Classifier Systems

A useful approach to machine learning?

Master's Thesis

by

Bart de Boer
Leiden University

August 31, 1994

Preface

This thesis was written as a final project in artificial intelligence research for the subject of applied computer science at the university of Leiden (Rijksuniversiteit Leiden), the Netherlands, internal report number IR94-02. It was written from august 1993 until march 1994 and was supervised by dr. Ida Sprinkhuizen-Kuyper and drs. Egbert Boers without whose support, suggestions and critique I would never have been able to do what I have done.

I also want to thank Jos de Graaff for correcting the mistakes in my english. Of course all the remaining errors are entirely my responsibility.

Abstract

Classifier systems are sub-symbolic or dynamic approaches to machine learning. These systems have been studied rather extensively. In this thesis some theoretical results about the long-term behaviour and the computational abilities of classifier systems are derived. Then some experiments are undertaken. The first experiment entails the implementation of a simple logic function, a multiplexer in a simple classifier system. It is shown that this task can be learned very well.

The second task that is taught to the system is a mushroom-classification problem that has been researched with other learning systems. It is shown that this task can be learned. The last problem is the parity problem. First it is shown that this problem does not scale linearly with its number of bits in a straightforward classifier system. An attempt is made to solve it with a multi-layer classifier-system, but this is found to be almost impossible. Explanations are given of why this should be the case.

Then some thought is given to analogies between classifier systems and neural networks. It is indicated that there are mappings between certain classifier systems and certain neural networks. It is suggested that this is a main concern for future classifier systems research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The differences between humans and computers are huge. Where computers are able to perform repetitive tasks with astounding accuracy and speed, humans are good at adapting to circumstances they have never encountered before and at finding solutions to completely new problems. Adaptation and finding solutions are closely related to the ability to learn. If computers are to perform well in the same areas as humans, it would be a good idea to investigate machine learning.

Therefore machine learning is a major topic in artificial intelligence research. There are many ways in which machines can learn. This thesis is concerned with one of these methods, John Holland's classifier system with bucket brigade, see for example [HOL75, HOL80, BGH89, GB89].

First in chapters 2, 3 and 4 an introduction is given into what machine learning is, after which the focus will be on what classifier systems are and how they learn. Then some results of mathematical research are presented. In chapter 5 a formal definition of a classifier system is attempted. A formal definition is something which is almost never given in papers about classifier systems. This is unfortunate, because such a definition provides a tool with which the research and the implementation of classifier systems is made much easier.

Then a mathematical analysis of the behaviour of a classifier system is undertaken in chapter 6. The results of this analysis are applied in a practical way by calculating parameters of the systems used in the experiments.

After that a method with which the computational strength of classifier systems can be investigated is presented in chapter 7. This method proves to be useful in determining whether adding features to a classifier system really increases its computational capabilities or if they are just superfluous.

In the practical part of this thesis several problems are taught to the classifier system. In chapter 8 an experiment done by Goldberg [GB89] is repeated in order to show that a classifier system is able to learn and how it learns. The problem is very simplistic however, and many difficulties that occur with larger

tasks do not appear here.

Chapter 9 is concerned with more serious problems. Two different tasks are taught with two different purposes. The first task is to learn to distinguish between edible and poisonous mushrooms. This can be learnt by a very simple classifier system, but involves quite a lot of information, so the classifiers used will be rather large.

The second problem is an eight-bit parity problem. This cannot easily be solved by a simple classifier system, so it is used to test the different learning-abilities of different classes of classifier-systems.

In the last chapter of this thesis an idea is put forward on how to implement classifier systems in a more efficient way by looking at them as a kind of activation-networks. No experiments were done, but some light is shed on how an implementation could be made and a discussion of speedup is presented. Also a few thoughts are given to the similarities between neural networks and classifier systems.

Chapter 2

Learning

One aspect that is usually associated with intelligence is a capability to adapt or to learn. In the study of artificial intelligence this has led to a large amount of research into learning systems. It is one of the classical areas of interest of AI. The research has not been entirely academic, though. Learning systems have been applied in many different domains.

2.1 Definitions

Before we can talk any further about learning systems, we must try to find a definition of learning. Different authors have used different, but similar definitions. They have also used different criteria to classify learning systems and different perspectives on which to base their research.

Narendra and Thathachar use the following, rather behaviouralist definition: *"Learning is the ability of systems to improve their responses based on past experience"* [NT89]. In the same book they give a more technical definition as well: *"Researchers [...] have consistently used the term 'learning automaton' to describe both deterministic and stochastic schemes used in discrete systems that improve their performance in random environments"*. Michalski, Carbonell and Mitchell define learning from a more cognitive point of view: *"Learning processes include the acquisition of new declarative knowledge, the development of motor and cognitive skills through instruction and practice, the organization of new knowledge into general, effective representations, and the discovery of new facts and theories through observation and experimentation"* [MCM83b]. Simon, a psychologist, uses a similar definition to Narendra and Thathachar's: *"Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same tasks drawn from the same population more efficiently and effectively the next time."*[SIM83] Hilgard and Bower, in their standard work on learning theories in psychology, avoid the subject alto-

gether: *"The controversy is over fact and interpretation, not over definition.[...] Learning is one of those loose "open-textured" concepts that include a variety of different species. [...] ...in casual conversation, it is satisfactory to continue to mean by learning that which conforms to the usual socially accepted meaning that is part of our common heritage"* [HIBO75].

These definitions have in common an improvement in the behaviour towards an environment, originating from repeated interactions with that environment. But whereas Narendra, Thathachar and Simon define learning from a behaviouralist point of view by looking only at the changing behaviour of a system, Carbonell, Michalski and Mitchell define it in a broader sense, including changes in a system which may not be directly apparent to the outside world. This last way of regarding learning is often used in artificial intelligence research, but in this paper we will mainly be considering the behaviour of systems.

It is also possible to give a more formal definition of a reinforcement learning system. For this we will need to define the learning system and its environment. We will need two sets: $\mathcal{A}$, the set of states of the environment and $\mathcal{B}$, the set of actions the learning system can perform on the environment. The learning system can now be defined as a (possibly stochastic) function $\psi = M(\phi, t)$ in which $\psi \in \mathcal{B}$ the action the learning system performs, $\phi \in \mathcal{A}$ the state of the environment and $t \in \mathbf{N}^+$ the time at which this happens.

The reaction of the environment can now be described by the (possibly stochastic) function $p = F(\psi, \phi, t)$ in which ψ, ϕ and t are the same as above and $p \in \mathbf{R}^+$ the payoff of the environment. This is a measure of how good the action of the learning system was. The higher p, the better.

A system can now be said to learn if it is true that:

$$\langle F(M(\phi_t, t), \phi_t, t)\rangle > \langle F(M(\phi_{t-1}, t-1), \phi_{t-1}, t-1)\rangle$$

in which $\langle\rangle$ is used to denote the expected value of a function. The variables ϕ_t and ϕ_{t-1} are the states of the environment at times t respectively t-1.

Note that this is not a very accurate definition of a learning system and that it doesn't cover every possible learning system. In particular it only covers monotonically learning stimulus-response systems. However, to get an idea of what is meant by a learning system in this paper it is sufficient.

2.2 Ways of learning

Many different approaches have been used in machine learning. These depended on the task to be learnt, the way in which the task was taught and on what was fashionable at the time the research was done.

The first classification of learning systems that we can make is on how the material to be learnt is presented to the system. Michalski, Carbonell and Mitchell [MCM83b] use the following classes, ordered approximately in descending need of supervision from a teacher: rote learning and direct implementation

of new knowledge, learning from instruction, learning by analogy, learning from examples and learning from observation and discovery.

Rote learning and direct implementation is the least interesting way of learning; it amounts to directly inserting knowledge into a system, either by programming it or putting it into a database. The system itself does nothing with the knowledge, except for executing it or storing and reproducing it.

Learning from instruction is more interesting. It is very much like education at school. The learning system must be able to understand the instruction it gets, store it and integrate it with what it already knows. This is a very interesting artificial intelligence problem, but the learning problems are relatively minor.

The third category, learning from analogy, gives a learning system more difficulties. It has to find in its knowledge something similar to the task to be learnt and change the knowledge already present until it is applicable to the situation at hand. Then it has to store this newly acquired knowledge in its knowledge base.

Learning from examples is the kind of learning that will be discussed in this paper. The system is presented with samples or examples from an environment, together with information to associate with this example. This information can be an indication whether an example is positive or negative, whether the response of the system was good or bad, or it can be some action to associate with the example. If the information is given at the same time as the example, it is called "true learning with examples". If the information is given after the system has generated a response, it can be called *reinforcement learning*, which is exactly the kind of learning that will be discussed in this paper.

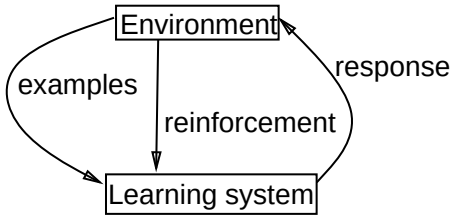

Figure 2.1: Reinforcement learning

The last class, learning from observation and discovery or unsupervised learning, is a kind of learning in which the learning system is left on its own to explore its environment and try to make classifications of phenomena it sees or form theories about it.

2.2.1 Environments

Another way to classify learning systems is on the kind of task or environment it learns to operate. The environment can be static or dynamic, deterministic or stochastic and discrete or continuous. A static environment does not change during the time a learning system is active, a dynamic environment does.

A deterministic environment always gives the same response in the same situation and it never gives examples that are flawed. Stochastic environments do not have these properties. They can have inherent stochastic features, (as in quantum-dynamic systems, gambling problems or predictions of real-life phenomena, like the weather) or be troubled by noise.

In finite environments, there is a finite number of actions a learning system can take and a finite number of responses it can get. In a continuous environment either the number of actions is infinite, or the number of responses is infinite, or both.

2.2.2 Algorithms

The last distinction between learning systems we will mention is the algorithm they use to learn with. The least precise way to make the distinction is to divide them by the line which divides all AI-research: the symbolic/sub-symbolic distinction. Winston [WIN92] gives a more precise list of different methods. They include learning by analyzing differences, learning by managing multiple models, learning by correcting mistakes, learning by recording cases, learning by building identification trees and learning by explaining experience. Most of these are symbolic methods, concerned with noise-free static environments. Two other methods Winston mentions are learning by training neural nets and learning by simulating evolution, both subsymbolic and more applicable to changing environments with noise. Classifier systems, the learning method used in this research, are not mentioned by Winston. They are also sub-symbolic and the hypothesis is, that they are also suited for noisy and changing environments.

2.3 Applications

Learning systems have been applied to many different tasks. Michalski, Carbonell and Mitchell [MCM83b] give an extensive list of areas where learning systems have been used. These include such things as game-playing [GB89], (one of the first areas of application) speech-recognition [HKP91, SERO87] and hand-writing-recognition (in commercial systems, like the Apple Newton), driving a car [POTO89, POM89], classification of objects, collecting expert-system knowledge, determining the spatial structure of proteins, controlling industrial processes [WIN92, LEE86], intelligent user-interfaces, [SUTY91] but learning computer systems are also used by psychologists to investigate human learning.

Chapter 3

The Genetic Algorithm

It is hard to believe that a seemingly random process can create very complex and efficient structures. Still Darwin [DAR1859] made it very likely that that is exactly the way nature has produced its dazzling variety of living organisms. His idea was that infinitesmal, more or less random changes in individuals, combined with an outside pressure, ('struggle for life') that gave the best adapted ones the greatest chance to produce offspring, can add up to produce the most astoundingly complex structures, like eyes, insect societies, or even brains.

To apply this simple technique to the area of computer search and discovery seems like a rather good idea. But it was not until the seventies that these ideas were applied to computers. Holland [HOL75, HOL80, GB89] was one of the pioneers of genetic algorithm research. Research into and applications of genetic algorithms have boomed ever since.

How do genetic algorithms work and what makes them different from other search methods? First of all genetic algorithms do not work with a single object in the search space. They work with an entire population of different objects. The individuals in the population are assigned a fitness-value. This value is an indication of how good a solution a certain individual is. Fitness is calculated by an objective evaluation-function. It must be stressed that there are many differences between genetic algorithms and evolution as found in nature. The way nature does genetics can be found in for example [ALB83].

This evaluation-function is the only domain-specific part of the genetic search. The genetic process itself is not guided by domain-specific knowledge. Even the individuals are encoded in neutral 'chromosomes', which are usually just bit-strings. All genetic operations are conducted on these chromosomes. Because genetic algorithms do not use domain specific information to guide the search, they are almost universally applicable, especially in domains where very little is known about what is being searched for.

With the old population and the fitnesses of its members the next generation is created. This goes analogously to natural evolution: the best individuals have

the biggest chance of producing offspring. Offspring are like their parents, but usually not entirely. The newly made individuals eventually replace the older (and less well adapted) ones.

In this way we cover a large part of a search space in a short time. It can be proven that fit individuals will grow in number and less fit ones will disappear. The search will end up in an optimum in the search-space, and because we work with a large number of individuals, this will probably be a good optimum.

3.1 Technical details

There are three basic actions in the genetic algorithm: selection, creation of a new individual from parents and the replacement of older individuals. These actions operate on encodings of the sought parameters, not on the parameters themselves. Thus the genetic algorithm does not have to know anything about the parameters themselves.

Every parameter can be encoded as a *gene* on a *chromosome*. In the technique that will be studied in this paper, genes are characters and chromosomes are strings, but they can be almost anything. The only thing the genetic algorithm must be able to do is to copy individual genes and chromosomes.

As mentioned before, genetic algorithms cause fit individuals to proliferate and less fit individuals to disappear. This is brought about by selecting some of the best individuals, crossing these with each other and then supplanting some of the least fit ones of the old generation with this new offspring.

Selection of individuals for procreation is the first step. The ones with the highest fitness will have the highest probability of being selected. Picking of individuals is usually done in one of two ways. Roulette-wheel selection or rank-based selection. In roulette-wheel selection every individual is assigned a certain probability which is proportional to its fitness. All probabilities of all individuals should add up to one. Candidates for procreation are then selected according to these probabilities. This is a bit like spinning a roulette-wheel.

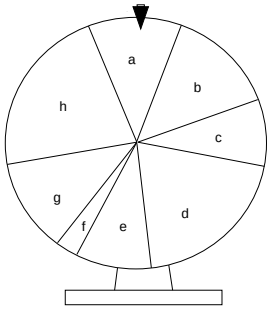

Figure 3.1: A roulette-wheel

In rank-based selection individuals are ordered according to their fitness. Individuals are then assigned fixed probabilities according to their rank, that is, their position in the list. Candidates are again selected with these probabilities.

The difference between rank-based selection and roulette-wheel selection is not very big. When using roulette-wheel selection one should be careful that the ratio between the highest and lowest probability is not too great; otherwise certain individuals will very quickly start to dominate the population. It will usually be necessary to scale the probabilities. Goldberg [GB89] suggests that the ratio should be between 1.2 and 2.

Rank-based selection has the problem that all the individuals have to be sorted first. It is much easier to control the different probabilities of the individuals in the population, however.

The next thing that happens in the genetic algorithm is the creation of new individuals from the selected parents. This is usually done with three techniques: crossover, mutation and inversion. In this research inversion will not be used, but we will explain it anyway, because it is a very basic technique of genetic algorithms.

Crossover is the mixing of two parents (usually there are two parents per child). The chromosomes of the parents have to be of the same length. A random point (between two genes) in one of the parents is chosen. The child will now consist of the genes before this point of the one parent and the genes after this point from the other parent. The purpose of crossover is to mix the genes of two parents, so good things ('partial solutions') from one parent will be mixed with good things from the other parent. This could be considered an exchange of information.

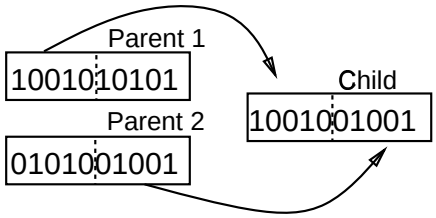

Figure 3.2: Crossover. Bits are genes

Mutation is a very simple operation. The value of a gene in the child is changed at random. This is done with a certain, rather small, probability. Its purpose is to introduce new solutions into the population, which were not present at the outset or were lost during the search.

Inversion is a more complex thing. Two points in a chromosome are chosen at random. The sequence of the genes in this part of the chromosome is then inverted. This is done to put partial solutions that reinforce each other closer together in the chromosome, so that they are less likely to be separated with

crossover.

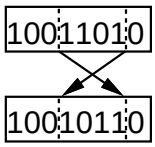

Figure 3.3: Inversion. Bits are genes

These techniques will not be used with every child. Sometimes a child is just a copy of one of its parents. Crossover, mutation and inversion are only used with a certain probability.

Supposing that we work with a fixed population-size, it is necessary for a genetic algorithm to replace old individuals in order to accommodate the new ones. This can be done in many different ways. One can choose individuals with a fitness lower than the new ones (and if these cannot be found, discard the new ones,) but it is also possible to replace the entire old population with its offspring, and there are many other possibilities [GB89, MIC92]. The method chosen depends on the application of the genetic algorithm.

3.2 Some theoretical explanation.

Why do all these very simple actions make such a powerful search-technique? To answer this question we must look a bit into the theories of genetic search. This section is partly based on the thesis by Boers and Kuiper [BK92] and [GB89].

First we define a thing which we will call a *schema*. A schema is a description of a class of chromosomes. It consists of a string with the possible values of the genes of the chromosome and a so-called wildcard. For convenience we will take bits for genes. A schema will then be of the form $\{0, 1, *\}^l$, where $*$ is the wildcard-character and l is the length of the chromosome. A schema describes a class of chromosomes with a 0 at the place where the schema has a 0, a 1 at the place where the schema has a 1 and either a 0 or a 1 at the place where a schema has a $*$. For example, the schema $1**0$ matches with 1000, 1010, 1100 and 1110. A chromosome satisfying a certain schema is called an *instance* of it.

The *defining length* δ of a schema is the number of possible crossover-points between the first and the last non-wildcard character. The defining length of the schema $**101***10*$, for example, is 6.

A highly fit schema with a short defining length is called a *building block*.

We will now give a simple proof that good schemas will grow exponentially in number when using a genetic algorithm that replaces the *entire* population. First we define the average chance, $p_{avg,\mathcal{S}}$ that an instance of schema $\mathcal{S}$ gets chosen for procreation. This can be calculated by adding up all the probabilities

of all the instances of $\mathcal{S}$ and dividing it by the number, n, of these instances. Then the total number of instances of $\mathcal{S}$ selected is given by $p_{avg,\mathcal{S}} \cdot n \cdot N$, where N is the number of individuals in the population. Note that this gives a value greater than n if $p_{avg,\mathcal{S}} > \frac{1}{N}$, which happens to be the average over the probabilities of being chosen of *all* the individuals in the population.

In order to calculate the number of instances of $\mathcal{S}$ that will appear in the new population, we need to know what the probability is that a certain schema is not destroyed by crossover. This is given by the formula:

$$p_s \geq 1 - \frac{\delta_{\mathcal{S}}}{l-1} p_c$$

where p_s is the probability of not being destroyed, $\delta_{\mathcal{S}}$ is the defining length of $\mathcal{S}$, l the length of a chromosome so $l-1$ is the number of crossoverpoints in the chromosome, and p_c the probability of crossover. The formula has a greater than or equal sign because a schema can survive a crossover, for example when both parents are instances of it. Also note that we do not look at the possibility of a mutation. One is allowed to do this, because the chance of a mutation occurring is usually so small as to be negligible.

By combining these two formulas we find the number of instances of $\mathcal{S}$ in the new population by:

$$n' = p_{avg,\mathcal{S}} \cdot n \cdot N \cdot (1 - \frac{\delta_{\mathcal{S}}}{l-1}) p_c$$

dividing this by n to find the ratio, r, gives:

$$r = p_{avg,\mathcal{S}} \cdot N \cdot (1 - \frac{\delta_{\mathcal{S}}}{l-1}) p_c$$

which is greater than one only for better than average schemas with a short defining length. Schemas satisfying these conditions, however, will grow exponentially in number. This gives a mathematical foundation for the intuitive idea that genetic algorithms can be a very efficient way of searching for a solution to a problem.

