Extracting Fuzzy Rules using Genetic Programming

Rutger ter Borg

August 2000

Acknowledgements

I would like to thank the following persons. Without their help it would not have been possible to complete this thesis work.

Professor Henk Koppelaar for giving me the opportunity to perform my thesis work in the interesting field of Knowledge Based Systems.

My coach Leon Rothkrantz for his hints and useful comments, which made it possible to enhance the quality and readability of this thesis work.

My girlfriend Fleur Brinkman for her love and care and for urging me to work steadily.

My parents for all their support.

My house mates for the needed distraction and for letting me blow off steam.

All authors and co-authors of lil-gp, libmozg and StarFLIP for making free high-quality reusable software libraries.

Abstract

This thesis work focuses on one half of the knowledge acquisition problem for fuzzy systems, namely the acquisition of a fuzzy rule set from a set of input-output data. This acquisition is commonly done by the so called fuzzy rule extraction methods. An example of an existing method for fuzzy rule extraction is the method of Ishibuchi published in [Ishibuchi 1995]. This method suffers from a combinatorial explosion and relies heavily on the generalizing power of a trained artificial neural network.

A novel algorithm is presented, which is called the genetic programming (GP) rule extractor. It makes use of Darwinian evolution to find rule sets which are suited to problem domain data sets. To test its quality a generic applicable method called the *research cycle* has been devised which is able to test the quality of any rule extractor. The research cycle enables a user to create a reference framework for the expected answer of a knowledge state changer such as a rule extractor. The implementation and integration of software representing the research cycle resulted in a workbench. This workbench was able to fulfill the demands of the tasks of the research cycle. It produced results on which conclusions can be drawn about the characteristics of the genetic programming rule extractor and the Ishibuchi rule extractor.

Unlike previous work in the area of rule extraction, the introduced genetic programming extractor is general in its applicability and scales well to problems with high-dimensional input spaces.

Contents

Chapter 1

Introduction

Since their renaissance in the mid-80's, Artificial Neural Network (ANN) techniques have been successfully applied across a broad spectrum of problem domains including in particular pattern recognition. However despite these capabilities, to an end user an ANN is an arcane web of interconnected input, hidden and output units. Moreover an ANN solution manifests itself entirely as a set of numbers in the form of activation function parameters and weight vectors. For many learning tasks, it is important to produce classifiers that are not only highly accurate, but also easily understood by humans. Neural networks are limited in this respect, since they are usually difficult to interpret after training. In contrast to neural networks, the solutions formed by "symbolic" learning systems are usually much more amenable to human comprehension.

There are several reasons why the comprehensibility of induced concept descriptions is often an important consideration. If the designers and end-users of a learning system are to be confident in the performance of the system, they must understand how it arrives at its decisions. Learning systems may also play an important role in the process of scientific discovery. A system may discover salient features and relationships in the input data whose importance was not previously recognized. If the representations formed by the learner are comprehensible, then these discoveries can be made accessible to human review. However, for many problems in which comprehensibility is important, neural networks provide better generalization than common symbolic learning algorithms. It is in these domains that it is important to be able to produce comprehensible concept descriptions.

Recently, many rule extraction methods were published. Rule extraction is the process of finding *if-then* descriptions which represent problems. Most of these published methods employ a neural network for the generalizing capabilities. All neural networks are derived from data related to problems. Thus, these rule extraction methods actually transform a problem to rules with the use of an ANN. Methods capable of forming rule sets directly from problems often lack strong generalizing power. It is questioned whether genetic programming can extract rule sets directly from problems.

A comprehensive literature study has been carried out to the field of genetic programming which is reported in chapter 2. Genetic Programming (GP) was introduced in 1992 by Koza with the publication of [Koza 1992]. GP is distinguished from other evolutionary algorithms in that it evolves flexible-size structured programs, rather than fixed-size linear strings. This representational power combined with the principle of Darwinian evolution allows for applications in very diverse areas, such as fuzzy rule extraction.

A less comprehensive introduction to the field of fuzzy logic controllers is written in chapter 3. Since Zadeh first published his seminal paper [Zadeh] on fuzzy sets in 1965, applications utilizing fuzzy logic have proliferated. Mamdani's development of fuzzy controllers in 1974 gave rise to the utilization of these controllers in ever expanding capacities, particularly in Japan where many industrial processes now employ fuzzy control.

A method has been devised which is capable of judging the quality of any rule extractor. In chapter 4, this framework called the *Research Cycle* is presented. Also, an overview of the research performed is described here. This includes the discussion of Ishibuchi's rule extraction method as well as the design of the genetic programming rule extractor.

A workbench which implements all needed features of the research cycle has been designed and implemented. The design of the workbench, an assessment of available software libraries and the implementation issues are discussed in chapter 5.

Chapter 6 discusses the experimental setup for the rule extraction test suite. This includes the presentation of a number of problem domains which are selected to test the rule extraction methods on. Also, the available parameter settings of implemented program modules including a number of default settings are described. When all parameters are set, the rule extractors are able to produce results of the various data sets which are presented in chapter **??**.

In chapter 7 the conclusions of this thesis work are discussed. Several critics and recommendations are also discussed.

Chapter 2

Genetic Programming

Automatic programming of computers can be seen as the “holy grail” of computer science. The idea is to just tell the computer what to do, and it will be done without the need of writing a program first. Because computer structures are amongst the most complicated objects made by human kind today, it is not relatively straightforward how to automatically make such structures. To go straight to the point, it is not yet possible to automatically make computer programs. For millennia, engineers have been inspired to learn from nature and to apply her recipes in all kinds of technology. This also holds for some computer science fields. For example, a group of algorithms showing analogy with nature are the *Artificial Neural Networks* (ANNs). These are based on the behavior of neurons in the human brain and can be used for a variety of classification tasks, such as pattern recognition, image processing and expert systems.

The part taken from nature which is used for *Genetic Programming* is somewhat older than the theory of neurons. Nature has a robust way of evolving successful organisms, so discovered Darwin back in 1859. Darwin’s writing [Darwin] was a piece of work which was the cause of a turning point in the history of science. Amongst many other things, he argued the following:

> *“...if variations useful to any organic being do occur, assuredly individuals thus characterized will have the best chance of being preserved in the struggle for life; and from the strong principle of inheritance they will tend to produce offspring similarly characterized. This principle of preservation, I have called, for the sake of brevity, Natural Selection.”*

This theory of *natural selection*, also known as *survival of the fittest* governs the *evolutionary adaptation* of the biological world from the smallest virus to the most complicated mammal. Natural selection operates on species through its performance of producing offspring. The more viable the offspring produced by an organism, the more successful the organism.

Offspring is usually the result from the growth of one single cell containing a piece of specific genetic material (DNA), which is provided by the parent(s). In case of two parents, the provided genotype (which is just an other word for DNA) from both parents is merged through sexual recombination. In other cases, mutation (random modification) of the DNA is the *only* process which is responsible for variation of the offspring.

In what way offspring develops depends on the ontogeny. Ontogeny is the link between genotype, the phenotype (the organism’s body and behavior), and the environment in which the organism’s development takes place. Figure 2.1 shows this mechanism and illustrates that ontogeny is a one-way street. Changes in the organism’s DNA can change the organism, but changes in the organism itself do not affect it’s DNA. So, all of the mechanisms of biological heredity take place at the genotype level. And on the other hand, natural selection acts only at the level of the phenotype. Therefore, natural selection only indirectly effects the genotype. The evaluation of the species through its reproductive performance (also called its *fitness*) and the way of its creation of the genetic plan of its offspring are responsible for the *evolutionary adaption* of biological organisms.

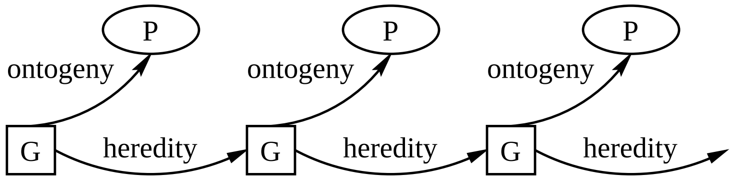

Figure 2.1: The mechanism of Darwinian heredity.

The properties of evolutionary adaptation applied to computer structures resulted in the field of *Evolutionary Computation*[1], which resides under the umbrella of the computer science field *Machine Learning*. The predominant styles of computing approaches found in Evolutionary Computation are briefly sketched below.

- **Genetic Algorithms.** Maybe this is the best known style of evolutionary computation. Introduced in 1975 by Holland [Holland] and followed up by a lot of publications. Genetic Algorithms frequently operate on fixed length character strings representing the structure which undergoes adaptation. Its fitness is determined by a task specific function, also known as the *fitness function,* which interprets the character string as a set of parameters. For example, the path of a classic TSP problem[2] could be considered as a string and the fitness could be determined using the path's length. Sexual recombination, also called *crossover*, is the principal *genetic operator,* while *mutation* often is of secondary importance. A great part of the foundation of Genetic Algorithms are used in Genetic Programming and is explained later.

- **Evolutionary Strategies.** The oldest of the mentioned styles is evolutionary strategies, which dates from 1965 when Rechenberg [Rechenberg] started the field. This style is often associated with engineering optimization problems. The structures used are typically sets of real-valued *objective variables* which are associated with real-valued *strategy variables* in an individual. Fitness is determined by a fitness function using objective variables as parameters. The strategy variables control the way in which mutation varies each objective variable during the production of new individuals. Recombination is usually employed for both objective variables as well as strategy variables.

- **Evolutionary Programming.** Also created in the 60s, evolutionary programming is a predecessor of genetic programming. It uses the mutation operator to change finite state machines (FSM[3]). Mutation is the sole genetic operator employed, containing a significant strategy built into the overall algorithm to direct the mutation in a computationally fruitful direction. The field was founded by Fogel, Owens and Walsh in 1966 with the publication of [Fogel].

- **Genetic Programming.** This is an offshoot of genetic algorithms in which the used computer structures undergoing adaption are computer programs themselves. Specialized genetic operators are used which generalize sexual recombination and mutation for the explanation of genetic programming. Koza was the first to recognize the importance of this method and demonstrated its feasibility for automatic programming in general. In his 1989 paper, he provided evidence in the form of several problems from five different areas. The book [Koza 1992] sparked the rapid growth of genetic programming and remains the standard work in the field.

[1] Sometimes referred to as *Evolutionary Algorithms*. However, *Evolutionary Computation* is the preferred term here due to the naming of the transactions done by the IEEE.

[2] TSP stands for the *Traveling Salesman Problem:* Find the shortest path along a number of cities with exactly one stop at each city.

[3] A *finite state machine* or *finite automation* is a very simple computer program which consists of a machine moving around in a graph of nodes called states.

2.1 Computational problem solving

General computational problem solving is usually considered as the jurisdiction of classical artificial intelligence (AI), a field which is as amorphous as the symbolic representations that dominate its techniques. Problem solving in AI can be done in two ways: by computational models of humans solving problems or by the construction of a program that solves problems as well as or better than humans. Clearly, the techniques of genetic programming and all evolutionary programming fit into the latter methodology of building intelligent programs.

A generic computational problem can be solved by searching through the problem's *state space*[4]. A problem solving program starts at a point in the state space and tries to follow a *path* which leads to a good solution. The way this search is performed (which routes are selected), depends on the *search strategy.* Search strategies can be divided into:

- An **uninformed search.** This means that there is no information available about the number of steps from the current state to the goal state. This method is often referred to as **blind search.** This kind of search only works if the solution space is small. For nearly all interesting learning domains, the search state space is far too large for blind search to be completed in reasonable time frames.

- An **informed search.** Also called heuristic search strategies, these kind of strategies consider what path is best or cheapest to reach their goal. Mostly, they search more effective than uninformed search methods. *Hill climbing* is one of these techniques. It starts at one spot of the state space, transforms that solution, and keeps the new solutions if it is a better solution. Otherwise the new solution is often (although not always) discarded and the original solution is again transformed, evaluated and discarded or kept until a termination criterion is met. *Simulated annealing* and many *artificial neural networks* training algorithms are typical of this approach. Only one solution is considered at a time and only one path through the state space is explored.

- A **beam search.** Both foregoing search algorithms are single point-to-point searches in the state space. A beam search maintains a *population* of search points in the state space. Beam search is a mixture of the first two search methods mentioned. It uses some sort of *evaluation metric* to select out a number of most promising solutions for further transformations. All others are discarded and doing so all solutions are kept inside the *beam.* Beam search has been a well-established technique in heuristic machine learning systems [Langley]. It was until 1993 that Angeline recognized in [Angeline] that GP is a form of beam search because it retains a population of candidate solutions that is smaller than the set of all possible solutions. Similarity between machine learning and GP has been found since the *beam* of machine learning can be compared to the *population* in GP and the *evaluation metric* is called *fitness function.*

2.2 Genetic Programming in detail

Genetic programs are often said to be distinct from simple genetic algorithms due to the dynamic nature of their representation and their interpretation as programs. Simple genetic algorithms [Holland], which use fixed length strings, n-point crossover and point mutation are in principle both representationally and operationally general. In fact, there is no theoretical reason why simple genetic algorithms could not construct genetic programs with similar properties. However, as so often happens in science, the practice of genetic algorithms focuses on only a small portion of what is possible.

Genetic algorithms (with the inclusion of genetic programming) uses a direct analogy of natural behavior. They work with a *population* of "individuals", each representing a possible solution to a

[4]The *state space* is a representation of *all* possible representations of the environment to a computational problem.

given problem. To each individual is assigned a "fitness score" according to how good a solution it is to the problem. For example, the fitness score might be the mean square error (of course, the lower, the better) to training data in case of symbolic regression[5]. In nature this is equivalent to assessing how effective an organism is at competing for resources. The highly fit individuals are given opportunities to "reproduce", by "cross breeding" with other individuals in the population. This produces new individuals as "offspring", which share some features taken from each "parent".

A whole new population of possible solutions is thus produced by selecting the best individuals from the current "generation", and mating them to produce a new set of individuals. This new generation contains a higher proportion of the characteristics possessed by the good members of the previous generation. By favoring the mating of the more fit individuals, the most promising areas of the search state space are explored.

The following features are shared by most genetic programming systems: *stochastic decision making, program structures, genetic operators* and a *simulated evolution of a population.* These features will be discussed successively.

2.2.1 Stochastic decision making

GP uses pseudo-random numbers to mimic the randomness of natural evolution. As a result, GP uses stochastic processes and probabilistic decision making at several stages of program development. The subjects of randomness, probability and number generation are mostly discussed in stochastic decision making. The subjects of permutation, combination, variation and so forth are available for a more detailed review in [Ross].

In GP, it is often necessary to estimate the size of the search state space and the difficulty of the problem being solved. The search state space in GP is basically the set of all programs that could be found by the GP algorithm, given a certain programming language.

2.2.2 Program structures

The program structures in GP are the *individuals* of the population. It's building blocks are:

- A **terminal set.** This is the basic input to a GP program, which can best be described as the leaves of every branch of a tree, and in which the tree is the program (in case of a tree-based representation). The output of a terminal (which is the input to the program) can function as an environment sensor, as a constant of any kind or it can be just plain empty. To put it in another way, a terminal node has an arity[6] of zero.

- A **function set.** These are the statements, operators and functions available to the GP individuals. The range of available functions is *very broad.* After all, this is genetic *programming. Any* function used in *any* computer program can be utilized. Examples of functions vary from simple Boolean functions such as `AND`, `OR` or `NOT` to user defined subroutines such as `read sensor`, `turn left`, `turn right` and `move ahead`.

The functions and terminals used for a GP system should be powerful enough to be able to represent a solution to the problem. For example, a function set consisting only of the `ADD` operator will probably not solve many interesting problems. On the other hand, it is better not to use a too large function set. This enlarges the search state space and can sometimes make the search for a solution harder. An approximate starting point for a function set might be the arithmetic and logic operators:

```
+, -, *, /, OR, AND, XOR
```

The range of problems that can be solved using these functions is astonishing (but no great surprise if the capabilities of the early computers are taken into account). These previously mentioned

[5] *Symbolic regression* is the art of finding a mathematical function which fits best to a set of numbers.
[6] The *arity* is the number of inputs to a function.

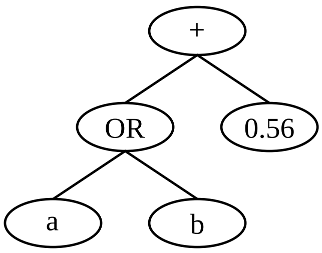

Figure 2.2: Koza's tree-based program structure.

primitives of GP, *terminals* and *functions*, are not programs. Functions and terminals must be assembled into a *structure* before they may be executed as programs. The evolution of programs is, of course, common to all genetic programming. Programs are structures of functions and terminals together with rules or conventions for when and how each function or terminal has to be executed.

Execution and memory locality are phenomic issues, these are issues regarding the *behavior* of the program (analogous to the mechanism shown in figure 2.1 on page 7). In the most used GP systems, the tree-based GP system, there is no separate phenotype. Therefore, in this case it seems that structural issues of execution, memory and variation are the same. But similarity exists only because of an implicit choice to blend the genome and the phenome. Other approaches are also possible. For example, Gruau used a tree-based structure as genome in [Gruau], and an artificial neural network as phenome.

In a program structure, the previously mentioned terminals and functions are tied together to form a *variable* length program which depends on the chosen program structure. The two most common used program structures are the following:

- A **tree-based representation.** This is the most used program structure representation. In [Koza 1992], Koza discusses why he has chosen the programming language LISP. Koza's choice is based on arguments like "both data and programs having the same form" and "the convenient handling of hierarchical structures". LISP uses S-expressions, in which a simple function like $(a \cup b) + 0.56$ is expressed `(+ (OR a b) 0.56)`. Figure 2.2 shows this expression in an explicit tree form. All ovals in figure 2.2 are terminals with arity zero or two. Execution of LISP structures are pretty straightforward, it proceeds by repeated evaluation of the leftmost terminal for which all inputs are available. Local memory is built into the tree structure itself. For example, the values of `a` and `b` are local to the `OR` node in figure 2.2.

- A **graph-based representation.** On the fundamental program structures, graphs are the newest arrival. PADO[7] is the name of the first graph-based GP system and was introduced by Teller and Veloso in 1995 for a system which was capable of difficult image classification. Curiously enough, the name PADO does not have anything to do with the fact that graphs are used for evolution. Graphs are capable of compactly representing very complex program structures. Very easy program structures, which have a sequential order of functions, can also be represented. For example, a program that computes and writes the successive elements of the Fibonacci series[8] is shown in figure 2.3. Unlike tree-based program structures, memory is not built into the graph structure. So, external memory must be available to the graph to be able to operate. In order to execute correctly, there are two actions every node in the graph has to perform. First, it must perform some function on the memory (such as `read`, `write`, `pop` or `push`). Second, each node has to decide which node will be the *next* node to execute. For example, all arcs in figure 2.3 should be initialized at 1 for the Fibonacci series to compute correctly. This could be done using a global memory array with length 5. Execution would constantly switch between the $\times$ and the $+$, starting at the $\times$, and table 2.1 shows the result of doing so. Arcs in figure 2.3 are numbered from left to right, arc 1 is connected

[7] *PADO* stands for Parallel Algorithm Discovery and Orchestration.
[8] The *Fibonacci series* is defined to be $Fib_n = Fib_{n-1} + Fib_{n-2}$ with $Fib_0 = Fib_1 = 1$.

