not finishing is then

$$P_{\text{illegal}} = \lim_{T \to \inf} \frac{\frac{q}{p} - 1}{\frac{q}{p}^{T} - 1} = \begin{cases} 0 & \text{if } q > p \\ 1 - q/p & \text{if } q < p \end{cases} \tag{6.6}$$

and if $p = q$,

$$P_{\text{illegal}} = \lim_{T \to \inf} \frac{1}{T} = 0 \tag{6.7}$$

Thus if the probability of drawing a terminal is equal or larger than the probability of drawing a binary function, the probability of ending up with a string that encodes a legal parse tree will tend to 1. However, when there are more binary functions than terminals, the probability of obtaining a legal string will converge to q/p. Thus if no limits on the tree are employed only a fraction of the trees can ever be generated.

The expected change in the size of the goal stack after randomly selecting a clause is simply $p - q$, the goal stack is thus expected to grow when there are more binary functions than terminals. Without bounds on the size or depth of the trees, the expected size of the trees is thus infinite when $p >= q$, as it grows at every step. When $p < q$, the expected size is the point where we expect the goal stack to be of size 0, this is simply $\frac{1}{q-p}$.

The process of creating a random tree and its influence on the size of the goal stack is depicted in Figure 6.4. The small graph depicts the size of the goal stack (the function $g(s, k)$ from Equation 6.3) with increasing string length for some random choices. As the graph of the size of the goal stack in these simple predicate programs contain all information about the shape of the parse tree that is produced, this graph will be called the **shape graph**. When and if the size of the goal stack drops to zero, the query is completed and a valid expression is obtained. The shape of the expression that results is depicted on the right of Figure 6.4.

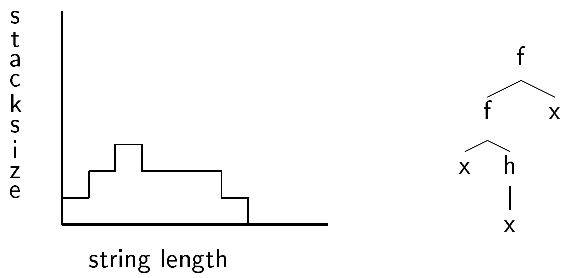


Figure 6.4: Creating a tree using a random walk. Depicted are the shape graph with the corresponding tree. The tree is created in a depth first manner.

By making the probabilities of selecting the clauses in the program dependent on the length of the string, in principal an expected shape graph can be enforced on an initial population.

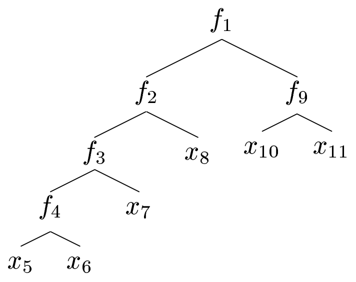
Ripple crossover It was stated above that to create a child, the crossover used in ALP grafts a randomly chosen suffix of one string upon a randomly chosen prefix of another, the points are chosen independently in the expressed part of the string. In the context of the programs defined in Table 6.2.1, this crossover has a very clear effect on the parse trees that are encoded by the strings. This effect was termed ripple crossover (Keijzer et al., 2001b). Although it resembles a variable length one-point crossover from genetic algorithms, the term *one-point crossover* has been used to describe a very different operator for genetic programming (Poli and Langdon, 1998). Therefore, the term ripple crossover will be used henceforth to designate the specific crossover used in the GE and ALP systems.

Consider the program:

```
(0)  expr(x).
(1)  expr(f(X,Y)) :-  expr(X), expr(Y).
```

The string of choices will be constructed from the two letter alphabet $[0, 1]$ with the mapping defined by the definition above. A string maps into a parse tree in the following way:

$[1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0] \rightarrow_{encodes}$



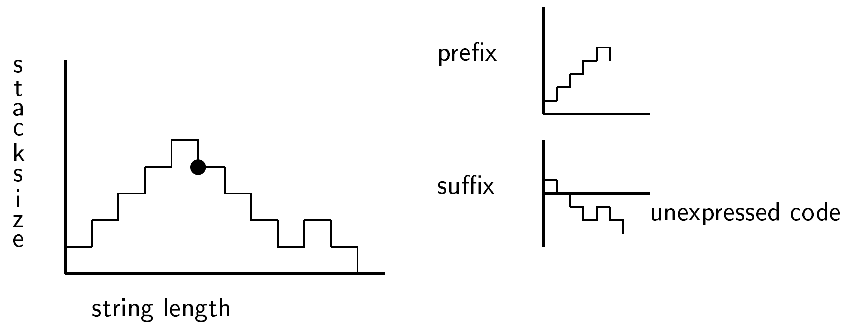
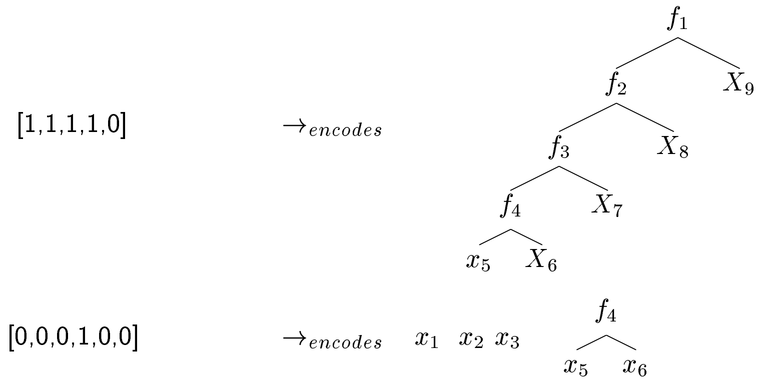


Figure 6.5: Splitting a tree into a prefix and suffix.

The symbols in the expression are subscripted with the location in the genotype that is used to encode them. Splitting the tree after the fifth position will result in an unfinished tree and a set of subtrees:



where the uppercase logic variables $X_6 \ldots X_9$ indicate that the expression is unfinished. The crossover point occurring after x_5 has the effect of removing all subtrees to the right of the point. Figure 6.5 depicts the splitting process in terms of the **shape graph**. Cutting the string thus results in cutting all the subtrees to the right of the string, leaving in this case 4 unfinished subtrees. The suffix of the string encodes these subtrees. When another string is spliced in a similar way, swapping the tails results in grafting missing subtrees from one tree upon the other tree. When the number of subtrees encoded by the tail is smaller than the number demanded by the head of the string, the resulting tree will encode an unfinished tree.

The example given here used a string that decodes to a full tree without any spare codons. In practice, there is often a tail of unexpressed code for any given string. If the string $[0, 0, 0, 1, 0, 0]$ from above would be used as a genotype, it would encode the expression x and would have three spare subtrees that can get expressed when needed. This is depicted in Figure 6.5 as well, where the suffix is used as a tree in its own right: the minimum value of the goal stack function $g(s, k)$ lies at -3: there are therefore three unexpressed subtrees present. The tail of unexpressed code can then function as a buffer of spare subtrees.

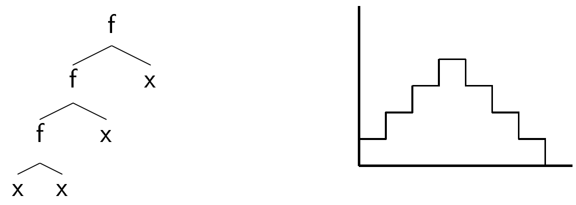
Thus, under the operation of ripple crossover, the genotype falls apart in two pieces: a prefix that encodes an unfinished parse tree, and a suffix that encodes a collection of subtrees.

A Bias induced by Ripple Crossover It is possible to show some bias that is induced by ripple crossover when there is no tail of unexpressed code. Equation 6.3 defined the number of goals in the goal stack for a string after k codons were translated, as:

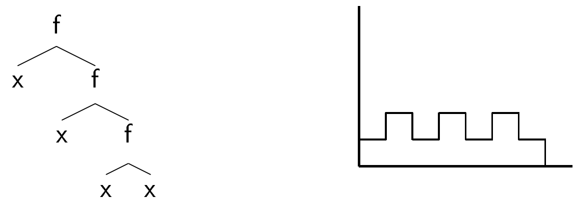
$$g(s,k) = g_0 + \sum_{i=1}^{k} b(s_i)$$

where the function b returns the number of literals in the body of the clause minus 1 and g_0 is the initial state of the goal stack. Ripple crossover selects crossover points at random in the string and subsequently splits the string in a prefix and a suffix. If the size of the goal stack for the prefix string is larger than the number of subtrees contained in the suffix string, the mapping process will not be able to produce a complete expression.

To clarify the relationship between the size of the goal stack (the shape graph defined by the function $g(s,k)$) and the shape of the encoded tree, two extreme cases are examined here: left-skewed and right-skewed binary trees. Consider a left skewed tree with corresponding shape graph:



The prefix encoding of this tree is $[f,f,f,x,x,x,x]$. Furthermore, consider the right skewed tree:



whose prefix encoding is $[f,x,f,x,f,x,x]$. In terms of the goal stack (and thus the shape graph), a left skewed tree needs a larger stack than a right skewed tree. When crossing these trees using ripple crossover, the shapes are crossed over as well (Figure 6.7).

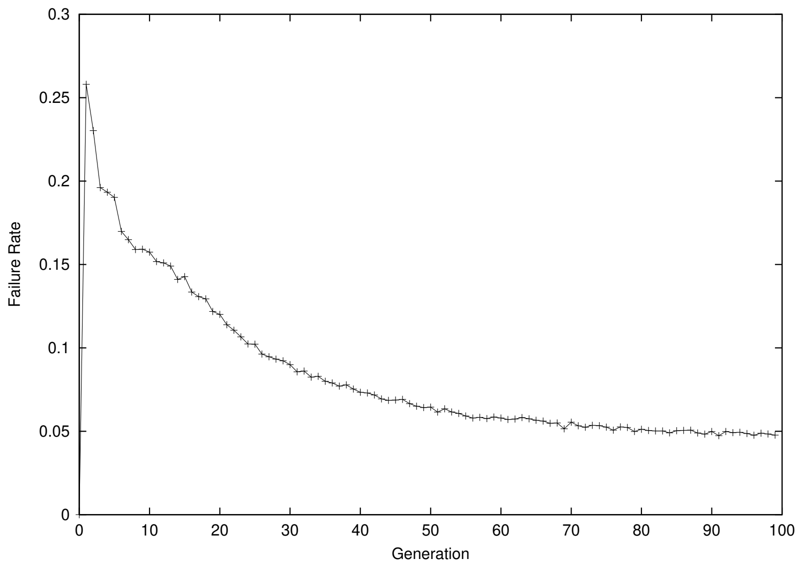


Figure 6.6: Typical failure rate with a ripple crossover that creates a single child.

The ripple crossover used in previous work (Keijzer et al., 2001b; Keijzer et al., 2001a), created one child only, and put it back in the population without regard to it being legal or not. This typically leads to a high failure rate in the beginning of the run. Figure 6.6 depicts the typical shape of the failure rate during a run. In the first generation, the failure rate rises sharply, depending on the program that is used lying between 1/4 to 1/3 of all crossovers. Subsequently, when a tail of unexpressed code forms, the failure rate drops, but as can be seen in the graph, even at generation 100, the failure rate is still significant.

In principle this could be remedied by *(i)* choosing the point for the suffix depending on the size of the goal stack at the point chosen for the prefix, or *(ii)* create two children, one using the first parent as prefix, the other using the second parent as the prefix. For the type of programs studied here, it would then be guaranteed to create at least one valid offspring because for two strings s_1 and s_2 with crossover points k_1 and k_2 it holds that $g(s_1, k_1) - g(s_2, k_2) \leq 0$, and/or $g(s_2, k_2) - g(s_1, k_1) \leq 0$. The equality holds when both are valid. Thus for two given crossover points, there is always an offspring that is valid.

It was chosen to use this second method as the first method is not guaranteed to succeed: the size of the goal stack for the first point can demand more subtrees than the second parent can deliver. This could again be fixed by choosing the points in an even more restricted fashion, but one of the purposes of ripple crossover is to have a simple, untyped and unrestricted crossover. Although the information about the size of the goal stack could be used to guide the crossover points, it was chosen to use the simple method of creating two offspring given two crossover points, putting the first one that succeeds in decoding to an expression back in the population.

Figure 6.7 shows the effect of this crossover in terms of the shape graph of the individuals. When one offspring is invalid, the other individual is necessarily valid

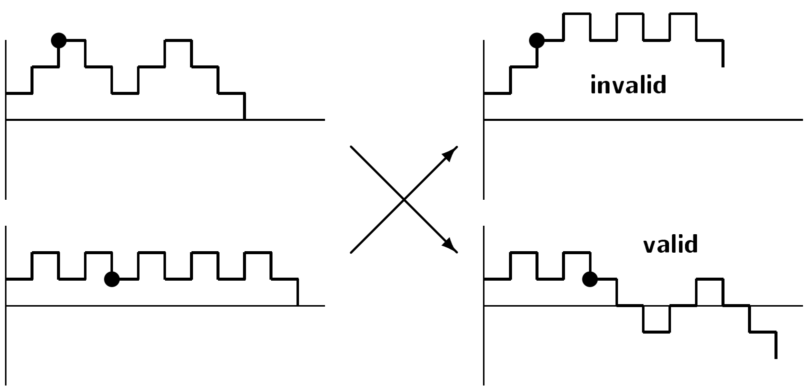


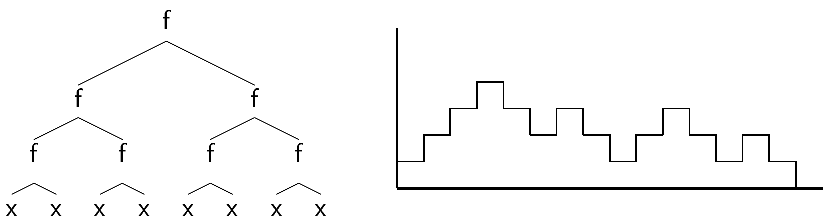
Figure 6.7: The effect of ripple crossover on the shape graph for two parents not containing any unexpressed code. The crossover points are indicated with a small circle. If one offspring is invalid, the other is necessarily valid and will have a tail of unexpressed code.

and will have a tail of unexpressed code. By selecting the valid offspring always, no invalid individuals can enter the population. A side-effect of the procedure is that the tail of unexpressed code quickly forms and that it will consist of *code that has undergone selection*. This is an important difference from generating a tail at random, as there is then for example no guarantee that the tail encodes valid subtrees.

Using this procedure will guarantee legal offspring when using single predicate programs without additional constraints, but the bias towards right-skewed trees used as the prefix remains. However, left-skewed trees are very suitable to be used as suffixes, simply because they encode for more subtrees then their right-skewed counterparts. They will initially become unexpressed, but are stored in the population and due to ripple crossover and mutation can soon become expressed again.

In the presence of a symmetric function set, this bias towards right-skewed expressions has no repercussions for the ability to express solutions. In effect, there is a memory advantage in preferring right-skewed expressions as the size of the goal stack is kept at low values. The effect of ripple crossover on asymmetric function sets is a left for future study.

For the sake of completeness, a balanced tree with corresponding shape graph is depicted below.



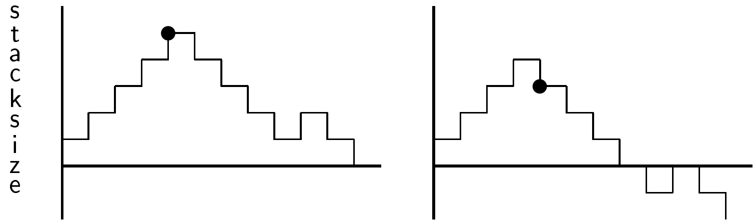
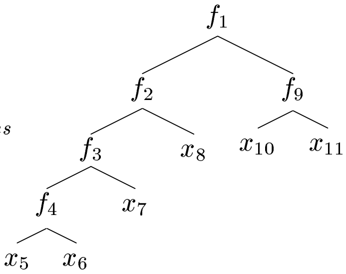


Figure 6.8: The effect of a mutation on the goal stack. The mutation point is indicated with a circle.

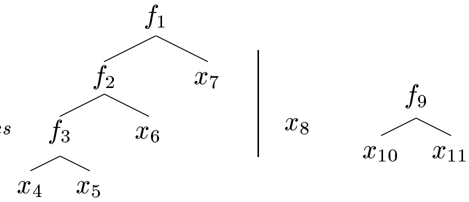
Ripple Mutation Many mutations can be defined, but here we focus on a single point mutation. A point is chosen in the string and replaced with a random integer from the admissible range. This mutation also has a rippling effect on the expression tree that is encoded in the string. Consider again the string and mapping:

$[1,1,1,1,0,0,0,0,1,0,0] \rightarrow_{encodes}$



Replacing the fourth element 1 with a 0, results in:

$[1,1,1,0,0,0,0|0,1,0,0] \rightarrow_{encodes}$



All variable to the right of the mutation point have shifted their location in the tree, and the last four codons have become unexpressed. If one were to mutate a terminal into a binary function however, parts of the tail of unexpressed code would get expressed again. In the absence of such a tail however, the resulting expression would be invalid. In the early generations when there is no or little unexpressed code, ripple mutation would thus be biased to sample expressions that are shorter than the parent. As it would create unexpressed code in that process, this effect will lessen in later stages of the run.

One of the characteristics of ripple crossover and ripple mutation is the disregard for the structure of the trees that are derived (i.e. the shape graph). Because of this changes propagate to the right of the tree, in particular the second and subsequent arguments of the root node are subject to change relatively easily. The ripple variation operators are then expected to be fairly destructive.

Multiple predicates and polymorphism.

Until now the programs that are considered used a single predicate without additional constraints, but multiple predicates are an obvious extension. Consider the program:

```
expr(x).
expr(X) :- bin_op(X, A1, A2), expr(A1), expr(A2).

bin_op(X + Y, X, Y).
bin_op(X * Y, X, Y).
```

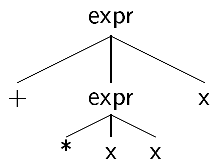
Here we have two predicates `expr/1` and `bin_op/2`, each having two clauses. There are several options for encoding such a program in a string of choices. For context-free grammars which translate directly to a logic program such as the one above, O'Neill and Ryan defined an upper bound for the codons in the string (usually 256) (O'Neill and Ryan, 2001) . Given a set of n predicates with a corresponding number of clauses $[c_1, \ldots, c_n]$ and given a predicate r, the mapping rule used is (O'Neill and Ryan, 2001):

$$\text{choice}(r) = \text{codon} \mod c_r \tag{6.8}$$

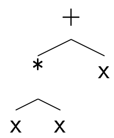
This *modulo* rule makes sure that the codon value is mapped into the interval $[0, c_r)$ and thus represents a valid choice for a clause. As the codons themselves are drawn from the interval $[0, 256)$, the mapping rule is degenerate: many codon values map to the same choice of rules. Unfortunately, in the case of the program considered here, this mapping rule introduces a linkage between the clauses of the different predicates. As we have two predicates, each having two clauses, the modulo rule will map all even codon values to the first clause and all odd values to the second clause, regardless of the predicates that are used. Above it was shown that the crossover and mutation operators can shift the location of the subtrees when using a single predicate. In the case of multiple predicates, this shift in location can also result in a shift in predicates: a codon value that previously encoded for a clause in one predicate can be re-interpreted to encode for a clause in another predicate. This property of this developmental genetic programming system is called *intrinsic polymorphism* (Keijzer et al., 2001b).

As Program 6.2.1 induces a fixed mapping for even and odd codon values, we can restrict our attention to bitstrings. In the context of the two predicate logic program, we can have the following mapping:

$[1,0,1,1,0,0,0] \rightarrow_{encodes}$

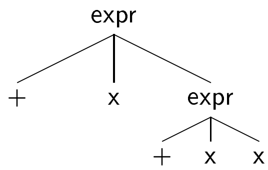


Where the name *expr* is used to denote the use of the recursive `expr/1` clause. The function symbols $+$ and $*$ have been put at the location where they are derived. The string of choices then corresponds again with a depth first traversal of the derivation tree. Due to Prolog's operator handling, the actual parse tree that is produced by the query will look like:



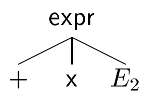
Mutating the third element of the string will result in:

$[1,0,0,1,0,0,0] \rightarrow_{encodes}$



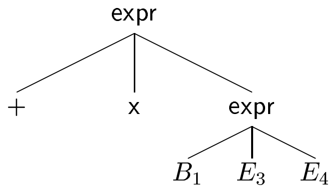
This is quite a significant change. This single mutation event changed the expression $((x*x)+x)$ into $(x+(x+x))$. It is instructive to inspect what happened. Changing the third element left the first two elements intact. The third element used to encode for the recursive `expr` rule, but is replaced by the non recursive rule, the terminal x. Because of this change, the fourth element of the string which used to encode for the multiplication clause of the string is re-interpreted as an encoding for the recursive `expr` clause. Similarly, the fifth element that used to encode for the terminal `expr` is changed into the addition operator as it is evaluated in the context of the predicate `bin_op`.

Thus by changing a single element in the string, the string is again split into a prefix and a suffix. This time the suffix is reinterpreted. Changing the third element from *expr* to *x* changes the derivation tree and the suffix that will be reinterpreted to:



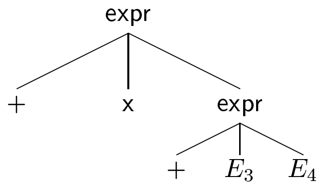
[1 0 0 0]

where the variable labelled E_2 indicates that the derivation is not complete until a subtree starting with the `expr/1` predicate is produced. The codon that used to encode for the second clause of the `bin_op` predicate (multiplication) will now be re-interpreted as the second clause of the `expr` predicate, thus:



[0 0 0]

The next codon to insert into the derivation tree used to encode for the first clause in the `expr` predicate, but the search process has reached the `bin_op` predicate at this point in the derivation. Hence the first clause in this predicate will be used, the addition.



[0 0]

The rest of the derivation process will not demand any re-interpretations, thus the remaining codons can be simply inserted as terminating x's.

In this case, the modulo rule thus defines a fixed transition when the codons are interpreted in the context of different predicates. The terminal x implicitly encodes for the addition operator $+$ and vice versa, while the recursive `expr` clause is linked to the multiplication operator in the `bin_op` predicate. This linkage depends on the ordering of the clauses as well as the number of clauses per predicate[2].

This linking between clauses of different predicates in the context of the variation operators introduces a bias in the search process. This bias is undesirable because it depends on the layout of the program and its impact on the search is not clear. In effect this means that the order in which the clauses are defined are expected to make a difference to the search efficiency.

To remove this bias the mapping rule is changed. Given again our set of clauses for n predicates $[c_1, \ldots, c_n]$, the codon values are now taken from the interval $[0, \prod_i^n c_i)$. The mapping rule is subsequently changed to:

$$\text{choice}(r) = \frac{\text{codon}}{\prod_{i=1}^{r-1} c_i} \mod c_r \tag{6.9}$$

This rule is simply the standard method for mapping multi-dimensional matrices into a contiguous array of values. The predicates form the dimensions, with the number of clauses as the coordinates. With this rule, every legal codon value encodes a unique set of clauses, one from each predicate. In the program discussed here,

[2]When using two predicates having 2 and 3 clauses, together with codons drawn from $[0, 6)$ all transitions are possible.

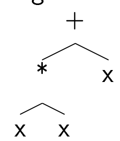
there are two predicates, each having two clauses. The codons are thus drawn from $[0, 4)$. Each codon value now encodes for a unique set of clauses:

codon	`expr`	`bin_op`	`expr`	`bin_op`
0	$0/1 = 0 \mod 2$	$0/2 = 0 \mod 2$	x	+
1	$1/1 = 1 \mod 2$	$1/2 = 0 \mod 2$	expr	+
2	$2/1 = 0 \mod 2$	$2/2 = 1 \mod 2$	x	*
3	$3/1 = 1 \mod 2$	$3/2 = 1 \mod 2$	expr	*

By using this mapping rule, the representation in effect becomes *polyploid*: the single string of integers is isomorph with a set of n strings, each encoding the clause to choose when it is evaluated in the context of each predicate. The active element is determined by which predicate is active in that part of the execution of the logic program generating the expression. All other elements at the same location are recessive, mutations on them do not have an effect on the resulting expression.