3.3 Variations

A great many variations on the basic theme of the genetic algorithm are possible. There has been extensive research on this subject and it is not at all clear what kinds of variations are possible or what their effects are.

For example, it is possible to use pairs of chromosomes with dominant and recessive genes, as they exist in nature, instead of the single chromosomes described above. This would be useful in preserving information which is not directly of use to an individual, but which might be useful at a later time.

Another possible variation is the use of diversity next to fitness as a criterion for choosing individuals for the next generation. This means that we pick individuals for reproduction not only because they are fit, but also because they are different from the individuals that are already present in the new population. This prevents loss of information through inbreeding and prevents the genetic search from ending up in a local maximum.

We can also vary the parameters of the genetic algorithm: the mutation rate, the probability of crossover, the number of individuals in the population, the ways of replacement and selection, etcetera, etcetera. A lot of research into these subjects is being conducted at the moment. This is not, however, the subject of this paper.

Chapter 4

Classifier Systems and the Bucket Brigade

The learning system that is going to be studied in this paper is called a *classifier system with bucket brigade* [HOL80, HOL86, GB89, SMI92]. These systems can be considered an intermediate between real non-symbolic systems (neural networks) and symbolic systems (especially expert systems). The classifier system was invented by Holland [HOL75, HOL80, HOL86] and was researched by many others [BGH89, BOO85, DOR91, FOR85, SMI92, WES85, ZHOU85]. The classifier systems with bucket brigade algorithms that were researched are basically the same, but have all kinds of subtle differences which are not directly apparent from the literature. The aim of this chapter and the next is to give an informal description of what a classifier system is and what it can do (this chapter) and then give a formal definition of the classifier system, in order to make clear exactly what we are talking about (chapter 5). Classifier systems are by no means simple or even standardized. As Goldberg et al. [GHD92] write: "*classifier systems are a quagmire—a glorious, wondrous, and inviting quagmire, but a quagmire nonetheless.*"

4.1 The Layers of the System

Just like every other reinforcement learning system the classifier system with bucket brigade consists of three layers (see figure 4.1). The first layer sees to it that the system is able to provide answers (be they right or wrong) to the problems it is confronted with. This is the classifier system. The second layer tries to evaluate the performance of the first layer. It can also adjust that layer's performance by using the payoff provided by the environment. This payoff is high if the behaviour of the system is good and low if the behaviour is not. It cannot, however, change the behaviour in a creative way, as it can only adjust

things that are already present in the system. The algorithm used for this aspect in this paper is called the *bucket brigade*. The task of the third layer is to try to find new ways in which the learning system can perform its function. This is done by a genetic algorithm (see chapter 3). Farmer [FAR90] calls these layers the transition rule, the learning rule respectively the graph dynamics.

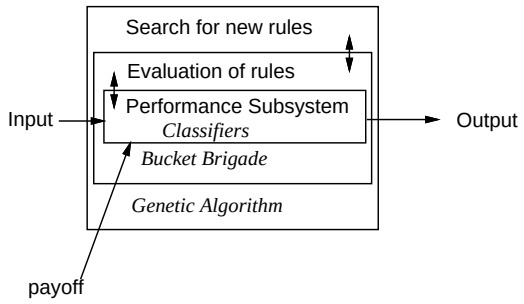

Figure 4.1: Model of a classifier system as a learning system [BGH89]

4.2 What is a classifier system?

A classifier system is a kind of production system. In general a production system is a set of rules that trigger each other and perform certain actions. The rules consist of a condition and an action. The action can make true the condition of another rule. This is the way in which rules influence each other. The action of a rule can also perform actions on the outside world. Examples of production systems in action can be found in compilers and some expert systems.

Classifier systems are parallel production systems, whereas expert systems are usually non-parallel. A parallel production system is a production system in which more than one rule can be active at the same time. There is no difference in computational power between parallel production systems and production systems in which there is only one active rule at once. Post [POST43] has shown that production systems can achieve computational completeness with only one active rule at once. Minsky [MIN67] gives a simplified proof of this.

4.2.1 The Message List

In order to make possible the parallel activation of rules the classifier system has an extra part: a message list. A rule will be activated if the messages in the list satisfy its conditions. All rules are checked this way. Activated rules may then place their actions in a new message list. Whether an active rule is actually allowed to place its message is determined by its *strength*. The

stronger a classifier is, the bigger the chance that it can place its message when it is activated.

The old message list is discarded. With the new message list, the rules can be activated again. In this way we can make an iterating system. The iterating is started by inserting a message from the outside world in the message list and it stops when one of the classifiers produces an action that affects the outside world. This is called output.

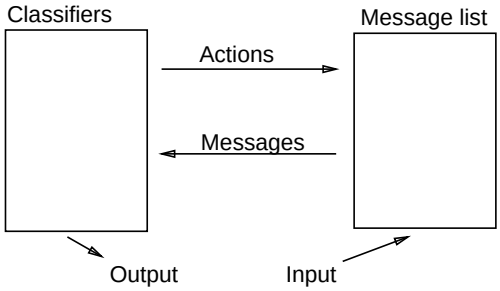

Figure 4.2: The cycle of a classifier system

The message list can also be omitted. The classifiers will then calculate an output directly from the input. Such a system is called a *single-layer* classifier system. A classifier system that does have a message list is sometimes called a *multi-layer* system.

4.2.2 The Rules

The rules of a classifier system are very simple. The conditions are strings consisting of zeroes, ones and wildcards. These will be matched against the messages in the message list, which are ordinary bitstrings (consisting of only zeroes and ones). Wildcards (written *) are characters that match both zeroes and ones. A condition can also be *unmatched true*. This means that it is activated if there are no messages in the message-list matching it. Otherwise it is called *matched true*. One rule can have more than one condition. The classifier-system discussed in this paper has a fixed number of conditions per rule.

The action of a rule consists of two parts: a part that can be placed in the message list and a part that can influence the environment. The first part consists of a string made up of zeroes, ones and passthroughs (written #). If the rule is activated and its action is to be placed in the message-list, passthroughs will be filled up by the corresponding characters from one of the messages that activated the rule. The other part is the action of the rule that can affect the outside world, also called the output. It is optional. The output can be almost anything.

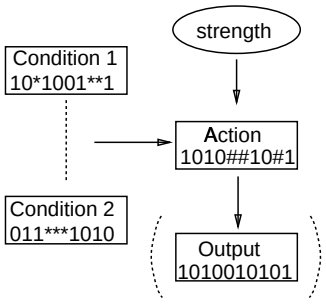

Figure 4.3: A classifier with conditions, action, strength and possibly an output

In short we can say that a classifier system takes an input, coded as a message. This message then activates rules which can then activate other rules or produce an output to the outside world. As a production system is computationally complete, a classifier system can compute every function between input and output. Note, however, that it is *not* a learning system yet. For that we need an extra algorithm, the bucket-brigade.

4.3 What is the Bucket Brigade?

The algorithm that is used to evaluate the rules of the classifier system is called the *bucket-brigade-algorithm* [HOL80]. In order to understand its funny name, we have to understand the way it works.

In the bucket-brigade every rule is rewarded if it is able to activate another rule. This reward is paid by the activated rule. Rules that are able to produce output are rewarded (or punished) by the environment. As we have seen, rules have a strength that is used to determine which rules are allowed to perform their action. This strength is used to determine a bid. The higher the strength of a classifier, the higher its bid. Also, the more specific its condition, the higher its bid. A classifier is called more specific than another if it contains fewer wildcards in its conditions. This way special classifiers are preferred over general classifiers, if they have the same strength. Specific classifiers are believed to be more applicable to a given situation.

Because of the fact that bids are dependent on the specificity of the classifiers, so called *default hierarchies* can be formed. In these a very general rule takes care of most of the cases, while more specific rules take care of the exceptions (and even more specific rules can take care of the exceptions on the exceptions.) As the specific rules are preferred over the more general ones these will handle all the cases in which they are applicable, while the more general rules will only be activated if there are no other suitable rules, and so effectively act as a default-rule.

After the bids are collected, a kind of auction is held. Only the classifiers with the highest bids are allowed to perform their actions. This is done in a non-deterministic way, to give less strong classifiers a chance as well. Now the bucket-brigade algorithm takes care that every classifier that was allowed to perform its action gets to pay its bid to the classifiers that helped in activating it. Usually the bid-value is distributed equally among the contributing classifiers. The bid of the activated classifier gets subtracted from its strength and the amounts paid to classifiers get added to their strengths. Furthermore the strength of every classifier that takes part in the auction is decreased by a certain percentage. This is called the bid-tax. There can also be an existence-tax for classifiers which means that in every cycle of the system their strength is decreased. This is done to reduce the strength of classifiers that do not have interactions with the outside world. These would otherwise survive and hinder the functioning of the genetic algorithm (see section 4.4.

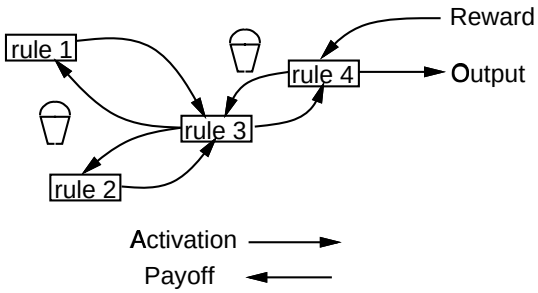

Figure 4.4: The bucket brigade in action

This process is sometimes compared to a service-economy with suppliers, consumers and middle men. It is a rather good analogy, but the analogy with a mediæval fire-brigade handing buckets to each other, from which the name is derived, is also interesting.

4.3.1 Why Should it Work?

The paying of classifiers for being activated is hoped to reinforce chains of good rules. This process is known as *rule chaining*. Chains of rules will be necessary, because if the function that has to be computed by the classifier-system is suitably complex, then it cannot be computed by single rules anymore in a small system. So if the answers given by the chain of rules are usually good, the last rule (the one producing the output) will get good rewards, so its strength will increase. The rules that help it to become activated will then get good payments, so their strength will increase as well. This way the good rewards 'flow' down the chain. Bad chains of rules will get very little reward (maybe even

a negative reward, which can be called punishment). Rules that activate the rule that produces the bad output will also get small rewards, so their strength will decrease and the strength of all the rules in the chain will, too.

If the strength of good rules increases and the strength of bad rules decreases, then the good rules will become activated more often and the performance of the system is expected to improve. The system, however, will still not be able to respond properly to inputs to which no chain of rules matches. To make this possible, new rules have to be invented. And that is what the genetic part does.

4.4 The Genetic Part

Genetic algorithms have been discussed in chapter 3. But the special case of the classifier system with bucket brigade puts some special demands on the genetic algorithm. In the classifier system with bucket-brigade one takes the rules as individuals in the population. It is also possible to take complete classifier systems as individuals, but this is not usually done with the bucket-brigade.

In the bucket-brigade the bid-value of individual rules is a very good candidate for use as a fitness, after an appropriate scaling to make the ratio between the highest and lowest fitness right. In order to get a good picture of which rules are good and which rules are bad, the classifier system will be cycled many times with the bucket-brigade learning-algorithm determining the fitness of the classifiers. When the strengths (and the bid-values) of the rules have stabilized, the genetic algorithm performs its action. Individual rules will then be selected, crossed and mutated to form a new population. This new population is then used as the classifier system.

A rather special constraint applies to genetic algorithms searching a classifier system. As there is not usually a single classifier that solves the learned problem, we need a set of cooperating rules. It is necessary then to use a genetic algorithm that allows (or indeed prefers) multiple optima. In this paper a scheme called crowding [GB89] is used. Crowding consists of finding a weak individual that is very similar to the newborn child as the candidate for replacement. Preliminary tests showed that a straightforward implementation of the genetic algorithm didn't work at all.

The purpose of the classifier system with the bucket-brigade algorithm is to have good on-line performance. The system should not forget what it has learnt between two steps of the genetic algorithm. Therefore the probability of crossover and mutation must be chosen very low or only a small part of the ruleset must be supplanted every generation. The set of rules will then remain rather constant under the action of the genetic algorithm.

With these three layers of the classifier system with bucket-brigade, it is possible to create a system that learns in an environment about which it knows nothing at all. It starts by performing random actions. The environment punishes it for bad actions and rewards it for good ones. Rules which produce bad

actions will tend to be active less and less often, and good rules will be active more often, so the overall performance of the system will improve. After a while, when the system cannot do better with the information it has, the genetic algorithm invents some new rules which, it is hoped, will be able to improve the action of the system. This is rather like learning as seen by animals.

4.5 Other Types of Classifier Systems

The research into classifier systems has brought forward many kinds of classifier systems. As Smith [SMI92] writes: "*...in many ways the LCS is more of an approach: a set of conceptual details that define a certain direction for developing methods.*" (Where LCS stands for Learning Classifier System.) A classifier system is usually seen as a set of rules in the form of bitstrings. However it is also possible to view other learning systems with multiple cooperating simple "devices" as classifier systems.

Moreover there are two main directions into classifier system research, the Michigan approach and the Pitt (=Pittsburgh) approach. The Michigan approach can be summed up by saying that it concentrates on single rules coded as chromosomes an thus on cooperation between chromosomes. The Pitt approach concentrates on optimization of complete rule-sets coded as chromosomes. It is not possible to say which one of the approaches is the best. A combination of both approaches is even conceivable, for example by coding small clusters of classifiers as chromosomes.

Chapter 5

Formal Definition of a Classifier System

In order to know exactly what we mean when we are talking about a classifier system, we have to make a formal definition of it. This not only makes clear to everybody what the properties of a classifier system in this paper are; it also facilitates its mathematical analysis and its implementation as a computer program. Unfortunately defining a classifier system in a mathematical way is something which is almost never done in the literature. So this effort is completely my own. It must be noted that in other research into classifier systems other definitions are used.

5.1 The Classifier System

A classifier system χ is a tuple of the form $\langle \mathcal{C}, B, \mathrm{F} \rangle$ in which

- $\mathcal{C}$ is a set of *classifiers*. The size of this set $|\mathcal{C}|$ is called M .
- B a set of *messages*, the *message list*.
- F a function $\mathrm{F}(\mathcal{C}, B) \rightarrow (B', \Upsilon)$, that converts a message list and a set of classifiers into another message list B' and *output* Υ. The output Υ can be empty.

5.1.1 the classifiers

A classifier is a tuple $\langle \Gamma, \mathrm{A}, \Upsilon, S \rangle$ in which

- Γ is a set of tuples, the *conditions*. They are of the form $\langle g, C \rangle$, in which $g \in \{false, true\}$, which will be called the *boolean part* of a condition,

and C a string of the form $\{0,1,*\}^l$ ($l \in \mathbf{N}^+$) which will be called the *string part* of a condition. For all $c \in \mathcal{C}$ it is true that $|\Gamma| = k$, in which $k \in \mathbf{N}^+ (k \geq 1)$, is a parameter of the system. One of the conditions is called the *preferred condition*, or γ_{pref} and is used in the substitution of passthroughs.

If the boolean part of a condition is true, then the condition will be a matched-true condition. If it is false then the condition will be unmatched-true.

- $\mathbf{A}$ is a string of the form $\{0,1,\#\}^l$ ($l \in \mathbf{N}^+$), the *action*.
- Υ is string of the form $\{0,1\}^l$ ($l \in \mathbf{N}^+$) or an empty string, ϵ, the *output*.
- $S \in \mathbf{R}$, the *strength*.

Note that in all these definitions l, called the length of the messages, must have the same value. It is a parameter of the system. The individual $0, 1, *$ and #'s will be referred to as *characters*.

5.1.2 The message list

B is a set of tuples of the form $< \theta, \phi >$, the *messages*. In these θ is of the form $\{0,1\}^l$. This is the string-part of the message. The classifier that was responsible for placing this message in the message list is stored in $\phi \in \mathcal{C} \cup \epsilon$. Note that this can also be no classifier at all in the case when the outside world is responsible for the message. This part is called the classifier-part.

The upper limit of the size $|B|$ of the message list is called N and is a parameter of the system.

5.2 The function F

In order to define the function F that converts an old message list into a new one, we first define a function to match characters: $\mathrm{f_m} : \{0,1\} \times \{0,1,*\} \rightarrow \{false, true\}$ as:

$\mathrm{f_m}(0,0) = true$
$\mathrm{f_m}(0,1) = false$
$\mathrm{f_m}(0,*) = true$
$\mathrm{f_m}(1,0) = false$
$\mathrm{f_m}(1,1) = true$
$\mathrm{f_m}(1,*) = true$

Because the $*$ character in a condition always gives true when matched against another character, it may be called a *wildcard*.

5.2.1 The matching-function

With this we define a matching-function for messages and conditions: $\mathrm{F_m}$: $\{0,1\}^l \times \{0,1,*\}^l \rightarrow \{false, true\}$ ($l \in \mathbf{N}^+$, as above) as:

$$\mathrm{F_m}(X,Y) = \wedge_{i=1}^{l} \mathrm{f_m}(x_i, y_i), \text{ in which } x_i\,(y_i) \text{ the i-th element of } X\,(Y).$$

A condition y is *satisfied* by a message x, if $\mathrm{F_m}(x, C_y) = true$, in which C_y is the string part of the condition y. A classifier c is *activated* if:

$$\forall \gamma \in \Gamma_c : \begin{cases} (g_\gamma = true \ \wedge \ \exists b \in B : & \text{(for which it is true that } C_\gamma \text{ is satisfied by } b)) \\ & \vee \\ (g_\gamma = false \ \wedge \ \forall b \in B : & (\ C_\gamma \text{ is not satisfied by } b)) \end{cases}$$

in which Γ_c is the set of conditions of classifier c, g_γ the boolean part of a condition γ, C_γ the string part of condition γ and B the message list (as above).

5.2.2 The conversion of an action into a message

In order to define a function $\mathrm{F_o}$ that converts the string part of a message, a condition with wildcards and an action with passthroughs into a new string part of a message, we use a function with the aid of a conversion-function for single characters: $\mathrm{f_o} : \{0,1,\epsilon\} \times \{0,1,*\} \times \{0,1,\#\} \rightarrow \{0,1\}$:

$\mathrm{f_o}(x,0,0) = 0$
$\mathrm{f_o}(x,0,1) = 1$
$\mathrm{f_o}(x,0,\#) = 0$
$\mathrm{f_o}(x,1,0) = 0$
$\mathrm{f_o}(x,1,1) = 1$
$\mathrm{f_o}(x,1,\#) = 1$
$\mathrm{f_o}(x,*,0) = 0$
$\mathrm{f_o}(x,*,1) = 1$
$\mathrm{f_o}(x,*,\#) = x$ if x is 0 or 1. If $x = \epsilon$ then the output of $\mathrm{f_o}(\epsilon,*,\#)$ is either 0 or 1 both with probability 0.5.

The function $\mathrm{F_o} : (\{0,1\}^l \cup \epsilon) \times \{0,1,*\}^l \times \{0,1,\#\}^l \rightarrow \{0,1\}^l$ is now defined as:

$$B_i = \mathrm{f_o}(m_i, \gamma_i, \mathrm{A}_i) \text{ for } i = 1 \ldots l$$

in which m_i, B_i, γ_i and A_i are the $\mathrm{i^{th}}$ symbols of m, B, γ and A, respectively.

5.2.3 The definition of the transition-function

We now assume two functions: one that determines if a certain classifier is allowed to place its action in the message list and another to determine if a classifier is allowed to generate output. The first function will be $f_g(c, \mathcal{C}) \rightarrow \{false, true\}$ ($c \in \mathcal{C}$), the second function $f_z(c, \mathcal{C}) \rightarrow \{false, true\}$ ($c \in \mathcal{C}$), in which $\mathcal{C}$ is the set of classifiers. Furthermore we can say that the cardinality

of the set $\{c \in \mathcal{C} \mid f_z(c, \mathcal{C}) = true\}$ is less than or equal to one. This means that at most one classifier is allowed to generate output. We will postpone the definition of these functions until later.

