Data Mining using Genetic Programming

Classification and Symbolic Regression

J. Eggermont

Data Mining using Genetic Programming

Classification and Symbolic Regression

proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus Dr. D.D. Breimer,
hoogleraar in de faculteit der Wiskunde en
Natuurwetenschappen en die der Geneeskunde,
volgens besluit van het College voor Promoties
te verdedigen op woensdag 14 september 2005
klokke 15.15 uur

door

Jeroen Eggermont
geboren te Purmerend
in 1975

Promotiecommissie

Promotor:	Prof. Dr. J.N. Kok	
Co-promotor:	Dr. W.A. Kosters	
Referent:	Dr. W.B. Langdon	(University of Essex)
Overige leden:	Prof. Dr. T.H.W. Bäck	
	Prof. Dr. A.E. Eiben	(Vrije Universiteit Amsterdam)
	Prof. Dr. G. Rozenberg	
	Prof. Dr. S.M. Verduyn Lunel	

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

ISBN-10: 90-9019760-5
ISBN-13: 978-90-9019760-9

Contents

1 Introduction **1**
- 1.1 Data Mining . . . 2
  - 1.1.1 Classification and Decision Trees . . . 2
  - 1.1.2 Regression . . . 3
- 1.2 Evolutionary Computation . . . 3
- 1.3 Genetic Programming . . . 4
- 1.4 Motivation . . . 5
- 1.5 Overview of the Thesis . . . 6
- 1.6 Overview of Publications . . . 8

2 Classification Using Genetic Programming **9**
- 2.1 Introduction . . . 9
- 2.2 Decision Tree Representations for Genetic Programming . . . 10
- 2.3 Top-Down Atomic Representations . . . 14
- 2.4 A Simple Representation . . . 15
- 2.5 Calculating the Size of the Search Space . . . 16
- 2.6 Multi-layered Fitness . . . 18
- 2.7 Experiments . . . 19
- 2.8 Results . . . 22
- 2.9 Fitness Cache . . . 28
- 2.10 Conclusions . . . 30

3 Refining the Search Space **33**
- 3.1 Introduction . . . 33
- 3.2 Decision Tree Construction . . . 35
  - 3.2.1 Gain . . . 35
  - 3.2.2 Gain_ratio . . . 38

3.3 Representations Using Partitioning . . . 40
3.4 A Representation Using Clustering . . . 43
3.5 Experiments and Results . . . 44
3.5.1 Search Space Sizes . . . 45
3.5.8 Scaling . . . 53
3.6 Conclusions . . . 54

4 Evolving Fuzzy Decision Trees 57
4.1 Introduction . . . 57
4.2 Fuzzy Set Theory . . . 59
4.2.1 Fuzzy Logic . . . 60
4.3 Fuzzy Decision Tree Representations . . . 61
4.3.1 Fuzzification . . . 62
4.3.2 Evaluation Using Fuzzy Logic . . . 65
4.4 Experiments and Results . . . 66
4.4.7 Comparing Fuzzy and Non-Fuzzy . . . 74
4.5 A Fuzzy Fitness Measure . . . 74
4.6 Conclusions . . . 77

5 Introns: Detection and Pruning 79
5.1 Introduction . . . 79
5.2 Genetic Programming Introns . . . 80
5.3 Intron Detection and Pruning . . . 81
5.3.1 Intron Subtrees . . . 84
5.3.2 Intron Nodes . . . 88
5.3.3 The Effect of Intron Nodes on the Search Space . . . 91
5.4 Experiments and Results . . . 94
5.4.1 Tree Sizes . . . 95
5.4.2 Fitness Cache . . . 99
5.5 Conclusions . . . 105

6 Stepwise Adaptation of Weights 107
6.1 Introduction . . . 107
6.2 The Method . . . 109
6.3 Symbolic Regression . . . 111
6.3.1 Experiments and Results: Koza functions . . . 112
6.3.2 Experiments and Results: Random Polynomials . . . 117
6.4 Data Classification . . . 120

6.4.1 Experiments and Results 121
6.5 Conclusions 124

A Tree-based Genetic Programming **133**
A.1 Initialization 133
A.1.1 Ramped Half-and-Half Method 135
A.2 Genetic Operators 136
A.2.1 Crossover 138
A.2.2 Mutation 139

Bibliography **141**

Nederlandse Samenvatting **153**

Acknowledgements **157**

Curriculum Vitae **159**

1 Introduction

Sir Francis Bacon said about four centuries ago: "*Knowledge is Power*". If we look at today's society, information is becoming increasingly important. According to [73] about five exabytes (5×10^{18} bytes) of new information were produced in 2002, 92% of which on magnetic media (e.g., hard-disks). This was more than double the amount of information produced in 1999 (2 exabytes). However, as Albert Einstein observed: "*Information is not Knowledge*".

One of the challenges of the large amounts of information stored in databases is to find or extract potentially useful, understandable and novel patterns in data which can lead to new insights. To quote T.S. Eliot: "*Where is the knowledge we have lost in information*?" [35]. This is the goal of a process called *Knowledge Discovery in Databases* (KDD) [36]. The KDD process consists of several phases: in the Data Mining phase the actual discovery of new knowledge takes place.

The outline of the rest of this introduction is as follows. We start with an introduction of Data Mining and more specifically the two subject areas of Data Mining we will be looking at: classification and regression. Next we give an introduction about evolutionary computation in general and tree-based genetic programming in particular. In Section 1.4 we give our motivation for using genetic programming for Data Mining. Finally, in the last sections we give an overview of the thesis and related publications.

1.1 Data Mining

Knowledge Discovery in Databases can be defined as "the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data"[36]. The KDD process consists of several steps one of which is the Data Mining phase. It is during the Data Mining phase of the KDD process that the actual identification, search or construction of patterns takes place. These patterns contain the "knowledge" acquired by the Data Mining algorithm about a collection of data. The goal of KDD and Data Mining is often to discover knowledge which can be used for predictive purposes [40]. Based on previously collected data the problem is to predict the future value of a certain attribute. We focus on two of such Data Mining problems: classification and regression. An example of classification or categorical prediction is whether or not a person should get credit from a bank. Regression or numerical prediction can for instance be used to predict the concentration of suspended sediment near the bed of a stream [62].

1.1.1 Classification and Decision Trees

In data classification the goal is to build or find a model in order to predict the category of data based on some predictor variables. The model is usually built using heuristics (e.g., entropy) or some kind of supervised learning algorithm. Probably the most popular form for a classification model is the decision tree. Decision tree constructing algorithms for data classification such as ID3 [86], C4.5 [87] and CART [14] are all loosely based on a common principle: *divide-and-conquer* [87]. The algorithms attempt to divide a training set T into multiple (disjoint) subsets such that each subset T_i belongs to a single target class. Since finding the smallest decision tree consistent with a specific training set is NP-complete [58], machine learning algorithms for constructing decision trees tend to be non-backtracking and greedy in nature. As a result they are relatively fast but depend heavily on the way the data set is divided into subsets.

Algorithms like ID3 and C4.5 proceed in a recursive manner. First an attribute A is selected for the root node and each of the branches to the child nodes corresponds with a possible value or range of values for this attribute. In this way the data set is split up into subsets according to the values of attribute A. This process is repeated recursively for each of the

branches using only the records that occur in a certain branch. If all the records in a subset have the same target class C the branch ends in a leaf node predicting target class C.

1.1.2 Regression

In regression the goal is similar to data classification except that we are interested in finding or building a model to predict numerical values (e.g., tomorrow's stock prices) rather than categorical or nominal values. In our case we will limit regression problems to 1-dimensional functions. Thus, given a set of values $X = \{x_1, \ldots, x_n\}$ drawn from a certain interval and a set of sample points $S = \{(x_i, f(x_i))|x_i \in X\}$ the object is to find a function $g(x)$ such that $f(x_i) \approx g(x_i)$ for all $x_i \in X$.

1.2 Evolutionary Computation

Evolutionary computation is an area of computer science which is inspired by the principles of natural evolution as introduced by Charles Darwin in "*On the Origin of Species: By Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*" [17] in 1859. As a result evolutionary computation draws much of its terminology from biology and genetics.

In evolutionary computation the principles of evolution are used to search for (approximate) solutions to problems using the computer. The problems to which evolutionary computation can be applied have to meet certain requirements. The main requirement is that the quality of that possible solution can be computed. Based on these computed qualities it should be possible to sort any two or more possible solutions in order of solution quality. Depending on the problem, there also has to be a test to determine if a solution solves the problem.

In Algorithm 1 we present the basic form of an evolutionary algorithm. At the start of the algorithm a set or *population* of possible solutions to a problem is generated. Each of those possible solutions, also called *individuals*, is evaluated to determine how well it solves the problem. This evaluation is called the *fitness* of the individual. After the initial population has been created, the actual evolutionary process starts. This is essentially an iteration of steps applied to the population of candidate solutions.

Algorithm 1 The basic form of an evolutionary algorithm.

initialize P_0
evaluate P_0
$t = 0$
while not *stop criterion* **do**
  $parents \leftarrow select\_parents(P_t)$
  $offspring \leftarrow variation(parents)$
  evaluate *offspring* (and if necessary P_t)
  select the new population P_{t+1} from P_t and *offspring*
  $t = t + 1$
od

The first step is to select which candidate solutions are best suited to serve as the *parents* for the future generation. This selection is usually done in such a way that candidate solutions with the best performance are chosen the most often to serve as parent. In the case of evolutionary computation the offspring are the result of the *variation* operator applied to the parents. Just as in biology offspring are similar but generally not identical to their parent(s). Next, these newly created individuals are evaluated to determine their fitness, and possibly the individuals in the current population are re-evaluated as well (e.g., in case the fitness function has changed). Finally, another selection takes place which determines which of the offspring (and potentially the current individuals) will form the new population. These steps are repeated until some kind of stop criterion is satisfied, usually when a maximum number of generations is reached or when the best individual is "good" enough.

1.3 Genetic Programming

There is no single representation for an individual used in evolutionary computation. Usually the representation of an individual is selected by the user based on the type of problem to be solved and personal preference. Historically we can distinguish the following subclasses of evolutionary computation which all have their own name:

- Evolutionary Programming (EP), introduced by Fogel et al. [37]. EP originally was based on Finite State Machines.

- Evolution Strategies (ES), introduced by Rechenberg [88] and Schwefel [93]. ES uses real valued vectors mainly for parameter optimization.

- Genetic Algorithms (GA), introduced by Holland [55]. GA uses fixed length bitstrings to encode solutions.

In 1992 Koza proposed a fourth class of evolutionary computation, named Genetic Programming (GP), in the publication of his monograph entitled "*Genetic Programming: On the Programming of Computers by Natural Selection*" [66]. In his book Koza shows how to evolve computer programs, in LISP, to solve a range of problems, among which symbolic regression. The programs evolved by Koza are in the form of parse trees, similar to those used by compilers as an intermediate format between the programming language used by the programmer (e.g., C or Java) and machine specific code. Using parse trees has advantages since it prevents syntax errors, which could lead to invalid individuals, and the hierarchy in a parse tree resolves any issues regarding function precedence.

Although genetic programming was initially based on the evolution of parse trees the current scope of Genetic Programming is much broader. In [4] Banzhaf et al. describe several GP systems using either trees, graphs or linear data structures for program evolution and in [70] Langdon discusses the evolution of data structures.

Our main focus is on the evolution of decision tree structures for data classification and we will therefore use a classical GP approach using trees. The specific initialization and variation routines for tree-based Genetic Programming can be found in Appendix A.

1.4 Motivation

We investigate the potential of tree-based Genetic Programming for Data Mining, more specifically data classification. At first sight evolutionary computation in general, and genetic programming in particular, may not seem to be the most suited choice for data classification. Traditional machine learning algorithms for decision tree construction such as C4.5 [87], CART [14] and OC1 [78] are generally faster.

The main advantage of evolutionary computation is that it performs a global search for a model, contrary to the local greedy search of most traditional machine learning algorithms [39]. ID3 and C4.5, for example, evaluate

the impact of each possible condition on a decision tree, while most evolutionary algorithms evaluate a model as a whole in the fitness function. As a result evolutionary algorithms cope well with attribute interaction [39, 38].

Another advantage of evolutionary computation is the fact that we can easily choose, change or extend a representation. All that is needed is a description of what a tree should look like and how to evaluate it. A good example of this can be found in Chapter 4 where we extend our decision tree representation to fuzzy decision trees, something which is much more difficult (if not impossible) for algorithms like C4.5, CART and OC1.

1.5 Overview of the Thesis

In the first chapters we look at decision tree representations and their effect on the classification performance in Genetic Programming. In Chapter 2 we focus our attention on decision tree representations for data classification. Before introducing our first decision tree representation we give an overview and analysis of other tree-based Genetic Programming (GP) representations for data classification.

We introduce a *simple* decision tree representation by defining which (internal) nodes can occur in a tree. Using this *simple* representation we investigate the potential and complexity of using tree-based GP algorithms for data classification tasks.

Next in Chapter 3 we introduce several new GP representations which are aimed at "refining" the search space. The idea is to use heuristics and machine learning methods to decrease and alter the search space for our GP classifiers, resulting in better classification performance. A comparison of our new GP algorithms and the *simple* GP shows that when a search space size is decreased using our methods, the classification performance of a GP algorithm can be greatly improved.

Standard decision tree representations have a number of limitations when it comes to modelling real world concepts and dealing with noisy data sets. In Chapter 4 we attack these problems by evolving *fuzzy* decision trees. Fuzzy decision trees are based on fuzzy logic and fuzzy set theory, unlike "standard" decision trees which are based on Boolean logic and set theory. By using fuzzy logic in our decision tree representations we intend to make our fuzzy decision trees more robust towards faulty and polluted input data. A comparison between the non-fuzzy representations of Chapter 3 and their *fuzzy* versions

confirm this as our GP algorithms are especially good in those cases in which the non-fuzzy GP algorithms failed.

In Chapter 5 we show how the understandability and speed of our GP classifiers can be enhanced, without affecting the classification accuracy. By analyzing the decision trees evolved by our GP algorithms, we can detect the unessential parts, called (GP) *introns*, in the discovered decision trees. Our results show that the detection and pruning of *introns* in our decision trees greatly reduces the size of the trees. As a result the decision trees found are easier to understand although in some cases they can still be quite large. The detection and pruning of *intron nodes* and *intron subtrees* also enables us to identify syntactically different trees which are semantically the same. By comparing and storing pruned decision trees in our fitness cache, rather than the original unpruned decision trees, we can greatly improve its effectiveness. The increase in cache hits means that less individuals have to be evaluated resulting in reduced computation times.

In the last chapter (Chapter 6) we focus our attention on another important part of our GP algorithms: the fitness function. Most evolutionary algorithms use a static fitness measure $f(x)$ which given an individual x always returns the same fitness value. Here we investigate an adaptive fitness measure, called Stepwise Adaptation of Weights (SAW). The SAW technique has been developed for and successfully used in solving constraint satisfaction problems with evolutionary computation. The idea behind the SAW method is to adapt the fitness function of an evolutionary algorithm during an evolutionary run in order to escape local optima, and improve the quality of the evolved solutions. We will demonstrate how the SAW mechanism can be applied to both data classification and symbolic regression problems using Genetic Programming. Moreover, we show how the different parameters of the SAW method influence the results for Genetic Programming applied to regression and classification problems.

1.6 Overview of Publications

Here we give an overview of the way in which parts of this thesis have been published.

Chapter 2: Classification using Genetic Programming
Parts of this chapter are published in the proceedings of the Fifthteenth Belgium-Netherlands Conference on Artificial Intelligence (BNAIC'03) [25].

Chapter 3: Refining the Search Space
A large portion of this chapter is published in the proceedings of the Nineteenth ACM Symposium on Applied Computing (SAC 2004) [27].

Chapter 4: Evolving Fuzzy Decision Trees
The content of this chapter is based on research published in the Proceedings of the Fifth European Conference on Genetic Programming (EuroGP'02) [21]. An extended abstract is published in the Proceedings of the Fourteenth Belgium-Netherlands Conference on Artificial Intelligence (BNAIC'02) [20].

Chapter 5: Introns: Detection and Pruning
Parts of this chapter are published in the Proceeding of the Eighth International Conference on Parallel Problem Solving from Nature (PPSN VIII, 2004) [26].

Chapter 6: Stepwise Adaptation of Weights
The parts of this chapter concerning classification are based on research published in the Proceedings of the Second European Workshop on Genetic Programming (EuroGP'99) [22], Advances in Intelligent Data Analysis, Proceedings of the Third International Symposium (IDA'99) [24], and as an extended abstract in the Proceedings of the Eleventh Belgium-Netherlands Conference on Artificial Intelligence (BNAIC'99) [23]. Parts of this chapter regarding symbolic regression are published in the Proceedings the Twelfth Belgium-Netherlands Conference on Artificial Intelligence (BNAIC'00) [28] and the Proceedings of the Fourth European Conference on Genetic Programming (EuroGP'01) [29].

2 Classification Using Genetic Programming

We focus our attention on decision tree representations for data classification. Before introducing our first decision tree representation we give an overview and analysis of other tree-based Genetic Programming (GP) representations for data classification.

Then we introduce a simple decision tree representation by defining which internal and external nodes can occur in a tree. Using this *simple* representation we investigate the potential and complexity of tree-based GP algorithms for data classification tasks and compare our *simple* GP algorithm to other evolutionary and non-evolutionary algorithms using a number of data sets.

2.1 Introduction

There are a lot of possible representations for classifiers (e.g., decision trees, rule-sets, neural networks) and it is not efficient to try to write a genetic programming algorithm to evolve them all. In fact, even if we choose one type of classifier, e.g., decision trees, we are forced to place restrictions on the shape of the decision trees. As a result the final solution quality of our decision trees is partially dependent on the chosen representation; instead of searching in the space of all possible decision trees we search in the space determined by the limitations we place on the representation. However, this does not mean that this search space is by any means small as we will show for different data sets.

The remainder of this chapter is as follows. In Section 2.2 we will give an overview of various decision tree representations which have been used in combination with Genetic Programming (GP) and discuss some of their strengths and weaknesses. In the following section we introduce the notion of *top-down atomic* representations which we have chosen as the basis for all the decision tree representations used in this thesis. A *simple* GP algorithm for data classification is introduced in Section 2.4. In Section 2.5 we will formulate how we can calculate the size of the search space for a specific *top-down atomic* representation and data set. We will then introduce the first *top-down atomic* representation which we have dubbed the *simple* representation. This *simple* representation will be used to investigate the potential of GP for data classification. The chapter continues in Section 2.7 with a description of the experiments, and the results of our *simple atomic* GP on those experiments in Section 2.8. In Section 2.9 we discuss how the computation time of our algorithm can be reduced by using a fitness cache. Finally, in Section 2.10 we present conclusions.

2.2 Decision Tree Representations for Genetic Programming

In 1992 Koza [66, Chapter 17] demonstrated how genetic programming can be used for different classification problems. One of the examples shows how ID3 style decision trees (see Figure 2.1) can be evolved in the form of LISP S-expressions.

In another example the task is to classify whether a point (x, y) belongs to the first or second of two intertwining spirals (with classes $+1$ and -1). In this case the function set consists of mathematical operators $(+, -, \times, /, sin$ and $cos)$ and a decision-making function $(if - less - then - else -)$. The terminal set consists of random floating-point constants and variables x and y. Since a tree of this type returns a floating-point number, the sign of the tree outcome determines the class $(+1, -1)$. The same approach is also used in [44] and [98]. The major disadvantage of this type of representation is the difficulty of humans in understanding the information contained in these decision trees. An example of a decision tree using mathematical operators is shown in Figure 2.2.

A problem of both representations described above is that neither repre-

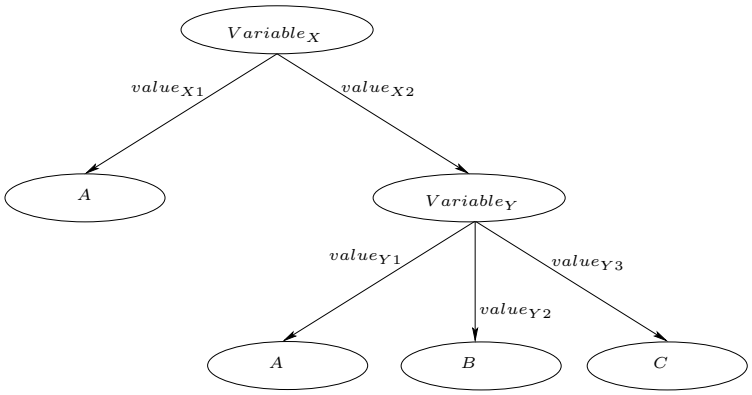

Figure 2.1: An example of an ID3 style decision tree. The tree first splits the data set on the two possible values of variable X ($Value_{X1}$ *and* $Value_{X2}$). The right subtree is then split into three parts by variable Y. The class outcome, A, B or C, is determined by the leaf nodes.

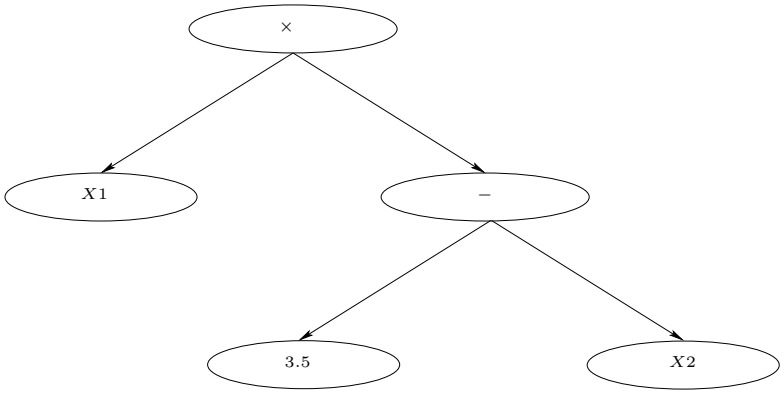

Figure 2.2: An example of a decision tree using mathematical operators in the function set and constants and variables in the terminal set. The sign of the tree outcome determines the class prediction.

sentation is designed to be used with both numerical and categorical inputs. For instance, if a variable X has 1,000 possible values then splitting the data set into a thousand parts will not result in a very understandable tree. In the case of the second represention, using mathematical functions, some operators in the function set (e.g., $+$, $-$, $\times$) cannot be used with categorical values such as *Male*, *Female*, *Cold* or *Warm*.

In an ideal case, a decision tree representation would be able to correctly handle both numerical and categorical values. Thus, numerical variables and values should only be compared to numerical values or variables and only be used in numerical functions. Similarly, categorical variables and values should only be compared to categorical variables or values. This is a problem for the standard GP operators (crossover, mutation and initialization) which assume that the output of any node can be used as the input of any other node. This is called the closure property of GP which ensures that only syntactically valid trees are created.

A solution to the closure property problem of GP is to use strongly typed genetic programming introduced by Montana [77]. Strongly typed GP uses special initialization, mutation and crossover operators. These special operators make sure that each generated tree is syntactically correct even if tree-nodes of different data types are used. Because of these special operators an extensive function set consisting of arithmetic $(+, -, \times, /)$, comparison $(\leq, >)$ and logical operators (and, or, if) can be used. An example of a strongly typed GP representation for classification was presented by Bhattacharyya, Pictet and Zumbach [6].

Another strongly typed GP representation was introduced by Bot [11, 12] in 1999. This *linear classification* GP algorithm uses a representation for oblique decision trees [78]. An example tree can be seen in Figure 2.3.

In 1998 a new representation was introduced, independent of each other, by Hu [57] and van Hemert [51] (see also [22, 24]) which copes with the closure property in another way. Their *atomic* representation booleanizes all attribute values in the terminal set using atoms. Each atom is syntactically a predicate of the form ($variable_i$ *operator constant*) where *operator* is a comparison operator (e.g., $\leq$ and $>$ for continuous attributes, $=$ for nominal or Boolean attributes). Since the leaf nodes always return a Boolean value (true or false) the function set consists of Boolean functions (e.g., *and*, *or*) and possibly a decision making function ($if - then - else$). An example of a decision tree using the *atomic* representation can be seen in Figure 2.4.

A similar representation was introduced by Bojarzcuk, Lopes and Freitas [10] in 1999. They used first-order logic rather than propositional logic. This first-order logic representation uses a predicate of the form ($variable_1$ *operator* $variable_2$) where $variable_1$ and $variable_2$ have the same data type.

In 2001 the first fuzzy decision tree representation for GP was introduced by Mendes et al. [75]. This fuzzy decision tree representation is similar to the *atomic* representation of Hu and van Hemert but it uses a function set

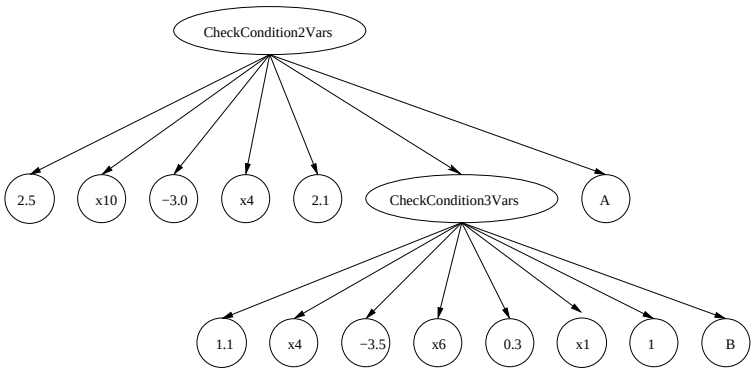

Figure 2.3: An example of an oblique decision tree from [11]. The leftmost children of function nodes (in this case *CheckCondition2Vars* and *CheckCondition3Vars*) are weights and variables for a linear combination. The rightmost children are other function nodes or target classes (in this case A or B). Function node *CheckCondition2Vars* is evaluated as: if $2.5x_{10} - 3.0x_4 \leq 2.1$ then evaluate the *CheckCondition3Vars* node in a similar way; otherwise the final classification is A and the evaluation of the decision tree on this particular case is finished.

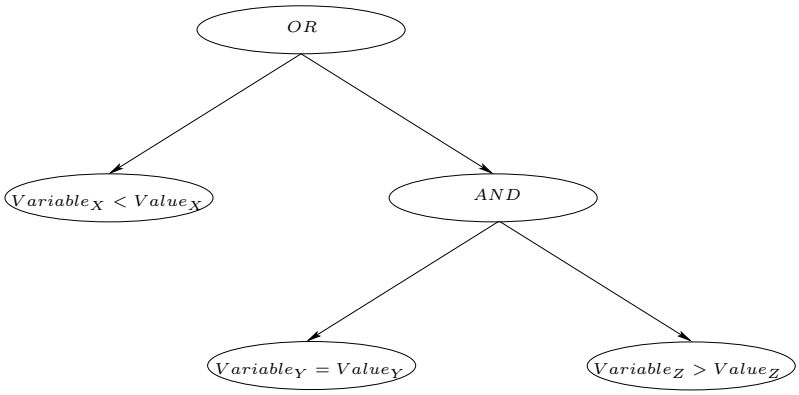

Figure 2.4: An example of a decision tree using an *atomic* representation. Input variables are booleanized by the use of atoms in the leaf nodes. The internal nodes consist of Boolean functions and possibly a decision making function.

consisting of fuzzy-logic operators (e.g., fuzzy *and*, fuzzy *or*, fuzzy *not*). The terminal set consists of atoms. Each atom is of the form (*variable* =

value). For a categorical attribute *value* corresponds to one of the possible values. In the case of numerical attributes *value* is a linguistic value (such as *Low*, *Medium* or *High*) corresponding with a fuzzy set [5, 101]. For each numerical attribute a small number of fuzzy sets are defined and each possible value of an attribute is a (partial) member of one or more of these sets. In order to avoid generating invalid rule antecedents some syntax constraints are enforced making this another kind of strongly typed GP.

In 2001 Rouwhorst [89] used a representation similar to that of decision tree algorithms like C4.5 [87]. Instead of having atoms in the leaf nodes it has conditional atoms in the internal nodes and employs a terminal set using classification assignments.

In conclusion there is a large number of different possibilities for the representation of decision trees. We will use a variant of the atomic representation which we discuss in the next section.

2.3 Top-Down Atomic Representations

An *atomic* tree is evaluated in a bottom-up fashion resulting in a Boolean value *true* or *false* corresponding with two classes. Because an *atomic* tree only returns a Boolean value it is limited to binary classification problems. In order to evolve decision trees for n-ary classification problems, without having to split them into n binary classification problems, we propose a decision tree representation using atoms that is evaluated in a top-down manner. Unlike the *atomic* representation of van Hemert which only employs atoms in its terminal set, a *top-down atomic* representation uses atoms in both the internal and leaf nodes. Each atom in an internal node is syntactically a predicate of the form ($attribute_i$ *operator* *value*(s)), where *operator* is a comparison operator (e.g., $<$, $>$ or $=$). In the leaf nodes we have *class assignment* atoms of the form ($class := C$), where C is a category selected from the domain of the attribute to be predicted. A small example tree can be seen in Figure 2.5. A *top-down atomic* tree classifies an instance I by traversing the tree from root to leaf node. In each non-leaf node an atom is evaluated. If the result is true the right branch is traversed, else the left branch is taken. This is done for all internal nodes until a leaf node containing a *class assignment* node is reached, resulting in the classification of the instance.

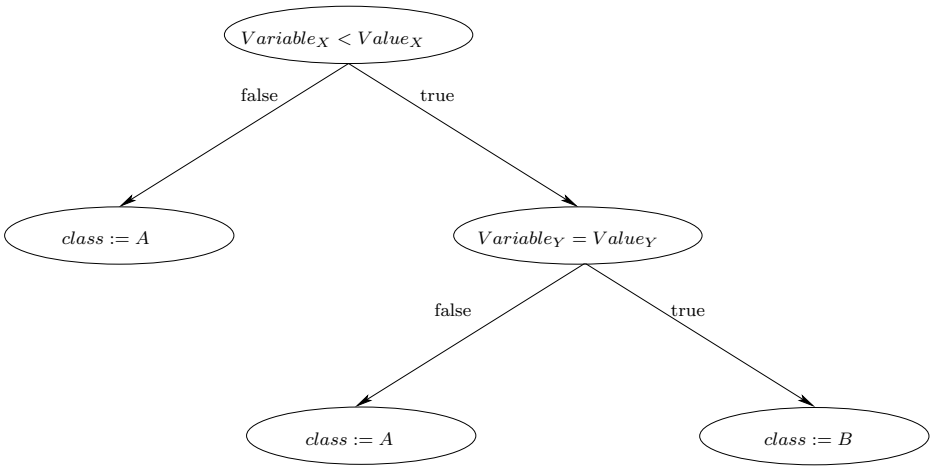

Figure 2.5: An example of a *top-down atomic* tree.

2.4 A Simple Representation

By using a *top-down atomic* representation we have defined in a general way what our decision trees look like and how they are evaluated. We can define the precise decision tree representation by specifying what atoms are to be used. Here we will introduce a simple, but powerful, decision tree representation that uses three different types of atoms based on the data type of an atom's *attribute*. For non-numerical attributes we use atoms of the form ($variable_i = value$) for each possible *attribute-value* combination found in the data set. For numerical attributes we also define a single operator: less-than ($<$). Again we use atoms for each possible *attribute-value* combination found in the data set. The idea in this approach is that the GP algorithm will be able to decide the best *value* at a given point in a tree. This *simple* representation is similar to the representation used by Rouwhorst [89]. An example of a *simple* tree can be seen in Figure 2.6.

Example 2.4.1 Observe the data set T depicted in Table 2.1.

In the case of our *simple* representation the following atoms are created:

- Since attribute **A** has four possible values {1,2,3,4} and is numerical valued we use the less-then operator ($<$): $(A < 1)$, $(A < 2)$, $(A < 3)$ and $(A < 4)$.

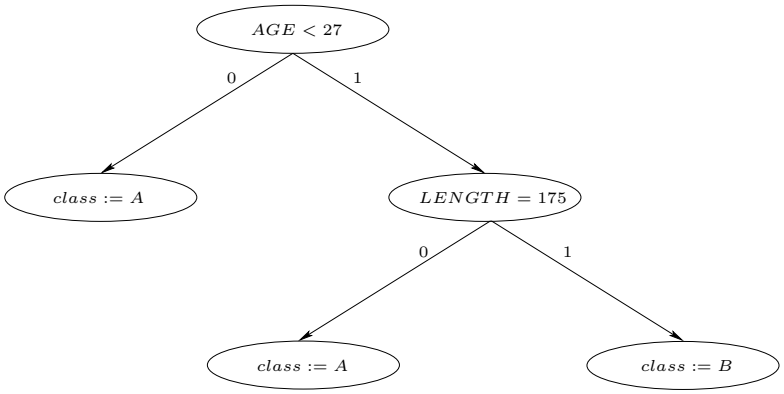

Figure 2.6: An example of a *simple* GP tree.

Table 2.1: A small data set with two input variables, **A** and **B**, and a target variable **class**.

A	**B**	**class**
1	a	yes
2	b	yes
3	c	no
4	d	no

- Attribute **B** is non-numerical and thus we use the is-equal operator ($=$): $(B = a)$, $(B = b)$, $(B = c)$ and $(B = d)$.
- Finally for the target class we have two terminal nodes: $(class := yes)$ and $(class := no)$.

2.5 Calculating the Size of the Search Space

Since every decision tree using our *top-down atomic* representation is also a full binary tree [15, Chapter 5.5.3] we can calculate the size of the search space for each specific *top-down atomic* representation and data set. In order to calculate the size of the search space for GP algorithms using a *top-down atomic* representation and a given data set we will introduce two well-known facts from discrete mathematics.

Let N be the number of tree nodes. The total number of binary trees with N nodes is the Catalan number

$$Cat(N) = \frac{1}{N+1}\binom{2N}{N}. \tag{2.1}$$

In a full binary tree each node is either a leaf node (meaning 0 children) or has two exactly 2 children. Let n be the number of internal tree nodes. The total number of tree nodes N in a full binary tree with n internal tree nodes is:

$$N = 2n + 1. \tag{2.2}$$

We can now combine these two equations into the following lemma:

Lemma 2.5.1 *The total number of full binary trees with* $2n + 1$ *nodes is* $\frac{1}{n+1}\binom{2n}{n}$.

Proof Let B be a tree with n nodes. In order to transform this tree into a full binary tree with $2n+1$ nodes we need to add $n+1$ nodes. This can only be done in one way. □

Since in a *top-down atomic* tree the contents of a node is dependent on the set of internal nodes and the set of external nodes we can compute the total number of *top-down atomic* trees with a maximum tree size of N nodes, a set of internal nodes I and a set of terminal nodes T as follows.

Lemma 2.5.2 *The total number of top-down atomic trees with at most* N *nodes (*N *odd), a set of internal nodes* I *and a set of terminal nodes* T *is*

$$\sum_{n=1}^{\frac{N-1}{2}} Cat(n) \times |I|^n \times |T|^{n+1}.$$

Example 2.5.1 In Example 2.4.1 we showed which atoms are created for the *simple* GP representation in the case of the example data set from Table 2.1. Once we have determined the atoms for the *simple* GP representation we can calculate the resulting search space size using Lemma 2.5.2. We will restrict the maximum size of our decision trees to 63 nodes, which is the number of nodes in a complete binary tree [15, Chapter 5.5.3] of depth 5.