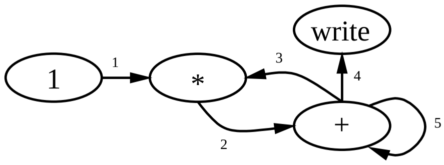

Figure 2.3: A graph-based structure.

Table 2.1: The result of the GP program shown in figure 2.3.

t	arc 1	arc 2	arc 3	arc 4	arc 5
0	1	1	1	1	1
1	1	1	2	2	2
2	1	2	3	3	3
3	1	3	5	5	5
4	1	5	8	8	8

to the 1-node and arc 5 is the looping arc. All values of arc 4 represent the solution to the Fibonacci series and are stored by the `write` node.

More difficult problems are hard to solve with the standard GP structures and progress on this front is topic of current GP research. What still is needed, is automatic discovery of problem representations which make a given problem more traceable. This is analogous to the creation of subroutines or sub-functions when this is done by a human programmer. The same process can also be viewed as finding a way to decompose a problem into smaller, potentially more easily solvable subproblems. For the tree-based program structure, two approaches have been examined to automatically perform these functions, generically known as *automatic function definition.* Koza's second book, [Koza 1994], is almost entirely dedicated to the first of the two approaches, Automatically Defined Functions (ADF). The tree-based structure is altered and now makes a distinction between a *result-producing branch* and a *function-defining* branch. With ADF it is possible, but not necessary, to define how many subroutines with how many parameters should evolve.

The second approach is Module Acquisition by Angeline published in [Angeline]. Module Acquisition creates a library of unique models, which are globally defined, and which essentially extend the function set for all individuals. Every so often, a module is *acquired* by selecting a sub-tree within an existing individual and making it into a globally defined module. A comparison of the two approaches can be found in [Kinnear]. This article showed that ADFs are substantially better than MAs in some cases. A point is that these approaches only speed up the evolutionary process, it can not do any more or less than the original program structures.

Other representations are also possible. In theory, a program structure of a GP system can be *any* possible computer program. For example, the program structure could vary from an artificial neural network (shown in [Gruau]) to native machine code (demonstrated in [Nordin 1994]). All machine learning systems run on computers. This means that all other systems of machine learning may be represented as a computer program. Therefore, theoretically it is possible to represent *all* other machine learning systems using GP. Thus, GP is a *superset* of all other machine learning representations.

2.2.3 Genetic operators

GP transforms the initial programs in the population using genetic operators. As illustrated by figure 2.1 on page 7, genetic operators operate on the genome while the natural selection is done through the performance of it's phenome. Because no distinction is made between genome and phenome in most GP systems, most genetic operators are not defined to operate on a genome or phenome. The most common genetic operators used are discussed below.

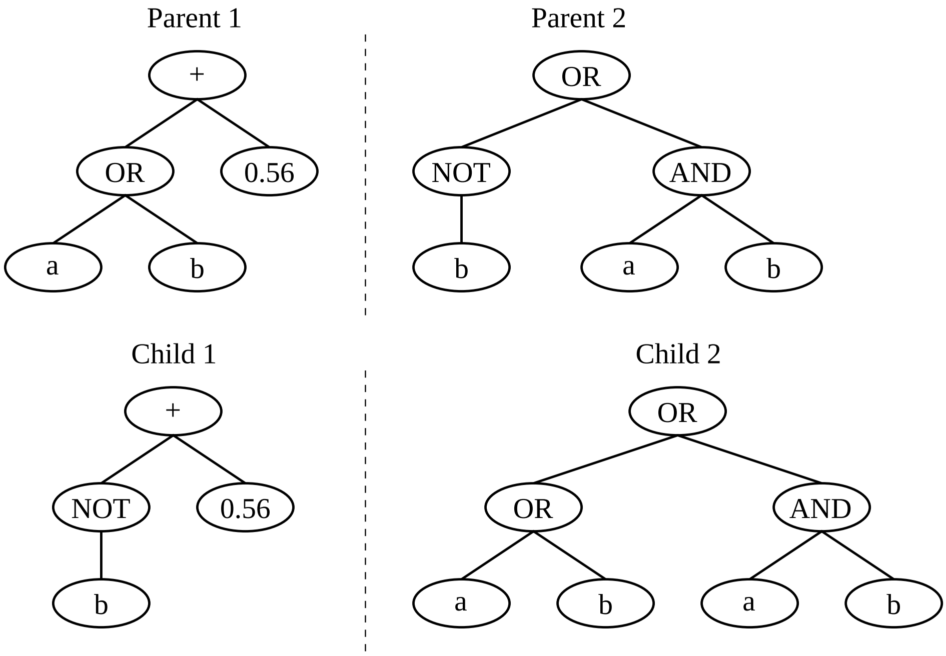

Figure 2.4: Tree-based crossover.

- **Crossover.** The crossover operator combines the genetic material of two parents by swapping a part of one parent with a part of the other. Different crossover operators work on different genotypes. Therefore two different crossover operators will be presented, one for tree-based and one for graph-based program structures.
  - Tree-based crossover is described graphically in figure 2.4. The basic tree-based crossover operator can be described in three steps. To start, choose two individuals as parents, based on a mating selection policy. The next step is to choose a random subtree in each parent. The selection of these subtrees are often biased so that subtrees constituting terminals are selected with lower probability than other subtrees. In figure 2.4, the selected subtree of parent 1 is `(OR a b)` and parent 2 contributes `(NOT b)`. Finally, the selected subtrees are swapped between the both parents. The resulting individuals are the children and are also shown in figure 2.4.
  - The graph-based crossover is somewhat more complicated. The following procedure is employed by Teller and works in four steps. First, choose two individuals as parents, based on mating selection policy. Then, divide each graph in two node sets (draw a line through a graph-based program and two parts remain, with "loose" arcs from both parts). The third step is to swap the selected fragments between the two parents. The last thing is to recombine all "loose" arcs randomly so that every arc is connected. If this procedure is executed correctly valid graphs will be generated as offspring.
- **Mutation.** Mutation operates on only one individual. Normally, after crossover has occurred, each child produced by the crossover undergoes mutation with a low probability. A graphical example of a tree-based mutation is shown in figure 2.5. It is based on figure 2.2 on page 10. The `(b)` is replaced by a randomly chosen subexpression `(NOT b)`. In a tree-based GP system, if a subtree mutates, it is replaced by a structure which is similar in size and shape. The same holds for a graph-based GP system.
- **Permutation.** Permutation is done by rearrangement of the input to a terminal and also operates on one individual. Suppose a terminal has an arity of three and that the inputs

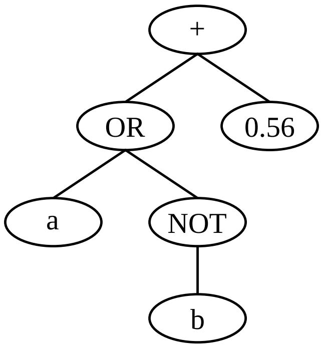

Figure 2.5: Tree-based mutation.

are called n_0, n_1 and n_2. Then the permutation of these inputs will be done by choosing one of the 3! possibilities of rearrangements. For example, permutating could result in the function being called with n_2, n_1 and n_0. This way of permutating also works for graph-based structures.

- **Editing.** Genetic programs suffer from introns. Examples of introns in genetic (tree-based) program structures are `(AND X X)`, `(OR X X)` and `(NOT (NOT X))`, these are all the same as `(X)`. An individual's program structure is regarded as the genotype and it's behavior as the phenotype. Since fitness is determined by the phenotype, introns have no effect on the individual's fitness. *Editing* removes introns from the program structure and Koza used it since [Koza 1992]. However, recent studies reveal that introns do affect the likelihood that the individual's descendents will survive. Thus, introns may not be as useless as they seem. The theoretical and experimental evidence today supports the hypothesis that introns emerge principally in response to the frequently destructive effects of genetic operators (see [Nordin 1996]).

- **Darwinian reproduction.** The *reproduction* operation for genetic programming is the basic engine of Darwinian natural selection and survival of the fittest. This operator is pretty straightforward. A fitness based selected individual is copied and the copy is placed into the next generation.

These genetic operators are still the beginning of what is expected to come. A problem of many of the previously defined genetic operators is their element of randomness. A crossover *randomly* chooses its crossover point, which is computationally *and* evolutionary not efficient. Recent years have seen a growing interest in genetic operators that are responsive to the needs of the population they act upon. An operator can be programmed, like the discussed ones, or can be learned. Teller [Teller] describes a procedure for the automatic design and use of new genetic operators. These SMART (genetic) operator programs are co-evolved with the main population of programs and learn to recombine the main population better than random genetic recombination. Other approaches, like adaptive recombination guided by semantic value measures (on S-expressions) are explored in [Iba 1996].

2.2.4 Simulated evolution of a population

Of course, GP must choose which members of the population will be subject to genetic operators as mentioned before. In making that choice, GP implements one of the most important part of its model of organic evolutionary learning, *fitness based selection.* Doing a beam search, fitness based selection affects both the ordering of the individuals in the beam and the contents of the beam. GP's evaluation metric is called a *fitness function* and the manner in which the fitness function affects the selection of individuals for genetic operators may be referred to as the GP selection algorithm. Fitness functions are very problem specific. Many GP systems suffer from premature convergence[9] so no global optimum is reached, often due to an unsuited fitness function. In [Banzhaf], the following definition is given:

> **Fitness** is the measure used by GP during simulated evolution of how well a program has learned to predict the output(s) from the input(s) - that is, the features of the learning domain.

To standardize the fitness function itself, more subtypes of fitness functions are defined:

- A **continuous fitness** function is capable of distinguishing individuals which perform almost equally.
- A **standardized fitness** function is a fitness function in which zero is the value assigned to the fittest individual.
- A **normalized fitness** function has its outcome in the range of $[0, 1]$.

A popular application of GP is symbolic regression. The learning domain of such problem is well defined, and a capable fitness function in this case is a *squared error fitness function.* For example, consider a training set of values of $\cos 2x$ on $x \epsilon [0, 2\pi]$. Suppose the training set consist out of n examples and let the output of the ith example in the training set be o_i and define the output of an GP individual p_i. In that case, the squared error fitness function is $f_p = \sum_{i=1}^{n}(p_i - o_i)^2$, which is a continuous, standardized and normalized fitness function. Koza investigated this in [Koza 1992] in the hope to auto-discover the trigonometric identity $\cos 2x = 1 - 2\sin^2 x$. In his experiment, the best-of-generation individual had a near perfect (almost zero) fitness, but did not hold the wanted identity. Instead, the system found an other identity: $\cos 2x = \sin(\frac{\pi}{2} - 2x)$.

For non-numeric domains, it is often possible to cast a fitness function onto the learning problem domain. Examples of this are:

- The number of correctly classified examples in a pattern classification task.
- The money won by a GP-controlled agent in a betting game.
- The amount of food found and eaten by an artificial agent in an artificial life application.
- The time of travel or the cost of the trip in a route optimization problem.

Because it is possible that the population is trapped in local minima, the fitness function is actively being researched and improved. Nikolaev introduces in [Nikolaev] a new fitness function which is based on analogy with a model of the biological immune system, in a way that the programs are viewed as lymphocyte clones that compete to recognize most of the examples, viewed as antigens. Nikolaev gives proof that this version is an improvement for diversity of the population and robustness of the system. Iba uses a fitness function based on the Minimum Description Length principle in [Iba 1994], which has a slight advantage over the standard fitness functions.

Another approach is to combine very different concepts in the fitness criteria. If different properties of the phenotype are used (such as length of the program and execution speed) in the fitness function, then such a function is referred to as a *multi-objective fitness function.* Very recent publications in this particular field [Zhang] show that Zhang's *co-evolutionary fitness switching* is capable of outperforming every aspect of traditional fitness functions. It seems that co-evolutionary fitness switching has the effect of avoiding over-fitting to one specific behavior. Also, it is possible to define a pool of fitness functions which are defined to reflect the problem structure but without too much need for domain knowledge.

After the quality of an individual has been assessed by applying a fitness function, the decision has to be made whether to apply genetic operators to that individual and whether to keep it in

[9] This means that the beam of the beam search is becoming too narrow too fast and the GP system is stuck at a local optimum.

the population or allow it to be replaced. This task is called selection and is assigned to a special operator, the selection operator. There are various different selection operators, and a decision about the method of selection to be applied under particular circumstances is one of the most important decisions to be made in a GP system. Selection is responsible for the speed of the evolution and is often cited as the culprit in cases where premature convergence stalls the success of an evolutionary algorithm. A few selection algorithms are presented here.

- **Fitness-proportional selection**. This type of selection is the most used one and is also used in the GA scenario (for a long time, it was already used in [Holland]). It gives every individual an chance to pass offspring into the next generation. An individual i is given a probability of $p_i = \frac{f_i}{\sum_j f_j}$ for being able to take part in the next generation (with f_i being it's fitness, calculated by it's fitness function).

- **Ranking selection**. Ranking selection is based on the fitness order, into which the individuals can be sorted. The selection probability is assigned to individuals as a function of their rank in the population. Linear and exponentional ranking functions can be utilized to determine the probability for selection for every individual.

- **Tournament selection**. A number of individuals, called the tournament size, is selected randomly for a selective competition. The "winners" of the tournaments are selected for having offspring. In the smallest possible tournament two individuals compete. Tournament selection has recently become a mainstream method for selection. The main reason for this is that it does not need a centralized fitness comparison between individuals. This does not only accelerate evolution considerably, but also provides an easy way to parallelize the algorithm. With fitness-proportional selection, the communication overhead between evaluations would be rather large.

A big contributor to premature convergence problem is the selection algorithm. The degree to which a population is directed tends to reduce the diversity in it, and for this reason elitism, the use of only the top percentage of the population for breeding. It tends to be overlooked when searching for a method of avoiding premature convergence. Lately, many new selection algorithms have been subject to research and good progress is made. The Pygmy Algorithm, which uses disassortative mating, taken directly from population genetics, has shown in [Ryan] to allow stronger selection pressure and the use of small population at very little extra computational cost and with little risk of losing genetic material.

Chapter 3

Fuzzy Rules

Fuzzy systems are very popular in Japan. During an international meeting of fuzzy researchers in Tokyo in 1987, Takeshi Yamakawa demonstrated the use of fuzzy control in an "inverted pendulum" experiment, which is a classical control problem in which a vehicle tries to keep a pole mounted on it's top by a hinge upright by moving back and forth. Observers were impressed by this demonstration, as well as later experiments by Yamakawa in which he mounted a live mouse on top of the pendulum and the system kept stability.

The enthusiasm of the Japanese for fuzzy logic is reflected in the wide range of other applications they have investigated or implemented: character and handwriting recognition, optical fuzzy systems, robots, voice-controlled robot helicopter, control flow of powders in film manufacture, elevator systems, and so forth.

Work on fuzzy systems is also proceeding in the US and Europe, though not with the same enthusiasm shown in Japan. The US Environmental Protection Agency had investigated fuzzy control for energy-efficient engines and NASA has studied fuzzy control for automated space docking: simulations that show that a fuzzy control system can greatly reduce fuel consumption. Firms as Boeing, General Motors, Allen-Bradley, Chrysler, Eaton, Whirlpool and Siemens have worked on fuzzy logic for use in low-power refrigerators, improved automotive transmissions and energy-efficient electric motors.

3.1 Fuzzy Logic Control

The fuzzy set theory introduced by Zadeh in [Zadeh] provides a foundation in which uncertainty can be expressed in a mathematical way. One fundamental difference between fuzzy logic and traditional logic is the possible truth values assigned to an element of a set. In the classical set theory a subset $B \subset S$ is defined by $\forall x \in S$: $x \mapsto \{0, 1\}$, where $x \mapsto 1$ means $x \in B$ and $x \mapsto 0$ means $x \notin B$. A fuzzy subset $F \subset S$ is represented by $\forall x \in S$: $x \mapsto [0, 1]$ with the meaning of 0 and 1 unchanged, but values in-between represent intermediate degrees of membership. This kind of mapping is usually described by the *membership function* of F, often denoted by $\mu_F(x)$. The symbol μ is standard fuzzy logic nomenclature for *truth value.*

For an example of a membership function, consider the statement "this person is of average length", and suppose an *average* person's length is 1.72 meters, as stated in [Haak]. A length of 1.58 meters is considered a *short* person and if someone reaches the length of 1.86 meters it is classified as *long.* Figure 3.1 illustrates a fuzzy as well as a Boolean membership function of the statement "this person is of average length". The fuzzy, triangular function is the most widely used in fuzzy logic because of its simplicity. Other formed functions such as a bell-shape or a trapezoidal shape are also found. The dotted line in figure 3.1 represents a Boolean membership function, in which only values zero and one may be used.

Fuzzy control is one of the more active areas of application of fuzzy logic and the underlying fuzzy set theory. Processing of uncertain information and saving energy using common-sense rules

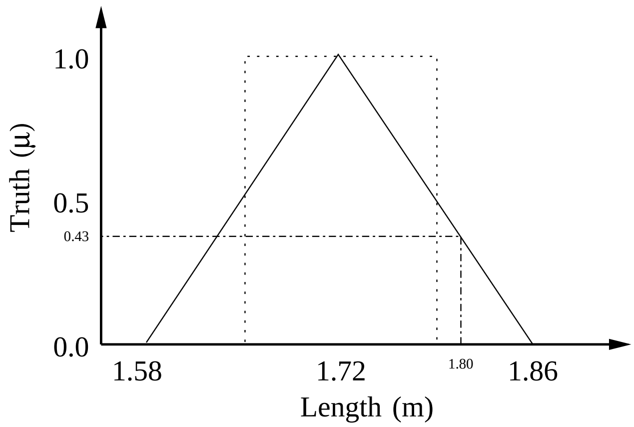

Figure 3.1: *Average length* membership functions.