We are now ready to define the first part of the function F, the part that determines the new message list, which we will call F_B as follows:

$$\mathrm{F}_B(\mathcal{C}, B) = \begin{cases} \{< \mathrm{F_o}(m, \gamma_{pref}, \mathrm{A}_c), c > | \ c \in \mathcal{C} \wedge c \text{ is activated } \wedge \mathrm{f_g}(c, \mathcal{C})\} & \text{if } \mathrm{F}_\Upsilon(C, B) = \emptyset \\ \emptyset & \text{if } \mathrm{F}_\Upsilon(C, B) \neq \emptyset. \end{cases}$$

In which A_c is the action-part of classifier c and F_Υ is the transition-function F which we will define below.

The second part of F,the part that determines the output of the system, which we will call F_Υ, is now defined as:

$$\mathrm{F}_\Upsilon(\mathcal{C}, B) = \begin{cases} \{\Upsilon_c \mid c \in \mathcal{C} \wedge \mathrm{f_z}(c, \mathcal{C})\} & \text{if } \exists c \in \mathcal{C} : (c \text{ is the best and } \Upsilon_c \neq \emptyset) \\ \emptyset & \text{if } \nexists c \in \mathcal{C} : (c \text{ is the best and } \Upsilon_c \neq \emptyset). \end{cases}$$

Where with *the best* we mean that a classifier is activated and has the highest bid. Of course other schemes of determining whether a classifier system is allowed to produce output are possible, but this was found to be one of the best in the experiments (see chapter 9).

The total function F can now be calculated by combining F_B and F_Υ. The output of the first is used as the output B' of F and the latter is used as the output Υ of F.

It appears as if the message list (the argument B) is not used by function F, but it *is* used to determine if a classifier is activated.

5.2.4 Calculating the bid-value

From the strength of a classifier we can calculate the *bid-value*. This is a value which is going to be used to determine which messages are going to be in the new message list. The bid-value is calculated from the strength of a classifier and its specificity as follows:

$$B_c = \beta \cdot f_{spec}(c) \cdot S_c$$

in which B_c is the bid of classifier c, β a constant $0 < \beta \leq 1$ that determines which fraction of the strength is used in the bidding, $f_{spec}(c)$ a function of the specificity of the classifier c and S_c the strength of classifier c.

The function $f_{spec}(c)$ can be defined in many different ways, but it should give a higher value for classifiers that match with fewer messages (i.e. are more specific), because if a classifier matches with fewer messages than another classifier, it can be considered more relevant for the messages with which it *does* match.

A possibility for $f_{spec}(c)$ is:

$$f_{spec}(c) = \frac{L - W}{L} + \alpha$$

in which α is a constant that determines the importance which is assigned to the specificity of a classifier. In this equation W is the number of $*$-characters in the conditions of a classifier. L is equal to the total number of characters in the conditions of a classifier, and can be calculated by multiplying l (the number of characters in a single condition) by k (the number of conditions in a classifier). Note that this function does not necessarily give a value between zero and one.

5.2.5 The functions f_g and f_z

The functions f_g and f_z that pick the classifiers that place an action, respectively an output, can be defined in many ways, but they both have to fulfill certain conditions, one of which (about the cardinality of f_z) has been mentioned in section 5.2.3. For a classifier system to function well, they should not be completely deterministic (otherwise the system will tend to favour the status quo too much) and they should give classifiers with greater strength a higher possibility to get a value of $true$. Furthermore these functions should base their behaviour on all the classifiers that are activated.

In this thesis we will use a definition of f_c and f_g that is also used by Goldberg [GB89], which consists of taking the bid-values of all active classifiers, adding a value taken from the normal distribution $\mathrm{N}(0, \sigma)$ and then assigning the value of true to at most M highest active ones ($M \in \mathbf{N}^+$) in the case of f_g and to the highest active one in the case of f_z. This seems to be the most effective choice for the functions and it is the one that is used in the practical experiments described in this paper. In chapter 8 some other possibilities (the roulette-wheel scheme) are tried and shown inferior.

5.2.6 Iterating

With these definitions we can make an iterating system. On each message-list and set of classifiers we can apply the function $\mathrm{F}(\mathcal{C}, B)$ to get a new message list. On *this* message-list we can apply the function again. We get a system of the form: $B_{t+1} = \mathrm{F}_B(\mathcal{C}, B_t)$ (F_B is the message-part of function F). We now have to define B_0 and a criterium on which to stop the iteration-process.

The definition of B_0 is easy: we generate a message ι , which we will call the *input* to the system. Now we take $B_0 = \{\iota\}$ (i.e. a set in which the only member is ι). The stopping criterium is as follows:

$$\mathrm{F}_\Upsilon(\mathcal{C}, B_t) \neq \emptyset \vee t > T$$

in which F_Υ is the output-part of function F and T is a parameter that gives the maximum number of iterations of the system.

5.3 The bucket-brigade algorithm

We have now defined a system with which we can compute a set of functions. We can now try to define an algorithm with which we can make this system learn certain functions. There are several possible approaches [GB89] of which the so called *bucket-brigade* [HOL80] is one. The formalization of the learning system will be based on this algorithm.

This learning algorithm is based on modifying the strengths of the classifiers. This modification will happen between two steps of the classifier system. All modifications of the strength will have to be done before a new input is fed to the system.

5.3.1 Paid and paying classifiers

To determine which classifiers will be "paid" and which have to "pay" we define a few sets. First the set P_t of classifiers that have placed a message on the message list or generated an output at timestep t:

$$P_t = \{c \mid c \in \mathcal{C} \wedge c \text{ is activated at timestep } t \wedge (\mathrm{f}_g(c, \mathcal{C}) \vee \mathrm{f}_z(c, \mathcal{C}))\}$$

Then $Q_{c,t}$, the set of classifiers that have placed messages in the message list that have made classifier c true at timestep t.

$$Q_{c,t} = \{x \mid x \in P_{t-1} \wedge \exists b \in B : (b_\phi = x \wedge \exists \gamma \in \Gamma_c : (\mathrm{F_m}(b_\phi, c_\gamma) = true \text{ and } x \text{ is the preferred classifier}))\}$$

In which x, or the *preferred classifier* of a condition is the one classifier chosen randomly from the classifiers that had messages placed in the message list that matched with this condition. In other words: only one classifier is rewarded per condition, even if there might have been more that placed a suitable message. Γ_c is the set of conditions of classifier c, A_x is the action part of classifier x and c_γ is the string-part of condition γ.

5.3.2 The amounts paid

With these sets we can determine the changes in strength of the classifiers, in other words: the amounts that have to be paid by and to the classifiers.

$$\forall c \in \mathcal{C}(c \notin P_{t-1}) : (S'_{c,t} = S_{c,t-1} - \tau_{life} \cdot S_{c,t-1})$$

Which means that all classifiers that were not active on time $t-1$ are paying a certain tax between steps of the system. The constant $\tau_{life}, 0 \leq \tau_{life} < 1$ is a parameter of the system that determines how much tax is paid. And for the classifiers that were active and took part in the bidding there is an additional tax, called the bidtax :

$$\forall c \in \mathcal{C}(c \in P_{t-1}) : (S'_{c,t} = S_{c,t-1} - (\tau_{bid} + \tau_{life})S_{c,t-1})$$

In which $0 \leq \tau_{bid} < 1$.

Then two other modifications take place:

$$\forall c \in P_t : (S''_{c,t} = S'_{c,t} - B_c)$$

In which B_c is the bid made by classifier c in order to get activated. This makes classifiers that have their action placed pay for it. Obviously we have to make sure that $(\tau_{life} + \tau_{bid})S_{c,t-1} + B_c < S_{c,t-1}$, so S_c will always be positive.

$$\forall c \in P_t : (\forall x \in Q_{c,t} : (S_{x,t} = S'_{x,t} + \frac{1}{\mid Q_{c,t} \mid} \cdot B_c))$$

This rewards classifiers that helped making other classifiers active, because we reward all classifiers that helped to make c active (from the set $Q_{c,t}$ with an equal share of the bid-value of c.

For all other classifiers we take the strength at time t ($S_{c,t}$) equal to either $S''_{c,t}$ (if present) or $S'_{c,t}$.

5.3.3 Exceptions: input and output

This definition of the bucket-brigade algorithm obviously works for classifier system steps that have nothing to do with input and output. But what happens when there is input or output?

At the input-stage, we put only one message in the message-list, $B_0 = \iota$, and the sets P_{-1}, which is the set of active classifiers before the system starts and $Q_{c,0}$ (for all c) are empty, so there are no classifiers to reward. The strength of the classifiers that have become active are still decreased, though. The net result is that the total strength of the system decreases. It is possible to see this as a payment to the outside world.

For output the situation is equally simple. The one classifier that was selected to generate the output is rewarded by the outside world with an amount ρ, which is determined by how good the response of the system was in the given situation. This amount is added to its strength. $S'_c = S_c + \rho$, in which S_c is the strength of the classifier that generated the output. Negative rewards are not possible in this system. This prevents the strength of classifiers from becoming negative.

Obviously many variations are possible in the details of a classifier system, some of which will cause a notable difference in performance and some of which hardly have any influence at all. Especially in the bucket-brigade part a lot of subtle interdependencies of the strengths of the classifiers exist. This definition was therefore not made from an entirely theoretical basis, but was finetuned by using the results of the experiments which formed the basis of chapter eight.

Chapter 6

Analysis of the Behaviour of the Classifier System

In order to get a good understanding of the operation of a classifier system it is necessary to analyze it mathematically. This analysis can be conducted from different points of view. We can try to figure out what the best values of the parameters of the system are. We can also try to find out what the behaviour of the system in the long run is and together with that, how long it will take for the system to run.

Another interesting subject of research is the computational strength of the different possible variations of the classifier system. The mathematical approach necessary for that is quite different from the one that will be used in this chapter, so we will postpone it until chapter 7.

6.1 Steady-State Behaviour

The classifier system with the bucket-brigade algorithm is a very complex system in which many interactions take place. These interactions consist of the activations of classifiers, the paying of bids and the updating of their strengths. This process is too complicated to analyze completely with mathematical tools. It is possible, however to say something about the long-term behaviour of the system and especially about its *steady-state behaviour*. The steady state is an abstract state of the classifier system in which all the classifiers have a fixed strength and a fixed bid. This is obviously something which will not happen quickly in any practical classifier system as there will always be conflicting rules and random variations in the input that cause perturbations. We should be very careful with these results then. As Forrest and Miller [FOMI90] write: "*Any learning system that must interact with a dynamic environment is highly unlikely to reach a meaningful asymptote in any space of reasonable dimensions.*

Further, even if it did we would be at least as interested in understanding how it approached asymptotic behavior as in understanding what particular asymptote it reached."

It is a very useful tool, however to estimate certain things about expected behaviour of classifiers and it is also useful to determine the values of the parameters of the system.

We therefore define the steady-state strength S_{ss} as the strength and the steady-state bid B_{ss} as the bid that a classifier will have in the long run assuming that a) there are no conflicting classifiers and b) a classifier always gets the same reward R_{ss} for every action it generates. These assumptions are not unreasonable if we consider a classifier system after it has been trained fully. They are, by the way, the same assumptions as we will make about a classifier system in chapter 7 when we deal with the computational strength.

It must also be stressed that in the formulas we will derive in this section and in the following section we will be dealing with a classifier system that contains only classifiers that have one condition that is of the matched-true type ($k = 1$ and $g = true$ for every condition and every classifier). The formulas presented are based on the work of Goldberg [GB89].

6.1.1 Classifier System without Life-Tax

The first formula that we will derive is exactly the same as the one Goldberg [GB89] presents. Its limitation is that it is only valid for a classifier system that has no life-tax. This means that the strength of non-bidding classifiers is never reduced. A possibility that individuals that do nothing will dominate the population in genetic search arises in this case. Therefore life-tax is quite important (although it would be possible to choose a low starting-strength for new individuals as well).

But although life-tax has great influence, the derivivation of the formula for the steady-state behaviour without life-tax proves very enlightening. The steady-state behaviour of a system *with* life-tax will be derived in the next paragraph.

The steady-state strength S_{ss} is given by:

$$S_{ss} = S_{ss} - B_{ss} - \tau_{bid} \cdot S_{ss} + R_{ss} \tag{6.1}$$

where B_{ss} and R_{ss} as above. The constant τ_{bid} is the tax-constant for the bid-tax, the tax that must be paid by classifiers taking part in the bidding. The steady-state bid B_{ss} is given by:

$$B_{ss} = f_{spec}(c) \cdot \beta \cdot S_{ss} \tag{6.2}$$

where $f_{spec}(c)$ gives the specificity-value for the classifier under consideration. An example of this function can be found in chapter 5. Combining equations

wildcards	$\frac{S_{ss}}{R_{ss}}$	$\frac{B_{ss}}{R_{ss}}$
0	7.4074	0.925926
1	8.4507	0.915493
2	9.8361	0.901639
3	11.7647	0.882353
4	14.6341	0.853659
5	19.3548	0.806452
6	28.5714	0.714286

Table 6.1: Example values for bids and strengths in the steady state

6.1 and 6.2 and solving for S_{ss} gives:

$$S_{ss} = \frac{R_{ss}}{f_{spec}(c) \cdot \beta + \tau_{bid}} \tag{6.3}$$

And for B_{ss} we combine equations 6.3 and 6.2 and get:

$$B_{ss} = \frac{f_{spec}(c) \cdot \beta \cdot R_{ss}}{f_{spec}(c) \cdot \beta + \tau_{bid}} \tag{6.4}$$

With these functions we can calculate the strengths and bids of classifiers in their steady-state. An example is shown in table 6.1 where the parameters and the specificity-formula of the simple classifier system from chapter 8 were used. This table shows the values for classifiers getting maximum reward only. The values are divided by the reward, so as to make the table independent of the reward. It is easy to see that for bad classifiers that do not get any reward at all, the strength and the bid will go to zero.

6.1.2 Classifier System with Life-Tax

In the case where we have a non-zero existence- or life-tax, the formulas describing the steady state of the system will be different. This is because of the fact that when the classifier is not being activated, its strength decreases anyway. Whereas in the steady-state without life-tax we could ignore the cycles in which the classifier does not become active —nothing happens after all— in the case where we have life-tax, we have to take into account the number of cycles a classifier is inactive between activations.

We will estimate the number of inactive cycles between two activations. First we assume that the input messages to a classifier are bit-strings that all have an equal probability of occurring. In practical situations this is somewhat unrealistic, because inputs to a classifier system represent a certain domain which hopefully is not random. Also for internal classifiers, the classifiers that get their inputs from the actions of other classifiers, the inputs can hardly be

considered random. For the sake of simplicity, however we assume that the probabilities are equal for all messages.

Now we can say something about the relation between the number of wildcards in a condition and the probability that it will be matched by a message. The probability is given as:

$$P = 2^{w-l} \tag{6.5}$$

where P is the probability that a match occurs, l is the length of a condition and w is the number of wildcards in the condition. The expected value of the number of steps between two activations is then given by:

$$E = P \cdot \sum_{i=1}^{\infty} i \cdot (1-P)^i$$

which can be simplified to:

$$E = \frac{1}{P} \tag{6.6}$$

We can now write down an equation for S_{ss} in the case of non-zero life-tax:

$$S_{ss} = (1-\tau_{life})^E (S_{ss} - B_{ss} - \tau_{bid} \cdot S_{ss}) + R_{ss} \tag{6.7}$$

By using equations 6.2, 6.5, 6.6 and 6.7 and solving for S_{ss} we get:

$$S_{ss} = \frac{R_{ss}}{1 - \left((1-\tau_{life})^E (1 - f_{spec}(c) \cdot \beta - \tau_{bid})\right)} \tag{6.8}$$

In the same way we can derive the formula for B_{ss} by applying equation 6.2. See also [MON93] for a similar formula.

It is clear that this equation is equal to equation 6.3 in the case that τ_{life} is zero. This is obviously necessary. To gain some understanding of the influence of the specificity-function and the life-tax constant on the bids in the steady state, graph 6.1 shows the steady state bids for various values of τ_{life} and various specificity-functions. These specificity-functions are all based on the formula $f_{spec}(C) = \frac{l-w}{l} + \alpha$ for different values of alpha. The bids were calculated for classifiers with length 57, the same length as the ones used in chapter 9. The constant β had the value of 0.1, the constant τ_{bid} had the value of 0.01.

From this graph we can see that we must be very careful in choosing the values of alpha (when using this particular f_{spec}) and especially the value of the lifetax-constant. This is because if this constant get too high, the steady-state bids of the more specific classifiers will be *lower* than the less specific ones, which is exactly the opposite of the desired situation.

6.1.3 Discussion

Whereas life-tax is important in decreasing the strengths of inactive classifiers, it can also cause the disruption of default-hierarchies. It must also be stated

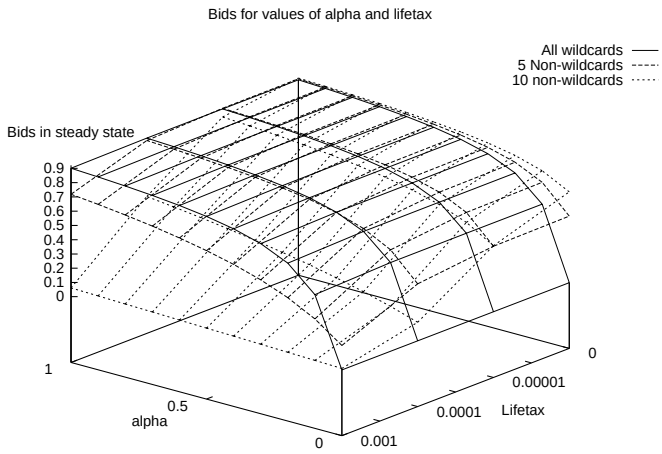

Figure 6.1: Values of the steady-state bids for different specificity-functions and values of life-tax

that the steady-state strength of a classifier with life-tax is not so steady as the name might suggest. The classifier has a (much) higher strength just after being activated and receiving payoff than after a lot of cycles of inactivity.

There are many other interesting questions that can be asked about the long-term behaviour of a classifier system, many of which also give possibilities for practical experiments. How long does it take for a classifier to reach its steady state? How does the real steady-state-behaviour of a classifier differ from the predicted one? What are the differences between internal classifiers (the ones that generate messages that activate other classifiers) and external classifiers (the ones generating output) ?

The predicted behaviour of the classifiers with life-tax also casts some doubt at the likelihood of classifiers with few wildcards occurring. The bids of classifiers with very few wildcards tends to decrease dramatically due to the long intervals between activations (and possible rewards). This causes them to have almost no chance at all in the bidding, so they will never become strong. How should we then detect classifiers that are inactive but of practical use and classifiers that just do nothing? These problems are interesting topics for further research.

6.2 Starting-Parameters

As we have seen in the previous section the parameters of the system can have an enormous influence on its performance. In the formula that was presented in that section we have an instrument for estimating the starting-values of the life-tax constant and the specificity-function.

With this formula we can also get an indication of the value for the variance of the noise added to the bids of the classifier system. This noise should not be so big as to obscure the differences between the bids, but it should also be not so low as to have no influence at all (the graph 8.5 shows evidence that a totally deterministic system does not learn optimally). The value of $\sigma = 0.05$ used throughout this paper was determined by experiments, not by using theoretical arguments. What the ratio of the differences between the steady-state bids and the ideal value of σ is remains a matter of further research.

Another interesting parameter is the number of wildcards in the classifier-system with which the learning-process is started. If there are too few classifiers, the system will not be able to respond to the problems it is confronted with. If there are too many wildcards the system will just produce nonsense and will hardly learn either. We therefore derive a formula to estimate the ideal starting-probability of wildcards in a given classifier-system.

It is quite easy to give an approximation of the number of wildcards necessary to have a fixed probability ϵ that there will be a matching classifier for any message that is input to the classifier system.