Thus, given a maximum tree size of 63 nodes, the example data set in Table 2.1 and a *simple atomic* representation we get:

- $I = \{(A < 1), (A < 2), (A < 3), (A < 4), (B = a), (B = b), (B = c), (B = d)\ \}$.
- $T = \{(class := yes), (class := no)\}$
- $N = 63$.

In this case the total number of possible decision trees, and thus the search space, for our *simple* GP algorithm is 6.29×10^{53}.

2.6 Multi-layered Fitness

Although we will compare our *top-down atomic* GP algorithms to other data classification algorithms based on their classification performance, there is a second objective for our *top-down atomic* GPs which is also important: understandability of the classifier. As we discussed in Section 2.2, some early GP algorithms for data classification used the representations with mathematical functions. The major disadvantage of this type of representation is the difficulty with which humans can understand the information contained in these decision trees. The *simple* representation introduced in the previous section is similar to the decision trees constructed by C4.5 and much easier to understand. However, even the most understandable decision tree representation can result in incomprehensible trees if the trees become too large.

One of the problems of variable length evolutionary algorithms, such as tree-based genetic programming, is that the genotypes of the individuals tend to increase in size until they reach the maximum allowed size. This phenomenon is, in genetic programming, commonly refered to as *bloat* [4, 97] and will be discussed in more detail in Chapter 5.2.

There are several methods to counteract *bloat* [69, Chapter 11.6]. We use a combination of two methods. The first method is a size limit: we use a built in system which prunes decision trees that have more than a pre-determined number of nodes, in our case 63.

The second method is the use of a *multi-layered* fitness. A multi-layered fitness is a fitness which consists of several fitness measures or objectives which are ranked according to their importance. In the case of our *simple* representation we use a multi-layered fitness consisting of two fitness mea-

sures which we want to minimize. The primary, and most important, fitness measure for a given individual tree x is the misclassification percentage:

$$\mathit{fitness}_{\mathit{standard}}(x) = \frac{\sum_{r \in \mathit{training\ set}} \chi(x, r)}{|\mathit{training\ set}|} \times 100\%, \tag{2.3}$$

where $\chi(x, r)$ is defined as:

$$\chi(x, r) = \begin{cases} 1 & \text{if } x \text{ classifies record } r \text{ incorrectly;} \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

The secondary fitness measure is the number of tree nodes. When the fitness of two individuals is to be compared we first look at the primary fitness. If both individuals have the same misclassification percentage we compare the secondary fitness measures. This corresponds to the suggestion in [46] that size should only be used as a fitness measure when comparing two individuals with otherwise identical fitness scores.

2.7 Experiments

We will compare our *top-down atomic* GP representations to some other evolutionary and machine learning algorithms using several data sets from the UCI machine learning data set repository [7]. An overview of the different data sets is given in Table 2.2.

Table 2.2: An overview of the data sets used in the experiments.

data set	**records**	**attributes**	**classes**
Australian Credit	690	14	2
German Credit	1000	23	2
Pima Indians Diabetes	768	8	2
Heart Disease	270	13	2
Ionosphere	351	34	2
Iris	150	4	3

Each algorithm is evaluated using 10-fold cross-validation and the performance is the average misclassification error over 10 folds. In 10-fold cross-

validation the total data set is divided into 10 parts. Each part is chosen once as the test set while the other 9 parts form the training set.

In order to compare our results to other evolutionary techniques we will also mention the results of two other evolutionary classification systems, CEFR-MINER [75] and ESIA [72], as reported in these respective papers. CEFR-MINER is a GP system for finding fuzzy decision trees and ESIA builds crisp decision trees using a genetic algorithm. Both also used a 10-fold cross-validation.

We also mention the results as reported in [43] of a number of non-evolutionary decision tree algorithms: Ltree[43], OC1 [78] and C4.5 [87]. We also report a *default* classification performance which is obtained by always predicting the class which occurs most in the data set. We performed 10 independent runs for our GP algorithms to obtain the results.

Table 2.3: The main GP parameters.

Parameter	**Value**
Population Size	100
Initialization	ramped half-and-half
Initial Maximum Tree Depth	6
Maximum Number of Nodes	63
Parent Selection	tournament selection
Tournament Size	5
Evolutionary Model	$(100, 200)$
Crossover Rate	0.9
Crossover Type	swap subtree
Mutation Rate	0.9
Mutation Type	branch mutation
Stop Condition	99 generations

The settings used for our GP system are displayed in Table 2.3. Most surprising is probably the high mutation rate (0.9) we used. The reason for choosing this high mutation rate is to explore a larger part of the search space. Early experiments using smaller mutation rates (e.g., 0.1, 0.3, 0.5, 0.7) showed that only a small number of the evaluated individuals were unique.

In our GP system we use the standard GP mutation and recombination operators for trees. The mutation operator replaces a subtree with a randomly created subtree and the crossover operator exchanges subtrees between two individuals. The population was initialized using the ramped half-and-half initialization [4, 66] method to create a combination of full and non-full trees with a maximum tree depth of 6.

One of the problems of supervised learning algorithms is finding the right balance between learning a model that closely fits the training data and learning a model that works well on unseen problem instances. If an algorithm produces a model that focusses too closely on the training samples at the expense of generalization power it is said to have *overfitted* the data.

A method to prevent overfitting during the training of an algorithm is to use a validation set: a validation set is a part of the data set disjoint from both the training and test set. When the classification performance on the validation set starts to decrease the algorithm can be overfitting the training set. If overfitting is detected the training is usually stopped. However, there is no guarantee that using a validation set will result in optimal classification performance on the test set. In the case of limited amounts of data this can be problematic because it also decreases the number of records in the training set. We will therefore try to prevent or reduce overfitting by other means which we discuss next:

- In [83] Paris et al. explore several potential aspects of overfitting in genetic programming. One of their conclusions is that big populations do not necessarily increase the performance and can even decrease performance.

- In [59] Jensen et al. show that overfitting occurs because a large number of models gives a high probability that a model will be found that fits the training data well purely by chance.

In the case of evolutionary computation the number of evaluated individuals is determined by population size, the number of generations and the number of offspring produced per generation. In order to reduce the chance of overfitting we have therefore chosen to run our *simple* GP algorithm with a small population size and for a small number of generations. We use a generational model (comma strategy) with population size of 100 creating 200 children per generation. The 100 children with the best fitness are selected for the next generation. Parents are chosen by using 5-tournament

selection. We do not use elitism as the best individual is stored outside the population. Each newly created individual, whether through initialization or recombination, is automatically pruned to a maximum number of 63 nodes. The algorithm stops after 99 generations which means that at most 19.900 $(100 + 99 \times 200)$ unique individuals are evaluated.

The *simple* GP algorithm was programmed using the *Evolving Objects* library (EOlib) [64]. EOlib is an Open Source C++ library for all forms of evolutionary computation and is available from `http://eodev.sourceforge.net`.

2.8 Results

We performed 10 independent runs for our *simple* GP algorithm to obtain the results (presented in Tables 2.4, 2.5, 2.6, 2.7, 2.8 and 2.9). To obtain the average misclassification rates and standard deviations we first computed the average misclassification rate for each fold (averaged over 10 random seeds).

When available from the literature the results of CEFR-MINER, ESIA, Ltree, OC1 and C4.5 are reported. N/A indicates that no results were available. In each table the lowest average misclassification result ("the best result") is printed in **bold**.

To determine if the results obtained by our *simple* GP algorithm are statistically significantly different from the results reported for ESIA, CEFR-MINER, Ltree, OC1 and C4.5, we have performed two-tailed independent samples t-tests with a 95% confidence level ($p = 0.05$) using the reported mean and standard deviations. The null-hypothesis in each test is that the means of the two algorithms involved are equal.

2.8.1 The Australian Credit Data Set

The Australian Credit data set contains data from credit card applications and comes from the STATLOG data set repository [76] (part of the UCI data repository [7]). Since both attributes and classes have been encoded it is impossible to interpret the trees found by our algorithms. In the original data, on which the data set is based, 37 examples ($\approx 5\%$) had missing values. The UCI data repository reports that 8 of the 14 attributes are categorical but our algorithms treat all numerical values in the same way. The two target classes are quite evenly distributed with 307 examples (roughly 44.5%) for class 1 and 383 examples for class 2 ($\approx 55.5\%$). On this data set our *simple*

GP constructed a set of internal nodes of size 1167 and a set of terminal nodes of size 2 (the two classes). This means that the size of the search space of our *simple* GP on the Australian Credit data set is approximately 7.5×10^{120}.

Table 2.4: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Australian Credit data set.

algorithm	**average**	**s.d.**
simple GP	22.0	3.9
Ltree	**13.9**	4.0
OC1	14.8	6.0
C4.5	15.3	6.0
CEFR-MINER	N/A	
ESIA	19.4	0.1
default	44.5	

If we look at the results (see Table 2.4) of the Australian Credit data set we see that average misclassification performance of our *simple* GP algorithm is clearly not the best. Compared to the results of Ltree, OC1 and C4.5 our *simple* GP algorithm performs significantly worse while the difference in performance with ESIA is not statistically significant. All algorithms definitely offer better classification performance than *default* classification. The smallest tree found by our *simple* GP can be seen in Figure 2.7. Although it is very small it can classify the complete data set (no 10-fold cross-validation) with a misclassification percentage of only 14.5%.

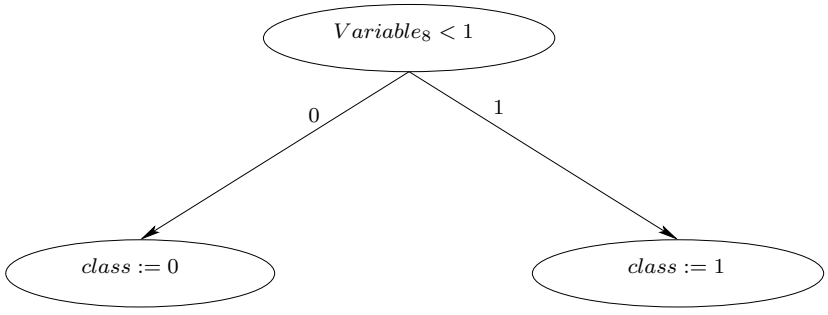

Figure 2.7: A *simple* tree found by our GP on the Australian Credit data set.

2.8.2 The German Credit Data Set

The German Credit data set also comes from the STATLOG data set repository [76]. The original data set consisted of a combination of symbolic and numerical attributes, but we used the version consisting of only numerical valued attributes. The data set is the largest data set used in our experiments with 1000 records of 24 attributes each. The two target classes are divided into 700 examples for class 1 and 300 examples for class 2. Although the data set itself is the largest one we used, the *simple* GP only constructed 269 possible internal nodes as well as 2 terminal nodes. As a result the search space of our *simple atomic* GP on the German Credit data set is much smaller (size $\approx 1.3 \times 10^{101}$) than on the Australian Credit data set.

Table 2.5: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the German Credit data set.

algorithm	**average**	**s.d.**
simple GP	27.1	2.0
Ltree	26.4	5.0
OC1	**25.7**	5.0
C4.5	29.1	4.0
CEFR-MINER	N/A	
ESIA	29.5	0.3
default	30.0	

Looking at the results on the German Credit data set (Table 2.5) we see that our *simple* GP performs a little better than C4.5 on average and a little worse than Ltree and OC1, but the differences are not statistically significant. Our *simple* GP algorithm does have a significantly lower average misclassification rate than ESIA which performs only slighlty better than *default* classification.

2.8.3 The Pima Indians Diabetes Data Set

The Pima Indians Diabetes data set is an example of a data set from the medical domain. It contains a number of physiological measurements and medical test results of 768 females of Pima Indian heritage of at least 21

years old. The classification task consists of predicting whether a patient would test positive for diabetes according to criteria from the WHO (World Health Organization). The data set contains 500 positive examples and 268 negative examples. In [76] a 12-fold cross-validation was used but we decided on using a 10-fold cross-validation in order to compare our results to those of the other algorithms. Because of the 10-fold cross validation the data set was divided into 8 folds of size 77 and 2 folds of size 76. Our *simple* GP constructed 1254 internal nodes as well as 2 terminal nodes for the target classes. This results in a search space on the Pima Indians Diabetes data set of size $\approx 7.0 \times 10^{121}$.

Table 2.6: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Pima Indians Diabetes data set.

algorithm	**average**	**s.d.**
simple GP	26.3	3.6
Ltree	**24.4**	5.0
OC1	28.0	8.0
C4.5	24.7	7.0
CEFR-MINER	N/A	
ESIA	29.8	0.2
default	34.9	

Although this data set is reported to be quite difficult by [76] it is possible to get good classification performance using linear discrimination on just one attribute. Although the average misclassification performance of our *simple* GP algorithm is somewhat higher than that of Ltree and C4.5, the difference is not statistically significant. Our *simple* GP algorithm does again perform significantly better than ESIA, while the difference in performance with OC1 is not significant.

2.8.4 The Heart Disease Data Set

The Heart Disease data set is another example of a medical data set. In the data set results are stored of various medical tests carried out on a patient. The data set is also part of the STATLOG [76] data set repository. The data

set was constructed from a larger data set consisting of 303 records with 75 attributes each. For various reasons some records and most of the attributes were left out when the Heart Disease data set of 270 records and 13 input variables was constructed. The classification task consists of predicting the presence or absence of Heart Disease. The two target classes are quite evenly distributed with 56% of the patients (records) having no Heart Disease and 44% having some kind of Heart Disease present. The Heart Disease data set is quite small with only 270 records and 13 input variables. However, our *simple* GP still constructed 384 internal nodes as well as the 2 terminal nodes for the target classes, resulting in a search space of size $\approx 8.1 \times 10^{105}$, which is larger than that for the German Credit data set.

Table 2.7: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Heart Disease data set.

algorithm	**average**	**s.d.**
simple GP	25.2	4.8
Ltree	**15.5**	4
OC1	29.3	7
C4.5	21.1	8
CEFR-MINER	17.8	7.1
ESIA	25.6	0.3
default	44.0	

On this data set our *simple* GP algorithm performs significantly worse than the Ltree and CEFR-MINER algorithms. Compared to OC1, C4.5 and ESIA the differences in misclassification performance are not statistically significant.

2.8.5 The Ionosphere Data Set

The Ionosphere data set contains information of radar returns from the ionosphere. According to [7] the data was collected by a phased array of 16 high-frequency antennas with a total transmitted power in the order of 6.4 kilowatts. The target class consists of the type of radar return. A "good" radar return shows evidence of some type of structure of electrons in the

ionosphere while a "bad" return does not. Although the number of records is quite small (351) the number of attributes is the largest (34) of the data sets on which we have tested our algorithms. All attributes are continuous valued. Because our *simple* GP constructs a node for each possible value of continuous valued attributes it constructs no less than 8147 possible internal nodes as well as 2 terminal nodes for the target classes. This results in a search space of size $\approx 1.1 \times 10^{147}$. One fold consists of 36 records while the other 9 folds consist of 35 records each.

Table 2.8: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Ionosphere data set.

algorithm	**average**	**s.d.**
simple GP	12.4	3.8
Ltree	9.4	4.0
OC1	11.9	3.0
C4.5	**9.1**	5.0
CEFR-MINER	11.4	6.0
ESIA	N/A	
default	35.9	

If we look at the results the performance of *simple* GP algorithm seems much worse than that of Ltree and C4.5. However, the differences between the *simple* GP algorithm and the other algorithms are not statistically significant.

2.8.6 The Iris Data Set

The Iris data set is the only data set, on which we test our algorithms, with more than two target classes. The data set contains the sepal and petal length and width of three types of iris plants: Iris Setosa, Iris Versicolour and Iris Virginica. One of the plants is linearly separable from the others using a single attribute and threshold value. The remaining two classes are not linearly separable. All three classes are distributed equally (50 records each). Because of the small number of records and attributes (only 4) the *simple* GP constructs only 123 internal nodes and 3 terminal nodes for the 3 classes. This results in a search space of size $\approx 1.7 \times 10^{96}$.

Table 2.9: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Iris data set.

algorithm	**average**	**s.d.**
simple GP	5.6	6.1
Ltree	**2.7**	3.0
OC1	7.3	6.0
C4.5	4.7	5.0
CEFR-MINER	4.7	7.1
ESIA	4.7	0.0
default	33.3	

On this data set the results of all the algorithms are quite close together and the differences between our *simple* GP algorithm and the other algorithms are not statistically significant.

2.9 Fitness Cache

Evolutionary algorithms generally spend a lot of their computation time on calculating the fitness of the individuals. However, if you look at the individuals during an evolutionary run, created either randomly in the beginning or as the result of recombination and mutation operators, you will often find that some of the genotypes occur more than once. We can use these genotypical reoccurences to speedup the fitness calculations by storing each evaluated genotype and its fitness in a *fitness cache*. We can use this cache by comparing each newly created individual to the genotypes in the *fitness cache*. If an individual's genotype is already in the cache its fitness can simply be retrieved from the cache instead of the time consuming calculation which would otherwise be needed.

In order to measure the percentage of genotypical reoccurences we will use the *resampling ratio* introduced by van Hemert et al. [53, 52].

Definition 2.9.1 *The resampling ratio is defined as the total number of hits in a run divided by the total number of generated points in the same run: resamplingratio = hits/evaluations. A hit is a point in the search space that we have seen before, i.e., it is already present in the searched space.*

In our case the resampling ratio corresponds to the number of cache hits in the fitness cache. The average resampling ratios and corresponding standard deviations for our *simple* GP algorithm on the six data sets from the previous section are shown in Table 2.10. Looking at the results it seems clear that there is no direct relationship between the size of a search space and the resampling ratio.

The lowest resampling ratio of 12.4% on the Ionosphere data set may seem quite high for such a simple fitness cache but early experiments using lower mutation and crossover rates resulted in even higher resampling ratio's for the different data sets. Although the resampling ratio does not give an indication as to the evolutionary search process will be succesfull we did not want it to become too high given the relatively small number of fitness evaluations (19.900).

Note that the resampling ratio's cannot be directly translated into decreased computation times. Not only do initialization, recombination and statistics take time, the total computation time of our GP algorithms is also heavily influenced by several other external factors such as computer platform (e.g., processor type and speed) and implementation. As a result the reductions in computation time achieved by the use of a fitness cache are less than the resampling ratios of Table 2.10.

Table 2.10: The search space sizes and average resampling ratios with standard deviations for our *simple* GP algorithm on the different data sets.

	resampling ratio		**search space**
dataset	avg.	s.d.	**size**
Australian Credit	16.9	4.3	7.5×10^{120}
German Credit	15.4	4.2	1.3×10^{101}
Pima Indian Diabetes	15.2	3.8	7.0×10^{121}
Heart Disease	13.7	3.1	8.1×10^{105}
Ionosphere	12.4	2.8	1.1×10^{147}
Iris	18.4	4.7	1.7×10^{96}

2.10 Conclusions

We introduced a *simple* GP algorithm for data classification. If we compare the results of our *simple* GP to the other evolutionary approaches ESIA and CEFR-MINER we see that on most data sets the results do not differ significantly. On the German Credit and Pima Indian Diabetes data sets our *simple* GP algorithm performs significantly better than ESIA. On the Ionosphere data set *simple* GP performs significantly worse than CEFR-MINER. If we look at the classification results of our *simple* GP algorithm and the non-evolutionary algorithms we also see that our *simple* GP does not perform significantly better or worse on most of the data sets. Only on the Australian Credit data set does our *simple* GP algorithm perform significantly worse than all three decision tree algorithms (Ltree, OC1 and C4.5). On the Heart Disease data set the classification performance of our *simple* GP algorithm is only significantly worse than Ltree.

The fact that on most data sets the results of our *simple* GP algorithm are neither statistically significantly better or worse than the other algorithms is partly due to the used two-tailed independent samples t-test we performed. In [43] paired t-tests are performed to compare Ltree, C4.5 and OC1 which show that some differences in performance between the algorithms are significant. An independent samples t-test does not always show the same difference to be statistically significant, and based on the data published in [43], [75] and [72] we cannot perform a paired t-test.

Compared to the ESIA, CEFR-MINER, Ltree, OC1 and C4.5 algorithms the classification performance of our *simple* GP algorithm is a little disappointing. One of the main goals in designing a (supervised) learning algorithm for data classification is that the trained model should perform well on unseen data (in our case the test set). In the design of our *simple* GP algorithm and the setup of our experiments we have made several choices which may influence the generalization power of the evolved models.

As we discussed in Section 2.7 one of the problems of supervised learning algorithms is overfitting. By keeping the population size, the number of offspring and the number of generations small we try to prevent this phenomenon. As far as we can see these measures are successful since we have not been able to detect overfitting during the training of our *simple* GP algorithm. A down-side of the limited number of generations is that our *simple* GP algorithm may not have enough time to find a good decision tree. This

might explain the disappointing classification performance compared to the other algorithms on some data sets.

In Section 2.6 we introduce a 2-layered fitness function both as a precaution against bloat and because we believe smaller decision trees are easier to understand than larger trees. For the same reasons we also employ a size limit using the tree pruning system built into the *Evolving Objects* library (EOlib) [64]. This size limit ensures that every tree which becomes larger than a fixed number of tree nodes as a result of mutation or crossover, is automatically pruned. However, according to Domingos [18, 19] larger more complex models should be preferred over smaller ones as they offer better classification accuracy on unseen data. Early experiments with and without the 2-layered fitness function did not indicate any negative effects from using the tree size as a secondary fitness measure. Other early experiments using smaller maximum tree sizes did result in lower classification performance.

Besides overfitting and decision tree complexity, another influence on the generalization power of our evolved decision trees might be the size of the search spaces for the different data sets. As can be seen in Table 2.10 the search space sizes of our *simple* GP algorithm for the different data sets are large. Given the restrictions we place on the population size, number of offspring and the maximum number of generations our *simple* GP algorithm is only capable of evaluating a relatively very small number of possible decision trees. In the next chapter we investigate if we can improve classification performance by reducing the size of the search space.

3 Refining the Search Space

An important aspect of algorithms for data classification is how well they can classify unseen data.

We investigate the influence of the search space size on the classification performance of our GP algorithms. We introduce three new GP decision tree representations. Two representations reduce the search space size for a data set by partitioning the domain of numerical valued attributes using information theory heuristics from ID3 and C4.5. The third representation uses K-means clustering to divide the domain of numerical valued attributes into a fixed number of clusters.

3.1 Introduction

At the end of Chapter 2 we discussed the influence of various aspects of our *simple* GP algorithm on its predictive accuracy towards unseen data. In [18, 19] Domingos argues, based on the mathematical proofs of Blumer et al. [9], that: "if a model with low training-set error is found within a sufficiently small set of models, it is likely to also have low generalization error". In the case of our *simple* GP algorithm the set of models, the search space size, is determined by the maximum number of nodes (63), the number of possible internal nodes and the number of terminals (see Lemma 2.5.2).

The easiest way to reduce the size of the search space in which our GP algorithms operate, would be to limit the maximum number of tree nodes. However, the maximum number of 63 tree nodes we selected for our experiments is already quite small and early experiments with smaller maximum

tree sizes resulted in lower classification performance. We will therefore reduce the size of the search spaces for the different data sets by limiting the number of possible internal nodes for numerical valued attributes. There are two reasons for only focusing on the numerical valued attributes. First, it is difficult to reduce the number of possible internal nodes for non-numerical attributes without detailed knowledge of the problem domain. Second, most of the possible internal nodes created by our *simple* GP algorithm were for the numerical valued attributes.

In order to limit the number of possible internal nodes for numerical valued attributes we will group values together. By grouping values together we in effect reduce the number of possible values and thus the number of possible internal nodes. To group the values of an attribute together we will borrow some ideas from other research areas.

The first technique we will look at is derived from decision tree algorithms, particularly C4.5 and its predecessor ID3. Decision tree algorithms like these two use information theory to decide how to construct a decision tree for a given data set. We will show how the information theory based criteria from ID3 and C4.5 can be used to divide the domain of numerical valued attributes into partitions. Using these partitions we can group values together and reduce the number of possible internal nodes and thus the size of the search space for a particular data set.

The second technique we look at is supervised clustering. Clustering is a technique from machine learning that is aimed at dividing a set of items into a (fixed) number of "natural" groups. In our case we will use a form of K-means clustering rather than an evolutionary algorithm as it is deterministic and faster.

The outline of the rest of this chapter is as follows. In Section 3.2 we describe how machine learning algorithms for constructing decision trees work in general, and we will examine C4.5 in particular. We will then introduce two new representations which reduce the size of the search spaces for the data sets by partitioning the domain of numerical valued attributes using the information theory criteria from ID3 and C4.5. Then in Section 3.4 we will introduce another representation which uses K-means clustering to divide the domain of numerical valued attributes into a fixed number of clusters. In Section 3.5 the results of the new representations will be compared with the results of the algorithms from the previous chapter. In the final section we will draw our conclusions.

3.2 Decision Tree Construction

Decision tree constructing algorithms for data classification such as ID3 [86], C4.5 [87] and CART [14] are all loosely based on a common principle: *divide-and-conquer* [87]. The algorithms attempt to divide a training set T into multiple (disjoint) subsets so that each subset T_i belongs to a single target class. In the simplest form a training set consisting of N records is divided into N subsets $\{T_1, \ldots, T_N\}$ such that each subset is associated with a single record and target class. However, the predictive capabilities of such a classifier would be limited. Therefore decision tree construction algorithms like C4.5 try to build more general decision trees by limiting the number of partitions (and thereby limiting the size of the constructed decision tree). Since the problem of finding the smallest decision tree consistent with a specific training set is NP-complete [58], machine learning algorithms for constructing decision trees tend to be non-backtracking and greedy in nature. Although the non-backtracking and greedy nature of the algorithms has its advantages, such as resulting in relatively fast algorithms, they do depend heavily on the way the training set is divided into subsets. Algorithms like ID3 and C4.5 proceed in a recursive manner. First an attribute is selected for the root node and each of the branches to the child nodes corresponds with a possible value for this attribute. In this way the data set is split up into subsets according to the value of the attribute. This process is repeated recursively for each of the branches using only the records that occur in a certain branch. If all the records in a subset have the same target class the branch ends in a leaf node with the class prediction. If there are no attributes left to split a subset the branch ends in a leaf node predicting the class that occurs most frequent in the subset.

3.2.1 Gain

In order to split a data set into two or more subsets ID3 uses an heuristic based on information theory [16, 94] called *gain*. In information theory the *information* criterion (or entropy) measures the amount of information (in bits) that is needed to identify a class in a single data set. The *information* measure *info(T)* for a single non-empty data set T is calculated as follows:

$$info(T) = -\sum_{i=1}^{\#classes} \frac{freq(C_i, T)}{|T|} \times log_2 \left(\frac{freq(C_i, T)}{|T|} \right), \tag{3.1}$$

where $freq(C_i, T)$ is the number of cases in data set T belonging to class C_i. If $freq(C_i, T)$ happens to be 0 the contribution of this term is defined to be 0. The information is given in bits.

In order to determine the average amount of information needed to classify an instance after a data set T has been split into several subsets T_i^X using a test X we can compute the *average information* criterion. This *average information* criterion is calculated by multiplying the *information* values of the subsets by their sizes relative to the size of the original data set. Thus

$$information[X|T] = \sum_{i=1}^{\#subsets} \frac{|T_i^X|}{|T|} \times info(T_i^X), \tag{3.2}$$

where T_i^X is the i-th subset after splitting data set T using a test X.

To decide which test should be used to split a data set ID3 employs the *gain* criterion. The *gain* criterion measures the amount of information that is gained by splitting a data set on a certain test. The information gained by splitting a data set T using a test X is calculated as

$$gain[X\,|\,T] = info(T) - information[X|T]. \tag{3.3}$$

In ID3 the test which offers the highest gain of information is chosen to split a data set into two or more subsets. Although the use of the *gain* criterion gives quite good results it has a major drawback. The *gain* criterion has a strong bias towards tests which result in a lot of different subsets.

Example 3.2.1 Consider the data set T in Table 3.1.

When ID3 is used to construct a decision tree for this data set it starts by calculating the amount of information needed to classify a record in data set T. Thus

$$info(T) = -\frac{2}{4} log_2 \frac{2}{4} - \frac{2}{4} log_2 \frac{2}{4} = 1 \ bit.$$

Now the amount of information that can be gained by splitting data set T on either attribute **A** or **B** has to be calculated. Since attribute **A** is numerical valued we look at tests of the form (**A** $<$ *threshold_value*) as is

Table 3.1: A small example data set.

A	**B**	**class**
1	a	yes
2	b	yes
3	c	no
4	d	no

done by C4.5, although not in the original ID3 algorithm. Attribute **B** is nominal valued so we use the attribute itself as a test. Note that we do not look at 1 for a possible threshold value for attribute **A** as it does not split the data set T.

1. Splitting on 2 as a threshold value gives:

$$\begin{aligned} information[\mathbf{A} < 2|T] &= \frac{1}{4} \times (-\frac{1}{1} log_2 \frac{1}{1}) + \frac{3}{4} \times (-\frac{1}{3} log_2 \frac{1}{3} - \frac{2}{3} log_2 \frac{2}{3}) \\ &\approx 0.69 \ bits. \end{aligned}$$

   The gain now becomes $gain[\mathbf{A} < 2|T] \approx 1 - 0.69 = 0.31 \ bits$.

2. Splitting on 3 as a threshold value gives:

$$\begin{aligned} information[\mathbf{A} < 3|T] &= \frac{2}{4} \times (-\frac{2}{2} log_2 \frac{2}{2}) + \frac{2}{4} \times (-\frac{2}{2} log_2 \frac{2}{2}) \\ &= 0 \ bits. \end{aligned}$$

   The *gain* now becomes $gain[\mathbf{A} < 3|T] = 1 - 0 = 1 \ bit$.

3. Splitting on 4 as a threshold value gives:

$$\begin{aligned} information[\mathbf{A} < 4|T] &= \frac{3}{4} \times (-\frac{2}{3} log_2 \frac{2}{3} - \frac{1}{3} log_2 \frac{1}{3}) + \frac{1}{4} \times (-\frac{1}{1} log_2 \frac{1}{1}) \\ &\approx 0.69 \ bits. \end{aligned}$$

   This *gain* now becomes $gain[\mathbf{A} < 4|T] \approx 1 - 0.69 \approx 0.31 \ bits$.

4. Splitting on attribute **B** gives:

$$\begin{aligned} \mathit{information}[\mathbf{B}|T] &= 4 \times (\frac{1}{4} \times (-\frac{1}{1} log_2 \frac{1}{1})) \\ &= 0 \ \mathit{bits}, \end{aligned}$$

where, by abuse of notation, "$\mathit{information}[\mathbf{B}|T]$" denotes the average information needed to classify an instance in the original data set T after splitting the data set on attribute **B**. Using a similar notation the *gain* becomes $\mathit{gain}[\mathbf{B}|T] = 1 - 0 = 1 \ \mathit{bit}$.

In this case either $(\mathbf{A} < 3)$ or attribute **B** would be chosen as a possible test for the root node by ID3. Since both tests can classify every instance in the data set perfectly an ID3 style algorithm would return one of the decision trees in Figure 3.1. This example also reveals a potential problem of the *gain* criterion as it shows no preference for the smaller tree with $(\mathbf{A} < 3)$ as the root node, although this tree offers more information about the data set.

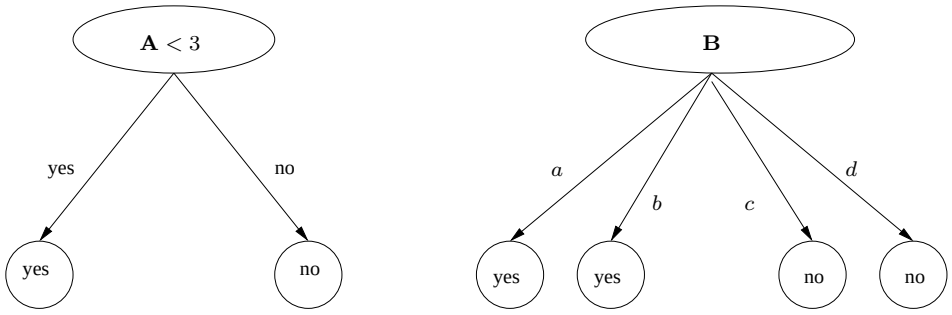

Figure 3.1: The two possible decision tree for Example 3.2.1 based on the *gain* criterion.

□

3.2.2 Gain_ratio

With the introduction of C4.5 came, among other improvements over ID3, a new criterion called *gain_ratio*. The *gain_ratio* criterion addresses the deficiency of the *gain* criterion expressed above, by dividing the information gain

of a test by the *split_info* of that test. The *split_info* measure is similar to the *info* measure in Equation 3.1, but instead of looking at the class distribution of the subsets it only looks at the sizes of the subsets. In this way *split_info* measures the potential of information generated by dividing a data set into several subsets. Thus

$$split\_info[X|T] = -\sum_{i=1}^{\#subsets} \frac{|T_i^X|}{|T|} \times log_2 \frac{|T_i^X|}{|T|}, \tag{3.4}$$

where as above T_i^X is the i-th subset after splitting data set T using a test X. The *gain_ratio* criterion now becomes

$$gain\_ratio[X|T] = \frac{gain[X|T]}{split\_info[X|T]}. \tag{3.5}$$

Unlike the *gain* criterion which measures the amount of information gained from splitting a data set into subsets, the *gain_ratio* criterion measures the *proportion* of information gained that is *useful* for classification.

Example 3.2.2 Consider again the data set T in Table 3.1. In Example 3.2.1 we calculated the *information* and *gain* measures for the possible root nodes. The C4.5 algorithm also computes these criteria but additionally calculates the *split_info* and *gain_ratio* criteria.

Continuing Example 3.2.1 we assume that the *gain* criteria for the possible tests are known. After calculating the *gain* for a possible test C4.5 computes the *split_info* measure for that test. The *gain* and *split_info* measures are then used to calculate the *gain_ratio* measure for that test:

1. $gain[\mathbf{A} < 2|T]$ is approximately 0.31 bits.
   The *split_info* for this split would be
   $split\_info[\mathbf{A} < 2|T] = -\frac{1}{4} \times log_2 \frac{1}{4} - \frac{3}{4} \times log_2 \frac{3}{4} \approx 0.81\ bits.$
   The *gain_ratio* now becomes
   $gain\_ratio[\mathbf{A} < 2|T] \approx \frac{0.31}{0.81} \approx 0.38.$

2. $gain[\mathbf{A} < 3|T] = 1\ bit.$
   We can calculate the *split_info* as
   $split\_info[\mathbf{A} < 3|T] = -\frac{2}{4} \times log_2 \frac{2}{4} - \frac{2}{4} \times log_2 \frac{2}{4} = 1\ bit.$
   This results in a *gain_ratio* of
   $gain\_ratio[\mathbf{A} < 3|T] = \frac{1}{1} = 1.$

3. $gain[\mathbf{A} < 4|T] \approx 0.31$ *bits*.
   The *split_info* for this split would be
   $split\_info[\mathbf{A} < 4|T] = -\frac{1}{4} \times log_2\frac{1}{4} - \frac{3}{4} \times log_2\frac{3}{4} \approx 0.81$ *bits*.
   The *gain_ratio* now becomes
   $gain\_ratio[\mathbf{A} < 4|T] \approx \frac{0.31}{0.81} \approx 0.38$.

