Elegance: Genetic Algorithms in Neural Reinforcement Control

Pieter Spronck

Graduation committee:
Prof. dr. H. Koppelaar
Prof. dr. ir. E.J.H. Kerckhoffs
Dr. drs. L.J.M. Rothkrantz

Delft University of Technology
Faculty of Technical Mathematics and Informatics
Delft, 1996

Spronck, Pieter.
Elegance: Genetic Algorithms in Neural Reinforcement Control.

Graduation thesis (Master's degree).
Delft University of Technology, The Netherlands.
Faculty of Technical Mathematics and Informatics.

First version August 1996.
Second version September 1996.
Third version December 1996.

Keywords: adaptive control, artificial intelligence, genetic algorithms, neural control, neural networks, non-linear systems, reinforcement control, software tools.

This is the third version of this document. The second version was almost the same as the first version, except for a few very small textual changes and the fact that the word "Neural" had been added to the title. The third version again contains a few minor textual changes, but also appendix D has been changed significantly, using the input of professor Kerckhoffs, who is now mentioned as co-author of the paper presented in this appendix.

Note: Formula 18 does not accurately represent the Elegance fitness calculation anymore, although the actual fitness calculation can be configured to adhere to this formula. At the moment, the fitness calculation has a bit more flexibility where the premature failure penalty is concerned, taking into account the time of failing. See the Elegance helpfile for a more detailed explanation.

© Copyright 1996 by Pieter Spronck. All rights reserved.

Abstract

Genetic algorithms are search algorithms based on the principles of natural selection and natural genetics, which are particularly suited for optimisation problems. Genetic algorithms work with a set of potential solutions to a problem, which all have been awarded a fitness rating, which is a measure of their relative success in solving the problem at hand. By using genetic operators and the fitness ratings, new potential solutions are created from the existing ones, in such a way that chances are that the features which make the better ones work, are passed on to their offspring.

A neurocontroller is a neural network which is used to control a plant. In plant control, a lot of research has been aimed at so-called model-based control, wherein first a model of the plant is built, which is then used to construct a controller. However, it is not always possible to build a plant model. In that case, an alternative for model-based control is reinforcement control. Reinforcement control evaluates the performance of a controller in a practice control situation, and uses that evaluation to adapt the controller in order to make it more successful. Because genetic algorithms need nothing more than a measure of success of a controller to produce better controllers, they seem especially suited for handling reinforcement control situations.

This thesis deals with the use of genetic algorithms in the design of controllers, particularly neurocontrollers, in a reinforcement control environment. This is called *genetic reinforcement control*. Although a lot of research has gone in the use of genetic algorithms to design neural networks, very little of that research has been aimed at neurocontrol and almost nothing at reinforcement control. For the research in genetic reinforcement control to be successful, many experiments have to be performed. For this purpose a flexible software environment has been built, in which a great many different genetic algorithm configurations can be tested on the design of a neurocontroller for several different plant simulations.

The preliminary experiments done with this environment show that genetic reinforcement control is certainly a viable technique, which is at least competitive with conventional methods used in this respect. They also show that the environment has the ability to test new ideas and to reach conclusions about what makes a genetic algorithm suitable for neurocontroller design in a reinforcement situation.

Contents

Abstract **3**

Contents **5**

Preface **9**

Part I An Overview of Genetic Algorithms **13**

1 Theory of Genetic Algorithms **15**
1.1 Biological evolution 15
1.2 The genetic algorithm 16
1.3 The simple genetic algorithm 22
1.4 GAs and conventional search algorithms 30
1.5 Schemata 31
1.6 Making GAs work 33
1.7 Parallel GAs 39
1.8 Designing a GA 41
1.9 Summary 42

2 Genetic Algorithm Applications **45**
2.1 Function optimisation 45
2.2 Classifier systems 48
2.3 Genetic programming 51
2.4 Hybrid systems 54
2.5 Gaspipe networks 55
2.6 Business modelling 57
2.7 Neural network architectures 59
2.8 Other applications 60
2.9 Tools 61
2.10 Summary 63

Conclusion of Part I 65

Part II Genetic Algorithms and Neural Networks in Control 69

3 GAs and Neural Networks 71
3.1 Artificial neural networks 71
3.2 Areas of GA application in ANN design 75
3.3 Competing conventions 77
3.4 The design of a GA for ANN evolution 80
3.5 Evolution of ANNs in practice 86
3.6 Summary 90