We start by determining the probability that an individual condition of length l with w wildcards is matched by a message. This probability is given by equation 6.5. If there are M messages, the probability that a condition is matched by at least one of these messages is:

$$1 - \left(1 - 2^{w-l}\right)^M$$

The probability that a classifier with k matched-true conditions becomes activated is now:

$$\left(1 - \left(1 - 2^{w-l}\right)^M\right)^k$$

So if there are N classifiers the probability that one of them is matched is:

$$1 - \left(1 - \left(1 - \left(1 - 2^{w-l}\right)^M\right)^k\right)^N \tag{6.9}$$

If we express the number of wildcards in a condition in terms of the probability p of a wildcard occurring then we can say:

$$l - w = (1 - p)l \tag{6.10}$$

Provided the desired ϵ is in the range:

$$1 - \left(1 - \left(1 - \left(1 - 2^{-l}\right)^M\right)^k\right)^N \leq \epsilon \leq 1$$

solving p from equation 6.9 and 6.10leads to:

$$p = \frac{\log_2\left(1 - \sqrt[M]{1 - \sqrt[k]{1 - \sqrt[N]{1-\epsilon}}}\right)}{l} + 1 \qquad (6.11)$$

With this equation we can estimate the probability we should choose for wildcards to occur when initializing a classifier system.

As an example we can calculate the values for the necessary probability of wildcards occurring for the mushroom-example we tested in chapter 9. We will not calculate the probability for the simple example in chapter 8 as the number of classifiers (50) in that system is rather large compared to the possible number of messages ($2^6 = 64$) and conditions ($3^6 = 729$). Equation 6.11 is not really applicable in such cases, as there are too many equal messages and conditions.

In the mushroom-example the number of possible messages is extremely large compared to the number of classifiers in the system, so we are allowed to apply our formula. We will first choose a probability ϵ. We have 57 bits in a message, we have only one message (as the classifier system is single-layer), only one condition per classifier and 50 classifiers. The value of ϵ must now lie between:

$$1 - \left(1 - 2^{-57}\right)^{50} \leq \epsilon \leq 1$$

The first term of this equation is almost zero (it can be shown that it is smaller than $1 - e^{2^{-51}}$) so we can choose almost every value between zero and one. We will choose the value of 95% for ϵ. The desired probability for wildcards will then be:

$$p = \frac{\log_2(1 - \sqrt[50]{1 - 0.95})}{l} + 1 = 0.93$$

Which seems quite near the probability that was chosen for the experiments, which was 90%, but it corresponds to an ϵ of 0.62, which points to ϵ being quite sensitive to small fluctuations in p.

Another interesting starting-parameter is the number of classifiers in a classifier system, which should be polynomially proportional to the number of bits in a classifier. If this were not the case classifier systems would not be practical. A method for dividing classifier systems in classes that have a polynomially comparable amount of classifiers for certain problems is given in chapter 7. What the ideal number of classifiers is remains an open question. What the ideal length for the message-list is, is also unknown. This can be especially task-dependent.

Another open problem is the length of the messages. The lengths do not have to be equal to the number of bits required for coding a problem. It is sometimes useful to use more in a multi-layer classifier system. This technique is called *tagging* [BGH89] and is used to divide the messages into types. The data derived from formula 6.8 gives an indication that there could be a maximum length for a classifier's conditions. When the conditions get very long and the number of non-wildcards is not too small, the difference in steady-state bids becomes

negligible or even negative. Also the difference between the steady-state bids of classifiers with zero and one non-wildcard is much larger than the difference between classifiers with, say, ten and eleven non-wildcards. This can cause a stable default-hierarchy to be practically impossible and classifiers to become overly general. This is quite a big drawback of classifier systems working in this particular way and maybe a solution should be found for this problem by implementing the calculation of bids (or even the paying of credit as Dorigo [DOR91] did) in a different way.

A starting-parameter that may not be considered a parameter is the type of classifier system used for a given problem. Some classifier systems are more applicable to a given problem than others (see chapter 7). In section 9.2 a problem is implemented which is not really learnable by the simplest kind of classifier system and it is investigated which additions to the basic system will improve the ability to learn.

6.3 Time-Complexity

The time it takes for a classifier system to learn a given problem is another unknown factor. In the experiments of chapter 8 we will see that approximately 2000 cycles are needed to reach a state in which no major changes in performance take place in the classifier system without genetic algorithm. Unfortunately a fundamental basis for this number is altogether missing.

It is quite easy however, to say something about the time-complexity of a single cycle of a classifier system in a computer program. In a straightforward implementation every message in the message-list is compared to every condition of every classifier. After that the activating classifiers are paid and the messages of the activated classifiers get placed in the new message-list. In the following M will be the number of messages, N will be the number of classifiers, k will be the number of conditions per classifier and l is the length of a message.

The first step costs $\Theta(M \cdot N \cdot k)$ comparisons. One comparison costs $\mathrm{O}(l)$ time (we use the O-notation because a comparison of two l-character strings usually costs less than l comparisons as we usually find that they are not equal before we have seen all characters). The total time is then $\mathrm{O}(M \cdot N \cdot k \cdot l)$ for the comparisons. The time taken for the paying-step is $O(N \cdot k)$ and the time for the placing of messages is $O(N \cdot l)$. The dominant term is obviously the first as both other terms will be smaller (especially when M is a function of N, as it usually is). If we estimate that $M = \mathrm{O}(N)$, the total time-complexity becomes $\mathrm{O}(N^2 \cdot k \cdot l)$. It is obvious that larger classifier-systems will run more slowly than small ones, a fact which is neatly confirmed by the experiments. In chapter 10 we will investigate a way of reducing the amount of work needed for a classifier system cycle.

We can try to make an estimation of the time it takes for a classifier system to become stable or in other words: to reach strengths which are very near the

steady-state strengths. To keep things simple we will do this only for a classifier system without life-tax. The steady-state is more clearly defined in this case and the formulas are a bit simpler.

In order to estimate the time we need an expression for the strength of a classifier at a certain moment of activation a. Note that we do not use t here, because t counts all the cycles of the classifier system, whereas a counts only the number of times a classifier is activated. This expression is as follows:

$$S_a = S_{a-1}(1 - (\tau_{bid} + f_{spec}(C)\beta)) + R_a$$

Which is essentially the same as the expression for the steady-state strength, equation 6.1. It is a recurrence that can be solved in the usual way. For convenience we will write η for $1 - (\tau_{bid} + f_{spec}(C)\beta)$ which gives:

$$S_a = \eta^a \cdot S_0 + \frac{1 - \eta^a}{1 - \eta} \cdot R \tag{6.12}$$

Where we assume that the reward R_a after every activation has the same value, R. We now want to know the number of times a that a classifier must be activated so that its strength differs only ϵ from the steady-state strength. This can be written as:

$$\epsilon = |S_a - S_{ss}| = \begin{cases} \eta^a \left(S_0 - \frac{1}{1-\eta} R \right), & \text{if } S_0 > \frac{1}{1-\eta} R; \\ \eta^a \left(\frac{1}{1-\eta} R - S_0 \right), & \text{if } S_0 \leq \frac{1}{1-\eta} R. \end{cases} \tag{6.13}$$

which can be solved for a straightforwardly to:

$$a = \log_\eta \left(\frac{\epsilon}{\left| (S_0 - \frac{1}{1-\eta} R \right|} \right) \tag{6.14}$$

This gives us an equation to determine the number of times a classifier should be activated to reach a strength near its steady-state strength. Note that it does not give the same value for every classifier. The value of η depends on the number of wildcards in the classifier. We will give in table 6.2 the values of a for different numbers of wildcards where the other parameters are the same as in chapter 8. Together with a we will give the value of $a \cdot 2^{l-w}$ (l is the length of and w the number of wildcards in a classifier) which is the expected value of the number of real classifier system steps for a classifier to reach its steady-state strength with $\epsilon = 0.01$ (which is well within the variance of the bids, which was $\sigma = 0.05$).

If we compare these results to the value used in chapter 8 we see that they are almost equal (2000 cycles versus the maximum number in the table, 2452). It is not at all clear, however how far the formula 6.14 is applicable.

w	a	$2^{l-w} \cdot a$
0	38.323	2452.67
1	40.042	1281.35
2	26.087	417.39
3	58.236	465.89
4	86.728	346.91
5	128.957	257.91
6	211.266	211.27

Table 6.2: Estimated number of cycles of classifier system to steady state

Chapter 7

Computational Strength of Classifier Systems

Although it can be shown in principle that a production system is equivalent to a Turing machine [MIN67, POST43] and although a classifier system is a kind of production system, it is not at all clear from the proofs of Post and Minsky what the computational strength of a classifier system is in practice. The proof of Minsky and Post is based on conditions and actions of variable length and on special pattern matching abilities of the conditions that an ordinary classifier system does not have. So we must try to find out what kind of functions we can compute with a finite classifier system.

7.1 Practical Computational Strength

It is rather obvious that a classifier system can compute any function from its inputs to its outputs. Suppose we have the simplest possible classifier system, consisting only of condition-action pairs of the form: $(\{0,1\}^l, \{0,1\}^k)$. Where the first part is the condition, which only matches with a message equal to it and the second part is the action. Any function $f : \{0,1\}^l \rightarrow \{0,1\}^k$ can now be implemented as a classifier system in the following way: for every string S from the domain of f we make a classifier consisting of condition matching with S (in effect equal to S) and taking the string $f(S)$ as the action. Of course, for any practical function the resulting classifier system will be extremely large.

Therefore we will try to define something which we will call the *practical computational strength* of a *class* of classifier systems. A class of classifier systems is intuitively the classifier systems that are possible according to some definition of classifier systems, or: the classifier systems with certain features. We have for example, the class of classifier systems with single condition and action, no wildcards and no message-list (as in the above proof of computational

completeness) or the class of classifier systems with conditions with wildcards, multiple conditions and message-list. All these classes can be defined formally, but we will use a more informal definition to keep the discussion compact. We will make an appeal to the reader's intuition to understand the semantics of the classes of classifier systems we will discuss. For a more thorough definition of the syntaxis and semantics of a classifier system, see chapter 5.

The practical computational strength of a class of classifier systems is intuitively the ability to implement a class of problems with a number of classifiers that scales linearly with the number of symbols in the system itself.

Of course this is not a very useful definition. What is a class of problems and how do we scale them? These are questions that are very hard to answer so we will not bother to find a solution. We will only use the term *practical computational strength* in a relative sense. We will say that a certain class of classifier systems Ξ has the same or higher practical computational strength as a class of classifier systems Ξ' if we can find a function f that converts a classifier system from Ξ' to one from Ξ that computes the same function. Together with that function there must be a constant $\zeta \in \mathbf{R}$ so that for all $\chi \in \Xi'$ $|f(\chi)| < \zeta|\chi|$. The $|\chi|$ means the number of classifiers in classifier system χ. We will write this as: $\Xi \geq \Xi'$.

Two classes Ξ and Ξ' have the same practical computational strength if $\Xi \geq \Xi'$ and $\Xi' \geq \Xi$. We write this as $\Xi \equiv \Xi'$. If for two classes Ξ and Ξ' we have $\Xi \geq \Xi'$ but not $\Xi' \geq \Xi$ we can write $\Xi > \Xi'$, or Ξ has higher practical computational strength than Ξ'. For convenience we can use the notation $\Xi' < \Xi$ if $\Xi > \Xi'$ and $\Xi' \leq \Xi$ if $\Xi \geq \Xi'$.

7.2 Example: Wildcards versus no Wildcards

In order to illustrate the notation defined above, we will consider two classes of classifier systems and prove something about their practical computational strength. The first class, ${}_X i_N$ will be the class of classifier systems without wildcards, message-list or multiple conditions, writable as a set of tuples of the form: $\{\{0,1\}^l, \{0,1\}^k\}$. The first part is the condition, matching with messages that are exactly the same and the second part is the action, or the function value associated with the input-message.

The second class, Ξ_W, will be the class of classifier systems with wildcards, but still without multiple conditions or a message-list, writable as a set of tuples of the form: $\{\{0,1,*\}^l, \{0,1\}^k, b\}$, where $b \in \mathbf{R}^+$. The real number b is used to resolve conflicts between classifiers. If two conditions match with a given input, the action of the classifier with the highest b is chosen. It is actually the bid of a classifier. This system is the same as the one implemented by the simple classifier system in chapter 8.

To prove that the practical computational strength of Ξ_W is greater than or equal to that of Ξ we need a function $f : N \rightarrow W$ that maps every classifier

system $n \in N$ to the identical classifier system in $w \in W$, by making the conditions and actions of the classifiers in w the same as the ones of the classifiers in n and setting the values of b in the classifiers from w to 1. The value of b does not matter as there are no wildcards in the conditions of the classifiers.

The number of classifiers in w is now obviously equal to the number of classifiers in n, so the condition $|f(n)| < \zeta|n|$ is satisfied for every $\zeta > 1$. We can now conclude that $W \geq N$.

We will now check that it is not true that $N \geq W$. For this it is sufficient to look at classifier systems from Ξ_W that contain only one classifier. There is no function f that maps these to equivalent classifier systems from Ξ_N so that there is a constant ζ that satisfies the condition $|f(w)| < \zeta|w|$. Suppose we think that we have found such a ζ. It is then possible to construct a counterexample by making a classifier system w from Ξ_W with only one classifier that has a condition containing more than $\lceil \log_2 \zeta \rceil + 1$ wildcards. The corresponding classifier system n from Ξ_N must then contain $2^{\lceil \log_2 \zeta \rceil + 1} > \zeta$ classifiers as it must have as many classifiers as there are messages matching the condition of the classifier from w. For this particular system ζ does not satisfy the condition. Thus it is not true that $N \geq W$. We can now conclude that $W > N$ or that classifier systems that allow wildcards have a greater practical computational strength than classifier systems without them. Their real computational strength is of course equal.

7.3 The Use of a Message List

A rather less trivial question is whether a message-list enhances the practical computational strength of a classifier system. It appears, as will be proven in the following section, that a classifier system without a message list that uses wildcards, but only has single matched-true conditions and that does not use passthroughs, has the same practical computational strength as such a classifier system with a message list.

7.3.1 Informal Definition

We will first briefly restate the definition of the classifier systems used. The one without the message-list is identical to the one with wildcards used in paragraph 7.2. We will call this system Ξ_S (Single layer). For convenience we will define this as a triple: $\langle Q, O, P \rangle$[†] , where Q, the condition part, is of the form: $\{0, 1, *\}^l$, O, the output part, is of the form: $\{0, 1\}^k$ or has the value *undefined*, which we will need later in the proof and $P \in \mathbf{R}^+$ is the bid.

For the other classifier system, which we will call Ξ_M (Multi layer) we must extend this definition a little bit. Classifiers in this system are of the form:

[†] $Q \equiv \Gamma, O \equiv \Upsilon, P \equiv f_{spec}(C)\beta S$ in the formal definition of classifier systems in chapter 5

$\langle C, A, B, T \rangle$ [‡] , where C is the condition, of the form $\{0, 1, *\}^l$, $B \in \mathbf{R}^+$ the strength, T is the type and A either the action or the output. If A is an action then it is of the form $\{0, 1\}^k$ and $k = l$ (l is the length of the condition) and $T = internal$. If A is an output, then it is of the form $\{0, 1\}^n$ and $T = output$.

When this classifier system is active, in the first phase classifiers are matched against the input. If the activated classifier with the highest bid has type $T = output$ then this one is chosen to generate the output of the system and the computation stops. If the classifier with the highest bid does not have $T = output$ then the actions produced by the active classifiers are used as the inputs for the next, similar cycle of the classifier system.

There is a possibility that the classifier system will not produce an output. This can happen because there are no activated classifiers or because the system ends in a cycle. We will not bother about these situations, because we will allow for inputs that have no outputs defined for them. It can be shown that it is possible to find out whether a given classifier system of this form has cycles or not.

7.3.2 $M \geq S$

In order to prove that Ξ_M has greater or equal practical computational strength than Ξ_S, we must find a function from classifier systems Ξ_S to classifier systems in Ξ_M that has the property: $\forall s \in S : |f(s)| < \zeta|s|$, where $\zeta \in \mathbf{R}^+$.

For this we take the function that translates a classifier from Ξ_S into a classifier from Ξ_M in the following way:

$C = Q$

$A = O$

$B = P$

$T = output$

It is obvious that this function generates a classifier system that is equivalent as all the classifiers produce output directly, and are for the rest of the same form. The number of classifiers in both systems is equal, because there is a direct one-to-one mapping. So for every $\zeta > 1$ the condition is satisfied. The practical computational strength of Ξ_M is higher than or equal to the practical computational strength of Ξ_S, which was rather obvious from the start as intuitively $S \subset M$.

7.3.3 $S \geq M$

The proof that $S \geq M$ is considerably less trivial. For this we must define a function that maps every classifier system $m \in M$ to a classifier system $s \in S$. In order to be able to do this, we first must define a graph that describes the behaviour of a multi-layer classifier system. This graph will be a directed graph

[‡] $C \equiv \Gamma, A \equiv A \equiv \Upsilon, B \equiv f_{spec}\beta S$ in the formal definition of classifier systems in chapter 5

(V, E) with the set of vertices V the input-messages, the classifiers themselves and the set of possible outputs. There will be an edge (a, b) in the set of edges E in between two vertices a and b if a is a message and the message activates classifier b or if a is a classifier ($T = internal$) whose action activates classifier b or if a is a classifier ($T = output$) and b is the output produced by that classifier. The input messages have only got outgoing edges, outputs only have incoming edges. Such a graph will be called an *activation graph*. An example of an activation graph is illustrated in figure 7.1.

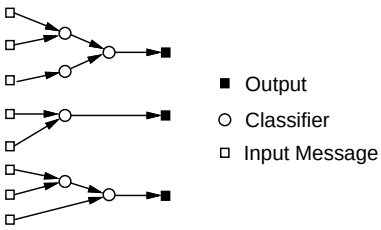

Figure 7.1: Example of an activation graph

All vertices representing input-messages and classifiers will have at most one outgoing edge. For classifiers with $T = output$ this is obvious as every classifier can have at most one output. For input-messages and classifiers with $T = internal$ this is less clear but it does become clear when you take B into account. An input message or the A (action) of an internal classifier will be able to match with multiple conditions of other classifiers, but they will only be able to activate one, as there is only one with the highest B (because $B \in \mathbf{R}^+$) and the one with the highest B will be the activated one.

We can see that from every vertex there is a path leading to at most one vertex representing an output or to a cycle. It is now possible to reduce the original graph G to a graph G' representing a single-layer classifier system. This is done in the following manner: We remove every vertex that represents a classifier and has no incoming edges connecting it to a vertex representing an input-message. We then remove all the edges that go into a vertex representing a classifier but that do not come from an input-message. In the last step we add a vertex representing the output *undefined* and lay edges between a vertex representing a classifier m and a vertex representing an output O if there is a path in the original graph G from the classifier m to the output O. If there is a path from classifier m to a cycle, we have an edge from the vertex representing m to the vertex representing the output *undefined*. The result for figure 7.1 is shown in figure 7.2.

This system is equivalent to a single layer classifier system. The classifiers from Ξ_M can be converted to classifiers from Ξ_S in the following way: there will be as many classifiers in the new system as there are vertices representing classifiers in the graph G'. For every such vertex we make a classifier with

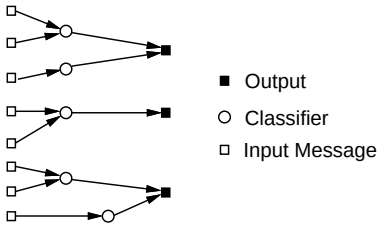

Figure 7.2: The reduced activation graph

a condition Q equal to the condition C of the classifier from Ξ_M that was associated with that vertex, with an output O that is the same as the output A associated with the vertex with which it is connected and with a strength P that is equal to the strength B of the original classifier.

The resulting classifier system computes the same function as the original one, because every classifier becomes activated from the same input-messages as in the original system, and every such classifier is connected to the same output, except for the ones that did not have an output in the original system because they ended in a cycle. These still have a value *undefined*, which can be used as a sign that the function value is undefined. We need this special value because we couldn't have left classifiers that end in a cycle out of the new system, as the activating messages could have activated other classifiers and altered the function.