4. $gain[\mathbf{B}|T] = 1$ *bit*.
   In this case the *split_info* becomes
   $split\_info = -4 \times \frac{1}{4} \times (-\frac{1}{4}log_2\frac{1}{4}) = 2$ *bits*.
   This results in a *gain_ratio* of
   $gain\_ratio[\mathbf{B}|T] = 0.5$,
   where, by abuse of notation, "$gain\_ratio[\mathbf{B}|T]$" denotes the *gain_ratio* after splitting the data set on attribute **B**.

Now, in the case of C4.5 it is clear that ($\mathbf{A} < 3$) and not attribute **B** should be chosen as the root node by C4.5 as it has the highest *gain_ratio*. Since this test can classify every instance in the data set perfectly, C4.5 would return the decision tree in Figure 3.2.

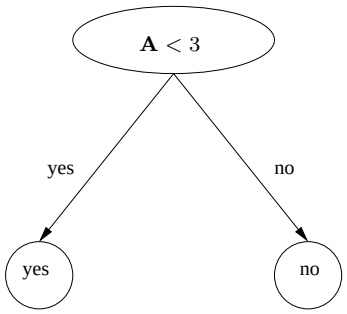

Figure 3.2: The optimal decision tree for Example 3.2.2 according to C4.5.

□

3.3 Representations Using Partitioning

In Section 2.4 we introduced the *simple* representation by specifying which atoms can occur. One of the drawbacks of the *simple* representation is that it creates a possible internal node for each combination of a numerical valued attribute and value that occurs in the data set. In order to reduce the huge

number of possible internal nodes that are generated in this way we can use the *gain* and *gain_ratio* criteria. In C4.5 a single threshold value is selected to split the domain of a numerical valued attribute into two partitions. For our new representations we do something similar. For each numerical valued attribute A_i with domain D_i let $V_i = \{v_1^i, \ldots, v_{k-2}^i\}$, with $n \leq k-1$ denote a set of threshold values, where k is the maximum number of partitions. In order to find the optimal set of threshold values we have to look at all possible combinations of at most $k-1$ threshold values and compare their *gain* or *gain_ratio* values so that $gain(\_ratio)[A_i < v_1^i, A_i \in [v_1^i, v_2^i), \ldots, A_i \geq v_n^i|T]$ is greater than or equal to any other combination of threshold values. The *gain_ratio* criterion should be especially useful as it is designed to find a balance between information gained by splitting a data set into a large number of data subsets and limiting the number of subsets.

Since the total number of sets of threshold values can become too large to effectively compare them all and our main aim is to reduce the size of the search spaces we will limit the maximum number of partitions to 5. If two sets of threshold values have the same *gain* or *gain_ratio* measures we will choose the set containing the least number of threshold values. In order to use the partitions specified by the optimal set of threshold values we need new types of atoms. If the optimal set of threshold values consists for instance of the three threshold values $threshold_1$, $threshold_2$ and $threshold_3$ we can construct atoms of the form

- $attribute < threshold_1$,
- $attribute \in [threshold_1, threshold_2)$,
- $attribute \in [threshold_2, threshold_3)$, and
- $attribute \geq threshold_3$.

Note that in case the optimal set of threshold values consists of only one threshold value just a single atom can be used ($attribute < threshold_1$) (see [25]).

Example 3.3.1 Consider the example data set in Table 3.2.

Table 3.2: Example data set

A	**B**	**class**
1	a	yes
2	b	yes
3	a	no
4	b	no
5	a	yes
6	b	yes

In the case of the *simple* representation we get the following atoms:

- Since attribute **A** has six possible values {1,2,3,4,5,6} and is numerical valued we use the $<$ operator: $(\mathbf{A} < 1)$, $(\mathbf{A} < 2)$, $(\mathbf{A} < 3)$, $(\mathbf{A} < 4)$, $(\mathbf{A} < 5)$ and $(\mathbf{A} < 6)$.
- Attribute **B** is non-numerical and thus we use the $=$ operator: $(\mathbf{B} = a)$ and $(\mathbf{B} = b)$.
- Finally for the target class we have two terminal nodes: $(class := yes)$ and $(class := no)$.

Now, if we set our maximum number of parititions k to 2 or more, we get the threshold values 3 and 5, using either the *gain* or *gain_ratio* criterion. In this case only two threshold values would be chosen as they result in a "perfect" partitioning of the domain of attribute **A**. This results in the following atoms for attribute **A**:

- $\mathbf{A} < 3$,
- $\mathbf{A} \in [3, 5)$,
- $\mathbf{A} \geq 5$.

□

3.4 A Representation Using Clustering

Another method to partition the domain of numerical valued attributes is supervised clustering. Clustering algorithms are usually employed to group collections of data together based on some measure of similarity. Each of these groups is called a cluster. Unlike the partitioning methods described in Section 3.3, clustering algorithms do not use the target class but rather divide the instances into "natural" groups. For our purposes we limit the clustering process to partitioning the domain of a single numerical valued attribute rather than clustering entire instances in the data set. In our case the clustering takes place in one-dimensional real space. Although we could use some kind of evolutionary algorithm for clustering, we decided to use K-means clustering algorithm since it is fast(er), easy to implement and deterministic (and the results are satisfactory).

The basic K-means clustering algorithm [100] works as follows. One starts by specifying the number of clusters k one wants to find and selects the initial cluster centers. Then all instances in the data set are assigned to the cluster center to which they are closest in Euclidean distance. Once all instances have been assigned to a cluster, the new cluster center or centroid is calculated as the mean of all instances assigned to that cluster. Next this process is repeated with the newly calculated cluster centers until the same instances are assigned to the same clusters in consecutive rounds.

In our approach all numerical valued attributes with a domain size larger than k are clustered, where k is one of the parameters of the algorithm. If the domain of a numerical valued attribute consists of k unique values or less we simply use atoms of the form $(attribute = value)$ for each attribute-value pair instead of clustering. The performance of the K-means clustering algorithm is dependent on the choice of the initial cluster centroids as well as instance order. We have therefore chosen to use the *Partition Around Medioids* initialization as proposed by Kaufman [61]. This initialization was (empirically) found to be the best of four classical initialization methods when looking at effectiveness, robustness and convergence speed [84].

After the clustering algorithm has determined the clusters, we can construct atoms based on the minimum and maximum value assigned to each cluster. Thus, if the K-means clustering algorithm has found three clusters we can construct the following atoms:

- $attribute \in [min_1, max_1]$,
- $attribute \in [min_2, max_2]$, and
- $attribute \in [min_3, max_3]$.

where min_i and max_i are the minimum and maximum value of cluster i respectively.

We will call this the *clustering* representation.

Example 3.4.1 Observe the data set in Example 3.3.1 (see Table 3.2).

Using our K-means clustering algorithm with $k = 3$ results in three clusters for attribute **A**: $[1, 2]$, $[3, 4]$ and $[5, 6]$. Thus, in this case the following atoms are constructed:

- $\mathbf{A} \in [1, 2]$,
- $\mathbf{A} \in [3, 4]$, and
- $\mathbf{A} \in [5, 6]$.

□

3.5 Experiments and Results

To compare the classification performance of our new representations with the *simple* representation of the previous chapter we have conducted the same experiments using the same settings and data sets (see Section 2.7). We will also compare our results to Ltree, OC1 and C4.5 and the other evolutionary algorithms (ESIA and CEFR-MINER) already mentioned in the previous chapter. The tables with results also contain an extra column, labeled k, to indicate the number of clusters in the case of our *clustering* GP algorithms or the maximum number of partitions in the case of the *gain* GP and *gain_ratio* GP algorithms. The best (average) result for each data set is printed in **bold** font. The entry N/A indicates that no results were available.

To determine if the results obtained by our algorithms are statistically significantly different from the results reported for ESIA, CEFR-MINER, Ltree, OC1 and C4.5, we have performed two-tailed independent samples t-tests

with a 95% confidence level ($p = 0.05$) using the reported mean and standard deviations. The null-hypothesis in each test is that the means of the two algorithms involved are equal. In order to determine whether the differences between our GP algorithms are statistically significant we used paired two-tailed t-tests with a 95% confidence level ($p = 0.05$) using the results of 100 runs (10 random seeds times 10 folds). In these tests the null-hypothesis is also that the means of the two algorithms involved are equal.

3.5.1 Search Space Sizes

Table 3.3: The number of possible internal nodes and the resulting search space sizes for the Australian Credit data set.

		size of function set			
algorithm	k	**avg.**	**min**	**max**	**search space size**
clustering GP	2	28.0	28	28	4.6×10^{70}
clustering GP	3	38.0	38	38	5.9×10^{74}
clustering GP	4	46.0	46	46	2.2×10^{77}
clustering GP	5	54.0	54	54	3.2×10^{79}
gain GP	2	28.0	28	28	4.6×10^{70}
gain GP	3	38.0	38	38	5.9×10^{74}
gain GP	4	46.0	46	46	2.2×10^{77}
gain GP	5	54.0	54	54	3.2×10^{79}
gain_ratio GP	2	28.0	28	28	4.6×10^{70}
gain_ratio GP	3	29.7	29	30	$1.4 \times 10^{71} \ldots 3.9 \times 10^{71}$
gain_ratio GP	4	31.2	30	33	$3.9 \times 10^{71} \ldots 7.5 \times 10^{72}$
gain_ratio GP	5	32.7	31	36	$1.1 \times 10^{72} \ldots 1.1 \times 10^{74}$
simple GP		1167.0	1167	1167	7.5×10^{120}

In Table 3.3 the average, minimum and maximum number of internal nodes used by our new GP algorithms for the Australian Credit data set are shown with the resulting search space sizes. In the case of our *clustering* GP algorithms the number of possible internal nodes is constant for all 10 folds and random seeds depending on the value of k. We see a clear distinction between the *gain* GP and *gain_ratio* GP algorithms. Because the *gain* criterion does not take the number of subsets into consideration it always results in

the maximum number of partitions allowed, similar to the *clustering* GP algorithm. Since the *gain_ratio* criteria also takes the number of partitions into consideration the number of partitions for each variable differs per fold. As a result using the *gain_ratio* criterion leads to a much smaller increase in maximum number of possible internal nodes if we increase the allowed maximum number of partitions k.

Table 3.4: The number of possible internal nodes and the resulting search space sizes for the Iris data set.

		# internal nodes			
algorithm	k	**avg.**	**min**	**max**	**search space size**
clustering GP	2	8.0	8	8	2.7×10^{59}
clustering GP	3	12.0	12	12	7.7×10^{64}
clustering GP	4	16.0	16	16	5.8×10^{68}
clustering GP	5	20.0	20	20	5.8×10^{71}
gain GP	2	8.0	8	8	2.7×10^{59}
gain GP	3	12.0	12	12	7.7×10^{64}
gain GP	4	16.0	16	16	5.8×10^{68}
gain GP	5	20.0	20	20	5.8×10^{71}
gain_ratio GP	2	8.0	8	8	2.7×10^{59}
gain_ratio GP	3	8.0	8	8	2.7×10^{59}
gain_ratio GP	4	8.0	8	8	2.7×10^{59}
gain_ratio GP	5	8.0	8	8	2.7×10^{59}
simple GP		123.0	123	123	1.7×10^{96}

For the other data sets the results are similar. Both our *clustering* GP and *gain* GP algorithms have the same search space sizes for the same maximum number of partitions or clusters k. In the case of the *gain_ratio* GP algorithm the search space size increases only slightly as we increase the maximum allowed number of partitions. The only exception is the Iris data set where the search space size did not increase (see Table 3.4).

3.5.2 The Australian Credit Data Set

The results on the Australian Credit data set are displayed in Table 3.5. On this data set our new GP algorithms perform significantly better than

our *simple* GP algorithm, regardless of the number of clusters or partitions used. More interestingly, the best performing GP algorithm, *clustering* GP with 2 clusters per numerical valued attribute, also has a significantly lower misclassification rate than our other GP algorithms with the exception of our *gain* GP algorithm with $k = 2$.

If we compare the results of our new GP algorithms with ESIA we see that all our GP algorithms perform significantly better, based on a two-tailed independent samples t-test. The differences between our new GP algorithms and the decision tree algorithms Ltree, OC1 and C4.5 are not statistically significant.

Table 3.5: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Australian Credit data set.

algorithm	**k**	**average**	**s.d.**
clustering GP	2	**13.7**	3.0
clustering GP	3	14.8	2.6
clustering GP	4	14.8	2.9
clustering GP	5	15.2	2.6
gain GP	2	14.2	2.7
gain GP	3	15.1	2.7
gain GP	4	14.9	3.6
gain GP	5	15.1	3.8
gain_ratio GP	2	15.7	3.2
gain_ratio GP	3	15.5	3.4
gain_ratio GP	4	15.5	3.2
gain_ratio GP	5	15.6	3.7
simple GP		22.0	3.9
Ltree		13.9	4.0
OC1		14.8	6.0
C4.5		15.3	6.0
CEFR-MINER		N/A	
ESIA		19.4	0.1
default		44.5	

3.5.3 The German Credit Data Set

When we consider the results on the German Credit data set in Table 3.6 we see that both our *simple* GP and *gain* GP algorithm with $k = 3$ perform significantly better than our other GP algorithms, with the exception of our *clustering* GP algorithm with $k = 2$. The differences with the decision tree algorithms Ltree, OC1 and C4.5 are not statistically significant.

If we compare the results of our GP algorithms with ESIA we see that the differences between our *simple* GP and *clustering* GP algorithms and ESIA are statistically significant. In the case of our *gain* GP algoritms the differences with ESIA are also statistically significant except for $k = 4$. Differences between ESIA and our *gain_ratio* GP algorithms are not statistically significant.

Table 3.6: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the German Credit data set.

algorithm	**k**	**average**	**s.d.**
clustering GP	2	27.8	1.3
clustering GP	3	28.0	1.4
clustering GP	4	27.9	1.5
clustering GP	5	28.4	1.5
gain GP	2	28.1	1.9
gain GP	3	27.1	1.9
gain GP	4	28.3	1.9
gain GP	5	28.2	1.6
gain_ratio GP	2	28.3	1.9
gain_ratio GP	3	28.5	2.3
gain_ratio GP	4	28.6	2.3
gain_ratio GP	5	28.5	2.2
simple GP		27.1	2.0
Ltree		26.4	5.0
OC1		**25.7**	5.0
C4.5		29.1	4.0
CEFR-MINER		N/A	
ESIA		29.5	0.2
default		30.0	

3.5.4 The Pima Indians Diabetes Data Set

The results for the Pima Indians Diabetes data set are displayed in Table 3.7. Although not particularly good, the results of the *gain_ratio* GP algorithms are the most surprising. For $k = 2$ and $k = 3$ these algorithms have exactly the same misclassification performance per fold, although the sets of possible internal nodes differ as a result of the maximum allowed number of partitions per numerical valued attribute. The same behavior is seen for $k = 4$ and $k = 5$. In all cases the discovered decision trees differ syntactically per fold and random seed. The most likely reason is that the evolved trees are similar and contain mostly atoms from attributes of which the domain was partitioned into 2 (or 4) parts.

Table 3.7: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Pima Indians Diabetes data set.

algorithm	**k**	**average**	**s.d.**
clustering GP	2	26.3	3.2
clustering GP	3	26.3	3.4
clustering GP	4	26.7	3.2
clustering GP	5	26.5	4.4
gain GP	2	27.0	4.3
gain GP	3	26.5	3.4
gain GP	4	25.9	3.4
gain GP	5	25.9	4.2
gain_ratio GP	2	27.6	4.6
gain_ratio GP	3	27.6	4.6
gain_ratio GP	4	27.7	4.7
gain_ratio GP	5	27.7	4.7
simple GP		26.3	3.6
Ltree		**24.4**	5.0
OC1		28.0	8.0
C4.5		24.7	7.0
CEFR-MINER		N/A	
ESIA		29.8	0.2
default		34.9	

On the Pima Indians Diabetes data set the results of the *gain_ratio* GP algorithms are significantly worse than the results of our other GP algorithms, except for our *gain* GP algorithm with $k = 2$. The differences between our other GP algorithms are not statistically significant, except for the difference between our *clustering* GP algorithm with $k = 4$ and our *gain_ratio* GP algorithms with $k = 4$ or $k = 5$.

Our *simple* GP and *clustering* GP algorithms perform significantly better on this data set than ESIA. With the exception of $k = 2$ our *gain* GP algorithms are also significantly better. The differences in misclassification performance between our new GP algorithms and Ltree, OC1 and C4.5 are not statistically significant.

Table 3.8: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Heart Disease data set.

algorithm	**k**	**average**	**s.d.**
clustering GP	2	19.9	4.6
clustering GP	3	21.3	5.4
clustering GP	4	22.5	4.3
clustering GP	5	22.1	4.0
gain GP	2	19.9	4.8
gain GP	3	22.8	4.3
gain GP	4	22.1	4.5
gain GP	5	21.5	4.7
gain_ratio GP	2	18.7	4.5
gain_ratio GP	3	20.3	5.3
gain_ratio GP	4	20.6	5.8
gain_ratio GP	5	19.8	4.1
simple GP		25.2	4.8
Ltree		**15.5**	4.0
OC1		29.3	7.0
C4.5		21.1	8.0
CEFR-MINER		17.8	7.1
ESIA		25.6	0.3
default		44.0	

3.5.5 The Heart Disease Data Set

The results on the Heart Disease data set are displayed in Table 3.8. All our new GP algorithms show a significant improvement in misclassification performance over our *simple* GP algorithm. Our best performing GP algorithm, *gain_ratio* GP with $k = 2$, is significantly better than all our other GP algorithms, with the exception of the *gain* GP algorithm with $k = 2$. Based on the results it seems clear that two clusters or partitions per numerical valued attribute offer the best performance for our GP algorithms.

With the exception of the *gain* GP algorithm using $k = 3$, all our new GP algorithms have a significantly lower misclassification rate than ESIA. The differences between our new GP algorithms and CEFR-MINER are not significant.

The differences in classification performance between Ltree and our GP algorithms are statistically significant except for the two best performing *gain_ratio* GP algorithms. OC1 performs very badly on this data set and as a result all our new GP algorithms are significantly better. The differences between C4.5 and our GP algorithms are not statistically significant.

3.5.6 The Ionosphere Data Set

When we consider the results of our GP algorithms on the Ionosphere data set in Table 3.9 we see that our *gain_ratio* GP algorithms perform the best. Together with the *gain* GP algorithm with $k = 2$ they are significantly better than our other GP algorithms. If we look at the results of our new GP algorithms with respect to k we see that the misclassification rate increases with k, except for our *clustering* GP with $k = 2$.

Our three best *gain_ratio* GP algorithms are also significantly better than OC1. Compared to the results of Ltree and C4.5 the differences in performance with our new GP algorithms are not statistically significant, except for the worst performing GP algorithm, *clustering* GP with $k = 5$. The differences between CEFR-MINER and our new algorithms are also no statistically significant.

Table 3.9: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Ionosphere data set.

algorithm	**k**	**average**	**s.d.**
clustering GP	2	13.1	4.1
clustering GP	3	10.5	5.1
clustering GP	4	12.1	3.0
clustering GP	5	13.3	3.8
gain GP	2	8.3	4.7
gain GP	3	10.5	4.8
gain GP	4	10.8	4.2
gain GP	5	11.6	4.4
gain_ratio GP	2	**7.6**	4.2
gain_ratio GP	3	8.1	4.3
gain_ratio GP	4	8.3	4.4
gain_ratio GP	5	9.1	3.6
simple GP		12.4	3.8
Ltree		9.4	4.0
OC1		11.9	3.0
C4.5		9.1	5.0
CEFR-MINER		11.4	6.0
ESIA		N/A	
default		35.9	

3.5.7 The Iris Data Set

When we consider the results of our new GP algorithms on the Iris data set in Table 3.10 we see that our *clustering* GP algorithm with $k = 3$ has a significantly lower misclassification rate than our other GP algorithms. If we look at Table 3.4 we see that the *gain_ratio* GP algorithms all split the domain of the numerical valued attributes into 2 partitions regardless of the maximum allowed number of partitions. This is probably the reason for the bad misclassification rate of these algorithms as both other new *top-down atomic* representations also classify badly when the domain of the numerical valued attributes is split into 2 partitions or clusters. As a result these algorithms perform significantly worse than the other algorithms.

Table 3.10: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Iris data set.

algorithm	**k**	**average**	**s.d.**
clustering GP	2	21.1	9.4
clustering GP	3	**2.1**	4.2
clustering GP	4	5.2	4.8
clustering GP	5	6.0	5.5
gain GP	2	29.6	6.3
gain GP	3	6.3	6.1
gain GP	4	5.1	6.4
gain GP	5	6.5	5.6
gain_ratio GP	2	31.7	5.0
gain_ratio GP	3	31.7	5.0
gain_ratio GP	4	31.7	5.0
gain_ratio GP	5	31.7	5.0
simple GP		5.6	6.1
Ltree		2.7	3.0
OC1		7.3	6.0
C4.5		4.7	5.0
CEFR-MINER		4.7	0.0
ESIA		4.7	7.1
default		33.3	

3.5.8 Scaling

In Table 3.11 the run-times of our algorithms on the Ionosphere data set are shown relative to the run-time of our *clustering* GP algorithm using 2 clusters per numerical valued attribute. Our *clustering* GP algorithm is the fastest and scales quite well when we increase the maximum number of partitions. When we look at the relative run-times of our *gain* GP and *gain_ratio* GP algorithms we see that they only scale well until $k = 3$. The problem with these GP algorithms is that they have to calculate the *gain* and *gain_ratio* criteria for each possible set of threshold values. Thus, as we increase the maximum number of partitions per numerical valued attribute the number of combinations of threshold values increases exponentially.

Table 3.11: The average relative run-times on the Ionosphere data set.

k	*clustering* GP	*gain* GP	*gain_ratio* GP
2	1	1.05	1.09
3	1.01	1.09	1.12
4	1.09	3.54	3.63
5	1.13	214.22	231.84

3.6 Conclusions

The results of our experiments reported above are a clear indication that deciding which atoms are going to be used when evolving decision trees can be crucial. However, it seems that even a sub-optimal refinement of the search space, whether through clustering or partitioning, can result in significantly better decision trees compared to the *simple* GP algorithms of Chapter 2. In the case of the Australian Credit and Heart Disease data sets all of our *clustering* and *partitioning* GP algorithms have a significantly lower misclassification rate than our *simple* GP algorithm, regardless of the number of clusters or partitions. On the Ionosphere data most of the new algorithms perform significantly better than *simple* GP, and none perform significantly worse.

On the other data sets we see that a smaller search space does not guarantee better classification performance. On the Iris data set the algorithms that use only two clusters or partitions for each numerical valued attribute fail to offer a reasonable classification performance. The method used to split the domain of numerical valued attributes into clusters or partitions also influences the classification performance. On the German Credit data set our *gain* GP algorithm with 2 partitions per numerical valued attribute performs similar to our *simple* GP algorithm while our other GP algorithms using $k = 2$ perform significantly worse. In the case of the Iris data set our *clustering* GP algorithm using $k = 3$ outperforms all our other GP algorithms regardless of the value of k. It seems that the method used to "refine" the search space is much more important than the actual reduction in search space size.

If we consider how well the clustering and partitioning parts of the algorithms scale with respect to the number of partitions it is clear that our *clustering* GP algorithm outperforms the *partitioning* GP algorithms (*gain*

GP and *gain_ratio* GP, see Table 3.11). However, if we look at the results in Section 3.5 we see that either two or three clusters or partitions for each numerical valued attribute generally offers the best results. This means that the scaling problems of our *refined* GP algorithms may not be an issue as long as the maximum number of partitions is restricted.

4 Evolving Fuzzy Decision Trees

Standard decision tree representations have difficulties when it comes to modelling real world concepts and dealing with polluted data sets. To solve some of these problems we extend our basic *full atomic* representation to include *fuzzy* decision trees. Fuzzy decision trees are based on fuzzy logic and fuzzy set theory unlike "standard" decision trees which are based on Boolean logic and set theory. Together with this new *fuzzy* representation we will introduce a new fitness measure based on fuzzy logic. The new *fuzzy* GP algorithms will be compared with the algorithms of the previous chapters to see if they improve classification performance.

4.1 Introduction

In the previous chapters we evolved decision trees based on our *top-down atomic* representation. These decision tree representations, as well as the representations used in ID3, C4.5 and CART, have a number of limitations.

First of all, they are used to model aspects of the real world from a "Boolean" point of view. In a "Boolean" view of the world everything is either true or false, black or white. This does not conform to the real world in which often many "shades of grey" exist. The Boolean *top-down atomic* representations used in the previous chapters cannot handle these intermediate values.

Secondly, the decision tree representations are based on the assumption that the data in the training set is precise, while in real world scenarios this is often not the case. In some cases the original data on which a data

set is based may have contained missing or empty values which have been replaced in the data preparation phase of the knowledge discovery process [85, Chapter 8]. There is also the possiblity of typing errors while inserting the data into the data base. Another potential problem is that the values in a data set may be the result of measurements which can be inaccurate. As a result there is often a degree of uncertainty as to whether the values in a data set are 100% accurate.

In this chapter we will show how we can evolve fuzzy decision trees based on the *top-down atomic* representations from Chapter 3 by using fuzzy sets and fuzzy logic [5]. Fuzzy set theory and fuzzy logic are generalizations of classical (Boolean) set theory and Boolean logic. They can be used to help computer programs deal with the uncertainty of the real world. By using fuzzy sets and fuzzy logic we aim to improve two aspects of the Boolean *top-down atomic* representations:

1. Classification accuracy: By using fuzzy sets in the atoms of our new *fuzzy* decision trees and evaluating those trees using fuzzy logic, they should be more robust towards faulty training data.

2. Understandability: The use of fuzzy terms in the atoms of our new *fuzzy* decision trees, instead of numerical values and "crisp" comparisons, should make the trees more intuitive.

Other work on discovering fuzzy decision trees using tree-based genetic programming was performed by Mendes, Voznika, Freitas and Nievola [75]. They used an innovative co-evolutionary approach employing strongly-typed Genetic Programming [77] and evolution strategies [92, 3] . They used strongly-typed Genetic Programming to evolve a population of fuzzy decision trees for binary classification while a $(1+5)$-evolution strategy was used to evolve the fuzzy membership functions for the continuous attributes. The results of their *Co-Evolutionary Fuzzy Rule Miner* (CEFR-MINER) system were very promising and we will compare the results of our own approach to theirs.

In Section 4.2 we discuss the basics of fuzzy set theory and fuzzy logic. Then in Section 4.3 we describe the new fuzzy representations. The results of the experiments are discussed in Section 4.4. In Section 4.5 we introduce a fuzzy fitness measure in order to try to improve the classification results and compare it with the standard "Boolean" fitness measure. Conclusions are reported in Section 4.6.

4.2 Fuzzy Set Theory

Fuzzy set theory [101, 5] is an extension of classical set theory. In classical set theory an element is either a member of a set, with all the characteristics of that set, or not. However, in the real world it is often not desirable or possible to define a set with all its characteristics. Consider, for example, the set of young people, described by

$$Young = \{x \in People \mid age(x) < 28\}. \tag{4.1}$$

According to this set a person under 28 is considered to be young. The membership function for the class of young people is then:

$$m_{Young}(x) = \begin{cases} 1 & \text{if } age(x) < 28; \\ 0 & \text{if } age(x) \geq 28. \end{cases} \tag{4.2}$$

In fuzzy set theory an element can be a partial member of a (fuzzy) set. The degree to which an element is a member of a fuzzy set is determined by a fuzzy membership function. More general than a membership function from classical set theory, a fuzzy membership function returns a real value between 0 and 1 including the endpoints, indicating the degree of membership. An example of a fuzzy membership function for a fuzzy set *Young* is given in Equation 4.3:

$$\mu_{Young}(x) = \begin{cases} 1 & \text{if } age(x) \leq 25; \\ 1 - \frac{age(x)-25}{30-25} & \text{if } 25 < age(x) \leq 30; \\ 0 & \text{if } age(x) > 30. \end{cases} \tag{4.3}$$

A person younger than 25 is always considered to be young (see Figure 4.1), but as that person becomes older he or she becomes less young and thus less a member of the set of young people. For instance a person of age 27 is young to a degree of $1 - \frac{27-25}{30-25} = 0.6$. At the same time that person also becomes older and thus his or her membership of set *old* increases. If that person becomes 28, and thus less young, the degree of membership of set Young decreases to $1 - \frac{28-25}{30-25} = 0.4$.

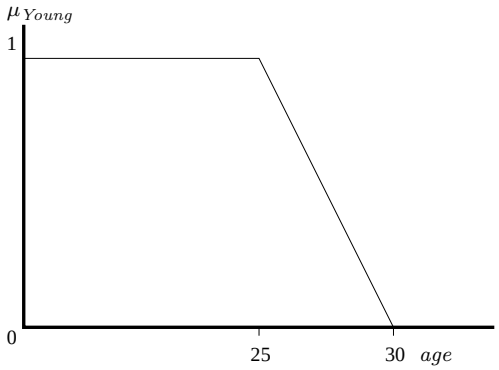

Figure 4.1: A graphical representation of the fuzzy membership function of Equation 4.3.

4.2.1 Fuzzy Logic

In classical set theory the basic operators conjunction, disjunction and complement are defined using Boolean logic. For example, the (classical) set of people who are both young and tall can be defined by the following membership function:

$$m_{Young \wedge Tall}(x) = \begin{cases} 1 & \text{if } m_{Young}(x) = 1 \text{ and } m_{Tall}(x) = 1, \\ 0 & \text{otherwise,} \end{cases} \tag{4.4}$$

where $m_{Young}(x)$ is the membership function defining the set of young people and $m_{Tall}(x)$ is the membership function of the set of *Tall* people.

Now assume that *Young* and *Tall* are fuzzy sets and an individual x belongs to set *Young* to a degree of 0.9 and to set *Tall* to a degree of 0.8. To what degree should individual x belong to the fuzzy set of young and tall people? These questions can be answered using fuzzy logic.

Fuzzy logic is an extension of Boolean logic just as fuzzy set theory is an extension of classical set theory. In 1965 Lofti Zadeh [101] defined the three basic operators conjunction, disjunction and complement as follows:

- conjunction: Let $\mu_A(x)$ and $\mu_B(x)$ be fuzzy membership functions of fuzzy sets A and B respectively. The conjunction of fuzzy sets A and B is given by:

$$\mu_{A \wedge B}(x) = min\{\mu_A(x), \mu_B(x)\}. \tag{4.5}$$

- disjunction: Let $\mu_A(x)$ and $\mu_B(x)$ be fuzzy membership functions of fuzzy sets A and B respectively. The disjunction of fuzzy sets A and B is given by:
$$\mu_{A \vee B}(x) = max\{\mu_A(x), \mu_B(x)\}. \tag{4.6}$$

- complement: Let $\mu_A(x)$ be the fuzzy membership function of a fuzzy set A. The fuzzy membership function of the complement of fuzzy set A is given by:
$$\mu_{\neg A}(x) = 1 - \mu_A(x). \tag{4.7}$$

According to Equation 4.5 an individual x which belongs to fuzzy set *Young* to a degree of 0.9 and to fuzzy set *Tall* to a degree of 0.8 would belong to a fuzzy set of young and tall people to a degree of 0.8 ($= min(0.9, 0.8)$).

One of the problems with the definitions of conjunction and disjunction in Equations 4.5 and 4.6 is that some tautologies from Boolean logic are not true for the fuzzy case. For instance, "A or not A" ($A \vee \neg A$) results in true in the case of Boolean logic. In fuzzy logic the result is $max\{\mu(A), 1 - \mu(A)\}$ which only results in "true" or "1" if $\mu(A)$ is either 0 or 1. We will therefore use the following two functions for conjunction and disjunction which have been proposed instead of the functions described above:

- conjunction: Let $\mu_A(x)$ and $\mu_B(x)$ be fuzzy membership functions of fuzzy sets A and B respectively. The conjunction of fuzzy sets A and B is given by:
$$\mu_{A \wedge B}(x) = \mu_A(x) \times \mu_B(x). \tag{4.8}$$

- disjunction: Let $\mu_A(x)$ and $\mu_B(x)$ be fuzzy membership functions of fuzzy sets A and B respectively. The disjunction of fuzzy sets A and B is given by:
$$\mu_{A \vee B}(x) = min\{\mu_A(x) + \mu_B(x), 1\}. \tag{4.9}$$

With these functions $\mu_{A \vee \neg A} = 1$ for the fuzzy case. However, some laws from Boolean logic (e.g., $A \wedge \neg A = 0(false)$) are not true for the fuzzy case.

4.3 Fuzzy Decision Tree Representations

In Chapter 3 we introduced several *top-down atomic* representations which were aimed at refining the search space by clustering or partitioning the domain of numerical valued attributes. In this section we will show how the

"Boolean" representations of Chapter 3 can be changed into fuzzy representations. An example of a *fuzzy* decision tree is shown in Figure 4.2, its meaning will be explained in Section . In a fuzzy decision tree we do not have the less-than ($<$), greater-equal-than ($\geq$) and set operators used by the *clustering* and *partitioning* representations for the numerical valued attributes, but instead we use more intuitive fuzzy terms like *Young* and *Tall* which correspond to specific fuzzy sets. These fuzzy sets are defined through a process, called fuzzification, which defines a (fixed) maximum number of fuzzy sets for the domain of each numerical valued attribute.

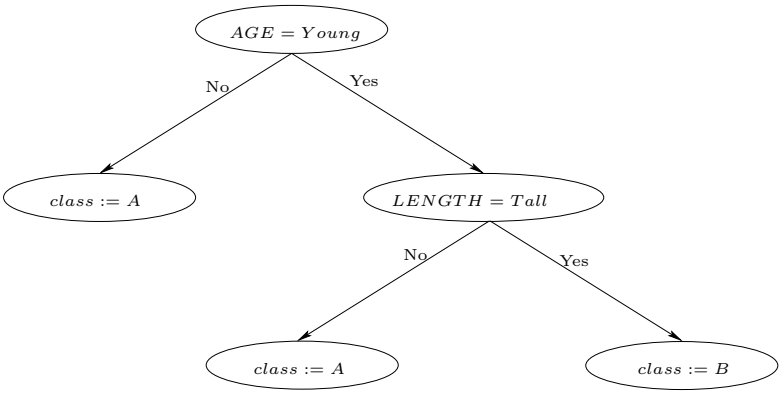

Figure 4.2: An example of a *fuzzy* tree.