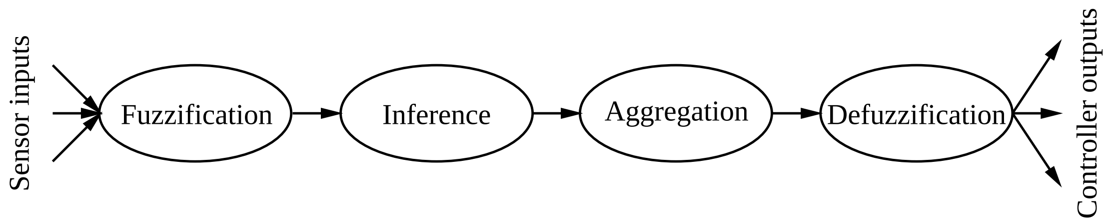

Figure 3.2: Processes of a fuzzy controller.

and linguistic statements are the basis for fuzzy logic control. Fuzzy controllers are very simple conceptually and consist out of three stages: an input stage, a processing stage and an output stage. The input stage maps sensor or other inputs to the appropriate membership functions and truth values, which is referred to as *fuzzification*. The processing stage consists out of two parts, *inference* and *aggregation*. Inference invokes each appropriate rule and generates a result for each, then aggregation combines the results of the rules. Finally, the output stage converts the combined results back into a specific control output value through *defuzzification*. A graphical representation of these processes of a fuzzy controller can be observed in figure 3.2 and all processes are discussed in more detail below.

- **Fuzzification.** This stage maps all input values to truth values using membership functions. For example, if a person's length is measured at 1.80 meters, the fuzzy membership function *average length* would result in a truth value of 0.43. This process is done for all membership functions, all further calculations are based on truth values.

- **Inference.** Given mappings of input variables into membership functions and truth values, a fuzzy logic controller has to determine the output based on a collection of logic rules in the form of `IF-THEN` statements. In this process, inference calculates the outcome of each single rule. A skeleton fuzzy rule has the form `IF` *antecedent* `THEN` *consequent*, in which both the antecedent and the consequent parts of the rule are constructed with terms like `x IS A`. Antecedents can contain multiple terms, which are combined using fuzzy operators such as `AND`, `OR` and `NOT`. The most popular definition of these operators are to take the minimum truth value for `AND`, the maximum for `OR`, and $(1 - \mu)$ for `NOT`. In several different ways the result of a rule can be computed, but the most common and simplest is the so-called *max-min* inference method, in which the output membership function is given the truth value generated by the premise. For example, consider the following rule for automatic car seat adjustment:

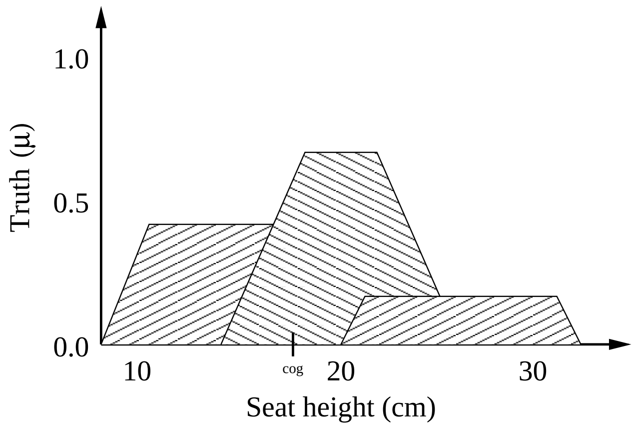

Figure 3.3: Centroid defuzzification.

```
IF length IS average THEN seat height IS medium
```

Using the membership function from figure 3.1, if a person's length is measured at 1.80, the premise has a truth value of 0.43. Using *max-min* inferencing, `seat height IS medium` is given the same truth value of 0.43.

- **Aggregation.** If multiple rules fire on the same consequent linguistic variables, the aggregation operator defines the combinatorial result. Most commonly this is done through a simple fuzzy `OR` operator. So, if multiple rules give truth values for `seat height IS medium`, the maximum is taken.

- **Defuzzification.** The final stage maps truth values on output values and can be realized in several ways. One common method is the *centroid* or *center of gravity* method. This method uses all mapped output truth values on the linguistic output variables to determine a trapezoidal "weight block". Figure 3.3 illustrates the combination of the weight blocks to find the output value. Another defuzzification operator is the *height* method, which favors the biggest contributing output value.

An example of a practical fuzzy logic controlled system is an industrial air conditioner designed by Mitsubishi which uses 25 heating rules and 25 cooling rules. A temperature sensor provides input, with control outputs feds to an inverter, a compressor valve and a fan motor. Compared to the previous design, the fuzzy controller heats and cools 5 times faster, reduces power consumption by 24%, increases temperature stability by factor 2 and uses fewer sensors.

3.2 α-Cuts

The theory of α-cuts is needed for Ishibuchi's projection of fuzzy numbers through a neural network, which is described in section 4.2.3. A graphical representation of such a projection through an ANN is presented in figure 4.7 on page 25. As described in [Kaufmann], α-cuts are invented to do calculations on multidimensional fuzzy numbers. Consider the fuzzy input vector (*large, medium*) as shown in figure 4.7, which can be represented as a 3-dimensional shape of a half pyramid. An α-cut is the "slice" of this half pyramid at a truth value α, which results in a rectangular shape. An example of an α-cut of (*large, medium*) with an α of 0.33 is shown in figure 3.4. If a random point from inside the rectangular from figure 3.4 is sent through a neural net, then the mapping will be valid for (*large, medium*) and will have a truth value of 0.33. If a mesh is generated with the same shape as the rectangular and this mesh is projected on the output vector, the result will be the output vector at truth value 0.33. If the same type of projections are

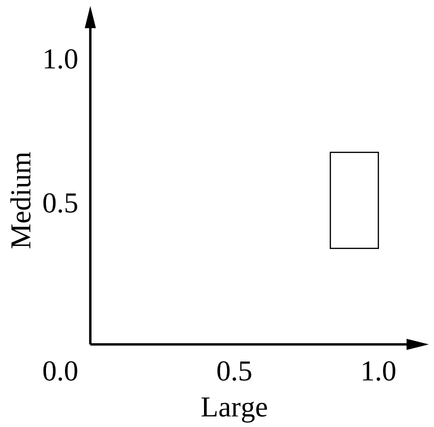

Figure 3.4: An α-cut.

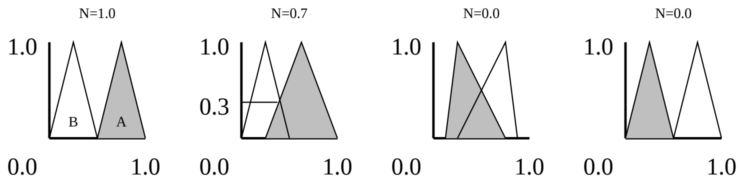

Figure 3.5: Necessity grade of $A > B$.

made for α's ranging from 0 to 1 with some step size, a projection of a fuzzy input vector through the neural net is approximated. Figure 4.7 illustrates such a projection.

3.3 Necessity Grade

Also needed for Ishibuchi's fuzzy rule extraction algorithm is the concept of the necessity grade in the ranking of fuzzy numbers as described in [Dubois]. It is used to make decisions in accordance to the rank of fuzzy numbers. A graphical exemplification of the necessity grade of $A > B$ is shown in figure 3.5, the grey membership function is A and the white is B. The values of necessity of $A > B$ are shown figure 3.5 and are from left to right 1.0, 0.7, 0.0 and 0.0. In other words, the necessity grade is the truth value of $A > B$. The necessity grade of the first (most left) example of figure 3.5 can be compared to the truth value of 1.0 for the statement $large > small$. With A being $large$ and B representing $small$, the necessity grade of this first example is 1.0. In the next example A is not totally larger than B thus the overlap value of 0.3 is subtracted from 1.0 to calculate the necessity grade of 0.7. In figure 3.5's third example A and B show overlap, but B is always larger than A so no truth can be assigned to the statement $A > B$, which results in a necessity grade of zero. The same holds for the last example to which also a value of 0.0 is given to the necessity grade.

Chapter 4

Research Cycle

One of the key aspects of artificial intelligence research is finding methods to transform implicit knowledge to explicit knowledge. Implicit knowledge can not easily be understood by humans, examples of this are data sets and artificial neural networks. The opposite, explicit knowledge, is knowledge represented in a human comprehensible way. The explicit knowledge description which is used here, takes the shape of a *fuzzy rule set*. A fuzzy rule set is a collection of symbolic *if-then* rules which generic appearance is described in section 3.1. *Fuzzy rule extraction* is the transformation from an implicit knowledge state to a fuzzy rule set. Most existing rule extraction methods operate on either a data set or an artificial neural network.

The ability is needed to make assessments on the quality of any rule extractor. The generic transformation all rule extractors perform is from a data set to a rule set. If an ANN is used by the extraction method, it is regarded as part of the extraction algorithm. Some rule extractors utilize ANNs as part of their extraction algorithm, some don't. The consideration that the ANN "belongs to" the extraction algorithm enables the design of a generic framework.

A method has been devised by which it is possible to test and compare any rule extraction algorithm. Data sets and fuzzy rules are regarded as *knowledge states*. A rule extractor can be viewed as a *state changer*, which changes a data set to a fuzzy rule set. The inverse of the rule extractor is a data set generator. These knowledge states and state changers are graphically represented in figure 4.1. Due to the circular shape of figure 4.1, it is named the *Research Cycle*. The method is capable of judging the quality of any rule extraction method, even without a comparison to other rule extractors. This assessment is possible if the research is started in the fuzzy rules knowledge state. A user can pre-define a fuzzy rule set of which a projection[1] is known. After changing from knowledge state twice, the extracted fuzzy rules must have a similar mapping. The more the extracted mapping matches the original mapping, the better a rule extraction method is. So, it is possible to make statements about the quality of rule extractors by measuring the differences between mappings of fuzzy rules before and after a circle motion has

[1] A fuzzy controller is able to make a continuous projection on the basis of the rule set the controller operates on.

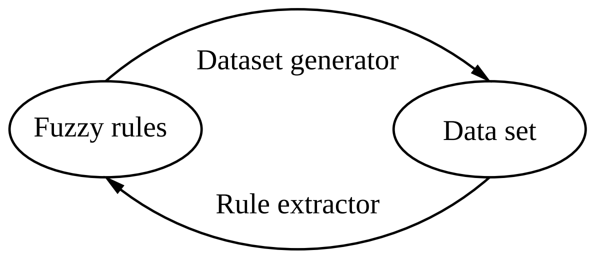

Figure 4.1: Research Cycle.

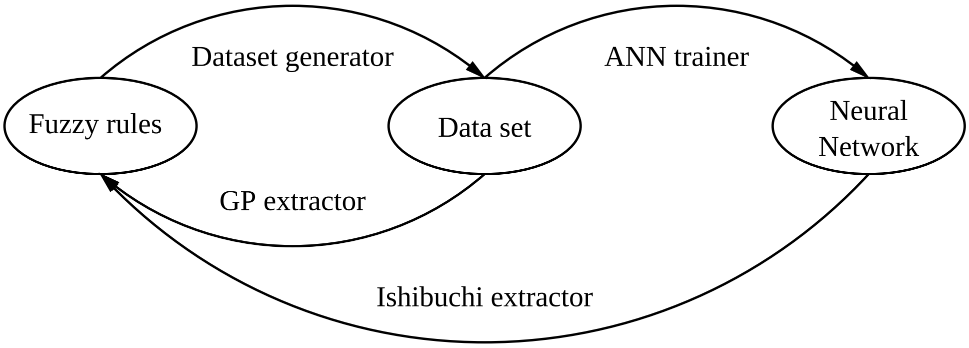

Figure 4.2: Performed research.

been completed. The actual investigations done in this work are derived from the research cycle and is presented in figure 4.2. Two fuzzy rule extraction methods are tested on the basis of the research cycle, namely the genetic programming extractor and the Ishibuchi extractor.

The rest of this chapter is divided into three parts. The first two are analogous to symbols used in figure 4.2. Started is at the subject of *knowledge states* in section 4.1 which are represented by ovals in figure 4.2. The second group of discussed subjects, the *state changers*, are characterized by edges in figure 4.2 and discussed in section 4.2. Finally, it is needed to have background information on a classification scheme for rule extractors. For this, a frame work is presented in section 4.3 which is called *rule extractor classification framework*.

The knowledge states section starts with describing the data set in subsection 4.1.1. Subsection data set is followed by a description of the representation of a fuzzy rule set in subsection 4.1.2. The last knowledge state presented in subsection 4.1.3 is embodied in a short introduction to artificial neural networks. State changers are capable of transforming one knowledge state to another. Subsection 4.2.1 discusses a simple procedure for generating a justified data set out of a given fuzzy rule set. How to train a feed-forward neural network is presented in subsection 4.2.2. A method for rule extraction from neural networks as described in [Ishibuchi 1995] is discussed in subsection 4.2.3. Its opponent, the genetic rule extractor in subsection 4.2.4 is designed on the basis of its foundation surveyed in chapter 2.

4.1 Knowledge states

4.1.1 Data set

For the rule extractors and neural network to work, a mathematical space representation of a data set is needed. A data set consists out of n attributes which is also referred to as a n-sized input vector $\vec{x}$. Each n attributes are mapped onto an accompanying class out of m possible classes. These m classes are also named the m-sized output vector $\vec{o}$. The data sets are capable of making two different types of projections, a Boolean type and a fuzzy type. Both are discussed successively below.

- **Boolean projection:** $[0,1]^n \mapsto \{0,1\}^m$**.** Many data sets from the UCI machine learning repository are of this form. Almost all rule extraction literature refer to one or more data sets of the UCI machine learning repository which is maintained by Blake and can be found on the Internet at [Blake]. This is a repository of databases, domain theories and data generators that are used by the machine learning community for the empirical analysis of machine learning algorithms. Each problem domain contains a description and an actual data set. The repository problem domain data set size characteristics vary from a few attributes to nearly 1560 and from few instances to an impressive $67,500$. Subjects of problem domains range from web server statistics to character recognition to heart diseases.

$$\begin{aligned}
\texttt{(5.1,3.5,1.4,0.2)} &\longmapsto \text{Iris Setosa}\\
\texttt{(7.0,3.2,4.7,1.4)} &\longmapsto \text{Iris Versicolor}\\
\texttt{(6.2,3.4,5.4,2.3)} &\longmapsto \text{Iris Virginica}
\end{aligned}$$

Figure 4.3: Original projection.

$$\begin{aligned}
\texttt{(0.22,0.63,0.07,0.04)} &\longmapsto \texttt{(1,0,0)}\\
\texttt{(0.75,0.50,0.63,0.54)} &\longmapsto \texttt{(0,1,0)}\\
\texttt{(0.44,0.42,0.69,0.71)} &\longmapsto \texttt{(0,0,1)}
\end{aligned}$$

Figure 4.4: Transformed projection.

Consider the Iris plant data set which consists out of 4 input attributes and 3 possible output classes. The original Iris data set file from [Blake] has entries like the ones shown in figure 4.3. The first four numbers are values of the input attributes sepal length, sepal width, pedal length and pedal with. The plant type name on which the input vector is mapped on is shown on the right en can be either Iris Setosa, Iris Versicolor or Iris Virginica.

It is desirable that all input points reside within a hypercube, therefore all input points p must undergo the transformation $p_{new} = \frac{p_{old} - p_{min}}{p_{max} - p_{min}}$ which standardizes all Iris plant attributes to fit within $[0,1]^4$. In this transformation, p_{new} stands for a point after transformation, p_{old} stands for a point before transformation, p_{min} is a composite point of which all elements hold the minimum value of the data set and p_{max} is like p_{min} but with maximum values. The output vector must also reside within a hypercube. The UCI data sets make use of a "hard" class separation and a class can be seen as a Boolean subset of all possible classes. In Boolean logic, a subset $B \subset S$ is defined by $\forall x \in S$: $x \mapsto \{0,1\}$, where $x \mapsto 1$ means $x \in B$ and $x \mapsto 0$ means $x \notin B$. This makes a mapping of all "hard" classes to $\{0,1\}$ possible. In case of the Iris plants problem, a 3 dimensional output vector is constructed using Boolean "element of class" values. The transformed result of the earlier shown entries of the Iris plants data set is shown in figure 4.4.

- **Fuzzy projection:** $[0,1]^n \mapsto [0,1]^m$. Using the Boolean projection scheme of $[0,1]^n \mapsto \{0,1\}^m$ it is not possible to describe the output of a fuzzy logic controller (FLC). As stated earlier, a fuzzy subset $F \subset S$ is represented by $\forall x \in S$: $x \mapsto [0,1]$ with the meaning of 0 and 1 unchanged, but values in-between represent intermediate degrees of membership. The fuzzy projection is capable of enfolding the Boolean projection.

4.1.2 Fuzzy rule set

The form of the fuzzy rules is chosen to look like a rule set which is used by most fuzzy logic controllers. To be used by both the genetic programming rule extractor and the Ishibuchi rule extraction method, rule sets are defined by the following framework:

$$\begin{array}{l}
R_1\texttt{: If } x_1 \texttt{ is } A_{11} \texttt{ and ... and } x_n \texttt{ is } A_{1n} \texttt{ then } B_1 \texttt{ with CF } c_1\\
R_2\texttt{: If } x_1 \texttt{ is } A_{21} \texttt{ and ... and } x_n \texttt{ is } A_{2n} \texttt{ then } B_2 \texttt{ with CF } c_2\\
\qquad\qquad\qquad\qquad\qquad\vdots\\
R_p\texttt{: If } x_1 \texttt{ is } A_{p1} \texttt{ and ... and } x_n \texttt{ is } A_{pn} \texttt{ then } B_p \texttt{ with CF } c_p
\end{array}$$

This framework consists out of p rules with rule k having handle R_k. The input vector is stated in the rule set as x_i with $i \in [1,n]$, which conforms to a n-dimensional input vector. Linguistic variable A can be any of {*small*, *medium small*, *medium*, *medium large*, *large*, *don't care*}, of which membership functions of *small..large* are depicted in figure 4.5. The linguistic value *don't care* returns truth value $\mu = 1$ on all inputs, which has no effect on the result when using a fuzzy *and* operator. In other words, a term "x_i is *don't care*" has no effect on the outcome of the

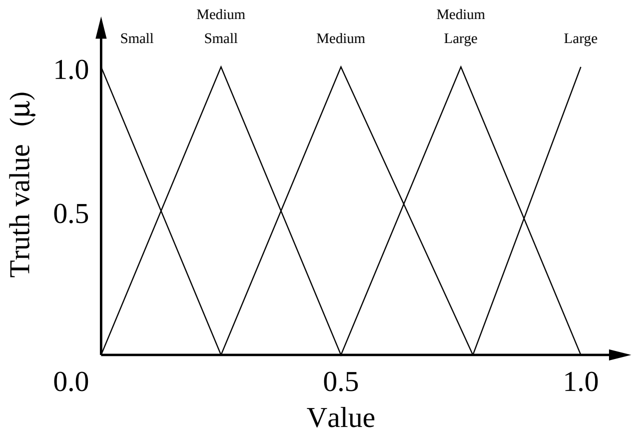

Figure 4.5: Membership functions.

antecedent part and can be left out. The consequent part of a rule R_k is not defined in accordance to standards of fuzzy rules. It contains only "B_k", while the standards use terms like "$output_k$ is *low*" and "$output_k$ is *high*". Because the use of a single linguistic value, for instance *high*, has no effect on the outcome of the fuzzy controller, it is left out. Linguistic variable B_k represents any class variable of the m-dimensional output $Class_1..Class_m$. The last variable discussed is the certainty factor c_k. The value of the consequent part is multiplied by the certainty factor to determine the "fire power" of each rule R_k.