The new system also has a number of rules that is equal to or less than the number of classifiers in the original system. We have only removed vertices from the graph representing the original system. After that we have made exactly one classifier per remaining vertex that represented a classifier, so the number of new classifiers must necessarily be less than the original number. For every $\zeta < 1$ it is true then that $|f(m)| < \zeta|m|$ which means that $S \geq M$. Quod erat demonstrandum.

7.4 Conclusions and Suggestions

The term practical computational strength can be used to show that classifier systems will need a comparable number of rules to compute a certain function. We have derived a non-trivial result in this way in proving that a message list makes no difference in a classifier system with simple conditions. It must be possible to prove other results in this way. The practical implications of this rather theoretic discussion are important: with a little theoretic effort we can save a lot of programming and computer time, as we can now choose the system that is the easiest to implement of all systems with equal computational strength.

Note however that from the conclusions in this chapter nothing can be derived about the learning-behaviour of the classifier systems. It is quite possible that certain systems that have an equal practical computational strength perform very differently on *learning* the same task.

The theoretic research into practical computational strength is also very interesting. Many things still need to be proven. It is for example still necessary to order the different extensions to classifier systems as mentioned in chapter 5 according to their practical computational strength. It is also an interesting question which classes of functions can be computed by the different classifier systems.

Chapter 8

A simplified Classifier System

Although the idea of classifier systems is rather simple, in practice these systems can become rather complicated. They have many free parameters and the implementing code tends to get very intricate. As Goldberg [GB89] says: *"It is very difficult to debug a classifier system as one monolithic block because of its size and its randomness."* My first version of a full-featured classifier system failed to learn almost anything at all, so I decided to implement a simpler version to check what exactly went wrong and to extend it step by step. Figure 8.1 shows how my classifier system failed miserably to learn the simple multiplexer-problem (which will be explained later.)

8.1 The simplified system

The system I then implemented is equivalent to Goldberg's [GB89](chapter 6), simple classifier system. This classifier system is extremely simple. It only has rules with a single condition and the conditions are only true if they are matched with a messages (so there are no unmatched-true conditions), it has no passthroughs and most important of all it has no message-list. Without a message-list it is only capable of implementing a single layer classifier system. This is a classifier system that only has rules which produce output directly from their input. There is no possibility of rule-chains being formed. Because of this the bucket-brigade algorithm also simplifies considerably.

Another change from my first attempt was the use of a different mechanism for bidding. In order to determine the classifier that is allowed to produce output, the classifier system uses a non-deterministic bidding-mechanism comparable to an auction. The mechanism that was used in my first, failed system was a roulette-wheel selection on the bids of the activated classifiers. It ap-

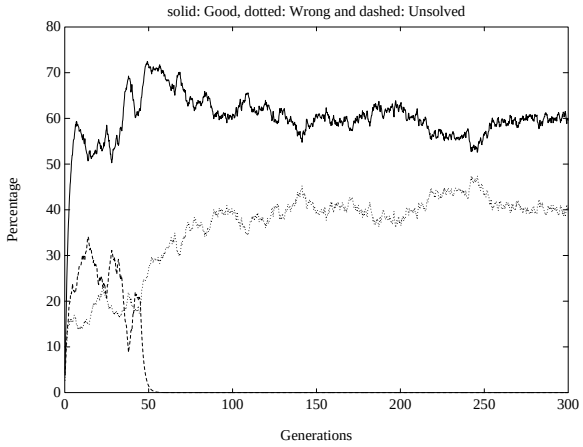

Figure 8.1: Failure of the (too) complex classifier system

peared after a few test-runs of the simplified system that this mechanism had a too high probability of choosing classifiers with a low bid. Therefore Goldberg's mechanism of adding a normal-distributed noise to the bids of active classifiers and then choosing the highest bid was adapted. His scheme also allowed for a higher level of control over the bidding.

8.2 Experiments with the correct rules present

Experiments were done with the same multiplexer system as Goldberg [GB89] describes. A multiplexer is a logic circuit that has data and an address as its inputs. The length of the address is k bits and the length of the data is 2^k bits. The output of the multiplexer then is the value of the bit in the data on the position given by the address (see figure 8.2). The size of the multiplexer is given as the number of bits in the data.

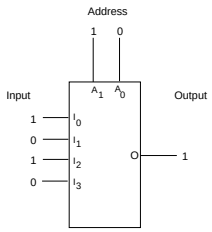

Figure 8.2: A 4-bit multiplexer

$00***1$	1
$01**1*$	1
$10*1**$	1
$111***$	1
$******$	0

Table 8.1: Default hierarchy for the four-bit multiplexer

The first experiment that was done was meant to test the ability of the classifier system to discern applicable rules from less applicable ones. This was done by constructing a classifier system that contains the correct rules for a given problem (given in table 8.1) together with a large number of random ones, which are called the monkey-wrenches. The total number of rules was fifty. The system must now be able to reach a 100% correct performance with these rules even without a genetic algorithm because the necessary rules are already present. If it would not able to do so the classifier system with bucket brigade cannot be considered a good way of determining the fitness of rules.

The experiments were done with two different bidding-schemes, with some different parameters. The graphs will show on the y-axis the average over 10000 runs of the system of the average of the payoff over ten consecutive answers of the classifier system. This payoff is 1 for a right answer and zero for either no answer or a wrong answer. This average of averages is necessary to prevent the graph from becoming too noisy. On the x-axis the number of training cycles that the classifier system has had are shown.

The first bidding-scheme was the roulette-wheel. The selection was done on the bids, the squares of the bids and the cubes of the bids. The results (see figure 8.3) were less than impressive even though the squared and cubed bids improved the performance a bit.

The second scheme was the noisy auction that Goldberg [GB89] has used. The amount of noise added was varied across the experiments. It shows from the results (see figure 8.4) that a certain small amount of noise gives the best results. The amount of noise should not be too low as a completely deterministic scheme does not appear to work very well probably because it tends to preserve the status quo too much. The deterministic scheme also is the only one that has a decrease in performance during the learning process. If this is significant it could be due to a phenomenon called *overtraining*, a problem that is also encountered in neural network research [HKP91]. Overtraining is a disproportional adaptation to the idiosyncrasies of the data set on which the system is trained. A noise-factor that decreases as the system is trained can be tried. However, this has not been attempted in this research.

These results show clearly that a classifier system can give credit to the right rules in a very effective way if the bidding system is right and if the rule set beforehand contains the rules that are sought for. Whether a classifier system

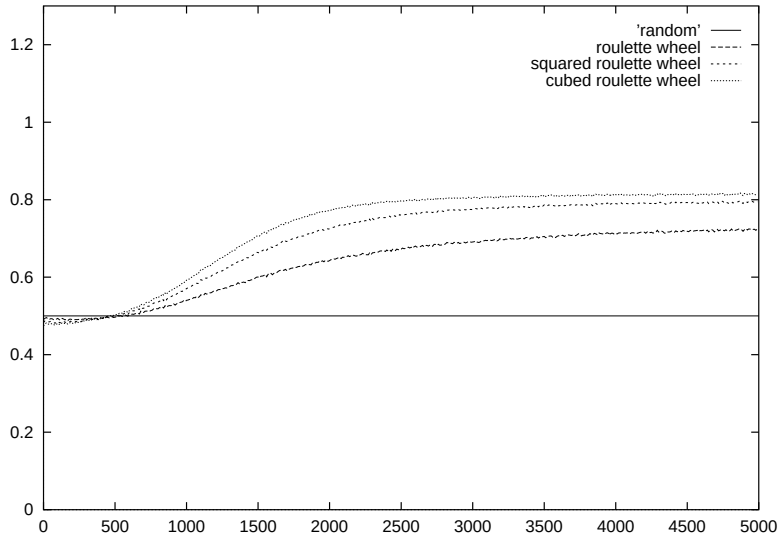

Figure 8.3: Monkey wrenches with roulette-wheels

with this credit assignment-scheme is also able to discover the right rule set is an altogether different question which we will discuss in the next paragraph.

8.3 Experiments without the Genetic Algorithm

If the classifier system is started with a random rule set it must be able to improve its performance by decreasing the influence of rules that give wrong answers. It would be a very great coincidence, however if the system would be able to reach a 100% correct answers with only fifty random rules as it is not very likely that these will contain all the necessary information to solve the problem at hand. Therefore we will introduce the genetic algorithm as a mechanism of inventing new rules for the system.

Before this we first need to consider the performance of a classifier system that starts with a random rule set, but that does not introduce new rules into the system, i.e. it will not use a genetic algorithm. This is shown in figure 8.5, which shows the same average of averages against the number of cycles as in the previous figures. In order to check if the bidding schemes influence the performance, the graph is repeated for four different values of sigma, the variance of the noise added.

From these results it is clear that the random classifier system without ge-

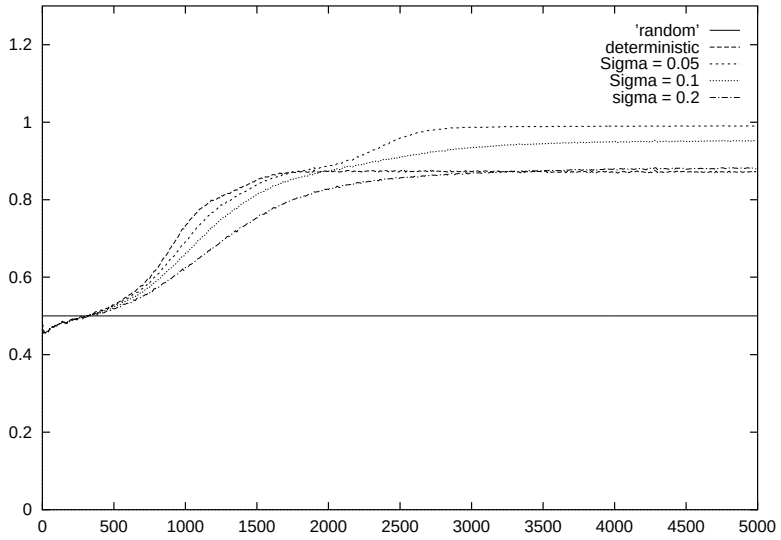

Figure 8.4: Monkey wrenches with noisy bids

netic algorithm is not a very good learning system. Whereas the performance of a random system would have been 50% correct, the performance of the tested system is 75% correct in the best case. This is reached after about 2000 cycles of the system and gives us an indication of how many cycles of the classifier system should be performed before the classifier system reaches a stable state and the genetic algorithm can safely take action.

The mediocre performance of the system can of course be explained by the fact that not all the rules that are necessary to solve the multiplexer problem are present at the outset and that no new rules are introduced. But we must also check if the ill-performing rules are weakened relatively to the well-performing ones. We expect that in situations where a good rule as well as a bad rule are applicable, the good rule will be the only one producing answers after a while. This is shown in figure 8.6.

However in situations where only a bad rule is applicable (approximately in 25% of the cases as it appears from the graph) the bid of this rule will become less and less and the rule will be rooted out by the genetic algorithm. The bids of rules giving good and bad answers are shown in figure 8.7. This graph shows the same average over 10000 runs of ten consecutive problems as the previous graphs.

We now have good evidence that application of a genetic algorithm on the rule set will improve the performance if we use the bid of a rule as the fitness

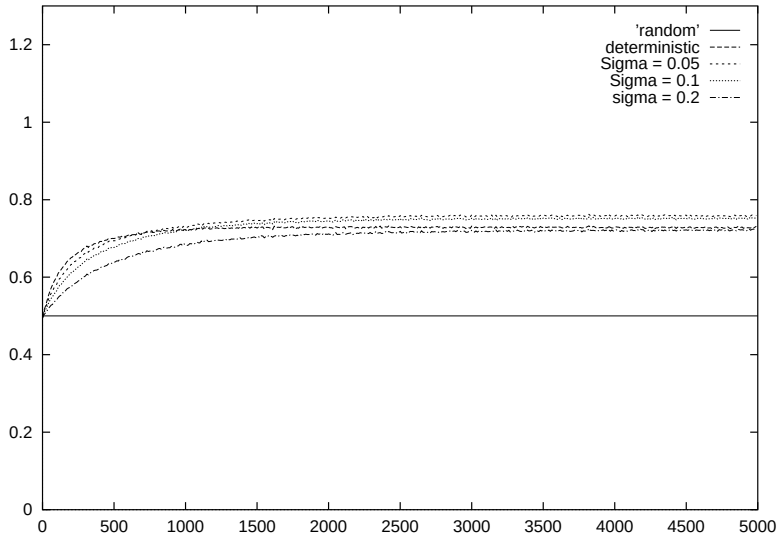

Figure 8.5: Random classifier system without genetic algorithm

of individuals in the population.

8.4 Experiments with the Genetic Algorithm

The application of a genetic algorithm to classifier systems when coding single rules as chromosomes is not really straightforward. An ordinary genetic algorithm tends to populate only one optimum, which translates to a search for a single best rule in a classifier system. But in a classifier system there must be a set of cooperating rules in order for it to be able to solve the multiplexer problem. We must therefore change the genetic algorithm so that it will not only optimize the fitness of the chromosomes but also their diversity.

A scheme that will do this is described by de Jong [DEJ75], and was adapted by Goldberg [GB89]. It is called crowding. The scheme used in the tests in this paper is directly adapted form Goldberg's book. In this scheme two parents are selected with roulette-wheel selection and crossed using crossover and mutation with a certain probability to produce a child. A chromosome with low fitness and which is very similar to the new child is then selected in the following way: a number of random subpopulations is generated. From each of these the least fit individual is chosen, and from this set of unfit individuals, the one chosen for replacement is the one that is the most similar to the new child.

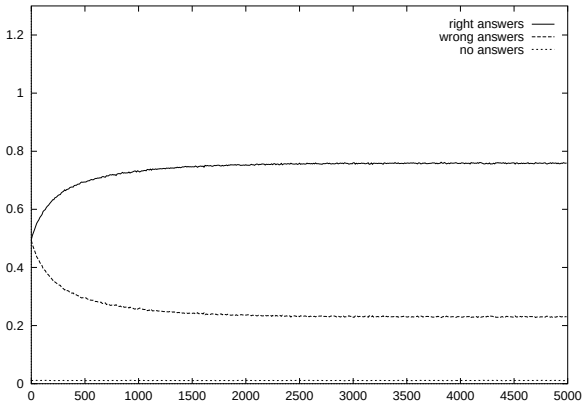

Figure 8.6: Good, Bad and no answers of a random CS without GA

That the use of a replacement-scheme using crowding is really necessary was shown by some preliminary results produced by the classifier system using a straightforward genetic algorithm. The performance soon dropped to less than 40% correct answers, where a random system would have reached 50%. This occurred probably because a lot of genetic information was lost as only one classifier was favoured.

With the use of crowding however the performance of the system shows a steady rise, shown in figure 8.8. The graph is shows the average of 500 runs of the classifier system, where every run has 50 generations of 2000 classifier system cycles each. The different graphs show the performance for different values of the ratio between the worst and best fitness used in roulette-wheel selection. In order to show more detail the graph is only shown for the performance-values in the range [0.7–0.95].

The graphs show a definite periodicity. This is caused by the action of the genetic algorithm, which tends to disturb the relations between the different classifiers a bit so that after the operation of the genetic algorithm the performance is somewhat less. Because the set of classifiers is a bit better though, the final performance after training is better on average so the overall performance is increasing.

The experiments have shown that a classifier system with bucket brigade is a useful way of machine learning. They also gave some clues as to how the learning-process proceeds. We can now start to study the learning-capacity of the system by presenting it with some real-world problems and by extending it with some extra features, like multiple conditions. These experiments are described in chapter 9.

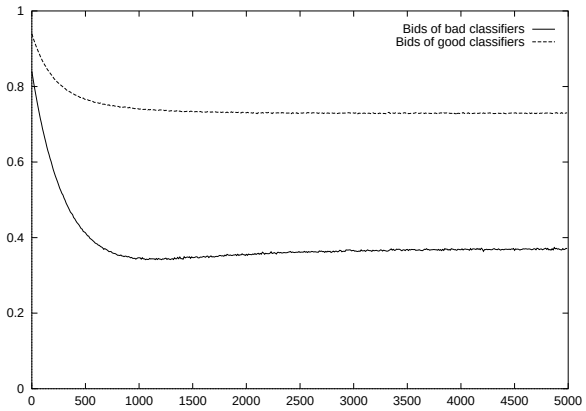

Figure 8.7: Bids of a random CS without GA

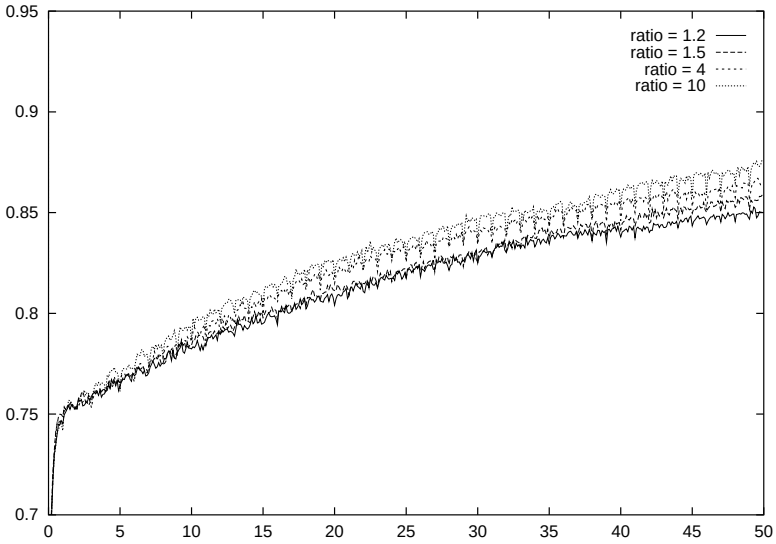

Figure 8.8: Performance of a CS with GA using crowding

name	value	description
populationsize	50	*Number of chromosomes in the population*
chromosomelength	20	*Length of the chromosomoes in genes*
classifiersystemsize	50	*Number of classifiers in the classifier system*
messagelength	6	*Length of a message*
crossoverprobability	0.8	*Probability of crossover between two chromosomes*
mutationprobability	0.001	*Probability of mutation of a gene*
alpha	0.25	*Ratio between the bids of two classifiers of equal strength, one without wildcards and one consisting of only wildcards*
initialstrength	10	*Initial strength of a classifier after initialization*
bidconstant	0.1	*Percentage of the strength of a classifier that is used as a bid*
wildcardprobability	0.33	*Probability of a wildcard when initializing a classifier*
badreward	0	*Reward for a bad answer*
goodreward	1	*Reward for a good answer*
bidtaxconstant	0.01	*Tax imposed on a bidding classifier*
lifetaxconstant	0.0005	*Tax imposed on every classifier at every cycle of the classifier system*
bidsigma	0.05	*Variance of the noise added to the bid of a classifier. Varied in the experiments without GA*
gen2gen	2000/ 5000	*Number of training cycles of the classifier system between two steps of the genetic algorithm*
worstbestratio	10	*Ratio between the worst and the best fitness in the population. Varied in the experiments*
maxgen	50	*Number of generations tested*
crowdingfactor	3	*Number of subpopulations tested to find most similar chromosome to new child*
crowdingsubpop	10	*Number of chromosomes in the subpopulation*
replaced	10	*Number of individuals replaced in a genetic step*

Table 8.2: Values of the parameters used

8.5 Parameters used in the Experiments

The results of these experiments must of course be reproducible. The software used can be obtained separately and will be described in an appendix In table 8.2 we will give the values of the most important parameters of the system.

Chapter 9

Further Experiments

In order to get a better picture of what the ability of classifier systems in practice are, some further experiments were done. These were performed with somewhat more realistic problems. The first problem was chosen to be bigger than the simple multiplexer problem. The second problem was chosen to be computationally more complicated.

9.1 The Mushrooms

In the first experiment, the classifier system was trained with data about the edibility of mushrooms. The dataset used is compiled by Schlimmer [SCH87] and contains 8124 samples of a poisonous and edible mushrooms from the agaricus and lepiota families. The mushrooms are described in terms of their form, colour, smell, habitat etcetera, etcetera. In total there are 22 attributes describing a single mushroom. From these 22 attributes the edibility of the mushrooms can be determined with about 95% accuracy. See [SCH87, IWL88, THOR92]. The database is available on internet [MUR87].