4.3.1 Fuzzification

In our fuzzy representations the partitions and clusters of the *partitioning* and *clustering* GP algorithms are converted into fuzzy sets by a process called fuzzification. The fuzzification process (see Figure 4.3) used here is based on the method that was used for fuzzy association rule mining in [48, 80]. We first select all the numerical attributes and cluster or partition each attribute's possible values into a finite and fixed maximum number of partitions or clusters as described in Sections 3.3 and 3.4. Once we have determined the "Boolean" clusters and partitions we can convert them into fuzzy sets using fuzzy membership functions.

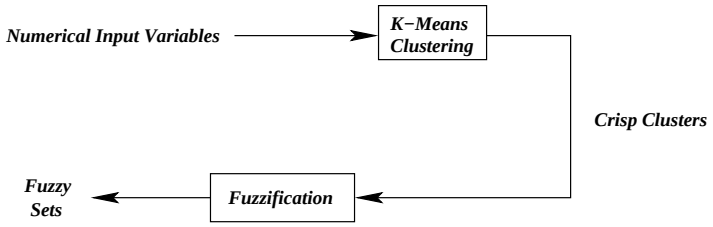

Figure 4.3: An overview of the fuzzification process.

Definition 4.3.1 *Let $F = \{F_1, \ldots, F_k\}$ be the set of fuzzy sets for a particular numerical valued attribute and let $C = \{c_1, \ldots, c_k\}$ be the set of its corresponding cluster or partition centroids, then we define the following fuzzy membership functions:*

$$\mu_{F_1}(x) = \begin{cases} 1 & \text{if } x \leq c_1, \\ \frac{c_2 - x}{c_2 - c_1} & \text{if } c_1 < x < c_2, \\ 0 & \text{if } x \geq c_2; \end{cases} \tag{4.10}$$

$$\mu_{F_i}(x) = \begin{cases} 0 & \text{if } x \leq c_{i-1}, \\ \frac{c_{i-1} - x}{c_{i-1} - c_i} & \text{if } c_{i-1} < x < c_i, \\ 1 & \text{if } x = c_i, \\ \frac{c_{i+1} - x}{c_{i+1} - c_i} & \text{if } c_i < x < c_{i+1}, \\ 0 & \text{if } x \geq c_{i+1}; \end{cases} \tag{4.11}$$

$$\mu_{F_k}(x) = \begin{cases} 0 & \text{if } x \leq c_{k-1}, \\ \frac{c_{k-1} - x}{c_{k-1} - c_k} & \text{if } c_{k-1} < x < c_k, \\ 1 & \text{if } x \geq c_k, \end{cases} \tag{4.12}$$

where $i = 2, 3, \ldots, k-1$ (see Figure 4.4).

Note that the centroids of clusters are calculated during the K-means clustering process. In the case of partitioning using the *gain* or *gain_ratio* criteria the average value within a partition is used as the centroid. Also note that the distances between the different centroids c_i are not necessarily equal. By using simple triangular membership functions instead of the often used trapezium shaped functions we avoid the need for extra parameters. Finally note that for every x-value the sum of all membership function equals 1.

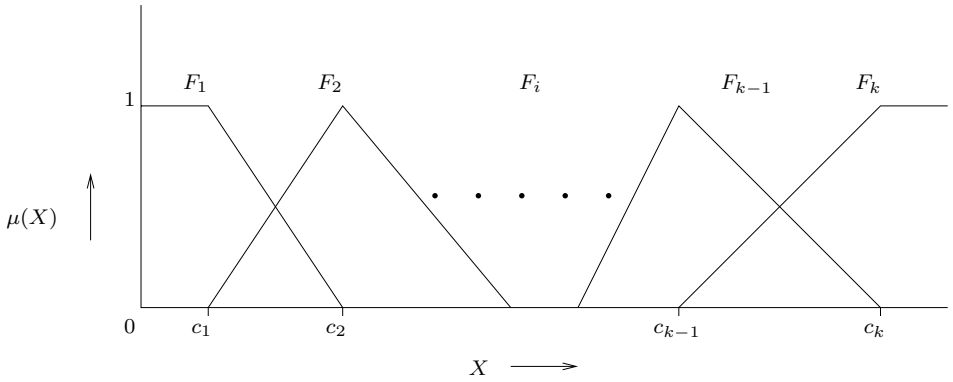

Figure 4.4: The k fuzzy membership functions $\mu_{F_1}, \ldots, \mu_{F_k}$ for a numerical valued attribute.

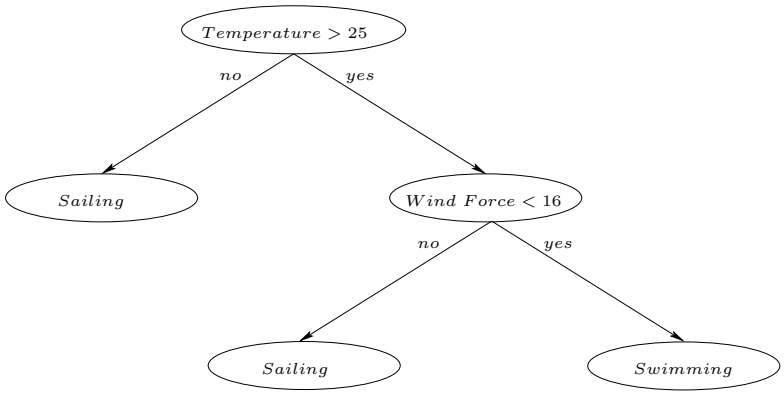

Figure 4.5: An example of a *simple top-down atomic* decision tree.

4.3.2 Evaluation Using Fuzzy Logic

Every *top-down atomic* tree or other "Boolean" decision tree can be rewritten as a set of classification rules using Boolean logic. For instance, the tree in Figure 4.5 can be written as:

$$\text{If } (\text{T} > 25) \wedge \ (\text{WF} < 16) \text{ then class} := \text{Swimming} \tag{4.13}$$

and

$$\text{If } (\text{T} \not> 25) \vee (\text{T} > 25 \ \wedge \ \text{WF} \not< 16) \text{ then class} := \text{Sailing}, \tag{4.14}$$

where T stands for Temperature and WF stands for Wind Force. If we would do the same for the similar fuzzy decision tree in Figure 4.6 we would get:

$$\text{If } (\text{T} = \text{Warm}) \wedge \ (\text{WF} = \text{Light}) \text{ then class} := \text{Swimming} \tag{4.15}$$

and

$$\text{If } (\text{T} \neq \text{Warm}) \vee (\text{T} = \text{Warm} \ \wedge \ \text{WF} \neq \text{Light}) \text{ then class} := \text{Sailing}. \tag{4.16}$$

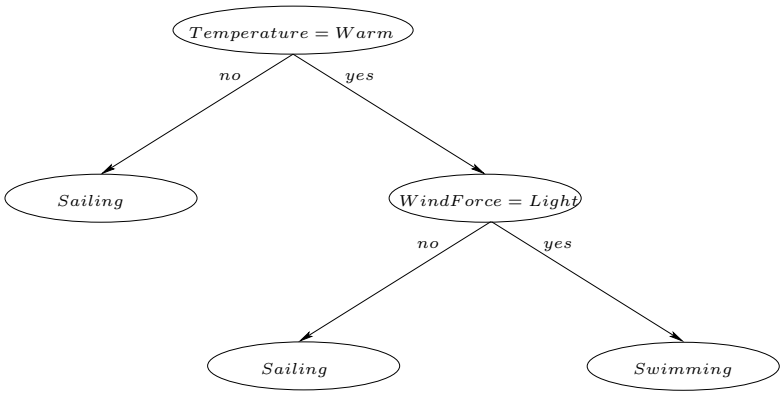

Figure 4.6: Example of Figure 4.5, recast as a *fuzzy* decision tree.

However, since the result of a fuzzy atom is a fuzzy membership value rather than a Boolean value we use fuzzy logic operators. By replacing the Boolean operators with their fuzzy logic counterparts defined in Equations 4.7, 4.8 and 4.9, Equations 4.15 and 4.16 become:

$$\mu_{\text{Swimming}} := \mu_{(\text{T=Warm})} \times \mu_{(\text{WF=Light})} \tag{4.17}$$

and

$$\mu_{\text{Sailing}} := \mu_{(\text{T}\neq\text{Warm})} + (\mu_{(\text{T=Warm})} \times \mu_{\ (\text{WF}\neq\text{Light})}). \tag{4.18}$$

Note that we can use $\mu_{A\vee B} = \mu_A + \mu_B$ instead of Equation 4.9 because the sum of the membership values for the possible target classes is always 1 for our decision (sub)trees. We also no longer have a simple "If − then" statement but a fuzzy membership function defining each class.

Example 4.3.1 Observe the fuzzy decision tree depicted in Figure 4.6. Suppose we need to classify an instance I with the attributes $I_{\text{Temperature}}$ and $I_{\text{WindForce}}$; assume the fuzzy membership function $\mu_{\text{Warm}}(I_{\text{Temperatute}})$ returns 0.9 and $\mu_{\text{Light}}(I_{\text{WindForce}})$ returns 0.3. We can then calculate the class membership values μ_{Swimming} and μ_{Sailing} using the fuzzy membership functions in Equations 4.17 and 4.18. This gives us:

$$\mu_{\text{Swimming}} := 0.9 \times 0.3 = 0.27$$

and

$$\mu_{\text{Sailing}} := (1 - 0.9) + (0.9 \times (1 - 0.3)) = 0.73.$$

Thus in this case we could say that the tree in Figure 4.6 "predicts" the class of instance I to be *Sailing* (since $0.73 > 0.27$). It also gives a sort of confidence of 73% to this classification. □

4.4 Experiments and Results

In order to assess the performance of our *fuzzy* representations we will compare them to our non-fuzzy representations from the previous chapters as well as with the other evolutionary (CEFR-MINER and ESIA) and non-evolutionary (Ltree, OC1 and C4.5) classification algorithms introduced in Chapter 2. N/A indicates that no results were available. The experimental setup and data sets are the same as in the previous chapters and are described in Sections 2.7 and 2.8. An overview of the most important GP parameters can be found in Table 4.1. In the case of our *partitioning* GP algorithms the criterion used,

either *gain* or *gain_ratio*, is indicated between brackets '(' and ')'. The tables with results contain a column, labeled k, to indicate the number of clusters in the case of our *clustering* GP algorithms or the maximum number of partitions in the case of the *partitioning* GP algorithms. The best (average) result for each data set is printed in bold font. Because the *partitioning* GP algorithms do not scale well with parameter k we will only look at a maximum of three partitions, clusters or fuzzy sets per numerical valued attribute.

Table 4.1: The main GP parameters.

Parameter	**Value**
Population Size	100
Initialization	ramped half-and-half
Initial maximum tree depth	6
Maximum number of nodes	63
Parent Selection	Tournament Selection
Tournament Size	5
Evolutionary model	(μ, λ)
Offspring Size	200
Crossover Rate	0.9
Crossover Type	swap subtree
Mutation Rate	0.9
Mutation Type	branch mutation
Stop condition	99 generations

To determine if the results obtained by our algorithms are statistically significantly different from the results reported for ESIA, CEFR-MINER, Ltree, OC1 and C4.5, we have performed two-tailed independent samples t-tests with a 95% confidence level ($p = 0.05$) using the reported mean and standard deviations. The null-hypothesis in each test is that the means of the two algorithms involved are equal. In order to determine whether the differences between our GP algorithms are statistically significant we used paired two-tailed t-tests with a 95% confidence level ($p = 0.05$) using the results of 100 runs (10 random seeds times 10 folds). In these tests the null-hypothesis is also that the means of the two algorithms involved are equal.

4.4.1 The Australian Credit Data Set

If we look at the results of our new *fuzzy* GP representations on the Australian Credit data set in Table 4.2 we see that the fuzzy and non-fuzzy algorithms perform similarly.

Table 4.2: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Australian Credit data set.

algorithm	**k**	**average**	**s.d.**
fuzzy clustering GP	2	13.8	2.6
fuzzy clustering GP	3	14.7	3.0
fuzzy gain GP	2	14.3	2.8
fuzzy fain GP	3	14.7	2.8
fuzzy gain_ratio GP	2	15.5	3.1
fuzzy gain_ratio GP	3	15.2	3.3
clustering GP	2	**13.7**	3.0
clustering GP	3	14.8	2.6
gain GP	2	14.2	2.7
gain GP	3	15.1	2.7
gain_ratio GP	2	15.7	3.2
gain_ratio GP	3	15.5	3.4
simple GP		22.0	3.9
Ltree		13.9	4.0
OC1		14.8	6.0
C4.5		15.3	6.0
CEFR-MINER		N/A	
ESIA		19.4	0.1
default		44.5	

4.4.2 The German Credit Data Set

On the German Credit data set (see Table 4.3) our *fuzzy gain_ratio* GP algorithms perform significantly better than their non-fuzzy counterparts. The differences in performance between our fuzzy and non-fuzzy *clustering* GP algorithms are not significant. In the case of the fuzzy and non-fuzzy *gain*

GP algorithms we see that for $k = 2$ the *fuzzy gain* GP algorithm is better, for $k = 3$ the non-fuzzy version is better.

All our fuzzy GP algorithms are significantly better than ESIA while the differences with Ltree, OC1 and C4.5 are not statistically significant.

Table 4.3: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the German Credit data set.

algorithm	**k**	**average**	**s.d.**
fuzzy clustering GP	2	27.4	1.5
fuzzy clustering GP	3	27.5	1.1
fuzzy gain GP	2	26.8	2.3
fuzzy gain GP	3	28.0	1.7
fuzzy gain_ratio GP	2	26.7	2.7
fuzzy gain_ratio GP	3	26.8	2.4
clustering GP	2	27.8	1.3
clustering GP	3	28.0	1.4
gain GP	2	28.1	1.9
gain GP	3	27.1	1.9
gain_ratio GP	2	28.3	1.9
gain_ratio GP	3	28.5	2.3
simple GP		27.1	2.0
Ltree		26.4	5.0
OC1		**25.7**	5.0
C4.5		29.1	4.0
CEFR-MINER		N/A	
ESIA		29.5	0.2
default		30.0	

4.4.3 The Pima Indians Diabetes Data Set

If we look at the results in Table 4.4 we see that all the *fuzzy* GP algorithms have a significantly better classification performance for $k = 2$ than their respective Boolean versions. For $k = 3$ the differences are only statistically significant for our fuzzy and non-fuzzy *gain_ratio* GP algorithms.

Compared to ESIA all our fuzzy GP algorithms perform better. The differences in classification performance between Ltree, OC1 and C4.5 and our fuzzy GP algorithms are not statistically significant.

Table 4.4: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Pima Indians Diabetes data set.

algorithm	**k**	**average**	**s.d.**
fuzzy clustering GP	2	**24.2**	4.2
fuzzy clustering GP	3	25.8	3.8
fuzzy gain GP	2	24.7	4.2
fuzzy gain GP	3	26.6	4.4
fuzzy gain_ratio GP	2	25.5	4.1
fuzzy gain_ratio GP	3	25.7	4.3
clustering GP	2	26.3	3.2
clustering GP	3	26.3	3.4
gain GP	2	27.0	4.3
gain GP	3	26.5	3.4
gain_ratio GP	2	27.6	4.6
gain_ratio GP	3	27.6	4.6
simple GP		26.3	3.6
Ltree		24.4	5.0
OC1		28.0	8.0
C4.5		24.7	7.0
CEFR-MINER		N/A	
ESIA		29.8	0.2
default		34.9	

4.4.4 The Heart Disease Data Set

The results on the Heart Disease dataset are displayed in Table 4.5. The differences between the fuzzy GP algorithms and their non-fuzzy versions are very similar and not statistically significant. All fuzzy GP algorithms perform significantly better than our *simple* GP algorithms, ESIA and OC1. Only the performance of our *fuzzy gain_ratio* GP algorithm with $k = 3$ is not significantly worse than Ltree.

Table 4.5: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Heart Disease data set.

algorithm	**k**	**average**	**s.d.**
fuzzy clustering GP	2	19.8	4.1
fuzzy clustering GP	3	20.9	4.4
fuzzy gain GP	2	19.6	4.3
fuzzy gain GP	3	21.8	3.1
fuzzy gain_ratio GP	2	20.6	5.0
fuzzy gain_ratio GP	3	19.6	5.0
clustering GP	2	19.9	4.6
clustering GP	3	21.3	5.4
gain GP	2	19.9	4.8
gain GP	3	22.8	4.3
gain_ratio GP	2	18.7	4.5
gain_ratio GP	3	20.3	5.3
simple GP		25.2	4.8
Ltree		**15.5**	4.0
OC1		29.3	7.0
C4.5		21.1	8.0
CEFR-MINER		17.8	7.1
ESIA		25.6	0.3
default		44.0	

4.4.5 The Ionosphere Data Set

On the Ionosphere data set (Table 4.6) we see again that fuzzification has a negative effect on classification performance when the performance of the original Boolean GP algorithms is quite good. Only in the case of our *clustering* GP algorithm using $k = 2$ does fuzzification significantly improve the performance. In the case of the *gain* GP and *gain_ratio* GP algorithms fuzzification significantly decreases the classification performance.

Table 4.6: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Ionosphere data set.

algorithm	**k**	**average**	**s.d.**
fuzzy clustering GP	2	11.4	3.8
fuzzy clustering GP	3	10.8	3.4
fuzzy gain GP	2	10.6	4.1
fuzzy gain GP	3	11.8	3.7
fuzzy gain_ratio GP	2	10.5	4.7
fuzzy gain_ratio GP	3	10.6	4.2
clustering GP	2	13.1	4.1
clustering GP	3	10.5	5.1
gain GP	2	8.3	4.7
gain GP	3	10.5	4.8
gain_ratio GP	2	**7.6**	4.2
gain_ratio GP	3	8.1	4.3
simple GP		12.4	3.8
Ltree		9.4	4.0
OC1		11.9	3.0
C4.5		9.1	5.0
CEFR-MINER		11.4	6.0
ESIA		N/A	
default		35.9	

Table 4.7: Average misclassification rates (in %) with standard deviation, using 10-fold cross-validation for the Iris data set.

algorithm	**k**	**average**	**s.d.**
fuzzy clustering GP	2	6.8	5.5
fuzzy clustering GP	3	5.3	5.1
fuzzy gain GP	2	6.0	5.7
fuzzy gain GP	3	4.7	4.7
fuzzy gain_ratio GP	2	5.9	5.2
fuzzy gain_ratio GP	3	5.9	5.2
clustering GP	2	21.1	9.4
clustering GP	3	**2.1**	4.2
gain GP	2	29.6	6.3
gain GP	3	6.3	6.1
gain_ratio GP	2	31.7	5.0
gain_ratio GP	3	31.7	5.0
simple GP		5.6	6.1
Ltree		2.7	3.0
OC1		7.3	6.0
C4.5		4.7	5.0
CEFR-MINER		4.7	0.0
ESIA		4.7	7.1
default		33.3	

4.4.6 The Iris Data Set

In Table 4.7 the results on the Iris data set are displayed. On this data set our *clustering* GP algorithms using $k = 3$ is still significantly better than all our other GP algorithms. In the case of the other *fuzzy* GP algorithms the fuzzy versions are significantly better than the non-fuzzy versions. We already noted above that it seems that fuzzification improves classification performance when the original clusters or partitions result in sub-optimal performance. However, on the Iris data set the performance increase for our non-fuzzy *clustering* and *partitioning* GP algorithms is huge. Fuzzification reduces the misclassificaton rate for our *clustering* GP algorithm with $k = 2$ by more than two-thirds, the misclassification rate of our *partitioning* GP

algorithm using the *gain* criterion and $k = 2$ by a factor of 5 and the misclassification rates for our *partitioning* GP algorithms using the *gain_ratio* criterion also with more than 80%. The differences between our fuzzy GP algorithms and the other algorithms are not statistically significant.

4.4.7 Comparing Fuzzy and Non-Fuzzy

To compare our *fuzzy* GP algorithms with their non-fuzzy counterparts we have calculated the average misclassification rate of each algorithm for the six data sets regardless of the number of clusters or partitions. As can be seen in Table 4.8 there is very little difference between the fuzzy and non-fuzzy versions. Only on the Pima Indians Diabetes and Iris data set can we see any real improvement of our *fuzzy* GP algorithms over their non-fuzzy counterparts. On the Ionosphere data set fuzzification has a negative effect on both the *gain* GP and *gain_ratio* GP algorithms.

Table 4.8: The average misclassification rates (in %) of the different algorithms for all six data sets.

data set	*clustering* GP		*gain* GP		*gain_ratio* GP	
	fuzzy	normal	*fuzzy*	normal	*fuzzy*	normal
Australian Credit	14.3	14.3	14.5	14.7	15.4	15.6
German Credit	27.5	27.9	27.4	27.6	26.8	28.4
Pima Indians	25.0	26.3	25.7	26.8	25.6	27.6
Heart disease	20.4	20.6	20.7	21.4	20.1	19.5
Ionosphere	11.1	11.8	11.2	9.4	10.6	7.9
Iris	6.1	11.6	5.4	18.0	5.9	31.7

4.5 A Fuzzy Fitness Measure

In Section 2.6 we introduced a multi-layered fitness for data classification consisting of two fitness measures: misclassification percentage and decision tree size. The misclassification percentage measure for an individual x is calculated by:

$$\mathit{fitness}_{standard}(x) = \frac{\sum_{r \in \mathit{training\ set}} \chi(x, r)}{|\mathit{training\ set}|} \times 100\%, \tag{4.19}$$

where

$$\chi(x, r) = \begin{cases} 1 & \text{if } x \text{ classifies record } r \text{ incorrectly;} \\ 0 & \text{otherwise.} \end{cases} \tag{4.20}$$

However, in the case of our fuzzy representation each individual returns a set of class membership values for each record to be classified. We can use these membership values in order to try to compute a more precise *fuzzy* fitness value:

$$\mathit{fitness}_{fuzzy}(x) = \frac{\sum_{r \in \mathit{training\ set}} (1 - \mu_c(x, r))}{|\mathit{training\ set}|} \times 100\%, \tag{4.21}$$

where $\mu_c(x, r)$ is the membership value for the target class c of record r returned by individual x. For Example 4.3.1, we would have $\mu_{Sailing}(x, I) = 0.73$ for the tree x under consideration. Note that this *fuzzy* fitness equals the standard fitness for non-fuzzy representations.

We can add this fuzzy fitness measure as the primary fitness measure to our multi-layered fitness. The secondary fitness measure is now the original standard fitness and the size of the decision tree (the number of tree nodes) becomes the third and last fitness measure. Since our algorithms are compared based on the number of misclassifications on the test set we have two choices when selecting the best individual found by our GP algorithms. We can use all three fitness values (fuzzy, standard and tree size) to select the best individual of an evolutionary run, but alternatively we can only use the second (*standard fitness*) and third (*tree size*) fitness measures as is done above and in the previous chapters.

In Table 4.9 the results for the three fitness configurations on all six data sets are displayed. In most cases the differences are very small. Only on the German Credit and Iris data sets we can see a clear distinction between using a fuzzy fitness measure or not. On the German Credit data set only using the standard fitness measure works best. However, on the Iris data set using the fuzzy fitness measure as primary fitness value both during evolution and selecting the final best individual seems to work best. In the case of the other data sets there is no clear best fitness measure for either evolution or selecting the best individual at the end of the evolutionary process.

Table 4.9: The average misclassification rates (in %) for the data sets using different fitness schemes.

		primary	**select best**	**fuzzy**		
data set	k	**fitness**	**fitness**	**clustering**	**gain**	**gain_ratio**
Australian Credit	2	standard	standard	13.8	14.3	15.5
Australian Credit	2	fuzzy	fuzzy	14.3	14.3	15.4
Australian Credit	2	fuzzy	standard	14.2	14.0	15.4
Australian Credit	3	standard	standard	14.7	14.7	15.2
Australian Credit	3	fuzzy	fuzzy	14.4	15.0	15.3
Australian Credit	3	fuzzy	standard	14.5	15.1	15.4
German Credit	2	standard	standard	27.4	26.8	26.7
German Credit	2	fuzzy	fuzzy	27.7	28.1	27.9
German Credit	2	fuzzy	standard	27.6	27.7	27.1
German Credit	3	standard	standard	27.5	28.0	26.8
German Credit	3	fuzzy	fuzzy	28.5	28.7	27.9
German Credit	3	fuzzy	standard	28.3	28.2	27.5
Pima Indians	2	standard	standard	24.2	24.7	25.5
Pima Indians	2	fuzzy	fuzzy	24.6	24.9	25.6
Pima Indians	2	fuzzy	standard	24.8	24.6	25.3
Pima Indians	3	standard	standard	25.8	26.6	25.7
Pima Indians	3	fuzzy	fuzzy	25.2	25.5	24.7
Pima Indians	3	fuzzy	standard	25.6	25.8	24.6
Heart Disease	2	standard	standard	19.8	19.6	20.6
Heart Disease	2	fuzzy	fuzzy	19.2	20.3	19.4
Heart Disease	2	fuzzy	standard	19.0	19.5	19.2
Heart Disease	3	standard	standard	20.8	21.8	19.6
Heart Disease	3	fuzzy	fuzzy	19.5	20.3	19.6
Heart Disease	3	fuzzy	standard	20.0	20.8	19.1
Ionosphere	2	standard	standard	11.4	10.6	10.6
Ionosphere	2	fuzzy	fuzzy	11.1	10.5	10.7
Ionosphere	2	fuzzy	standard	10.9	9.9	10.4
Ionosphere	3	standard	standard	10.8	11.8	10.6
Ionosphere	3	fuzzy	fuzzy	9.8	10.8	10.7
Ionosphere	3	fuzzy	standard	10.1	11.0	10.4
Iris	2	standard	standard	6.8	6.0	5.9
Iris	2	fuzzy	fuzzy	4.4	4.7	4.6
Iris	2	fuzzy	standard	5.2	5.5	5.3
Iris	3	standard	standard	5.3	4.7	5.9
Iris	3	fuzzy	fuzzy	5.2	3.7	4.6
Iris	3	fuzzy	standard	4.6	4.3	5.3

4.6 Conclusions

We have introduced new representations using a combination of clustering, partitioning and fuzzification for classification problems using genetic programming. By evolving fuzzy decision trees we have a method for dealing with continuous valued attributes in an intuitive and natural manner. In theory the use of fuzzy atoms should make our decision trees more comprehensible. Additionally the fuzzy membership functions found for the target classes can give additional information about the relations, if any, between those classes.

Looking at the results we see that in general our fuzzy GP algorithms perform similar to or better than their non-fuzzy counterparts. Our *fuzzy* GP algorithms are especially good in cases where our non-fuzzy algorithms failed. On the Iris data set using only 2 partitions or clusters our *partitioning* and *clustering* GP algorithms failed while fuzzy versions of the algorithms have a misclassification rate which is 3 to 5 times lower. However, in some cases where a non-fuzzy algorithm managed to perform very well fuzzification seems to have a negative effect. A possible explanation is that the fuzzification process, which is meant to make our fuzzy decision trees more robust towards faulty and polluted input data, also makes the fuzzy decision trees more robust towards non-optimal clustering and partitioning. It is also possible that the data sets on which our *fuzzy* GP algorithms outperformed the non-fuzzy algorithms contained more faulty or polluted attributes.

We also experimented with a new fuzzy fitness measure in combination with the new fuzzy representation. At this point there is no clear indication whether this fuzzy fitness measure should be used, or when.

5 Introns: Detection and Pruning

We show how the understandability and speed of genetic programming classification algorithms can be improved, without affecting the classification accuracy. By analyzing the decision trees evolved we can remove unessential parts, called GP *introns*, from the discovered decision trees. The resulting trees are smaller and easier to understand. Moreover, by using these pruned decision trees in a fitness cache we can reduce the number of fitness calculations.

5.1 Introduction

Algorithms for data classification are generally assessed on how well they can classify one or more data sets. However, good classification performance alone is not always enough. Almost equally important can be the understandability of the results. Another aspect is the time it takes them to learn to classify a data set.

In the previous chapters we focussed mainly on improving the classification accuracy of our GP algorithms. The classification accuracy is generally the most important aspect when comparing algorithms for data classification. However, other characteristics of classification algorithms can also be important. An algorithm which classifies a data set perfectly (0% misclassification rate) but takes a long time to find an answer, might be less attractive than a fast algorithm with an acceptable (e.g., 5%) misclassification rate. Another (perhaps more critical) aspect for data classification algorithms is the understandability of the results.

We already applied several methods to improve/optimize the understandability of the decision trees evolved by our GP algorithms. All our GP classifiers use a maximum tree size, thereby limiting not only the size of the search space but also the size of the solutions found. We also introduced the tree size fitness measure, as part of our *multi-layered* fitness, such that smaller, and thus often easier to understand, decision trees are favored over equally good classifying larger trees. The fuzzy decision trees evolved by our *fuzzy* GP algorithms in Chapter 4 are designed to be more intuitive, which not only results in more accurate classifiers but also in decision trees which should be easier to understand.

Unfortunately, despite the use of the tree size fitness measure the decision trees are on average still larger than strictly necessary. Analysis of the decision trees evolved by our GP algorithms shows that the trees sometimes contain parts, nodes or subtrees, which can be removed without influencing the classification outcome of the tree. In this chapter we introduce methods to detect and prune these extraneous parts, called *introns* or ineffective code [71], of our *top-down atomic* trees (see Section 2.3) and investigate to what extent their removal decreases the size of our decision trees. Additionally, we will show how, in combination with the *fitness cache* introduced in Section 2.9, the detection and pruning methods allow us to reduce the time spent on fitness evaluations.

The outline of the rest of this chapter is as follows. In Section 5.2 we give a short introduction regarding *introns*. Then in Section 5.3 we describe the *intron* detection and pruning process. The results of several different *intron* detection and pruning strategies are discussed in Section 5.4. In Section 5.5 we draw our conclusions.

5.2 Genetic Programming Introns

One of the problems of variable length evolutionary algorithms, such as tree-based Genetic Programming, is that the genotypes of the individuals tend to increase in size until they reach the maximum allowed size. This phenomenon is, in Genetic Programming, commonly refered to as *bloat* [97, 4] and is caused by GP *introns* [79, 96, 95, 71]. The term *introns* was first introduced in the field of Genetic Programming, and evolutionary computation in general, by Angeline [1] who compared the emergence of extraneous code in variable length GP structures to biological introns. In biology the term introns is used

for parts of the DNA, sometimes referred to as junk DNA, which do not have any apparent function as they are not transcribed to RNA. In Genetic Programming, the term *introns* is used to indicate parts of an evolved solution which do not influence the result produced by, and thus fitness of, the solution (other than increasing its size).

As our GP classifiers use variable length (decision) tree structures they are also subject to *bloat* and they will thus also contain *introns*. In the case of our *top-down atomic* decision tree representations we can distinguish between two types of *introns*:

1. *intron subtrees*: subtrees which are never traversed (e.g., if (true) then *normal* subtree else *intron* subtree), and

2. *intron nodes*: nodes which do not influence the outcome (e.g., if $(X > 1)$ then *class:=* A else *class:=* A).

The occurrence of *introns* in variable length evolutionary algorithms has both positive and negative effects. A positive effect of *introns* is that they can offer protection against the destructive effects of crossover operators [79, 74]. However, earlier studies [79, 96, 95] also show that more than 40% of the code in a population can consist of *introns*.

In the case of our *top-down atomic* GP algorithms the negative effects of *bloat* and *introns* are two-fold. The bloated decision trees found by our algorithms can contain *introns* which make them less understandable than semantically equivalent trees without *introns*. The second problem of *introns* in our decision trees is related to computation times. Although *intron nodes* do not influence the classification outcome of a decision tree their evaluation takes time. More importantly, *introns* also reduce the effectiveness of our fitness cache (see Section 2.9) since it does not recognize semantically equivalent but syntactically different trees.

In the next section we will discuss methods to detect *introns* in our decision trees so the trees can be pruned, removing most of the *introns* and their negative effects from the trees.

5.3 Intron Detection and Pruning

In [60] Johnson proposes to replace the standard fitness measures generally used with evolutionary algorithms with static analysis methods: by us-

ing static analysis techniques it should be possible to evaluate an individual's behaviour across the entire input space instead of a limited number of test cases. However, for data classification replacing the fitness measure with static analysis techniques does not seem feasible due to the high dimensional nature of the search space. Instead of replacing the fitness function with static analysis methods, Keijzer [63] showed that static analysis can also be used as a pre-processing step for fitness evaluations. By using interval arithmetic to calculate the bounds of regression trees, functions containing undefined values are either deleted or assigned the worst possible performance value.

We will use a combination of static analysis techniques and (semantic) pruning to detect and remove *introns* from our *top-down atomic* decision trees in order to address the problem caused by GP *introns*. A schematic overview of the GP intron detection and pruning process can be seen in Figure 5.1 where the black nodes indicate an *intron subtree* and the gray node indicates an *intron node*.

After a new individual is created, either as a result of initialization or as the result of recombination and mutation of its parent(s), it is usually evaluated in order to determine its fitness. In our case we first scan the individual for the different types of *introns* using static analysis, resulting in an individual in which the different types of *introns* are marked. Depending on the types of *introns* the individual is pruned into a more *condensed* decision tree which is semantically the same as the original individual. The *condensed* decision tree is then used to evaluate the individual. Additionally, we will use the *condensed* trees in our *fitness cache*. This should improve the effectiveness of the cache as several syntactically different decision trees have the same *condensed* form. For the evolutionary process (e.g., crossover and mutation) as well as the tree size fitness measure we will still use the original *top-down atomic* trees so that the classification performance of our algorithms remains the same as without intron detection and pruning.

The approach described above is related to several code-editing techniques [69, Section 11.6]. In [13] Brameier and Banzhaf distinguish two types of *introns* in their Linear Genetic Programming (LGP) system. The first type, *structural introns*, are single noneffective intructions that manipulate variables that are not used to calculate the outcome of their genetic programs. The second type, *semantical introns*, are instructions or sequences of instructions in which the state of the relevant variables remains constant. By removing all *structural introns* from a genetic program before executing it during fitness calculation they achieve a "significant decrease in runtime".