The capability of a fuzzy logic controller utilizing the discussed framework is a n-dimensional to m-dimensional projection: $\Re^n \mapsto \Re^m$. For the sake of uniformity, all used values are normalized to $[0, 1]$, resulting in a restricted projection of $[0, 1]^n \mapsto [0, 1]^m$. This projection is compatible with fuzzy data sets. If a Boolean projection is needed, the output of the fuzzy logic controller is transformed to Boolean. This is achieved by using a decision rule to translate the output vector from $[0, 1]^m$ to $\{0, 1\}^m$:

$$\forall i \in [1, m]\text{: } \texttt{If } o_i > o_j, \forall i \neq j \texttt{ then } o_i = 1 \texttt{ else } o_i = 0$$

In which o_i is an element of the output vector $\vec{o}$. If two identical maximum values are found, the first encountered is assigned 1 and the rest is assigned 0.

4.1.3 Artificial neural network

Artificial neural networks, commonly referred to as "neural networks", are computational models which consist of a large number of densely connected simple processing units. The simple processing units are often called artificial neurons or nodes. The connections between simple processing units are also often called links. Each link is associated with a weight which is often called synapse strength of the link. Neural networks are typically arranged in a number of layers. In the case of feed forward networks, there are connections only between adjacent layers. Figure 4.7 also shows a feed forward network which consists of an input layer, one hidden layer, and an output layer.

When neural network solutions are used for process modeling, the multi-layer perceptron (MLP) network seems to be a very attractive choice. This is because it has been theoretically proven that MLP can approximate any continuous function arbitrarily well provided that enough neurons are used. MLP networks are implemented in almost all commercial neural network simulators and many of the shareware neural network simulators. It is probably due to this wide availability and use that it is one of the most referred type of ANN in rule extraction literature. MLP networks are known for being difficult to extract rules from. This makes it a challenge for the research that is being done on the field of rule extraction.

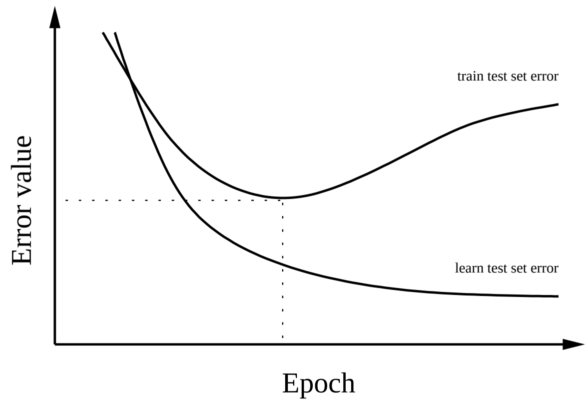

Figure 4.6: Learn and train set error.

4.2 State changers

4.2.1 Data set generator

Generating a data set out of input-output vector pairs representing approaching the mapping of any black box can be achieved by sending an input mesh through the black box. Using a mesh is the most common way to generate an equally spreaded dataset. Suppose an input vector width size of n is used, then easiest results are achieved when it is decided to generate k^n data set points, with $k, n \in \aleph$. All dimensions within the input vector are split up in k equally sized parts and from within each part a random point is chosen. This procedure is done for all possible k^n combinations of unique parts from all input vectors, so a high quality mesh can be formed. All generated points which form the mesh are sent through the black box and a data set is formed with equally spreaded input-output vector pairs.

4.2.2 ANN trainer

Training of a neural network is the procedure of finding the proper weights of the network in such a way that the inputs correspond with the outputs. In this case the error back-propagation training method is employed. The error back-propagation learning algorithm is a form of supervised learning used to train mainly feed-forward neural networks to perform pattern classification tasks. In outline, the algorithm is as follows:

- **Initialization.** The weights of the network are initialized to small random values.
- **Forward pass.** Inputs of each training pattern are presented to the network. The outputs are computed using the inputs and the current weights of the network. Certain statistics are kept from this computation, and used in the next phase. The target outputs from each training pattern are compared with the actual activation levels of the output units, the difference between the two is termed the error. Training may be pattern-by-pattern or epoch-by-epoch. With pattern-by-pattern training, the pattern error is provided directly to the backward pass. With epoch-by-epoch training, the pattern errors are summed across all training patterns, and the total error is provided to the backward pass.
- **Backward pass**. In this phase, the weights of the neural network are updated.

This procedure is repeated until some stopping criterion is reached. Often, a separate learn data set and test data set are used to decide when to stop learning. This is exemplified by the progress of a learn set error and test set error, which is depicted in figure 4.6. This figure also illustrates a good moment to stop training, the best point to stop the training process is marked by the dotted line.

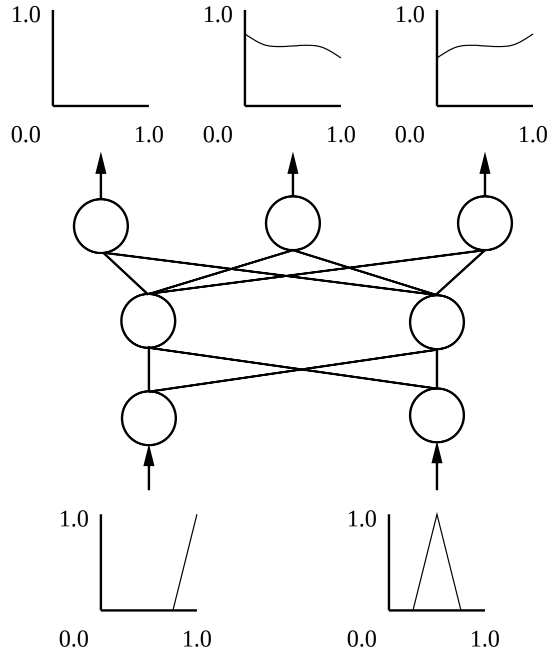

Figure 4.7: Fuzzy projection through a neural net.

4.2.3 Ishibuchi extractor

The method described in [Ishibuchi 1995] is able to extract fuzzy *if-then* rules from trained neural networks. In outline, the procedure works as follows. It does a brute-force search on all possible combinations of linguistic input vectors. All these vectors are presented to the neural network for a test on validity. If a linguistic combination is valid, a rule is added to the set of extracted rules. A more detailed description of the search state space and the linguistic vector validation is presented below.

- **Search state space.** The search state space consists out of input vectors of dimension n containing all possible combinations of fuzzy linguistic values[2]. With an example input vector size of 2, vectors element of the search state space are (*small*, *small*), (*small*, *medium*), (*small*, *large*), et cetera. The used linguistic values are *small..large* and *don't care*. With these 6 different linguistic variables, a total of 6^n combinations must be examined.

- **Linguistic vector validation.** In this phase, all linguistic input vectors are tested for validity which can lead to an extracted rule. The linguistic input vector is presented to the neural network to calculate its output values. In this way the output consequent class and certainty factor are determined by using the necessity grade. If the found certainty factor is larger than an introduced threshold value θ, rules are considered to be valid and added to the resulting rule set. For example, consider the 2-dimensional input vector $\overrightarrow{A}$ (*large,medium*). Figure 4.7 represents this 2-dimensional fuzzy input vector projected through a neural network on a 3-dimensional fuzzy output vector called $\overrightarrow{O}$. The projection of the linguistic input vector onto its output vector is determined numerically using α-cuts, which are described in section 3.2. The decision rule to which class a linguistic input vector belong to is an extension to the decision rule which classifies *crisp* input vectors. Suppose a *crisp* input vector $\overrightarrow{x}$ is presented to the neural network and the ANN returns the *crisp* output vector $\overrightarrow{o}$. This output vector is contained in $[0,1]^m$ and to determine to which class $\overrightarrow{x}$ belongs, the projection $\overrightarrow{o} \mapsto \{0,1\}^m$ is made using the decision rule described in subsection 4.1.2:

[2]such as *small* and *large*

$\forall i \in [1, m]$: If $o_i > o_j, \forall i \neq j$ then $o_i = 1$ else $o_i = 0$

In this case the output class can be found where $o_i = 1$, which means $\overrightarrow{x}$ belongs to class i. A similar rule for classifying a *fuzzy* input vector $\overrightarrow{A}$ follows:

If $O_i > O_j, \forall i \neq j$ then $\overrightarrow{A}$ is classified as $Class\,i$

The necessity grade (described in section 3.3) is used to calculate the outcome of the inequality $O_i > O_j$. The definition of the necessity grade that the linguistic input vector $\overrightarrow{A}$ (such as (*large*, *medium*)) belongs to class i is:

$$Ness(\overrightarrow{A} \in Class\,i) = \min\{\, Ness(O_i > O_j) \mid \forall i \neq j \,\}$$

This necessity grade of can be used as the grade of certainty of this statement: the linguistic input vector $\overrightarrow{A}$ belongs to class i with certainty $Ness(\overrightarrow{A} \in Class\,i)$. Using the threshold value θ, generate *if-then* rules if $Ness(\overrightarrow{A} \in Class\,i) > \theta$, which look like:

If x_1 is A_1 and .. and x_m is A_m then $Class\,i$ with CF $Ness(\overrightarrow{A} \in Class\,i)$

One of the flaws of this algorithm is the application to high-dimensional problems. The main difficulty is the exponential increase of the number of possible combinations of antecedent linguistic values. For example, the total number of possible combinations of six linguistic values in a 13-dimensional input vector is an impressive $13,060,694,016$. To deal with this problem, Ishibuchi proposed an enhancement to the algorithm in [Ishibuchi 1999].

The enhancement involves setting a maximum rule length. For example, three antecedent linguistic values are allowed in stead of thirteen. If a linguistic rule with length two with the antecedent condition "If x_1 is *small* and x_{13} is *medium small*", present the following fuzzy vector to the trained neural network: $A_p = (S, DC, DC, DC, DC, DC, DC, DC, DC, DC, DC, DC, MS)$[3]. Also, a very large threshold θ is chosen at 0.99. Because of the excess use of the *don't care* fuzzy value, outputs from the neural net can have a very large overlap which makes the classification of rules even more difficult.

As a statement in [Ishibuchi 1999], this approach can find a small number of generic linguistic rules from trained neural networks without assuming any specific network architectures or learning algorithms. High understandability of extracted knowledge and high applicability are two characteristic features.

4.2.4 Genetic programming extractor

Sufficient genetic programming building blocks have been discussed to be able to define a genetic programming system which is capable of fuzzy rule extraction. For the setup of a genetic programming system, *program structures*, *genetic operators* and *simulated evolution of a population* have to be defined.

- **Genetic program structure.** To define a genetic program structure, a terminal set and function set are needed. The terminal set T of the GP program consists out of the crisp input vector x_p, the set of possible antecedent and consequent linguistic fuzzy values and a certainty factor CF. Possible antecedent values are chosen equal to Ishibuchi's, *small, medium small, medium, medium large* and *large.* For the consequent part, just the *high* linguistic value is needed. *Low* is not utilized since it would effect a difference in behavior compared to Ishibuchi's method. A suitable function set F needed for the fuzzy rule set creation consists out of `IS`, `AND`, `IF` and `OR`.

  In many genetic programming applications, unrestricted tree constructions are sufficient to solve a problem given a function set and a terminal set that satisfies the closure property.

[3] S, DC and MS denote respectively *small, don't care* and *medium small.*

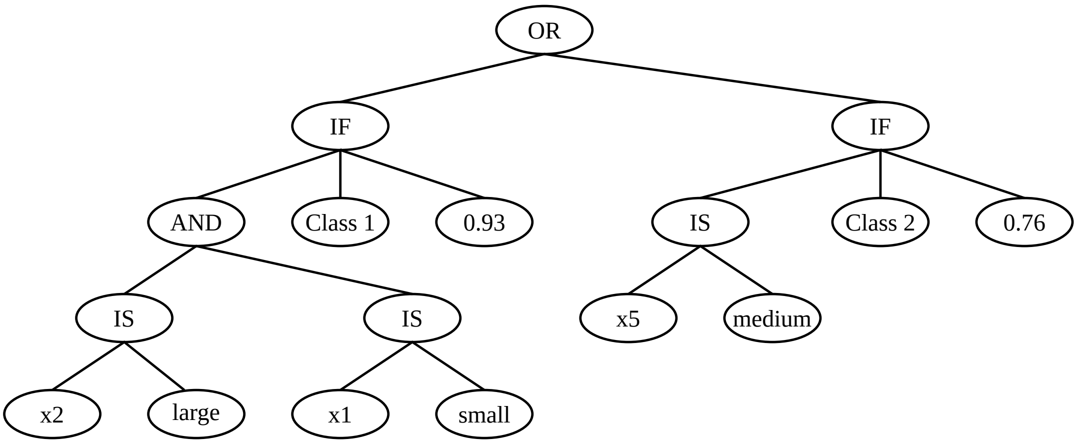

Figure 4.8: Fuzzy rule set genetic tree.

That is, each function in F should be well defined and closed for any combination of arguments that it may encounter. As a result, individuals may have any composition of elements from the combined set, $F \cup T$, occupy the nodes of the tree with the only restriction being that the root must be a function and the leaves must be terminals. This is not the case here. The genetic programming implementation imposes strong constraints on the syntax of a rule set that are defined by special rules of construction. One can imagine how arbitrary placement of functions and terminals in a fuzzy rule tree could lead to severe syntactic violations. Valid rule bases must conform to the following syntactic rules of construction: a BNF[4] description of a fuzzy rule set is given.

```
syntax    ::= rule { "OR" rule }
rule      ::= "IF" termlist "THEN" classn "WITH" cf
termlist  ::= term { "AND" term }
term      ::= xn "IS" lingval
xn        ::= "x" xn
classn    ::= "class" cn
lingval   ::= "small" | "medium small" | "medium" | "medium large" |
              "large"
```

An example of a correct tree constructed using the BNF description is shown in figure 4.8. For example, any other combination of input types for `IF` than (*termlist, classn, cf*) will evidently result in an incorrect tree.

- **Genetic operators.** A syntactic strong definition of a genetic program structure is also referred to as *strongly typed* genetic programming or *STGP*. Strongly typed GP affects all operators used by genetic programming systems, as structural integrity has to be maintained. Achievement of strongly typed GP is realized with the addition of a type level on which genetic operators may operate. Each function is given input- and output types and every terminal acquires an output type. As genetic operators start, all types have to match in order to create a correct tree structure. More details of strongly typed genetic programming can be found in [Haynes].

- **Simulated evolution of a population.** Maybe the most important decision in the design of a genetic programming system is the *fitness function,* which is described on page 14. The choice of fitness function depends on the desired resulting individual. In case of rule extraction, a *correct* rule set is wanted which also has to be *human comprehensible.* The

[4] *Backus Naur Form* [Naur] is a formal notation to describe the syntax of a given language.

correctness is calculated on basis of a comparison of the outcome of a data set $o^{data\,set}$ to the outcome of the genetic programming structure o^{GP}. Human comprehensibility is measured on basis of the dimension *size* of the individual. The *size* is measured by counting the number of nodes in a genetic tree structure. For example, the size of the genetic tree shown in figure 4.8 is equal to 16. Two types of data sets were introduced in subsection 4.1.1, Boolean data sets and fuzzy data sets. Therefore, two types of fitness functions are introduced of which the first is well suited to a Boolean data set and the other is well suited to a fuzzy data set. Both types of fitness functions presented next are standardized fitness functions, which means the lower the fitness value, the better the individual.

- **Boolean fitness function:** $f_{bool} = \sum_{i=0}^{all\,points}(o_i^{data\,set} \neq o_i^{GP}) + (1 - \frac{1}{size})$. This is the fitness function which is used for the calculation of fitness when dealing with Boolean data sets. In this case, the fitness is measured by counting the number of wrong classifications against examples. This is achieved by the first part of the Boolean fitness function. These examples will be in the form of a Boolean data set. For the sake of *comprehensibility,* the smaller individuals with an equal classification rate will be favored. Counting the number of wrongly or correctly classified elements is a common way to set up a fitness function in GP. For example, the artificial ant problem described in [Koza 1992] counts the pieces of food eaten by an ant. The addition of a fitness penalty for the *size* of the individual makes the fitness function multi-objective. It should be noted that "of two individuals, the better classifying individual wins". Size takes a second place.
- **Fuzzy fitness function:** $f_{fuzzy} = \sum_{i=0}^{all\,points}\sqrt{\sum_{j=0}^{m}(o_{i,j}^{data\,set} - o_{i,j}^{GP})^2} + (1 - \frac{1}{size})$. The definition of a fitness function for the fuzzy data set is achieved in another way than the fitness function for a Boolean data set. A fuzzy rule set projection has the same type of output as the fuzzy data set, namely $[0,1]^m$. This makes it possible to calculate the distance between the GP individual and the data set. The summed square error function which is element of f_{fuzzy} will calculate such a distance. This distance is a commonly used element of fitness functions employed in GP problem solving. For example, *symbolic regression* problems commonly use a variant of it, the mean square error function.

All rule sets in the initial population are created randomly and enough trees will be generated to fill a population containing unique individuals. Descendant populations are created by the reproduction, crossover and mutation operators. The offspring of rule sets modified by crossover and mutation must also conform the syntactic structure.

4.3 Rule extractor classification framework

Over time, a number of different strategies have been developed to achieve the desired goals for rule extraction. Not much has been published on the subject of a rule extraction classifier, apart from the proposed classification scheme for comparing results of rule extraction algorithms which is stated in [Andrews]. Andrews mentions several rule classification terms, of which most applicable will be discussed for both the Ishibuchi rule extractor and the genetic programming rule extractor.

- **Expressive power of the extracted rules.** This dimension is chosen as primary dimension by Andrews, as it directly focuses on the actual output presented to the end user by the rule extraction algorithm. Boolean rules are the most common extraction medium. However, it may be clear that the *expressive power* of fuzzy rule sets exceeds that of Boolean rule sets, as Boolean logic is a subset of fuzzy logic. No differences can be mentioned between Ishibuchi and the genetic rule extractor here, as they both extract fuzzy rule sets.
- **Complexity.** The inclusion of such a dimension reflects an almost universal requirement for the algorithms powering the rule extraction process to be as efficient as possible. In

particular a crucial issue in developing a rule extraction algorithm is how to constrain the size of the solution space to be searched.

- **Quality of the extracted rules.** In this case, probably the most important dimension of the classification scheme is the quality of the extracted rules, as it enables a comparison between the Ishibuchi and genetic rule extraction methods against their results. Amongst several other criteria, *accuracy* and *comprehensibility* are mentioned by Andrews. A rule set is considered to be accurate if it is capable of correctly classifying a set of examples. A *classification rate* can be employed, which is the total correct classifications divided by the number of examples. The *comprehensibility* of a rule set is determined by measuring the size of the rule set in terms of number of rules and the number of antecedent linguistic values per rule. Another suggestion for the *comprehensibility* is to count the total number of antecedent linguistic values.