The mushrooms were coded as 57-bit messages and were given to a single-layer classifier system consisting of classifiers with single matched-tue conditions. There was no message list. The outputs consisted of a single bit, which was zero for a poisonous mushroom and one for an edible mushroom. The results obtained are shown in figure 9.1. This graph shows the average over the results of ten runs of 500 generations of classifier systems which were all trained with 2000 examples. These examples were randomly picked from a subset of 2000 mushrooms from the original database. This subset contained 1025 poisonous and 975 edible mushrooms.

The results show that this problem which is much more complicated than the multiplexer of chapter 8 because of its size can also be learned rather well, although the results are less good than the 95% accuracy obtained by other

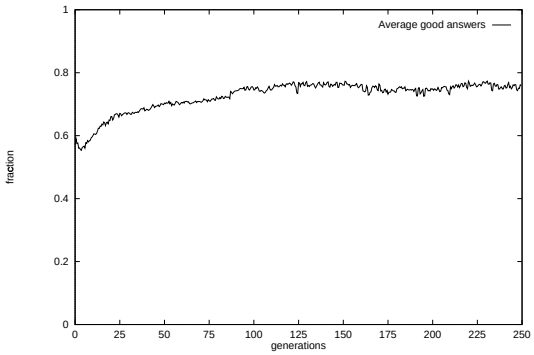

Figure 9.1: Results of experiment with mushrooms

learning-techniques.

9.1.1 Interpretation of the Learned Classifier System

It would be interesting to interpret the rules in the classifier system that results from the learning of the mushroom database. The strongest rules from the classifier system consisting of 50 rules originally after 100 generations of 2000 training-cycles each are presented in table 9.1. The system classified about 75% of the mushrooms correctly, which is not extremely good, but satisfying nevertheless. The value for life-tax used was 0.00005.

The interpretation of the classifiers is rather difficult. The bids give an indication as to which classifiers will be chosen for certain messages, but the noise added to them can change the answers of the system. Remember that the standard-deviation of the noise added is 0.01. The system is also rather ambiguous: a single message can activate different classifiers and classifiers that classify mushrooms as poisonous are activated by the same messages as classifiers classifying mushrooms as edible and vice versa. Furthermore no understandable default hierarchies are present, although it can be seen that some rules are preferred over others for certain messages.

However, some conclusions can be made, the most obvious one being that most parameters seem not to have any relation at all to the edibility of the mushrooms. These are the parameters that only have wildcards in all table-entries (parameters 2, 4, 6, 13, 16, 17, 18, 19 and 21). This does not say very much as another experiment (with some different parameters) gave a rather different classifier system. Some of the columns that have only wildcards in them in the experiment shown also had only wildcards in the other, but other columns had non-wildcards in them.

General patterns of which properties are important for the edibility and

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
**0	**	*1**	*	****	**	*0	*	*0**	*	***	**
**1	**	****	*	****	**	*0	*	*0**	*	***	**
***	**	****	*	****	**	*0	0	**0*	0	0**	**
**0	**	****	*	0**0	**	*0	*	****	*	***	**
**0	**	*1**	*	****	**	*0	*	*0**	*	***	**
**1	**	****	*	****	**	*0	*	*0**	*	***	**
***	**	****	*	****	**	*0	*	****	*	*0*	**
**1	**	****	*	****	**	**	*	****	*	***	**
**1	**	****	*	****	**	*0	0	**0*	0	00*	*0
***	**	****	*	****	**	*0	0	**1*	0	0**	**

13	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*edible?*	*bid*
**	****	*1**	*	**	**	***	****	***	*0*	yes	0.48
**	****	*1**	*	**	**	***	****	***	*0*	yes	0.40
1*	****	****	*	**	**	***	****	***	***	yes	0.39
**	*1**	****	*	**	**	***	****	***	*0*	yes	0.37
**	****	*1**	*	**	**	***	****	***	*0*	yes	0.28
**	****	*1**	*	**	**	***	****	***	*0*	yes	0.28
**	****	**0*	*	**	**	***	1***	***	***	no	0.32
**	***0	****	*	**	**	***	****	***	1**	no	0.30
**	****	*1**	*	**	**	***	****	***	***	no	0.28
1*	****	**0*	*	**	**	***	1***	***	*0*	no	0.24

number	*description*	*number*	*description*
1	cap-shape	12	stalk-surface above ring
2	cap-surface	13	stalk-surface below ring
3	cap-colour	14	stalk-colour above ring
4	presence of bruises	15	stalk-colour below ring
5	odour	16	veil-type
6	gill-attachment	17	veil-colour
7	gill-spacing	18	ring-number
8	gill-size	19	ring-type
9	gill-colour	20	spore-print-colour
10	stalk-shape	21	population
11	stalk-root	22	habitat

Table 9.1: The ten most fit classifiers after 100 generations and descriptions of the parameters

which ones are not are almost impossible to detect. This could be due to the fact that the percentage of correctly classified mushrooms is still too low or to the fact that the classifiers are too general.

Other conclusions are also very hard to draw. One thing to notice is that none of the classifiers seem to have more than one non-wildcard per parameter. This is a bit strange, because it was to be expected that in a rather random coding of the mushrooms (as the coding from mushroom-descriptions into binary strings obviously is) the relevant parameters should have to be specified completely. Maybe this is related to the fact that the starting probability of wildcards occurring was 90%. The percentage of wildcards in this final system is 93%, which is about the same. Further experiments would have to be conducted to find out what the influence of the starting-probability of wildcards is on the final classifier system.

The number of wildcards mirrors the results of section 6.1.2 in that the number of wildcards in the fittest classifiers seems to be dependent on the value of lifetax. In an experiment with higher lifetax (=0.0005) the number of non-wildcards in the fittest classifiers was about four. In this experiment it seems to be around eight. The dependency of the resulting classifier system on its starting-parameters is not desirable. A solution to this problem is difficult and relates to the basic problems of classifier systems: the emergence of stable default-hierarchies and the search for cooperating classifiers. The main conclusion that can be drawn from this system (and the much larger system resulting from the experiment in the next section) is that the computation that takes place in a classifier system seems to be of an extremely distributed nature.

9.1.2 A larger Population

The fraction of 75% correctly classified mushrooms seems a bit meager compared to the fraction of 95% obtained in other experiments. The experiments were therefore reran with a larger population. The population was 200 instead of 50 in the original experiment. The results 9.2 of this test-run are much more impressive: a fraction of 90% correctly classified mushrooms was reached. A small modification to the bucket-brigade algorithm improved the performance even more. The classifiers still adjusted their bids according to their specificity, but the amounts they paid were now made independent of the specificity (the formula for calculating the amount to be paid now became: βS). The results of this experiment are also shown in figure 9.2. The fraction of correctly classified mushrooms now became about 95%, which happens to be the maximum obtainable score. No interpretation of the learned classifier system was undertaken, as the number of relevant classifiers (the classifiers with a reasonably high bid) was much too large.

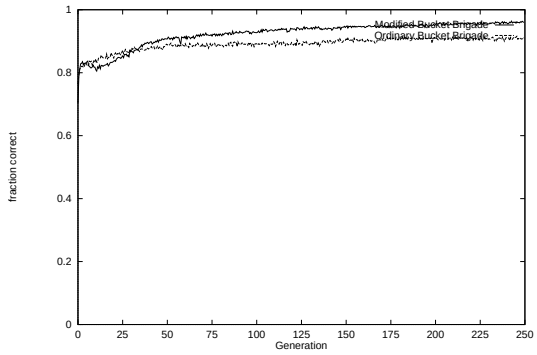

Figure 9.2: Result of mushrooms with larger population

9.1.3 Generalization

Another small experiment was conducted to examine the ability of the system to generalize what it has learned. This was done with the large classifier system with the ordinary bucket-brigade.

Generalization is the ability to classify correctly samples from the problem-space that the system has not seen during training. In the test that was done, the system was trained first with a random subset of 2000 mushrooms from 8124. After this generalization was tested with 100 other mushrooms that were randomly selected from the 8124. The system classified correctly about 90% of the mushrooms that were not in the training-set and it also classified correctly about 90% of the mushrooms in the training-set. This is evidence for good generalization.

9.1.4 Parameters

We will present the parameters of the system used in the experiment in table 9.2. Most of them are equal to the ones used in the experiment with the simple multiplexer-problem. Some important ones have essentially different values. These include the life-tax, the value of alpha and the size of the classifiers and chromosomes.

9.2 The Parity Problem

The single-layer classifier systems investigated thus far were only able to learn problems that were solvable by logical expressions in disjunctive normal form that have a number of clauses that is less than or equal to the number of classifiers as each classifier can take care of a clause in the disjunctive normal form.

name	value	description
populationsize	50/200	*Number of chromosomes in the population*
chromosomelength	180	*Length of the chromosomoes in genes*
classifiersystemsize	50	*Number of classifiers in the classifier system*
messagelength	57	*Length of a message*
crossoverprobability	0.8	*Probability of crossover between two chromosomes*
mutationprobability	0.001	*Probability of mutation of a gene*
alpha	0.01	*Ratio between the bids of two classifiers of equal strength, one without wildcards and one consisting of only wildcards*
initialstrength	10	*Initial strength of a classifier after initialization*
bidconstant	0.1	*Percentage of the strength of a classifier that is used as a bid*
wildcardprobability	0.9	*Probability of a wildcard when initializing a classifier*
badreward	0	*Reward for a bad answer*
goodreward	1	*Reward for a good answer*
bidtaxconstant	0.01	*Tax imposed on a bidding classifier*
lifetaxconstant	0.00005	*Tax imposed on every classifier at every cycle of the classifier system*
bidsigma	0.05/ 0.01	*Variance of the noise added to the bid of a classifier. Varied in the experiments without GA*
gen2gen	2000	*Number of training cycles of the classifier system between two steps of the genetic algorithm*
worstbestratio	10	*Ratio between the worst and the best fitness in the population. Varied in the experiments*
maxgen	100	*Number of generations tested*
crowdingfactor	3	*Number of subpopulations tested to find most similar chromosome to new child*
crowdingsubpop	10	*Number of chromosomes in the subpopulation*
replaced	10	*Number of individuals replaced in a genetic step*

Table 9.2: Values of the parameters used in the mushroom-experiment

Some problems require an extremely large number of clauses in the disjunctive normal form. One of these problems is the parity-problem.

The parity of a string of bits is one if the number of bits is odd and zero if the number of bits is even. The problem of determining the parity of a bitstring of n bits is known as the *n-bit parity problem*. The parity problem is a problem that is not solvable with a logical expression in a disjunctive normal form with a number of clauses that is polynomial in the number of bits in the bitstring. In order for a classifier system that has a number of classifiers that is polynomial in the number of bits in the messages to be able to learn this problem we have to add extra features, like a message-list and multiple conditions to our simple classifier system.

9.2.1 The Problem

The first problem studied was the four-bit parity problem. This problem is in fact extremely trivial. It can be solved by a classifier system consisting of 16 classifiers, each of which gives the parity for one of the 16 possible strings of four bits.

However we can use it as an example to illustrate a method to solve the parity-problem that only needs classifier systems that have an amount of classifiers that is linear in the number of bits. The method is of the divide-and-conquer type. We know that the one-bit parity problem is easily solved. And if we know the parities of two strings of n bits, we can easily calculate the parity of the concatenation of these strings of $2n$ bits by taking the XOR of the two parities.

In this way we can calculate the parity of a bitstring with only an amount of classifiers that is linear in the number of bits of the strings. The classifier system must have three features: it must have a message list, it must have two conditions per classifier and it must have messages that are one bit longer than the bitstrings of which the parity is to be calculated in order to allow for a distinction between internal and external messages.

The total number of classifier is then $6n-4$ where n is the number of bits of the parity-problem. We now need $2n$ classifiers to calculate the parities of the 1-bit strings in the problem and $4 \cdot \frac{n}{2^k}$ classifiers in the kth layer, in which the first layer that has internal messages as inputs has number 1. The total number of classifiers N is then (for n a power of two):

$$N = 2n + 4n \sum_{i=1}^{log_2 n} \frac{1}{2^i}$$

which solves to $6n-4$. This is, by the way, also the basis of the proof that a multi-layer classifier system with multiple conditions has a higher practical computational strength (see chapter 7) than a (multi-layer) classifier system with single conditions. An example of this kind of classifier system is given in table 9.3.

9.2.2 The Test Runs

The first experiment that was done was one with a 4-bit parity problem. This task can easily be learned by a single-layer classifier system by just finding the rules for the 16 different possible 4-bit strings. Actually this would just be rote-learning. The average number of correctly classified strings over 10 testruns of 50 generations each is shown in figure 9.3. It is clear that this simple task can be learnt almost perfectly.

Also shown in this figure is the average over 10 testruns of 50 generations for an 8 bit parity-problem. It is clear that the system has great difficulties to learn. The behaviour of the system shows some initial improvements, but it

layer 0 *conditions*	*action*	*parity*	*layer 1* *conditions*	*action*	*parity*	*layer 2* *conditions*	*output*	*parity*
00***			10000			11000		
00***	10000	0	10010	11000	0	11100	0	0
01***			10000			11001		
01***	10001	1	10011	11001	1	11100	1	1
0*0**			10001			11000		
0*0**	10010	0	10010	11001	1	11101	1	1
0*1**			10001			11001		
0*1**	10011	1	10011	11000	0	11101	0	0
0**0*			10100					
0**0*	10100	0	10110	11100	0			
0**1*			10101					
0**1*	10101	1	10110	11101	1			
0***0			10100					
0***0	10110	0	10111	11101	1			
0***1			10101					
0***1	10111	1	10111	11100	0			

Table 9.3: Four-bit parity problem with internal messages

seems to degrade after a while. This has to do with the fact that a single-layer classifier system needs an exponential number of classifiers in order to solve the parity-problem. It appears to be necessary to have a multi-layer classifier system to solve larger parity-problems.

That a classifier system with bucket brigade alone—that is, without the genetic algorithm—is able to make a distinction between good and bad rules has been shown in chapter 8. We have repeated the experiment from that chapter with the classifier system without the genetic algorithm that has the proper rules (together with 20 random rules) present, which we called the monkey-wrenches-experiment. In figure 9.4 the results are shown. It is clear that the classifier system is perfectly capable of finding the right rules to perform the task.

The classifier system *with* genetic algorithm seems to have great difficulties to learn the task. Even if we start with a classifier system that does have the correct rules, it appears that it is not able to preserve these in the long run. In figure 9.5 we show the results over ten generations of the classifier system initially has the correct rules (together with twenty random rules) present. It appears that the performance degrades rapidly when we apply the genetic algorithm.

This is not unheard of in classifier systems. As Compiani et al. write: "*The combined action of bucket brigade and genetics leads to performance instabilities in the long term...*" [CMS90]. In our particular example this seems to be caused by two factors: the lower strength of rules at the beginning of a rule-chain and

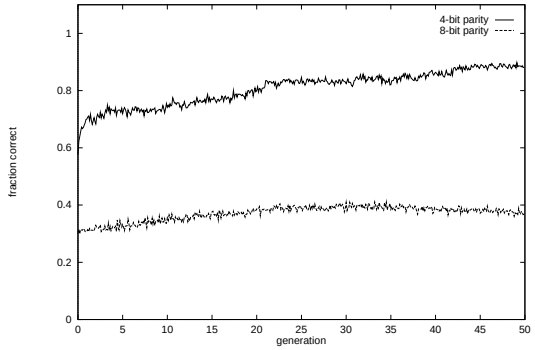

Figure 9.3: Single-layer classifier system learning parity-problem

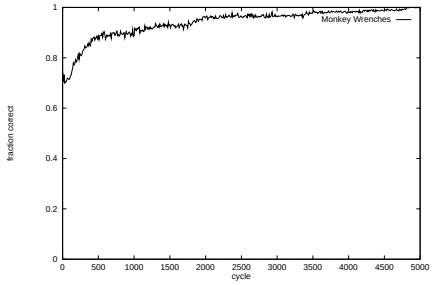

Figure 9.4: Parity-problem with monkey wrenches

the completion between similar highly fit rules.

The lower strength of classifiers early in a rule-chain is caused by the presence of taxes. The amount a certain classifier in its steady state pays to the classifiers activating it is always lower than the amount of reward it receives. This is especially the case when a classifier is activated by two different classifiers. Then each of the activating classifiers receives less than half the amount of pay the activated classifier receives. The net effect is that classifiers at the end of a rule-chain get a much higher steady-state-strength and bid than classifiers at the beginning and therefore the action of the genetic algorithm is biased in favour of rules at the end of the chain, while actually all the rules are equally valuable.

An experiment done with the monkey wrenches without the genetic algorithm confirmed this nicely. The output-producing rules had a strength of 7.47 after 5000 generations (which agrees quite well with the value of 7.41 predicted

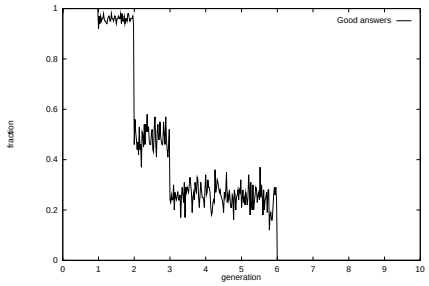

Figure 9.5: Failure of the parity problem with monkey-wrenches when applying a genetic algorithm

by formula 6.3 [†]). The rules in the layer below this one, however, had a strength of only 3.46, (which agrees exactly with the predicted value). But as neither had any wildcards in their conditions, the bid and the fitness of the lower layer was much lower. They were both equally necessary, so the fitness does not describe the usefulness very well.

The second factor that was found to cause instabilities in the action of the genetic algorithm was the fact that the genetic algorithm tended to create copies of extremely fit rules. These copies then competed with the original rules for the same resources, effectively halving the rewards both rules received. The rewards that all the rules down the rule-chains received were then halved as well, causing their strength, bids and fitness to drop even more, making them likely candidates for replacement by the genetic algorithm.

Once the genetic algorithm removed one of the rules in the correct solution of the parity-problem, the performance of the system decreased dramatically, because all the rules were of equally vital importance for its functioning. As the rules did not do anything useful anymore, their strengths decreased even more, causing the genetic algorithm to eliminate them.

Many solutions to the problems mentioned above can be proposed. The first could be to change the paying-mechanism of the bucket brigade. This could solve the first problem, but not the second. A solution to the second problem could be to change the parameters of the genetic algorithm.

One possibility is to increase the number of individuals. The chance that a good individual is replaced will then be smaller. However the problem of the duplication of highly fit individuals will remain. Another problem is then that the classifier system has difficulties picking the right solution from among the random rules. An experiment with the monkey-wrenches system showed that

[†]It is possible to use the function for the steady state strength without life-tax, even if we do have a very small life-tax. It so happens that the classifier we are estimating the steady-state-strength for is activated almost every cycle.

this didn't really work.

Another possible modification to the genetic algorithm is to choose only individuals that are very similar for crossover. This to preserve diversity in the population and to prevent fit schemes from becoming disrupted too quickly. In the original scheme the technique used for this purpose was to select for replacement only the individuals that were rather similar to the new child. This idea was shown to have very little effect in the monkey-wrenches experiment as well.

We can also try to modify the bucket-brigade algorithm. The idea that was used in the mushroom-experiment (to have all classifiers, regardless of their specificity, make the same pay if they have the same strength, but to have the specificity still influence the bidding) didn't seem to have any effect on the monkey-wrenches-experiment.

The emergence of rule chains that remain stable under the influence of the genetic algorithm seems to remain a major problem for classifier systems. Recent literature should be studied to search for solutions, be it in modifications of the genetic algorithm, the bucket brigade or in more radical changes in the classifier system paradigm (e.g. [DOR91, SMI92, SMIBO93, VAL93]) and for mathematical research into the long-term behaviour of classifier systems, for example with techniques from the dynamic systems research (e.g. [CMS90, FAR90, FOR90, MON93]).