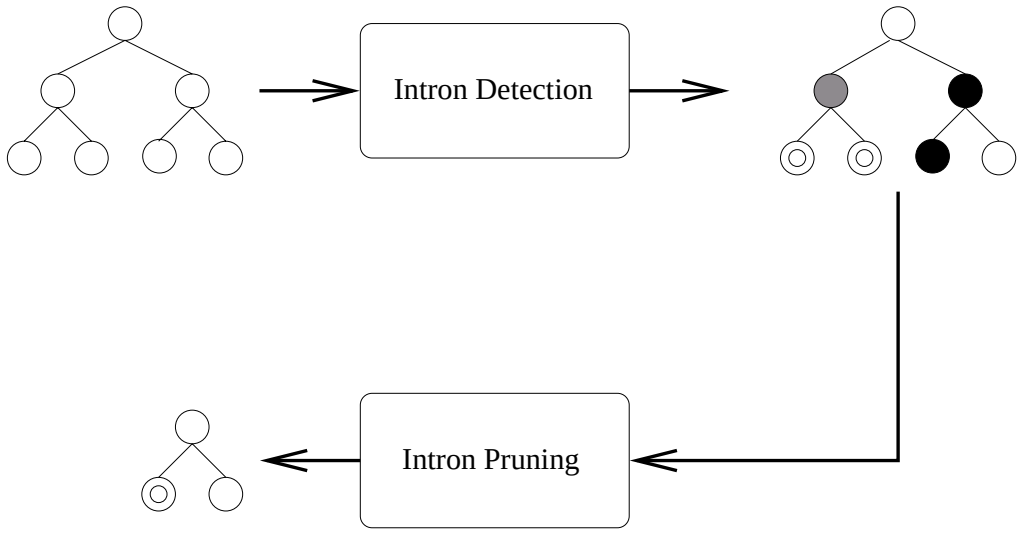

Figure 5.1: The intron detection and pruning process. The upper left tree is presented to the intron detection algorithm. The gray node in the upper right tree indicates an *intron node* that has been detected in the original left subtree. The black nodes indicate a detected *intron subtree*. After the introns have been detected they are pruned resulting in the lower left tree. The gray *intron node* is replaced by one of its child nodes (with an extra circle inside). The *intron subtree* is replaced by the (non-intron) child of the root node of the *intron subtree*, in this case the white leaf node without extra circle.

Since the altered genetic programs are not written back there are no changes to the individuals in the population, similar to our approach.

There are also approaches in which the geneotypes of the individuals in the population are altered. In [8] Blickle investigates the use of a *deleting crossover* on regression problems. This crossover operator replaces all subtrees that were not traversed during the evaluation in the fitness function with a randomly chosen terminal. This operator was found to only work for discrete regression problems and not for continuous ones. In [56] Hooper and Flann use expression simplification to replace *introns* with simpler but equivalent expressions in order to improve accuracy and robustness. To control the application of expression simplification they use a *simplification rate* parameter that defines the probability that an individual in a generation will be simplified.

5.3.1 Intron Subtrees

Intron subtrees are parts (subtrees) of non-fuzzy *top-down atomic* decision trees which can and will never be traversed because of the outcome of nodes higher up in the tree. An example of an intron subtree is shown in Figure 5.2.

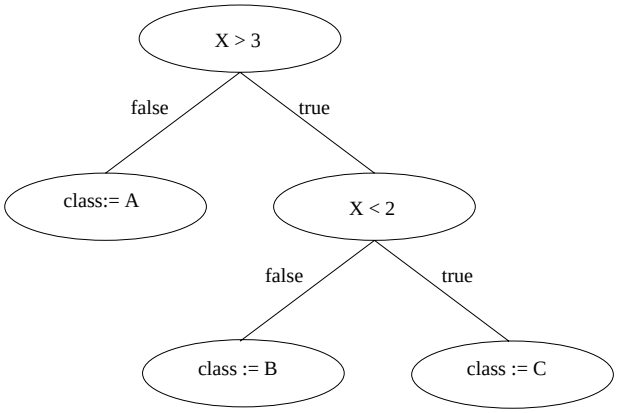

Figure 5.2: A *top-down atomic* decision tree containing an intron subtree ($class := C$).

Once the root node containing $X > 3$ has been evaluated either the left or the right branch is traversed depending on the value of X. If X is less than or equal to 3 the left branch is traversed resulting in class **A**. If X has a value higher than 3 the right branch is traversed and the second internal node $X < 2$ is reached. Normally this node would be evaluated for every possible data record (with a value of X greater than 3). However, an analysis of the decision tree traversed so far shows that this node will always result in *false* since node $X < 2$ is reached only if X has a value higher than 3. Therefore, the right branch of node ($X < 2$) will never be traversed, making it an *intron subtree* since it does not influence the behaviour of the decision tree.

In order to detect *intron subtrees*, we recursively propagate the possible domains of the attributes through the decision trees in a top-down manner. Given a certain data set we determine for each attribute X_i the domain $D(X_i)$ of possible values. By propagating the domain of each attribute X_i recursively through the non-fuzzy *top-down atomic* trees we can identify situations in which the domain of an attribute becomes empty ($\emptyset$), indicating the presence of an *intron subtree*.

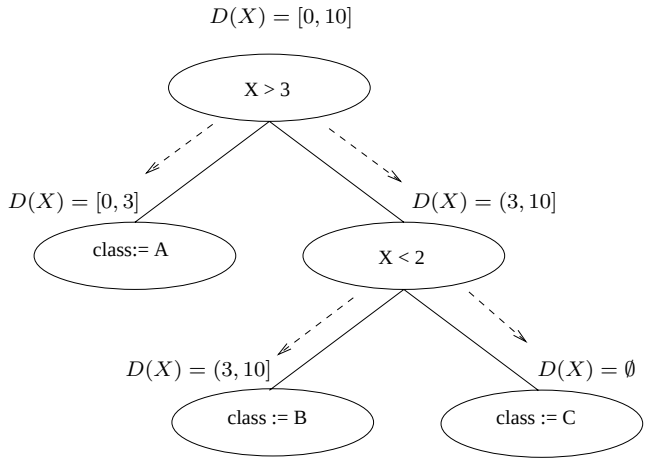

Figure 5.3: A *top-down atomic* decision tree containing an intron subtree with the domain of attribute X displayed at each point in the tree.

Observe the *top-down atomic* decision tree in Figure 5.3. Let X be a continuous valued attribute in the range $[0, 10]$. Before evaluation of the atom in the root node ($X > 3$) the domain of X is $[0, 10]$. Just as the atom splits a data set into two parts, the domain of possible values of X is split into two. In the left subtree the domain of X is limited to $[0, 3]$ and in the right subtree the domain of possible values for X is $(3, 10]$. Thus if the second internal node of the tree is reached it means that X has a value greater than 3 and less than or equal to 10. After the evaluation of the second internal node ($X < 2$) of the decision tree in Figure 5.3 the possible domain of X for the left tree is the same as before the atom was evaluated. However, the possible domain of X for the right subtree is reduced to $\emptyset$. Since the domain of X for the right subtree of this node is empty this subtree is marked as an *intron subtree* (see Figure 5.4).

After all possible *intron subtrees* in a *top-down atomic* decision tree have been detected and marked the tree can be pruned. During the pruning phase the marked *intron subtrees* are removed and their originating root node ($X < 2$ in our example) is replaced by the remaining valid subtree. The resulting subtree for the example can be seen in Figure 5.5.

After the *intron subtrees* in a Boolean *top-down atomic* decision tree have been pruned a smaller, but semantically the same, *top-down atomic* tree remains. By using these smaller trees in our fitness cache rather than the

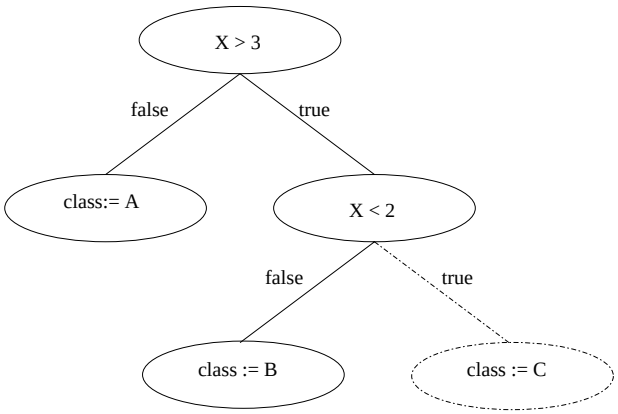

Figure 5.4: A *top-down atomic* decision tree containing a marked intron subtree.

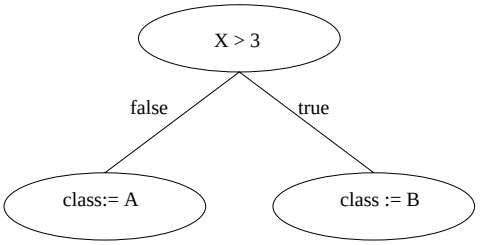

Figure 5.5: A pruned *top-down atomic* decision tree.

original trees the number of cache hits should increase resulting in shorter computation times. The smaller trees should also easier be to understand. Note that since several syntactically different trees can be semantically the same the semantic search space of our Boolean *top-down atomic* representations is smaller than the syntactic search space.

Note that we assume that all attributes in our data sets are independent. In real life there may be some relation between attributes (e.g., attribute X is always greater than attribute Y) in which case *intron subtrees* may go undetected.

Intron Subtrees in Fuzzy Representations

The detection of *intron subtrees* as described above only works for Boolean *top-down atomic* trees. In the case of our *fuzzy* GP algorithms introduced in Chapter 4 there are no *intron subtrees* as a result of the manner in which fuzzy decision trees are evaluated. Consider the two fuzzy *top-down atomic* trees in Figure 5.6. The trees are similar to the Boolean trees in Figures 5.2 and 5.5. However, since the trees in Figure 5.6 are fuzzy trees they are not semantically the same.

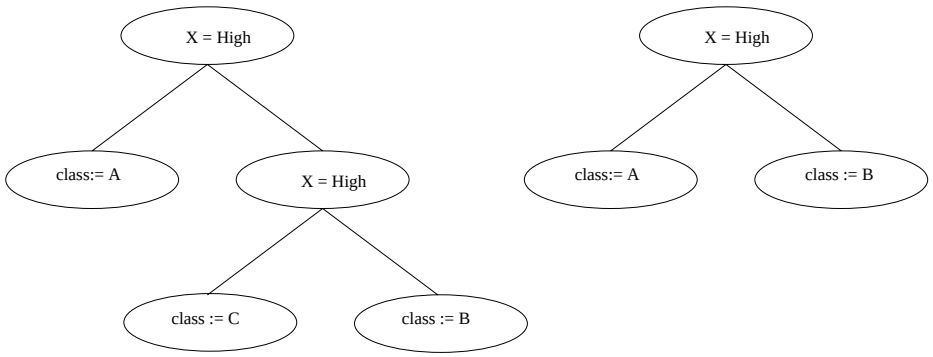

Figure 5.6: Two fuzzy *top-down atomic* trees which are syntactically and semantically different.

The fuzzy decision tree on the left contains two internal nodes containing the same atom ($X = \text{High}$). If we transform the left fuzzy decision tree into fuzzy membership functions for each target class using fuzzy logic we get:

- $\mu_A(X) = 1 - \mu_{\text{High}}(X)$
- $\mu_B(X) = \mu_{\text{High}}(X) \times \mu_{\text{High}}(X)$
- $\mu_C(X) = \mu_{\text{High}}(X) \times (1 - \mu_{\text{High}}(X))$

However, in the case of the fuzzy decision tree on the right the fuzzy membership functions for the possible target classes are:

- $\mu_A(X) = 1 - \mu_{\text{High}}(X)$
- $\mu_B(X) = \mu_{\text{High}}(X)$

Not only will the right decision tree never return a fuzzy membership value for class C other than 0, but the fuzzy membership function for class B also differs. Thus, in the case of a fuzzy *top-down atomic* tree all subtrees influence the classification outcome of the tree and *intron subtrees* cannot occur. This also means that, given the same number of internal and leaf nodes, the semantic search space for *fuzzy top-down atomic* decision trees is larger than that of Boolean *top-down atomic* trees although the syntactic search spaces are the same.

5.3.2 Intron Nodes

Intron nodes are root nodes of subtrees of which each leaf node contains the same *class assignment* atom (e.g., *class* := A). Regardless of the path traversed through such trees, as a result of the tests in the root node or internal nodes, the resulting target class will always be the same (e.g., **A**). Since the internal nodes in such a subtree do not influence the classification outcome and can be replaced by any other possible internal node without affecting the classification outcome, they are called *intron nodes*. The negative effects of *intron subtrees* are mostly related to the size and thus understandability of our decision trees. However, *intron nodes* also have a negative influence on the computation time needed to evaluate a tree since *intron nodes* are evaluated but do not contribute to the classification outcome of a tree.

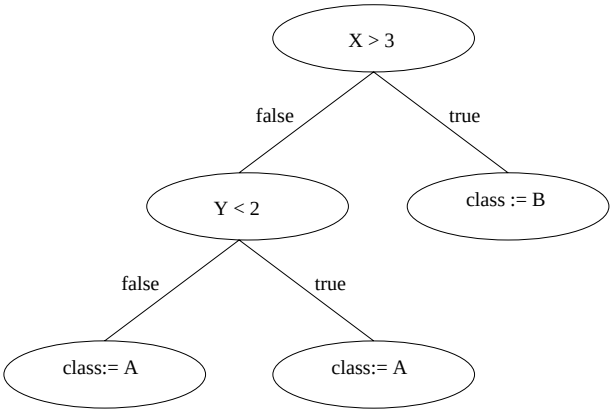

Figure 5.7: A *top-down atomic* tree containing an *intron node* ($Y < 2$) since both its children result in the same class.

Consider the *top-down atomic* tree displayed in Figure 5.7. Regardless of the outcome the second internal node (Y < 2) the right subtree will always result in a class assignment node for class A. In this case the second internal node of the tree is an *intron node.*

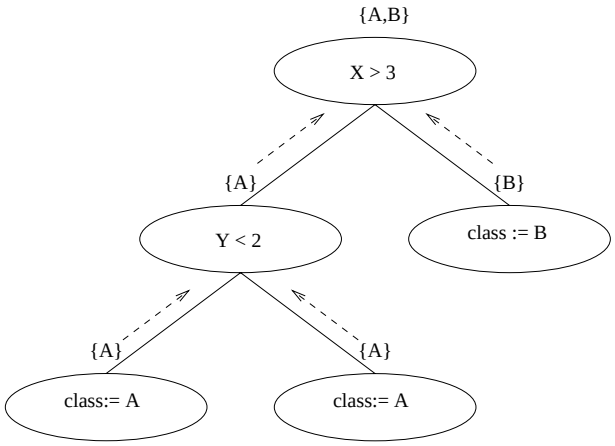

Figure 5.8: A *top-down atomic* tree with the set of possible target classes for each node.

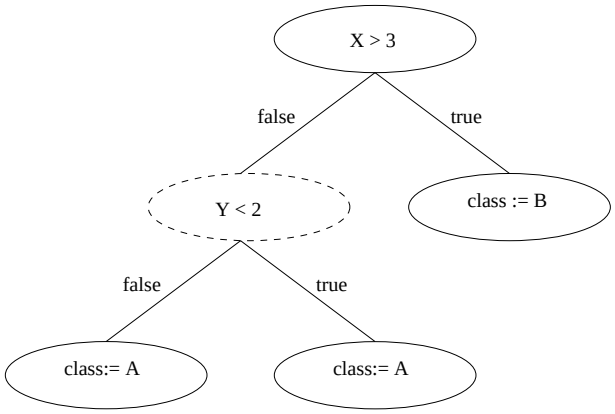

Figure 5.9: A *top-down atomic* tree containing a marked (dotted) *intron node.*

In order to detect all *intron nodes* in a *top-down atomic* decision tree we recursively propagate the set of possible class outcomes through the tree in a bottom-up manner. A class assignment node always returns a set containing a single class. In each internal node the sets of possible classes of its children are joined. If the set of possible class outcomes for an internal node contains

only a single class the node is marked as an *intron node*. Once all the *intron nodes* have been detected the tree can be pruned. During the pruning phase the tree is traversed in a top-down manner and subtrees with an *intron node* as the root node are replaced by class assignment nodes corresponding to their possible class outcome detected earlier.

Observe the *top-down atomic* tree in Figure 5.8. The set of possible class outcomes for each leaf node consists of a single class, namely the target class. In the case of the second internal node ($Y < 2$), the set of possible class outcomes depends on the sets of possible class outcomes of its subtrees. Since the sets of possible class outcomes for both subtrees are the same and contain only a single class outcome, the set of possible class outcomes for the second internal node also contains only a single value (**A**) and it is therefore marked as an *intron node* (see Figure 5.9). In the case of the root node ($X > 3$) the set of possible class outcomes consists of two classes and this node is therefore not an *intron node*.

After all the *intron nodes* have been detected they can be pruned. The resulting tree for the example can be seen in Figure 5.10.

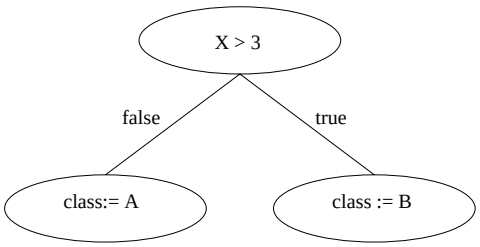

Figure 5.10: A pruned *top-down atomic* tree

Note that the pruned decision trees in Figures 5.10 and 5.5 are both semantically and syntactically the same although they are derived from syntactically different trees (Figures 5.7 and 5.2). When *intron node* detection and pruning is used in conjunction with *intron subtree* detection it is important to apply both detection and pruning strategies in the right order. *Intron nodes* should be detected and pruned after *intron subtrees* to assure that all *introns* are found as the pruning of *intron subtrees* can influence the detection of *intron nodes* (see Figure 5.11). Note that in the original (left) tree no *intron nodes* would be detected before *intron subtree* detection and pruning. However, after *intron subtree* detection an *intron node* is detected (see the center tree) which can be pruned (right tree).

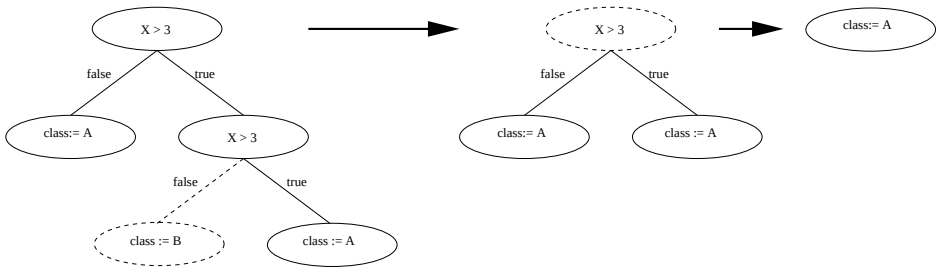

Figure 5.11: Example of the intron detection and pruning process. Detected introns are shown with dotted lines.

5.3.3 The Effect of Intron Nodes on the Search Space

In Section 2.5 we demonstrated how Lemma 2.5.2 can be used to calculate the number of syntactically different *top-down atomic* decision trees given a specific *top-down atomic* representation, maximum tree size and data set. However, as demonstrated above, several syntactically different trees can have the same semantic behaviour. Using discrete mathematics we can calculate the number of syntactically different *top-down atomic* decision trees without *intron nodes* (but possibly still containing *intron subtrees*). A *top-down atomic* tree contains one or more *intron nodes* if one of the internal nodes in the tree has two leaf nodes as children and both leaf nodes contain the same class assignment.

Depending on the shape of a *top-down atomic* decision tree there are between 1 and $\lceil n/2 \rceil$ nodes with two leaf nodes as children, where n is the total number of internal nodes. If a node has two leaf nodes as children then in $\frac{1}{|T|}$th of the cases the two leaf nodes will contain the same class assignment node, where T is the set of possible terminal nodes.

In Lemma 2.5.2 the total number of syntactically different *top-down atomic* decision trees with at most N nodes (N odd) is given as:

$$\sum_{n=1}^{\frac{N-1}{2}} \frac{1}{n+1} \binom{2n}{n} \times |I|^n \times |T|^{n+1}. \tag{5.1}$$

Here I is the set of possible internal nodes.

Let $M(n, i)$ denote the number of *top-down atomic* decision trees with n internal nodes of which i nodes have two children which are leaf nodes. The total number of syntactically different top-down atomic trees not containing

intron nodes with at most N nodes (N odd), a set of internal nodes I and a set of terminal nodes T then becomes:

$$\sum_{n=1}^{\frac{N-1}{2}} \sum_{i=1}^{\lceil n/2 \rceil} M(n,i) \times (1 - \frac{1}{|T|})^i \times |I|^n \times |T|^{n+1}. \tag{5.2}$$

We know from Lemma 2.5.1 that the number of full binary trees with n internal nodes equals the number of binary trees with n nodes. Thus, the number of full binary trees with n internal nodes of which i nodes have two leaf nodes as children is the same as the number of binary trees with n nodes with i leaf nodes (with no children). Therefore, $M(n,i)$ equals the number of binary trees with n nodes in which i nodes have no children. To determine $M(n,i)$ we will use the following lemma from combinatorics [49, Chapter 21].

Theorem 5.3.1 *The number of k-tuples of ordered trees in which a total of n_j nodes have j children equals:*

$$\frac{k}{n}\binom{n}{n_0, n_1, n_2, \ldots}, \tag{5.3}$$

where $n = \sum_j n_j$, if $\sum_j j n_j = n - k$, and 0 otherwise.

Thus the total number of ordered *binary* trees with n nodes in which a total of n_j nodes have j children is:

$$\frac{1}{n}\binom{n}{n_0, n_1, n_2}, \tag{5.4}$$

where $n = \sum_j n_j$, if $\sum_j j n_j = n - k$, and 0 otherwise.

This can be rewritten as:

$$Cat(n_0 - 1)\binom{n-1}{2(n_0-1)}, \tag{5.5}$$

where $n_1 = n - 2n_0 + 1$ and $n_2 = n_0 - 1$.

However, the subtree of all internal nodes of a *top-down atomic* decision tree is not an ordered tree. In normal (positional) binary trees, if a node has only one child that node is either the left or the right child. In ordered binary trees there is no distinction between left and right child nodes. Thus the number of binary trees with n nodes is:

$$O_n \times 2^{n_1},$$

where O_n is the number of ordered binary trees with n nodes of which n_1 nodes have one child.

Thus the number of binary trees in which a total of i $(= n_0)$ nodes have no children is:

$$Cat(i-1) \times \binom{n-1}{2(i-1)} \times 2^{n-2i+1} = M(n,i). \tag{5.6}$$

This means that:

Lemma 5.3.2 *The total number of syntactically different top-down atomic trees not containing intron nodes with at most N nodes (N odd), a set of internal nodes I and a set of terminal nodes T is:*

$$\sum_{n=1}^{\frac{N-1}{2}} \sum_{i=1}^{\lceil n/2 \rceil} Cat(i-1) \times \binom{n-1}{2(i-1)} \times 2^{n-2i+1} \times (1 - \frac{1}{|T|})^i \times |I|^n \times |T|^{n+1}. \tag{5.7}$$

Note that since our *fuzzy* decision tree representations cannot contain *intron subtrees* Lemma 5.3.2 can be used to calculate the semantic search space size of a *fuzzy* GP algorithm on a specific data set. Since our non-fuzzy GP algorithms can also contain *intron subtrees* their semantic search spaces sizes can be smaller.

Example 5.3.1 To demonstrate the influence of *intron nodes* on the syntactic and semantic search space sizes of our *fuzzy* GP algorithms we give both the syntactic search space sizes (including *introns*) according to Lemma 2.5.2 as well as the semantic search space sizes (without *intron nodes*) according to Lemma 5.3.2 on the Australian Credit data set. The average, minimum and maximum number of possible internal nodes are given in Table 3.3.

When we look at Table 5.1 we see that on the Australian Credit data set the syntactic search spaces of our *fuzzy* GP algorithms are around 170 times larger than the semantic search spaces. On the Iris data set we found the semantic search space size to be approximately 23 times smaller than the syntactic search space size.

Table 5.1: The approximate syntactic and semantic search sizes for our *fuzzy* GP algorithms on the Australian Credit data set.

algorithm	k	**syntactic search space size**	**semantic search space size**
fuzzy clustering GP	2	4.6×10^{70}	2.6×10^{68}
fuzzy clustering GP	3	5.9×10^{74}	3.4×10^{72}
fuzzy clustering GP	4	1.1×10^{77}	1.3×10^{75}
fuzzy clustering GP	5	3.2×10^{79}	1.8×10^{77}
fuzzy refined GP (*gain*)	2	4.6×10^{70}	2.6×10^{68}
fuzzy refined GP (*gain*)	3	5.9×10^{74}	3.4×10^{72}
fuzzy refined GP (*gain*)	4	1.1×10^{77}	1.3×10^{75}
fuzzy refined GP (*gain*)	5	3.2×10^{79}	1.8×10^{77}
fuzzy refined GP (*gain_ratio*)	2	4.7×10^{70}	92.6×10^{68}
fuzzy refined GP (*gain_ratio*)	3	$1.4 \times 10^{71} \ldots 3.9 \times 10^{71}$	$7.7 \times 10^{68} \ldots 2.2 \times 10^{69}$
fuzzy refined GP (*gain_ratio*)	4	$3.9 \times 10^{71} \ldots 7.5 \times 10^{72}$	$2.2 \times 10^{69} \ldots 4.2 \times 10^{70}$
fuzzy refined GP (*gain_ratio*)	5	$1.1 \times 10^{72} \ldots 1.1 \times 10^{74}$	$6.1 \times 10^{69} \ldots 6.3 \times 10^{71}$

5.4 Experiments and Results

In order to determine the effects of *introns* on our GP algorithms we have repeated the experiments of the previous chapters for two of our GP classifiers using various combinations of intron detection and pruning. For the algorithms we have chosen the *simple* GP algorithm, since it has the largest (syntactic) search space on each of the data sets, and our *clustering* GP algorithm with 2 clusters for each numerical valued attribute, which has the smallest search space size for each of the data sets.

On each data set we have applied the two algorithms without intron detection, with *intron node* detection, with *intron subtree* detection and with *intron subtree* detection followed by *intron node* detection. The results of the experiments are split into two parts. First, we will report on the effect of intron detection and pruning on the size of our *top-down atomic* decision trees. In the second part we show how intron detection and pruning increases the effectiveness of our fitness cache.

5.4.1 Tree Sizes

To determine the effect of intron detection and pruning we have measured the average size of all trees after pruning during an evolutionary run. We also measured the size of trees found to be the best, based on the classification performance on the training set, at the end of an evolutionary run. The results are given in Tables 5.2, 5.3, 5.4, 5.5, 5.6 and 5.7. The average (avg), minimum (min) and maximum (max) sizes are computed over a 100 runs (10 random seeds times 10 folds). Note that when no intron detection and pruning strategy is used the pruned tree size equals the original non-pruned tree size.

Australian Credit

Table 5.2: The pruned tree sizes of the best and average trees for the *simple* GP and *clustering* GP algorithms on the Australian Credit data set.

algorithm	**intron detection**	**pruned tree size**					
		best tree			**average tree**		
	scheme	**avg**	**min**	**max**	**avg**	**min**	**max**
simple GP	none	43.9	3	63	33.4	16.0	43.6
simple GP	nodes	40.4	3	63	28.2	13.0	37.6
simple GP	subtrees	34.8	3	55	22.4	9.8	32.8
simple GP	both	33.1	3	55	19.4	8.1	28.9
clustering GP	none	39.2	7	63	30.8	12.5	40.4
clustering GP	nodes	36.9	7	61	26.3	10.2	34.6
clustering GP	subtrees	31.7	7	53	20.3	7.6	28.6
clustering GP	both	30.8	7	49	17.8	6.2	25.4

The tree size results for the Australian Credit data set are displayed in Table 5.2. As can be seen there is virtually no effect of the representation (*simple* or *cluster*) on the sizes of the evolved decision trees. There is also no visible relation between the size of the (syntactic) search space and the occurrence of *introns*. If we look at the effect of the different intron detection and pruning strategies we see that detecting and pruning *intron subtrees* has the largest effect on the size of the trees. As expected, the combination of

detecting both *intron subtrees* and *intron nodes* reduces the size of the trees the most. When we compare the tree sizes of the best trees with the average trees we see that the average tree is generally a little smaller but contains relatively more *introns*. The detection of *intron nodes* and *intron subtrees* reduces the average size of the best found trees by approximately 25% while the average size of all (evolved) trees is reduced by approximately 40%.

German Credit

On the German Credit data set (see Table 5.3) both algorithms evolve larger trees on average. Whereas on the Australian Credit data set the *clustering* GP algorithm evolved smaller trees on average, here it is the *simple* GP algorithm. The difference in size is again very small and indicates that the number of possible internal nodes has no real influence on the size of evolved decision trees. If we look at the effect of our intron detection and pruning methods on the average and best tree sizes we observe the same effects as on the Australian Credit data set.

Table 5.3: The pruned tree sizes of the best and average trees for the *simple* GP and *clustering* GP algorithms on the German Credit data set.

algorithm	**intron detection scheme**	**pruned tree size**					
		best tree			**average tree**		
		avg	**min**	**max**	**avg**	**min**	**max**
simple GP	none	50.0	23	63	36.1	29.8	44.3
simple GP	nodes	46.4	23	63	30.9	25.3	37.9
simple GP	subtrees	37.8	19	51	22.2	16.1	32.4
simple GP	both	36.3	19	47	19.7	14.2	28.6
clustering GP	none	53.1	35	63	36.3	29.9	44.3
clustering GP	nodes	49.1	35	61	30.8	24.7	37.4
clustering GP	subtrees	44.4	31	59	27.0	19.1	38.2
clustering GP	both	41.9	31	53	23.4	17.2	32.5

Pima Indians Diabetes

When we look at the results on the Pima Indians diabetes data in Table 5.4 we see that the differences in best found tree size between the *simple* GP algorithm and *clustering* GP algorithm are a bit larger than on the Credit data sets above. However, if we look at the average, minimum and maximum sizes for the average tree we see that the results are virtually the same.

Table 5.4: The pruned tree sizes of the best and average trees for the *simple* GP and *clustering* GP algorithms on the Pima Indians Diabetes data set.

algorithm	**intron detection scheme**	**pruned tree size**					
		best tree			**average tree**		
		avg	**min**	**max**	**avg**	**min**	**max**
simple GP	none	53.0	29	63	36.4	24.0	44.5
simple GP	nodes	49.2	27	61	31.1	20.2	38.8
simple GP	subtrees	40.8	23	51	24.1	14.3	33.0
simple GP	both	39.3	23	49	21.2	12.5	28.9
clustering GP	none	44.1	25	63	36.4	23.4	42.6
clustering GP	nodes	42.0	23	63	31.4	19.7	37.6
clustering GP	subtrees	34.5	23	45	22.2	13.8	28.1
clustering GP	both	33.9	23	45	20.1	12.0	25.3

Heart Disease

On the Heart Disease data set (see Table 5.5) we see the same results as on the previous data sets. There is virtually no difference in tree sizes between the *simple* GP and *clustering* GP algorithms, and the detection and pruning of *intron subtrees* leads to a larger reduction in tree size than the pruning (and detection) of *intron nodes*.

Table 5.5: The pruned tree sizes of the best and average trees for the *simple* GP and *clustering* GP algorithms on the Heart Disease data set.

algorithm	**intron detection scheme**	**pruned tree size**					
		best tree			**average tree**		
		avg	**min**	**max**	**avg**	**min**	**max**
simple GP	none	47.8	27	63	36.5	27.9	44.2
simple GP	nodes	45.1	27	61	31.2	23.4	37.4
simple GP	subtrees	38.5	23	53	24.8	17.3	33.7
simple GP	both	37.1	23	49	21.9	15.1	29.4
clustering GP	none	47.9	23	63	38.7	28.4	44.1
clustering GP	nodes	45.7	23	59	33.2	24.1	37.7
clustering GP	subtrees	39.6	23	53	26.5	17.0	34.5
clustering GP	both	38.4	23	51	23.4	14.7	30.8

Ionosphere

The search spaces for our GP algorithms are the largest for the Ionosphere data set because of the large number of real-valued attributes and the number of unique values per attribute. This seems to have a small effect on the sizes of both the best trees and average trees as given in Table 5.6. The difference between detecting and pruning both types of *introns* and no intron detection is less than on the previous data sets for both the best and average trees.

Iris

The Iris data set resulted in the smallest search space sizes since it only has 4 attributes and 150 records. Because of the small number of possible internal nodes (8) used by the *clustering* GP algorithm we see a relatively large difference in average, minimum and maximum tree sizes for both the best and average trees compared with the *simple* GP algorithm. However, both algorithms evolve trees which are a lot smaller than on the previous data sets, and the relative effects of the *intron* detection and pruning strategies on the sizes of the average trees is about the same for both algorithms. The fact that all the trees found to be the best by our *clustering* GP algorithm, regardless of fold or random seed, consisted of exactly 5 nodes might be caused by the use of the tree size in the multi-layered fitness meassure.

Table 5.6: The pruned tree sizes of the best and average trees for the *simple* GP and *clustering* GP algorithms on the Ionosphere data set.

algorithm	**intron detection scheme**	**pruned tree size**					
		best tree			**average tree**		
		avg	**min**	**max**	**avg**	**min**	**max**
simple GP	none	48.6	23	63	39.0	24.8	45.3
simple GP	nodes	45.1	21	61	33.0	20.9	38.8
simple GP	subtrees	43.2	15	61	32.1	17.8	42.3
simple GP	both	40.5	13	59	27.3	15.1	35.3
clustering GP	none	44.4	15	61	34.2	21.0	44.3
clustering GP	nodes	41.5	15	57	29.1	17.4	38.1
clustering GP	subtrees	37.9	15	55	25.6	14.5	38.9
clustering GP	both	36.2	15	49	22.1	12.2	33.5

Table 5.7: The pruned tree sizes of the best and average trees for the *simple* GP and *clustering* GP algorithms on the Iris data set.

algorithm	**intron detection scheme**	**pruned tree size**					
		best tree			**average tree**		
		avg	**min**	**max**	**avg**	**min**	**max**
simple GP	none	12.9	5	35	20.7	12.4	30.0
simple GP	nodes	12.6	5	33	18.4	10.7	27.2
simple GP	subtrees	11.7	5	23	11.5	6.8	18.3
simple GP	both	11.6	5	21	10.3	5.9	16.9
clustering GP	none	5.0	5	5	9.8	9.0	12.2
clustering GP	nodes	5.0	5	5	8.6	7.9	10.9
clustering GP	subtrees	5.0	5	5	5.2	4.9	6.8
clustering GP	both	5.0	5	5	4.6	4.3	6.1

5.4.2 Fitness Cache

In Section 2.9 we introduced a fitness cache to reduce the time spent on fitness evaluations by storing the fitness of each (syntactically) unique individual. We can improve the effectiveness of this cache by storing the pruned

trees with their fitness rather than the original trees, since several syntactically different trees can be "mapped" to a single pruned tree. The results of the fitness experiments cache are displayed in Tables 5.8, 5.9, 5.10, 5.11, 5.12 and 5.13. The results are computed over a 100 runs (10 random seeds times 10 folds). For each algorithm and intron detection and pruning variant the average (avg), minimum (min) and maximum (max) percentage of cache hits is reported. Note that when no intron detection and pruning is used the number of cache hits equals the resampling ratio's (as reported in Section 2.9). We also report the runtime for each algorithm relative to the runtime of the algorithm without *intron* detection and pruning.