Clearly an assessment of the quality of the rules produced by a given rule extraction technique is potentially of significant value to a prospective user. It should also be observed that in many real world problem domains the simultaneous optimization of all multiple evaluation criteria may not always be desirable. For example, in safety critical applications it is imperative that the classification rate is 100% under all circumstances. This may create a situation for example where comment on the issue of rule comprehensibility is sacrificed for rule accuracy.

Chapter 5

Implementation Issues

To prevent yet another invention of the wheel, the total implementation process was performed in four sequential steps. First, the research cycle was inspected from an implementation viewpoint. Because the research cycle depicted in detail in figure 4.2 is a model and not a program description, aspects concerning a computer program are missing. An assessment of required tasks of software to be used in the modules is done, from which a workbench has been designed. The structure and capabilities of this workbench will be presented in section 5.1. In the second step as many as possible existing software was found using the Internet, of which some proved to be very mature and useful. Of a large collection of programming libraries found, the most suitable were chosen and are presented in section 5.2. The third action undertaken was the design of a simple database structure which is capable of performing the tasks required by the workbench introduced in the first step. This database structure is subject of discussing in section 5.3. As a final step the actual coding of program modules was performed of which the most interesting parts are subject of review in section 5.4.

5.1 Workbench setup

A workbench has been designed which is capable of handling the needed functionality as described in sections *knowledge states* and *state changers* of chapter 4. An overview of the workbench is presented in figure 5.1. In this figure, the database table symbols represent the knowledge states *rule sets*, *data sets* and *neural networks* which are discussed in section 4.1. The procedure symbols stand for separated program modules, which are connected to the tables with edges. The direction of these edges match the direction of the data flow. The procedures are discussed briefly below.

- **GP extractor.** This module performs a genetic programming rule extractor as described in subsection 4.2.4, in which a data set is transformed to a fuzzy rule set.
- **Ishibuchi extractor.** Ishibuchi's rule extractor module is implemented with functionality in accordance to subsection 4.2.3.
- **Rule set tester.** A verification tool which checks a fuzzy rule set against a data set is implemented in this program module.
- **Data set generator.** This module generates a data set out of a fuzzy rule set, given the procedure described in subsection 4.1.1.
- **ANN trainer.** Training a neural network on a data set is the capability of this module.
- **ANN tester.** This is a module which verifies a trained neural network against a data set.

Translating all possible knowledge states is key issue of most program modules. Some extra testing functionality is needed to ensure no errors are introduced during a "translation" process. Modules operate stand-alone without the use of a graphical user interface (GUI).

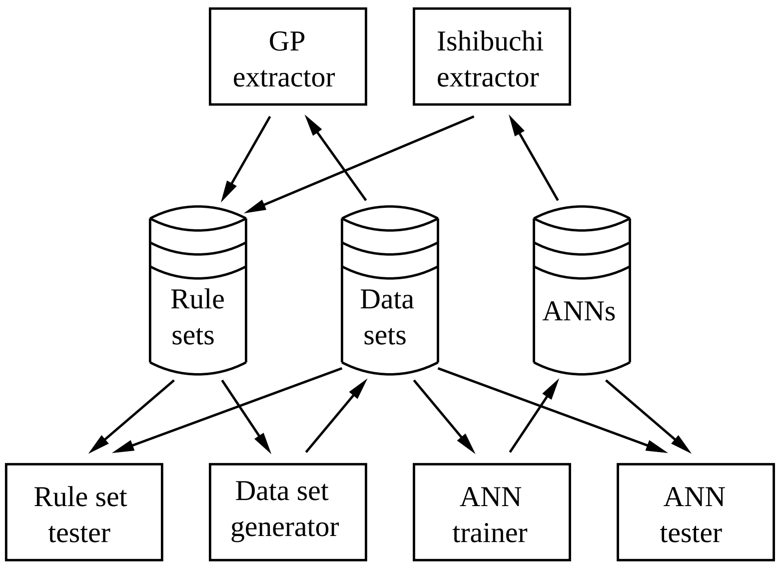

Figure 5.1: Workbench setup.

5.2 Available library assessment

Not for fuzzy logic, neural networks nor genetic programming exists some sort of ready-to-use implementation which is broadly used by the scientific community. As all modules communicate using plain text files, they become programming language independent so the choice of language per module depends on the availability of libraries for their specific task. However, Object Oriented (OO) libraries are preferred, since the OO paradigm should decrease development time substantially. The following libraries are used:

- **FLIP++.** A `C++` library which is called the Fuzzy Logic Inference Processor library, or FLIP++ is used. It is part of StarFLIP, a reusable iterative optimization library for combinatorial problems with fuzzy constraints as described in [Gottlob]. It is capable of doing literally every aspect of fuzzy logic control, such as described chapter 3.

- **Libmozg.** A `C++` neural network library called "mozg" by Alexy Filin has been utilized. Mozg is a flexible, fast neurolibrary. Up to the moment, it allows one to create, learn and use multi-layer perceptron (MLP), which is the most popular artificial neural network for the large number of problems solved by ANN today.

- **Lil-gp.** Michigan State University's lil-gp is a `C` language system for developing genetic programming applications based on the LISP work in [Koza 1992] and [Koza 1994]. Lil-gp evolves trees whose nodes are `C` function pointers, so tree evaluation is done entirely with compiled code. This gives a manifold speed increase, allows the program to handle much larger problems (bigger populations, more generations), and is portable to a wide variety of platforms. Lil-gp individuals can be composed of multiple trees, some of which can evaluate the others to give ADF[1] capabilities. This system is the first to support multiple-population runs with arbitrary interchange topologies. Lil-gp has many options for controlling breeding, and is capable of emulating the "Simple LISP Code" given in the appendix of [Koza 1992]. A patched[2] version of lil-gp is available on the Internet at [Luke], which enables the needed *strongly typed* GP as described in subsection 4.2.4.

[1] *Automatically Defined Functions* are discussed in subsection 2.2.2 on page 11.
[2] A *patch* is used to upgrade or improve software.

Of course, this is only a small selection of the tools which are available at the moment. Other tools like the Vienna University Genetic Programming kernel, GPC++, QuickGP, SpiderWeb, Billnet, Jet's Neural Architecture, C++ Matrix Class and the Fuzzy Logic toolbox for Matlab have been investigated for possible use. Either the tools did not have the capabilities searched for, were not platform independent, cost money, had poor documentation or were unnecessary extensive.

5.3 Database structure

The database symbols used in figure 5.1 indicates that three different tables are needed to fulfill the data warehousing capabilities. Because of the relatively low technical requirements of the capabilities of the database, a basic database structure is chosen which consists out of a directories containing text files. Directories represent tables holding plain text files which represent the entries the tables. In a to the workbench assigned root directory, three directories named `datasets`, `anns` and `rulesets` are created to store and retrieve text files. These text files are named in such a way that they are easily distinguishable by the user. The structure of the file syntaxes for the different knowledge states are discussed next.

- **Data set. Directory**: `datasets`. A data set is represented by input-output vector pairs. An input vector $\overrightarrow{x}$ with size n should be inside $[0,1]^n$, an output vector $\overrightarrow{o}$ representing m classes should fit within $[0,1]^m$. This representation is capable of holding both Boolean and fuzzy rule sets. The file syntax used for transferring data set information first three lines define n, m and the number of data set points. Then, on all lines following, each data point is given in a comma separated input vector followed by a comma separated output vector. A screen dump is given of the beginning of the Iris plants data set file:

```
4                                   // define nr of attributes
3                                   // define nr of classes
150                                 // define nr of data points
0.222,0.625,0.068,0.042,1,0,0       // datapoint 1
0.167,0.416,0.068,0.042,1,0,0       // datapoint 2
0.111,0.500,0.051,0.042,1,0,0       // et cetera
0.083,0.458,0.085,0.042,1,0,0
```

- **ANNs. Directory**: `anns`. What exactly the file representation of a neural network looks like, is not necessary to know. Mozg is very convenient in file handling, ANNs need two lines of `C++` code to do initialization and loading:

```
mozgMLP *neural_network;
input_file >> &neural_network;
```

  If neural network access is desired in any program module, it will be sufficient to add these two lines of code.

- **Fuzzy rule set. Directory**: `rulesets`. The fuzzy rule set files must be human definable. Due to the complex nature of the file syntax used by FLIP++, the original file syntax is not used. A special designed file syntax in combination with FLIP++ library calls are used to define the fuzzy rule set ready to use inside the program. An example of two rules is used to illustrate the file syntax, with comments on the right side:

```
5                                  // number of linguistic values
small
medium_small
medium
```

```
medium_large
large
2                                         // number of linguistic variables
x%d
2
low                                       // use consequent ling vals
high                                      // for compatibility with FLIP++
3
class%d
1                                          // 1 term in antecedent
x1 is small                                // antecedent part
class1 is high                             // consequent part
0.85                                       // CF
2                                          // 2 terms in antecedent
x1 is medium_small x2 is medium_large
class2 is high
0.27
-1                                         // no more rules, EOF
```

The file starts with the definition of 5 antecedent variables *small..large*. The two-sized input vector (x_1, x_2) is automatically named with the `x%d` using a `sprintf` function call. Two output variables `low` and `high` are succeeded by the definition of three output classes `Class1`..`Class3`. After initializing all possible linguistic variables and values, each additional rule can finally be defined in four lines per rule. The first line states the number of terms in the antecedent part, next line gives the actual antecedent part followed by the consequent part of the rule. To finish the initialization of each individual rule, the certainty factor is given. Finally, the program is instructed the file ends by giving -1 as the number of terms at line one.

5.4 Realization of modules

In this section, all realized modules will be discussed. Names of subsections will be the name of the implemented module. For all modules, a input-output relation is given which illustrates the capabilities of every individual program module.

5.4.1 Data set

Input: various, see separate submodules. **Output**: various, see separate submodules.

The basis capability of the data set program is to do most simple procedures which acts upon a data set. It is not a computationally expensive program, and it has a very low execution time compared to the other modules. Actually it enfolds three of the program modules of the workbench presented in figure 5.1, namely the data set generator, the ANN tester and the rule set tester. These submodules will be discussed successively.

- **Data set generator. Input**: a fuzzy rule set. **Output**: a data set.
  This module generates a data set out of a fuzzy rule set, given the procedure described in 4.1.1. To generate a data set, a simple mesh is formed on the basis of the procedure described in section 4.1.1. One combinatorial heuristic problem is to generate a sequence of all different combinations for parts of a vector of variable width. For example, if a vector is to be split up in three parts and has a width of two, the following combinations of parts represents all 3^2 possible indexed pairs: $\{(0,0),(1,0),(2,0),(0,1),(1,1),(2,1),(0,2),(1,2),(2,2)\}$. A variable sized vector size n and a variable number of parts per vector k are used. Consider the vector to have the naming scheme of $a_0, a_1, ..., a_{n-1}$ and all possible index combinations are named $C_0, C_1, ..., C_{n^k-1}$. Then, $C_i a_j = (\frac{i}{k^j} \bmod k)$ is the required index $\forall(i,j) = ([0, n^k - 1], [0, n -$

1]). Note that $i, j, k \in \aleph$ so a divide operation loses its remainder[3], which is the intention. The following piece of C++ code prints all possible combinations of indexed parts for a vector of a given width.

```
for (int i=0; i<pow(parts_per_vector,vector_width); i++) {
  for (int j=0; j<vector_width; j++) {
    cout << i/pow(parts_per_vector,j)%parts_per_vector << " ";
  };
  cout << endl;
};
```

The same example put through this part of code with `parts_per_vector` set to 3 and `vector_width` set to 2 delivers exactly the same pairs and in the same order as stated a few lines back. With these generated pairs, it is relatively easy to generate a complete fairly distributed grid. If a random point is chosen from inside every generated part, sent through a black box and the result is stored, a input-output pair related grid is achieved.

- **ANN tester. Input**: A data set, a neural network. **Output**: an overview of possible classification errors against a data set made by a neural network.
  This is a module which checks a trained neural network against a data set. A neural network is loaded, after which on a per-point basis, the data set is compared to the neural network. The input vector of a point in the data set is fed to the neural net, which produces an output vector. Then, using the decision rule introduced in subsection 4.1.2

  $\forall i \in [1, m]$: `If` $o_i > o_j, \forall i \neq j$ `then` $o_i = 1$ `else` $o_i = 0$

  the output vector from the data set file is compared to the output of the neural net. Using this decision rule, the quality of the classificational power of a trained neural network is determined. If a point is classified wrongly, the ANN tester module states the point number, input vector, output vector present in the data set file and the output vector from the neural network. An example of the output of this module is shown below.

```
Starting Neural Network test...
-------------------------------
Input vector size:   8
Output vector size:  3
Number of points:    178
ERROR at case 72.
Input vector: 0.65 0.18 0.47 0.69 0.19 0.31 0.32 0.26
Output vectors: 0 1 0 <-> 2.4e-06 5.63e-06 1
ERROR at case 164.
Input vector: 0.72 0.40 0.50 0.59 0.22 0.13 0.072 0.53
Output vectors: 0 0 1 <-> 0.000729 1 3.02e-07
-------------------------------
Neural Network Test complete.
```

- **Rule set tester. Input**: a data set, a rule set. **Output**: an overview of possible classification errors against a data set made by a fuzzy rule set.
  This is a module which checks a fuzzy rule set against a data set. The needed actions are almost identical to those executed by the ANN tester, which checks a neural network against a data file. But, in stead of sending the input vector obtained from the data set file through the neural net, it is sent through the fuzzy controller. The same decision rule is applied to classify the point and the output produced matches that of the ANN tester.

[3] A *truncate* or *floor* operator has the same effect for numbers element of $\Re^+$.

5.4.2 ANN trainer

Input: a data set. **Output**: a neural network.

The solely task of the ANN trainer module is to train a neural network using a data set file as input. Taking a closer look at libmozg shows that structural parts of libmozg are:

- **Kernel library.** This is a library of fast vector and matrix operations. It consists of `mozgVector` and `mozgMatrix` class templates and functions templates using them.
- **mozgMLP.** This class which uses the kernel library simulates the multi-layer perceptron and its behavior. Using the constructor, one can create an ANN with desirable number of layers, number of neurons (called units in mozg) in them, output function in each layer, with/without bias term. Other functions are available to learn, use, save and load a neural network.

A neural network's input should be inside the hypercube $[0,1]^n$, which is a restriction incorporated in the data set file itself. Training a neural network using mozg is pretty straight forward and can be achieved in three steps, initialization, setting parameters and training. These steps are described below.

- **Initialization.** In this phase, all topological settings are defined. The following constructor is used to create an ANN:

  ```
  mozgMLP (int number_of_layers, int* num_outs_layers, int* func_number,
           float weight_sweep, bool bias_term_flag);
  ```

  In which a neural network is initialized with a reference to its number of layers, number of units at each layer, output functions to use at each layer, an initial weight distribution and whether a bias term should be included or not. A bias term, which is a unit set with an activation level set to -1, is useful when mapping a zero vector onto a non-zero vector.

- **Setting parameters.** Parameters can be set to initialize the neural net to simulate a chosen behavior, such as the back propagation method:

  ```
  putBackProp(float learning_rate, int rate_mode, int energy_function,
              int num_for_chunk, int steps_number);
  ```

  This initializes learning parameters for the back-propagation rule without any extensions. Parameters set by this function call are the `learning rate`, a `rate mode` which states if a layer's ϑ should be divided by the number of inputs the layer has, or not, and finally the to be used energy function. This energy function can be set to three states:

  - Summed squared error: $E = \sum_p \sum_i (t_i^p - o_i)^2$
  - Cross entropy: $E = \sum_p \sum_i (t_i^p - \min o) \cdot \ln(o_i - \min g) + (\max o - t_i^p) \cdot \ln(\max g - o_i)$
  - Log-squared error: $E = \sum_p \sum_i \ln(1 - (o_i - t_i^p)^2)$

  The number of learning examples in a chunk (usually set to 1 when using standard back-propagation) and the number of steps per each chunk number of examples. This means a learning example is used successively `steps number` times for each epoch. This function sets all parameters used by the neural net to a back-propagation learning method. Other training settings can be called such as the Quick-prop or Rprop, but are not utilized here.

- **Training.** The actual training is started by calling an appropriate training function. The termination of training is a crucial aspect for the quality of the ANN. To start learning a neural network, a function `learnNet` has to be called, which looks like this:

```
learnNet(float** learn_inputs, float** learn_outputs, int learn_size,
         int vector_order, int epnum, int pepnum);
```

In which `learn inputs` represents a pointer to a input vector and the same holds for `learn outputs`. `Vector order` may be 0 for random order of presenting learning examples to the ANN or 1 for a ordered presentation to the neural network. Using random distributed learning examples helps generalizing the network better. `Epnum` is the maximum epoch number of for learning, learning is always terminated at `epnum`. Each `pepnum`'th epoch, a function `AnswerMessage` is called, which is a user defined procedure which determines whether the training should stop or not.

The following function of `AnswerMessage` is an example of how it can be defined. It will stop training at the lowest possible test error.

```
answerMessage(float learn_error, float test_error)
{
  static float best_error = 1e9;
  if (test_error < best_error) {
    ofstream output_file("result.ann");
    output_file << *this;
    output_file.close();
    best_error = test_error;
  }
}
```

This procedure saves the best possible found neural net, the one with smallest test error. However, attention should be paid to local variations of the test error, as the neural net training curve has spikes on which the test error is near zero.

5.4.3 Ishibuchi extractor

Input: a neural network. **Output**: a fuzzy rule set.

As no code used to produce the results presented in Ishibuchi's paper could be found on the Internet, it had to be reproduced with hints spread across two of Ishibuchi's papers.

The Ishibuchi heuristic search space consists out of all possible combinations of linguistic variables. However, in most cases not *all* possible combinations. A fuzzy rule containing a DC linguistic value is identical to the presence of five other rules, containing [*small, large*] in stead of the DC. The search heuristic starts with as many as possible *don't cares,* if a rule is found containing don't cares, all other redundant rules are removed from the search space. A search space is initialized in the form of an array which holds all the possible combinations of [*small, don't care*] which should be examined. Ishibuchi's method starts with as many as possible don't cares, tries to find rules and decreases the number of DCs if all possible combinations with a certain number of DCs were searched. If correct rules are found, all rules which are "embodied" by the DCs are removed from the search space.

Inside the main loop, in which procedure sequentially are presented combinations of linguistic values, a test is performed to check whether the antecedent part it represents is valid for any output class. Using the concept of the necessity grade combined with the α-cuts as described in section 4.2.3, a search is done for the smallest possible α for which a valid rule can be found. Smallest possible α means starting at a low value and incrementing it every time nothing is found until a maximum is reached. A rule is supposed to be valid if one the minimum of a one output is greater than any other output's maximum.

A commonly used feed-forward error back-propagation neural network is used. Training is terminated after 50000 learning steps without the use of both a learn data set and a test data set, only a learn data set is used to to train the ANN.