4 GAs and Neurocontroller Design 93
4.1 Plant control 93
4.2 AI in plant control 95
4.3 Neurocontrol 96
4.4 GAs in neurocontrol 100
4.5 An experiment with GAs in neurocontrol 102
4.6 Summary 104

Conclusion of Part II 105

Part III Elegance 107

5 The Design of Elegance 109
5.1 Purpose 109
5.2 Functional design 111
5.3 Technical design 113
5.4 The genetic algorithm 122
5.5 Implementation 127
5.6 Using the program 129
5.7 Evaluation 133
5.8 Future versions 134
5.9 Summary 134

6 Elegance Experiments 137
6.1 Preparing the experiments 137
6.2 A pole balancing controller 138
6.3 A trolley controller 145
6.4 A bioreactor controller 151
6.5 Conclusions and further work 161
6.6 Summary 163

Conclusion of Part III 165

Summary 169

Appendices 171

A Elegance user manual 173

B Plants 181
B.1 SISO system 181
B.2 DIDO system 181

B.3 Titration 182
B.4 Trolley 182
B.5 Trolley 2-dimensional 183
B.6 Pole balancing system 184
B.7 Bang-bang pole balancing system 185
B.8 Bioreactor 186

C Extending Elegance 187
C.1 Adding a plant 187
C.2 Adding a genetic operator 189
C.3 Other extensions 191

D A paper on Elegance 193

Bibliography 207

Index 213

Preface

Palmiter's Protozoa

Like most sciences, computer science can be fun, and I think it's a pity that, although computers and programming are quite popular nowadays, the recreational *scientific* aspects of computers get very little attention in the media. A refreshing exception is the column *Computer Recreations* by A.K. Dewdney, which appears regularly in Scientific American Magazine.

A few years ago I found in a bookstore a collection of Dewdney's columns, called *The Magic Machine* (Dewdney 1990). I've experimented with many of the subjects described in this book. Most fascinating I found the entry called "Palmiter's Protozoa". It describes a program called "Simulated Evolution", devised by Michael Palmiter, a Californian high-school teacher.

"Simulated Evolution" moves bugs on a computer screen. The bugs feed on bacteria, which are placed randomly on the screen and which are constantly replenished. A bug eats those bacteria which it meets on its voyage across the screen. Each bug has a chromosome, which decides how the bug moves. For instance, the chromosome may say that each step the bug takes, it has a 50% chance to continue on its path, a 30% chance it will turn right and a 20% chance it will turn left. If a bug does not find enough food, it dies. If it lives long enough and has food enough, it will split into two new bugs, each slightly different from the original bug. The difference is decided randomly.

When running this program, we find that the bugs with which the program starts are jittering randomly and without purpose across the screen, but that later on bugs tend to travel longer distances before making a turn, thereby seeking out food more efficiently. These more efficient bugs are characterised with the name "cruisers".

A strange thing happens when the pattern of replenishment of bacteria is changed. The program is altered so that a small area of the screen, called the Garden of Eden, gets 90% of all bacteria. Cruisers still evolve, but in the Garden of Eden another kind of bug is found, which runs in tight little circles, thereby staying within the Garden and profiting maximally from the abundance of food. These bugs are characterised with the name "twirlers".

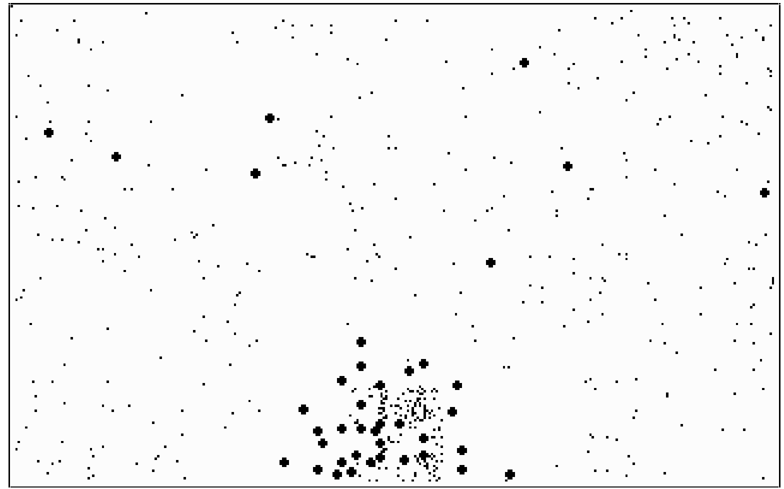


figure 1: Palmiter's Protozoa: big dots are bugs, small dots are bacteria. The Garden of Eden is located at the bottom middle.