9.2.3 Parameters

The parameters of the experiments of the parity-problem are given in table 9.4.

name	**value**	**description**
populationsize	40	*Number of chromosomes in the population*
chromosomelength	40	*Length of the chromosomoes in genes*
classifiersystemsize	40	*Number of classifiers in the classifier system*
messagelength	5	*Length of a message*
crossoverprobability	0.8	*Probability of crossover between two chromosomes*
mutationprobability	0.001	*Probability of mutation of a gene*
alpha	0.25	*Ratio between the bids of two classifiers of equal strength, one without wildcards and one consisting of only wildcards*
initialstrength	1	*Initial strength of a classifier after initialization*
bidconstant	0.1	*Percentage of the strength of a classifier that is used as a bid*
wildcardprobability	0.39	*Probability of a wildcard when initializing a classifier*
badreward	0	*Reward for a bad answer*
goodreward	1	*Reward for a good answer*
bidtaxconstant	0.01	*Tax imposed on a bidding classifier*
lifetaxconstant	0.00001	*Tax imposed on every classifier at every cycle of the classifier system*
bidsigma	0.05	*Variance of the noise added to the bid of a classifier.*
gen2gen	5000	*Number of training cycles of the classifier system between two steps of the genetic algorithm*
worstbestratio	10	*Ratio between the worst and the best fitness in the population. Varied in the experiments*
maxgen	10/50	*Number of generations tested*
crowdingfactor	3/10	*Number of subpopulations tested to find most similar chromosome to new child*
crowdingsubpop	10	*Number of chromosomes in the subpopulation*
replaced	10	*Number of individuals replaced in a genetic step*
matingfactor	3	*Number of subpopulations tested to find most similar partner for mating.*
matingsubpop	10	*Size of the subpopulation tested to find most similar partner.*

Table 9.4: Values of the parameters used in the parity-experiment

Chapter 10

Classifier Systems seen as Networks

The way in which classifier systems are usually implemented is not always very efficient. All conditions of all classifiers are compared completely against all messages in every cycle of the system. But it is possible to avoid a lot of these comparisons by using information of how the classifiers in the system interact.

If we have a multi-layer classifier system, then in all the layers except for the input-layer, we know which messages can be present (only the ones that can be an action of the classifiers in the system) and with which conditions these can match. It is therefore perfectly possible to determine beforehand which classifiers can be activated —or inhibited in the case of unmatched-true conditions— by which other classifiers. This information can be expressed in the form of activation- or inhibition links in a network of classifiers. We now only need to match the initial input-message against all the conditions of the classifiers.

This process of determining the links between classifiers is like "compiling" a classifier system as opposed to the usual way of implementing it which can be compared to interpretation of programming-languages. Instead of determining at every cycle of the system which message matches against which condition, we only check which message matches against which condition once in the compilation-process and then follow the links between classifiers. No need to say that this compilation is only efficient in a multi-layer classifier system. Passthroughs give special problems as well.

10.1 Implementing it

The implementation of a "compiled" classifier system to compare it with an "interpreted" one was not part of the research of this paper, but we will give

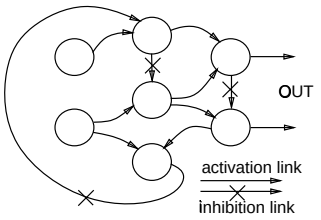

Figure 10.1: The classifier system as a network

some suggestions of how this can be done as a future project.

10.1.1 The activation

First of all the activation- and inhibition links that exist between the classifiers in the system should be determined. If there are no passthroughs in the system this is a straightforward procedure so we will assume that there are none in this section. The next section deals with the case were there are passthroughs.

The links are found by matching all the available messages (of the classifiers that do not produce output) in the system against all the conditions of classifiers in the system. If a message of a classifier matches with a matched-true condition of a classifier, an activation-link is placed between these two classifiers. If the message matches with an unmatched-true condition, an inhibition-link is placed between them.

When the compiled system is run, the first step that is taken is to check which classifiers are activated by the input-message. These classifiers are then considered active and the ones with the highest bids that would have been allowed to place their messages in the message-list in the interpreted system now activate their outgoing links to the other classifiers in the system. The new set of active classifiers will now be the set of classifiers that has an incoming active activation-link for every matched-true condition and no incoming activated inhibition-links for every unmatched-true condition. The process can now start again.

Of course, when the highest bidding activated classifier *does* have output the cycling is stopped and its output is taken as output of the system.

10.1.2 The paying of bids

For the system to be able to learn there must be a way to update the strengths of the classifiers. This is done by selecting one incoming link per matched-true condition of an activated classifier. For example only the links connected with the highest-bidding classifiers are selected. A random scheme, in which a link is randomly selected is also possible. This would be more like the definition in

section 5.2.5. The classifiers responsible for the activation of these links now receive a payment from the freshly activated classifier. This payment is, as in the ordinary classifier system equal to the bid of this classifier divided by the number of conditions. The process is illustrated in figure 10.2.

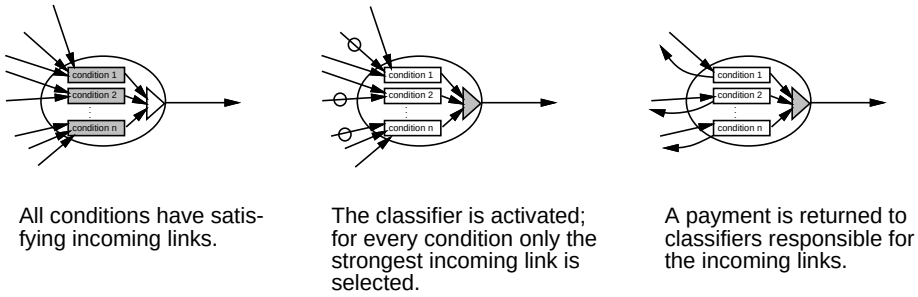

Figure 10.2: The activation of a classifier in a network

The activations are propagated through forward links between classifiers, the updates of the strengths are propagated through backward links. This is not unlike the neural network learning paradigm known as backpropagation [HKP91]. The similarities are discussed in section 10.4.

10.2 Problems with Passthroughs

Passthroughs were defined in chapter 5 as positions in the action of a classifier that will be filled in by the corresponding positions of one condition (which we called the "preferred condition" in section 5.2.2). If there are wildcards at these positions in the condition then the corresponding character of the message that was responsible for activating this condition is substituted. If there are more messages that could have activated the condition, the one from the highest-bidding classifier is taken.

It might seem that this poses no problems at all, as we can pass passthroughs along via the activation-links (or by using parallel passthrough links) so there will be an extra network that passes variables between classifiers. A problem arises when we are building the network and we try to determine where the activation- (and inhibition-) links will come. We usually do not know whether the passthroughs will be substituted by a one or a zero so there will be a possibility that the action containing the passthrough will match with a certain condition in one case and not in the other. One solution to this problem is the use of *conditional links*. A conditional link is activated when the classifier from which it departs is activated and the passthrough that it is associated with has the right value.

A classifier system with passthroughs can thus be implemented by augmenting the activation- and inhibition-links with variables (having the value of the associated passthroughs in the original system) and conditions, which only activate a certain link when the passthroughs have a certain value. This is illustrated in figure 10.3.

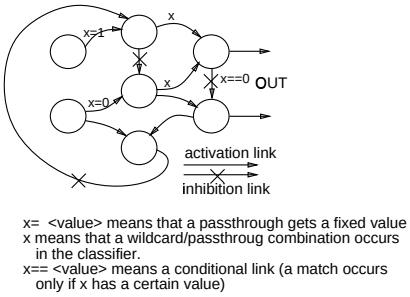

Figure 10.3: Classifier system with passthroughs as an augmented network

10.3 Speedup

The question now is: what is the speedup that is achievable when we compile a classifier system into a network? First we must estimate how much time the compilation process itself takes. For the compilation it is necessary to match all the actions of the classifiers against all the conditions. The conditions and actions will have to be compared position by position. Using the notation of section 6.3 where N is the number of classifiers, k the number of conditions per classifier and l the length of a condition the comparison of all the conditions of all the classifiers with all the actions of the classifiers costs $O(N^2 \cdot k \cdot l)$ time. The determining of the passthrough-links and the conditional links takes time in the order $O(N^2 \cdot l)$ as we must check for every passthrough (at most l) in every message with every preferred condition (only one per classifier), if it is initialized, passed on or if it causes a conditional link.

The time the compilation takes is extra compared to what is needed for an ordinary classifier system. Any speedup must be caused by speedup in the cycling-process.

The cycling of an ordinary multi-layer classifier system takes $O(N^2 \cdot k \cdot l)$ time (see section 6.3). A network takes time $O(N \cdot k \cdot l)$ to match the input-message with all the conditions of all the classifiers (which is also the time taken for a single-layer ordinary classifier system). Subsequent activations of classifiers take $O(N^2 \cdot k)$ time for the propagation of activation because at most N classifiers are active and will match with at most N other classifiers that have k conditions each. Paying of bids will cost at most $O(N \cdot k)$ time (at most N classifiers have to

pay k conditions). The propagation of passthroughs causes at most N classifiers to pass l passthroughs to N other classifiers. This would add $O(N^2 \cdot l)$ time. As l is usually a lot greater than k the total time needed would be $O(N^2 \cdot l)$ so the speedup would be at least a factor k.

The theoretical calculation of the speedup is not very useful, however, as we know very little about the real number of passthroughs or outgoing links per classifier. In a more intuitive estimation of speedup we can say that in order to determine the active classifiers in a network as opposed to an ordinary classifier system we do not have to compare the complete messages to the conditions, position by position, as we have done that while compiling the system. This can reduce the necessary work by a factor l. Also we do not have to compare the message of an active classifier to the conditions of *all* the classifiers as we have determined beforehand which classifiers can be activated by any other classifier. If we assume that actions of classifiers activate only a small subset of the total number of classifiers, this removal of possible candidates to be activated can also give a lot of speedup.

All in all the compilation of an average classifier system into a network can save a lot of time when cycling the system. Two conditions must be met however. First the number of cycles of the system must be large compared to the amount of work necessary to compile it. This is almost always the case (2000 cycles in the experiments performed in this paper). The second and most important condition is that the classifier system should be a multi-layer system. The matching of the input-message costs as much time in a network as in an ordinary system. The speedup must be achieved in subsequent cycles of the system.

10.4 Neural Network Analogy?

As classifier systems can be made to resemble a network, we can now ask ourselves the question: in what ways are classifier systems comparable with *neural* networks. Neural networks (see for example [HKP91, SIMP89]) are learning systems that are modelled after the way the animal neural system works (see e.g. [ALB83]). They consist of nodes that are connected with links. The nodes can be activated. This activation is usually expressed by a real number that can have values within a certain range, or have a number of discrete values. The links into a node have a certain weight that can also be expressed as a real number.

The activation of a node is determined by a function of the sum of the weights of the incoming links multiplied by the activations of the nodes from which these links originate. In this way nodes influence each other.

A network learns by updating the weights of the links in a manner that agrees with the task to be learned. Many different paradigms of neural networks exist, the main ones being Hopfield networks (which have symmetrical

connections) with Hebbian learning and multi-layer feedforward networks with error backpropagation (these do not have cycles).

As classifier systems can have asymmetric connections as well as cycles, we cannot directly compare them with these paradigms. However the description of neural networks and classifier systems seen as networks are so much alike that it appears interesting to find some analogies. Some efforts have been undertaken to find mappings from classifier systems to neural networks, see for example [SMIBO93, FAR90]. As farmer [FAR90] writes: "*The classifier system is rich with structure, nomenclature, and lore, and has a literature of its own that has evolved more or less independently of the neural network literature. Nonetheless, the two are quite similar[...]*"

10.4.1 The activation-function

Instead of concentrating on finding an exact mapping from classifier systems to neural networks, which is always possible by stretching the definitions of both concepts far enough, we will try to say something about the conceptual differences between the classifier system as used in this paper and between neural networks as they are usually implemented.

The neurons in neural networks have an activation function. The classifiers in fact have activation functions as well. These were described in the previous section. The main difference between the activation function of a classifier and that of a neuron is that the one of the classifier system requires non-local information. This because of the fact that only a limited number of the highest bidding classifiers is allowed to become active. The activation of only a limited number of nodes is not unheard of in neural networks and can be implemented by connections between nodes. It still tends to complicate the networks very much however and is not a desirable situation, especially in configurations with many nodes.

Another difference between neural networks and classifier systems is that the activation of a neuron is determined by the weights of incoming *links*, the activation of a classifier is determined by the weight of *the node itself*. This also has some implications for the learning-process, as we will see below.

The way input is handled in a network of classifiers is also a bit of a problem. As classifiers can be activated by internal as well as external messages, we effectively have two different kinds of activation. The activation of classifiers by external messages can be handled by just making these classifiers active in the first step of the cycling of the classifier system. The strongest of the activated classifiers will then be allowed to stay active and activate other classifiers. The other possibility is to add an extra part to the network that converts an input message into the activation of the right classifiers. This can be done by creating an extra node for every bit in the input-message. These nodes output -1 if the message has zero and 1 if the message has a 1 at that position. The links have weight 1 to every classifier that needs a 1 at that position, a -1 for every needed

0. There is no link for every wildcard. The classifier is then activated if the sum over all the incoming activations times the weights of the links is equal to the number of non-wildcards in the conditions. Of course the classifier is also still activated if all the incoming links from other classifiers are activated.

10.4.2 The learning-rule

The way in which a network of classifiers learns has been described in section 10.1.2. There are some more differences between classifier systems and neural networks here. First of all we are dealing with an update of the strengths of nodes instead of the strengths of links. As there are usually many more links than nodes, this could be an indication of less flexibility of classifier systems, for example in the case where a classifier produces an action that simultaneously generates good behaviour in one classifier and bad behaviour in another.

Also in classifier systems we only have updates of connections between nodes of which at least one has been activated. In some neural network learning schemes weights are updated between all nodes, regardless of their being activated or not.

There is much more that can be done with classifier systems as networks. It is very well possible to generate a network with a classifier system and then train it with a specialized neural network training algorithm. Obviously we must make some changes to the definitions of fitness of a classifier in this case. This is the idea behind the paper of Smith [SMIBO93]. The combination of classifier systems and neural networks is a promising area of research, especially where finding ideal structures for neural network is concerned.

Chapter 11

Conclusions

The area of classifier systems is still a very novel area of research. This means that there is still a lot of confusion about terminology, what exactly classifier systems are and that there still is no accepted set of benchmarks to compare the performance of different kinds of classifier systems with. In this thesis some means to compare classes of classifier systems with each other have been suggested. A formal definition has been attempted; a means to compare the practical computational strengths of classifier systems has been presented and two examples from the literature have been trained to a classifier system. A third problem, the parity-problem, which is infeasible for single-layer classifier systems, has been proposed as a test for more complex classifier systems, but was found to be extremely hard and the systems tested in this thesis were not able to learn it.

11.1 Possibilities

Still the two problems that my classifier system was able to learn show that this particular learning system has great possibilities. Generalization was very good, learning was smooth and it is suspected that classifier systems are reasonably immune to noise, but this was not tested in the experiments in this thesis. It was found however, that the interpretation of classifier systems is quite difficult. The number of rules necessary to solve any realistic problem is too large to allow for a straightforward and comprehensive interpretation.

The implemented system was also found to confirm some of the theoretically derived formulas. These were the ones about the steady state strength and the one about the starting number of wildcards. The other formulas that were presented in this thesis still have to be tested.

Classifier systems are also suspected to be very much akin to neural networks. A method to convert a classifier system into a network was suggested, but not

implemented. This technique was quite different from other methods to convert classifier systems into neural networks presented in the literature. It does also not result in a network that is like any known neural network, but it was mostly intended as a method to speed up the execution of classifier systems. It should be quite possible, however, to change the learning method of the classifier system so that it is much more like a neural network.

11.2 Problems

Some problems were encountered in the research performed for this thesis. One of these, the difficult interpretation of the learnt classifier systems has already been mentioned. No understandable default-hierarchies were found and chaining of rules was also not found. The lack of understandability of default-hierarchies can probably be explained by the differences that exist between a hand-coded optimal solution to a problem (which usually is very sensitive to small changes) and the optimal solution found by a genetic algorithm (which is expected to have more distributed and thus less understandable computation).

Another problem is the strong dependency of the performance on the starting parameters. The difference in performance between the small system and the large system learning the mushroom-task in section 9.1 is illustrative in this respect. The only difference was the size of the population of classifiers learning the task.

There are also more subtle dependencies on the starting-parameters. As also mentioned in section 9.1 the number of wildcards in the resulting systems depended on the value for τ_{life}, the lifetax. It is suspected that these subtle dependencies on the starting-parameters can greatly influence the learning-capabilities of the classifier system. Also the size of the messages, the message-list and the number and kind of conditions are thought to be of great influence on the classifier system.

Another, quite different problem is the stability of classifier system under the influence of the genetic algorithm. In section 9.2 it was found that the genetic algorithm can destroy a good solution. This is obviously a great problem. It is hoped that in practice classifier systems will be able to learn more distributed and thus more stable solutions to tasks, instead of the rather brittle hand-coded solution tested in that section. No experiments were done to investigate this.

Many variations to the combination of bucket-brigade and genetic algorithms can be tested to find one that produces a stable solution. Some of these have been tested in section 9.2 but have been found inadequate. Maybe a more radical restructuring of the genetic algorithm or the bucket-brigade is necessary here. Maybe it will even be found necessary to use non-local information, although this really would violate the philosophy behind the classifier system with bucket-brigade, which stresses the use of local information only. Maybe a coding of classifier systems as a graph-grammar (see [BK92] for an example of using graph

grammars in the search for neural network structures) could be tried.

It is probably also possible to find out more about classifier system behaviour in the long run by applying analysis-techniques from dynamic systems research to them in a bit less ad hoc way than in this thesis.

11.3 Suggestions for Future Research

Many questions remain unanswered and many topics remain unresearched by this thesis. The only kind of learning system that was investigated by this thesis was of the simple stimulus-response kind. Classifier systems that are recurrent have not been researched at all. Recurrent classifier systems are systems that can take a new input while they are still processing the previous ones. This can be used to give classifier systems a memory of past events. The stimulus-response kind is not able to do that and is therefore not able to learn time-dependent tasks.

Also the issue of noise was not studied. It was actually assumed that classifier systems are relatively immune to noise. This because so many examples are needed to learn a certain task. Noise will then be "averaged out". But this was not tested in experiments. It must still be determined how noise-independent classifier systems really are.

Another important thing is the stability and tolerance to damage of classifier systems. It is possible to regard a classifier system as a program in a parallel programming language. This program can be made robust by means of redundancy: a certain task is then performed by many rules that can back each other up. If one of the rules is removed, the performance of the system will be hardly affected.

Handcoded classifier systems, just like hand-coded computer programs in ordinary programming languages are hardly robust, as has been shown in section 9.2. Removal of one rule degrades the performance enormously. It is hoped, however that classifier systems that are found by the genetic algorithm are less brittle, after all, this is what classifier systems were supposed to be good at in the first place [HOL86]. This could be tested by removing rules from a classifier system that was found by a genetic algorithm and then checking how this affects its performance.

Then there is the problem of starting-parameters. It was found that the learning-behaviour of classifier systems can sometimes be highly dependent on the parameters of the system. It must be determined how these parameters influence the behaviour of the system and what their optimal value is. This is a task that is very computationally intensive. Maybe the best thing is to use parameters that change value during the learning-process. It might be a good idea, for example, to change the value of the variance of the noise added to the bids. Much noise in the beginning, little noise in the end. Other parameters, especially the ones that determine the behaviour of the genetic algorithm are

good candidates for modification during the learning-process.

Then the last open research topic that we will discuss in these conclusions are the possible extensions to classifier systems. We have defined unmatched-true conditions and passthroughs in the formal definitions in chapter 5, but we have not implemented these. These specific extensions are thought to be able to extend the possible tasks a classifier system can learn even further (although this has not been proven yet). It is therefore very necessary to implement these extensions. Unmatched-true conditions *have* been implemented in the program used for this thesis, but have not been used in the experiments.