For an example of this polyploidity, consider the following encoding:

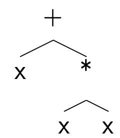
$$[1,0,3,3,2,2,0] = \left[\begin{array}{ll} \texttt{expr}: & [\mathbf{1},0,\mathbf{1},1,\mathbf{0},\mathbf{0},\mathbf{0}] \\ \texttt{bin\_op}: & [0,\mathbf{0},1,\mathbf{1},1,1,0] \end{array} \right] \rightarrow$$



where the elements in bold are the active elements of the set of strings. As in this particular program we have three terminating clauses and one recursive one, only the recursive clause determines how the rest of the string is interpreted.

If we again consider a mutation in the third element, several things can happen. If we mutate the integer 3 to the integers 0 or 2, the clause encoded for changes from the recursive `expr` (clause 2) to the terminal `expr` (clause 1), by virtue of the mapping rule defined in Equation 6.9. Given that the change happens be $3 \rightarrow 0$, the individual will ultimately decode to:

$$[1,0,0,3,2,2,0] = \left[\begin{array}{ll} \texttt{expr}: & [\mathbf{1},0,\mathbf{0},\mathbf{1},0,\mathbf{0},\mathbf{0}] \\ \texttt{bin\_op}: & [0,\mathbf{0},0,1,\mathbf{1},1,0] \end{array} \right] \rightarrow$$



Because the clauses belonging to different predicates are independently encoded, the integer on the fifth position now encodes both the terminal `expr` and the multiplication `bin_op`. When the integer 3 at the third position is changed to the integer 1 however, nothing changes directly in the phenotype, as the third element will still encode for the recursive `expr`. The mutation is thus neutral. Even though the coding is degenerate as many integer values decode to the same clause, there is no redundancy: if the codon would be used in a later generation in the context of the predicate `bin_op`, it would encode for the addition function rather than the multiplication.

Thus, every location in the genotype can encode for a unique clause for each predicate. This gives the genotype in principle the opportunity to encode many different trees, which tree is derived depends on the context it is used in. Whether this capability is helpful in search and optimization needs to be ascertained. What it does achieve is make the system independent of the order in which the clauses are defined by removing linkages between the order of clauses in different predicates.

Ripple crossover does not look at the names of the predicates to pick crossover points. It is thus quite common that it will select a prefix that ends in one predicate and a suffix that used to start with another predicate. This re-interpretation of genetic material will then happen quite often. It is left for future work to examine the worth of this *intrinsic polymorphism*.

Context-Sensitive Programs

A logic program is capable of using context-sensitive information. Consider Program 6.2.2 that calculates the bounds of an expression.

```
interval(x, 1, 10).
interval(y, -5, 5).

interval(X + Y, L, U) :-
       interval(X,Lx, Ux),
       interval(Y, Ly, Uy),
       L is Lx + Ly,
       U is Ux + Uy.

interval(-X, L, U) :-
       interval(X, Lx, Ux),
       L is -Ux,
       U is -Lx.

interval(X * Y, L, U) :-
       interval(X,Lx, Ux),
       interval(Y, Ly, Uy),
       B1 is Lx * Ly, B2 is Lx * Uy,
       B3 is Ux * Ly, B4 is Ux * Uy,
       L is min(min(B1, B2), min(B3,B4)),
       U is max(max(B1, B2), max(B3,B4)).

interval(1 / X, L, U) :-
       interval(X,Lx, Ux),
       catch(A is 1/Lx,_,fail),           % fail on math exceptions
       catch(B is 1/Ux,_,fail),           % idem
       sign(A) + sign(B) =\= 0,           % make sure the interval
                                          % does not contain zero
       L is min(A,B),
       U is max(A,B).

interval(sqrt(X), L, U) :-
       interval(X,Lx, Ux),
       Lx >= 0,
       L is sqrt(Lx), U is sqrt(Ux).
```

Program 6.2.2: A context-sensitive program that calculates the the numerical interval for an expression and only derives those expressions that are well-defined

This logic program is capable of generating expressions that are guaranteed to avoid a division by zero or taking the square root of a negative number, given the specified range of the terminals. It will furthermore calculate bounds on the range of the expression. In this program, the lower and upper bound `L` and `U` are used as *attributes* in the program and their values are used in the clause that handles division to make sure that no expression will be derived that can theoretically divide by zero. This is done by checking whether the bounds of the argument contain zero. Likewise the square root function is protected by checking whether the lower bound of the argument is smaller than zero. This is one example of the convenience of the logic programming paradigm to specify constraints.

When increasing the number of constraints, care must be taken that it is still feasible to derive numerous solutions. The aim of the ALP system is not to find a single or a small set of feasible solution, but to find the best expression under a number of relatively mild constraints.

In the example program, a query of the form `?- interval(X, 0, 1).` would ask the system to find a program that lies exactly within the specified interval. The find a feasible solution is then a search problem in its own right, let alone finding an expression that will fit some data well.

It is again instructive to see what happens with the function trees when a mutation or crossover event occurs. As Program 6.2.2 uses a single predicate, the genotype phenotype mapping is monomorph, there is no re-interpretation of genetic material. Consider the mapping:

$$[+, *, x, y, \mathsf{sqrt}, x] \rightarrow_{encodes} xy + \sqrt{x}$$

and

$$[+, +, x, y, \mathsf{sqrt}, x] \rightarrow_{encodes} x + y + \sqrt{x}$$

For reasons of clarity, the integers in the string have been replaced with the symbols for the functors they encode for[3]. Now suppose the first string is cut after the fifth position, leaving $[+, *, x, y, \mathsf{sqrt}]$ and the second is cut after the first position, thus leaving $[+, x, y, \mathsf{sqrt}, x]$. After merging the parts, the complete string will read: $[+, *, x, y, \mathsf{sqrt}, +, x, y, \mathsf{sqrt}, x]$. Without the additional constraints on the square root function, this string would encode for:

$$[+, *, x, y, \mathsf{sqrt}, +, x, y | \mathsf{sqrt}, x] \rightarrow_{encodes} xy + \sqrt{x + y}$$

where the tail $[\mathsf{sqrt}, x]$ is unexpressed. However, when using this string of codons in the logic program 6.2.2, the following situation would occur:

```
interval(sqrt(X), L, U) :-
       interval(X,Lx, Ux),    % X gets bound to x+y
                              % Then Lx = -4, Ux = 15
       Lx >= 0, ...           % fail! Lx equals -4
```

[3] This can be done because there exists a one-to-one mapping between codon value and the clause that is selected.

The clause thus fails because the subtree $sqrt(x + y)$ can possibly include a mathematical error. The backtracking mechanism will now try to redo the last goal that succeeded, which was in this case `interval(y,-5,5)`. Reading the next codons from the genotype would result in generating $sqrt(x)$, which, when added to x leads to a correct expression. The real mapping would thus be:

$$[+, *, x, y, \mathsf{sqrt}, +, x, \mathbf{y}, \mathsf{sqrt}, x] \rightarrow_{encodes} xy + \sqrt{x + \sqrt{x}}$$

The codon encoding **y** above thus disappears from the derivation, and the subtree at the back is used in its place. By using backtracking, performing a crossover is not an all or nothing proposition: if a constraint gets violated in a crossover, there is still a possibility of creating a valid expression by simply trying to apply the next codon. In Section 7.1.1 it will be shown that it is possible to effectively use this mechanism to constrain the search space without removing the effectiveness of the variation operators.

6.2.2 Backtracking

To enable backtracking, the system maintains a list of clauses that have been tried at each point in the resolution process. Once a failure occurs because some constraints have been violated or a maximum depth is reached, the system extract a new codon from the genotype, and the mapping rule is applied. When the codon decodes to an untried clause the process continues. However, when the codon decodes to an already tried clause — that has already proven to fail — the codon will decode to the next untried clause that is applicable. If there is no such clause, the procedure will start at the beginning of the list of applicable clauses. When no more choices remain, thus all clauses are tried and all failed, the system will backtrack to the previous level.

With this procedure, the ALP system is equivalent with Prolog when it is run in the context of a genotype containing only zeros. When a failure occurs, the second clause is tried, and so on.

6.2.3 Initialization

Initialization is performed by doing a random walk through the grammar, maintaining the choices made, backtracking on failure or when a specified depth limit is reached. After a successful derivation is found, the shortest, non-backtracking path to the complete derivation is calculated. An occurrence check is performed and if the path is not present in the current population, a new individual is initialized with this shortest non-backtracking path. Individuals in the initial population will thus consist solely of non-backtracking derivations to sentences.

As a maximum depth limit is used, the possible problem of not creating viable offspring indicated in Section 6.2.1 is not encountered. By making uniform random choices in the initialization procedure, it is however sensitive to the distribution of terminal and binary functions in the logic program. It is not clear at this point what the repercussions of changing this initialization procedure are when using more heavily constrained programs than the ones considered in Section 6.2.1. Therefore, it was chosen to use the initialization procedure described here to create a reference implementation that produces a fully viable initial population without clones. This reference implementation can be used to check possible improvements against.

6.2.4 Performance Evaluation

The performance of expressions (computer programs) is typically calculated in a special module, written in a compiled language such as C. This program walks through the tree structure and evaluates each node. This is however not necessary if the performance can be readily evaluated in the logic program itself. The query investigated typically has the form: find that derivation for $expression(X)$, such that $eval(X, F)$ returns the maximal or minimal F. Typically, the top level predicate for this system will have the form:

```
objective(F) :-  expression(X), eval(X,F).
```

This predicate is called by the system with a non-grounded (free) `F`. The predicate `expression/1` will derive a parse tree, the `eval/2` predicate will evaluate the parse tree and bind the objective function value to `F`.

6.2.5 Special Predicates

All Prolog built-in clauses such as assignment (`is/2`) are evaluated in Prolog directly. This is done as often such clauses are deterministic and depend on the Prolog depth-first search strategy, or they expect some variables to be bound to ground terms. Calls to external libraries are evaluated directly as well.

Often, there is a need for arbitrary real valued and integer constants to appear in the expressions. A prime example is symbolic regression. In previous work, a recursive logic program was used to derive and evaluate such constants (Keijzer et al., 2001a).

In the current implementation of the ALP system, two special predicates are used that can retrieve integer and floating point values. These are called `ext_int/2` and `fp/1` respectively. Those values are stored in a separate string, which is kept at the same length as the string of choices. Although this involves storing many values that are not expressed, for the current purposes the additional memory overhead does not present difficulties.

The derivation engine recognizes these predicates, and retrieves values from the genotype. This way, variation operators can be defined on those constants, which then co-evolve with the string of choices.

6.3 Summary

In the preceding sections, the ALP system is outlined, discussing programs using a single predicate, programs using multiple predicates and programs that have additional, context sensitive constraints. Some variation operators have been discussed,

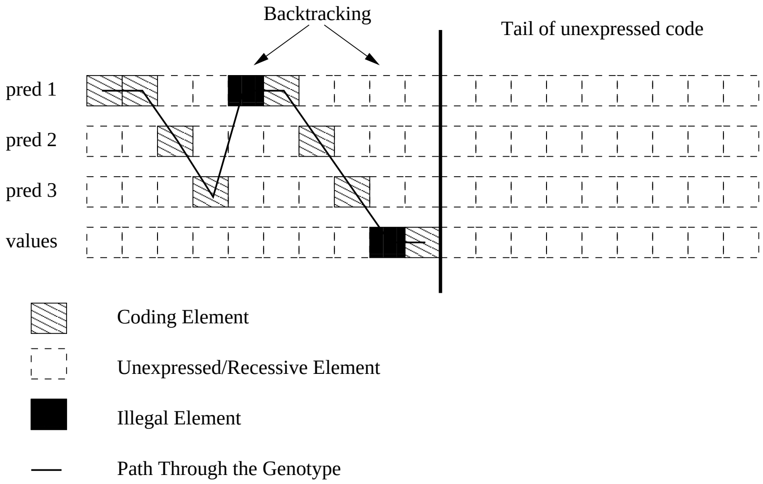


Figure 6.9: Overview of the genotype used in the ALP system for a program consisting of three predicates. It uses a separate string for each predicate in the program (although this is in the implementation packed in a single string) together with a string of floating point values. The choices made in the resolution of the logic program determine a path through the genotype, while it is possible that constraint-violating codons get ignored. Usually a tail of unexpressed code evolves, that contains unexpressed genetic material. The expressed code is in general a small fraction of the complete genetic code.

together with backtracking and initialization. Figure 6.9 depicts the genotype of the individuals that are used in the ALP system. The representation undergoing variation and selection is depicted with multiple strings, one for each predicate in the program. In practice these strings are packed in a single string, using the linearization formula from Equation 6.9.

There can be a massive amount of unused information in such an individual: at every location there is only one coding element, when the string is finished decoding into a complete expression, the rest of the string is ignored.

Prefixes and suffixes are central in the ALP system. Not only does ripple crossover explicitly manipulate these structures, they also have a straightforward interpretation in terms of the search tree for the logic program. A prefix determines a partial path through the search tree; it thus effectively sets the starting point for the search. The suffixes on the other hand encode continuations of these search paths. When working with recursively defined programs, these continuations encode for similar paths in different contexts.

The mapping process that defines how the genotype gets translated into a computer program is depicted in Figure 6.10. The genotype gets implicitly translated into a string of integers that represents the shortest non-backtracking path to the goal-state. When inducing computer programs, the goal-state is equivalent to a successfully induced computer program given the constraints.

Although it is possible to find some biological analogies between this mapping process and the translation-transcription cycle in DNA, these are not engineered into

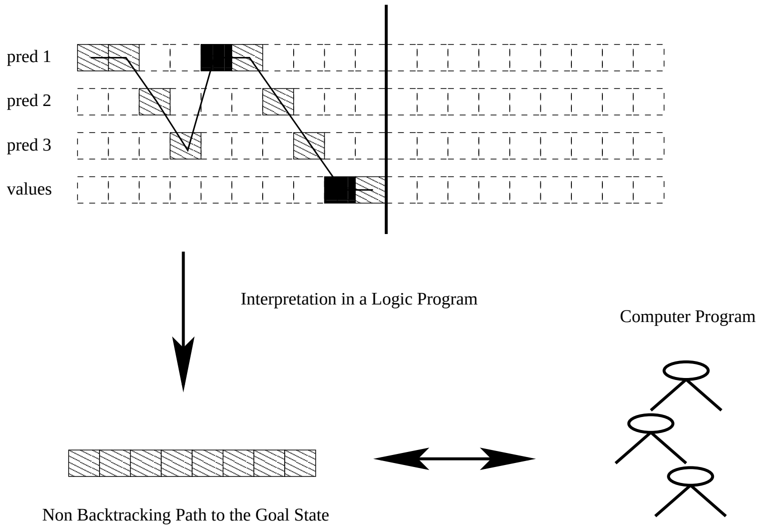


Figure 6.10: Overview of the mapping process. The multiple strands represent a selection from one of the predicates in the program, this selection process results in a single string of choices. This string of choices is translated into a parse tree.

the system with the purpose of mimicking their biological counterparts in the vague hope that when nature uses such constructs they might be useful, but are direct consequences of the choice to induce arbitrary length search paths through a logic program. The design choices are made to enhance the optimization process in ALP, not to mimic any natural phenomenon. Multiple strands of genetic material appeared when the linkages between clauses belonging to different predicates were removed. A logical result of this choice is that the "translation" at one part of the string influences the "transcription" further to the right.

The emergence of unexpressed code is a logical consequence of the use of ripple crossover. Ripple crossover operates very unlike any variation operator appearing in nature, it is however a relatively easy and sensible operator to apply to the recursively defined programs considered here. The appearance of illegal strands of code that do not get expressed is caused by the use of a backtracking mechanism. Backtracking is used because it proved to be invaluable when using context sensitive constraints (Keijzer et al., 2001a).

6.4 Related Work

Wong and Leung (Wong and Leung, 1997) hybridized logic programming and genetic programming in their system LOGENPRO. In LOGENPRO, one defines a grammar consisting of syntactic and semantic definitions in the form of a Definite Clause Grammar (DCG). This grammar is transformed into a logic program by automatic means in such a way that next to the parse tree for the expressions that are evaluated, a parse tree of information about the derivation is generated. This second parse tree describes the rules that are applied and the variable substitutions that are made. This is the structure that is manipulated by the variation operators.

Due to the semantic constraints, some fairly intricate subtree crossover and mutation operators are used. Even then, a semantic validation — checking whether the newly created parse tree is accepted by the logic program – needs to be performed.

Ross (Ross, 1999) describes a similar system that uses Definite Clause Translation Grammars (DCTG). The difference between a DCTG and a DCG lies in the explicit separation between syntax and semantics in the DCTG, while the DCG mixes the syntax and semantics in the body of the clauses. Like in (Wong and Leung, 1997), the DCTG in (Ross, 1999) is translated into a logic program and parse trees of the derivation process are manipulated. The crossover described in (Ross, 1999) seems to only use type information contained in the predicate names and arity at the heads of the clauses and swaps derivation subtrees that contain the same head. A semantic verification (running the Prolog program on the derivation tree), is subsequently performed.

In both approaches the variation operators are strongly typed, and subsequently the number of allowable crossover pairs in the parse trees can be significantly reduced. This is in stark contrast with the ALP system, where all parts of the string can be subject to crossover with any other part of any other string. Another difference between these approaches and the ALP system lies in the method of creating expressions. Especially in the approach outlined by Ross (Ross, 1999), the strict separation between context-free syntactic rules used to generate sentences and the semantic validation used to validate the expressions can lead to a wasteful generate and test cycle in the algorithm. The LOGENPRO system on the other hand tries to push the test-cycle inside the crossover operator, leading to a potentially more efficient system. However, the net effect of both approaches is that crossover is restricted to swapping subtrees that directly lead to a valid program. The crossover operator is thus *strongly typed*.

The ALP system on the other hand is not constrained to use logic grammars as it works with logic programs directly. Although a logic program is eminently suitable to define a logic grammar, logic programs can also take more direct, constructive, approaches towards generating structures. The use of untyped variation operators together with a fault tolerant mapping method is also very different from the tree based genetic programming systems described above.

As an example of the difference between a grammatical approach and a constructive approach, consider the problem of generating a permutation of a list of items. Such a permuted list can be needed as a subtask for a larger programming task. It is already not perfectly clear how to approach this problem using a grammar as we want to transform one sequence into another sequence. Grammars are usually used to define legal sentences, not transformations. Assume for the sake of simplicity however that the object is to create a permutation of a fixed length list $[1, 2, \ldots, 10]$. In a grammatical approach, one approach would be to define the syntax to be a list of length 10, where the elements are numbers. In a DCG this would look like

```
number(1).
...
number(10).

list([], 0).
list([N|L], Sz)  →  number(N), list(L, S), { Sz =:= S + 1 }.
list10(L)        →  list(L, 10).
```

Which would define all lists containing the numbers 1 to 10. The curly brackets are used to indicate the semantic checks, the functor =:=/2 denotes arithmetic equality. The important thing to notice is that it presents a syntactical definition, which is eminently suitable for parsing lists. The check for the size is a small semantic component. However, we are interested in a permutation, thus there is a need to check if a number already occurs in the list. A straightforward implementation of this would be to simply check if the number already occurred:

```
perm_list([],0).
perm_list([N|L], Sz)  →  number(N), perm_list(L, S),
                         {not(member(N,L)), Sz =:= S+1}.
perm_list10(L)        →  perm_list(L,10).
```

This will indeed define all possible permutations, but in the process of generating them, it would generate all possible sublists, discarding the illegal lists. This is not a problem when parsing a possible permutation, but when generating such a list it can be very wasteful. Although possibly less wasteful ways of defining a grammar for this problem are imaginable, the main point to be made is that many constructions such as permuted lists are not best modelled by a grammar, making a distinction between a context-free syntax together with semantic constraints. Although for parsing sentences the grammatical approach is sufficiently powerful, for generating sentences a more procedural approach seems to be needed. ALP can deliver such a constructive approach: in a logic program, it is possible to define a permutation procedurally

```
int(0,_).
int(N,Z) :- int(N1,Z), N is N1+1, N < Z.

permute([],[]).
permute(List,[PermHead|PermRest]) :-
      length(List, L),
      L > 0,
      int(Choice,L),                      % choose an integer
      nth0(Choice, List, PermHead),       % get the choice and
                                          % make it the head
      delete_nth0(Choice, List, Smaller), % create smaller list
      permute(Smaller,PermRest).
```

The predicate permute/2 transforms an arbitrary list into a permuted version of the same list. It does this by making an arbitrary choice from the input list and putting that one at the head of the output list. The element that is chosen gets deleted from the input list and the algorithm recurses. It ultimately hinges on the predicate int/2, that selects an integer bounded above by the length of the list[4]. The program thus defines a procedural way of permuting a list, it is difficult to identify *grammatical* elements in this small program. Furthermore, this program is generic, it does not make any assumption about the contents of the list. The predicate permute/2 is set up to do a permutation of any list.

[4]When randomly generating strings, the int/2 predicate is heavily biased towards sampling the lowest numbers. Therefore ALP uses the special predicate ext_int/2 (See Section 6.2.5) that retrieves an integer from the genotype.

This example also illustrates some possible problems with typed crossovers in constrained domains. In the `list/2` predicate defined above, even though the definition is recursive, the check for the size of the list implies a very stringent constraint for a subtree crossover. Only crossovers that swap lists of the same size are valid, all other crossovers are invalid. ALP, using an inherently more messy mapping process, would however be able to migrate any part of the string encoding a list to any other part, as long as it can decode to a list of the proper length. The tail of unexpressed code would be helpful to achieve this.

Subtree and substring crossovers in the ALP system It is however possible to implement a subtree crossover in the ALP system by utilizing information gathered about the evolution of the goal stack and the predicate that was used at every point in the derivation. Selecting a crossover point k_1, its subtree is spanned by the first point $k_2 > k_1$ such that:

$$g(s, k_2) = g(s, k_1) - 1$$

Where $g(s, k)$ is the goal stack function defined in Equation 6.3. The point k_2 is the point where all the unresolved literals pushed on the stack by the predicate encoded by s_{k_1} are resolved and removed from the goal stack, which in turn means that all arguments needed by the predicate used at point k_1 have been bound to ground terms.

The condition can be simplified to

$$\begin{aligned} g(s, k_2) - g(s, k_1) &= g_0 + \sum_{i=0}^{k_2} b(s_i) - g_0 - \sum_{j=0}^{k_1} b(s_j) = \\ \sum_{i=k_1}^{k_2} b(s_i) &= -1 \end{aligned}$$

The condition that it is the first point that has this property makes sure that we swap a single subtree. Subtree crossover is then a special case of a general structure preserving two-point crossover on strings, where given two strings s_1 and s_2 with crossover points $k_1 < k_2$ and $l_1 < l_2$

$$g(s_1, k_1) - g(s_1, k_2) = g(s_2, l_1) - g(s_2, l_2)$$

Interpreted in terms of the goal stack, this condition simply states that the differences in goal stack size for the substrings must be equal to each other. When swapping substrings that abide this constraint, the integrity of the goal stack is guaranteed and no tail of unexpressed code is needed. This does however not necessarily work for programs with multiple predicates and context-sensitive constraints.

In terms of the shape graph, a subtree is simply the first point to the right that is located one notch lower than the current level. In terms of the shape graph and the parse tree this can be depicted as

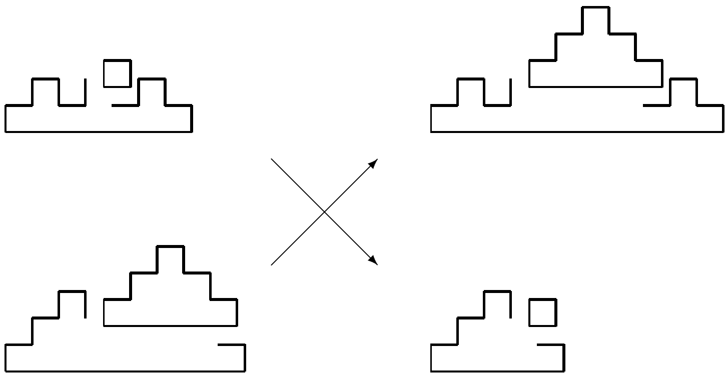
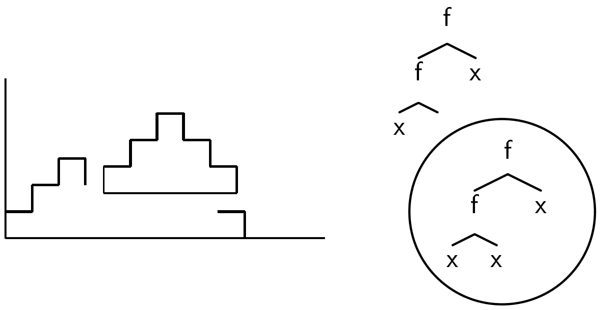


Figure 6.11: Subtree crossover on the shape graph.