Table 5.1: Function and terminal types.

Function/Terminal	Output type	Input types
`OR`	0	0,0
`IF`	0	1,2,3
`AND`	1	1,1
`IS`	1	none
`CLASS`	2	none
`CF`	3	none

5.4.4 Genetic programming extractor

Input: a data set. **Output**: a fuzzy rule set.

The implementation of a genetic programming rule extractor using lil-gp can be divided in three categories: code which defines the structure of the tree, code which does all basic definitions and code which implements all functions and terminals. The first type of code also loads the data set at startup and initializes all global variables. The second group of code defines global variables and data types which all `C` functions rely on. The last group of code is the actual `C` implementation of the terminals and functions. Also, the total tree evaluation which implies the calculation of fitness is included in this last category. All three groups of code are discussed below.

- **Defining tree structure.** Before an implementation of the tree structure could be performed, a transformation had to be made from the BNF description of the fuzzy rule set tree described in subsection 4.2.4 to lil-gp's framework. This framework of strongly typed genetic programming is required for lil-gp's genetic operators to function properly. Each function is assigned an output type and can be assigned multiple input types. A terminal has just one output type. When constructing a genetic tree structure, it is made sure that all outputs linked to inputs are of the same type. Hence the term strongly *typed* genetic programming exists. The BNF description is translated to three functions and three terminals utilizing four types. An overview is presented in table 5.1. A more detailed description of respectively these types, functions and terminals are discussed next.
  - **Types.** Four types are used in the implementation of the genetic tree and these types are numbered type 0 through type 3. The first type is the output of a fuzzy rule set, which is represented by a m-dimensional vector of the form $[0,1]^m$. The next type is the truth value of antecedent (sub)part(s). For example, `x3 IS large` is assigned a truth value which is of type 1. The third type can take a value from 1 to m, which selects one of the available classes. A number within $[0.1, 1]$ is used for the certainty factor (CF) which is of type 3.
  - **Functions.** The implemented functions are named `OR`, `IF` and `AND`. The `OR` function takes the output of two fuzzy rule sets as input and will return a combined output by utilizing the maximum operator. The `IF` function implements a single fuzzy rule, which is a special case of a fuzzy rule set with just one element. This function has an arity of three and the following input types: the antecedent part truth value, the certainty factor that has to be applied and the consequent class. After calculating truth values, the `IF` function will return an output of a fuzzy rule set. The last function discussed here is the `AND` function. This function is used to combine terms of the antecedent part of the rule. The output of `AND` is computed using a minimum operator.
  - **Terminals.** Basically, the only terminals used to form the genetic tree structure are named `IS`, `CF` and `CLASS`. Table 5.1 shows the output types of all three implemented terminals. All terminals `IS`, `CF` and `CLASS` are Ephemerical Random Constants[4] (ERC) using different ranges of values. The `IS` ERC chooses a random combination of an

[4]An *Ephemerical Random Constant* is a randomly initialized value which is valid as long as it exists.

antecedent linguistic variable and value. When presented an input vector from a data set it will return a truth value. The certainty factor terminal CF is chosen within the range $[0.1, 1]$ with a step size of 0.01. Choosing a class is done by the CLASS terminal, which chooses a valid class.

- **Basic definitions.** This portion of the code is responsible for all global declarations. Amongst these declarations is the definition of the DATATYPE data type. This particular data type is used by all other implemented functions and terminals as an union data type of different data types. The piece of code in which DATATYPE is defined, looks like this:

```
typedef union app_datatype_ { int class; is_res truth_vector;
                              float rand_cf; } app_datatype;
#define DATATYPE app_datatype
```

The truth vector is an implementation of values of type 1 stored in an array-like structure called is res. The representation of type 2 is an integer called class, while type 3 is implemented using a float called rand_cf. Type 0 is implemented through a global variable which is called gpres. All global variables are defined in a struct called globaldata. This struct is defined as follows:

```
typedef struct { int input_vector_size; int output_vector_size;
                 int nr_of_points; float **input_vectors;
                 float **output_vectors; float **gpres; } globaldata;
```

The data set characteristics are defined by input vector size, output vector size and nr of points. Global arrays of arrays with input and output vectors are available through the use of input vectors and output vectors. A global array of array is used to contain the outcome of a genetic tree. All implemented functions operate on vectors and not single numbers, to make the implementation more efficient.

- **Implementing functions and terminals.** For every ordinary function and terminal in a problem, a C function has implement the action of that function or terminal. Each C function is pass two arguments. What it does with these arguments depends on whether it is implementing a function or a terminal. If it is a function, it also depends on the type of function. All these C functions should return the DATATYPE datatype, as described earlier by the basic definitions. An example of the implementation of the AND function is presented.

```
DATATYPE f_and ( int tree, farg *args ) {
  globaldata *g = get_globaldata();
  int i;
  DATATYPE answer;
  for (i=0; i<g->nr_of_points; i++)
    answer.truth_vector.points[i] =
                 minimum(args[0].d.truth_vector.points[i],
                         args[1].d.truth_vector.points[i]);
  return answer;
}
```

A terminal function that acts as an ERC needs two C functions. One will generate a new constant and the other will print it's value to a string. The first is passed a pointer to DATATYPE, it should generate a new value and place it in the pointer. An example of both functions is the implementation of the certainty factor function. It's code is shown below.

```
void f_rand_cf_gen ( DATATYPE *put_here ) {
  (*put_here).rand_cf = (rand() % 91) / 100.0 + 0.10;
}
```

```
char *f_rand_cf_print ( DATATYPE value ) {
  static char buffer[20];
  sprintf ( buffer, "CF %.2f", value.rand_cf );
  return buffer;
}
```

Exactly in what steps the lil-gp genetic programming system itself simulates the evolutionary process is available for review in [Zongker].

Chapter 6

Experimental Results

To assess the abilities of both the Ishibuchi rule extractor and the genetic programming rule extractor, a series of tests will be conducted. These tests are performed by executing program modules from the workbench introduced in chapter 5. Five problem domain data sets which will be subjected to the rule extractors are introduced in section 6.1. Two of these problems are specially designed to enable the research cycle discussed in chapter 4 to make a complete circular motion. The other three data sets are commonly found in rule extraction literature. To solve a problem, it is likely that more than one modules have to be executed before the desired answer is found. For example, if one wishes to perform a rule extraction from a data set using the Ishibuchi extractor, then a neural network has to be trained on the data set for the Ishibuchi extractor to operate on. Plain execution of these program modules does not guarantee a satisfying solution to a problem. Available parameters for all program modules which can be set differently to be able to achieve better results are discussed in section 6.2. Extracting rules using both rule extractor methods and the acquired results are presented in section 6.3. A discussion about the performed experiments is stated in section 6.4.

6.1 Problem domains

To determine the quality and the operational domain of a rule extractor, the sum of all tests has to be well-balanced. Therefore, five problem domains which have specific characteristics of types of problems are used for testing. The first problem domain is a warming-up for the rule extractors. The *Square darts board* problem is introduced in subsection 6.1.1 and has an easy understandable structure, which is described by a non-redundant rule set. *One decides all* is the name of the second problem presented in subsection 6.1.2, which tests the redundant power of a rule extractor. The data set contains a very high level of redundancy, only one fifth of the attributes is needed for a correct representation. The third problem described in subsection 6.1.3 is a standard test which is often referred to in literature, the *Iris plants* problem. Next, a high dimensional problem is used to test the scalability of the rule extractor. The *Wine* classification problem, which data set consists out of as many as thirteen attributes and three classes, is introduced in subsection 6.1.4. The final test will be one of the noisiest domains available, the *Breast cancer* problem domain and it is reviewed in subsection 6.1.5. This problem will also test the representational power of the fuzzy rules itself. Of all problem domains, the number of attributes, number of classes and the number of data set points are given, which gives an indication of the magnitude of difficulty of the problem. Additionally the type of data set is denoted.

6.1.1 Square darts board data set

Attributes: 2. **Classes:** 3. **Size:** 100. **Data set type:** Boolean, fuzzy.

The first rule set to be put through all tests is special designed one. The primary goal is to

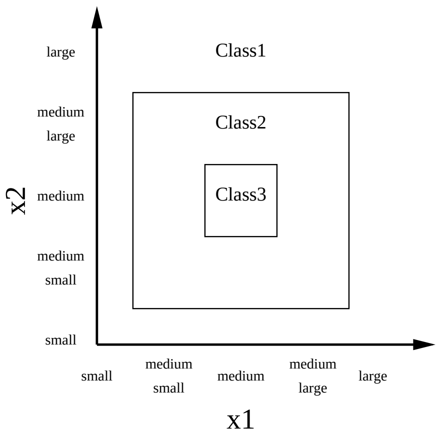

Figure 6.1: Square darts board.

have a human comprehensible structure, so differences between the original and the outcome of a rule extractor can help determine potential errors made by the rule extractors. A two-dimensional input (x_1, x_2) is defined which project on three output classes. The depiction of the designed mapping can be best described as a square formed "darts board" and is shown in figure 6.1. The minimum rule set which describes the *square darts board* consists out of thirteen rules which are stated below:

```
IF x1 IS small THEN Class1 WITH CF 1.00
IF x1 IS large THEN Class1 WITH CF 1.00
IF x2 IS small THEN Class1 WITH CF 1.00
IF x2 IS large THEN Class1 WITH CF 1.00
IF x1 IS medium_small AND x2 IS medium_small THEN Class2 WITH CF 1.00
IF x1 IS medium_small AND x2 IS medium THEN Class2 WITH CF 1.00
IF x1 IS medium_small AND x2 IS medium_large THEN Class2 WITH CF 1.00
IF x1 IS medium AND x2 IS medium_small THEN Class2 WITH CF 1.00
IF x1 IS medium AND x2 IS medium_large THEN Class2 WITH CF 1.00
IF x1 IS medium_large AND x2 IS medium_small THEN Class2 WITH CF 1.00
IF x1 IS medium_large AND x2 IS medium THEN Class2 WITH CF 1.00
IF x1 IS medium_large AND x2 IS medium_large THEN Class2 WITH CF 1.00
IF x1 IS medium AND x2 IS medium THEN Class3 WITH CF 1.00
```

All certainty values are chosen equal to 1.0 so no marginal area comes into being, which could cause unwanted complexity. The data set generator module was called with parts per vector set to 10, so it generated $10^2 = 100$ points. Both Boolean typed and fuzzy typed data sets were generated and stored.

6.1.2 One decides all data set

Attributes: 5. **Classes:** 5. **Size:** 243. **Data set type:** Boolean, fuzzy.

In order to test the capability of removing redundant information from a rule set, the *One decides all* data set was designed. While the number of attributes is equal to five, only one attribute is needed to make the full separation between five classes. The fuzzy rule set describing this class decision using one attribute is stated next.

```
IF x1 IS small THEN Class1 WITH CF 1.00
IF x1 IS medium_small THEN Class2 WITH CF 1.00
IF x1 IS medium THEN Class3 WITH CF 1.00
IF x1 IS medium_large THEN Class4 WITH 1.00
```

```
IF x1 IS large THEN Class5 WITH 1.00
```

Dummy values x_2 through x_5 are used during the generation process of the data set file of this problem. The values for $[x_1, x_5]$ were stored to the data set as if the entire input vector was used for output vector determination. In input mesh for 5-dimensional problems was utilized to generate fairly distributed points. The data set generator module was called with parts per vector set to 3 which then generated $3^5 = 243$ points. Boolean as well as fuzzy typed data sets were created and written to the file.

6.1.3 Iris plants data set

Attributes: 4. **Classes:** 3. **Size:** 150. **Data set type:** Boolean.

The *Iris Plants Database* is perhaps the best known database to be found in the pattern recognition literature. The data set contains three classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other two; the latter are not linearly separable from each other. The attributes sepal length, sepal width, petal length and petal width in are available in 150 instances to predict its class, Iris Setosa, Iris Versicolour or Iris Virginica. The main reason for the usage of it here is to be able to compare three outcomes: the rules stated in [Ishibuchi 1995], the product of the Ishibuchi rule extractor and the output of the genetic programming rule extractor.

6.1.4 Wine data set

Attributes: 13. **Classes:** 3. **Size:** 178. **Data set type:** Boolean.

A widely used high dimensional problem is the *Wine recognition data*. This data set are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines. The attributes are successively alcohol, malic acid, ash, alcalinity of ash, magnesium, total phenols, flavanoids, nonflavanoid phenols, proanthocyanins, color intensity, hue, OD280/OD315 of diluted wines and proline.

6.1.5 Breast cancer data set

Attributes: 9. **Classes:** 2. **Size:** 286. **Data set type:** Boolean.

This *breast cancer* domain was obtained from the University Medical Centre, Institute of Oncology, Ljubljana, Yugoslavia. Thanks go to M. Zwitter and M. Soklic for providing the data[1]. This database is one of three domains provided by the Oncology Institute that has repeatedly appeared in the machine learning literature.

This is a very noisy domain, so here the combination of the representational power of the fuzzy rule set and the rule extractor are tested. Of previous classifier results, the highest achieved accuracy so far was by Cestnik's Assistant-86 [Cestnik], which reached a 78% accuracy. All attributes of which an instance existed with lacking data were expanded by the possibility *unknown*. The attributes are age, menopause, tumor-size, inv-nodes, node-caps, deg-malig, breast, breast-quad, and irradiat.

6.2 Parameter settings

For a program module to function properly to the different problem domains, several parameters can be set. For most modules, the number of parameters which can be set to alter the behavior of the program module is just one or two. In all but the genetic programming module the parameters are hard-coded into the program source code, so if a parameter is changed the program has to be recompiled.

[1] It is asked to include this citation for the use of this database.

Table 6.1: Dataset generator parameters.

Parameter	Default value
Input rule set file name	n/a
Output data set file name	n/a
Number of parts per vector	n/a

Table 6.2: ANN tester parameters.

Parameter	Default value
Input ANN file name	n/a
Input data set file name	n/a

6.2.1 Dataset

As described in subsection 5.4.1, this program module enfolds three smaller submodules with different capabilities. For most submodules, selecting the correct files on which they operate is all that has to be done. The ANN tester submodule and the rule tester submodule only accept file names as parameters. For the data set generator module, two file names and the number of parts per vector as described in subsection 5.4.1 have to be set. Tables 6.1, 6.2 and 6.3 illustrate the parameters of respectively the dataset generator, the ANN tester and the rule set tester submodules.

6.2.2 ANN trainer

A neural network is defined by a larger quantity of settings as can be observed in subsection 5.4.2 as well as table 6.4. The number of input neurons is set to match the input vector dimension and the same is done for the output neurons and output vector. One topological setting of the neural network is adjustable, namely the number of neurons in the hidden layer. The more neurons in the hidden layer, the easier the network adopts to a given data set. The less the number of hidden neurons, the more difficult it is to find a suited parameter set to get a well-trained neural network, but the better the generalizing power of the neural net will be. In [Ishibuchi 1995] is stated that 3 hidden neurons were used in the artificial neural net, so a high level of generalization is needed for neural networks on which the Ishibuchi rule extractor operates.

To some domains neural networks are harder to adopt to than to other domains. Finding the correct parameter set for the neural network learning algorithm is achieved by trial and error determination. For some parameters, table 6.4 gives an indication of the values of the parameters.

6.2.3 Ishibuchi extractor

The functionality of the Ishibuchi rule extractor is encapsulated by a well defined procedure. Not many different settings are supported, as can be observed in table 6.5. Setting file name for the input neural network and the output rule set is pretty straightforward. The minimum number of don't cares has influence on the maximum length of the antecedent part of a fuzzy rule. The higher the minimum number of don't cares, the smaller the antecedent part can be. The threshold θ represents the minimum value of the necessity grade for a fuzzy rule to be accepted or not. It has an impact on the quantity and quality of the rules extracted. The lower θ, the more rules will

Table 6.3: Rule set tester parameters.

Parameter	Default value
Input rule set file name	n/a
Input data set file name	n/a

Table 6.4: ANN trainer parameters.

Parameter	Default value
Input data set file name	n/a
Output ANN file name	n/a
Number hidden layers	1
Number of neurons in hidden layer	n/a
Width of initial weight distribution	~ 0.3
Bias term	1
Learning rate	~ 0.5
Layer's ϑ fan-in	disabled
Learning examples in chunk	1
Number of steps per chunk	2
Presentation order of examples	n/a
Epoch on which to stop	$\sim 50,000$

Table 6.5: Ishibuchi extractor parameters.

Parameter	Default value
Input ANN file name	n/a
Output rule set file name	n/a
Minimum number of *don't cares*	n/a
Threshold θ	~ 0.10

be considered acceptable and added to the output rule set. Higher values of θ imply the opposite, fewer rules with higher quality per rule. However, it should be noted that the impact of θ on the rule set as a whole is different. In this case, it seems that "the lower θ, the more rules and the higher the classification rate" is a valid statement. The value for θ is chosen to be ~ 0.10, this value was also used by Ishibuchi. The number of don't cares is often selected in such way that no more than three antecedent linguistic values will be used.

6.2.4 Genetic programming extractor

The lil-gp genetic programming system supports many settings to be customized which influent the behavior of the evolutionary process. An overview of parameters used is presented in table 6.6. All parameters utilized by lil-gp are stored in a parameter file. For the input to the GP system a data set file has to be selected. Currently the implementation of the GP rule extractor does not support output of fuzzy rule sets in accordance to the database structure presented in section 5.3. Manual translation is needed for the rule tester module to verify the output of the GP extractor.

Table 6.6: GP extractor parameters.

Parameter	Default value
Input dataset file name	n/a
Output rule set file name	`fuzzy.bst`
Maximum generation	$2,500$
Population size	$10,000$
Initialization method	grow
Initialization depth	$1-20$
Maximum nodes	1500
Breed phases	n/a
Selection type	n/a

However, an indication of the quality of an individual can directly be observed from the fitness value it has.

The setting of parameter values in a genetic programming system has a large impact on any possible premature convergence. Therefore some rules of thumb are taken into consideration. The used rules of thumb are abstracted from a parameter study done by Weibrenner described in [Weinbrenner] and are stated below:

- The faster the convergence in the first generations, the worst is the best achievable result, and vice versa.

- The population size should not fall below a certain minimum value. This minimum value depends on the problem. When extremely good results are needed, nothing is better than a large population size. For example, a population size of 500 which runs for 2,000 generations does not yield an equally good result than a population size of 2,000 run for 200 generations.

- Tournament selection performs better, except when running a lot of generations.

- The tournament size has a big effect. The smaller it is, the better the best achievable results and the slower convergence takes place.

Following the order of parameters presented in table 6.6 several settings have been chosen. The maximum generation is set to a number which a GP run almost never reaches, the GP run is mostly stopped by the user. Population size is set to 10,000, which is limited by the amount of RAM of the test platform. Initialization method and initialization depth decide what characteristics the population of the first generation will have. No individuals can grow larger than the maximum nodes parameter indicates.