Australian Credit

When we look at the cache hit percentages of both algorithms it is clear that detecting and pruning *intron subtrees* results in a larger increase in cache hits than detecting and pruning *intron nodes*. As expected the combination of detecting and pruning both *intron subtrees* and *intron nodes* offers the highest number of cache hits (around 50% on average). Since the difference between detecting and pruning both types of *introns* on the one hand and detecting and pruning only *intron subtrees* on the other hand is larger than the difference between no intron detection and detecting and pruning *intron nodes* it is clear that first removing the *intron subtrees* allows for a better detection of *intron nodes*.

Looking at the relative runtimes of our GP algorithms in combination with the different *intron* detection and pruning strategies we see that the detection and pruning of *intron subtrees* is more expensive than detecting and pruning *intron nodes*. As expected we can see a relationship between the number of cache hits and the relative runtimes. Unfortunately, the relative increase in cache hits is much larger than the relative decrease in computation times. This difference can partially be explained by the time spent by our GP algorithms on initialization (e.g., clustering), and other parts of the evolutionary process (e.g., crossover, mutation and selection).

German Credit

On the German Credit data set the influence of our intron detection and pruning strategies on the fitness cache is similar to the results on the Australian Credit data set, except that our *clustering* GP algorithm does not

Table 5.8: The number of cache hits for the *simple* GP and *clustering* GP algorithms on the Australian Credit data set.

algorithm	**intron detection**	**% cache hits**			**relative runtime**
		avg	**min**	**max**	
simple GP	none	16.9	10.3	31.7	1.0
simple GP	nodes	21.5	12.9	41.0	0.9
simple GP	subtrees	39.0	24.7	53.8	1.0
simple GP	both	46.8	28.9	64.7	0.9
clustering GP	none	20.4	12.0	41.2	1.0
clustering GP	nodes	24.6	14.4	49.9	0.9
clustering GP	subtrees	47.3	30.0	68.7	0.9
clustering GP	both	53.3	37.6	75.8	0.8

benefit as much as it did on the Australian Credit data set. This reduced effectiveness of the fitness cache can partially be explained if we look at the pruned tree sizes. For our *clustering* GP algorithm the average pruned tree contains three nodes more after both *intron subtrees* and *intron nodes* have been detected and pruned than the average pruned tree in our *simple* GP algorithm.

Table 5.9: The number of cache hits for the *simple* GP and *clustering* GP algorithms on the German Credit data set.

algorithm	**intron detection**	**% cache hits**			**relative runtime**
		avg	**min**	**max**	
simple GP	none	15.4	9.5	24.7	1.0
simple GP	nodes	19.0	11.8	29.2	0.9
simple GP	subtrees	44.1	25.4	56.9	0.7
simple GP	both	51.1	30.1	65.2	0.6
clustering GP	none	19.1	10.8	30.2	1.0
clustering GP	nodes	22.6	12.9	33.5	0.9
clustering GP	subtrees	34.2	18.3	46.9	0.8
clustering GP	both	39.9	22.6	53.9	0.8

If we look at the relative runtimes of our algorithms on the German Credit data set we see again a relation with the number of cache hits. The detecting and pruning of both types of *introns* more than triples the number of cache hits for our *simple* GP algorithm, resulting in a decrease in computation time of more than 37%. The same strategy doubles the number of cache hits for our *clustering* GP algorithm, decreasing the computation time by 15%. Compared to the results on the Australian Credit data set above we see that an increase in cache hits does have a larger impact on the runtimes.

Pima Indians Diabetes

The results of the Pima Indians Diabetes data set are similar to the results on the Australian Credit data set. The average number of cache hits increases by at least a factor of 3 when both *intron subtrees* and *intron nodes* are detected and pruned.

If we look at the relative runtimes we see that detecting and pruning *intron subtrees* applied to our *simple* GP algorithm actually increases the computation time by almost 10%. For our *clustering* GP algorithm detecting and pruning *intron subtrees* does decrease the computation time probably because the number of cache hits is much larger.

Table 5.10: The number of cache hits for the *simple* GP and *clustering* GP algorithms on the Pima Indians Diabetes data set.

algorithm	**intron detection**	**% cache hits**			**relative runtime**
		avg	**min**	**max**	
simple GP	none	15.2	9.0	24.7	1.0
simple GP	nodes	19.1	11.7	29.4	0.9
simple GP	subtrees	38.6	23.3	57.9	1.1
simple GP	both	45.7	27.6	65.9	1.0
clustering GP	none	15.0	11.0	27.2	1.0
clustering GP	nodes	18.0	12.9	30.7	0.9
clustering GP	subtrees	52.3	41.9	68.5	0.9
clustering GP	both	59.0	47.9	73.9	0.8

Heart Disease

Although on the Heart Disease data set the percentage of cache hits is generally lower than on the data sets above, the effect of *intron* detection and pruning is similar. The combination of *intron subtree* and *intron node* detection and pruning triples the number of cache hits and reduces the pruned tree sizes by 40%.

The detection and pruning of *introns* does not result in a large decrease in computation times but on the Heart Disease data set this was also not to be expected because of the small number of records (270).

Table 5.11: The number of cache hits for the *simple* GP and *clustering* GP algorithms on the Heart Disease data set.

algorithm	**intron detection**	**% cache hits**			**relative runtime**
		avg	**min**	**max**	
simple GP	none	13.7	8.9	22.9	1.0
simple GP	nodes	17.3	11.2	27.8	0.9
simple GP	subtrees	35.3	22.4	49.7	1.0
simple GP	both	42.1	26.4	58.5	0.9
clustering GP	none	13.4	9.7	23.3	1.0
clustering GP	nodes	16.1	11.5	26.1	1.0
clustering GP	subtrees	38.0	23.4	54.1	0.9
clustering GP	both	44.6	27.7	61.9	0.8

Ionosphere

On the Ionosphere data set the search spaces for our GP algorithms are the largest because of the large number of real-valued attributes. As a result the increase in the percentage of cache hits doubles rather than triples as it did on most of the data sets above.

The large number of attributes combined with the small number of records has a negative effect on the effects of *intron* detection and pruning. Only our *clustering* GP algorithm combined with *intron node* detection and pruning manages to decrease the computation time. In all other cases the computation time increases by up to 50%.

Table 5.12: The number of cache hits for the *simple* GP and *clustering* GP algorithms on the Ionosphere data set.

algorithm	**intron detection**	**% cache hits**			**relative runtime**
		avg	**min**	**max**	
simple GP	none	12.4	8.8	21.6	1.0
simple GP	nodes	16.1	11.4	28.7	1.0
simple GP	subtrees	22.7	12.0	45.9	1.5
simple GP	both	28.1	15.2	54.6	1.4
clustering GP	none	17.8	9.6	28.8	1.0
clustering GP	nodes	21.8	11.5	35.7	0.9
clustering GP	subtrees	33.9	15.5	52.4	1.3
clustering GP	both	40.2	18.7	61.4	1.2

Iris

Table 5.13: The number of cache hits for the *simple* GP and *clustering* GP algorithms on the Iris data set.

algorithm	**intron detection**	**% cache hits**			**relative runtime**
		avg	**min**	**max**	
simple GP	none	18.4	9.1	30.8	1.0
simple GP	nodes	21.5	10.4	36.4	1.0
simple GP	subtrees	53.0	33.2	65.0	0.7
simple GP	both	58.0	36.1	70.4	0.7
clustering GP	none	46.8	38.7	50.4	1.0
clustering GP	nodes	50.5	42.4	54.1	0.9
clustering GP	subtrees	82.2	76.0	84.9	0.5
clustering GP	both	84.7	79.6	87.0	0.5

On the Iris data set we see very high cache hit rates for the *clustering* GP algorithm which is probably caused by the small number of possible internal nodes (in this case only 8). The cache hit rates for our *simple* GP algorithm are also a little higher than on the previous data sets. However, even with-

out intron detection and pruning our *clustering* GP algorithm with only two clusters per attribute has a cache hit percentage (also known as resampling ratio) which is almost three times as high as that of our *simple* GP algorithm.

If we look at the relative runtimes we see that especially *intron subtree* detection and pruning greatly reduces the computation times for our algorithms. The relatively large decrease in computation time is clearly caused by the large number of cache hits despite the small number of records of the Iris data set.

5.5 Conclusions

Looking at the results in Section 5.4 it is clear that the detection and pruning of *introns* in our decision trees reduces the effective size of the trees. As a result the decision trees found should be easier to understand although in some cases they can still be quite large. The detection and pruning of *intron nodes* and *intron subtrees* also enables us to identify syntactically different trees which are semantically the same. By comparing and storing pruned decision trees in our fitness cache, rather than the original unpruned decision trees, we can improve its effectiveness. The increase in cache hits means that less individuals have to be evaluated resulting in reduced computation times. As a consequence our algorithms will probably scale better with larger data sets.

If we compare the runtimes of the algorithms we note that the combination of both intron detection and pruning methods has a noticeable effect on the computation times. The decrease in computation time is different from what we would expect when looking at the increase in cache hits and the reduction in tree sizes achieved by our detection and pruning strategies. This difference can be partially explained by taking into account the time spent by our algorithms on detecting and pruning the *introns*, mutation, crossover, selection and initialization. The main reason for the small difference in runtimes compared to the large increase in cache hits is probably due to certain (unfortunate) choices made in the implementation of the algorithms.

Nevertheless, on the smallest data set (Iris) detecting and pruning both *intron subtrees* and *intron nodes* reduces the computation times of our *clustering* GP and *simple* GP algorithms by approximately 50% and 30%, respectively. On the German Credit data set, which contains the most records, the detection and pruning of both types of *introns* reduces the average compu-

tation time by over 37% for our *simple* GP and 15% for our *clustering* GP algorithm. However, the computation time increases on the Pima Indians and Ionosphere data sets show that *intron* detection and pruning does not always work as expected. This can partially be explained by the number and nature of the attributes relative to number of records in those data sets.

Based on the results in Section 5.4 we think that the combination of a fitness cache combined with intron detection and pruning strategies can potentially substantially reduce the runtimes of variable-length evolutionary algorithms such as our tree-based GP, especially in cases where relatively much computation time is spent on fitness evaluations. Another important factor is the number of different types of *introns* occurring and the computation time needed to detect the different types of *introns*.

6 Stepwise Adaptation of Weights

The Stepwise Adaptation of Weights (SAW) technique has been successfully used in solving constraint satisfaction problems with evolutionary computation. It adapts the fitness function during the evolutionary algorithm in order to escape local optima and hence to improve the quality of the evolved solutions. We will show how the SAW mechanism can be applied to both data classification and symbolic regression problems using Genetic Programming. Moreover, we investigate the influence of the SAW parameters on the results.

6.1 Introduction

In the previous chapters we looked at decision tree representations and their effect on the classification performance in Genetic Programming. In this chapter we focus our attention on another important part of our GP algorithms: the fitness function. Most evolutionary algorithms use a static fitness measure $f(x)$ which given an individual x always returns the same fitness value. Here we investigate an adaptive fitness measure which should allow our GP algorithms to escape local optima and improve classification performance.

Previous studies [2, 30, 31, 32, 33, 52] on constraint satisfaction problems show that the Stepwise Adaptation of Weights (SAW) technique can be used to boost the performance of evolutionary algorithms (EAs). The SAW method gathers information about a problem during the run of an evolutionary algorithm. At certain moments during the run it uses this information to identify the parts of a problem that are the hardest to solve and to adjust the fitness function accordingly. By focussing on different parts of a problem at differ-

ent times during an evolutionary run the EA should be able to escape local optima and potentially improve the quality of the evolved solutions.

In this chapter we identify the underlying principles of the SAW method and demonstrate how they can be applied to data classification and symbolic regression problems. We start by describing the general Stepwise Adaptation of Weights technique and discuss its underlying ideas in Section 6.2. Apart from showing the general applicability of the Stepwise Adaptation of Weights technique to different data mining tasks we investigate influence and robustness of the SAW parameters.

The SAW method shares some characteristics with the Dynamic Subset Selection (DSS) [44, 45] and limited-error fitness (LEF) [47] approaches of Gathercole and Ross. The main goal of Dynamic Subset Selection is to reduce the number of records needed for fitness evaluations, thereby decreasing computation time. In every generation the DSS method randomly selects a number of cases from the whole training set using a bias, so that cases which are "difficult" to classify or have not been selected for several generations are more likely to be selected. The LEF method uses an error limit to determine on how many training set examples an individual is evaluated. Once this error limit is exceeded all remaining training set examples are counted as misclassified. This way only "good" individuals are evaluated on the entire training set. After each generation the error limit is altered and the training set is re-ordered depending on the performance of the best individual in the preceding generation.

Stepwise Adaptation of Weights also has some similarities with (competitive) artificial co-evolution. Research conducted by for instance Hillis [54] and Paredis [81, 82] incorporates predator-prey interactions to evolve solutions for static problems. In competitive co-evolution a second population is introduced which contains a subset of the test cases (problems). These problems may evolve in the same fashion as the solutions. As a result their algorithms create an arms-race between both populations, i.e., the population of problems evolves and tries to outsmart the population of solutions.

Boosting [41, 42, 91] is also a technique which assigns weights to examples in the training set based on their difficulty, similar to SAW. Alternatively, only a subset of training set examples is used, similar to DSS, to train a weak classifier. Boosting differs from both SAW and DSS in that it combines several weak classifiers using a weighted majority vote into a final hypothesis.

In Section 6.3 we start by showing how SAW can be applied to symbolic regression. Moreover, we show how we can use the real-valued results of our

regression trees to make SAW more "precise". We compare the different SAW strategies on two functions from literature as well as randomly generated polynomials.

Then in Section 6.4 we investigate the usefulness of the SAW technique when applied to some of our GP classifiers from the previous chapters. Apart from determining the influence SAW has on the (mis)classification performance of our algorithms we also investigate the influence of one of the SAW parameters. Finally, general conclusions will be drawn in Section 6.5.

6.2 The Method

The Stepwise Adaptation of Weights method is a technique which adapts the fitness function during an evolutionary run with the purpose of escaping local optima and boost search efficiency and effectiveness. It was first introduced by Eiben et al. [34] and is designed to aid evolutionary algorithms with solving constraint satisfaction problems (CSPs) [68, 90]. A contraint satisfaction problem is defined by a finite fixed set of variables V, each variable $v_i \in V$ with a finite fixed size domain D_i, and a finite fixed set of constraints C. Each constraint $c_j \in C$ specifies for a subset of variables the allowable combinations of values for that subset. The goal is to find instantiations for all variables v_i such that none of the contraints $c_j \in C$ is violated.

In evolutionary computation constraint satisfaction problems can be approached using penalty functions. Each penalty function checks whether a contraint has been violated and returns the appropriate penalty in the case of a constraint violation. The fitness of an arbitrary candidate is computed by adding up the penalties of the given contraints. Formally, the fitness measure f for a candidate solution x is defined as:

$$f(x) = \sum_{j=1}^{|C|} w_j \cdot \Phi(x, c_j) \ , \tag{6.1}$$

where w_j is the penalty (or weight) assigned to constraint c_j, and

$$\Phi(x, c_j) = \begin{cases} 1 & \text{if } x \text{ violates constraint } c_j, \\ 0 & \text{otherwise.} \end{cases} \tag{6.2}$$

One of the problems in solving constraint satisfaction problems using the penalty approach is finding the correct weights for the contraints. The weights

should ideally correspond with the difficulty of meeting the contraints since an evolutionary algorithm will primarily focus on solving the constraints resulting in the highest penalties when violated. The problem is that the difficulty of meeting an individual constraint is often not available or only at substantial costs.

The Stepwise Adaptation of Weights (SAW) method is designed to circumvent this problem by learning the contraint hardness during an evolutionary run and adapt the weights in the fitness function accordingly. In a SAW-ing EA the weights w_j of the contraints are initially set to a default value (typically $w_j = 1$). These weights are periodically updated every ΔT iterations or generations, in the case of a steady-state or generational model respectively, during the run. The weight update takes place by evaluating the then best individual in the population and increasing the weight of the constraints that are violated with a certain step size Δw (also typically 1). A general evolutionary algorithm using SAW is given in Algorithm 2.

Algorithm 2 Stepwise Adaptation of Weights (SAW).

```
set initial weights (thus fitness function f)
set G = 0
while not termination do
  run one generation of EA with f
  G = G + 1
  if G ≡ 0 mod ΔT then
    redefine f and recalculate fitness of individuals
  fi
od
```

The Stepwise Adaptation of Weights mechanism has been shown to improve the performance of evolutionary algorithms on different types of constraint satisfaction problems such as randomly generated binary CSPs [33, 51, 52], graph 3-colouring [31, 32, 52] and 3-SAT [2, 30].

If we look at the basic concept behind the Stepwise Adaptation of Weights method we can conclude that it is not restricted to constraint satisfaction problems.

To use SAW the only requirement is that the solution for a certain problem is determined by its performance on some elementary unit of quality judgement (e.g., constraint violations). The quality or fitness of the solution candidate can then be defined as the weighted sum of these elementary units

of quality judgement, where the weights indicate their hardness or importance.

In the rest of this chapter we investigate the usability of Stepwise Adaptation of Weights on tree-based Genetic Programming algorithms. First, we will show how SAW can be applied to GP algorithms for solving simple symbolic regression problems. Second, we will show how Stepwise Adaptation of Weights can be applied to the GP algorithms of the previous chapters for solving data classification problems. Since Stepwise Adaptation of Weights depends on two parameters (ΔT and Δw) we will test their robustness through their influence on the performance of our algorithms.

6.3 Symbolic Regression

In a regression problem the goal is to find a function that matches an unknown function defined by a finite set of sample points on a certain interval. In our case we will be looking at 2-dimensional functions. Thus, given a set of values $X = \{x_1, \ldots, x_n\}$ drawn from a certain interval and a corresponding set of sample points $S = \{(x_i, f(x_i) \mid x_i \in X\}$ the object is to find a function $g(x)$ such that $f(x_i) = g(x_i)$ for all $x_i \in X$.

To represent the candidate solutions we use a tree representation with unary (e.g., $y \mapsto y^3$, abbreviated y^3) and binary (e.g., $+, -, \times$) operators in internal nodes and variables and constants in the terminal nodes. An example tree is given in Figure 6.1.

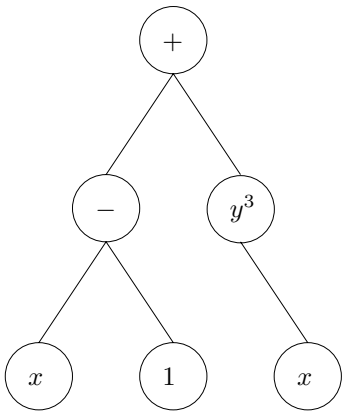

Figure 6.1: A regression tree representing the function $(x - 1) + x^3$.

In order to determine the quality of a candidate solution we need to define a fitness function. In symbolic regression we want to minimize the total error over all sample points. This is

$$\mathit{fitness}(g, X) = \sum_{x_i \in X} |f(x_i) - g(x_i)|, \tag{6.3}$$

where X is the set of values for variable x in a certain interval.

Note that instead of this absolute error measure other fitness functions are possible, for instance based on the mean squared error.

As we observed in Section 6.2 the SAW technique is not restricted to one type of problem. The only requirement for the SAW method is that the overall quality of the candidate solution depends on its performance on some elementary units of quality judgement. In the case of symbolic regression each sample point $(x_i, f(x_i))$ can be seen as such. Thus, by assigning a weight w_i to each sample point, we can transform Equation 6.3 into

$$\mathit{fitness}_{saw}(g, X) = \sum_{x_i \in X} w_i \ |f(x_i) - g(x_i)|, \tag{6.4}$$

where X is the set of values for variable x in a certain interval and w_i is a weight corresponding to the hardness of the sample point.

In constraint satisfaction problems a constraint is either violated or not and we can observe a similar thing for regression problems. For each sample point $(x_i, f(x_i))$ for which $f(x_i) \neq g(x_i)$ we can increase the weight by 1. However, unlike constraint satisfaction problems, we can also determine a distance measure to indicate if $g(x_i)$ is close to $f(x_i)$ or not. Therefore, instead of a fixed weight increase we can add a penalty $\Delta w_i = |f(x_i) - g(x_i)|$ to each weight w_i. This way the error on each sample point x_i is used to update the corresponding weight. We call this approach Precision SAW (PSAW).

6.3.1 Experiments and Results: Koza functions

To compare the SAW and PSAW methods to a standard GP algorithm we will use two functions introduced by Koza [67, pages 109–120]:

- quintic polynomial:

$$f(x) = x^5 - 2x^3 + x, \ x \in [-1, 1]. \tag{6.5}$$

- sextic polynomial:

$$f(x) = x^6 - 2x^4 + x^2,\ x \in [-1, 1]. \tag{6.6}$$

Each of the three GP algorithms (standard GP, GP+SAW and GP+PSAW) is tested using two different population sizes as shown in Table 6.1. On each of the six combinations of population size and number of generations we performed 99 independent runs in which we measure the mean, median, standard deviation, minimum and maximum absolute error (standard fitness). We will also consider the number of successful runs, whereby a successful run means that the algorithm has found a function with a standard fitness below 10^{-6}. The set of sample points consists of 50 points uniformly drawn from the domain $[-1, 1]$. Note that unlike the data classification experiments of the previous chapters we do not split the set of sample points into a test and training set but use the whole set for both the fitness calculations and determining the absolute error.

Table 6.1: Experiment parameters: six different experiments where each experiment consists of 99 independent runs.

experiment	populations size	number of generations
1	100	100
2	100	200
3	100	500
4	100	1000
5	500	100
6	500	200

The Genetic Programming parameters are displayed in Table 6.2. We use a generational model and a very small function set including a protected divide function (*pdiv*). This special division function is needed to prevent errors when dividing a number by zero. The *pdiv* function is defined by:

$$pdiv(a, b) = \begin{cases} a/b & \text{if } b \neq 0; \\ 0 & \text{otherwise.} \end{cases} \tag{6.7}$$

The terminal set consists only of variable x. The SAW and PSAW weights were updated every 5 generations ($\Delta T = 5$). The program for the Koza functions

Table 6.2: Parameters and characteristics of the Genetic Programming algorithms for experiments with the Koza functions.

Parameter	**value**
Evolutionary Model	$(\mu, 7\mu)$
Stop Criterion	maximum generations or perfect fit
Functions Set	$\{\times, pdiv, -, +\}$
Terminal Set	$\{x\}$
Populations Size (μ)	see Table 6.1
Initial Depth	3
Maximum Depth	5
Maximum Generations	see Table 6.1
Parent Selection	random
ΔT (for SAW)	5

was created with the GP kernel from the Data to Knowledge research project of the Danish Hydraulic Institute (`http://www.dhi.dk`).

Quintic Polynomial

The quintic polynomial is defined by Equation 6.5 and displayed in Fig. 6.2. Since the number of succesfull runs (a standard fitness below 10^{-6}) differs per algorithm and experiment it is difficult to determine which algorithm is the best. A closer look at the individual runs of experiment 4 (population size of 100 and 1000 generations) shows that GP has 76 successful runs out of 99, GP+SAW has 78 successful runs out of 99 and GP+PSAW has 85 successful runs out of 99. On experiments 5 and 6 both the GP+SAW and GP+PSAW algorithms are successful on all 99 independent runs, while the GP algorithm fails 2 times. However, none of these differences are statistically significant with a 95% confidence level. Based on the median results it would seem that GP+PSAW is the best choice but the differences are very small.

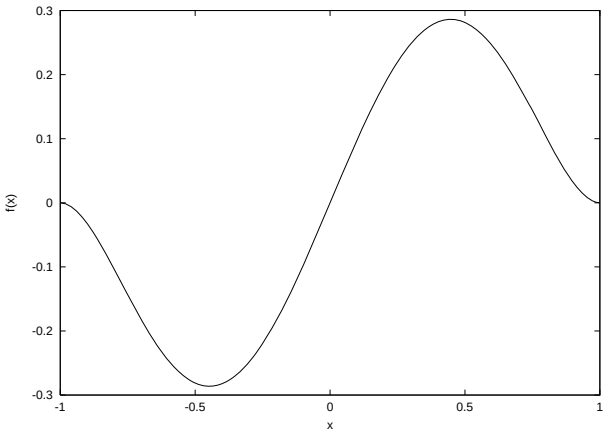

Figure 6.2: The quintic polynomial on the interval $[-1, 1]$.

Table 6.3: Experimental results for the quintic polynomial (all measurements with standard fitness function).

	experiment	median $\times 10^{-7}$	mean	st. deviation	minimum $\times 10^{-7}$	maximum
1.	GP	4.610	1.351×10^{-1}	2.679×10^{-1}	2.640	1.050
	GP+SAW	4.605	1.339×10^{-1}	2.599×10^{-1}	2.445	1.102
	GP+PSAW	4.391	1.286×10^{-1}	2.972×10^{-1}	2.598	1.559
2.	GP	4.354	1.274×10^{-1}	2.610×10^{-1}	2.640	1.034
	GP+SAW	4.303	1.226×10^{-1}	2.376×10^{-1}	2.445	0.853
	GP+PSAW	4.200	1.049×10^{-1}	2.254×10^{-1}	2.543	1.317
3.	GP	3.972	1.120×10^{-1}	2.571×10^{-1}	2.640	1.034
	GP+SAW	4.019	1.107×10^{-1}	2.204×10^{-1}	1.704	0.853
	GP+PSAW	3.855	7.785×10^{-2}	2.049×10^{-1}	2.449	1.111
4.	GP	3.763	1.161×10^{-1}	2.547×10^{-1}	2.324	1.034
	GP+SAW	3.693	8.323×10^{-2}	1.803×10^{-1}	1.704	0.6.97
	GP+PSAW	3.669	6.513×10^{-2}	1.856×10^{-1}	2.114	1.111
5.	GP	3.465	2.045×10^{-3}	1.544×10^{-2}	1.965	0.143
	GP+SAW	3.465	3.570×10^{-7}	8.463×10^{-8}	2.412	9.965×10^{-7}
	GP+PSAW	3.395	3.382×10^{-7}	4.384×10^{-8}	1.974	5.071×10^{-7}
6.	GP	3.343	2.045×10^{-3}	1.544×10^{-2}	1.965	0.143
	GP+SAW	3.446	3.512×10^{-7}	8.337×10^{-8}	2.412	9.965×10^{-7}
	GP+PSAW	3.325	3.331×10^{-7}	4.533×10^{-8}	1.937	5.071×10^{-7}

Sextic polynomial

The sextic polynomial is also taken from [67] and is defined in Equation 6.6. Figure 6.3 shows this function on the interval $[-1, 1]$.

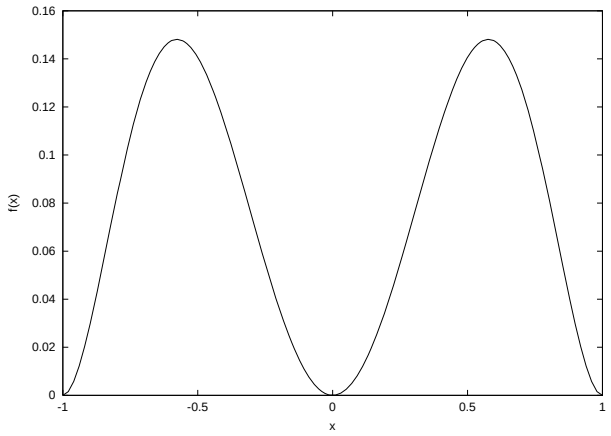

Figure 6.3: The sextic polynomial on the interval $[-1, 1]$.

In Table 6.4 we give the results of the experiments on this sextic polynomial. Similar to the quintic polynomial we see that the differrent algorithms have a different number of succesful runs. On experiment 4 (population size of 100 and 1000 generations) GP has 84 successful runs out of 99, GP+SAW has 81 successful runs and GP+PSAW has 87 successful runs. On experiment 6 we have a different outcome compared to the quintic polynomial. Here GP+SAW fails twice, GP+PSAW fails once and GP always succeeds. Again these results are not statistically significant with a 95% confidence level. Although the differences in the number of successful runs are not statistically significant, they do have a large impact on the mean and standard deviations. If we compare the median results of the different algorithms we see very little difference and in some cases no difference at all.

Table 6.4: Experimental results for the sextic polynomial (all measurements with standard fitness function).

	experiment	median $\times 10^{-7}$	mean	st. deviation	minimum $\times 10^{-7}$	maximum
1.	GP	2.182	1.490×10^{-1}	3.987×10^{-1}	1.723	2.844
	GP+SAW	2.182	1.525×10^{-1}	4.036×10^{-1}	1.353	2.844
	GP+PSAW	2.182	1.212×10^{-1}	2.569×10^{-1}	1.213	1.720
2.	GP	2.182	1.179×10^{-1}	2.882×10^{-1}	1.172	1.730
	GP+SAW	2.098	1.244×10^{-1}	3.626×10^{-1}	1.244	2.491
	GP+PSAW	2.115	1.135×10^{-1}	2.495×10^{-1}	1.013	1.720
3.	GP	1.953	8.001×10^{-2}	2.318×10^{-1}	1.171	1.730
	GP+SAW	1.916	8.366×10^{-2}	2.328×10^{-1}	1.172	1.222
	GP+PSAW	1.984	8.403×10^{-2}	2.226×10^{-1}	1.013	1.720
4.	GP	1.888	6.963×10^{-2}	2.258×10^{-1}	1.135	1.730
	GP+SAW	1.824	6.100×10^{-2}	1.741×10^{-1}	1.048	1.161
	GP+PSAW	1.899	5.084×10^{-2}	1.418×10^{-1}	1.013	0.547
5.	GP	1.385	1.507×10^{-7}	3.280×10^{-8}	1.087	2.912×10^{-7}
	GP+SAW	1.385	3.390×10^{-3}	2.500×10^{-2}	1.013	0.226
	GP+PSAW	1.385	2.485×10^{-3}	2.460×10^{-2}	1.125	0.246
6.	GP	**1.260**	1.417×10^{-7}	3.029×10^{-8}	1.087	2.912×10^{-7}
	GP+SAW	1.363	3.390×10^{-3}	2.500×10^{-2}	1.013	0.226
	GP+PSAW	1.260	2.485×10^{-3}	2.460×10^{-2}	1.115	0.246

6.3.2 Experiments and Results: Random Polynomials

In the previous section we tested the performance of the SAW and PSAW techniques on only two symbolic regression problems from literature, with somewhat dissapointing results. To study the performance of SAW enhanced GP algorithms on a large set of regression problems of a higher degree we will randomly generate polynomials.

We generate the polynomials using a model of two integer parameters $\langle a, b\rangle$, where a stands for the highest possible degree and b determines the domain size from which every coefficient is chosen. Using these parameters, we are able to generate polynomials of the form as shown in (6.8).

$$f(x)\sum_{i=0}^{a} e_i x^i, \text{ where } e_i \in \{w \mid w \in \mathbb{Z} \wedge -b \leq w \leq b\}, \tag{6.8}$$

where $\mathbb{Z}$ is the set of positive and negative integers including 0. The function $f(x)$ is presented to the regression algorithms by generating 50 points $(x, f(x))$ uniformly from the domain $[-1, 1]$. The values e_i are drawn uniform random from the integer domain bounded by $-b$ and b.

The parameters of the underlying GP system are given in Table 6.5. Since we will be looking at higher order polynomials with constant values we have extended the function set with two unary functions, negation (e.g., $x \mapsto -x$) and $y \mapsto y^3$, compared to the Koza functions. The terminal set consists of variable x and all possible constant integer values from the domain $[-b, b]$ that may occur in the polynomials.

Table 6.5: Parameters and characteristics of the Genetic Programming algorithms for experiments with the random polynomials.

Parameter	**value**
Evolutionary Model	steady state $(\mu + 1)$
Fitness Standard GP	see Equation (6.3)
Fitness SAW Variants	see Equation (6.4)
Stop Criterion	maximum evaluations or perfect fit
Functions Set	$\{\times, \mathit{pdiv}, -, +, x \mapsto -x, y \mapsto y^3\}$
Terminal Set	$\{x\} \cup \{w \mid w \in \mathbb{Z} \wedge -b \leq w \leq b\}$
Populations Size	100
Initial Maximum Depth	10
Maximum Size	100 nodes
Maximum Evaluations	20,000
Survivors Selection	reverse 5-tournament
Parent Selection	5-tournament
ΔT (for SAW)	1000, 2000 and 5000 evaluations

We generate polynomials randomly using parameters $\langle 12, 5 \rangle$. We generate 100 polynomials and do 85 independent runs of each algorithm where each run is started with a unique random seed. The GP algorithm was programmed using the *Evolving Objects* library (EOlib) [64]. EOlib is an Open Source C++ library for all forms of evolutionary computation and is available from `http://eodev.sourceforge.net`.

To determine whether the differences between our GP algorithms are statistically significant we used paired two-tailed t-tests with a 95% confidence level ($p = 0.05$) using the results of 8500 runs (100 random functions times 85 random seeds). In these tests the null-hypothesis is that the means of the two algorithms involved are equal.

Table 6.6: Experimental results for 100 randomly generated polynomials, results are averaged over the 85 independent runs.

algorithm	ΔT	mean	stddev	minimum	maximum
GP		21.20	10.30	5.82	48.60
GP+SAW	1000	21.55	10.33	5.79	48.05
GP+SAW	2000	21.21	10.24	5.83	48.74
GP+SAW	5000	21.25	10.47	5.75	49.31
GP+PSAW	1000	21.46	10.52	6.41	50.80
GP+PSAW	2000	20.45	10.05	5.90	48.06
GP+PSAW	5000	**20.28**	9.87	5.96	48.18

We present the results for the set of randomly generated polynomials in Table 6.6. The GP+SAW algorithm with a stepsize of 1000 performs significantly worse than the standard GP algorithm. The GP+PSAW algorithm with stepsizes 2000 and 5000 perform significantly better than any of the other GP algorithms. It looks like the new variant GP-PSAW performs better if we do not update the SAW weights too many times. This seems like a contradiction as we would expect the SAW mechanism to improve performance. However, we should not forget that after updating the weights the fitness function has changed (much like in a dynamic environment), so we need to give the GP algorithm time to adapt. It is also important to note that every extra weight update has less influence than the previous one as the relative weight difference between the sample points decreases with every weight update. Thus, the fitness function becomes "less dynamic" over time. Based on the results in Table 6.6 it seems that the SAW method is to some degree dependent on both the weight update (Δw) and the number of generations between weight updates (ΔT). In the next section we will perform a more thorough investigation into the influence of the stepsize parameter on our tree-based GP algorithms.