With the additional provision that two subtrees can only be exchanged when they encode for a clause in the same predicate, this subtree crossover is equivalent with the subtree crossover outlined by Ross (Ross, 1999). It has the additional benefit that instead of having a semantical validation check and possible direct failure in the case of context-sensitive constraints, the derivation process would still be able to re-interpret the rest of the string and possibly return a complete expression.

Figure 6.11 depicts the process of subtree crossover in terms of the shape graph.

6.5 ALP, ILP and CLP

Inductive Logic Programming (ILP) aims at inducing *logic programs*. based on data (Muggleton and Raedt, 1994). As logic programs themselves can be readily expressed as parse trees in a logic program[5], ALP could conceivably be used to induce logic programs. Even in that case, a large difference between ILP and ALP would remain: ILP usually works by transforming an overly specific logic program (the set of all positive and negative examples) into a more general program, using various heuristics to ascertain which generalizations are allowed.

[5]logic programming shares this ability with LISP

ALP usually operates in a constructive way, creating a parse tree (in this case a logic program) out of an inductive definition of possible parse trees (logic programs). However, due to ALP's ability to use transformations and procedural rules as well, it might be possible to constrain ALP in such a way that it will only use those rules that are used in ILP. In that case ALP can be used merely as an alternative to depth first search in a highly constrained search space. The worth of this approach is not explored further here.

Constrained Logic Programming (CLP) tries to find solutions in heavily constrained search domains defined by logic programs (Jaffar and Maher, 1994). Here the object of search is usually a single or small set of solutions that abide the constraints. ALP is is general not capable of significantly optimizing expressions when generating a single solution is a difficult search problem in its own right. Creating the first generation should then already solve the problem. ALP works best when there is an abundance of solutions of different merit — which is the case in program induction. However, in Section 7.1.2 and Chapter 8, it will be shown how in some circumstances declarative (hard) constraints can be transformed to preferential (soft) constraints. Using a multi-objective search then provides a viable approach to optimize in the presence of difficult constraints.

6.6 Summary

In this chapter the adaptive logic programming system has been outlined. Particular attention was given to the mapping process from a string of codons (a path through a logic program) and the resulting parse tree that defines the computer program. The effect of the variation operators on the string of codons was examined in the context of simple programs, syntactically constrained programs and semantically constrained programs.

The shape graph was introduced to visualize the shape of a parse tree generated by a path through a logic program. This shape graph is defined as the number of literals on the goal stack for each point in the tree generation process. In the case of simple single predicate programs without constraints, the shape graph gives all information needed to implement structure preserving one point and two point crossovers.

ALP, being not bound to a grammatical formalism but to the capabilities of a Turing-complete programming environment, might have promise as a powerful and robust approach to the problem of automatic program induction in the context of syntactic and semantic constraints. The next chapter is devoted to a feasibility study of ALP's untyped variation operators and to some limited case studies in problems involving a variety of constraints.

Chapter 7

Applications for the ALP System

The ALP system is implemented using SWI-Prolog[1]. SWI-Prolog defines a two-way C API, that allows to call Prolog from C (used by the resolution engine) and C from Prolog (used for evaluation of parse trees). The genetic algorithm is implemented using the evolutionary objects library[2]. The system is capable of performing multi-objective optimization, using the elitist non-dominated sorting algorithm NSGA-II (Deb et al., 2000).

The general optimization cycle in the algorithm that is used consists of a variation step in which the population is doubled in size, using a tournament selection of size 2 to obtain the parents, and a subsequent selection step in which the population is sorted and halved. In evolutionary strategy terminology this is a $(\mu{+}\mu)$ strategy with an additional tournament selection step. In the case of multi-objective optimization, the sorting is based on linear ranking, consisting of an integer value that designates which front the individual belongs to, and a fractional value that is used to break ties. This fractional value represents the uniqueness of the individual in its front of non-dominated solutions. The NSGA-II algorithm improved upon the original NSGA algorithm by removing the dependence on a sharing parameter (Deb et al., 2000).

This algorithm is strongly elitist as a new individual has to improve upon at least one individual in the previous generation in order to be included in the breeding population for the current generation. One of the ramifications of this choice is that there is no straight reproduction in the system, all newly added individuals will have undergone some form of variation.

7.1 Applications

Here we will examine the feasibility of the ALP system on various applications with varying constraints. First a small experiment showing how some background

[1] http://www.swi.psy.uva.nl/projects/SWI-Prolog

[2] http://www.sourceforge.net/projects/eodev

Strategy	$(\mu + \mu)$
Population Size (μ)	500
No. of Generations	100
No. of runs	100
Crossover Probability	0.97
Constant Mutation	0.03
Maximum depth at init	6
Maximum depth	15

Table 7.1: Parameters for the ALP system used for nearly all experiments.

knowledge about the function set on the artificial ant problem can be used to remove various source of redundancy. Secondly, the system is used to find an equation that is well-adapted to some data set, while at the same time is constrained to find equations that produce outputs that are provably in the right domain. The program uses interval arithmetic to achieve this. Thirdly, the system is tried upon the problem of finding a dimensionally correct equation. Finally the system is constrained to produce equations in the language of matrix algebra.

The main aim of these experiments are to give a proof of principle that the adaptive logic programming system introduced here is capable of dealing with a wide range of applications. The applications involving interval logic, units of measurement and matrix algebra are important to make genetic programming more suitable for use in scientific and engineering settings. The artificial ant problem is more of a toy problem, but points to the possibility of extending the power of genetic programming by disallowing some a priori bad or redundant constructs.

Each of the programs studied here have context-sensitive constraints. Where the performance between subtree crossover and ripple crossover are compared, the subtree crossover is modified to have a second try with new crossover points if a crossover does not produce a valid offspring. This to level the field somewhat[3]. The comparative experiments performed here are mainly used as evidence on the large difference between ripple crossover and subtree crossover when applied to inducing expressions subject to context-sensitive constraints.

7.1.1 A Sensible Ant on the Santa Fe Trail

The artificial ant problem has been studied intensively in (Langdon and Poli, 1998) which showed that it is a difficult problem with multiple ridges and local optima. The goal is to find a computer program that steers an ant over a trail of food pieces, eating as much food as possible. The trail that is used is the well-known Santa Fe trail that contains 89 pieces of food. The success criterion for an artificial ant program is then to steer the ant to eat these 89 pieces of food within 600 steps.

The logic program that defines the space of allowable ant programs can be stated as follows.

[3] As described in Section 6.2.1 ripple crossover was modified to attempt a reversal of roles for prefixes and suffixes in case of failure.

```
ant(move)
ant(left)
ant(right)
ant(iffoodahead(X,Y)) :-  ant(X), ant(Y).
ant(seq(X,Y)) :-          ant(X), ant(Y).
```

Program 7.1.1: Program for inducing an artificial ant.

The `move` atom indicates a move forward by the ant, `left` and `right` turn the ant 90 degrees on the grid, the `iffoodahead` instruction branches on the information whether there is a food pellet present in the cell the ant is facing, while the `seq` operator simply applies its arguments in sequence. A program consisting of these instructions is iteratively applied until time runs out.

In the usual genetic programming notation of terminal and function sets, this logic program above can be described by: $T = \{$ `move/0`, `left/0`, `right/0`$\}$ and $F = \{$ `iffoodahead/2`, `seq/2` $\}$. The Program 7.1.1 will be extended to implement a few context-sensitive constraints.

The space of possible ant programs contains many ineffective pieces of code that can be identified even before trying to find a program for a specific trail. It is for example ineffective to let the ant move left and subsequently move right without any commands in between as its overall state would not have changed. Furthermore, directly nesting `iffoodahead/2` calls is also ineffective as the outcome of the check is already known. This embodies knowledge about the semantics of the function set. We might also assume that if there is food ahead, moving toward the food seems to be a good idea.

These constraints are readily implemented in the ALP system. For this an `ant/3` predicate will be used. As usual, the first argument will be used to induce the computer program that is evaluated. The second argument is used to specify the input constraints, constraints that are imposed by the caller, while the third argument is used to specify some output constraints: constraints subsequent clauses need to abide.

The start clause is of the form `ant/1` that calls `ant/3`, not demanding any constraints on the program (signified by the empty list `[]`), and ignores the output constraints (the underscore symbol). Thus:

```
ant(X) :- ant(X,[],_).
```

The sequence operator is used to propagate the constraints.

```
ant(seq(X,Y),In,Out) :- ant(X, In, C), ant(Y, C, Out).
```

The two additional arguments `In` and `Out` get imposed on the arguments of the sequence operator. The intermediate variable `C` is used to propagate the constraints from the first subexpression to the other.

The clause that induces the move atom is defined as follows

```
ant(move, C, []) :- \+ member(no_move, C).
```

where \+ is the Prolog symbol for negation by failure: it will only succeed when the execution of its argument fails. The member/2 function is a built-in clause, and is executed in Prolog directly. This clause constrains the move atom to be only applicable when the set of constraints does not contain the no_move atom. A move furthermore removes all constraints, it thus returns the empty list []. Turning to left:

```
ant(left, C, [no_right]) :- \+ member(no_left, C).
```

Is applicable only when there is no no_left constraint. It imposes a no_right constraint to the next action. So when the ant turns left, it will not immediately turn right, because:

```
ant(right, C, [no_left]) :- \+ member(no_right, C).
```

For the iffoodahead function the following constraints are imposed:

```
ant(iffoodahead(X, Y), In, Out) :-
      \+ member(no_if, In),
      ant(X,[no_left, no_right, no_if], OutLeft),
      ant(Y, [no_move, no_if], OutRight),
      intersection(OutLeft, OutRight, Out).
```

The first literal in the body of the clause imposes the constraint that the clause can only be used when there is no no_if constraint pending. Furthermore it specifies that when food is spotted, turning is not allowed, which will necessarily lead to a move as the next action (note that due to the propagating of constraints any number of intermediate sequence operators can occur as long as the first action that is applied is a move). When there is no food ahead however, moving is not allowed: this will mean that a construction such as 'iffoodahead(seq(move,X), seq(move,Y))' is not allowed, as this could equivalently (and shorter) be specified as 'seq(move,iffoodahead(X,Y))'. For both branches, it is not allowed to immediately check for food again. The clause will return the intersection of the constraints imposed by the two branches, thus if both branches end with imposing the same constraint(s) these will be propagated to the next action.

This in effect implements information about some immediate redundancies in the function set for the artificial ant problem. It reduces the search space by disallowing specific combinations of code. No knowledge about the trail is included in the program, the constraints are imposed to remove redundancies and to take one maybe sensible action: when there's food spotted, eat it.

The effects of subtree crossover and ripple crossover in the context of these constraints are expected to be quite different. In the case of subtree crossover, attempting to move a subtree to an illegal context will result in a failure. Consider for example inserting the subtree

```
   seq              into location X of          seq
  /   \                                        /   \
left  move                                  right   X
```

thus trying to form the tree

```
      seq
     /   \
 right    seq
         /   \
      left   move
```

The constraint against turning would make this tree invalid. When using ripple crossover however, the backtracking operator would simply skip the `left` action and the intermediate tree would become

```
      seq
     /   \
 right    seq
         /   \
      move    Y
```

Which is as such invalid. If there is however still genetic material left, possibly in the tail of unexpressed code, the resolution process would use that to fill in the value of Y and then create a valid tree. As the constraints in this grammar only exclude certain pairs of actions or conditions, the backtracking operator is expected to be relatively successful in creating valid offspring.

The effect of using these constraints is dramatic in the ability of ALP to find solutions to the problem. Figure 7.1 shows the cumulative probability of success over the generations for both ripple crossover and subtree crossover for both logic programs. The success rate for solving the problem goes to 97% for ripple crossover, while subtree crossover's performance goes to 80%. On the standard formulation of the problem, success rates are much lower. We cannot conclude that ripple crossover together with backtracking is better suited to handle the constraints, as the success rates on the standard formulation of the problem already show a significant advantage for ripple crossover. It appears that for this type of problem the more destructive variation applied through ripple crossover has a benefit over the more localized changes of subtree crossover. This is especially pronounced in the later stages of the run, where cumulative success for subtree crossover levels off, while the runs using ripple crossover keep finding solutions.

It is however instructive to examine the failure rate of both methods. These are depicted in Figure 7.2. Initially the failure rates of both methods peak when the first illegal crossovers occur. Very soon however, the tail of unexpressed code begins to form and together with the backtracking mechanism, the failure rate of ripple crossover drops to very low values. The failure rate of subtree crossover on the other hand initially rises and levels off at a rate of 5%. In both cases the crossover operators have two tries in creating a valid individual. Due to the elitist truncation selection method that is used, invalid individuals only slow down the search. The

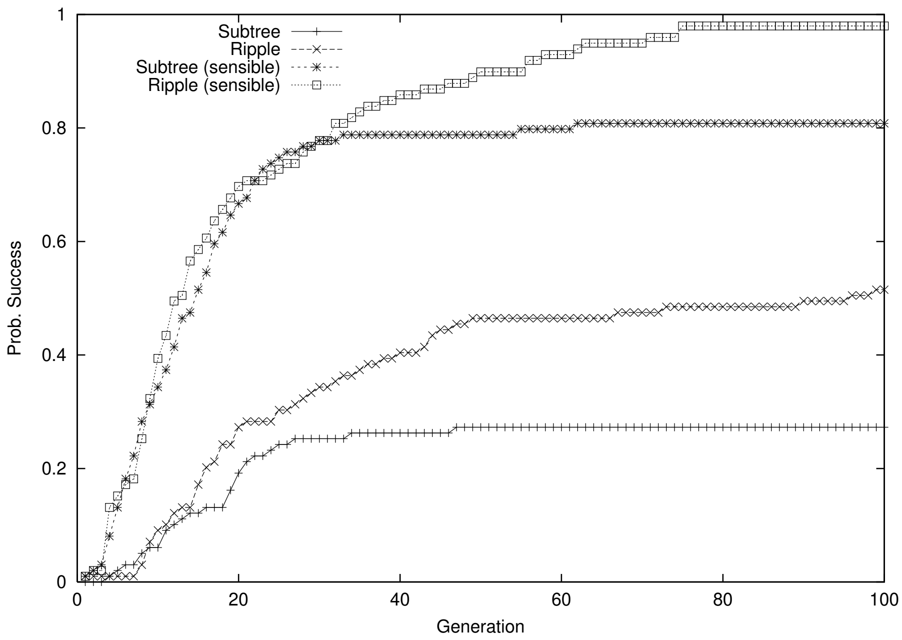


Figure 7.1: Probability of success in eating 89 pieces of food scattered on the Santa Fe trail for subtree crossover and ripple crossover using the straightforward Program 7.1.1 and the 'sensible' ant program.

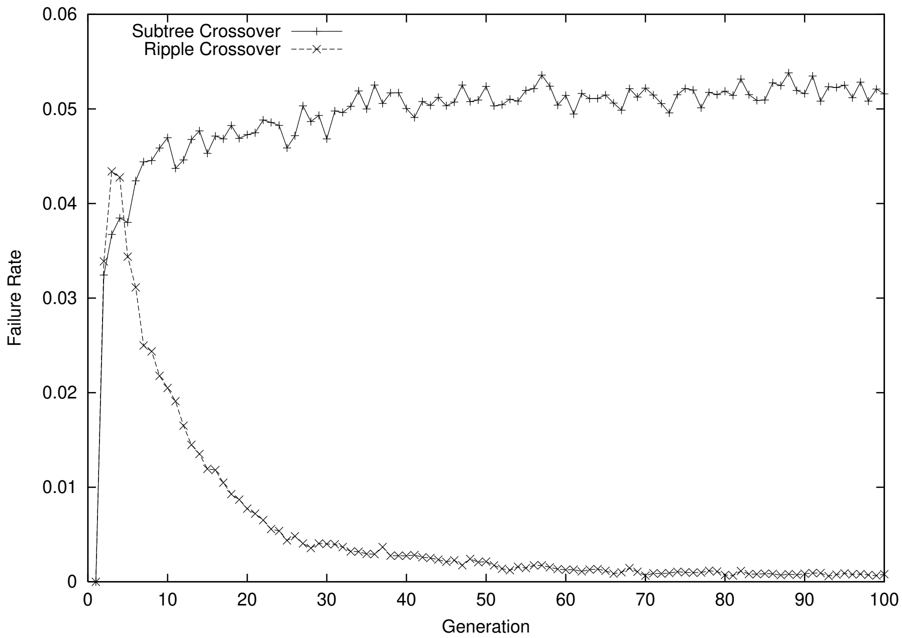


Figure 7.2: Failure rates of subtree crossover and ripple crossover on the sensible artificial ant problem.

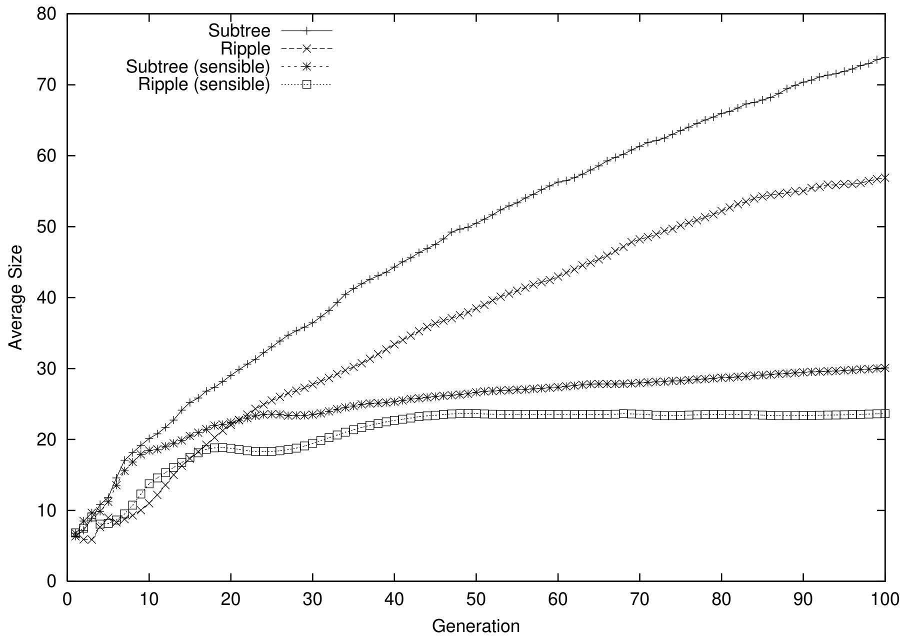


Figure 7.3: The evolution of the average size of the population on the artificial ant problems.

5% invalid individuals that are created by subtree crossover would mean that the genetic algorithm effectively processes 475 individuals per generation rather than 500.

It is clear from this experiment that the 'sensible' program helps in solving the problem faster than the straightforward program, regardless of the variation operator that is used. The 'sensible' program uses context-sensitive constraints to introduce some a priori information to exclude redundant subexpressions.

By reducing the search space to exclude certain constructs, an implicit bias towards shorter solutions is introduced. It was estimated in (Langdon and Poli, 1998) that the space of possible ant programs has a high density of solutions of a relative short size. Figure 7.3 shows the evolution of the size of the population undergoing both crossover operators. The 'sensible' ant constraints induce populations containing programs that are significantly shorter than when using the unconstrained definition of possible ant programs.

Concluding the Experiment

Although the 'sensible' ant-problem studied here is a toy problem, the way the constraints are introduced point to a more general application: disallowing certain constructs because they are redundant or meaningless. This can often be identified just examining the definition and description of the function set. It proved to be fairly easy to disallow some commonly occurring fragments from the ant programs, the reduced search space allowed to search with great success. Many function sets have known redundancies or allow nonsensical combinations. By using a logic programming approach it is possible to disallow specific constructs. This can be contrasted with a pure syntactical approach (context-free grammars), where it is only possible to express what is allowed.

A possible application of this technique in symbolic regression would be to specifically disallow nesting of some functions. For example, the clause

```
expr(exp(X),C) :- \+ member(no_exp,C), expr(X, [no_exp|C]).
```

would disallow nested exponentiation functions. In the practice of performing symbolic regression it occurs fairly often that such nested expressions appear, and their presence makes it usually difficult to interpret the equation. When using a subtree crossover using such a grammar, any attempt of inserting a subtree containing an `exp` function underneath an `exp` function that is already present would lead to a failure. Using ripple crossover with backtracking, such an offending operation would simply be skipped and the next instruction would be read from the genotype. Which of the two approaches is the best cannot be answered conclusively at this point.

7.1.2 Interval Arithmetic

The solutions provided by genetic programming in the area of symbolic regression can exhibit several types of overfitting behaviour; the most destructive ones occur when arithmetical errors are induced. When the function sets include functions that are not defined in the full range of possible inputs such as division and the square root function, the operators are usually protected (Koza, 1992) to return default values in the case of an error. Unfortunately, this protection only works well when the arithmetical error occurs in the training set: if errors occur on a different set, the default values are plainly returned, which might lead to strange behaviour of the expressions. For the division operator, protecting just a division by zero does not solve the problem of ill-defined expression: consider Figure 7.4 where an expression is trained on the indicated data points and subsequently evaluated over the full range. It induces an asymptote and the usual protection mechanism will only protect the point where the actual division by zero occurs. Values close to this point will lead to a prediction of arbitrary large values.

A method to avoid mathematical errors and calculate the domain of an expression is to use interval arithmetic that calculates upper and lower bounds for each part in the expression. Interval arithmetic is readily expressed in the ALP system, the logic program 6.2.2 that was used to illustrate the use of backtracking implements this. The program calculates the theoretical upper and lower bounds of an expression.

The logic program that generates expressions and their bounds while avoiding mathematical exceptions is used on a sediment transportation problem which has been described in Section 3.1. Here we are interested in inducing an empirical formula that predicts the concentration of sediment near the bed of a stream. A *concentration* cannot have arbitrary values, it is constrained to lie within 0 and 1, where 1 means complete saturation. To implement such constraints, we can use the logic program 6.2.2 defined in Section 6.2.1.

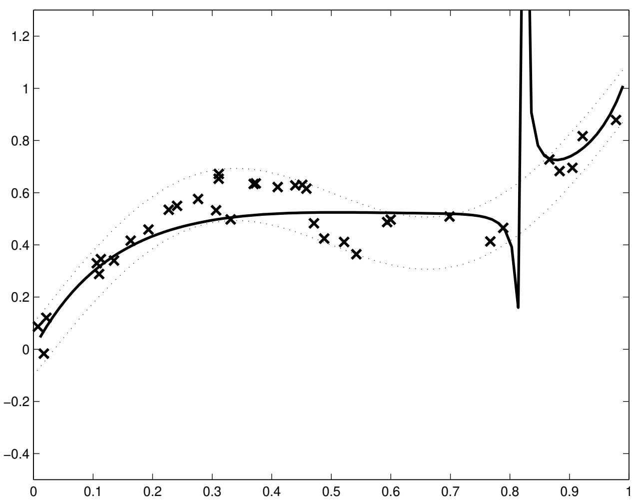


Figure 7.4: An automatically induced expression using the indicated points as the training set. It is evaluated over the full domain. Note that the usual protection mechanism of disallowing a division by zero would not eliminate the asymptote itself. This leads to wildly inaccurate values in the neighbourhood of the invalid point.