Using lil-gp, it is possible to select different numbers and types of procedures for breeding the next generation. An exact description for possible parameters can be found in [Zongker]. As an example, the settings which comply to the settings Koza used in [Koza 1994] are shown.

```
breed_phases = 2
breed[1].operator = crossover, select=(tournament, size=2)
breed[1].rate = 0.80
breed[2].operator = reproduction, select=(tournament, size=2)
breed[2].rate = 0.20
```

The genetic programming system often is capable of finding satisfying results. Started is with a tournament selection, this often achieves the wanted results. If a problem is not solved immediately, then different settings are tried. Early characteristics of the expected quality of the evolutionary process can be deducted from the first rule of thumb: *the faster the convergence in the first generations, the worst is the best achievable result, and vice versa.* This makes it possible to perform a trial and error procedure on finding the best parameter set.

6.3 Results

This section gives an overview of the results achieved using the Ishibuchi rule extractor and the genetic programming rule extractor. To measure the results, both the Ishibuchi extractor and GP extractor are tested on data sets introduced in section 6.1. In case of the square darts data set and the one decides all data set, a complete circular motion of the research cycle is completed. This makes an assessment of the quality of the rule extractors possible. Detailed results in the form of rule sets are available in the appendix which starts at page 58.

Table 6.7: Ishibuchi extractor results.

Boolean	Square Darts	One decides all	Iris	Wine	Breast cancer
Nr of rules	13	5	9	17	n/a
Nr of antecedents	22	5	16	51	n/a
Nr of attributes	2	5	3	13	n/a
Accuracy (%)	100	100	97.3	100	n/a

6.3.1 Ishibuchi extractor

The procedure to extract rules by the Ishibuchi extractor consists out of executing the following four program modules sequentially:

ANN trainer $\rightarrow$ **ANN tester** $\rightarrow$ **Ishibuchi extractor** $\rightarrow$ **Rule set tester**

The first of four modules trains an ANN on the basis the parameters described in subsection 6.2.2. Trial and error determination is used to find a suitable set of parameters which enables the ANN achieve a very small learn error. This resulting neural network is tested with the ANN tester against the original data set. If the ANN is *not* able to correctly classify all points, then the maximum allowed learn error is lowered and the ANN training procedure is repeated. In the third phase, the Ishibuchi extractor module extracts a rule set from the tested neural network. Finally, the produced rule set is qualified by the rule set tester of which all characteristics are gathered and presented in table 6.7. Of this results the *number of rules*, *number of antecedents*, *number of attributes* and *accuracy* are used for measuring the quality. Comments on how these results are obtained are described separately for each data set below.

- **Square darts board data set.** This was the first test for the Ishibuchi rule extraction method. A neural network was created and trained on the Boolean typed version of the data set until the learn error was small enough. The ANN tester approved the first found neural net. After this the Ishibuchi rule extractor extracted a rule set which was identical to the original rule set presented in subsection 6.1.1. The certainty factors changed compared to the original, as can be observed in the result in the appendix. However, this change of CF does not affect the classifying behavior of the extraction method on this particular data set.

- **One decides all data set.** A neural network with 6 neurons in the hidden layer was adopted to the one decides all data set. The ANN tester approved the neural network and the Ishibuchi rule extractor was executed with a minimum of 1 don't care. Because of the larger input vector size, it was a rather exhaustive search. The resulting rule set matched the original rule set. As with the square darts board problem, the certainty factors changed compared to the original. But again, this had no effect on the classification quality.

- **Iris plants data set.** Training the neural network was a difficult task with the Iris plants data set, but succeeded. The Ishibuchi extractor was set to perform a rule set search and extracted over 200 rules with a 100% classification rate. But, these results were not like the results published in [Ishibuchi 1995]. So, the Ishibuchi extractor was set to perform another search with a minimum of 2 *don't cares*. The results found for this second run are a very close match to the results published in [Ishibuchi 1995]. Compared to the publication, two extra rules were found with a very low certainty factor. The final result of 9 rules has a 97.3% classification rate. This test confirmed the quality of trained neural network and the correctness of the implementation of the Ishibuchi rule extractor.

- **Wine data set.** Ishibuchi's rule extraction method works very well for low-dimensional pattern classification problems as described in [Ishibuchi 1995], but it can not be directly applied to high-dimensional problems. As this method searches all possible combinations of antecedent linguistic variables, the search space explodes with larger input vector sizes.

Table 6.8: GP extractor results on Binary data sets.

Boolean	Square Darts	One decides all	Iris	Wine	Breast cancer
Nr of rules	9	5	4	7	29
Nr of antecedents	9	5	4	7	56
Nr of attributes	2	5	2	6	9
Accuracy (%)	100	100	97.3	100	91

Ishibuchi tried to remedy this problem in [Ishibuchi 1999]. It is beyond the scope of this report to implement all these extra features into the Ishibuchi rule extraction method, so a description of Ishibuchi's adaptations and results will be given.

A problem for the Ishibuchi method is the combinatorial explosion of possible combinations of antecedent linguistic values. By restricting the number of possible combinations of antecedent linguistic values, only generic linguistic rules with a small number of antecedent conditions define a search space which is considerably smaller. Ishibuchi tried to extract linguistic rules with the length three or less, filling up the rest of the antecedent linguistic values with *don't cares*. An advantage is that shorter rules enhance the understandability of extracted knowledge. Some work is done on an improvement of the α-cuts (explained on page 18), because dividing each input vector in subintervals also leads to a combinatorial explosion. Instead of using the previously employed simple subdivision, hierarchical subdivision is utilized. Hierarchical subdivision uses variable sized subintervals, in which an internal vector is subdivided into only two subintervals at a time. This decreases the number of calculations to an acceptable level.

In this way, Ishibuchi examined $37,765$ fuzzy input vectors for generating linguistic rules with antecedent sizes of three or less. The number of extracted rules was $7,381$, which is not near human comprehensible. Also, these $7,381$ rules did have a 80.9% classification rate against the wine dataset, which was not satisfactory. A genetic algorithm based rule selection method was used for selecting only a small number of significant rules from the $7,381$ generated rules. The genetic algorithm selected 17 linguistic rules which had a 100% classification rate.

- **Breast cancer data set.** The Breast cancer data set has an input vector which is too large for the search space creation procedure utilized by the Ishibuchi rule extractor. Therefore this data set is excluded from the tests performed by the Ishibuchi extractor.

6.3.2 Genetic programming extractor

In the procedure used to extract rules with the GP rule extractor, the following two steps were performed:

GP extractor $\rightarrow$ Rule set tester

After setting population size, type of fitness function (Boolean or fuzzy) and selection functions, a genetic programming rule extraction can be started. The outcome of the Boolean fitness function can be compared to the classification rate of the individual. So, in case of Boolean data sets it is possible to directly determine the quality of the extracted rule set. If the fuzzy fitness function is employed, it is not straightforward to determine the classification rate of the extraction. But, to keep the test procedure consistent, results are always tested by the rule set tester module. Results of extracted rules from Binary data sets and fuzzy data sets are presented in table 6.8 respectively table 6.9. Comments on extracting rules using genetic programming are stated below.

- **Square darts board data set.** The GP rule extractor performed two rule extractions for the square darts board problem, the first on the Boolean data set type and the second on the fuzzy data set type. In the first GP run, a top individual with 100% classification rate

Table 6.9: GP extractor results on fuzzy data sets.

Fuzzy	Square Darts	One decides all	Iris	Wine	Breast cancer
Nr of rules	13	5	n/a	n/a	n/a
Nr of antecedents	22	5	n/a	n/a	n/a
Nr of attributes	2	5	n/a	n/a	n/a
Accuracy (%)	100	100	n/a	n/a	n/a

was found in generation 94. A few generations later, the individual has become smaller and after some editing[2] it resembled the rule set shown in the appendix. Remarkable is that not the original fuzzy rule set is found, but a 4 rule smaller one. The rule set tester pointed out that all 100 points of the data set are correctly classified. The second GP run, which operated on the fuzzy rule set, also found a 100% top individual. This top individual was very alike the original rule set presented in subsection 6.1.1, even the certainty factors come close to the original values.

- **One decides all data set.** As with the square darts board problem, two GP extractions were performed. Both the Boolean type and the fuzzy typed extractions resulted in the same, original rule set. The difference between the Boolean extraction and the fuzzy extraction can be found in the certainty factors in the extracted rules. The fuzzy extraction was *100% identical* to the original data set.

- **Iris plants data set.** The GP rule extraction method had a classification rate of 97.3% at generation 11 of the first run. Two generations later, the top individual was found which is presented in the appendix. Four rules using two attributes can correctly identify 146 out of 150 points of the data set. This means that only the attributes pedal length and pedal width are needed to do a near perfect classification.

- **Wine data set.** Despite the high-dimension ability, apparently the wine data set is a very easy one to adopt a fuzzy rule set to. The GP fuzzy rule extractor found a 100% classifying rule set within 10 generations and at generation 23 it had shrunk the solution to the rule set which is found in the appendix. Only 7 rules are needed to make a 100% classification of the wine data set. Even more interesting is the number of different antecedents which are needed. This validates the conclusion that only six attributes are needed to correctly identify the type of wine: alcohol, magnesium, flavanoids, color intensity, hue and proline.

- **Breast cancer data set.** This problem domain proved to be a very task for the GP rule extractor. Several GP runs were performed and it was not until the seventh run before a satisfying solution was found. A local maximum was reached in the other runs and after a study of the results of these runs, it is comprehensible why. Suppose a rule `IF x1 IS small THEN Class1` is found, which correctly identifies 98% of the points of its valid domain. Also suppose that this rule should be split up to three rules to make a 100% identification. Because of a fitness penalty for larger rule sets, this split up to three rules should happen at once. A GP rule extractor can come to a correct solution, since many individuals contain double rules which have not yet been removed by the evolutionary process. The classification rate of 91% has not been matched by any other knowledge representation form but the data set.

6.4 Discussion of experiments

One of the bigger differences between the Ishibuchi extractor and its genetic programming counterpart does not fit in the classification scheme of Andrews. Whilst the genetic programming rule extractor is capable of extracting rules from plain data sets, Ishibuchi's rule extractor is not.

[2] *Editing* is altering the solution, removing bloat, while its fitness remains the same.

Ishibuchi's rule extractor uses the generalizing power of neural networks to be able to represent a data set in the form of fuzzy rules. For a genetic programming rule extractor, a neural network is just a form of noise applied on the original data set. This difference goes on with the rule-by-rule extraction of the Ishibuchi extractor and the complete rule set extraction of the genetic programming extractor. Because of the qualification on a per-rule basis by the Ishibuchi extractor, this method is not able to extract the most compact rule sets which are able to correctly represent the data set. Evidence for this can be found in tables 6.7, 6.8 and 6.9 under *number of rules*, *number of antecedents* and *number of attributes*. The GP extractor consequently extracts more compact solutions.

Since the Ishibuchi extractor uses a brute-force approach in its method, it is very weak in extracting rules from high-dimensional problems. Additional modifications as introduced in [Ishibuchi 1999] only "moves" the problem a few dimensions higher. With larger attribute sizes, the Ishibuchi rule extractor suffers from a combinatorial explosive search state space, which makes the method unable to use it for making knowledge explicit for larger problems. The GP extractor proved to be able to handle higher dimensional problems with no additional modifications.

Often the generalizing power of symbolic knowledge extraction method is regarded as a weak point. The genetic programming extractor demonstrated it can find compact solutions. The more compact a representation form is, the higher the generalizing power of an extraction method. The breast cancer data set test revealed that the GP extractor also is able to find solutions to very noisy problem domains. The previous classification record on the breast cancer data set was improved from 78% to 91%.

If the fitness function does bring forth a smooth surface, the GP extractor can halt at local minima or maxima. In this case, it can be difficult to find a satisfying result. Remedies are executing many runs with variation of the initial random seed or introducing an active mutation genetic operator. The Ishibuchi rule extractor guarantees a solution, which is something that can not be guaranteed by the genetic programming rule extractor. However, using the fuzzy fitness function it was able to find solutions which were near identical to the original. The landscape of the fuzzy fitness function is very smooth which resulted in better and more consequent rule extractions.

Chapter 7

Conclusion

The performed investigation has shown that genetic programming can be used to create a tool which can make knowledge explicit. For the purpose of knowledge acquisition, it is able to "translate" raw data sets to the understandable form of a fuzzy rule set. It facilitates manipulation of the linguistic variables directly associated with the problem and it allows for a population of fuzzy rule sets of various sizes and dimensions.

The literature study revealed a glimpse of what is to be expected from the field of genetic programming in the near future. The study was performed in such content that it was possible to put the theory of genetic programming into practice. The introduction to the field of fuzzy logic controllers was satisfactory for the purpose of the investigation.

The design of the research cycle proved to be useful, it allowed for a qualitative judgment of the rule extractors. It uses knowledge state changers to switch back and forth between two knowledge representations, namely data sets and fuzzy rule sets. The research cycle enables a user to create a reference framework for the expected answer of a knowledge state changer such as a rule extractor. The implementation and integration of software representing the research cycle resulted in a workbench. This workbench was able to fulfill the demands of the tasks of the research cycle. It produced results on which conclusions can be drawn about the tested rule extraction methods.

A number of fuzzy rule sets have been evolved by the genetic programming rule extractor whose performance have been found to compare favorably with that of a rule set extracted by the Ishibuchi rule extraction method. Genetic programming rule extraction was able to produce sets of fuzzy rules which require substantially fewer rules and less attributes than Ishibuchi's method and still is able to correctly classify an equal near-perfect amount of objects. It is capable of dealing with high dimensional rule extraction problems with no additional heuristics. The fitness functions were chosen in such way that the GP extractor repeatedly found those spots of the search space in which good answers could be found. In some cases when using the Boolean fitness function, the GP system suffered from premature convergence and got stuck at local minima. If the spectrum of fitness values is not sufficiently rich, the fitness function may not provide enough information to guide GP towards regions of the search state space where improved solutions might be found.

Ishibuchi's method relies on the neural network for its generalizing capabilities. This means that the generalization capabilities of Ishibuchi's rule extractor can be at most as good as that of the neural net. Without the use of both a train and test data set when training a neural net, not much can be said about the quality of the associative capabilities besides the train set. As shown in the results, the GP method is capable of removing attributes which are redundant for object classification. The generalizing power of the GP rule extractor is encapsulated within one of the goals of the fitness function: "the shorter the rule set, the better". Differences between the generalizing power of GP and a neural network is beyond the scope of this investigation.

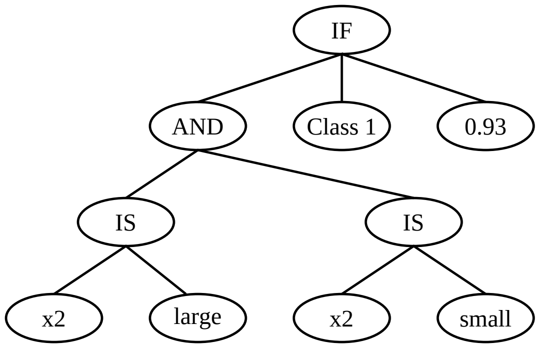

Figure 7.1: "Correct" rule.

7.1 Recommendations

The investigation showed that premature convergence can take place in case the fitness function only produces a small variety of values. Much research is being done by all types of evolutionary computing to improve the fitness function. However, in case of genetic programming it is not only the fitness function which is responsible for premature convergence. It is a combination of the defined environment, the fitness function(s), all genetic operators (such as the cross-over operator), et cetera, which determine the stability of a GP system. Research can be done on almost all aspects of genetic programming. This already resulted in variants of genetic programming ranging from cellular GP to GP *without* fitness functions.

A lacking feature of the currently used fitness functions is to take into account the total number of different attributes used by an individual. For example, consider two rule sets of equal sizes and both rule sets correctly identify an equal amount of data set points. Rule set 1 makes use of x_1 and x_5 while rule set 2 deploys x_2, x_3 and x_5. In this case, rule set 1 should be favored because it is using less elements of the input vector. Using several fitness functions can be achieved when using co-evolutionary fitness-switching. At the moment of implementation, no good software support could be found for co-evolutionary fitness switching.

Besides co-evolutionary fitness switching, another approach for using several fitness functions can be used, which is described next. For the simulated evolution of a population, a fitness function is used for ordering the individuals. Mostly, a rank is assigned using the fitness function and selection type (such as tournament selection) which influence the chances of reproduction, cross-over, et cetera. Suppose several ranks are assigned to an individual using several fitness functions. For the example case of the two rule sets mentioned earlier, rule set 1 is assigned fitness values $(85, 10, 2)$ and rule set 2 is assigned fitness values $(85, 10, 3)$. Now the genetic programming system can make a decision on the quality of these individuals using the last values 2 and 3.

The syntactical constraints generated by the BNF description of the GP rule extractor structure proved not to be strong enough to prevent useless rules to be formed. For example, the rule `IF x2 IS large AND x2 IS small THEN Class1 WITH CF 0.93`, which can be observed in figure 7.1, is correctly formed in accordance to the construction rules of the rule language. The antecedent part of this rule makes sure the rule *never* fires. To remedy this, a feature is needed which invokes more constraints on the construction of the individual. This brings the needed restrictions for a GP creature to such a level that even the currently used strongly typed genetic programming (STGP) is not sufficient. For example, only strong typing the input and output of functions and terminals is not strong enough to prevent a rule being present twice in a rule set. If another construction type is able to create structures without bloat, a considerable smaller search state space is realized. This can result in more effective searches by the genetic programming system. It is not without a reason that structure formation is one of the most active areas of research on the field of genetic programming.

One flaw in the design of the rule extractors is the consequent part in the used fuzzy rule design:

`If` x_1 `is` A_1 `and .. and` x_m `is` A_m `then` $Class\,h$ `with CF` c

Using just "$Class\,h$" is not in accordance to the standards of fuzzy rules. To comply to the standard, it can be adjusted to $Class\,h$ `IS high`. Using this added functionality the rule only produces output to the linguistic value *high* but it does not use the real power of *fuzzy rules*. It uses the antecedent part of the rule to separate classes in the attribute space and points residing in marginal areas are separated using the certainty factor. It is possible to add more possibilities to the consequent linguistic values, for example $Class\,h$ `IS low`. In this example, a fuzzy rule set becomes capable of excluding out-biters by adding one rule, in stead of having to work around out-biters using many rules. This example illustrates that it is possible to extend the projectional power of a fuzzy logic controller.

Another approach to the problem of finding a suitable design of the fuzzy rule structure, is to automate it. For example, in stead of performing a GP search using only the predefined linguistic values *small..large*, a GP system can be designed which is able of evolving of linguistic values. It chooses a number of linguistic values and searches membership functions of all chosen linguistic values. It is imaginable that some problem data sets only need three linguistic values *low, little high, high* with bell shaped membership functions to achieve a 100% classification.