We can conclude that classifier systems are a very interesting computational concept and a rather powerful machine-learning paradigm. There are still a lot of things that are unclear and need to be researched. Therefore classifier systems are a very rich subject of research, but very much a *Terra Incognita* that has a lot of pitfalls, but that also has great promise.

Appendix A

Symbols used in this Thesis

A lot of symbols have been used in different contexts in this thesis, especially in the theoretic parts. In this appendix we will give a list of these symbols, with their meanings and the pages on which they are defined.

A The action of a classifier. (p. 22)

α Symbol used for the "offset" in the specificity function. Causes a non-zero bid for classifiers consisting of all wildcards. (p. 25)

β Symbol used for the bidconstant. This is the fraction of the strength (after correction for specificity) of a classifier that is used as a bid. (p. 24)

B_{ss} The bid of a classifier in its steady state. (p. 30)

c Symbol used for a single classifier in the classifier system. Used throughout chapter 5.

C The string-part of a condition. (p. 22)

$\mathcal{C}$ Symbol used for the set of classifiers in a classifier system. (p. 21)

Γ The set of conditions of a classifier. (p. 21)

$\delta_{\mathcal{S}}$ The defining length of a schema. (p. 10)

θ The string part of a message. (p. 22)

f_{spec} The specificity-function. Causes the bid of a specialized classifier to be higher than that of a general one. (p. 24)

F_B The function that determines the message list a classifier system produces. (p. 22)

f_g The function that selects the active classifiers that are allowed to place their messages. (p. 25)

$\mathrm{f_m}$ The single character matching-function for messages and actions. (p. 22)

$\mathrm{F_m}$ The function that matches complete messages with complete conditions. (p. 23)

$\mathrm{f_o}$ The single-character transformation function from message, condition and action to message. (p. 23)

$\mathrm{F_o}$ The function that transfers complete messages, conditions and actions into a message. (p. 23)

F_Υ The function that determines the output of a classifier system. (p. 22)

f_z The function that selects the active classifier that is allowed to produce output. (p. 25)

k The number of conditions in a classifier. (p. 22)

l The length of a message (p. 22) or the length of a chromosome. (p. 10)

L The total number of symbols in the classifier's conditions. (p. 25)

M The number of messages in the message list. (p. 21)

N The number of classifiers in the system (p. 22) or the number of chromosomes in a population. (p. 11)

S The strength of a classifier. (p. 22)

$\mathcal{S}$ A schema. (p. 10)

S_{ss} The strength of a classifier in its steady state. (p. 30)

σ Standard deviation of the noise added to the bids of the classifiers. (p. 25)

τ_{bid} The bidtaxconstant. This is the fraction with which the strength of a classifier is decreased when it places a message on the message-list or when it produces output. (p. 26)

τ_{life} The lifetaxconstant. This is the fraction with which the strength of a classifier is decreased in every cycle of the classifier system. (p. 26)

Υ The output of a classifier. (p. 22)

ϕ The classifier-part of a message. (p. 22)

Ξ A class of classifier systems. (p. 40)

χ A classifier system. (p. 21)

W Total number of wildcards in the conditions of a classifier (p. 25)

ζ The constant factor with which the number of classifiers in two classifier systems from two different, but equivalent classes of classifier systems can maximally increase.

Appendix B

The Program Used for the Experiments

In this thesis a number of experiments have been done. These were performed by a program written as part of the research. This program was written in the programming language C++ in a modular way and was adapted several times during the research. The compiler used was the gnu C++ compiler. We will now attempt a description of the code so that it can be used for further research. The code is to be made available through ftp.

B.1 The modules

The code has been written in a modular way as to facilitate the debugging and modification of the code. The program has been split in three modules. The first module is a support module, consisting of the files `initialize.C`, `initialize.h`, `random.C`, `random.h`, `standards.h` and `main.C`. In these files the initialization of the parameters in the system is performed (`initialize.h` and `initialize.C`), some useful functions for generating random numbers (`random.h` and `random.C`)are implemented and the standard values for the parameters of the system (`standards.h`) are defined. In `main.C` the initialization-file is read, classifier system routines are called, the results are written to disk and the number of classifier system cycles, genetic steps and number of trials over which the average is taken are determined. The file `main.C` has to be changed sometimes to accommodate different kinds of experiments.

The next module is the module implementing the learning system. This consists of the files `conversion.h`, `conversion.C`, `genetic.h`, `genetic.C`, `classsyst.h`, `classsyst.C`, `list.h` and `list.C`.

In `genetic.h` and `genetic.C` the functions that perform the genetic algorithm are implemented. In `classsyst.h` and `classsyst.C` the classifier system

is implemented. A classifier system cannot be used directly in the genetic algorithm, so we need a conversion-function between classifiers and chromosomes and vice versa. The functions for this are implemented in `conversion.h` and `conversion.C`. Furthermore we need some functions implementing a message list. These are implemented in `list.h` and `list.C`.

The last module consists only of the files `world.h` and `world.C`. In these files the task to be learnt must be implemented. As there have been experiments with different tasks, there are different versions of `world.C`. The one for the multiplexer is called `multworld.C`, the one for the mushrooms is called `mushworld.C` and the one for the parity-problem is called `parworld.C`. One of these should be renamed into `world.C` when compiling the system.

The compilation itself can be performed with the accompanying `makefile`. It is recommended to use this to make the project, as there are a lot of rather intricate interdependencies between the files in the system.

B.1.1 The Support-Module

In this section and the following two sections, the inner workings of the different modules is explained a bit further.

We will first discuss the details of `main.C` and `initialize.C`, the main support functions. The file `main.C` contains two functions: `main` and `cycle`. Of these `cycle` is the one in which the most interesting changes can (and in fact were) made. One of these is whether a classifier system is initialized randomly at every trial, or if it is initialized by a standard classifier system read from disk. This can be done by changing the initial assignment (the statements between the `for`-loop for the number of trials and the one for the number of generations). It should not be too difficult to find out how to do this.

In `initialize.C` the values of all the different parameters of the system are read in. The input file should contain lines of the following form:

```
<variable> value
```

The possible variables, their meaning and their possible values are listed below:

`alpha` The offset for the specificity function. Should be a small, positive real number. Default initialization by `StandardAlpha`.

`badreward` The reward that an ill-responding classifier gets. Should be a positive real number. Default initialization by `StandardBadReward`.

`bidconstant` The fraction of the strength that is used in the bidding. Should be a real number between 0 and 1. Default initialization by `StandardBidConstant`.

`bidsigma` Standard deviation of the noise added to the bid of a classifier in order to get a non-deterministic effect. Should be a small real number. Default initialization by `StandardBidSigma`.

`bidtaxconstant` The fraction of the strength of a classifier that the classifier has to pay when placing a message or an output. Should be a real number between 0 and 1. Default initialization by `StandardBidTaxConstant`.

`chromosomelength` The length of the chromosomes in the genetic population. Should be a positive integer. Default initialization by `StandardChromosomeLength`.

`classifiersystemsize` The number of classifier in the classifier system. Should be a positive integer and equal to `populationsize`. Initialization by `StandardSystemSize`.

`crossoverprobability` The probability of crossover between two mating chromosomes. Should be a real number between 0 and 1. Default initialization by `StandardCrossoverProb`.

`crowdingfactor` The number of subpopulations from which the worst chromosomes are compared with a new one, used in the crowding-replacement algorithm. Should be a positive integer. Default initialization by `StandardCrowdingFactor`.

`crowdingsubpop` The number of subpopulations in crowding. Should be a positive integer. Default initialization by `StandardCrowdingSubpop`.

`datapoints` The number of datapoints in the output-datafile. Should be a positive integer. Default initialization by `StandardDataPoints`.

`gen2gen` The number of classifier system cycles between genetic search steps. Should be a positive integer. Default initialization by `StandardGen2Gen`.

`goodreward` The reward a classifier gets when providing a good response to the environment. Should be a positive real number. Default initialization by `StandardGoodReward`.

`initialstrength` The initial strength of a freshly initialized classifier. Should be a real number. Default initialization by `StandardInitialStrength`.

`lifetaxconstant` The part of its strength a classifier has to pay at every cycle of the classifier system. Should be a real number between zero and one. Default initialization by `StandardLifeTaxConstant`.

`listlength` The maximum length of the message-list. Should be a positive integer. Default initialization by `StandardListLength`.

`maxcycles` The maximum number a classifier system is allowed to cycle between input and output. Should be a positive integer. Default initialization by `StandardMaxCycles`.

`maxgen` The maximum number of genetic search steps. Should be a positive integer. default initialization by `StandardMaxGen`].

`messagelength` The length of messages, conditions and actions in the classifiers. Should be a positive integer. Default initialization by `MessageLength`.

`mutationprobability` The probability of the mutation of a gene in a chromosome. Should be a real number between zero an one. Initialization by `StandardMutationProb`.

`nomessagelist` Indicates whether the system will use a message-list. Should be zero if it does and one if it doesn't use a message-list. Default initialization 0.

`numberofconditions` The number of conditions in a classifier. Should be a positive integer. Default initialization by `StandardNumber`.

`outputprobability` The probability of initializing a classifier as producing output. Should be a real number between zero and one. Initialization by `StandardOutputProb`.

`populationsize` The size of the genetic population. Should be a positive integer and equal to `classifiersystemsize`. Default initialization by `StandardSize`.

`replaced` Number of chromosomes that is replaced in every genetic step. Should be a positive integer. Default initialization by `StandardReplaced`.

`steps` The number of times the system is going to repeat a certain experiment (or: the number of trials). Should be a positive integer. Default initialization by `StandardSteps`.

`unmatchedtrueprob` The probability of a condition being unmatched true after initialization. Should be a real number between zero and one. Default initialization by `StandardUnmatchedTrueProb`.

`wildcardprobability` The probability of wildcards occurring in the initialization of a condition. Should be a real number between zero and one. Default initialization by `StandardWildCardProb`.

`worstbestratio` The ratio between the worst and the best fitness of the chromosomes that are the result of converting a classifier system into a population of chromosomes. Should be a positive real number. Default initialization by `StandardWorstBestRatio.`

B.1.2 The Learning-Module

In the learning-module, the main function is `int classsyst::Cycle( message, output&)` . This function takes an object of type `message` generated by the environment as input, and generates an object of type `output` as output. It also returns an integer. If this integer is true (or one), then the classifier system was able to find an answer (be it right or wrong) to the problem. If the integer has the value of false (or zero), then the classifier system was not able to find an answer.

The classifier system can be rewarded by the function `void classsyst:: Reward( double )`. This function rewards the classifier that produced the output. The reward is usually taken from the environment.

For the genetic cycles there is the function `population population:: NewGeneration()` that produces the offspring of a population. But first the classifier system should be converted into a population of chromosomes. That is done by the functions `void Pop2Syst ( classsyst&, const population& )` and `void Syst2Pop ( const classsyst&, population&, int )`. The first function takes as input a population and converts this into a classifier system. The second takes as input a classifier system and converts this into a population. If the integer that is also an input to this function, is one, then the whole system is converted. If the integer is zero, then only the strengths of the chromosomes in the population are updated. Obviously these chromosomes must then correspond to the classifiers in the classifier system.

With these functions one should be able to implement the learning-process of the classifier system. There must be one other component: the environment in which the classifier system is trained. This is implemented in the task-module.

B.1.3 The Task-module

The task-module implements the tasks the classifier system has to learn. It should be called `world.C` and should contain the following functions (whose prototypes can be found in `world.h`): `message world::problem()`, `void answer( int )` and `double payoff()`. The function `problem()` should return a message that can be used as input to the classifier system. The function `answer( int )` should be used to give the answer of the classifier system to the environment. The function `in output2int( const output&)` can be used to convert objects of type `output` to integers.

The function `payoff()` should be used to return the payoff assigned to the previous answer to the classifier system. For this purpose the object `world` contains some extra variables: `int right` to store the correct answers and `double reward` to store the amount of payoff.

These objects and functions make up a complete classifier system that can rather easily be adapted for other tasks and variations on the basic classifier system.

Bibliography

[ALB83] Bruce Alberts, Dennis Bray, Julian Lewis, Martin Raff, Keith Roberts, James D. Watson, *Molecular Biology of the Cell*, Garland Publishing, Inc., New York 1983.

[BAI93] F. Baiardi, A.M. Lomartire, D. Montanari, *A parallel MIMD architecture for asynchronous classifier systems*, available through ftp from: **ftp.aic.nrl.navy.mil** in **/pub/EC/smith**, 1993.

[BGH89] L.B. Booker, D.E. Goldberg and J.H. Holland, *Classifier Systems and Genetic Algorithms*. Artificial Intelligence 40, 1989, pp 235–282.

[BK92] E.J.W. Boers and H. Kuiper, *Biological metaphors and the design of modular artificial neural networks*. Unpublished Master's thesis, Leiden University, Leiden 1992, available through FTP from **ftp.wi.leidenuniv.nl (132.229.128.44)** as **/pub/cs-techreports/thesis/boers-kuiper.92.ps.gz**.

[BOO85] Lashon B. Booker, *Improving the performance of genetic algorithms in classifier systems*, in [GREF85], pp 80–92.

[CMS90] M. Compiani, D. Montanari, R. Serra, *Learning and Bucket Brigade Dynamics in Classifier Systems*, in [FOR90] pp.202–212.

[DAR1859] Charles Darwin, *The origin of species*, 1859.

[DEJ75] K.A. De Jong, *An analysis of the behavior of a class of genetic adaptive systems*, (Doctoral dissertation, University of Michigan), Dissertation abstracts international 36(10), 5140B. (University Microfilm No. 76–9381)

[DOR91] M. Dorigo, *Message Based Bucket Brigade: An Algorithm for the Apportionment of Credit Problem*, in [KOD91], pp235–244.

[FAR90] J. Doyne Farmer *A Rosetta Stone for Connectionism*, in [FOR90], pp. 153–187.

[FOMI90] Stephanie Forrest, John H. Miller, *Emergent behavior in classifier systems,* in [FOR90], pp. 213–227.

[FOR85] Stephanie Forrest *Implementing semantic network structures using the classifier system*, in [GREF85], pp 24–44.

[FOR90] Stephanie Forrest (ed.), *Emergent Computation, proceedings of the Ninth Annual International Conference of the Center for Nonlinear Studies on Self-organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks*, in Physica D, vol 42. North Holland, 1990.

[GB89] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, Reading MS, 1989.

[GHD92] David E. Goldberg, Jeffrey Horn, Kalyanmoy Deb, *What Makes a Problem Hard for a Classifier System?*, available through FTP as **afi.santafe.edu** in **/pub/EC/CFS/papers/lcs92-2.ps.gz**, 1992.

[GREF85] John J. Grefenstette (ed.) *Proceedings of the first international conference on genetic algorithms and their applications*, Lawrence Erlbaum assoc., Hillsdale NJ, 1988.

[HIBO75] Ernest R. Hilgard, Gordon H. Bower, *Theories of Learning, fourth edition*, Century Psychology series, Prentice-Hall, Englewood Cliffs NJ, 1975.

[HKP91] J. Hertz, A. Krogh and R. G. Palmer, *Introduction to the Theory of neural Computation*, Addison Wesley, Reading MS, 1991.

[HOL75] J.H. Holland, *Adaptation in Natural and Artificial systems*, The University of Michigan Press, Ann Arbor, 1975.

[HOL80] J.H. Holland, *Adaptive Algorithms for Discovering and Using General Patterns in Growing Knowledge Bases* . International Journal for Policy Analysis and Information Systems, Vol 4, No 3, 1980. pp 245–268.

[HOL86] J. H. Holland, *Escaping Brittleness, The possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems*, in [MCM86] pp. 593–623.

[IWL88] W. Iba, J. Wogulis, P. Langley, *Trading off implicity and Coverage in Incremental Concept Learning.* In Proceedings of the 5th International Conference on Machine Learning, Ann Arbor, Michigan: Morgan Kaufmann, 1988, pp. 73–79.

[KOD91] Yves Kodratoff (ed.) *LNAI 482 Machine Learning EWSL 91*, Springer Verlag, Berlin, 1991.

[LEE86] W.J. Leech, *A Rule Based Process Control Method with Feedback*, Proceedings of the International Conference and Exhibit, Instrument Society of America, 1986.

[MCM83a] Ryszard S. Michalski, Jaime G. Carbonell, Tom M. Mitchell: *Machine Learning: an artificial intelligence approach*, Tioga publishing company, Palo Alto CA, 1983.

[MCM83b] Ryszard S. Michalski, Jaime G. Carbonell, Tom M. Mitchell, *An overview of machine learning*, in [MCM83a] pp 3–23.

[MCM86] R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: an Artificial intelligence Approach, Volume 2* Morgan Kaufmann, Los Altos CA, 1986.

[MIC92] Zbigniew Michalewicz, *Genetic Algorithms + Datastructures = Evolution Programs*, Springer Verlag, Berlin, 1992.

[MIN67] M. Minsky, *Computation with Finite and Infinite Automata*, Prentice-Hall, Englewood Cliffs NJ, 1967.

[MON93] Daniele Montanari, *Classifier systems with a constant-profile bucket brigade*, available through ftp from: **ftp.aic.nrl.navy.mil** in **/pub/EC/smith**, 1993.

[MUR87] Patrick M. Murphy, David. W. Aha, *Mushroom Database*, UCI Repository of machine learning databases [Machine readable data repository]. University of California, Department of Informaton and Computer Science, Irvine, California. (internet: **ics.uci.edu** (128.195.1.1))

[NT89] Kumpati Narendra, M.A.L. Thathachar *Learning automata, an introduction*, Prentice Hall International, Englewood Cliffs NJ, 1989.

[POM89] D.A. Pomerleau, *ALVINN: An Autonomous Land Vehicle in a Neural Network*, In Advances in Neural Information Processing Systems 1, D.S. Touretzky, ed. Morgan Kaufmann Publishers, San Mateo CA, 1989.

[POTO89] Dean A. Pomerleau, David S. Touretzky, *What's hidden in the Hidden Layers?*, in BYTE, August 1989, Mc Graw-Hill, 1989, pp. 227–233.

[POST43] E.L. Post, *Formal reductions of the general combinatorial decision problem*, American Journal of Mathematics 65, pp 197–268, The John Hopkins Press, Baltimore, 1943.

[SCH87] Jeffrey S. Schlimmer, *Concept Acquisition Through Representational Adjustment*, Technical Report 87-19, Doctoral dissertation, department of Information and Computer Science, University of california, Irvine, 1987.

[SERO87] T.J. Sejnowski, C.R. Rosenberg, *Parallel Networks that Learn to Pronounce English Text*, Complex Systems 1, pp. 145–168.

[SIM83] Herbert A. Simon, *Why should machines learn?*, in [MCM83a] pp 25–37.

[SIMP89] Patrick K. Simpson, *Artificial Neural Systems*, Foundations, Paradigms, Applications, and Implementations, Pergamon Press, New York, 1989.

[SMI92] Robert. E. Smith, *A Report on the First International Workshop on Learning Classifier Systems*, available through FTP as **sfi.santafe.edu**:/pub/EC/CFS/papers/lcs92.ps.gz, 1992.

[SMIBO93] Robert E. Smith, H. Brown Cribbs III, *Is a Learning Classifier System a Type of Neural Network?*, available through ftp from: **ftp.aic.nrl.navy.mil** in **/pub/EC/smith**, 1993.

[SUTY91] Joseph W. Sullivan, Sherman W. Tyler, *Intelligent User Interfaces*, ACM Press frontier series, Addison Wesley, Englewood Cliffs MS, 1991.

[THOR92] C.J. Thornton, *Techniques in Computational Learning, an introduction*, Chapman & Hall, London 1992.

[VAL93] Manuel Valenzuela Rendón, *Reinforcement Learning in the Fuzzy Classi-*
fier System, available through ftp from: **ftp.aic.nrl.navy.mil** in **/pub/EC/smith**, 1993.

[WES85] Thomas H. Westerdale, *The bucket brigade is not genetic*, in [GREF85] pp 45–59.

[WIN92] Patrick Henry Winston *Artificial Intelligence, third edition* Addison Wesley, Reading MS, 1992.

[ZHOU85] Hayong Zhou, *Classifier systems with long term memory*, in [GREF85] pp 178–182.