6.4 Data Classification

In the previous section we showed how Stepwise Adaptation of Weights can be applied to symbolic regression problems. Here we show how SAW can also be applied to data classification problems. The goal in data classification is to either minimize the number of misclassifications or maximize the number of correct classifications. For our GP algorithms we defined the standard fitness measure of a GP classifier x in Equation 2.3 as:

$$fitness_{standard}(x) = \frac{\sum_{r \in trainingset} \chi(x, r)}{|trainingset|} \times 100\%, \tag{6.9}$$

where $\chi(x, r)$ is defined as:

$$\chi(x, r) = \begin{cases} 1 & \text{if } x \text{ classifies record } r \text{ incorrectly;} \\ 0 & \text{otherwise.} \end{cases} \tag{6.10}$$

As we concluded in Section 6.2 the basic concept of the Stepwise Adaptation of Weights is not restricted to constraint satisfaction problems. The only requirement needed in order to use the SAW method is that the overall quality of a candidate solution is based on its performance on some elementary units of quality judgement (typically constraint violations). However, if we look at the standard misclassification fitness measure in Equation 6.9 and we compare it to the fitness measure used for constraint satisfaction problems (see Equation 6.1) we see that they are very similar. Instead of constraint violations we count misclassified records. Another difference between the two equations comes from the unknown penalties or weights that were used with constraint satisfaction problems. In constraint satisfaction problems these weights correspond to the hardness of the various constraints. In the case of data classification we can add a weight to each data record. We will use these weights to correspond to the classification hardness of the various data records. Thus we can transform Equation 6.9 to:

$$fitness_{saw}(x) = \frac{\sum_{r \in trainingset} w_r \times \chi(x, r)}{|trainingset|} \times 100\%, \tag{6.11}$$

where w_r corresponds to the classification hardness of record r.

Initially all weights w_r will be set to 1. Since our GP algorithms use a generational model the weights w_r will be updated every ΔT generations. Similar to constraint satisfaction the weights of the records misclassified by the then best individual in the population are increased by $\Delta w = 1$.

We should note that the weights w_r specified here are different from the weights usually specified in a data set cost matrix. A cost matrix determines the penalties assigned to false positive misclassifications and false negative misclassifications. One could for instance specify that classifying somebody as ill when he or she is healthy is not as bad as classifying somebody who is ill as healthy. Here we only consider data sets with default cost matrices where every misclassification has the same penalty.

We will add the *saw fitness* measure as the primary fitness value to our multi-layered fitness. Thus, the primary fitness measure becomes Equation 6.11, the secondary fitness value becomes Equation 6.9 and the third and final fitness measure is the number of nodes in the tree.

We only use the *saw fitness* measure during the evolutionary process. For selecting the best individual of an evolutionary run we first look at the *standard fitness* and if necessary the number of tree nodes (originally the third fitness measure). The reason for ignoring the *saw fitness* is that it is dependent on the weights while we are interrested in finding a decision tree which offers the best (unweighted) classification performance.

6.4.1 Experiments and Results

According to extensive tests performed on graph colouring and 3-SAT [50, 99] fine tuning the parameters for SAW, ΔT and Δw, is not necessary. However, the results in Section 6.3 indicate that both the choice of stepsize and weight function can influence the predictive capabilities of our GP algorithms. To check the influence of the stepsize parameter for Genetic Programming applied to Data Classification we will fix Δw on 1 but test 25 different settings of ΔT (1 till 25). All other parameters are left unchanged and are the same as in the previous chapters (see Table 6.7). We applied SAW to the *simple* GP algorithm and *clustering* GP algorithm with $k = 2$. Instead of a table with the results outcomes are shown in graphs in Figures 6.4 through 6.9 at the end of this chapter. Each graph has the stepsize on the x-axis and the misclassification rate on the y-axis and contains two lines. A solid horizonatal line indicates the misclassification rate of the algorithm without SAW. The dashed line indicates the misclassification rate of the algorithms using SAW.

To determine whether the differences between our GP algorithms are statistically significant we used paired two-tailed t-tests with a 95% confidence level ($p = 0.05$) using the results of 100 runs (10 random seeds times 10 folds). In these tests the null-hypothesis is also that the means of the two algorithms involved are equal.

Table 6.7: The main GP parameters.

Parameter	**Value**
Population Size	100
Initialization	ramped half-and-half
Initial Maximum Tree Depth	6
Maximum Number of Nodes	63
Parent Selection	tournament selection
Tournament Size	5
Evolutionary Model	$(100, 200)$
Crossover Rate	0.9
Crossover Type	swap subtree
Mutation Rate	0.9
Mutation Type	branch mutation
Stop Condition	99 generations

Australian Credit

In Figure 6.4 the average misclassification rates on the Australian Credit data set for our *simple* GP algorithm and *clustering* GP algorithm with $k = 2$ are plotted for stepsizes 1 to 25. When used in combination with our *simple* GP algorithm SAW seems to improve the misclassification performance for all but one value of ΔT. For stepsizes 1, 2, 3, 8, 10, 17, 18 and 24 these differences are statistically significant.

If we look at the influence of SAW on our *clustering* GP algorithm using $k = 2$ we see that SAW has a negative effect. In 12 of the stepsizes GP+SAW has a significantly higher misclassification rate. A possible reason for the different effect SAW has on the two algorithms is that on the Australian Credit data set our *clustering* using $k = 2$ performs the best of all our GP representations while the performance of our *simple* GP algorithm was the worst. It is easier

for SAW to improve the bad performance of our *simple* GP than to improve the already good performance of our *clustering* GP algorithm.

German Credit

The results for the German Credit data set are plotted in Figure 6.5. On the Australian Credit data set SAW seemed to improve the performance of our *simple* GP algorithm but on the German Credit data set SAW significantly decreases the performance in some cases. However, on the Australian Credit data set our *simple* GP algorithm had the worst performance of all our GP algorithms, whereas on the German Credit data set our *simple* GP algorithm performed the best. Thus, again Stepwise Adaptation of Weights doesn't improve an already "optimal" GP classifier. Our *clustering* GP algorithms using $k = 2$ also performed better than most of our GP algorithms on this data set and only SAW in combination with a stepsize of 7 performs significantly worse.

Pima Indians Diabetes

The average misclassification rates on the Pima Indians Diabetes data set (see Figure 6.6) were virtually the same for our *simple* and *clustering* GP algorithms. However, when SAW is applied to our *simple* GP algorithm we see no significant difference in performance. In the case of our *clustering* GP algorithm SAW significantly decreases the misclassification rate for stepsizes 2, 9, 12, 20 and 21. This could indicate that the classification performance of our *clustering* GP algorithm with $k = 2$ was close to optimal given the restrictions on the representation.

Heart Disease

Observe Figure 6.7 which shows the results for the Heart Disease data set. On this data set applying SAW to both GP algorithms seems to improve the misclassification rates for most stepsizes. However, in the case of our *simple* GP algorithm the differences are not statistically significant. Our *clustering* GP performed better than most of our GP classifiers but apparently did not achieve its best performance using the standard fitness measure. Using SAW with stepsizes 14 and 23 resulted in a significantly lower misclassification performance.

Ionosphere

The results for the Ionosphere data set are plotted in Figure 6.8. The combination of our *simple* GP algorithm and SAW significantly reduces classifiction performance for 10 of the 24 tested stepsizes. In the case of our *clustering* GP algorithm SAW with a stepsize of 16 significantly improves the classification results.

Iris

In Figure 6.9 the average misclassificaton rates for the Iris data set are plotted. Although the classification performance of the *simple* GP algorithm on the Iris data set was better than average, the application of SAW shows that it could have been even better. In the case of our *clustering* GP algorithm using $k = 2$ the results clearly show that the bad classification performance is caused by the restrictions of the representation rather than the evolutionary process or the fitness measures. However, according to a paired t-test, applying SAW to our *clustering* GP algorithm with a stepsize of 2 does significantly improve classification performance. Combined with our *simple* GP algorithm SAW only significantly improves performance with a stepsize of 14.

6.5 Conclusions

We have shown how the SAW technique can be applied to genetic programming for symbolic regression and data classification problems. For symbolic regression, we compared two variants of Stepwise Adaptation of Weights. The first variant, standard SAW, updates the weight using a constant value. This is similar to the way SAW is used for constraint satisfaction problems. Although this variant occassionally improves the performance (e.g., for randomly created polynomials) compared to a standard GP algorithm, it is outperformed by our more "precise" weight update method in most experiments. This second SAW variant, *precision* SAW (PSAW), updates the weights based on the distance between the target function and the best individual. Unlike SAW which increases the weight of a sample point $(x_i, f(x_i))$ by 1 if the current best individual $g(x)$ fails to predict $f(x_i)$ exactly, PSAW increases the weights based on the absolute difference between $f(x_i)$ and $g(x_i)$. This way, if there is only a small difference between $f(x_i)$ and $g(x_i)$, the weight of the corresponding sample point will be only slightly increased.

The experiments on the two functions from Koza [67] are inconclusive since the differences in the number of successful runs are not statistically significant. If we look at the results on the randomly created polynomials we see that the GP+PSAW algorithm perform the best, as long as there is enough time between weight updates.

In Section 6.4 we investigated the influence of the stepsize parameter ΔT on the classification outcome. Although earlier research [50, 99] on constraint satisfaction and 3-SAT problems indicated that fine-tuning the stepsize (ΔT) and weight-update (Δw) parameters is not necessary, this does not seem to be the case for genetic programming applied to data classification. On virtually all data set/algorithm combinations there was a value for ΔT which significantly increased or decreased the classification performance. Looking at the result graphs it is clear that selecting the optimal stepsize parameter setting for a certain data set and algorithm is virtually impossible. Moreover, given the small difference between the best and worst misclassification results per data set for each algorithm using SAW, it is questionable whether SAW is useful for data classification problems. It is difficult to speculate about the reasons why Stepwise Adaptation of Weights performs as it does for data classification problems. One potential reason could be the relatively small number of generations we use to evolve decision trees. As a result the GP algorithms may not have enough time to adapt to the changing of the SAW weights.

We like to point out that the simple concept behind SAW makes it very easy to implement the technique in existing algorithms, thereby making it suitable for doing quick "try outs" to improve the performance of an evolutionary algorithms. As SAW solely focuses on the fitness function it can be used in virtually any evolutionary algorithm. However, it is up to the user to find a good way of updating the weights mechanism depending on the problem at hand.

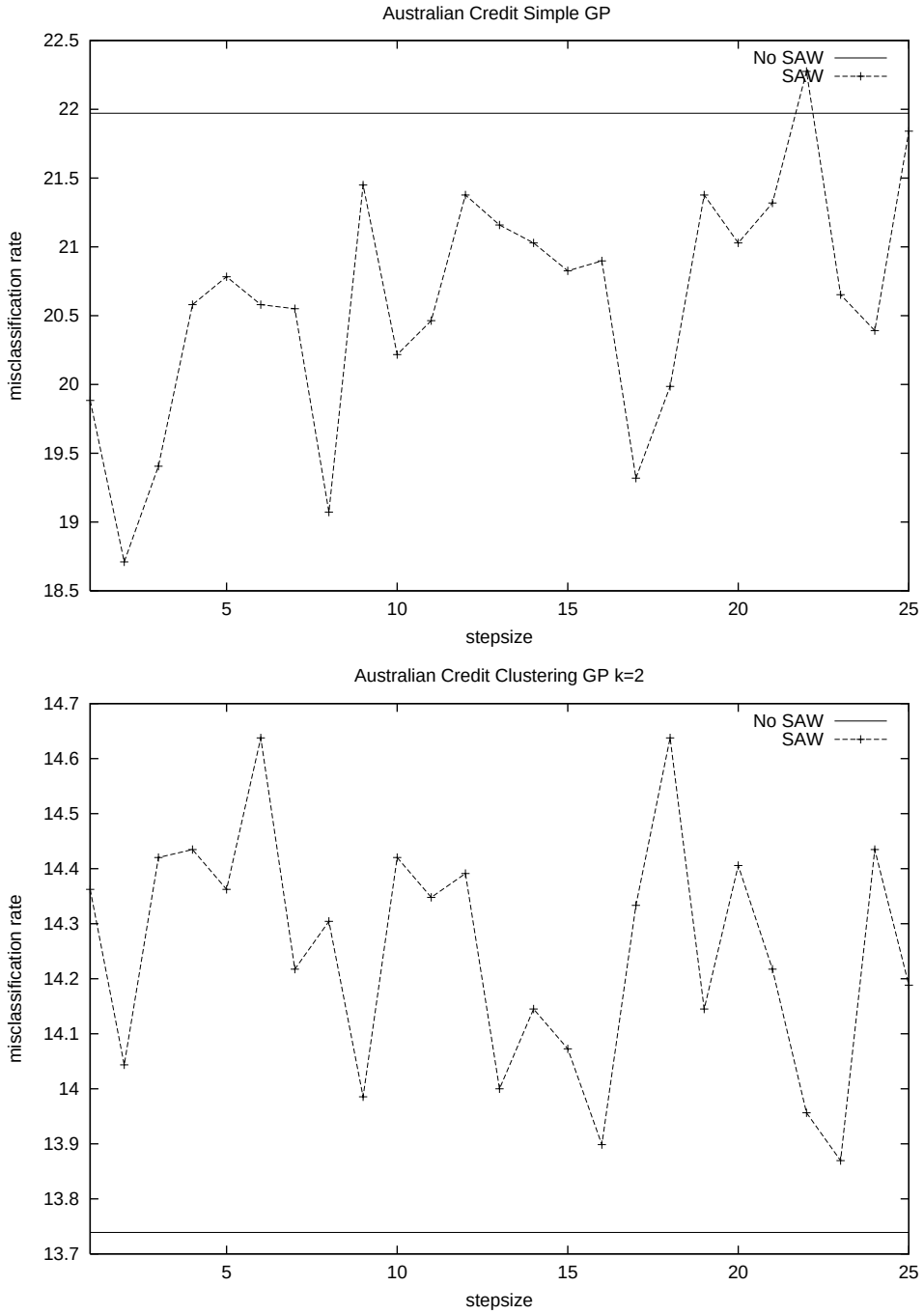

Figure 6.4: The average misclassification rates for the *simple* and *clustering* GP algorithms with $k = 2$ with stepsizes 1 through 25 on the Australian Credit data set.

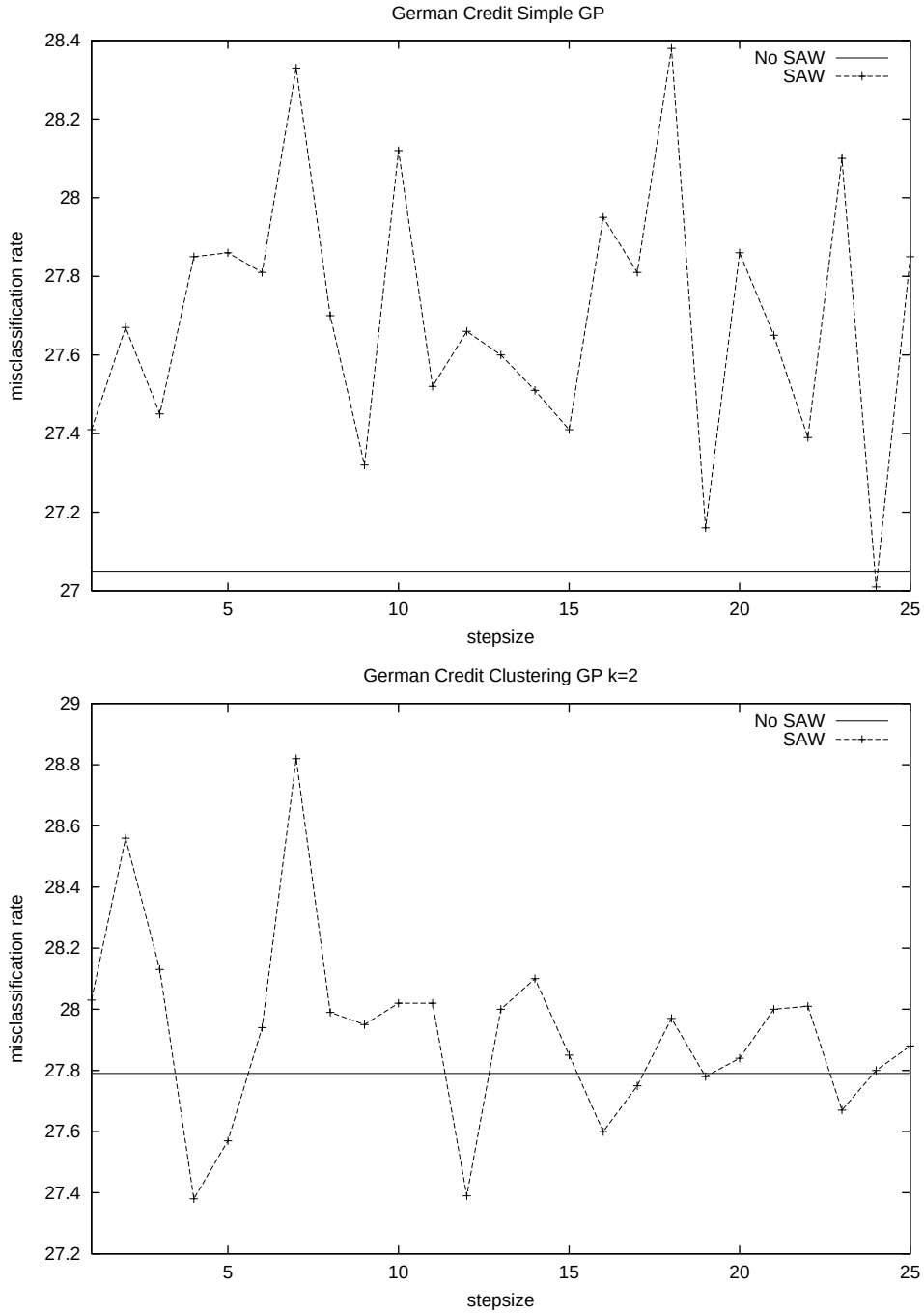

Figure 6.5: The average misclassification rates for the *simple* and *clustering* GP algorithms with $k = 2$ with stepsizes 1 through 25 on the German Credit data set.

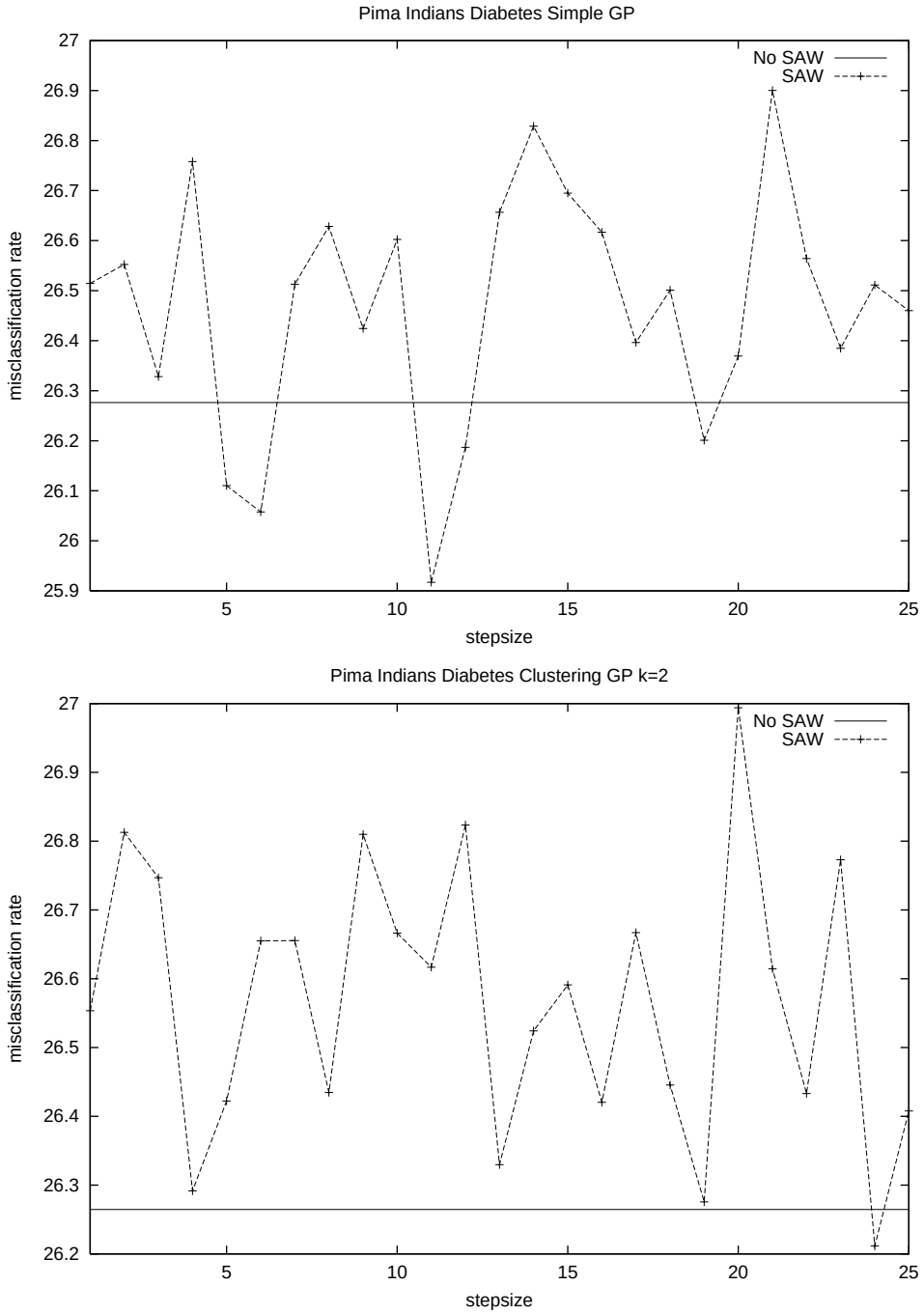

Figure 6.6: The average misclassification rates for the *simple* and *clustering* GP algorithms with $k = 2$ with stepsizes 1 through 25 on the Pima Indians Diabetes data set.

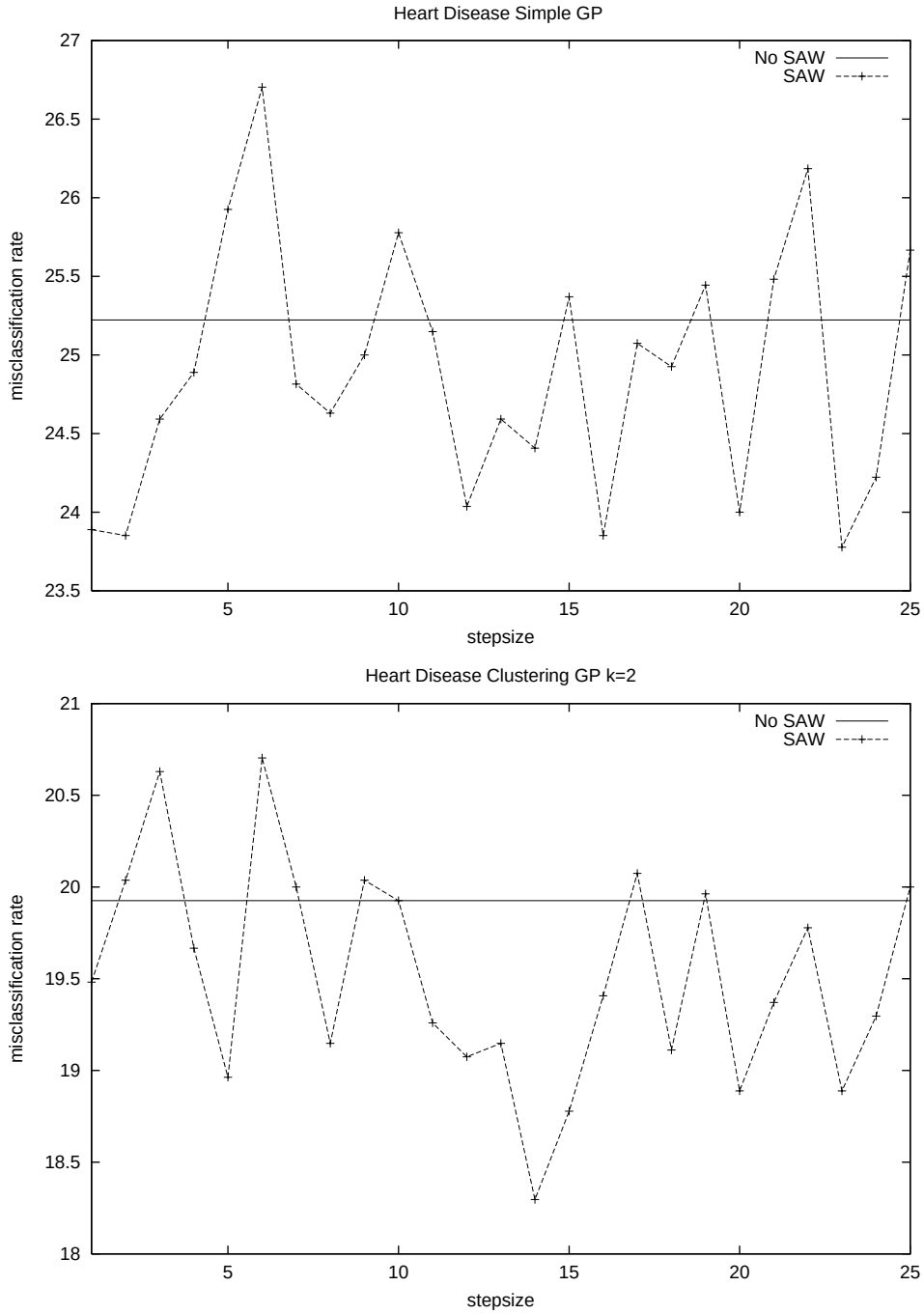

Figure 6.7: The average misclassification rates for the *simple* and *clustering* GP algorithms with $k = 2$ with stepsizes 1 through 25 on the Heart Disease data set.

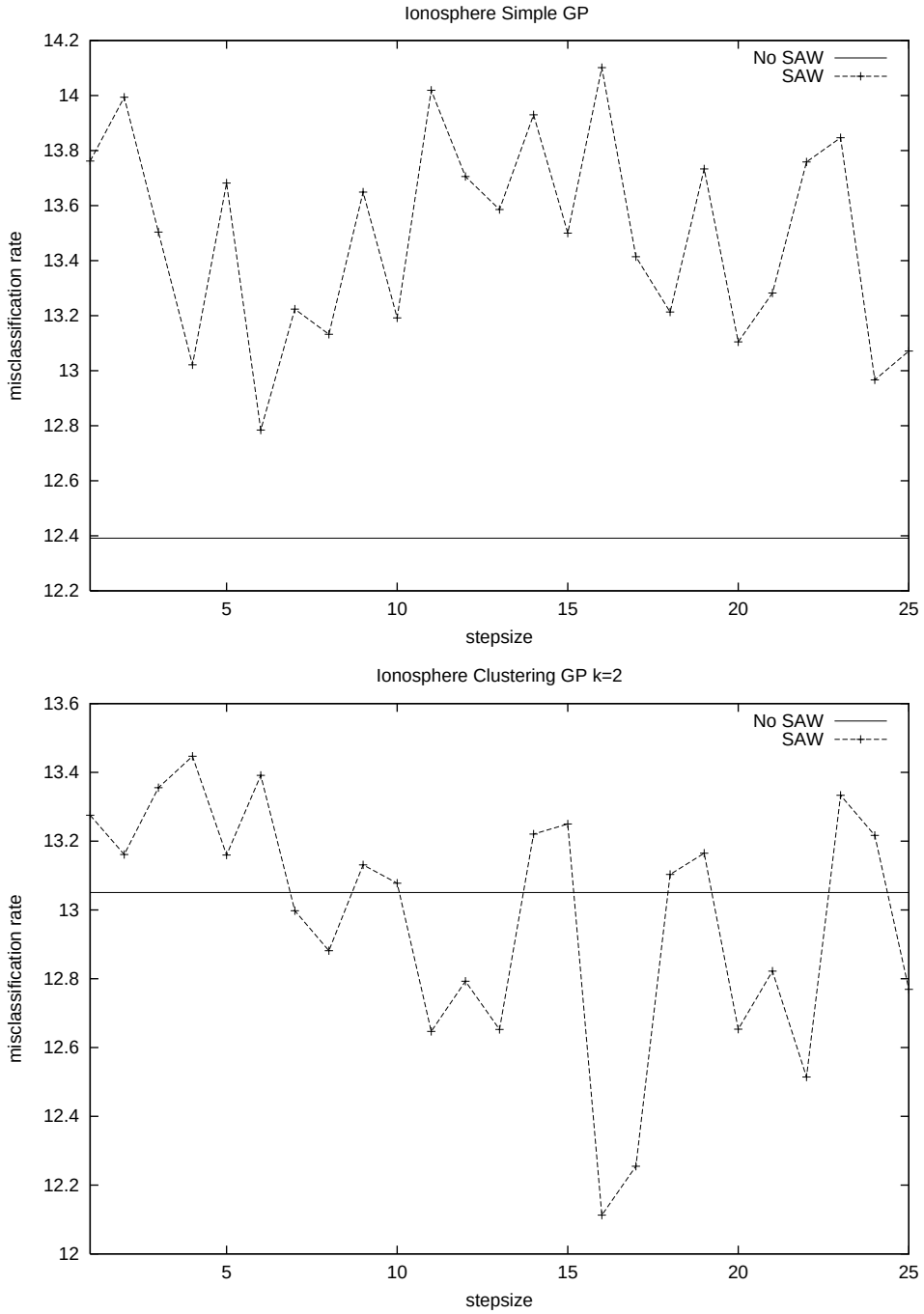

Figure 6.8: The average misclassification rates for the *simple* and *clustering* GP algorithms with $k = 2$ with stepsizes 1 through 25 on the Ionosphere data set.

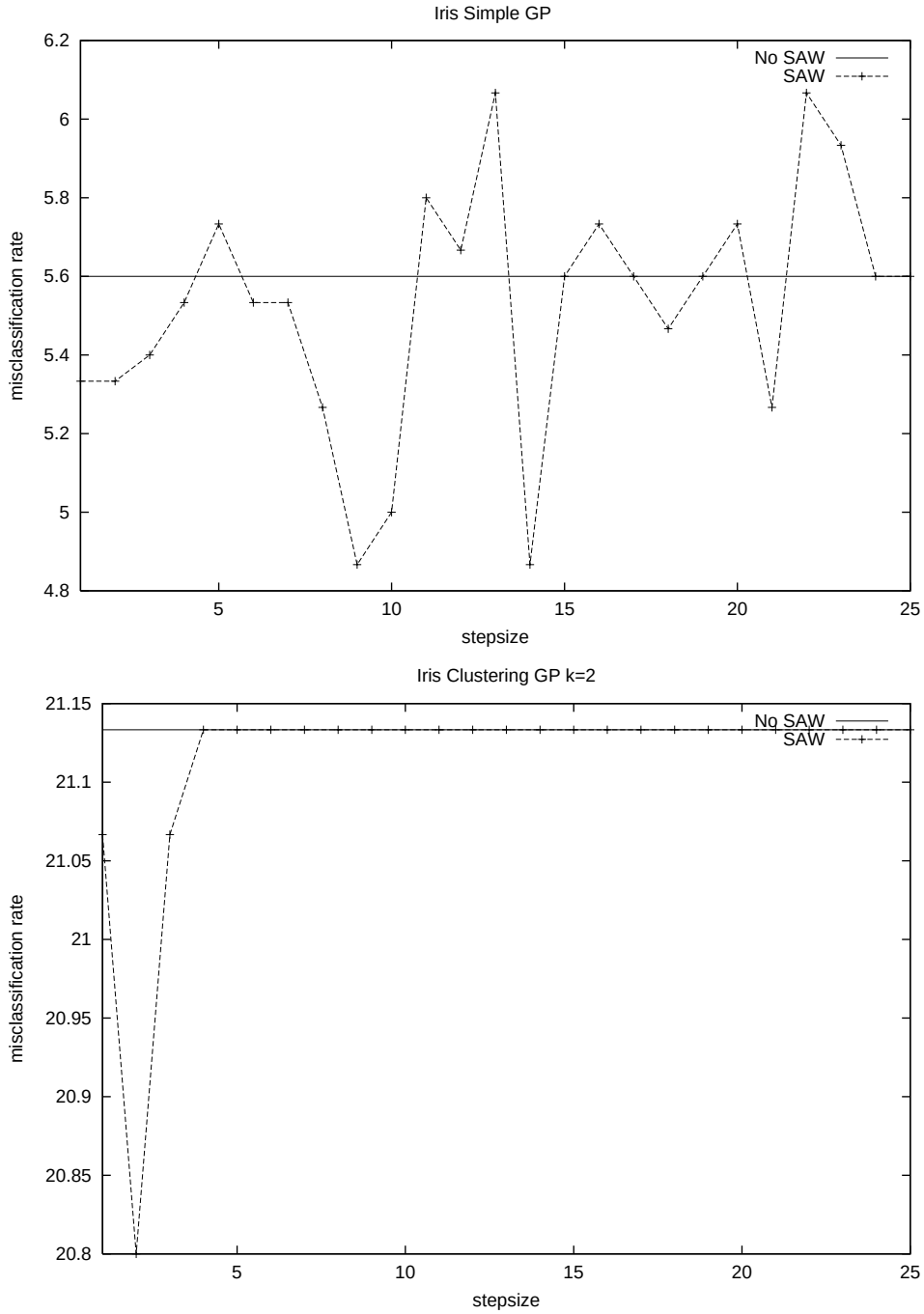

Figure 6.9: The average misclassification rates for the *simple* and *clustering* GP algorithms with $k = 2$ with stepsizes 1 through 25 on the Iris data set.

A Tree-based Genetic Programming

The main differences between the different subclasses of evolutionary computation are their representations. In order to use a specific representation for an evolutionary algorithm one needs to specify the initialization method and variation operators. In this appendix we discuss the (standard) initialization and variation routines we used for our tree-based Genetic Programming algorithms. We start with the initialization methods in Section A.1. In Section A.2 we continue with the variation process and the genetic operators.

A.1 Initialization

The first step of an evolutionary algorithm is the initialization of the population. In the case of tree-based Genetic Programming this means we have to construct syntactically valid trees. The main parameter for the initialization method is the *maximum tree depth*. This parameter is used to restrict the size of the initialized trees. Apart from the *maximum tree depth* parameter the initialization function needs a set of possible terminals T and a set of possible internal nodes I (non-terminals). For tree-based GP there are two common methods to construct trees: the *grow* method and the *full* method.

The *grow* method is given in Algorithm 3 as a recursive function returning a *node* and taking the depth of the *node* to be created as argument. The function is initially called with *depth* 0. If *depth* is smaller than the *maximum tree depth* a node is chosen randomly from the set of terminals and non-terminals. Next depending on whether *node* is a terminal (without child nodes) or internal node the *grow* method is called to create the children of *node*. If *depth* does equal *maximum tree depth* a node is chosen from the set of terminals.

Algorithm 3 *node* **grow**(*depth*)

```
if depth < maximum tree depth
    node ← random(T ∪ I)
    for i = 1 to number of children of node do
       child_i = grow(depth+1)
    od
else
    node ← random(T)
fi
return node
```

An example of the *grow* method with a *maximum tree depth* of 2 can be seen in Figure A.1. The root node with non-terminal I_5 is created first. The first child of the root node becomes a terminal T_3 while the second child becomes another internal node containing non-terminal I_1. Because of the limitations set by the *maximum tree depth* both children of node I_1 become terminals.