The terminals for this problem will be the variables θ and θ' that are dimensionless variables (Shields parameters) that are derived from measured variables. The lower and upper bounds for these variables are empirically determined by examining the dataset, but in principle these could be set to theoretical values. Because the lower bounds are close to zero for the dependent and independent variables, they were set to 0. Now the object of search is an expression that *(i)* fits the data well, and *(ii)* keeps within bounds. This is easiest to set up with a multi-objective search, especially considering the fact that the `interval` predicate calculates worst case bounds[4]. It was however not tried to calculate better bounds in the grammar for fear of overcomplexifying the logic program.

This leads to the following setup:

[4] Consider for example a variable x with a data range of $[-1, 1]$, and the expression $x * x$. The interval that is calculated for this expression would be $[-1, 1]$, while in reality, the minus sign would always cancel out, leaving the tighter bounds of $[0, 1]$.

```
interval(theta, 0, 6.08).
interval(theta_p, 0, 1.98).


\* Rest of the interval/3 predicate, see Program 6.2.2 *\


interval_error(O, L, U) :-
  O is abs(L-0) + abs(U-0.55).  % bounds for cb [0,0.55]

objective([O1,O2]) :-
  interval(X, L, U),
  eval(X,O1),                   % evaluate on data
  interval_error(O2, L, U).     % calculate the interval error
```

where the `objective`/1 predicate is called by the resolution engine. The rest of the definition of the `interval`/3 predicate can found in Program 6.2.2 which defines the interval arithmetic for addition, multiplication, subtraction, division and the square root function. It returns two objective values that will be handled by the NSGA-II algorithm (Deb et al., 2000). Not only will ALP try to induce an expression that follows the data range, the use of Program 6.2.2 also removes the need of using protected algebraic operators, as the expression is guaranteed to be valid in the domain defined by the input variables.

Results

To obtain a baseline measure of the performance of the system, first a set of 100 runs are performed where the second objective is not used. Here the operators are protected in the data range, thus divisions by zero are impossible and square roots of negative numbers can also not occur, but there is no selection pressure towards expressions that follow the desired data range. The errors on the data are reported as normalized root mean squared errors.

Next to this experiments are performed that use the second objective: the interval error. All runs are performed on the same training data for the sediment transportation problem. As the multi-objective run ends up with a front of non-dominated solutions, in a post-processing step a choice must be made from this front. It was decided that the best performing expression on the data that had an interval error smaller than 0.1 would be selected. It was not insisted to have an exact match on the bounds because the bounds calculated in the logic program are not tight.

Although no conclusive evidence can be expected from applying a system on a single problem, two questions will be investigated. The first question is concerned with the optimization ability: does the addition of a second objective help or hinder the search for a well-fitted expression? The second question is concerned with the performance of the expressions on the test-set: does the addition of a second objective optimized on the data range help in avoiding overfitting?

Figure 7.5 shows the evolution of the training performance for subtree and ripple crossover on both the multi- and single objective problems. The use of a single

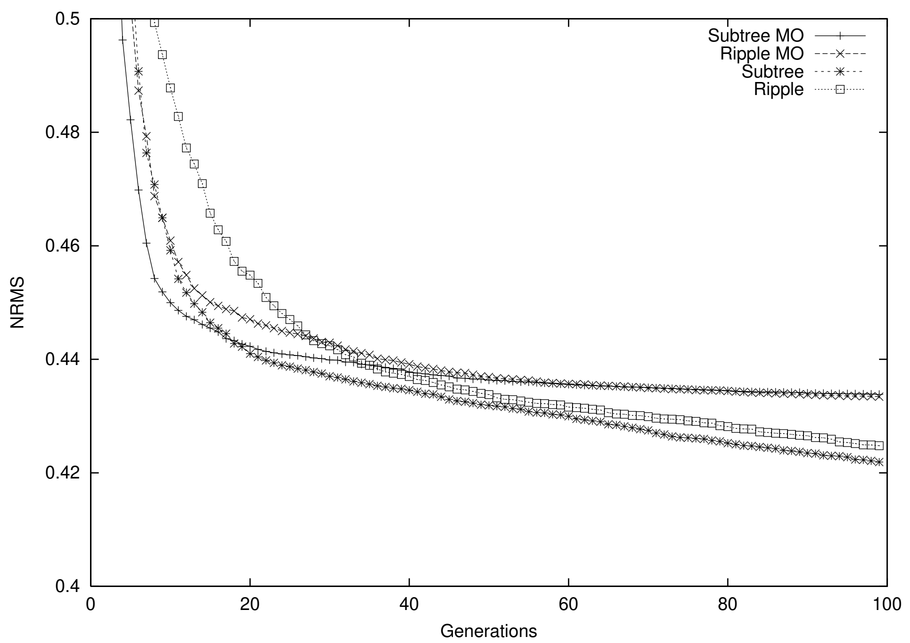


Figure 7.5: Evolution of the mean performance on the training set for the multi-objective and single objective runs. The mean performance is calculated as an average over the performance of the best expression in a generation.

objective leads to significantly better results on the training set than the multi-objective setup. Table 7.2 shows the results of a two-tailed t-test on the mean performance at the final generation.

When evaluating the expressions on the test set however, these differences disappear. Figure 7.6 shows the performance of the selected expressions from each run on the testset. Even though the best of those are very similar in their ability to generalize, the worst performing expressions taken from the single objective runs do however show a large overfit. Note that with the normalized RMS measure used here, an error of 1.0 would be produced by an expression that has a constant value — the mean of the target signal. Expressions that on the test set score worse than an error of 1.0 are thus worse than the performance of a constant. This level of overfitting

	Subtree MO	Ripple MO	Subtree	Ripple
Subtree MO		0.62	10^{-6}	10^{-7}
Ripple MO	0.62		10^{-6}	10^{-7}
Subtree	10^{-6}	10^{-6}		0.29
Ripple	10^{-7}	10^{-7}	0.29	

Table 7.2: Probability that the difference in observed mean performance in the final generation is caused by random effects using a two tailed t-test on the **training set**. The label 'MO' designates the multi-objective runs. There's no significant difference between subtree crossover and ripple crossovers; the differences between the multi-objective runs and the single objective runs is however significant.

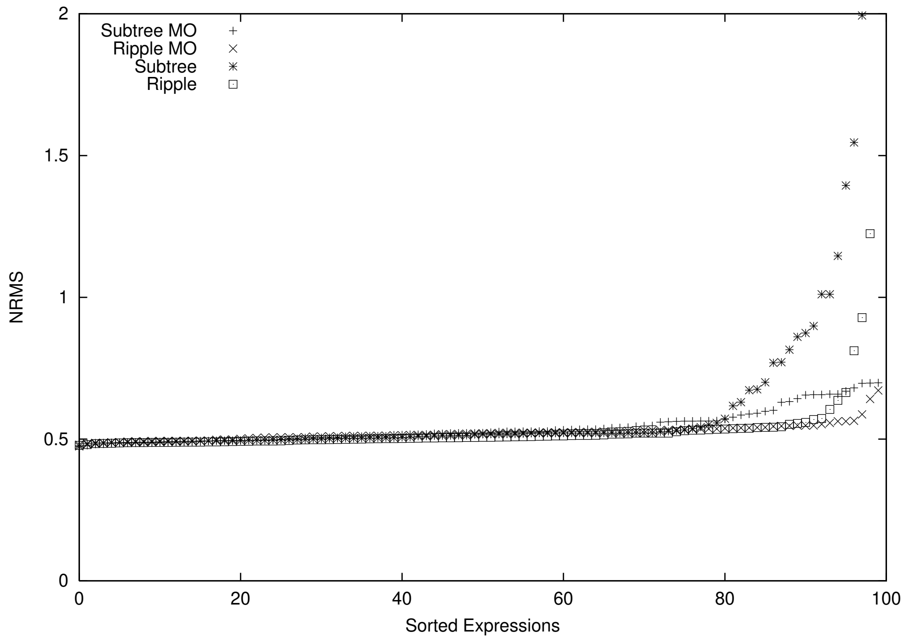


Figure 7.6: Performance of the selected expressions of the multi-objective and single objective runs on the testset. The error measure that is used is the normalized RMS error, which is defined in such a way that an error of 1 is achieved by a correct prediction of the target average.

can be called destructive. The use of a second objective that selects expressions on their ability to be valid in a prespecified output range seems to help in reducing the level of destructive overfitting behaviour.

Concluding the Experiment

The interval program used here is a general approach to symbolic regression where the issue of protected mathematical operators is solved. If the ranges on the input variables are set correctly, there is no possibility that mathematical exceptions occur.

If the bounds are set to theoretically known values, it is likewise guaranteed that the expressions induced by this program are well-defined for all possible input values.

Here an approach was examined where the correct range of the output interval was used as a second objective in a multi-objective search. The constraints, thus imposed, did not hamper the search for a well-fitting equation by much. It did however help in avoiding overfitted equations. This is to be expected as the outputs of the expressions that are induced in this way are guaranteed to lie within a certain data range.

The use of interval arithmetic in symbolic regression can take many forms. Primarily it can take care of avoiding the induction of destructive under and overflows in the input-output mapping. By guiding the search to find expressions that produce outputs in the appropriate data range, destructive overfitting can be avoided. The resulting expressions are then provably in the right range for all possible inputs.

A multi-objective approach might also not be the only approach that can achieve this. As often, a tailor made wrapper can help. By using the program for interval arithmetic, the output bounds of each induced expression are known. It is then a fairly simple matter to find a slope and an intercept such that the bounds of the expression coincides with the desired output range. This can for instance be achieved with the following clause:

```
scaled_interval(I+S*X) :-
  interval(X,L,U),
  catch(S is 0.55/(U-L),_,fail), % 0.55: new range
  I is - L*0.55/(U-L).
```

Which will induce expressions that are always in the right domain. It is up to the genetic algorithm to find an expression subject to this scaling that will fit the data well.

Although the constraints imposed by interval arithmetic are fairly simple and can be programmed into a regular genetic programming system without much trouble, the use of the logic programming representation made it possible to use it without any changes to the search engine. It shows the versatility of the approach, where a change in the definition of logic programs can help in finding more reliable solutions.

7.1.3 Units of Measurement

In the physical sciences, data represents careful observations of a physical system under study. Apart from the raw numbers that are collected, units of measurement of the observed variables provide additional information about the physical process. In Chapter 5 a method was proposed that utilized the information in the units of measurement in a preferential manner: a multi-objective strategy was used to minimize both the error on the data and the error in the dimensions of the evolving expressions. The expressiveness of the ALP system can however be used to declaratively constrain the search such that only dimensionally correct formulations will be considered. In fact, tackling problems involving units of measurement was the main inspiration for defining the ALP system.

A comparison between declarative and preferential methods of formula induction in the context of units of measurements can be found in chapter 8.

The problem used here involves the sediment transport problem, already encountered in Chapters 3 and 5. In contrast with the approach outlined in Chapter 5, the system is constrained to generate only dimensionally correct equations. It thus uses a declarative bias towards the use of units of measurements. Another approach for this class of problems is studied in (Ratle and Sebag, 2000) where a context free grammar is generated that models a subset of the language of units of measurement.

Implementation of the units of measurement system in a Logic Program

The constraints imposed by the units of measurement can be effectively implemented in a logic program. In order to implement the system a predicate `uom/2` is defined. The first argument of the predicate provides the algebraic expression and the second argument a list containing the exponents of the units such that the fact 'uom(ws,[1,-1])' for example denotes an input variable ws stated in the units of length over time: a velocity.

For addition and subtraction, the program needs to ensure that both arguments are of the same type, thus:

```
uom(X + Y, UOM) :-
   uom(X, UOM),
   uom(Y, UOM).
```

Although this clause can be used for both modes (input and output) of the second (UOM) argument of the predicate, the behaviour of the ALP system will be different for each mode. If the UOM argument is bound to a ground term i.e., it specifies the need for a particular unit of measurement, the recursive calls to find the arguments X and Y will be constrained to be stated in these measurements. If on the other hand the UOM argument is not grounded, the recursive procedure to find the X part of the addition is unconstrained. However, after the induction of this first part, the UOM argument will have been bound to a ground term: the units of the first argument. The search for the Y part of the expression is then constrained to be stated in the same units as the first part of the expression. In this case no special cases needed to be considered, but below the meta-logical predicate ground/1 will be used to check for groundedness or non-groundedness of the inputs.

For multiplication and division, two clauses need to be defined: one when the UOM variable is defined (grounded) and another when it is not defined. This is necessary because in the case when the units are known beforehand, a different calculation needs to be performed then when they are induced by the arguments of the expressions. It was chosen to implement this using a helper predicate multiplication/3, to make it possible for the genotype to code for the operation, and subsequently let backtracking help in choosing the appropriate clause. The following set of clauses implements multiplication (implementation of division is similar):

```
uom(X*Y, UOM) :-
   multiplication(X,Y,UOM).

multiplication(X, Y, UOM) :-
   ground(UOM),                     % is UOM set to a value?
   uom(X, UOMx),                    % get the units for the first argument
   call(minus(UOM, UOMx, UOMy)),    % calculate the units for the
                                    % second argument
                                    % such that y = output - x
   uom(Y, UOMy).                    % constrain the expression
                                    % to be of units UOMy

multiplication(X, Y, UOM) :-
   not(ground(UOM)),                % is UOM unknown?
   uom(X, UOMx),
   uom(Y, UOMy),                    % Do not constrain the units
   call(plus(UOMx, UOMy, UOM)).     % Calculate the output units:
                                    % output = x + y
```

The predicates minus/3 and plus/3 perform subtraction and addition on lists of exponent values. They are wrapped inside Prolog's built-in call/1 predicate to make sure that they are evaluated in Prolog directly (See Section 6.2.5). The ground/1 predicate checks whether the variable is bound to a ground term. In the first clause, that only applies when the units are grounded, the first argument for the multiplication is found in an unconstrained way. Subsequently, the difference between the output and this argument's units is calculated, the result are the units the second argument needs to be stated in to obtain a consistent formulation. When no units are demanded, both the first and the second argument are induced without constraints on the units they're stated in. They are added together to calculate the output units. The two clauses for applying multiplication are thus defined for different modes.

In the experiments described below one additional function is used, sqrt/1, defined as:

```
uom(sqrt(X), UOM) :-
   square_root(X,UOM).

square_root(X, UOM, C) :-
   ground(UOM),                  % is UOM set to a value?
   call(mult(UOM, 2.0, UOMx)),   % multiply by two
   uom(X, UOMx).                 % constrain the operand

square_root(X, UOM, C) :-
   not(ground(UOM)),             % is UOM unknown?
   uom(X, UOMx),                 % find an operand (unconstrained)
   call(mult(UOMx, 0.5, UOM)).   % calculate the output UOM
```

Where the mult/3 predicate calculates a multiplication with a scalar value.

Together with clauses defining the variables and retrieving constant values (which are constrained to dimensionless units in order to disallow arbitrary coercions), this logic program implements the system in full generality. The ALP system evolves paths through the logic program that result in correctly typed expression, which are subsequently subject to evaluation on the available observations.

The Sediment Transport Problem

For the first set of experiments the sediment transport problem is revisited. The inputs for this program are the independent variables, together with their dimensions, represented as a list of exponents.

```
uom(X,U) :- leaf(X,U).

leaf(nu, [2,-1]).    % kinematic viscosity
leaf(uf, [1,-1]).    % shear velocity
leaf(uf_p, [1,-1]).  % sheer vel. related to skin friction
leaf(ws, [1,-1]).    % settling velocity
leaf(d50, [1,0]).    % median diameter of sand grains

leaf(9.81, [1,-2]).  % gravity acceleration
leaf(C, [0,0]) :-    % dimensionless constant
  fp(C).             % obtain float from genotype
```

This logic program declaring the **uom** then defines a search where most constraints are context-sensitive. Because of this, subtree crossover will find it difficult to exchange subtrees, as it does not have a fallback mechanism in case constraints are violated. Apart from the case where there is no desired output unit and the entire expression consists of non-linear operators, subtree crossover will only be able to exchange subtrees that are stated in the same units. It will then be constrained to only search in the space of units that are present in the initial generation.

The untyped ripple crossover however will be able to exchange expressions stated in arbitrary units. The backtracking mechanism helps in reinterpreting the remainder of the string to induce a correct formulation. By using the helper predicate `multiplication/3`, `division/3` and `square_root/2` that define the operation of the program in the case of differing modes, the backtracking mechanism will work as intended. When a multiplication is chosen for instance, selecting the clause for the wrong mode (for instance the clause that checks for groundedness when the UOM is ungrounded), would only lead to skipping a single codon, the next codon will automatically code for the proper clause.

It is unfortunately not clear what consequences breaking up the program in several predicates has for subtree crossover. Subtree crossover will exchange subtrees starting with the same predicate, and thus when selecting a `multiplication/2` predicate in one tree, it will search for the same predicate in the other tree. When such a tree is found, but the modes of the predicates differ, the crossover will fail. The selection of subtrees is done randomly, thus the probability of selecting a predicate will be determined by its frequency of occurrence in the tree. Whigham (Whigham, 1996a) advocates setting a priori probabilities for the selection of different non-terminals (predicates) in a context-free grammar. This would however add another set of parameters to the system. The impact of these parameters on the search is difficult to determine, as it is quite likely that their importance depends on the state of the population.

This program is designed with the possibility of backtracking in mind and is thus tilted in favour of the untyped variational operator called ripple crossover. The main drawback for typed subtree crossover is however not easily solvable: due to it mostly exchanging subtrees of the same type (units), it will be heavily biased towards the types present in the initial population. The units of measurement language has an infinitude of possible types, leading to the assumption that subtree crossover will be ineffective in this domain. This a priori drawback of subtree crossover on the unit of measurement problem was the primary motivation that lead to creating the ALP system. With this system in place, together with an implementation of subtree crossover, it is finally possible to check whether this assertion is founded.

The desired output for this problem is a dimensionless quantity, a concentration. Two experiments were performed, one where the desired output is given and one experiment where no desired output is given. The second experiment thus seeks for a dimensionally consistent formulation stated in any units. It is quite common for empirical equations to multiply the resulting equation with a constant stated in some units to obtain an equation stated in the desired units of measurement[5], this is usually a residual coefficient that tries to describe some unmodelled phenomena. In that case the search is for an expression that is internally consistent, without necessarily inferring the desired dimension. In the logic program this is accomplished with the following goals:

```
(1) ?- uom(X,[0,0]).
(2) ?- uom(X, _).
```

Where the underscore symbol '_' denotes an unnamed variable. The experiments were run for 300 generations.

From Figure 7.7 and Figure 7.8 it can be inferred that subtree crossover is not capable of optimizing well on this problem. In contrast with this, ripple crossover searches reasonably well, the average performance is in both experiments significantly better than subtree crossover, even the less constrained subtree crossover experiment is not able to improve upon the fully constrained ripple crossover runs.

On the problem setup where the expression can be stated in arbitrary units (i.e. the goal statement is of the second form), ripple crossover is capable of outperforming the human-induced equation on *average*[6].

Settling velocity of Faecal Pellets

To investigate if the lack of search capabilities when employing a subtree crossover is structural or coincidental, a second experiment is performed using a different dataset. The object of search in this case is to find a dimensionally correct expression that describes the settling velocity of faecal pellets. This problem is defined with the following variables and query:

[5]A famous example is Chezy's roughness coefficient, stated in the unit $m^{1/2}/s$.

[6]This is however the performance on the training set only.

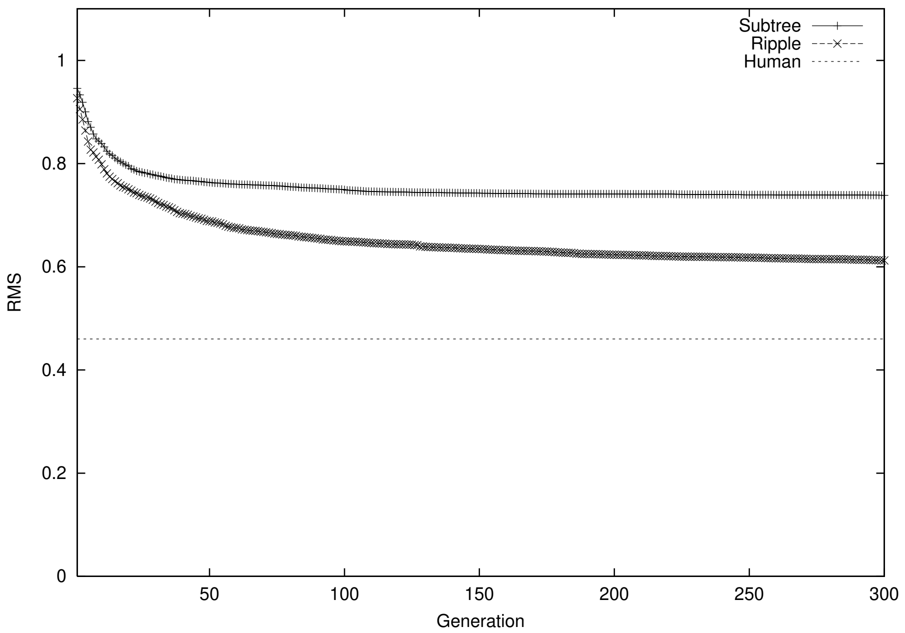


Figure 7.7: The evolution of the average performance for subtree crossover and ripple crossover on the sediment transportation problem. The problem setup requires the expressions to be stated in dimensionless units. The performance of a human-proposed alternative formulation is depicted with a straight line.

```
leaf(l, [1,0,0]).        % length of the pellet
leaf(w, [1,0,0]).        % width of the pellet
leaf(rhos, [-3,0,1]).    % density of sea water
leaf(flrhos, [-3,0,1]). % density relative to fresh water
leaf(9.81, [1,-2,0]).    % gravity acceleration
leaf(X,[0,0,0]) :-       % dimensionless constant
  fp(X)

?- uom(X,[1,-1,0]).
```

than the sediment problem described above purely from the perspective of obtaining legal expressions. Not only is the dimension of mass added, there's no obvious way to manipulate the variables in length units with the variables stated in density units. In order to produce a valid expression, the gravity acceleration term must be used. The simplest expression that abides all constraints is of the form $\sqrt{gl}$ or $\sqrt{gw}$. The only degree of freedom that is allowed is multiplying this basic expression with an arbitrary expression stated in dimensionless units.

Due to the highly constrained nature of this problem, it was necessary to increase the depth at initialization to 8. This to allow the creation of 500 unique individuals in the first generation. Figure 7.9 shows the average performance of the two systems. Again, subtree crossover gets stuck at a suboptimal performance fairly early in the run.

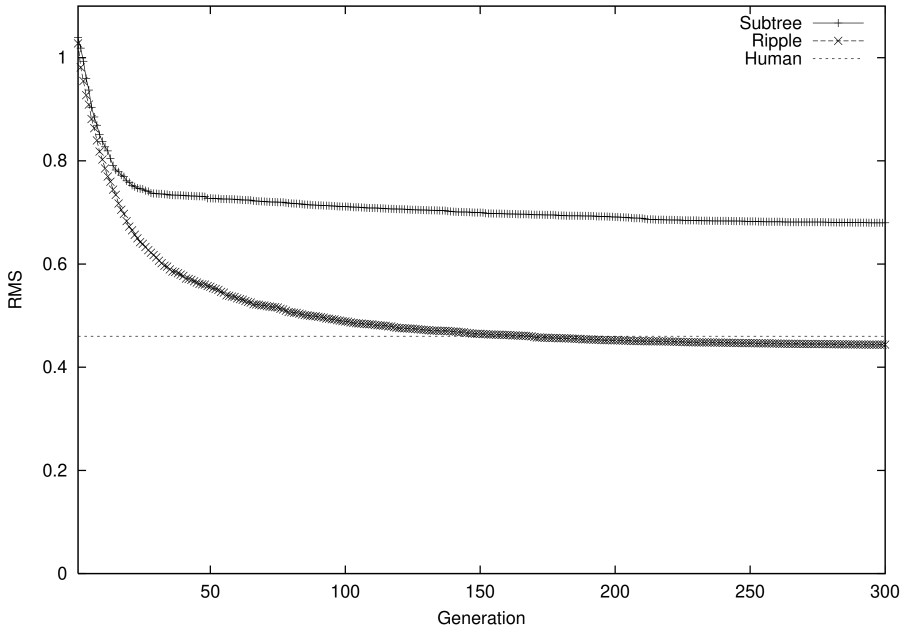


Figure 7.8: The evolution of the average performance for subtree crossover and ripple crossover on the sediment transportation problem. The problem setup allows the expressions to be stated in any units, as long as they are internally consistent. The performance of a human-proposed alternative formulation is depicted with a straight line.