If more compact rule representations are needed, more different types of operators can be added to the antecedent part of the fuzzy rules. For example, if a fuzzy `OR` operator is added to the possibilities, it becomes possible to represent two "old" rules using one "new". Not only this, it would also become possible to represent rule sets with just as many rules as output classes. This can result in large antecedent parts of the rules which is not good for the comprehensibility.

A fuzzy rule set can grow to a size which is too complex to understand for humans. An example of this is the output generated by the genetic programming rule extractor on the basis of the breast cancer data set, which consists out of 29 rules. An other knowledge representation form than fuzzy rules which remains comprehensible for larger sizes is a *decision tree*. A (binary) decision tree starts in the root tree node and evaluates its nodes downwards until it reaches a leaf node which holds the answer. The tree evaluation depends on these decision nodes, which mostly consist out of *if-then* functions built up with inequality functions. For example, consider the breast cancer data set. A decision tree evaluation could have decision nodes holding decisions like $age > 40$?. After evaluation a leaf node returns either *class* 1 or *class* 2 for indicating the output class. A difference between the capability of a fuzzy logic controller and a decision tree is that a decision tree is unable to project on more than one output (class). However, this difference can be solved by using several decision trees for each possible output class. It is also possible to introduce functions in the leaf nodes in a decision tree, which makes it an even more powerful knowledge representation form.

Bibliography

[Andrews] Andrews, R. *An evaluation and comparison of techniques for extracting and refining rules from artificial neural networks.* Neurocomputing Research Centre, Queensland University of Technology, 1996.
`http://www.fit.qut.edu.au/~robert/`

[Angeline] Angeline, P. *Evolutionary Algorithms and Emergent Intelligence.* In Kinnear, Jr., K.E., editor, *Advances in Genetic Programming,* chapter 4, pp. 75-98. MIT press, Cambridge, MA, 1994.
`http://www.natural-selection.com/people/pja/`

[Banzhaf] Banzhaf, W. *Genetic Programming: An introduction. On the automatic evolution of computer programs and its applications.* San Francisco : Morgan Kaufmann, 1998. ISBN 1-55860-510-X.
`http://ls11-www.informatik.uni-dortmund.de/people/banzhaf/`

[Blake] Blake, C. *UCI Repository of machine learning databases,* `http://www.ics.uci.edu/~mlearn/mlrepository.html`. University of California, Department of Information and Computer Science. Irvine, CA, 1998.
`http://www.ics.uci.edu/~cblake/`

[Cestnik] Cestnik, G. *Assistant-86: A Knowledge-Elicitation Tool for Sophisticated Users.* In I. Bratko & N. Lavrac, editors, *Progress in Machine Learning*, pp. 31-45, Sigma Press, 1987.

[Darwin] Darwin, C. *The Origin of Species by Means of Natural Selection or The Preservation of Favoured Races in the Struggle for Life.* London: Murray, 1859.

[Dubois] Dubois, D. *Ranking Fuzzy numbers in the Setting of Possibility Theory*, information sciences 30, pp. 183-224, 1983.
`http://www.irit.fr/activites/rpdmp/didier.html`

[Fogel] Fogel, L. *Artificial Intelligence through simulated evolution.* New York: Wiley, 1966.

[Geyer-Schultz] Geyer-Schulz, A. *Fuzzy Rule-Based Expert Systems and Genetic Machine Learning*, Studies in Fuzziness Vol. 3, Heidelberg 1996.

[Gottlob] Gottlob, G. *StarFLIP++, A reusable iterative optimization library for combinatorial problems with fuzzy constraints.* Technical Report DBAI-TR-97-11, Institute of Information Systems (E184-2), Vienna University of Technology, 1997.
`http://www.dbai.tuwien.ac.at/staff/gottlob/`

[Gruau] Gruau, F. *Genetic Micro Programming of Neural Networks.* In Kinnear, Jr., K.E., editor, *Advances in Genetic Programming,* chapter 24, pp. 495-518. MIT press, Cambridge, MA, 1994.

[Haak] Haak, A. *De menselijke maat*, Delftse Universitaire Pers, 1994. ISBN 90-6275-048-6.

[Haynes] Haynes, D. *Type Inheritance in Strongly Typed Genetic Programming.* In Angeline, P.J. and Kinnear, Jr., K.E., editors, *Advances in Genetic Programming 2,* chapter 18, pp. 359-376. MIT Press, Cambridge, MA, 1996.
`http://adept.cs.twsu.edu/~thomas/`

[Hecht-Nielsen] Hecht-Nielsen, R. *Neurocomputing.* Addison-Wesley Publishing Company, 1991. ISBN 0-201-09355-3.

[Holland] Holland, J. *Adaptation in natural and artificial systems; an introductory analysis with applications to biology, control, and artificial intelligence.* Ann Arbor: University of Michigan Press, 1975.

[Hörner] Hörner, H. *The Vienna University of Economics Genetic Programming Kernel.* The Internet, `http://aif.wu-wien.ac.at/%7Egeyers/archive/gpk/vuegpk.html`, 1997.

[Iba 1994] Iba, H. *Genetic programming using a minimum description length principle.* In Kinnear, Jr., K.E., editor, *Advances in Genetic Programming,* chapter 12, pp. 265-284. MIT press, Cambridge, MA, 1994.
`http://www.miv.t.u-tokyo.ac.jp/~iba/`

[Iba 1996] Iba, H. *Extending genetic programming with recombinative guidance.* In Angeline, P.J. and Kinnear, Jr., K.E., editors, *Advances in Genetic Programming 2,* chapter 4, pp. 69-88. MIT Press, Cambridge, MA, 1996.
`http://www.miv.t.u-tokyo.ac.jp/~iba/`

[Ishibuchi 1995] Ishibuchi, H. *Generating Fuzzy If-Then Rules from Trained Neural Networks: Linguistic Analysis of Neural Networks.* 1996 Proceedings of IEEE International Conference on Neural Networks, pp. 1133-1138, 1996.

[Ishibuchi 1999] Ishibuchi, H. *Linguistic rule extraction from neural networks for high-dimensional classification problems.* Complexity International Volume 6, 1999. ISSN 1320-0682.

[Jamshidi] Jamshidi, M. *Fuzzy logic and control; software and hardware applications,* 1993. ISBN 0-13-334251-4.

[Jou] Jou, H. *Scientific Applications on Linux.* The Internet, `http://sal.kachinatech.com/`, 2000.
`http://www.kachinatech.com/~hjjou/`

[Kaufmann] Kaufmann, A. *Introduction to fuzzy arithmetic, Theory and Applications.* Van Nostrand and Reinhold, New York 1991. ISBN 0-442-00899-6pbk.

[Kinnear] Kinnear, K. *Alternatives in Automatic Function Definition: A comparison of Performance.* In Kinnear, Jr., K.E., editor, *Advances in Genetic Programming,* chapter 6, pp. 119-141. MIT press, Cambridge, MA, 1994.

[Klawonn] Klawonn, F. *Generating Rules from Data by Fuzzy and Neuro-Fuzzy Methods.* Department of Computer Science, Technical University of Braunschweig. Proceedings of the third German GI-Workshop "Fuzzy-Neuro-Systeme'95", Darmstadt, Germany, 1995.

[Koza 1992] Koza, J. *Genetic programming; on the programming of computers by means of natural selection.* MIT Press, Cambridge, MA, 1992. ISBN 0-262-11170-5.
`http://smi-web.stanford.edu/people/koza/`

[Koza 1994] Koza, J. *Genetic programming II; automatic discovery of reusable programs.* MIT Press, Cambridge, MA, 1994. ISBN 0-262-11189-6.
http://smi-web.stanford.edu/people/koza/

[Langley] Langley, P. *Elements of Machine Learning.* Morgan Kaufmann, San Fransisco, CA., 1996.
http://robotics.stanford.edu/users/langley/

[Luke] Luke, S. *Sean's patched lil-gp kernel.* The Internet, http://www.cs.umd.edu/users/seanl/gp/patched-gp/, 1998.

[Meier] Meier, S. *The K Development Environment.* The Internet, http://www.kdevelop.org, 2000.

[Naur] Naur, P. *Revised Report on the Algorithmic Language ALGOL 60.* Communications of the ACM, vol. 3, no. 5, pp. 299-314, 1960.

[Nikolaev] Nikolaev, N. *Inductive Genetic Programming with Immune Network Dynamics.* In Lee Spector et al., editors, *Advances in Genetic Programming 3*, chapter 15, pp. 355-376. MIT Press, Cambridge, MA, 1999.
http://www.aubg.bg/faculty/cs/nikolaev/

[Nordin 1994] Nordin, P. *A Compiling Genetic Programming System that Directly Manipulates the Machine Code.* In Kinnear, Jr., K.E., editor, *Advances in Genetic Programming,* chapter 14, pp. 311-331. MIT press, Cambridge, MA, 1994.
http://ls11-www.informatik.uni-dortmund.de/people/nordin/

[Nordin 1996] Nordin, P. *Explicitly defined introns and destructive crossover in genetic programming.* In Angeline, P.J. and Kinnear, Jr., K.E., editors, *Advances in Genetic Programming 2,* chapter 6, pp. 111-134. MIT Press, Cambridge, MA, 1996.
http://ls11-www.informatik.uni-dortmund.de/people/nordin/

[Rechenberg] Rechenberg, I. *Cybernetic solution path of an experimental problem.* Royal Aircraft Estabilishment Translocation No. 1122, Farnborough Hants: Ministry of Aviation, Royal Aircraft Estabilishment.
http://www.bionik.tu-berlin.de/user/rechenb/rechenb.html

[Ross] Ross, S. *Probability models, the fifth edition.* Academic Press Inc., Berkeley, California, 1993. ISBN 0-12-598455-3.

[Ryan] Ryan, C. *Pygmies and Civil Servants.* In Kinnear, Jr., K.E., editor, *Advances in Genetic Programming,* chapter 11, pp. 243-263. MIT press, Cambridge, MA, 1994.
http://www.csis.ul.ie/staff/conorryan/

[Stroustrup] Stroustrup, B. *The C++ Programming Language.* AddisonWesley, 1997. ISBN 0-201-88954-4.
http://www.research.att.com/~bs/

[Teller] Teller, A. *Evolving programmers: the co-evolution of intelligent recombination operators.* In Angeline, P.J. and Kinnear, Jr., K.E., editors, *Advances in Genetic Programming 2,* chapter 3, pp. 45-68. MIT Press, Cambridge, MA, 1996.
http://www.cs.cmu.edu/~astro/

[Torvalds] Torvalds, L. *Linux.* The Internet, http://www.linux.com, 2000.
http://www.cs.helsinki.fi/u/torvalds/

[Weinbrenner] Weinbrenner, T. *GPC++, Genetic Programming C++ Class Library.* The Internet, http://thor.emk.e-technik.tu-darmstadt.de/%7ethomasw/gp.html, 1997.

[Zadeh] Zadeh, L. *Fuzzy sets*, Information and Control vol. 12, 1965. pp. 338-353.

[Zhang] Zhang, B. *Co-evolutionary fitness switching: learning complex collective behaviours using genetic programming.* In Lee Spector et al., editors, *Advances in Genetic Programming 3*, chapter 18, pp. 425-445. MIT Press, Cambridge, MA, 1999.
`http://cse.snu.ac.kr/~btzhang/`

[Zongker] Zongker, D. *Lil-gp.* Computer science departement, Michigan State University, 1995.

Appendix

Results Ishibuchi rule extractor

Square darts board

```
IF x1 IS small THEN Class1 WITH CF 0.44
IF x1 IS large THEN Class1 WITH CF 0.38
IF x2 IS small THEN Class1 WITH CF 0.47
IF x2 IS large THEN Class1 WITH CF 0.42
IF x1 IS medium_small AND x2 IS medium_small THEN Class2 WITH CF 0.40
IF x1 IS medium_small AND x2 IS medium THEN Class2 WITH CF 0.45
IF x1 IS medium_small AND x2 IS medium_large THEN Class2 WITH CF 0.39
IF x1 IS medium AND x2 IS medium_small THEN Class2 WITH CF 0.34
IF x1 IS medium AND x2 IS medium_large THEN Class2 WITH CF 0.42
IF x1 IS medium_large AND x2 IS medium_small THEN Class2 WITH CF 0.38
IF x1 IS medium_large AND x2 IS medium THEN Class2 WITH CF 0.32
IF x1 IS medium_large AND x2 IS medium_large THEN Class2 WITH CF 0.45
IF x1 IS medium AND x2 IS medium THEN Class3 WITH CF 0.42
```

One decides all

```
IF x1 IS small THEN Class1 WITH CF 0.43
IF x1 IS medium_small THEN Class2 WITH CF 0.36
IF x1 IS medium THEN Class3 WITH CF 0.39
IF x1 IS medium_large THEN Class4 WITH CF 0.47
IF x1 IS large THEN Class5 WITH CF 0.43
```

Iris plants

```
IF x3 IS large THEN Class3 WITH CF 0.98
IF x4 IS large THEN Class3 WITH CF 0.90
IF x3 IS small AND x4 IS small THEN Class1 WITH CF 0.03
IF x3 IS medium AND x4 IS medium THEN Class2 WITH CF 0.31
IF x2 IS small AND x3 IS medium_large THEN Class3 WITH CF 0.40
IF x2 IS small AND x4 IS medium_large THEN Class3 WITH CF 0.65
IF x2 IS medium_small AND x4 IS medium_large THEN Class3 WITH CF 0.39
IF x3 IS medium_large AND x4 IS small THEN Class3 WITH CF 0.74
IF x3 IS medium_large AND x4 IS medium_small THEN Class3 WITH CF 0.49
```

Results GP rule extractor

Square darts board, Boolean

```
IF x1 IS small THEN Class1 WITH CF 0.63
```

```
IF x1 IS large THEN Class1 WITH CF 0.87
IF x2 IS large THEN Class1 WITH CF 0.97
IF x2 IS small THEN Class1 WITH CF 0.89
IF x1 IS medium_large THEN Class2 WITH CF 0.69
IF x1 IS medium_small THEN Class2 WITH CF 0.66
IF x2 IS medium_small THEN Class2 WITH CF 0.41
IF x2 IS medium_large THEN Class2 WITH CF 0.35
IF x2 IS medium THEN Class3 WITH CF 0.37
```

Square darts board, Fuzzy

```
IF x1 IS small THEN Class1 WITH CF 0.99
IF x1 IS large THEN Class1 WITH CF 0.97
IF x2 IS small THEN Class1 WITH CF 1.00
IF x2 IS large THEN Class1 WITH CF 0.98
IF x1 IS medium_small AND x2 IS medium_small THEN Class2 WITH CF 0.99
IF x1 IS medium_small AND x2 IS medium THEN Class2 WITH CF 0.96
IF x1 IS medium_small AND x2 IS medium_large THEN Class2 WITH CF 0.98
IF x1 IS medium AND x2 IS medium_small THEN Class2 WITH CF 0.97
IF x1 IS medium AND x2 IS medium_large THEN Class2 WITH CF 0.99
IF x1 IS medium_large AND x2 IS medium_small THEN Class2 WITH CF 1.00
IF x1 IS medium_large AND x2 IS medium THEN Class2 WITH CF 0.99
IF x1 IS medium_large AND x2 IS medium_large THEN Class2 WITH CF 0.98
IF x1 IS medium AND x2 IS medium THEN Class3 WITH CF 0.99
```

One decides all, Boolean

```
IF x1 IS small THEN Class1 WITH CF 0.79
IF x1 IS medium_small THEN Class2 WITH CF 0.84
IF x1 IS medium THEN Class3 WITH CF 0.80
IF x1 IS medium_large THEN Class4 WITH CF 0.81
IF x1 IS large THEN Class5 WITH CF 0.79
```

One decides all, Fuzzy

```
IF x1 IS small THEN Class1 WITH CF 1.00
IF x1 IS medium_small THEN Class2 WITH CF 1.00
IF x1 IS medium THEN Class3 WITH CF 1.00
IF x1 IS medium_large THEN Class4 WITH CF 1.00
IF x1 IS large THEN Class5 WITH CF 1.00
```

Iris plants

```
IF x3 IS small THEN Class1 WITH CF 0.58
IF x3 IS medium THEN Class2 WITH CF 0.81
IF x3 IS medium_large THEN Class3 WITH CF 0.31
IF x4 IS medium_large THEN Class3 WITH CF 0.81
```

Wine

```
IF x13 IS medium_large THEN Class1 WITH CF 0.65
IF x7 IS medium THEN Class1 WITH CF 0.22
IF x1 IS medium_small THEN Class2 WITH CF 0.45
IF x5 IS small THEN Class2 WITH CF 0.90
```

```
IF x10 IS small THEN Class2 WITH CF 0.72
IF x7 IS small THEN Class3 WITH CF 0.63
IF x11 IS small THEN Class3 WITH CF 0.62
```

Breast cancer

```
IF x1 IS medium THEN Class1 CF 0.76
IF x3 IS medium_small THEN Class1 CF 0.92
IF x4 IS small THEN Class1 CF 0.64
IF x4 IS medium_large THEN Class1 CF 1.00
IF x5 IS large THEN Class1 CF 0.69
IF x8 IS medium THEN Class1 CF 0.86
IF x2 IS large AND x3 IS medium_large AND x5 IS medium THEN Class1 CF 0.86
IF x2 IS medium AND x6 IS medium AND x8 IS small THEN Class1 CF 1.00
IF x2 IS medium AND x3 IS medium THEN Class1 CF 0.67
IF x1 IS medium_small AND x8 IS medium_large THEN Class1 CF 0.97
IF x4 IS medium_small AND x8 IS medium_large THEN Class1 CF 0.97
IF x4 IS small AND x6 IS medium AND x8 IS small THEN Class1 CF 1.00
IF x3 IS large THEN Class2 CF 0.58
IF x5 IS small THEN Class2 CF 0.22
IF x6 IS large THEN Class2 CF 0.48
IF x7 IS small THEN Class2 CF 0.43
IF x8 IS large THEN Class2 CF 0.85
IF x1 IS medium_large AND x7 IS large THEN Class2 CF 0.85
IF x1 IS large AND x9 IS large THEN Class2 CF 0.97
IF x2 IS small AND x5 IS large THEN Class2 CF 0.97
IF x3 IS medium AND x6 IS large AND x7 IS small THEN Class2 CF 0.97
IF x3 IS medium AND x8 IS medium_large THEN Class2 CF 0.97
IF x3 IS large AND x9 IS large THEN Class2 CF 0.97
IF x8 IS medium AND x9 IS large THEN Class2 CF 0.97
IF x1 IS medium_small AND x3 IS medium_large AND x6 IS large THEN Class2 CF 0.97
IF x1 IS medium_small AND x6 IS small AND x8 IS small THEN Class2 CF 0.97
IF x1 IS medium_small AND x7 IS small AND x8 IS small THEN Class2 CF 0.97
IF x2 IS large AND x6 IS large AND x8 IS medium_large THEN Class2 CF 0.97
IF x3 IS medium_small AND x5 IS medium AND x6 IS large THEN Class2 CF 0.97
```