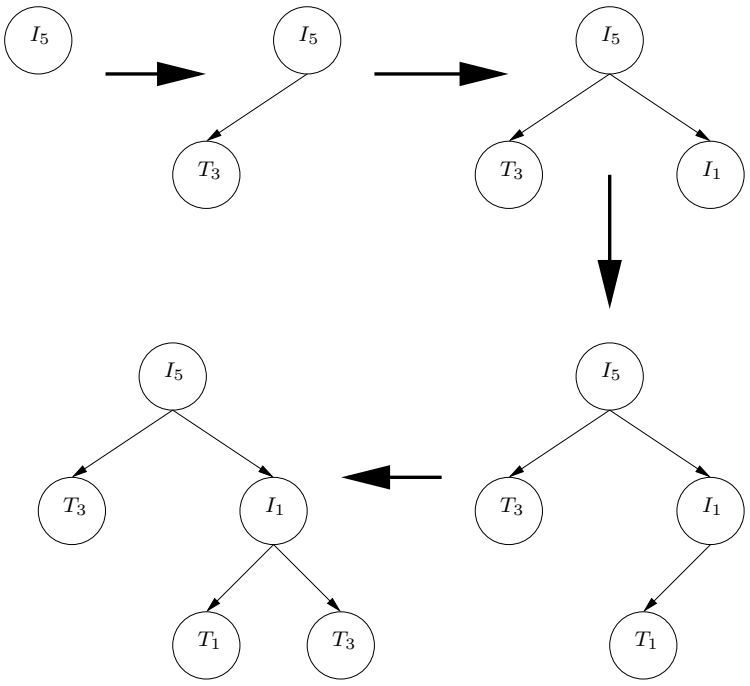

Figure A.1: An example of a tree constructed using the *grow* method.

Algorithm 4 *node* **full**(*depth*)

```
if depth < maximum tree depth
    node ← random(I)
    for i = 1 to number of children of node do
      child_i = full(depth+1 )
    od
else
    node ← random(T)
fi
return node
```

The *full* method, given in Algorithm 4, is similar to the *grow* method with one important exception. If *depth* is smaller than the maximum allowed tree depth a node is chosen randomly from the set of internal nodes I and not from the combined set of terminals (T) and non-terminals (I). An example of a tree created using the *full* method with *maximum tree depth* 2 can be seen in Figure A.2. First a root node containing non-terminal I_5 is created, which has two children. The first child is created with non-terminal I_2 which has only a single child which becomes a terminal. The second child is created with non-terminal I_1 which has two terminals as children.

A.1.1 Ramped Half-and-Half Method

To ensure that there is enough diversity in the population a technique has been devised called *Ramped Half-and-Half* [65] which combines the *grow* and *full* initialization methods. Given a *maximum tree depth* D the *Ramped Half-and-Half* method divides the population size into $D-1$ groups. Each group uses a different *maximum tree depth* $(2, \ldots, D)$, whereby half of each group is created using the *grow* method and the other half using the *full* method. The result is a mix of irregular trees of different depths created by the *grow* method and more regular trees created by the *full* method.

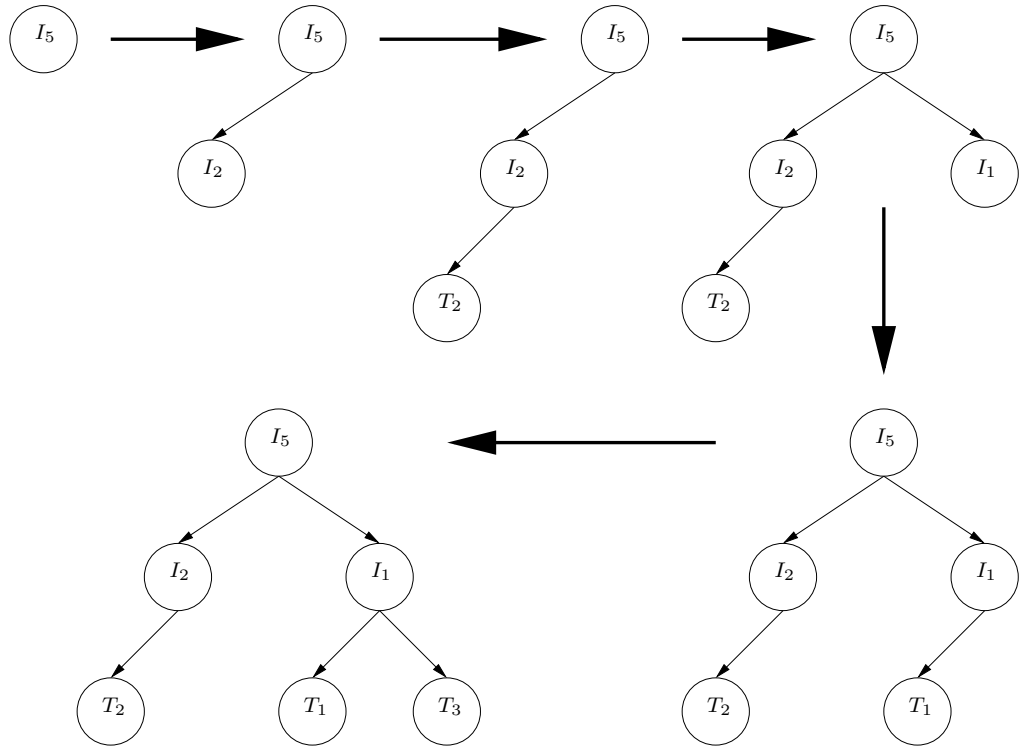

Figure A.2: An example of a tree constructed using the *full* method.

A.2 Genetic Operators

After the initial population has been initialized and evaluated by a fitness function the actual evolutionary process starts. The first step of each generation is the selection of parents from the current population. These parents are employed to produce offspring using one or more genetic operators. In evolutionary computation we can distinguish between two different types of operators: crossover and mutation:

- The crossover or recombination operator works by exchanging "genetic material" between two or more parent individuals and may result in several offspring individuals.
- The mutation operator is applied to a single individual at a time and makes (small) changes in the genetic code of an individual. Mutation is often applied to the individuals produced by the crossover operator.

The combination of crossover and mutation mimics procreation in nature where the DNA of a child is a combination of the DNA of its parents whereby as a result of DNA copying errors also small mutations arise. An overview of the variation process can be seen in Figure A.3.

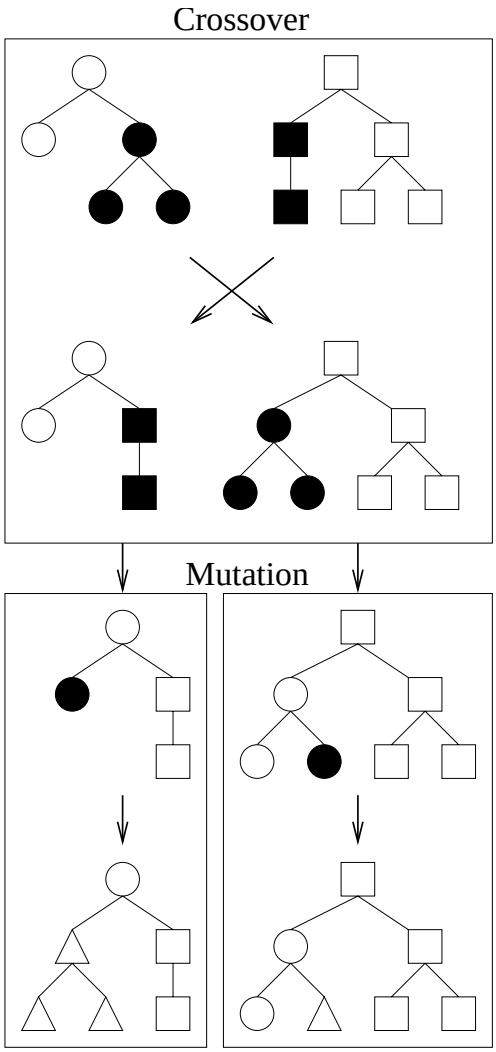

Figure A.3: Overview of the variation process.

A.2.1 Crossover

There are several different crossover operators possible for tree-based Genetic Programming [4]. In this thesis we use the standard tree-crossover operator used for GP: *subtree exchange crossover*. A schematic example of *subtree exchange crossover* can be seen in Figure A.4. After two parent individuals have been selected, in this case the "round" tree and the "square" tree, a subtree is randomly chosen in each of the parents. Next the two subtrees are exchanged resulting in two trees each of which is a different combination of the two parents.

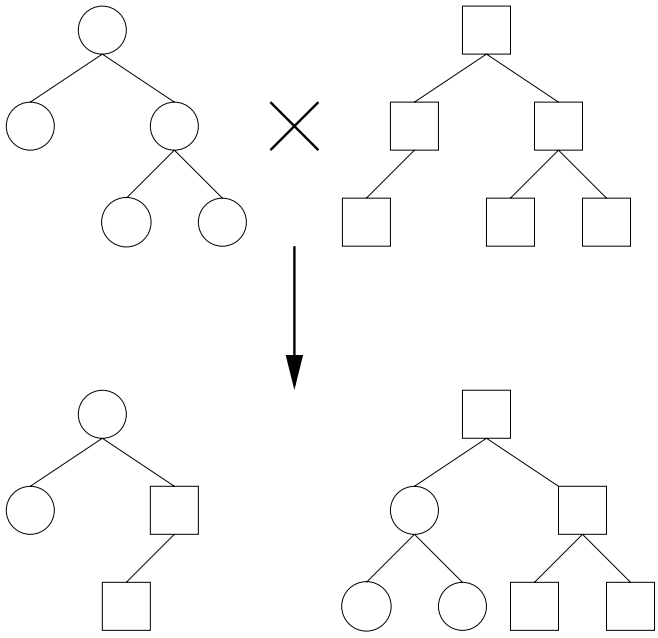

Figure A.4: Subtree crossover.

A.2.2 Mutation

After the crossover operator we apply the mutation operator. In this thesis we use *subtree replacement* as mutation operator. A schematic example of *subtree replacement mutation* can be seen in Figure A.5. *Subtree replacement mutation* selects a subtree in the individual to which it is applied and replaces the subtree with a randomly created tree. This subtree is usually created using one of the initialization methods described in Section A.1.

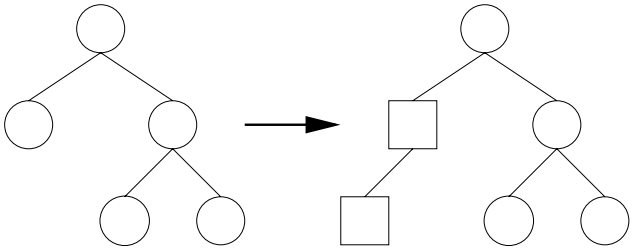

Figure A.5: Subtree mutation.

Bibliography

[1] P.J. Angeline. Genetic programming and emergent intelligence. In K.E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press, 1994.

[2] T. Bäck, A.E. Eiben, and M.E. Vink. A superior evolutionary algorithm for 3-SAT. In V. W. Porto, N. Saravanan, D. Waagen, and A.E. Eiben, editors, *Proceedings of the 7th Annual Conference on Evolutionary Programming*, volume 1477 of *LNCS*, pages 125–136. Springer-Verlag, 1998.

[3] T. Bäck, D.B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. IOP Publishing Ltd., 1997.

[4] W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone. *Genetic Programming — An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, 1998.

[5] M. Berthold. Fuzzy logic. In M. Berthold and D.J. Hand, editors, *Intelligent Data Analysis, An Introduction*, pages 269–298. Springer, 1999.

[6] S. Bhattacharyya, O. Pictet, and G. Zumbach. Representational semantics for genetic programming based learning in high-frequency financial data. In J.R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 11–16, University of Wisconsin, Madison, Wisconsin, USA, 1998. Morgan Kaufmann.

[7] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998.

[8] T. Blickle. Evolving compact solutions in genetic programming: A case study. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving From Nature IV. Proceedings of the International Conference on Evolutionary Computation*, volume 1141 of *LNCS*, pages 564–573. Springer-Verlag, 1996.

[9] A. Blumer, A. Ehrenfeucht, D. Haussler, and M.K. Warmuth. Occam's razor. *Inf. Process. Lett.*, 24:377–380, 1987.

[10] C.C. Bojarczuk, H.S. Lopes, and A.A. Freitas. Discovering comprehensible classification rules by using genetic programming: A case study in a medical domain. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 953–958, Orlando, Florida, USA, 1999. Morgan Kaufmann.

[11] M. Bot. Application of genetic programming to the induction of linear programming trees. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, 1999.

[12] M.C.J. Bot and W.B. Langdon. Application of genetic programming to induction of linear classification trees. In R. Poli, W. Banzhaf, W.B. Langdon, J.F. Miller, P. Nordin, and T.C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 247–258, Edinburgh, 2000. Springer-Verlag.

[13] M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5:17–26, 2001.

[14] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

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

[16] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. John Wiley, New York, 1991.

[17] C. Darwin. *On the origin of species: By Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life.* Murray, London, 1859.

[18] P. Domingos. Occam's two razors: The sharp and the blunt. In R. Agrawal and P. Stolorz, editors, *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, pages 37–43. AAAI Press, 1998.

[19] P. Domingos. The role of Occam's razor in knowledge discovery. *Data Mining and Knowledge Discovery*, 3:409–425, 1999.

[20] J. Eggermont. Evolving fuzzy decision trees for data classification. In H. Blockeel and M. Denecker, editors, *Proceedings of the 14th Belgium/Netherlands Conference on Artificial Intelligence (BNAIC'02)*, pages 417–418, Leuven, Belgium, 2002.

[21] J. Eggermont. Evolving fuzzy decision trees with genetic programming and clustering. In J.A. Foster, E. Lutton, J. Miller, C. Ryan, and A.G.B. Tettamanzi, editors, *Proceedings on the Fifth European Conference on Genetic Programming (EuroGP'02)*, volume 2278 of *LNCS*, pages 71–82, Kinsale, Ireland, 2002. Springer-Verlag.

[22] J. Eggermont, A.E. Eiben, and J.I. van Hemert. Adapting the fitness function in GP for data mining. In R. Poli, P. Nordin, W.B. Langdon, and T.C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 195–204, Goteborg, Sweden, 1999. Springer-Verlag.

[23] J. Eggermont, A.E. Eiben, and J.I. van Hemert. Comparing genetic programming variants for data classification. In E. Postma and M. Gyssens, editors, *Proceedings of the Eleventh Belgium/Netherlands Conference on Artificial Intelligence (BNAIC'99)*, pages 253–254, Maastricht, The Netherlands, 1999.

[24] J. Eggermont, A.E. Eiben, and J.I. van Hemert. A comparison of genetic programming variants for data classification. In D.J. Hand, J.N. Kok, and M.R. Berthold, editors, *Advances in Intelligent Data Analysis, Third International Symposium, IDA-99*, volume 1642 of *LNCS*, pages 281–290, Amsterdam, The Netherlands, 1999. Springer-Verlag.

[25] J. Eggermont, J.N. Kok, and W.A. Kosters. Genetic programming for data classification: Refining the search space. In T. Heskes, P. Lucas, L. Vuurpijl, and W. Wiegerinck, editors, *Proceedings of the 15th Belgium/Netherlands Conference on Artificial Intelligence (BNAIC'03)*, pages 123–130, 2003.

[26] J. Eggermont, J.N. Kok, and W.A. Kosters. Detecting and pruning introns for faster decision tree evolution. In X. Yao, E. Burke, J.A. Lozano, J.Smith, J.J. Merelo-Guervós, J.A. Bullinaria, J.Rowe, P. Tiňo, A. Kabán, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature — PPSN VIII*, volume 3242 of *LNCS*, pages 1071–1080, Birmingham, United Kingdom, 2004. Springer-Verlag.

[27] J. Eggermont, J.N. Kok, and W.A. Kosters. Genetic programming for data classification: Partitioning the search space. In *Proceedings of the 2004 Symposium on applied computing (ACM SAC'04)*, pages 1001–1005, Nicosia, Cyprus, 2004.

[28] J. Eggermont and J. I. van Hemert. Stepwise adaptation of weights for symbolic regression with genetic programming. In A. van den Bosch and H. Weigand, editors, *Proceedings of the Twelfth Belgium/Netherlands Conference on Artificial Intelligence (BNAIC'00)*, pages 259–267, Kaatsheuvel, The Netherlands, 2000.

[29] J. Eggermont and J. I. van Hemert. Adaptive genetic programming applied to new and existing simple regression problems. In J. Miller, M. Tomassini, P.L. Lanzi, C. Ryan, A.G.B. Tetamanzi, and W.B. Langdon, editors, *Proceedings on the Fourth European Conference on Genetic Programming (EuroGP'01)*, volume 2038 of *LNCS*, pages 23–35. Springer-Verlag, 2001.

[30] A.E. Eiben and J.K. van der Hauw. Solving 3-SAT with adaptive Genetic Algorithms. In *Proceedings of the 4th IEEE Conference on Evolutionary Computation*, pages 81–86. IEEE Press, 1997.

[31] A.E. Eiben and J.K. van der Hauw. Adaptive penalties for evolutionary graph-coloring. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution '97*, number 1363 in LNCS, pages 95–106. Springer-Verlag, 1998.

[32] A.E. Eiben, J.K. van der Hauw, and J.I. van Hemert. Graph coloring with adaptive evolutionary algorithms. *Journal of Heuristics*, 4:25–46, 1998.

[33] A.E. Eiben, J.I. van Hemert, E. Marchiori, and A.G. Steenbeek. Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function. In A.E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Proceedings of the 5th Conference on Parallel Problem Solving from Nature*, number 1498 in LNCS, pages 196–205. Springer-Verlag, 1998.

[34] A.E. Eiben, P.-E. Raué, and Zs. Ruttkay. GA-easy and GA-hard constraint satisfaction problems. In M. Meyer, editor, *Proceedings of the ECAI-94 Workshop on Constraint Processing*, number 923 in LNCS, pages 267–284. Springer-Verlag, 1995.

[35] T.S. Eliot. *The Rock*. Faber and Faber, London, 1934.

[36] U.M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 1–30. AAAI Press, 1996.

[37] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons, Inc., New York, 1966.

[38] A.A. Freitas. Understanding the crucial role of attribute interaction in data mining. *Artif. Intell. Rev.*, 16:177–199, 2001.

[39] A.A. Freitas. Evolutionary computation. In W. Klösgen and J. Zytkow, editors, *Handbook of Data Mining and Knowledge Discovery*, pages 698–706. Oxford University Press, 2002.

[40] A.A. Freitas. A survey of evolutionary algorithms for data mining and knowledge discovery. In A. Ghosh and S. Tsutsui, editors, *Advances in Evolutionary Computation*, pages 819–845. Springer-Verlag, 2002.

[41] Y. Freund and R.E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Proceedings of the Second European Conference on Computational Learning Theory*, volume 904 of *LNCS*, pages 23–37. Springer-Verlag, 1995.

[42] Y. Freund and R.E. Schapire. Experiments with a new boosting algorithm. In *Proceedings 13th International Conference on Machine Learning*, pages 148–146. Morgan Kaufmann, 1996.

[43] J. Gama. Oblique linear tree. In X. Liu, P. Cohen, and M. Berthold, editors, *IDA '97: Proceedings of the Second International Symposium on Advances in Intelligent Data Analysis, Reasoning about Data*, volume 1280 of *LNCS*, pages 187–198. Springer-Verlag, 1997.

[44] C. Gathercole and P. Ross. Dynamic training subset selection for supervised learning in genetic programming. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature III*, volume 866 of *LNCS*, pages 312–321, Jerusalem, 1994. Springer-Verlag.

[45] C. Gathercole and P. Ross. Some training subset selection methods for supervised learning in genetic programming. Presented at ECAI'94 Workshop on Applied Genetic and other Evolutionary Algorithms, 1994.

[46] C. Gathercole and P. Ross. An adverse interaction between crossover and restricted tree depth in genetic programming. In J.R. Koza, D.E. Goldberg, D.B. Fogel, and R.L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 291–296, Stanford University, CA, USA, 1996. MIT Press.

[47] C. Gathercole and P. Ross. Tackling the boolean even N parity problem with genetic programming and limited-error fitness. In J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba, and R.L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 119–127, Stanford University, CA, USA, 1997. Morgan Kaufmann.

[48] J.M. de Graaf, W.A. Kosters, and J.J.W. Witteman. Interesting fuzzy association rules in quantitative databases. In L. de Raedt and A. Siebes, editors, *5th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'01)*, volume 2168 of *LNAI*, pages 140–151. Springer-Verlag, 2001.

[49] R.L. Graham, M. Grötschel, and L. Lovász. *Handbook of Combinatorics — Volume II.* Elsevier, 1995.

[50] J.K. van der Hauw. Evaluating and improving steady state evolutionary algorithms on constraint satisfaction problems. Master's thesis, Leiden University, 1996.

[51] J.I. van Hemert. Applying adaptive evolutionary algorithms to hard problems. Master's thesis, Leiden University, 1998.

[52] J.I. van Hemert. *Application of Evolutionary Computation to Constraint Satisfaction and Data Mining*. PhD thesis, Universiteit Leiden, 2002.

[53] J.I. van Hemert and T. Bäck. Measuring the searched space to guide efficiency: The principle and evidence on constraint satisfaction. In J.J. Merelo, A. Panagiotis, H.-G. Beyer, J.-L. Fernández-Villacañas, and H.-P. Schwefel, editors, *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, number 2439 in LNCS, pages 23–32. Springer-Verlag, 2002.

[54] W. Hillis. Co-evolving parasites improve simulated evolution as an opimization procedure. *Artificial Life II*, pages 313–324, 1992.

[55] J.H. Holland. *Adaptation in natural artificial systems*. University of Michigan Press, Ann Arbor, 1975.

[56] D. Hooper and N.S. Flann. Improving the accuracy and robustness of genetic programming through expression simplification. In J.R. Koza, D.E. Goldberg, D.B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 428, Stanford University, CA, USA, 1996. MIT Press.

[57] Y. Hu. A genetic programming approach to constructive induction. In J.R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 146–151, University of Wisconsin, Madison, Wisconsin, USA, 1998. Morgan Kaufmann.

[58] L. Hyafil and R.L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5:15–17, 1976.

[59] D.D. Jensen and P.R. Cohen. Multiple comparisons in induction algorithms. *Mach. Learn.*, 38:309–338, 2000.

[60] C. Johnson. Deriving genetic programming fitness properties by static analysis. In J.A. Foster, E. Lutton, J. Miller, C. Ryan, and A.G.B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 298–307, Kinsale, Ireland, 2002. Springer-Verlag.

[61] L. Kaufman. *Finding Groups in Data: An Introduction to Cluster Analysis.* Wiley, New York, 1990.

[62] M. Keijzer. *Scientific Discovery Using Genetic Programming.* PhD thesis, Danish Technical University, 2002.

[63] M. Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 71–83, Essex, 2003. Springer-Verlag.

[64] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving objects: A general purpose evolutionary computation library. In P. Collet, C. Fonlupt, J.K. Hao, E. Lutton, and Schoenauer M, editors, *Proceedings of Evolution Artificielle'01*, volume 2310 of *LNCS*. Springer-Verlag, 2001.

[65] J.R. Koza. A genetic approach to the truck backer upper problem and the inter-twined spiral problem. In *Proceedings of IJCNN International Joint Conference on Neural Networks*, volume IV, pages 310–318. IEEE Press, 1992.

[66] J.R. Koza. *Genetic Programming.* MIT Press, 1992.

[67] J.R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, MA, 1994.

[68] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13:32–44, 1992.

[69] W. B. Langdon and R. Poli. *Foundations of Genetic Programming.* Springer-Verlag, 2002.

[70] W.B. Langdon. *Genetic Programming + Data Structures = Automatic Programming!* Kluwer, 1998.

[71] W.B. Langdon, T. Soule, R. Poli, and J.A. Foster. The evolution of size and shape. In L. Spector, W.B. Langdon, U-M. O'Reilly, and P.J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, 1999.

[72] J.J. Liu and J.T. Kwok. An extended genetic rule induction algorithm. In *Proceedings of the 2000 Congress on Evolutionary Computation*, pages 458–463, Piscataway, NJ, 2000. IEEE.

[73] P. Lyman and H.R. Varian. *How Much Information? 2003.* http://www.sims.berkeley.edu/how-much-info-2003, 2004.

[74] N.F. McPhee and J.D. Miller. Accurate replication in genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, Pittsburgh, PA, USA, 1995. Morgan Kaufmann.

[75] R.R.F. Mendes, F.B. Voznika, A.A. Freitas, and J.C. Nievola. Discovering fuzzy classification rules with genetic programming and co-evolution. In L. de Raedt and A. Siebes, editors, *5th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'01)*, volume 2168 of *LNAI*, pages 314–325. Springer-Verlag, 2001.

[76] D. Michie, D.J. Spiegelhalter, and C.C. Taylor (eds). *Machine Learning, Neural and Statistical Classification.* http://www.amsta.leeds.ac.uk/~charles/statlog, 1994.

[77] D.J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[78] S.K. Murthy, S. Kasif, and S. Salzberg. A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2:1–32, 1994.

[79] P. Nordin, F. Francone, and W. Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In J.P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory*

to Real-World Applications, pages 6–22, Tahoe City, California, USA, 1995.

[80] D. Palomo van Es. Fuzzy association rules and promotional sales data. Master's thesis, Leiden University, 2001.

[81] J. Paredis. Co-evolutionary constraint satisfaction. In *PPSN III: Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature*, volume 866 of *LNCS*, pages 46–55. Springer-Verlag, 1994.

[82] J. Paredis. Coevolutionary computation. *Artif. Life*, 2:355–375, 1995.

[83] G. Paris, D. Robilliard, and C. Fonlupt. Exploring overfitting in genetic programming. In P. Liardet, P. Collet, C. Fonlupt, E. Lutton, and M. Schoenauer, editors, *Evolution Artificielle, 6th International Conference*, volume 2936 of *LNCS*, pages 267–277, Marseilles, France, 2003. Springer-Verlag.

[84] J. Pena, J. Lozano, and P. Larranaga. An empirical comparison of four initialization methods for the k-means algorithm. *Pattern Recognition Letters*, 20:1027–1040, 1999.

[85] D. Pyle. *Data Preparation for Data Mining.* Morgan Kaufmann, 1999.

[86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[87] J.R. Quinlan. *C4.5 Programs for Machine Learning.* Morgan Kaufmann, 1993.

[88] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Fromman-Holzboog Verlag, Stuttgart, 1973.

[89] S.E. Rouwhorst and A.P. Engelbrecht. Searching the forest: Using decision trees as building blocks for evolutionary search in classification databases. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 633–638, Seoul, Korea, 2001. IEEE Press.

[90] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, second edition, 2003.

[91] R.E. Schapire. A brief introduction to boosting. In T. Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1401–1406, Stockholm, Sweden, 1999. Morgan Kaufmann.

[92] H.-P. Schwefel. *Numerical Optimization of Computer Models*. John Wiley & Sons, Inc., 1981.

[93] H.-P. Schwefel. *Evolution and Optimum Seeking*. Wiley, New York, 1995.

[94] C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, 1948.

[95] T. Soule and J.A. Foster. Code size and depth flows in genetic programming. In J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba, and R.L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 313–320, Stanford University, CA, USA, 1997. Morgan Kaufmann.

[96] T. Soule, J.A. Foster, and J. Dickinson. Code growth in genetic programming. In J.R. Koza, D.E. Goldberg, D.B. Fogel, and R.L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223, Stanford University, CA, USA, 1996. MIT Press.

[97] W.A. Tacket. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, 1994.

[98] W.A. Tackett. Genetic programming for feature discovery and image discrimination. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 303–309, University of Illinois at Urbana-Champaign, 1993. Morgan Kaufmann.

[99] M. Vink. Solving combinatorial problems using evolutionary algorithms. Master's thesis, Leiden University, 1997.

[100] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 2000.

[101] L. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

Nederlandse Samenvatting

Sir Francis Bacon zei ongeveer vier eeuwen geleden: "*knowledge is power*" (kennis is macht). Als we kijken naar de huidige maatschapij, dan zien we dat informatie steeds belangrijker wordt. Volgens [73] werd er in 2002 ongeveer 5 exabytes (5×10^{18} *bytes* $\approx$ 600 miljard (dual-layer, single-side) DVD's) aan informatie geproduceerd, waarvan het meeste digitaal is opgeslagen. Dit is meer dan twee maal zoveel als de hoeveelheid informatie die in 1999 werd geproduceerd (2 exabytes). Echter, zoals Albert Einstein opmerkte: "*information is not knowledge*" (informatie is geen kennis).

Eén van de uitdagingen van de grote hoeveelheden informatie die ligt opgeslagen in databases is het vinden van mogelijk nuttige, begrijpelijke en nieuwe patronen in gegevens, die kunnen leiden tot nieuwe inzichten. Zoals T.S. Eliot schreef: "*Where is the knowledge we have lost in information?*" [35] (Waar is de kennis die we zijn kwijt geraakt in de informatie?). Dit is het doel van een process genaamd Knowledge Discovery in Databases (KDD, kennisontdekking in databases). Dit process bestaat uit verschillende stappen: in de Data Mining fase vindt de eigenlijke ontdekking van nieuwe kennis plaats. Het doel van KDD en Data Mining is het vinden van verbanden die gebruikt kunnen worden om voorspellingen te doen op basis van gegevens uit het verleden. Wij richten ons op twee van zulke Data Mining gebieden: classificatie en regressie. Een voorbeeld van een classificatie probleem is of iemand wel of geen krediet moet krijgen bij een bank. Een voorbeeld van een regressie probleem, ook wel numerieke voorspelling genoemd, is het voorspellen van de waterhoogte bij verschillende weersomstandigheden en maanstanden.

Evolutionary computation (evolutionaire rekenmethoden) is een gebied binnen de informatica dat is geïnspireerd door de principes van natuurlijke evolutie zoals die door Charles Darwin zijn geïntroduceerd in 1859 [17]. In evolutionary computation gebruiken we principes uit de evolutietheorie om te zoeken naar zo goed mogelijke oplossingen voor problemen met behulp van de computer.

In dit proefschrift onderzoeken we het gebruik van "tree-based Genetic Programming" (een specifieke vorm van evolutionary computation) voor Data Mining doeleinden, voornamelijke classificatie-problemen. Zo op het eerste gezicht lijkt evolutionary computation in het algemeen, en Genetic Programming in het bijzonder, wellicht niet de meest voor de hand liggende keuze voor data classificatie. Traditionele algoritmen om beslissingsbomen te bouwen zoals C4.5, CART en OC1 zijn over het algemeen sneller. Echter deze algoritmen zoeken/vinden/bouwen beslissingsbomen door middel van lokale optimalisatie. Een voordeel van evolutionairy computation is dat het een globale optimalisator is die beslissingsbomen in hun geheel evalueert, in plaats van iedere knoop/tak/blad apart te optimaliseren. Daardoor zouden evolutionaire algoritmen voor Data Mining beter met attribuut-interactie moeten kunnen omgaan [38, 39]. Een ander voordeel van evolutionary computation is dat het relatief simpel is om een representatie van een beslissingsboom te kiezen, veranderen of uitbreiden. De enige vereiste is een beschrijving van hoe de boom eruit moet zien en hoe de boom geëvalueerd moet worden. Een voorbeeld hiervan is Hoofdstuk 4 waar we onze beslissingsboom-representatie uitbreiden naar "fuzzy" (vage) beslissingsbomen: iets wat veel moeilijker is, of zelfs wellicht onmogelijk, voor algoritmen zoals C4.5, CART en OC1.

Hoofdstukken 2 en 3 richten zich vooral op het effect van de beslissingsboom-representatie (mogelijke keuzes) op het classificatie-vermogen. We laten zien dat door de zoekruimte, het totaal aan mogelijke beslissingsbomen, te beperken de nauwkeurigheid van de classificatie een kan worden verbeterd.

In Hoofstuk 4 beschrijven we hoe we onze algoritmen kunnen aanpassen om "fuzzy" (vage) beslissingsbomen te laten evolueren. Doordat zulke "fuzzy" beslissingsbomen niet beperkt zijn tot beslissingen als *waar* en *niet waar* zijn ze robuuster ten aanzien van fouten in de invoergegevens. Ook kunnen vage "menselijke" begrippen als koud, warm, jong en oud beter in de modellen worden verwerkt.

Behalve de classificatie-nauwkeurigheid zijn ook de snelheid van een algoritme en de begrijpelijkheid van de uitkomst belangrijk bij Data Mining algoritmen. In Hoofstuk 5 laten we zien hoe we deze aspecten van onze algoritmen kunnen verbeteren. Door onze algoritmen geëvolueerde beslissingsbomen te analyseren kunnen we onnodige delen, genaamd *introns*, verwijderen. Hierdoor worden de beslissingsbomen kleiner, en dus begrijpelijker, zonder de nauwkeurigheid van de classificatie aan te tasten. Op deze manier kunnen we ook beslissingsbomen herkennen die syntactisch (uiterlijk) verschillen maar semantisch (qua gedrag) hetzelfde zijn. Door niet meer iedere syntactisch

unieke beslissingsboom te evalueren, maar alleen nog maar alle semantisch unieke beslissingsbomen, kunnen we rekentijd besparen waardoor onze algoritmen sneller worden.

In het laatste hoofstuk richten we onze aandacht op een ander belangrijk onderdeel van evolutionaire algoritmen: de fitness functie. De fitness functie bepaald van iedere mogelijke oplossing hoe goed die oplossing is ten opzichte van andere kandidaat-oplossingen. In Hoofdstuk 6 passen we een adaptieve fitness methode genaamd "Stepwise Adaptation of Weights" (SAW) toe op classificatie en regressie problemen. De SAW techniek verandert de fitness waarden van individuen gedurende het evolutie process zodat moeilijke deelproblemen een zwaarder gewicht krijgen dan makkelijke deelproblemen. We introduceren een nieuwe variant van het SAW algoritme, genaamd *precision* SAW, dat beter werkt voor regressie problemen en we onderzoeken de effecten van de parameters van het SAW algoritme op de werking.

Acknowledgements

First and foremost I would like to thank my parents and brother for their support during the years I worked on this thesis.

Furthermore I would like to thank Tom and Jano for all the work we did together, and Rudy for his help on counting trees.

Last but not least I would like to thank all my fellow PhD students for all the fun we had together, especially my roommates over the years Jano, Martijn, Robert and Peter.

Curriculum Vitae

Jeroen Eggermont is geboren in Purmerend, Noord-Holland, op 27 mei 1975. Van 1987 tot 1993 doorliep hij het atheneum (VWO) op het Alfrink College te Zoetermeer. Van 1993 tot 1998 studeerde hij Informatica aan de Universiteit Leiden. De laatste drie jaar gaf hij als student-assistent ook werkgroepen en practicum-assistentie aan studenten. Vanaf 1998 was hij verbonden aan het Leiden Insititute of Advanced Computer Science waar hij zijn promotie-onderzoek uitvoerde en betrokken was bij onderwijs en contract-onderzoek.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of

Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12