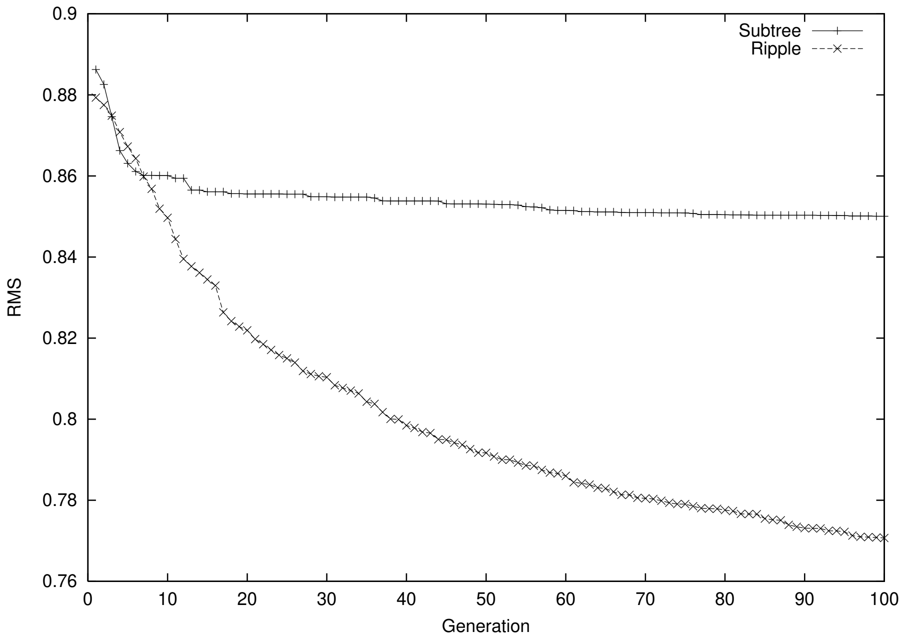


Figure 7.9: The evolution of the average performance for subtree crossover and ripple crossover on the settling velocity of faecal pellets problem. The problem setup requires the expressions to be stated in velocity units.

To appreciate the difficulty in searching in this constrained space, consider the failure rates depicted in Figure 7.10. For both crossovers, failure rates are high. Interestingly enough, the failure rate of subtree crossover is much lower than the failure rate of ripple crossover. This quite obviously does not mean that subtree crossover searches more effectively as it's final performance is much worse than that of ripple crossover.

Inspection of the resulting expressions for the subtree crossover runs showed that more than half of the expressions produced at the end of the run are of the particular form $\sqrt{gw + gw}$. These runs converged on this expression fairly rapidly as legal crossovers on this structure produce very often clones. As there was no mutation present, this expression forms a local optimum in the search. The runs employing ripple crossover were able to induce a diversity of well-performing expressions.

Concluding the Experiments

It has been verified that the ALP system in combination with ripple crossover is capable of searching in the area of dimensionally correct equations. The use of a subtree crossover is however problematic. The existence of a great diversity of possible types in this type of problem prevents subtree crossover from searching well.

Again, no mutation was used. As ripple crossover is capable of creating new types, whereas subtree crossover is not, the comparison is not completely fair. However, implementing a typed mutation in this highly constrained set of possible expressions

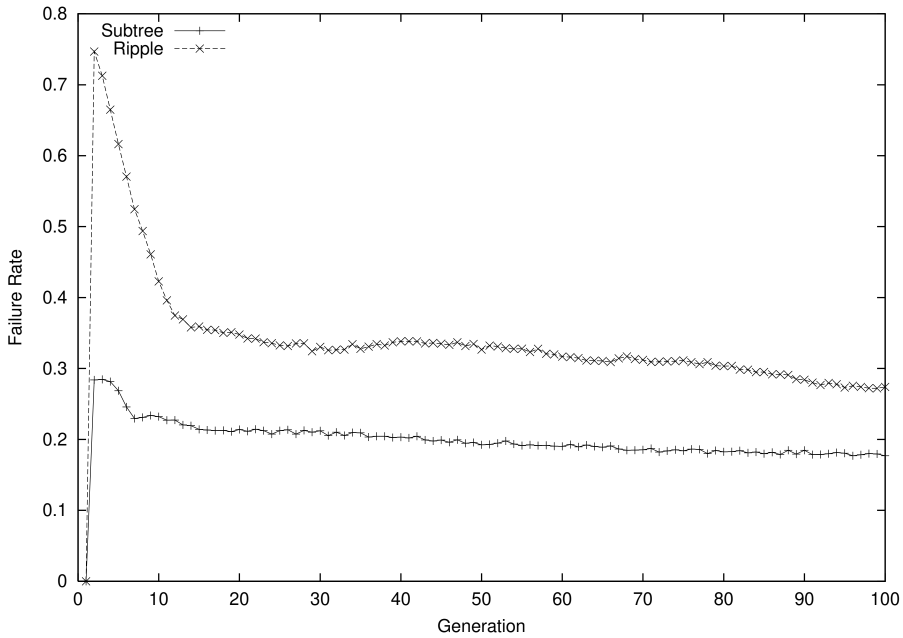


Figure 7.10: The evolution of the failure rate for subtree crossover and ripple crossover on the settling velocity of faecal pellets problem.

is not without problems itself. A typed subtree mutation routine would involve the fresh initialization of a subtree starting from a specific context. For the pellets problem, this initialization was in itself a non-trivial task. Experiments with an initialization operator used as a mutation showed that a significant runtime penalty is associated with using this operator. It relies heavily on backtracking to find an expression in the constrained domain and suggests the use of extra parameters to control the speed and quality of the operator.

Ripple crossover has a high mutation flavour. Individuals undergoing ripple crossover are subject to re-interpretation of genetic material and have a tail of unexpressed code that can become used again. This tail is a store of genetic material that at least in one context has lead to a finished expression. The runtime penalty for using the tail is lower than that of using a strongly typed mutation operator. The strings involved are always finite: if the end of the string is reached during the derivation process, the individual is marked invalid. As the individuals here are initialized without a tail, this tail is formed exclusively from genetic material that has undergone selection. In particular this means that the string of integers in the tail encode for at least one finishing derivation in one particular context. Revisiting that context enables the re-creation of that expression.

7.1.4 Matrix Algebra

Matrix algebra defines a set of very powerful mathematical expressions. It allows grouping of for instance spatial and temporal data into a small set of variable symbols. There have been a few attempts at inducing expressions in matrix algebra by means of evolutionary computation (Montana, 1995; Martin et al., 1999), but either no experimental results were given (Martin et al., 1999), or the proposed implementation did not implement the constraints in a general fashion, and were able to only induce expressions within a given set of matrix dimensions (Montana, 1995).

The use of logic programming in specifying the constraints allows writing a program that is capable of inducing correct matrix expressions given arbitrary sized input matrices.

The problem studied here is a problem in the induction of a rainfall-runoff model for the Orgeval river, located in France. The model uses past precipitation and tries to predict the discharge, the amount of water flowing through the river at a certain point in time. This data was sampled at hourly intervals. It was chosen to limit the precipitation data to one week of observations prior to the predictions. In effect this means that there are $7 \times 24 = 168$ inputs for each prediction.

In this setting, regular symbolic regression would have to cope with 168 terminals, even disregarding the use of moving averages that are sensible to add in a preprocessing step (See for instance (Whigham and Crapper, 1999) where some moving averages were introduced for a rainfall-runoff application). In contrast with this, using the language of matrix algebra, these 168 terminals are replaced by a single terminal: a vector of 168 observations. The system is allowed to index the vector and to apply summation and averaging operators, leading to the inclusion of all potential moving averages and lump sums in the space of possible programs. The use of vectors and matrices allows a concise symbolic formulation of a solution involving all input variables.

The operations that are used are split in several groups:

1. matrix algebra (sum, matrix product, vectorized product)
2. aggregating (sum, mean)
3. concatenation
4. indexing (select a range of values from an input variable)
5. non-linear unary operators (sqrt, log, exp)

The implementation of these operators is non-trivial, and the ALP system is pushed to its limits in accommodating for them. Here we will discuss the key clauses. The main predicate is `matrix/3`. As usual, the first argument to the function is the symbolic expression that is to be generated. The second an third arguments are the dimensions of the matrix: the number of rows and the number of columns. As we are interested in the a single value as output, the toplevel query will be `matrix(X,1,1)`.

There are a number of binary functions defined that only work properly on matrices of the same size. These are addition and subtraction, but also element-wise multiplication and division. These all have the form:

```
matrix(f(X,Y), R, C) :- matrix(X, R, C), matrix(Y, R, C).
```

where f is the function in question (the function name f does not appear in the logic program, it is used here to indicate a number of clauses, all varying on this same theme). For matrix multiplication the number of rows of the first arguments needs to be equal to the number of columns in the second arguments. In a clause this becomes:

```
matrix(X * Y, R, C) :- matrix(X, R, N), matrix(Y, N, C).
```

The output dimensions are thus the outer dimensions of the input matrices. The inner dimensions are unconstrained, but once set (by inducing an expression for X), they function as a constraint. There is an exception to these rules: when one of the matrices is a scalar, all operations are again allowed:

```
matrix(f(S,X), R, C) :- matrix(S, 1, 1), matrix(X, R, C).
```

Where f now ranges over all functions defined so far.

Aggregating values by applying an average or a summation is fairly intricate. It was tried to mimic the language used by specialized matrix algebra languages such as Matlab and Octave. In these language an average operator is defined so that if both the number of rows and columns are larger than one (i.e., it is a matrix), it will produce a row vector, applying the aggregation operator to the individual columns, while if the input is a vector it will return a scalar value. The logic program was enhanced to accommodate for these special cases.

The transpose of a matrix is simply defined as

```
matrix(transpose(X), R, C) :- matrix(X, C, R).
```

Matrix based languages usually also accommodate concatenation of smaller matrices such that they form larger matrices. This is again only allowed when a regular matrix can be formed. Thus depending on the type of concatenation — vertical or horizontal — either the rows or the columns of the two arguments need to be the same. Concatenation and accompanying constraints are implemented in the program by checking these requirements.

Indexing the variables can be used to create moving averages. The syntax for a moving average of the first 20 elements with a stepsize of 2 for a column vector prec would be mean(prec(1:2:20,:)). The dimensions of this vector would be 10×1. A fairly intricate mechanism is used here to make sure that a variable is selected and that the indices selected are in the proper range. Also here special cases for different modes of the matrix dimensions are used.

Because such moving averages can be seen as an extension of the terminal set, an additional mechanism is used that changes the program while deriving the expression, such that at any point, the system can add such an indexed expression and subsequently re-use it. This mechanism uses the meta-logical function assert/1, that can add clauses to a logic program. These are thus automatically defined terminals, and the algorithm is then capable of changing the program while running it.

Nonlinear operators such as $\log$ and $\exp$ are included as well. As they do not manipulate the dimensions, but are just applied elementwise, no special care has to be taken to use them. If this application leads to mathematical exceptions, the expression is marked invalid. There is thus no interval arithmetic applied here.

The overall program defining the matrix expressions is quite intricate and although it works reasonably well, efforts are underway to improve upon it.

The system is trained using a sequence of 1000 hourly measurement points where the only input variable is defined as:

`matrix(prec,168,1)`

Identifying the precipitation of the last week such that `prec(1,1)` is the current precipitation and `prec(168,1)` is the precipitation one week in the past.

Results

One of the best performing expressions on the training set was:

```
-46.281 + 17.157 * exp(mean(prec).*
  mean([ prec(7:1:57);
  sum([prec(64:9:151);
  sum(prec(3:1:27))])]))
```

Due to its shortness, it was selected and it achieved admirable performance on the testing data. The program critically makes use of the square bracket operator that denotes concatenation. The semi column means that the concatenation is performed over the rows. The inner term of `sum([prec(64:9:151) ; sum(prec(3:1:27)])` calculates a sum on a vector of dimensions 11×1, the first 10 elements are taken from the range starting at 64 hours in the past with stepsize 9, while the 11th and final element is in itself a sum of the more recent rainfall. The nested sums are a summation of $10 + 24 = 34$ different rainfall observations, that are concatenated to the `prec(7:1:57)` term. The mean that is calculated of this vector of dimension 51×1 thus includes as its 51st term, these 34 observations lumped together. It calculates a weighted average with the short term rain fall and the long term rainfall being more important than the medium term. The whole average is multiplied with the mean rainfall in the week before and exponentiated to give the final prediction.

To appreciate the difficulty in creating such a model it is instructive to compare the graph of the precipitation with the predicted and actual flows in Figure 7.11. Although there's an obvious connection between rainfall intensity and the subsequent runoff in the Orgeval river, it is by now means a straightforward relationship due to the spiked distribution of precipitation. In particular, the model is capable of predicting the peaks with admirable accuracy. This is important when forecasting floods. The method introduced here shows promise in inducing equations that can model such relationships. It does this without preprocessing of the data other than setting the maximum history (a week).

The expression that is described above provides an additional view of the main characteristics of the Orgeval catchment. The moving averages and moving sums propose some specific intervals of observing the rainfall for the fast flow of the

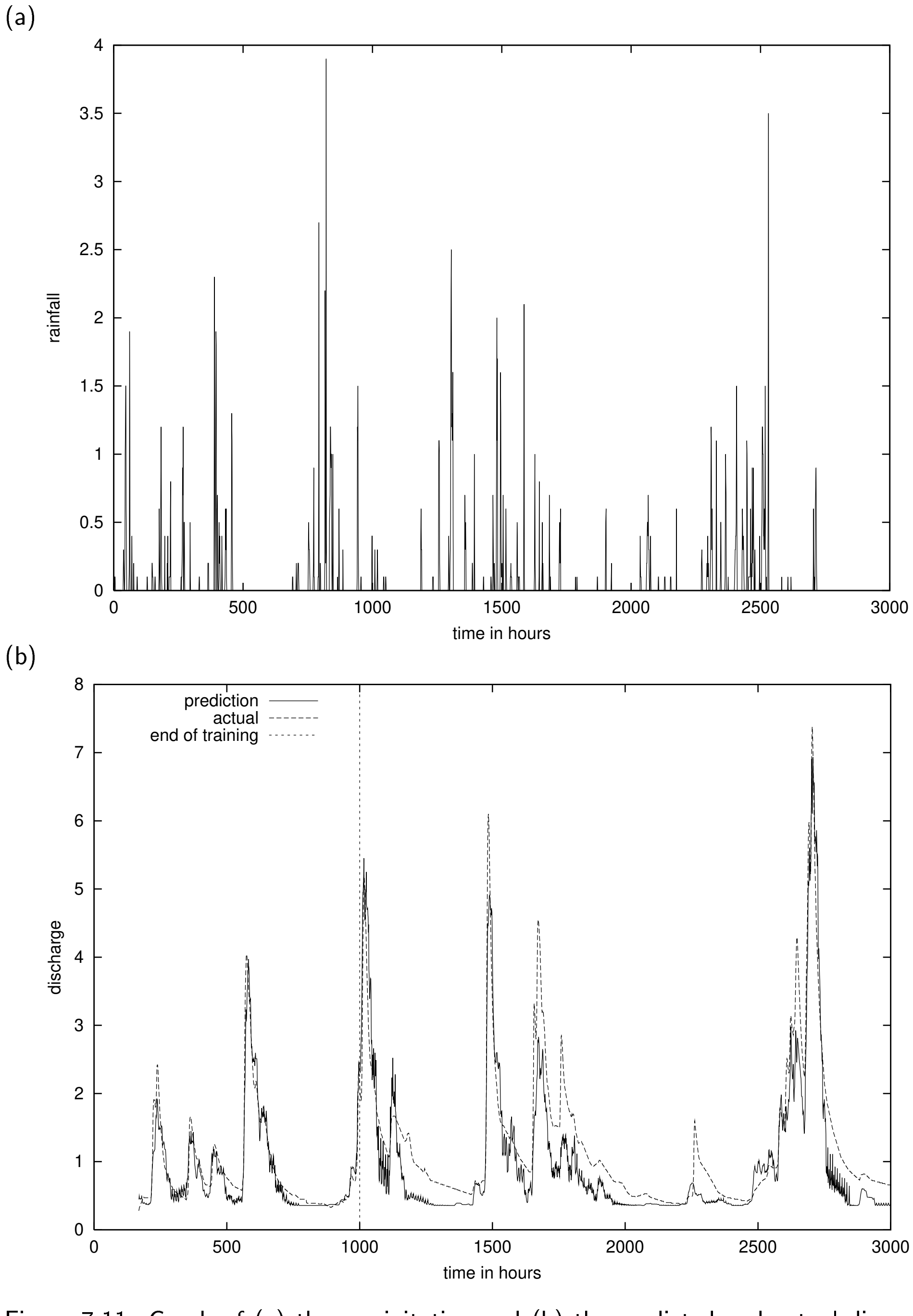


Figure 7.11: Graph of (a) the precipitation and (b) the predicted and actual discharge on the rainfall-runoff problem.

catchment (from 3 hours to 27 hours in the past), the medium flow of the catchment (7 hours to 57 hours), and the low flow (larger than 64 hours in the past). Breaking up the flows in a catchment in a couple of such 'reservoirs' is standard practice in building a conceptual model. Here this breakup in the number of reservoirs and their temporal effect on the runoff of the river has been automatically found.

Concluding the Experiment

The experiment performed for the rainfall-runoff problem is a first experiment into the domain of matrix algebra using the expressiveness of the ALP system. This experiment already provided a reasonable performance, but it is conjectured here that better performance is possible by including some domain knowledge. Hardly any of the runs used the matrix product. It seemed not necessary to use this to induce the models. It was included in the function set because it was thought that it would make it possible to perform filtering. It did not seem to be helpful. A possible avenue of further research might be to simplify the language to first let the evolving programs set up some moving averages, lump sums, maxima and minima based on indexing the available data, and subsequently let it induce a symbolic expressions that combines these proposed aggregate variables. A second approach could be to perform a simple linear regression on these variables.

These approaches are however left for future work. The experiment described here showed that it is possible to induce expressions in the very promising area of matrix manipulations.

7.2 Discussion

The ALP system was originally developed to be able to search in the area of dimensionally correct expressions. The use of untyped variation operators was hypothesized to be necessary to be able to search in this area. The experiments performed on two real-world problems in this area in Section 7.1.3 seem to confirm this.

The main problem with a typed crossover operator lies in its non-explorative nature in the space of types: it is only capable of swapping subtrees that are of exactly the same type: in the case of typing with units of measurement, the number of possible types is very large and subsequently the number of legal crossover points can drop dramatically, leaving subtree crossover to focus on a small region of the search space. The experiments did not employ a typed mutation operator to create new subtrees and possibly new types. The reason for not including such an operator lies in the runtime performance penalties associated with this operator.

Ripple crossover works robustly on these problems. However, it cannot be ruled out at this point that its main search capabilities come from it being simply a good global randomization operator. Even in that case, one advantage of a ripple randomizer is that the genetic material used in this randomization are contained in the genotype. The computational effort in creating new solutions is then known. There exists some selection pressure on this 'randomizing' material to encode for a computation that finishes before it reaches the end; genotypes that do not have this ability will on average produce more failures and will have greater difficulty in multiplying. In domains where initialization is already a non-trivial problem in itself this property of encoding for completing the resolution process is important in its own right.

The experiments in this chapter tried to highlight these differences between the use of a typed and an untyped crossover operator. It also showed that an untyped operator such as ripple crossover is feasible to use as a variation operator in highly constrained domains. It is however left for future work to implement and investigate the use of typed mutation operators to help in searching more efficiently.

7.3 The Art of Genetic Programming

In the book "The Art of Prolog" (Sterling and Shapiro, 1994) Sterling and Shapiro state that writing an elegant and powerful Prolog program is a skill that can only be learned through practice. Recognizing a concise logic program is one thing, writing one quite another.

Compared to the programs used for the ALP system, writing a program for execution in Prolog is comparatively easy, as the execution path of the system is deterministic. For the ALP system, this does not hold. The genetic programmer using the ALP system has to take into account that the programs will be executed using any path through the search tree, and also that these paths are finite. Also the effects of backtracking need to be considered in writing a logic program for generating computer programs. Fortunately, when deterministic calculations are needed, a direct call can execute statements in the Prolog resolution model. This allows to use the efficiency of Prolog whenever that is needed. Determining what to calculate deterministically and what to generate by the genetic algorithm is another issue in writing a logic program in the ALP system.

In this chapter several logic programs were used, some with more success than others. The program for the sensible ant was maybe fortunate: the way the constraints were imposed reduced the search space considerably. The program for interval arithmetic was relatively straightforward, but transforming constraints on the range for the output to something manageable for a genetic algorithm involved a multi-objective search. It also showed the convenience of the system in setting up wrappers.

The program for generating expressions that are valid in the language of units of measurement and in the language of matrix manipulations involved creating clauses for multiple modes: clauses that handle the case when the values of the attributes are known (grounded) and clauses when the values are unknown. It is expected that especially the matrix algebra program will undergo considerable refinement in the future. An interesting extension to this work would be the combination of interval constraints, units of measurements constraints and matrix constraints in a single program, in order to induce short expressions that are mathematically stable, dimensionally correct and can be applied to massive amounts of data.

Investing time in setting up a program such that the ALP system can search effectively can be very rewarding: these programs are used to generate expressions in *problem domains*, not just for single problem instances. The art of genetic programming lies in the declaration of a generative program that is optimally attuned to the underlying genetic algorithm. Once such a program has been written, subsequent application of the system involves setting up some specific information for a problem instance: the variables and desired outputs. It is expected that both the genetic algorithm and the programs themselves need to undergo considerable refinement.

The experiments and discussion presented in this chapter indicate that the combination of logic programming to define problem domains and a genetic algorithm to find optimal expressions for problem instances is a sensible approach to the problem of automatic induction of computer programs to solve hard problems.

Chapter 8

Experiments in Scientific Discovery

Chapter 4 outlined the main goal of the work: to use the information in the units of measurement in a problem description to make the expressions produced by genetic programming more amenable to interpretation and analysis. To achieve this two main approaches have been defined: Dimensionally Aware Genetic Programming (Chapter 5) that uses a *preferential bias* towards dimensionally correct expressions; and an Adaptive Logic Programming system (Chapter 6), where a *declarative bias* is implemented that can reduce the search space to only those expressions that are dimensionally correct. The logic program that implements the language of dimensionally correct expression was presented in Chapter 7.

In order to intercompare performance a number of experiments are carried out, using the following settings:

- Strongly Typed (STGP), using the program described in Section 7.1.3.
- Dimensionally Aware GP (DAGP), using a program to induce symbolic expressions and the coercion calculation described in Section 5.1.1.
- Symbolic Regression GP (SRGP), using the same program as DAGP, but without the coercion calculation.

These three settings have each been applied to four different problems, all using the ALP system. Each problem involves a largely unsolved scientific problem in its own right. The problems are described below. The four problems have a rather diverse range of specifications: the available data vary from very sparse (57 cases) to abundant (4800 cases), while the specification of the **uom** varies from poor to descriptive. These choices were driven by the intention to examine the robustness of the methods and the quality of the provided solutions.

8.1 Problem 1: Settling Velocity of Sand Particles

The settling velocity of sand grains is an important parameter in the study of sedimentary processes in a coastal environment. A number of different settling velocity equations have been presented in the literature. This case study concentrates on the settling velocity of sand grains.