The problem presented to “Palmiter’s Protozoa” is to design a movement pattern which is suitable for finding food in some area. The solution is found by just *testing* randomly generated solutions, and by generating new solutions by making small, random changes in those solutions that seem to perform better than others. Thus we find that there are two viable solutions: either cruising along the screen, or twirling in the Garden of Eden.

Although I didn’t know it at the time, this was my first contact with evolutionary systems. The charm of evolutionary systems is found in the fact that surprising solutions may be found efficiently by a process of seemingly random changes, imitating natural evolution. And although very simplistic in nature, “Palmiter’s Protozoa” gives an indication of the power of these systems. The basics of this charming program can be used to tackle complex, real-world problems in business, mathematics, engineering and many other areas.

Genetic algorithms

The genetic algorithm (GA) is one of the most successful implementations of an evolutionary system. GAs have in recent years become a major interest in the field of artificial intelligence (AI). It’s not difficult to find books and essays about them. However, it’s significant that most essays start with a short explanation of what GAs are and how they work, indicating that they are still not common tools for information scientists.

Because of my fascination with this technique, I requested permission to let genetic algorithms be the subject of my graduation thesis. Since as yet there is no regular course on this subject at my university, I started with an exploration of the field by studying literature. The first part of this text, “An Overview of Genetic Algorithms”, is the result of that exploration. It can be seen as a condensed version of a book on GAs. It’s not as introductory as most essays which present GAs, but won’t take as long to study as a book, since the information is presented short and to-the-point, without

sacrificing clarity. There is no information found in this part which is not also available in standard literature.

The overview is roughly divided into two chapters. The first chapter focuses on the theory of GAs. It starts with an introduction to the biological basics of GAs. Then the theory of GAs is explained, using the simple genetic algorithm (SGA) as an example. Some attention is given to the reasons why GAs work, and how they should be applied.

The second chapter is about GA applications. Some generic ways of using GAs, like classifier systems and genetic programming, are discussed, which is followed by some real-world examples of the application of GAs.

These two chapters are based on information which is found in literature. Since in my opinion this information is sometimes over-optimistic about the possibilities of GAs, I've concluded the first part with some of my own insights in the subject matter.

Genetic algorithms and neural networks for control

After I'd written the first part, my graduation co-ordinator, professor Kerckhoffs, suggested I'd focus my studies on the application of GAs in neurocontrol problems. Neurocontrol is about controlling an industrial plant with an artificial neural network (ANN). The design of a neural network suitable for such a task by using GAs would be a worthy subject for a graduation thesis. This was affirmed by my discovery of the fact that literature showed that in 1995 this subject was still very much in the research phase, and that there were little or no concrete results in this field, other than that is seemed to be very promising.

The second part of this text, "Genetic Algorithms and Neural Networks for Control", is the textual result of my literature study of the use of GAs in the design of ANNs, especially for control problems. It starts with a chapter about ANNs, which gives a condensed overview of ANNs and especially about the design of ANNs with GAs. Since ANNs suitable for neurocontrol are a special case, a separate chapter is installed about this subject.

The conclusion I reached from my research, is that the most promising application of GAs in neurocontrol is in the area of reinforcement control. Reinforcement control is a form of control wherein an evaluation of performance of a controller in a practice situation is used to adapt that controller. With reinforcement control there is no access to a model of the process that needs to be controlled. If there is a model available, there are several other successful methods of designing a suitable controller, but with reinforcement control such methods are as yet undocumented, if they exist at all. GAs may help us out here. This application of GAs is called genetic reinforcement control.

Elegance

To make it possible to evaluate the possibilities and the suitability of the application of GAs in the design of neurocontrollers in reinforcement control situations, I decided to finish my graduation project with the design and implementation of a software environment which could be used to run several different experiments with this subject matter. The result is the program "Elegance". A description of the design of this program is found in the third part of this thesis.

Elegance is a very flexible, easily extensible program. As such it is very suitable to run a large number of the desired experiments. Whether or not GAs are indeed a viable approach to reinforcement control problems is still unclear, since in the time allotted for a graduation project I could run only a fraction of them. The first results I reached are found in the last chapter of this text.