Background

Sand grain settling velocity data has been gathered and presented by Hallermeier (Hallermeier, 1981). The data (see also Table 8.1) consists of the sand grain diameter d, the fluid kinematic viscosity ν , and the relative density defined as $\gamma' = (\rho - \rho_f)/\rho_f$. The data were organized in different ranges of the calculated non-dimensional Archimedes Buoyancy Index, defined as $A = \left(\frac{\rho}{\rho_f} - 1\right)\frac{gd^3}{\nu^2}$. It is quite obvious that only d and ν represent raw observations, whereas γ' and A are derived from other raw observations (such as ρ and ρ_f) which were not directly available. It should be noted that such a preprocessing of raw observations into derived quantities inevitably introduces a degree of bias. The authors opted to use raw observations and to avoid the use of derived quantities whenever possible. However, in this case the observation of the density of sand ρ is not available, and the derived quantities are employed instead. The data were limited to 115 different laboratory experiments.

variable	description	**uom**
d	sand grain diameter	cm
ν	kinematic viscosity	cm^2/s
A	Archimedes Buoyancy Index	dimensionless
γ'	relative density	dimensionless
g	gravity acceleration:	$981cm/s^2$
w_s	settling velocity	cm/s

Table 8.1: **uom** of the independent and the dependent variables for the problem of determining the settling velocity of sand.

Human proposed relationships

A large number of settling velocity equations for sand particles have been proposed. Here, we present only the most accurate one proposed by Hallermeier (Hallermeier, 1981). The Hallermeier equations were fitted using settling data involving fine to coarse sand grains. These equations (in the cgs unit system and for varying ranges of A) read:

$$
\begin{array}{ll}
w_s = \frac{gd^2(\rho-\rho_f)}{18\nu} & A \leq 39 \\
w_s = \left(\left(\frac{\rho}{\rho_f} - 1\right) g\right)^{0.7} \left(\frac{d^1.1}{6\nu^{0.4}}\right) & 39 < A \leq 10^4 \\
w_s = \left(\left((\frac{\rho}{\rho_f} - 1\right) \frac{gd}{0.91}\right)^{0.5} & 10^4 < A < 3 \times 10^6
\end{array}
$$

8.2 Problem 2: Settling Velocity of Faecal Pellets

The settling velocity of faecal pellets produced by marine organisms contributes to different oceanic processes including sedimentation rates, geochemical cycles and nutrient availability. Because faecal pellets are aggregates of smaller particles, the pellet sinking rates can be much larger than the rates of the individual particles. This increases the sedimentation flux and possibly the rate of particle deposition. Faecal pellets influence sediment transport processes in the benthic boundary layer, and an evaluation of faecal pellet settling rates contributes to the study of sediment mobility on the sea floor.

Background

Faecal pellet settling velocity equations have been presented in the literature for both pelagic and benthic organisms. Significantly larger pellets with higher settling velocities are produced by these benthic organisms. The present case study concentrates on faecal pellets of benthic origin produced by the benthic feeder *Amphicteis scaphobranchiata*.

Human proposed relationships

The existing settling velocity equations for faecal pellets of benthic origin are based on measured faecal pellet settling velocity data (Tahgon et al., 1984). Two main approaches are typically adopted by human analysts when approximating this data set: (i) either equations are fitted to the data or (ii) equations are based on the description of natural sedimentary processes.

A number of equations presented in the literature (Tahgon et al., 1984), (Komar and Taghon, 1985) have been fitted to the data. (Tahgon et al., 1984) calculated the nominal diameter (d_n) based on the equal volume sphere. They analyzed two separate groups of data: Group 1 (where $37 < Re < 178$) consisted of pellets produced by feeding on $< 61\mu m$ sediment fraction and Group 2 (where $45 < Re < 117$) consisted of pellets produced by feeding on 61- 250 μm sediment fraction. Here Re denotes the Reynolds number calculated as $Re = \frac{w_s d_n}{\nu}$. Taghon *et al.* (Tahgon et al., 1984) used a regression analysis to yield (in the cgs unit system):

$$w_s = 1.30d_n + \rho - 9.08 \tag{8.1}$$

(Komar and Taghon, 1985) used the pellets nominal diameter (d_n) to produce the following:

$$w_s = 0.275 \left(\frac{(\rho_{fl} g)^3 d_n^4}{\nu} \right)^{0.2} \tag{8.2}$$

where ρ_{fl} denotes the difference between the densities of fresh water and salt water. (Komar and Taghon, 1985) also found a relationship between the pellet settling velocity (w_s) and the settling velocity (w_t) of the 'equivalent' sphere *i.e.*, the sphere defined by the pellets nominal diameter. w_t was calculated either using (Gibbs et al., 1971) or (Davies, 1945). They presented the following (in cm/sec) using (Gibbs et al., 1971)

$$w_s = 0.824 w_t^{0.767} \tag{8.3}$$

and using (Davies, 1945)

$$w_s = 1.08 w_t^{0.686} \tag{8.4}$$

Although the original settling velocity data were measured in sea water it appears that (Komar and Taghon, 1985) used freshwater conditions in developing the above equations.

In addition to these equations which were purely fitted to data, a number of natural sedimentary particle settling velocity equations have also been developed. However, the accuracy of these equations is orders of magnitude worse than the fitted equations. A full inter-comparison falls outside of the scope of the paper and these equations are not analyzed here in further detail. Instead, the interested reader is referred to (Babovic et al., 2001) for a more thorough survey and discussion.

Data

The measured faecal pellet data (see Table 8.2) include length (l), width at widest point (w), density (ρ) and measured settling velocity (w_s) for each individual pellet. Settling velocities were measured in sea water.

variable	description	**uom**
l	pellet length	cm
w	pellet width at the widest point	cm
ρ_s	density of salt water	g/cm^3
ρ_{fl}	density difference between salt and fresh water	g/cm^3
d_n	nominal diameter	cm
csf	Corey shape factor $csf = \frac{w}{\sqrt{lw}}$	dimensionless
g	gravity acceleration	$981 cm/s^2$
w_s	settling velocity	m/s

Table 8.2: **uom** of the independent and the dependent variables for the problem of determining the settling velocity of pellets.

8.3 Problem 3: Concentration of sediment near bed

The description of the sediment problem can be found in Chapter 3, here only the inputs and desired output is repeated.

Name	**uom**	description
ν	m^2/s	kinematic viscosity
u_f	m/s	sheer velocity
u'_f	m/s	sheer velocity related to skin friction
w_s	m/s	settling velocity
d_{50}	m	median grain diameter
g	$9.81 m/s^2$	gravity acceleration
c_b	dimensionless	concentration of suspended sediment

Table 8.3: Dimensioned terminal set for the sediment transport problem.

8.4 Problem 4: Roughness induced by flexible vegetation

The influence of rigid and flexible vegetation on flow conditions is not well understood. Some laboratory experiments using physical scale modeling have been performed (Tsujimoto et al., 1993), (Larsen et al., 1990) but only over a limit range and with variable success. Similarly, although field experiments are continuing, the data availability remains poor.

More recently, a numerical model has been developed with the intention of deepening the understanding of the underlying processes (Kutija and Hong, 1996). This model is a one-dimensional vertical model based on the equations of conservation of momentum in the horizontal direction. This numerical model is used here as an experimental apparatus in the sense that this fully deterministic model is used as a source of data. It is here further processed in order to induce a more compact model of the additional bed resistance induced by vegetation.

The model takes into account the effects of shear stresses at the bed and the additional forces induced by flow through vegetation. For a more detailed description and a discussion of the specifics of the model, the reader is referred to (Kutija and Hong, 1996).

Data

The Kutija-Hong model, was in effect used as a truthful representation of a physical reality, while providing the conveniences of fast calculation and an ability to produce results with any degree of scale refinement.

The model has been run with a wide range of input parameters in order to create training data. Altogether, some 4800 training data were generated. The input data were varied in the ranges: 2.5 $\leq h_w \leq$ 4.0; 0.25 $\leq h_r \leq$ 2.25; 50 $\leq m \leq$ 350; 0.001 $\leq d \leq$ 0.004; 0.4 $\leq p \leq$ 1.0.

This problem is rather different from the three case studies already described. On one side this is the only problem where data were abundant (see Table 8.5). On the other side, even at the outset it is obvious that no observations about the flexibility of the vegetation are provided (for example in a form of Young's modulus of elasticity). So, it can already be stated that the problem is well covered by the observations, but not by the **uom**, since nothing can be postulated about the structural properties of the vegetation (whether it is grass, bamboo or a tree).

variable	description	**uom**
h_w	water depth	m
h_r	reed height	m
d	reed diameter	m
m	number of reed shoots per unit area	dimensionless
p	eddy-viscosity approximation and its relation to the vegetated layer height	dimensionless
c_{th}	theoretical value of Chezy's coefficient in the absence of vegetation	$m^{0.5}/s$
g	gravity acceleration	$9.81m/s^2$
c	measured value of Chezy's coefficient	$m^{0.5}/s$

Table 8.4: **uom** of the independent and the dependent variables for the problem of determining the Chezy roughness coefficient.

8.5 Experimental Setup

For all problems the same setup was chosen. As the basic strategy, the elitist, non-dominated sorting GA II (NSGA-II) (Deb et al., 2000) was selected. When a single objective is used, this algorithm reduces to an elitist $(\mu + \lambda)$ strategy. Table 8.6 specifies the various parameters used in the experiments. It was chosen to experiment with a minimal function set consisting only of basic arithmetic and a square root function, regardless of the data. Table 8.5 summarizes the availability of data. The error function used here is the normalized root mean squared error.

8.6 Quantitative Results

Comparing the three approaches is not a straightforward task, as we are concerned with two different objectives. On the one hand, an expression with a low generalization error is required. On the other hand, the expressions need to be interpretable in the symbol system used in physics. However, strict adherence to the **uom** system might not be the best approach in all cases, as the measured data may not provide

Experiment	Train	Test	Total
sand	78	37	115
pellets	38	19	57
sediment	171	86	257
roughness	3200	1600	4800

Table 8.5: Data availability.

Objective 1	Minimize the normalized RMS
Objective 2	Minimize the dimension error
Function Set	$\{+, -, \times, /, \mathrm{sqrt}\}$
	Either with or without type constraints
Population Size	250
No. of Runs	50
No. of Generations	400
Crossover Prob.	0.8
Mutation Prob.	0.2
Tournament Size	2
Strategy	elitist (250 + 250)
Multi Objective strategy	NSGA II
Size of Training Set	$\frac{2}{3}$ of the full set
Size of Test Set	remaining $\frac{1}{3}$ of the full set
Output type	Problem dependent

Table 8.6: The parameters for the experiments

a complete description of all relevant phenomena in the problem. Sometimes, when no adequate fit with a dimensionally correct formulation can be obtained, it might be more prudent to select a formulation that has a good fit and a non-zero coercion error. As DAGP produces a front of non-dominated solutions rather than a single solution, a selection from the front needs to be made. Therefore the following three post-processing rules are used:

- Select the best fitting equation (DAGP-FIT). This choice produces results that are comparable with the symbolic regression runs. No explicit interest in the dimensional correctness of the expression is enforced since the selected equations are from the high goodness-of-fit and low goodness-of-dimension edge of Pareto front. This choice is made in order to investigate whether the use of **uom** improves or hinders the search for finding well generalizing expressions.
- Select the best fitting equation with a coercion error of 0 (DAGP-DIM). This choice restricts the accepted solutions to dimensionally correct expressions only. These results are directly comparable to STGP.
- Select the best fitting equation with a coercion error smaller than 1 (DAGP-MID). This allows to tolerate a small coercion error if this helps the fitting capability.

These three post-processing rules mechanize the selection of expressions from the Pareto front, making comparisons more objective. Table 8.7 summarizes the comparison between the three DAGP post-processing rules and their adversaries. The table highlights systems that result in significantly better average performance on the test sets for the four problems. This is therefore a measure of the reliability of the method to produce good results.

The first column in Table 8.7 compares the two methods that are capable of producing dimensionally correct formulations. For the *sand* and *roughness* problems

Experiment	Correct (STGP DAGP-DIM)	Almost Correct (STGP DAGP-MID)	Not Correct (DAGP-FIT SRGP)
sand	DAGP	DAGP	none
pellets	none	DAGP	none
sediment	none	DAGP	none
roughness	DAGP	DAGP	none

Table 8.7: Results on the four problems, comparing the (50 run) average NRMS over the test set using a two-tailed t-test with a significance level of 5%. The label denotes the setting that was significantly better than its adversary, and 'none' when there was no significant difference. 'Correct' denotes the test using only dimensionally correct formulations. 'Almost Correct' denotes a test between STGP and the best fitting formulation on the training set that has a coercion error lower than 1. 'Not Correct' denotes the test between the best fitting expressions on the training set from DAGP and SRGP (Symbolic Regression GP), without any regard to the coercion errors the expressions made.

	STGP	DAGP-DIM	DAGP-MID	DAGP-FIT	SRGP
sand	0.36 / 0.36	0.32 / 0.28	0.28 / 0.25	0.27 / 0.24	0.25 / 0.24
pellets	0.73 / 0.57	0.80 / 0.60	0.64 / 0.44	0.62 / 0.44	0.57 / 0.74
sediment	0.63 / 0.65	0.60 / 0.62	0.49 / 0.56	0.42 / 0.55	0.41 / 0.55
roughness	0.48 / 0.48	0.40 / 0.40	0.30 / 0.30	0.29 / 0.29	0.26 / 0.27

Table 8.8: Comparison of the train / test errors. Average NRMS over 50 runs.

DAGP produces significantly better results, while for the others no significant difference is found. If a small dimension error is tolerated (DAGP-MID), DAGP performs equivalently to or better than STGP on all four problems. The comparison between DAGP-FIT and symbolic regression in Table 8.7 shows that neither produces significantly better results. The preferential bias in DAGP does not prevent finding good expressions. Figure 8.1 provides a plot of the performance on the test set of the 50 selected individuals.

In order to investigate the performance the different systems achieve on the test set, Table 8.8 summarizes the errors on both the training and the testing set.

8.6.1 Bias/Variance Analysis

Even a quick reference to Tables 8.7 and 8.8 reveals that DAGP seems to be a reasonable approach for this class of scientific discovery problems. In order to learn where this performance originates and to explain the fitting capabilities of the various typing systems, an additional analysis was carried out by decomposing the errors into bias and variance terms (Geman et al., 1992; Keijzer and Babovic, 2000a). Given N data points and M models, the decomposition is based on the following equality:

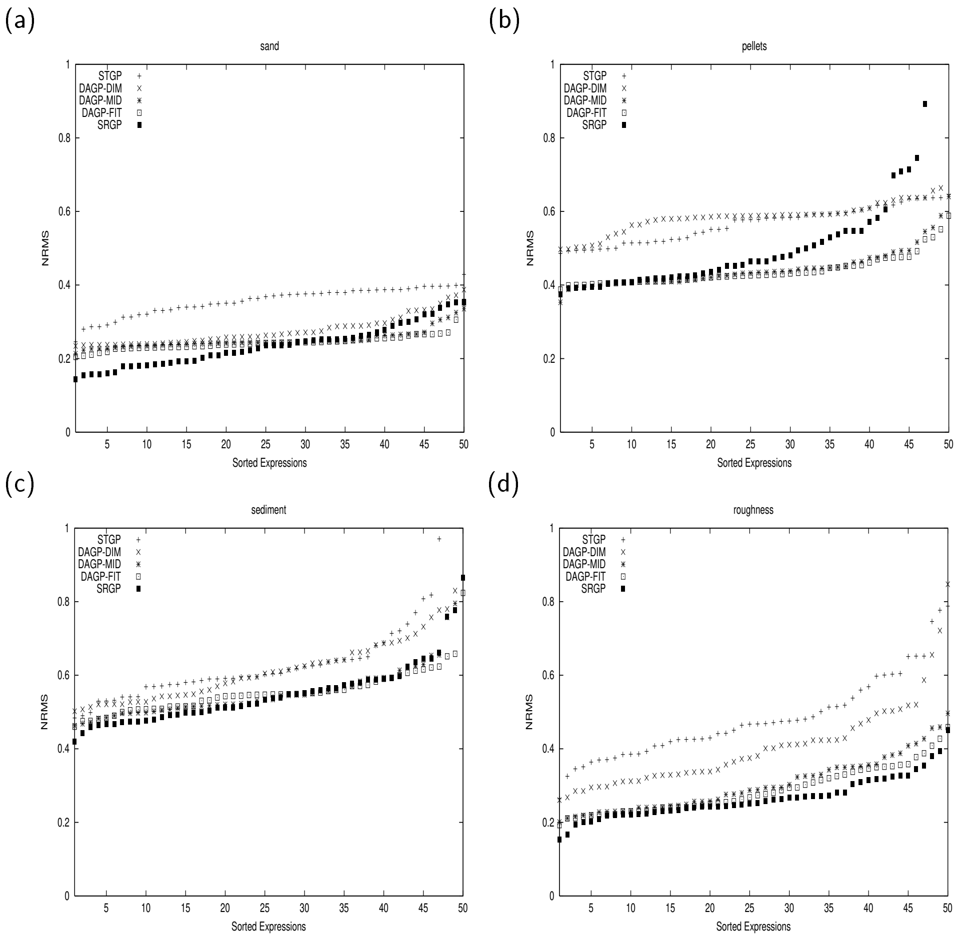


Figure 8.1: Overview of the 50 best performing expressions on the test set for the four problems described in the text. (a) Settling velocity of sand. (b) Settling velocity of faecal pellets. (c) Concentration of sediment. (d) Roughness induced by flexible vegetation.

$$\underbrace{\frac{1}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M}(t_i - y_{ij})^2}_{\text{mean squared error}} = \underbrace{\frac{1}{N}\sum_{i=1}^{N}(t_i - \bar{y}_i)^2}_{\text{bias}^2} + \underbrace{\frac{1}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M}(\bar{y}_i - y_{ij})^2}_{\text{variance}} \quad (8.5)$$

where t_i denotes the desired output, y_{ij} the ith output of the jth model, and $\bar{y}_i = 1/M\sum_j^M y_{ij}$ is the *average model* calculated as the average of all predictions for input i. The variance term does not depend on the desired output and measures the variability in the predictions of the expressions. The bias error measures the performance of the *ensemble*, and is an indication of the intrinsic capability of the method to model the phenomenon under study. The equality (8.5) is the empirical version of the decomposition; the theoretical version defined over infinite data includes a term addressing the noise in the data. In the empirical equation, the noise is absorbed by the bias term.

It is important to emphasize that the *error* due to bias is different from the bias of the system. The bias of the system entails the tendency to sample certain kind of solutions more regularly than others. A heavily biased method will have a small error due to bias if and only if the introduced bias is appropriate for the problem. With an inappropriate bias, the error will grow. An unbiased method on the other hand will always have a low bias error — however, it is the error due to variance that explains the fitting capability.

In an ideal setting, one can expect that biased methods generally have low variance error, while the level of bias error determines the appropriateness of the bias to the problem at hand. This can be exemplified by considering a maximally biased method that produces the same answer regardless of the data that is available. Such a method will always have zero variance error as its predictions are always the same. The performance of the system is then completely determined by its bias error, having a low bias error when the expression happens to fit the data, and having a large bias error otherwise.

An unbiased method on the other hand can be identified by a low bias error, though a significant variance error due to overfitting will remain. A prototypical example of such a method is a nearest neighbour method, that uses the value of the most similar point in the data to make a prediction. This method is unbiased as near the stored points the average of all nearest neighbour models will produce the correct answer. The error due to variance will however be equal to the noise in the data as each individual prediction will return a stored, noisy, data point. When predictions are made further away from the stored data, this error due to variance will increase.

However, these are the two extremes. In realistic circumstances the methods under study can exhibit various ranges of biased/unbiased behaviour. The STGP system is expected to have a significant bias, since it only samples dimensionally correct equations, and from these it tries to find the best fitting one. At the same time, the SRGP method is expected to have a low bias, due to the ability to fit the data in whatever way, using the set of functions that are available. DAGP tries to strike some middle ground by allowing incorrectly typed expressions to proliferate in addition to correctly typed expressions. The bias introduced in DAGP is thus hoped to be less stringent than the bias of STGP, possibly leading to lower bias error.

	STGP	DAGP-DIM	DAGP-MID	DAGP-FIT	SRGP
sand	0.23 (0.19)	0.07	0.06	0.06 (0.04)	0.04
pellets	0.28	0.30	0.19	0.19	10^4 (0.20)
sediment	0.34 (0.29)	0.30	0.26	0.27	0.24
roughness	0.14	0.10	0.06	0.05	0.03

Table 8.9: Errors due to bias, normalized using the variance of the target variable. Errors between brackets have been calculated using the middle 90% of the predictions, and are reported when they differ from the unprocessed values.

	STGP	DAGP-DIM	DAGP-MID	DAGP-FIT	SRGP
sand	0.41 (0.30)	0.05 (0.01)	0.06 (0.01)	0.14 (0.01)	0.94 (0.03)
pellets	0.11 (0.06)	0.08 (0.05)	0.07 (0.04)	0.07 (0.04)	10^6 (0.06)
sediment	0.26 (0.03)	0.09 (0.06)	0.06 (0.03)	0.13 (0.03)	6.33 (0.03)
roughness	0.12 (0.07)	0.04 (0.02)	0.02 (0.01)	0.02 (0.01)	0.04 (0.02)

Table 8.10: Errors due to variance, normalized using the variance of the target variable. Errors between brackets have been calculated using the middle 90% of the predictions.

In order to estimate the errors due to bias and variance, a new batch of 500 runs was set up for each system and problem. The same test set as before was held back, but for each run, a new training set of the same size was created using a bootstrapped sample drawn with replacement from the original training set. This is done to avoid overestimating the error due to bias and subsequently underestimating the error due to variance that can be expected when using a fixed training set for all runs.

Tables 8.9 and 8.10 provide an overview of the errors due to bias and variance respectively. These errors are normalized using the variance of the targets. It has been reported (Keijzer and Babovic, 2000a) that for modelling algebraic expressions on the basis of data, genetic programming can quite regularly produce out of range predictions. In order to obtain a more robust measure of the bias, Tables 8.9 and 8.10 also show post-processed values for the bias and variance errors between brackets whereby the highest and lowest 5% of the predictions were excluded from the calculation in Equation 8.5.

8.6.2 Settling velocity of sand particles

Examination of the errors due to bias and variance, reveals that for the *sand* problem, for STGP both terms are much higher than for the other systems. This is a strong indication that the bias introduced by the **uom** is not the most appropriate for this problem. It forces STGP to sample solutions that are on average not fitting well (high bias error) and, given different bootstrapped training data sets, evolves solutions that are different from each other (high variance error). Figure 8.1(a) shows the performance on the test set for the 50 resulting expressions. These results show that STGP invariably produces equations with similar poor performance. The three statistics: high bias error, high variance error when trained on different data sets, together with level performance when trained on the same data, indicate that STGP might suffer from premature convergence due to broken ergodicity (see Section 4.5.1). Inspection of the history of the runs confirms this. At the initialization stage, the declarative bias forces STGP to sample a specific kind of solution (in this case of the general form $\sqrt{dgc}$ where c is some dimensionless term). The remainder of the run time is used to enhance the fit of this general solution by adding various dimensionless terms. This increases the variance error. It appears that the constraints that are enforced partition the search space which drastically reduce the number of admissible solutions. Strongly influenced by initialization, throughout the run STGP continues to sample only admissible sections of the search space without exploring the search space well enough. This results in premature convergence and sub-level performance.

On this same problem, the bias and variance errors for DAGP-DIM, which produces expressions from the same set as STGP, are comparable to the less constrained expressions. This further reinforces the suspicion that STGP suffers from premature convergence as it clearly shows that the constrained search space does contain good solutions.

8.6.3 Settling velocity of faecal pellets

For the *pellets* problem Figure 8.1 shows a clear division between correctly and incorrectly typed results; the latter perform much better. The main cause for this behaviour can again be attributed to the error due to bias. The *pellets* problem is the only problem with the presence of mass units (Table 8.2) which in turn increases the set of constraints. A possible reason for the inability of any of the dimensionally correct expressions to provide an adequate fit might lie in a poor connection between the units of the measured data and the data itself. The **uom** themselves are also poorly connected: the gravitational acceleration g was needed to make it even possible to represent dimensionally correct expressions (see Table 8.2). Since there are only two density units present, ρ_s and ρ_{fl}, there is only one way to use those in a dimensionally correct equation, which is to divide them by each other. The *pellets* problem seems to be underspecified both with respect to coverage by data and with respect to the choice of measurements. Only when coercion errors are tolerated a reasonable fit can be obtained. The poor data coverage enables SRGP to overfit the data, while the DAGP results remain remarkably regularized. Examination of the evolution of the size of the equations (Figure 8.2) shows that SRGP in general evolves much larger solutions than either STGP or DAGP. On the average, the SRGP expressions contain more functions than data points available.

8.6.4 Concentration of suspended sediment near bed

For the *sediment* problem, not much difference between the constrained results can be found. Both approaches perform well in modelling the data. Analysis on the test set reveals that the constrained approaches are as capable in fitting the data as the unconstrained methods. However, the fashion in which the unconstrained methods arrive at these results is instructive: there is a much smaller decline in error from the train to the test set than for the unconstrained solutions. It appears that, for the *sediment* problem, the information contained in the **uom** helps in obtaining solutions that generalize well. It has been stated already in the problem description

(Section 8.3) that the data set covers all relevant physical phenomena. In this setup it seems that the knowledge provided by the **uom** is well correlated with the data.

8.6.5 Additional roughness induced by vegetation

The *roughness* problem is characterized by abundance of systematically sampled data with fixed increments over the entire range of inputs. However, it has been argued earlier (Section 8.4) that the measurements were not complete as nothing has been recorded about the flexibility of the vegetation. As the information about the **uom** does not cover all aspects of the problem, it can be expected that the more heavily biased methods would not benefit as much or could even be hindered by this information. Table 8.9 confirms this: STGP has the highest bias error, followed by DAGP-DIM, DAGP-MID, DAGP-FIT and ultimately SRGP, which has the lowest error due to bias. The significantly better results of DAGP-DIM over STGP can for the largest part be attributed to the lower variance error.

8.6.6 Summary of the quantitative analysis

Quantitatively speaking, for the four problems, it seems that the inclusion of knowledge about the **uom** is best done through expressing a preference rather than imposing syntactical constraints. The preferential bias of DAGP does not exhibit the problems related to broken ergodicity that characterize the syntactical approach. Moreover, DAGP appears to be able to find good solutions even when the **uom** information is only partially relevant. Furthermore, the declarative bias in STGP seems not only to introduce a high bias error when the constraints are not particularly relevant, but it also has a high error due to variance. STGP thus shows disadvantages for being biased (a high bias error when the bias is not appropriate) and being an unbiased method (a high variance error). DAGP only shows the disadvantage due to its biased nature, as its error due to variance is quite low.

The comparison between DAGP and standard symbolic regression is clear cut: the inclusion of the additional objective based on coercion error does not preclude DAGP of searching well. More importantly, the additional objective seems to have an *regularizing* effect on the produced solutions. Table 8.10 shows a considerably smaller tendency of DAGP to produce destructively overfitted expressions. This is also confirmed by inspection of Figure 8.1 where the SRGP runs routinely produce overfitted equations.

The regularizing effect can be most clearly seen in the errors due to bias and variance. Though the errors due to bias are comparable between SRGP and DAGP-FIT, the errors due to variance are significantly higher for SRGP. This is in accordance with findings reported in (Keijzer and Babovic, 2000a). However, it should be emphasized that regularization is not caused by the introduction of units of measurements *per se*, but rather the fashion in which **uom** are introduced.

Furthermore, and as illustrated in Figure 8.2, DAGP generally considers smaller solutions than either STGP and SRGP. This preference to parsimonious solutions seems to be another side-effect of using the **uom** in a multi-objective setting and has been noted elsewhere (Keijzer and Babovic, 1999), which might partially explain the observed regularization effect.

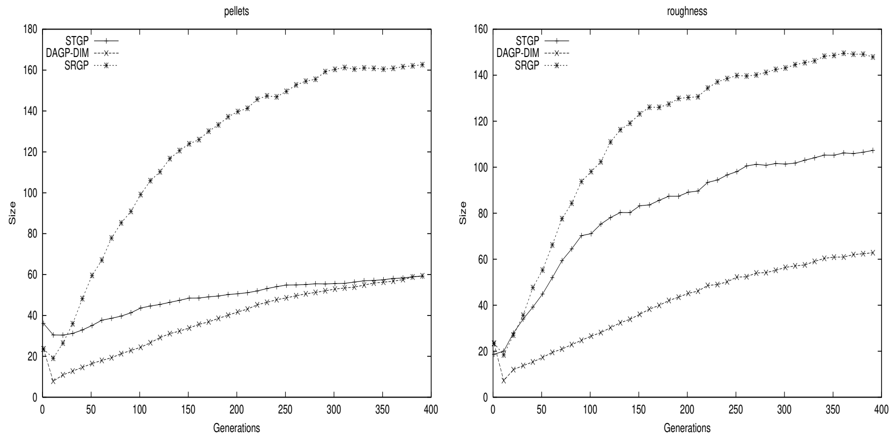


Figure 8.2: Evolution of the average size for the *pellets* and *roughness* problems. The evolution of the size on the other two problems exhibit a similar trend as the graph for the *roughness* problem.

8.7 Qualitative Results

In order to provide an indication of the quality of the solutions that are generated, a short analysis is provided below. This is an inherently subjective process as it involves the interpretation of the equations, and even more subjective reasons such as aesthetic appeal. Judgements need to be made regarding the expressions in order to determine whether the proposed interactions are meaningful or coincidental. However, since the amount of knowledge about the physical processes is very limited no definite statements about the physics should be expected from the hypothesis induction engines.

The expressions below are selected by taking the best performing expressions over the entire data set using both the training and testing sets. These expressions are inspected for their value in describing the problem itself, with the aim to learn something about the interactions occurring in the processes under study and possibly to guide further data collection campaigns. For each method, only one expression is examined. In a more realistic setting a short-list of interesting expressions would be selected and analyzed further. This is not done here, as it would increase the already high degree of subjectivity.

The problems considered in this paper are from highly specialized sub-fields of hydraulics and the authors feel ill-equipped to address them here appropriately. Such a discussion would also fall outside the scope of the present work. The aim of this analysis is not to select the ultimate expression, but rather to point out the interpretability in these equations. The expressions are simplified and in constants the first three significant digits are presented.

The quantitative results have already revealed that there is a trade-off between the information contained in the data (the numbers) and the information contained in the units of measurements. Since dimensionally aware GP produces equations that more-or-less abide the constraints, it is possible to investigate the expressions themselves and possibly learn something about these discrepancies in order to understand the problem more fully.

8.7.1 Interpretability of unconstrained expressions

Consider the best expression produced by symbolic regression (SRGP) for the sediment problem in Formulation box 8.7.1. Although this formulation has the best performance over the training as well as over the test set of all expressions induced in this experiment, it would prove tough if not impossible to interpret this expression. It is not only the sheer size of the formulation which makes the exercise almost impossible, but also the dimensionally inconsistent fashion in which the variables are combined, provides no help in determining the physical interactions for this problem.

$$
\begin{aligned}
c_b \simeq 0.284 \Bigg(& \left(u_f' - w_s\right)^3 \left(u_f' - g\right) \left(g + \frac{u_f' + u_f}{u_f - g}\right) g^{-5} {u_f}^{-1} \\
& \Bigg(g + 13.0 \left(w_s + g^3 u_f' \, w_s {u_f}^{-1} \left(g + \frac{u_f'}{g}\right)^{-1}\right) {u_f}^{-1} \\
& \left(u_f' - 11.3 \frac{g u_f'}{\left(u_f' - w_s\right)^2} - g^2\right)^{-1} \Bigg(g + \Bigg(d_{50} + \sqrt{\frac{\left(u_f' - w_s\right)^2 w_s}{g^4}} + g\Bigg) \\
& \left(2g + \frac{u_f'}{w_s} + u_f' - u_f - w_s + g^2\right)^{-1}\Bigg)^{-1}\Bigg)^{-1}\Bigg)^{\frac{1}{2}}
\end{aligned}
$$

Formulation 8.7.1: The best expression for the sediment problem produced by symbolic regression (SRGP). Even though this formulation produces the best fit to the training and test data (NRMS 0.36), it is very hard to distill some information out of this equation.

This "formula" is taken as an indication of the sort of expression unconstrained genetic programming induces. Space restrictions prevent us from presenting the best expressions SRGP produces for the other problems. It will suffice to state that these do not provide a brighter picture.

As has been stated in the introduction, one can also perform a dimensional analysis and transform all dimensioned variables to dimensionless groups of numbers. Using this approach one would avoid problems related to units of measurement, but certainly not guarantee creation of solutions of lower complexity (Babovic and Keijzer, 2000). Formulation 8.7.1 clearly demonstrates the need for constraints if one wants to use GP in a knowledge discovery setting.

8.7.2 Settling velocity of sand particles

The best expressions for the *sand* problem can be found in Formulation 8.7.2. In all three expressions a distinct pattern emerges revealing a commonly appearing subexpression $\sqrt{g\gamma' d}$. This is not only dimensionally correct and physically relevant but

$$\text{STGP} \quad w_s \simeq \sqrt{g\gamma' d}\left(1.223 - \frac{1.09}{\sqrt[8]{A}}\right)$$

$$\text{DAGP-DIM} \quad w_s \simeq 0.447\sqrt{g\gamma' d}\sqrt[8]{A^{\frac{1}{2}}\gamma'^{\frac{1}{4}} - 1.93}$$

$$\text{DAGP-MID} \quad w_s \simeq 0.170\ (gd\gamma')^{\frac{17}{32}}\gamma'^{\frac{1}{16}}d^{\frac{1}{8}}g^{\frac{1}{4}}\sqrt{1.43 - \sqrt{\frac{\gamma'}{\sqrt{A\left(\frac{1.99}{\sqrt[4]{\gamma'}} - \gamma'\right)}}}}$$

Formulation 8.7.2: Hypotheses generated using STGP, DAGP-DIM and DAGP-MID for the *sand* problem. The NRMS errors of these equations are 0.26, 0.28 and 0.22 respectively.

it also reduces the NRMS error to 0.40 when evaluated as is. In order to increase accuracy while remaining dimensionally correct, this basic expression is scaled by the dimensionless terms A and γ'. The grain diameter d cannot be involved in this scaling without sacrificing dimensional correctness. However, since DAGP-MID allows small dimensional errors, it can use the variable d and succeeds in significantly reducing the error by manipulating d, A and γ'. Internal consistency for DAGP-MID remains; only the output units are different from the desired **uom**.

Another structure that emerges in these experiments reveals that it is beneficial to take repeated roots of the variable A, most often three times, resulting in the term $\sqrt[8]{A}$. Further examination of the range of A reveals that $\sqrt[8]{A} \sim e \times log(A)$. Taking these bits of information together could lead to an experiment where the desired output is divided by the term $\sqrt{g\gamma' d}$ and a symbolic regression experiment including the logarithm in the function set in order to find the optimal scaling factor.

These considerations and manipulations are described to indicate how this approach fits in scientific work. Scientists generally approach discovery from various angles: proposing tentative formulations, scrutinizing them based on first principles and also by manipulating expressions both symbolically and numerically. The induction of dimensioned expressions using GP can provide a fertile ground for such experimentation.

8.7.3 Settling velocity of faecal pellets

The expressions for the pellets problem can be found in Formulation box 8.7.3. STGP models the relationship by taking the square root of two terms relating g and a length term l, while DAGP-DIM produces a short expression relating the settling velocity to the rectangular surface of the pellets $l \times w$, scaled by the ratio of densities. The equation indicates that the faecal pellet settling velocity increases with increased values of the nominal diameter and the differential (or floating) density. This general relationship has also been developed for the settling velocities of other types of particles. The general equation describing these relationships is given as:

$$\text{STGP} \quad w_s \simeq \sqrt{\frac{1}{50}\,gw - 0.0189\,g\left(0.764\,w - 0.664\,\frac{\rho_{fl}\,(2\,w+l)}{\rho_s}\right)}$$

$$\text{DAGP-DIM} \quad w_s \simeq 0.144\sqrt{g}\sqrt[4]{\frac{wl\rho_{fl}}{\rho_s}}$$

$$\text{DAGP-MID} \quad w_s \simeq \frac{\sqrt{gw}}{10} + \frac{g^{\frac{13}{16}}\rho_{fl}{}^{\frac{49}{32}}l^{\frac{19}{16}}}{100\rho_s{}^{-\frac{19}{16}}}$$

Formulation 8.7.3: Hypotheses generated using STGP, DAGP-DIM and DAGP-MID for the *pellets* problem. The NRMS errors of these equations are 0.58, 0.60 and 0.54 respectively.

Equation	A	B	C
Stokes Settling (Low Re) - Sphere	1	2	1
High Re - Sphere	0.5	0.5	0.5
Equation (8.2)	0.6	0.8	0.6

Table 8.11: Typical parameters for various settling parameters

$$w_s \sim \alpha g^A d_n^B \left(\frac{\rho - \rho_f}{\rho_f}\right) \tag{8.6}$$

where values for A, B, and C are given in Table 8.11.

The faecal pellet data examined here are in the intermediate Reynolds number range $(37 < Re < 178)$, which explains why the semi-empirical Equation 8.2 has values for A, B and C lying between the range of the values for Stokes and High Re settling.

Comparing Equation 8.2 with the expression induced with DAGP-DIM reveals that the dependence of the settling velocity on the geometrical properties is essentially the same (albeit in the DAGP-DIM case the dependence is on l and w and not d_n). Also, the power value is lower for the ratio of the relative density to the fluid density than in Equation 8.2. Nevertheless, it can be concluded, that the DAGP-DIM equation has the same general form as other particle settling velocity equations. It is only the dependence of the settling velocity on the geometrical properties and the relative particle density that is slightly different from other cases.

DAGP-MID produces an equation that fits better than other approaches although in this case it does not remain internally consistent. Taking this equation as a prototypical example of a GP-generated hypothesis, it is possible to further manipulate the expression manually. The purpose is to demonstrate how a domain specialist can use additional insights and symbolic gymnastics to distill some meaning out of tentative formulations such as the one above. For example, the power of the ρ_{fl} variable of magnitude $49/32$ is rather close to the 'nicer' power of $48/32 = 1.5$. In the same spirit, one can change the $\rho_s^{-19/16}$ term to $\rho_s^{-24/16} = \rho_s^{-1.5}$. After these manipulations, which in effect reduce the error, the density term becomes a true ratio and their **uom** disappear in the overall expression. The simplified expressions reads:

$$w_s \simeq \frac{\sqrt{gw}}{10} + \frac{g^{\frac{13}{16}} l^{\frac{19}{16}} \left(\frac{\rho_{fl}}{\rho_s}\right)^{\frac{3}{2}}}{100}$$

which has an NRMS error of 0.53, however with a coercion error. The two terms do point at separate effects in the physics of the settling of the pellets. The first term depends on the width, while the second term depends on the length and the densities. Note that the STGP solution has a similar form, thus reinforcing a 'theory' that such a decomposition represents a valid approach.

8.7.4 Concentration of suspended sediment near bed

$$\begin{array}{lll}
\text{STGP} & c_b \simeq & 0.0132 \, \dfrac{\left(u'_f - w_s\right)^2 {w_s}^2 u'_f}{d_{50} \; u_f \; g \left(6 \, {w_s}^2 + 6 \, {u'_f}^2 - 5 \, u'_f \; w_s\right)} \\
\text{DAGP-DIM} & c_b \simeq & - \dfrac{u'_f \left(9.87 \times 10^{-6} \, u_f - 1.27 \times 10^{-4} \, u'_f\right)}{g d_{50}} \\
\text{DAGP-MID} & c_b \simeq & 0.0143 \, \dfrac{\left(w_s - u'_f\right)^{\frac{13}{8}} \left(u_f - w_s\right)^{\frac{1}{8}}}{g \, {u_f}^{\frac{7}{8}} {u'_f}^{\frac{1}{8}} w_s^{\frac{1}{4}}}
\end{array}$$

Formulation 8.7.4: Hypotheses generated using STGP, DAGP-DIM and DAGP-MID for the *sediment* problem. The NRMS errors of these equations are 0.46, 0.46 and 0.42 respectively.

For the *sediment* problem, DAGP-DIM produces an elegant formulation balancing the shear forces with the median diameter of the settling particles. No reference to the settling velocity w_s is made which makes this formula more parsimonious than the STGP formulation. At this point one can also forward an argument of a different nature. The first two case studies were concerned with finding reasonable expressions for w_s itself, which is intrinsically difficult to measure and characterize, and consequently inevitably polluted by noise. It appears that DAGP provides a high quality fit with 'smoother' expressions (or at least depending on 'smoother' variables).

Finally and very importantly, it should be noted that all equations presented in Formulation box 8.7.4 provide a higher degrees of accuracy than the human-generated formula 3.17. Furthermore, equation 8.7.4 was fitted on entire data set with resulting $NRMS = 0.47$.

$$
\begin{array}{lll}
\text{STGP} & c \simeq & 3\, h_w \sqrt{g h_r}^{-1} \dfrac{1}{\sqrt{2\, h_r + 2\, d + h_w + 4\, dm}} \\
& & \left(\sqrt{h_r} + \sqrt{h_r\, p \left(\sqrt{p} + 4\, p^2 + \dfrac{h_r}{d\,(p+m)} \right) (2\, p + m)^{-1}} \right) \\
\text{DAGP-DIM} & c \simeq & 1.58 \sqrt[4]{\dfrac{g^2\,(h_w - h_r)\, h_w^2\, p}{m\, d\, h_r^2}} \\
\text{DAGP-MID} & c \simeq & 1.87 \dfrac{(h_w\,(h_w + d))^{\frac{1}{8}}\,(h_w - h_r)^{\frac{3}{16}}\, p^{\frac{5}{32}}\, g^{\frac{37}{64}}}{h_r^{\frac{17}{32}}\,(d\,m)^{\frac{1}{4}}}
\end{array}
$$

Formulation 8.7.5: Hypotheses generated using STGP, DAGP-DIM and DAGP-MID for the *roughness* problem. The NRMS errors of these equations are 0.27, 0.26 and 0.20 respectively.

8.7.5 Additional roughness induced by vegetation

The formulations for the *roughness* problem are presented in Formulation box 8.7.5. The STGP expression shows that using the **uom** does not necessarily lead to understandable formulations. In the denominator it adds water depths and diameters of the reeds to form its dimensionally correct expression. It is difficult to imagine the physical significance of this addition.

The best formulations for DAGP-DIM and DAGP-MID came from the same run, and closer inspection reveals a high degree of similarity in the results. Rewriting the equations in the form of a product of simple terms raised to a certain power reveals structural similarity. Ignoring the constant terms, for DAGP-DIM and DAGP-MID respectively this formatting results in the following expressions:

$$
\begin{array}{rl}
h_w^{\frac{1}{2}} & (h_w - h_r)^{\frac{1}{4}} p^{\frac{1}{4}} m^{-\frac{1}{4}} d^{-\frac{1}{4}} h_r^{-\frac{1}{2}} \\
h_w (h_w - d)^{\frac{1}{8}} & (h_w - h_r)^{\frac{3}{16}} p^{\frac{5}{32}} m^{-\frac{1}{4}} d^{-\frac{1}{4}} h_r^{-\frac{17}{32}}
\end{array}
$$

The two equations written in this form reveal a high degree of similarity. The powers of the second — dimensionally incorrect — expression are rather close to the correct powers in the first expression. The main difference between the dimensionally correct and the dimensionally incorrect equation lies in the $\sqrt{h_w}$ term, that appears in the DAGP-DIM equation. The dimensionally incorrect expression uses $\sqrt[8]{h_w(h_w - d)}$. Furthermore, removing the d variable from this expression does not increase the error. Replacing the $\sqrt{h_w}$ term in the dimensionally correct equation with $\sqrt[4]{h_w}$ and rescaling the formula, indeed reduced the NRMS error from 0.26 to 0.20. This is a similar procedure as outlined in Section 8.7.3, where 'strange' powers are rounded to nearest 'nice' powers. The output units of this expression would however still be stated in incorrect units. If this h_w term was stated in surface units however, the expression would be dimensionally correct. One possibility to consider is that the term is used as a proxy for a variable that is stated in surface units which has values proportional to h_w.

8.7.6 Summary and scope of GP-based scientific discovery

The overall process described in this study forms in the authors' opinion a first iteration of applying genetic programming in the domain of scientific discovery. As the results produced by genetic programming are knowledge-rich, the hypotheses induced can be used to further refine the experimental setup or inspire a new set of experiments in an iterative fashion. In this study, the raw data were taken at face value and the expressions were induced and post-processed by automatic means. We firmly believe that the post-processing should be carried out by domain specialists that can use their background knowledge and sense of aesthetics to judge which of the proposed hypotheses is the most appropriate formulation. Such a judgement is not offered here; related work (Babovic, 1996; Babovic and Keijzer, 1999; Babovic et al., 2001; Babovic and Keijzer, 2000) does attempt to select an appropriate formulation and sets up a small theory of worth of the expressions produced by GP. These 'theories' are set up after examining the hypotheses generated by GP and provide ground for discussion and further experimentation. The qualitative analyses given above gave a few examples on how to use the hypothesis generated by GP to increase the usability of the expressions and how they fit into scientific work. Specifically it was shown that the scientist using systems like these can:

- **Exploit numerical similarity.** A persistent repetition of consecutive roots taken of a single variable in Section 8.7.2 lead to the discovery of a numerical near equivalence with a logarithmic relation in the domain. This inspired the proposal of a new experiment using GP that would include this logarithmic relation. Entering this different sub-expression into the equation can also be done manually.
- **Exploit syntactic similarity to existing equations.** Section 8.7.3 showed that one of the GP-induced expressions was quite similar to an existing, human-proposed expression, even though it used different variables. This GP-induced equation used measurable variables rather than approximations. This can reinforce the acceptance of the existing equation and, by virtue of the new equation being stated using different variables, suggest an extension of this equation to a family of related equations — applicable to different measurement collection campaigns.
- **Use symbolic manipulations for manual improvement.** Section 8.7.3 also presented an possibility to improve a GP-induced equation by manual intervention. By changing the powers of an expression we were able to improve both the goodness of fit of the equation and its aesthetic appeal. A scientist, intimately familiar with the domain, is even more likely to use such manipulations to provide genuine advances in model elegance and ability to explain the data.
- **Use within-run syntactic differences to examine the descriptiveness of the data.** In Section 8.7.5, the difference between a well-fitting expression that was dimensionally incorrect and a less accurate expression that was dimensionally correct could be narrowed down to a difference in a single term. These expressions evolved in the same run using DAGP. The term in question depended on a single variable stated in length units. It was possible to make the well-fitted expression dimensionally correct by verifying that the variable

stated in length units was used as a proxy for a variable, proportional to the original, stated in surface units. The definition of such a variable and a subsequent new measurement campaign to measure this variable could lead to enhanced understanding of the physical process. This is a very good example of the worth of more-or-less correct expressions as they can point to discrepancies in the problem description.

- **Produce expressions that perform better than human-proposed ones.** Finally and very importantly, GP is capable of producing expressions that are better than those developed by a scientist. From the four problems, there was only one instance (the sediment problem), where a human-proposed expression was stated solely in terms of available data. This allowed a direct comparison between the GP-induced expressions and the human-proposed expression on their ability to fit the data (Section 8.7.4). The GP-induced expressions did not only perform better, they were also stated in more basic units, making interpretation easier.

The main advantage of using genetic programming in scientific discovery is its ability to generate a large number of different, yet meaningful hypotheses in a very short amount of time. These hypotheses are based on the experimental data while satisfying constraints and are stated in a language that is considered well suited for these problems: mathematics. GP contains no notion about the problem other than the constraints and the available data. GP is thus able to propose solutions that are non-intuitive and sometimes provocative. The time scale of human invention runs on the scale of months, if not years. Using a hypothesis generator can considerably accelerate this process, once the scientist is able to interpret these hypotheses. The use of **uom** to constrain or bias the search has proven to be very helpful in this setting.

8.8 Discussion

Many issues have surfaced in the preceding sections. Although it is clearly possible to evolve dimensionally correct equations based on data a trade-off has been observed. Allowing small dimensional errors can improve the ability to provide well fitting equations, sometimes with radically better results. This principally occurs when the **uom** of the problem definition does not provide a complete coverage of the dynamics of the system under study.

Genetic programming is an opportunistic search algorithm: it provides expressions that fit the data while satisfying the constraints. Since the only feedback from the problem domain is in the form of error functions, the algorithm produces expressions that model the relationship in whatever fashion that reduces this error. When the numerical values of a particular measurement are indicative of some other underlying phenomenon that could be stated in different **uom**, the genetic programming system uses the measurements in a very different way than the **uom** prescribe. It is thus susceptible of modelling by association, where a set of numerical values are used as a proxy for an underlying phenomenon. This holds in general for any form of data-driven modelling. The use of the **uom** serves two purposes: to reduce this modelling by association, and to aid in interpreting the expressions.

The resulting front of non-dominated solutions produced by DAGP makes it possible to examine the differences between the correct and the more-or-less correct expressions. As the expressions are usually related — they share the same evolutionary history — it is insightful to examine the trade-off between the coercion error and the fit on the data. For the *roughness* problem this trade-off was used to improve upon the expressions.

Even though with STGP the search space is vastly reduced in comparison to SRGP, no evidence is found that the reduction of the search space leads to the evolution of better solutions in a shorter amount of time. On the contrary, the relaxation of the constraints helped in evolving better fitting equations. This might be an artifact of this particular application and use of units of measurement. Still it offers a strong case against the prevailing intuition that the reduction of the search space helps in solving problems faster.

Chapter 9

Conclusion

This work introduced several approaches that enhance genetic programming to be able to induce symbolic expressions on data while taking care of the units of measurement. The goal of this approach is to automatically induce expressions that can be analyzed with numerical and symbolic means by an informed user.

It was shown in Chapter 3 that symbolic regression as such does not provide much benefit over other methods of regression. The use of the symbolic nature of the expressions induced by standard GP as a vehicle for obtaining insight appears to be problematic. As the only feedback supplied to standard GP is the (numerical) error, the expressions are then mostly numerical recipes to reduce this error.

To obtain interpretable expressions, an approach that exploits the symbolic nature of of genetic programming is developed. It critically uses units of measurement as a method of typing the expressions. A completely typed result in this system corresponds with a fully dimensioned mathematical relation. The derivative scales that are proposed by such an expression attempt to establish some relationship between a physical process and the automatically induced expression.

Chapter 4 showed that units of measurement can be implemented using a type system where the exponents of the units are used as separate types. Mathematical operations on the values then correspond with operations on the types. It was shown that context-free grammars cannot model the system of units of measurement in full generality. A method based on parametric polymorphism enhanced with explicit type arithmetic seems to be needed. It was argued that to provide maximal aid in the explorative field of scientific discovery, rigorously abiding this type system is not necessarily the optimal approach. Two main problems with using the units of measurement as a *formal* system, subject to purely formal manipulations, have been identified.

- **Formally correct expressions can be meaningless.**
  Units of measurement can not exhaustively specify an experimental setup. There are then several ways how meaningless expressions can be formed. A basic example of a meaningless expression would be the formally correct application of an arcsine on a ratio of two weight measurements. Formally

this would produce an angular measurement, physically this manipulation is meaningless.

This might be considered a pathological example, but in one of the case studies the following situation was encountered. In the roughness problem, two length measurements were used in an addition: one measurement was the mean diameter of a plant, the other the water depth. It is questionable what the physical meaning is of this addition as not only the scales of the two variables differ enormously, the measurements themselves apply to different directions in the experimental setting.

- **Formally incorrect expressions can be meaningful.**
  Because physical experiments are limited in scope and because it is unknown in advance what variables need to be combined to provide the answer, many possibly relevant variables are not measured, or are kept at constant values. The values that are measured can then be equivalent (up to a multiplicative constant) to a whole set of phenomena stated in different units. A length measurement can be proportional to a velocity if all measurements are performed using a constant period of time. A length measurement can be proportional to a rectangular surface measurement if the length at the other axis is kept constant.

  The case study on the roughness problem is again illustrative for the potential meaningfulness of formally incorrect equations. In this problem, a formally incorrect expression was induced that compared to its formally correct counterpart 'misused' a length measurement, the water depth, as if it were a surface measurement. In the experimental setup, there was however no measurement of the width of the channel or the blocking surface of the plants. As the data was produced by a numerical model, this width is likely to be kept constant, maybe even implicitly. The increase in accuracy of the incorrect expression over its formally correct counterpart seems to support an hypothesis that in the roughness problem, a 'blocking surface' measurement might be needed in the formulation of an empirical equation. Regardless of whether this hypothesis is true, without considering formally incorrect expressions, such alternative views of the meaning of the variables in the experimental setup are impossible.

It is then unlikely that a purely formal approach will be the ultimate tool in scientific discovery. A formal approach is only likely to give the optimal answer if the user can a priori state that each measurement can be combined according to formal rules to form meaningful results. This situation seems to be only obtainable in the context of a predictive theory or an empirical equation; such a theory is exactly what is being searched for.

It is therefore claimed here that the designation of the input data in *appropriate* units of measurement is as much part of the process of scientific discovery as the formulation of combinations of this data. It is important to note that the argumentation does not dispute the worth of units of measurement in forming models of physical processes, but merely points at the difficulties a purely formal view of scientific discovery brings.

Apart from considerations about the nature of units of measurement, the exact way to search the space of dimensionally (in)correct expressions through the means of genetic programming has been researched. Two possible ways of biasing the search of a genetic programming system have been identified. One is the inclusion of the units of measurement as a *declarative* bias, where the space of possible expressions is reduced to those that are dimensionally correct. The other is the implementation of a *preferential* bias: here dimensional correctness is not seen as an all or nothing proposition, but a gradation between the severity of constraint violations is used to induce a set of expressions that balance accuracy on the data and dimensional (in)correctness.

Two different systems have been developed and described that implement these different biases. The method that uses a preferential bias is described in Chapter 5. It critically depends on a multi-objective search strategy to balance goodness-of-fit and dimensional correctness. The general method is based on *coercion* of types, applied to the induction of expressions using units of measurement it is called Dimensionally Aware GP (DAGP).

The system that implements the declarative bias is introduced in Chapter 6 and is applied to the induction of dimensionally correct expressions in Chapter 7. It is used as a strongly typed genetic programming system (STGP). Although this STGP system uses a genetic algorithm as the main search engine, it is envisioned that other weak search algorithms such as simulated annealing can be applied as well. This in contrast with the DAGP system, where the multi-objective search that is central to its operations quite likely prevents a non population based search to be applicable.

The comparative study between straightforward symbolic regression, dimensionally aware GP and strongly typed (dimensionally correct) GP in Chapter 8 showed that on four real-world problems, a trade-off exists between the information contained in the observations and the information contained in the units of measurement. The use of units of measurement actually hinders the search for accurate formulations, even though it helps in interpreting them. This was shown through the means of a bias-variance analysis, where the *error due to bias* for dimensionally correct expressions was found to be structurally higher than the bias error for less constrained expressions. Even worse, the strongly typed (declarative) approach to the induction of dimensionally correct expressions showed on one of the four problems that it can combine a high bias error with a high variance error, thus exhibiting very poor search performance. It is hypothesized that this is caused by problems with the ergodicity of the search space. The dimensionally aware approach does not exhibit this problem, but rather shows a graceful degradation when constraints are hard to satisfy.

Purely from the perspective of search towards dimensionally correct expressions, the DAGP approach already appears to be better suited for these types of problems than the STGP approach.

Furthermore, if one compares the dimensionally aware approach with standard symbolic regression on the basis of purely the fitting capability, no significant difference between the two methods is found. This indicates that the multi-objective search that implements the preferential bias does not prevent the system of fitting the data equally well as an unbiased system. The balance between obtaining a good fit on the data and presenting a dimensioned expression is thus able to give a good sample of the range of possible expressions. On the one hand, the best fitted expressions are not worse than what can be obtained from using a technique such as symbolic regression, while on the other hand the dimensionally correct formulations are not worse (and sometimes better) than what can be obtained using a strongly typed system. This balance between well-fittedness and dimensional correctness is dynamically established during the search.

The implementation of the preferential bias in a multi-objective search toward a front of non-dominated solutions helped in regularizing the solutions. A side-effect towards parsimonious solutions was observed and it is hypothesized that this is one of the causes for the regularizing effect. The method employing declarative bias does not exhibit these side-effects.

It appears that the approach based on coercion achieves the best balance amongst satisfying the constraints, fitting the data and regularization of the induced expressions. The (im)balance between the constraints and the fitting ability explicitly catered for in this dimensionally aware GP provides additional insights into the problem.

Using DAGP and STGP, several expressions have been found that provide a consistent hypothesis of the main characteristics of the physical process. This demonstrates the value of the expressions that use units of measurement to help in the interpretation of the results. It is this possibility of directly interpreting the results that distinguishes genetic programming from other methods. The modifications to genetic programming presented here can deliver this interpretability in a more profound way than straightforward symbolic regression.

The overall goal of the work — aiding in the interpretation of the symbolic results produced by genetic programming — has been achieved. By using additional information, units of measurement, as a type system, the genetic programming system is forced to produce expressions that have some limited semantic content. This semantic content can however not be extracted without any effort. The user is critical in relating the application of arithmetic with the derivation of physical concepts.

In the perspective of measurement theory, the use of these systems without additional human effort is unsound. Measurement theory prescribes the process of substituting numbers for observables and ultimately substituting equations for physical processes as a one way street: an arithmetical operation on two measurements can be performed *if and only if* there is a meaningful physical analogy of this operation in the physical process. Examples of this can be found in the experiments studied here. In the roughness problem: subtraction of the reed height from the water depth is a meaningful operation as it describes the length in the channel where water can flow unhindered by vegetation. However, adding the water depth to the reed diameter appears to be meaningless, even though the operation is formally correct.

There is thus no formal guarantee that a well-fitted dimensionally correct expression proposes a meaningful relation, hence the unsoundness of *merely* using units of measurements to make equations meaningful. However, an informed user — the domain specialist — can use the equations to find meaningful relations. As the equations are well-fitted to the data, there is a high likeliness that the relationships that are proposed have some connection with the physical process that is modeled. It is shown in this work that such connections, can be found by examining the resulting equations. In particular, it was shown that by not insisting on dimensional correctness at all times (DAGP), better fitting expressions can be found, without necessarily sacrificing this interpretability.

The main benefit of these methods then lies in proposing well-fitted equations that can be used to obtain a better insight in the physical process that underly the data. Rather than having the human scientist research all possible combinations of variables and derivative measurements, while at the same time trying to obtain accuracy (a low error), the systems here automatically induce expressions with high accuracy and tentative relations. The scientist trying to interpret the expressions functions as a reality check. In the absence of a formal syntax and semantics of physical reality this human influence is necessary.

The system described in this work is envisioned to be a source of additional information, to be used next to the measurements themselves. The scientist can use the additional set of equations to more accurately describe the physical process under study. Ultimately, the goal is to form an empirical equation together with a justification for its use. The system is capable of providing highly compressed views on this data in the form of symbolic expressions. By balancing forces, velocities and length measurements, the equations produced by the system are not necessarily black box models whose performance can only be measured in their ability to reliably 'explain' the variance in the data, but can be inspected by the scientist and used as a vehicle for interpretation and as a form of inspiration for alternative views of the problem. By combining the power of an automated system to produce equations, and the ingenuity of a human observer to form hypotheses from these equations, it is thought that the best of two worlds are combined: fast exploration of the space of symbolic descriptions by automatic means and the creativity of a human mind to find a justification or a refutation for the well-fittedness of these expressions.

This combination of data-driven search and knowledge-driven justification is hoped to lead to new advances in science. The unbiasedness of data-driven search might lead the knowledge-driven process of theory-formation to consider novel concepts or novel combinations of existing concepts.

List of Tables

List of Figures

Bibliography

Babovic, V. (1996). *Emergence, Evolution, Intelligence: Hydroinformatics*. Balkema, Rotterdam.

Babovic, V. and Keijzer, M. (1999). Data to knowledge - the new scientific paradigm. In Savic, D. and Walters, G., editors, *Water Industry Systems: Modelling and Optimisation Applications*, pages 3–14. Research Studies Press, Exeter.

Babovic, V. and Keijzer, M. (2000). Genetic programming as a model induction engine. *Journal of Hydroinformatics*, 2(1):35–60.

Babovic, V., Keijzer, M., Aguilera, D. R., and Harrington, J. (2001). Analysis of settling processes using genetic programming. D2K Technical Report 0501-1, http://www.d2k.dk.

Banzhaf, W. (1994). Genotype-phenotype-mapping and neutral variation – A case study in genetic programming. In Davidor, Y., Schwefel, H.-P., and Männer, R., editors, *Parallel Problem Solving from Nature III*, volume 866 of *LNCS*, pages 322–332, Jerusalem. Springer-Verlag.

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag.

Buckingham, E. (1914). On physically similar systems: Illustrations of the use of dimensional equations. *Physical Review*, IV(4):345–376.

Burke, E. and Foxley, E. (1996). *Logic and Its Applications*. Prentice Hall.

Cramer, N. L. (1985). A representation for the adaptive generation of simple sequential programs. In Grefenstette, J. J., editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA.

Darwin, C. (1859). *On the Origin of the Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*.

Davies, C. (1945). Definitive equations for the fluid resistance of spheres. *Proceedings of the Physical Society*, 57(322).

de Jong, E. D., Watson, R. A., and Pollack, J. B. (2001). Reducing bloat and promoting diversity using multi-objective methods. In Spector, L., Goodman, E. D., Wu, A., Langdon, W. B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M. H., and Burke, E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 11–18, San Francisco, California, USA. Morgan Kaufmann.

Deb, K., Agrawal, S., Pratab, A., and Meyarivan, T. (2000). A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In Schoenauer, M., editor, *Proceedings of PPSN-6*, pages 849–858. Springer-Verlag.

Einstein, H. A. (1950). The bed-load function for sediment transport in open channel flow. Technical Bulletin 1026, U.S. Department of Agriculture, Washington, D.C.

Fogel, L., Owens, A., and Walsh, M. (1966). *Artificial Intelligence through Simulated Evolution*. John Wiley.

Garcia, M. and Parker, G. (1991). Entrainment of bed sediment into suspension. *Journal of Hydraulic Engineering*, 117(4):414–435.

Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural Computation*, 4:1–58.

Gibbs, R., Matthews, M., and Link, D. (1971). The relationship between sphere size and settling velocity. *Journal of Sediment Petrology*, 41(1):7–18.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley.

Gruau, F. (1996). On using syntactic constraints with genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 19, pages 377–394. MIT Press, Cambridge, MA, USA.

Guy, H., Simons, D., and Richardson, E. (1966). Summary of alluvial channel data from flume experiments, 1956-61. Professional Paper 462-I, U.S. Geological Survey, Washington D.C.

Hallermeier, R. J. (1981). Terminal settling velocity of commonly occuring sands. *Sedimentology*, 28:859–865.

Handley, S. (1994). On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 154–159, Orlando, Florida, USA. IEEE Press.

Harries, K. and Smith, P. (1997). Exploring alternative operators and search strategies in genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 147–155, Stanford University, CA, USA. Morgan Kaufmann.

Haynes, T. D., Schoenefeld, D. A., and Wainwright, R. L. (1996). Type inheritance in strongly typed genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 18, pages 359–376. MIT Press, Cambridge, MA, USA.

Holland, J. H. (1980). *Adaptation in Natural and Artificial Systems.* University of Michigan Press.

Holland, J. H., Halyoak, K. J., Nisbett, R. E., and Thagard, P. R. (1986). *Induction, Processes of Inference, Learning and Discovery.* The MIT Press.

Iba, H., de Garis, H., and Sato, T. (1994). Genetic programming using a minimum description length principle. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 12, pages 265–284. MIT Press.

Jaffar, J. and Maher, M. J. (1994). Constraint logic programming: A survey. *Journal of Logic Programming*, 20:503–581.

Keijzer, M. (1996). Efficiently representing populations in genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 13, pages 259–278. MIT Press, Cambridge, MA, USA.

Keijzer, M. and Babovic, V. (1999). Dimensionally aware genetic programming. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1069–1076, Orlando, Florida, USA. Morgan Kaufmann.

Keijzer, M. and Babovic, V. (2000a). Genetic programming, ensemble methods and the bias/variance tradeoff - introductory investigations. In Poli, R., Banzhaf, W., Langdon, W. B., Miller, J. F., Nordin, P., and Fogarty, T. C., editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 76–90, Edinburgh. Springer-Verlag.

Keijzer, M. and Babovic, V. (2000b). Genetic programming within a framework of computer-aided discovery of scientific knowledge. In Whitley, D., Goldberg, D., Cantu-Paz, E., Spector, L., Parmee, I., and Beyer, H.-G., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, Las Vegas, Nevada, USA. Morgan Kaufmann.

Keijzer, M., Babovic, V., Ryan, C., O'Neill, M., and Cattolico, M. (2001a). Adaptive logic programming. In Spector, L., Goodman, E. D., Wu, A., Langdon, W. B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M. H., and Burke, E., editors, *Proceedings of GECCO 2001*. Morgan Kaufmann.

Keijzer, M., Ryan, C., O'Neill, M., Cattolico, M., and Babovic, V. (2001b). Ripple crossover in genetic programming. In Miller, J. F., Tomassini, M., Lanzi, P. L., Ryan, C., Tettamanzi, A. G. B., and Langdon, W. B., editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 74–86, Lake Como, Italy. Springer-Verlag.

Keith, M. J. and Martin, M. C. (1994). Genetic programming in C++: Implementation issues. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 13, pages 285–310. MIT Press.

Komar, P. D. and Taghon, G. L. (1985). Analysis of settling velocities of faecal pellets from the subtidal polychaete amphicteis scaphobronchiate. *Journal of Marine Research*, 43(3):605–614.

Kompare, B. (1995). *The Use of Artificial Intelligence in Ecological Modelling*. PhD thesis, University of Copenhagen, Denmark.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Kutija, V. and Hong, H. (1996). A numerical model for addressing the additional resistance to flow introduced by flexible vegetation. *Journal of Hydraulics Research*, 34(1):99–114.

Langdon, W. B. (1999). Size fair and homologous tree crossovers. Technical Report SEN-R9907, Centrum voor Wiskunde en Informatica, CWI, P.O. Box 94079, Kruislaan 413, NL-1090 GB Amsterdam, The Netherlands.

Langdon, W. B. and Poli, R. (1998). Why ants are hard. Technical Report CSRP-98-4, University of Birmingham, School of Computer Science. Presented at GP-98.

Langley, P., Simon, H. A., Bradshaw, G. L., and Zytkow, J. M. (1987). *Scientific Discovery, Computational Explorations of the Creative Processes*. The MIT Press.

Larsen, T., Frier, J., and Vestergraard, K. (1990). Discharge/stage relation in vegetated danish stream. In *International Conference on River Flood Hydraulics*.

Luke, S. and Panait, L. (2001). A survey and comparison of tree generation algorithms. In Spector, L., Goodman, E. D., Wu, A., Langdon, W. B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M. H., and Burke, E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 81–88, San Francisco, California, USA. Morgan Kaufmann.

Martin, L., Moal, F., and Vrain, C. (1999). Declarative expression of biases in genetic programming. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 401–408, Orlando, Florida, USA. Morgan Kaufmann.

Maynard-Smith, J. (1975). *The Theory of Evolution*. Penguin Books, third edition.

Montana, D. J. (1995). Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230.

Muggleton, S. and Raedt, L. D. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679.

O'Neill, M. and Ryan, C. (2001). Grammatical evolution. *IEEE Transaction on Evolutionary Compuation*. Forthcomming.

Palmer, R. (1982). Broken ergodicity. *Advances in Physics*, 31:669–736.

Park, J. and Sandberg, I. W. (1991). Universal approximation using radial basis function networks. *Neural Computation*, 3(2):246–257.

Poli, R. and Langdon, W. B. (1998). On the ability to search the space of programs of standard, one-point and uniform crossover in genetic programming. Technical Report CSRP-98-7, University of Birmingham, School of Computer Science. Presented at GP-98.

Poli, R. and Langdon, W. B. (1999). Sub-machine-code genetic programming. In Spector, L., Langdon, W. B., O'Reilly, U.-M., and Angeline, P. J., editors, *Advances in Genetic Programming 3*, chapter 13, pages 301–323. MIT Press, Cambridge, MA, USA.

Ratle, A. and Sebag, M. (2000). Genetic programming and domain knowledge: Beyond the limitations of grammar-guided machine discovery. In Schoenauer, M., Deb, K., Rudolph, G., Yao, X., Lutton, E., Merelo, J. J., and Schwefel, H.-P., editors, *Parallel Problem Solving from Nature - PPSN VI 6th International Conference*, pages 211–220, Paris, France. Springer Verlag. LNCS 1917.

Rechenberg, I. (1965). Cybernetic solution path of an experimental problem.

Ross, B. (1999). Logic based genetic programming with definite clause translation grammars. Technical report, Department of Computer Science, Brock University, Ontario Canada.

Rouse, H. (1939). Experiments on the mechanics of sediment transport. In *Proceedings of 5th International Congress of Applied Mechanics*, pages 550–554. Cambridge, Massachusetts.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229.

Schwefel, H.-P. (1995). *Evolution and Optimum Seeking*. John Wiley & Sons.

Sterling, L. and Shapiro, E. (1994). *The Art of Prolog*. MIT press.

Stevens, S. (1959). Measurement, psychophysics, and utility. In Churchman, W. C. and Ratoosh, P., editors, *Measurement: Definitions and Theories*, chapter 2, pages 18–37. Wiley, New York.

Stroustrup, B. (1997). *The C++ Programming Language*. Addison Wesley, third edition.

Tahgon, G., Nowell, A., and Jumars, P. (1984). Transport and breakdown of faecal pellets: biological and sedimentological consequences. *Limnological Oceanography*, 29(1):64–72.

Topchy, A. and Punch, W. F. (2001). Faster genetic programming based on local gradient search of numeric leaf values. In Spector, L., Goodman, E. D., Wu, A., Langdon, W. B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M. H., and Burke, E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 155–162, San Francisco, California, USA. Morgan Kaufmann.

Tsujimoto, T., Okada, T., and Omata, A. (1993). Field measurements of turbulent flow over vegetation on flood plain of river kakehaski. Khl progressive report, Hydraulic Laboratory, Kanazawa University.

Whigham, P. A. (1996a). *Grammatical Bias for Evolutionary Learning*. PhD thesis, School of Computer Science, University College, University of New South Wales, Australian Defence Force Academy.

Whigham, P. A. (1996b). Search bias, language bias, and genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 230–237, Stanford University, CA, USA. MIT Press.

Whigham, P. A. and Crapper, P. F. (1999). Time series modelling using genetic programming: An application to rainfall-runoff models. In Spector, L., Langdon, W. B., O'Reilly, U.-M., and Angeline, P. J., editors, *Advances in Genetic Programming 3*, chapter 5, pages 89–104. MIT Press, Cambridge, MA, USA.

Wong, M. L. and Leung, K. S. (1997). Evolutionary program induction directed by logic grammars. *Evolutionary Computation*, 5(2):143–180.

Yu, T. and Bentley, P. (1998). Methods to evolve legal phenotypes. In Eiben, A. E., Back, T., Schoenauer, M., and Schwefel, H.-P., editors, *Fifth International Conference on Parallel Problem Solving from Nature*, volume 1498 of *LNCS*, pages 280–291, Amsterdam. Springer.

Yu, T. and Clack, C. (1998). PolyGP: A polymorphic genetic programming system in haskell. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 416–421, University of Wisconsin, Madison, Wisconsin, USA. Morgan Kaufmann.

Zhang, B.-T. (2000). Bayesian methods for efficient genetic programming. *Genetic Programming and Evolvable Machines*, 1(3):217–242.

Zhang, B.-T. and Mühlenbein, H. (1994). Synthesis of sigma-pi neural networks by the breeder genetic programming. In *Proceedings of IEEE International Conference on Evolutionary Computation (ICEC-94), World Congress on Computational Intelligence*, pages 318–323, Orlando, Florida, USA. IEEE Computer Society Press.

Zhang, B.-T. and Mühlenbein, H. (1996). Adaptive fitness functions for dynamic growing/pruning of program trees. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 12, pages 241–256. MIT Press, Cambridge, MA, USA.

Zyserman, J. A. and Fredsøe, J. (1994). Data analysis of bed concentration of suspended sediment. *Journal of Hydraulic Engineering*, 120(9):1021–1042.