There are many more experiments to perform with Elegance, and I think many nice results still to be discovered. It is my hope that the program will be used by others in the coming years, and I've tried to make it suitable for that.

The last appendix of this thesis contains a first draft of a paper about Elegance and genetic reinforcement control, which is written on request. It can be read as a summary of the entire text.

Acknowledgements

I'm grateful to the staff of the TU Delft, who made it possible for me to obtain a master's degree in computer sciences next to having a job, by offering a part-time course in the evenings. I know it requires a lot of effort from many people, to the benefit of only a small number of students. I'm glad they still find it worthwhile.

Many thanks go out to my graduation co-ordinator professor Kerckhoffs of the Artificial Intelligence department of the Technical University of Delft, who has guided me in my exploration of GAs and neurocontrol. His advice and especially his enthusiasm during our discussions really supported and stimulated me to finish my work.

I would like to thank ir. Hans Vogel and ir. Jacek Jarmulak for the help they gave me with my graduation project; one of my fellow students, drs. Tom Dokoupil, who was a very reliable partner in many practical assignments during my studies and from whom I've learned a great deal; and my current employer, Generale Bank Nederland, and previous employer, Coss Holland BV, for the financial support of my studies.

My parents paid my way through college the first time I tried. I didn't succeed then, although I could use the knowledge I obtained at that time on my second attempt. However, I suppose their greatest contribution to my career and my studies is the home computer they gave me more than a decade ago. I still have it today. It is never turned on anymore, but it runs the first programs I wrote in which I expressed my creativity, and as such it is very dear to me. I would like to thank my father and mother for their generosity, and for all the support they gave me.

The one person who has invested almost as much in my studies as myself, is my girlfriend Muriël. I haven't always realised how difficult these past years have been for her. Now it's over, I expect there are better times ahead, and I'm grateful for her tremendous patience.

Pieter Spronck
Capelle a/d IJssel
August 1996

Part I

An Overview of Genetic Algorithms

"Whence arises all that order and beauty we see in the world?"
--- Isaac Newton

"In the beginning, there were no reasons; there were only causes."
--- Daniel C. Dennett, *Consciousness Explained*

1 Theory of Genetic Algorithms

Genetic algorithms (GAs) are search algorithms based on the principles of natural selection and natural genetics. This chapter is about how GAs work and how they should be applied. After a short introduction on the history of natural genetics (1.1), the basic operations of GAs are explained (1.2). To get insight in the GA process the simple genetic algorithm (SGA) is used to solve an optimisation problem (1.3). A comparison between GAs and conventional search algorithms is made (1.4), followed by an explanation of the Schema Theorem, which gives an indication of why GAs work (1.5). Some alternatives for the basic genetic processes are given (1.6), it is explained how to take advantage of parallel computers when using GAs (1.7), and the chapter ends with some tips and insights in how GAs can be applied successfully (1.8).

1.1 Biological evolution

For centuries the most convincing argument for the biblical explanation of the origin of life, was the fact that living beings exist in such diversity and such complexity that it seems inconceivable that they developed from nothing by mere chance. Divine intervention was deemed necessary to explain the fact that not only life exists at all, but especially that living beings in all their complexity are perfectly tuned to their surroundings and to each other.

Around the year 1850 Gregor Mendel developed his theory of genes. Genes are bits of hereditary information found in living beings. Every gene describes a small aspect of the being it is part of. The complete collection of genes in a being describes that entire being. This theory is known as the genetic theory or simply *genetics*.

Genes are coded in a collection of very complex molecules which are called *DNA* or *chromosomes*. Each and every cell in the body of an organism contains a copy of all the different chromosomes that organism has. So, each cell contains a complete description of the whole organism.

During the development of an organism, the DNA is used as a blueprint for the production of new cells with a specific purpose. When a new organism is created, its

DNA is constructed by combining parts of the DNA of its parents. The characteristics of this new organism are therefore determined by its parents' DNA.

In 1859 Charles Darwin published his *Origin of Species*. In this work he explains his theory of evolution, which describes how the emergence of life as we know it can be explained as a natural phenomenon.

Darwin's theory removed the necessity of divine intervention for the creation of life. Naturally he found many opponents on his path, not least of which was the Church. Mendel's theory of genes would have been a great help to Darwin in support of his theory, but sadly he never knew about it. Nowadays, there are still many opponents of Darwin's theory, most of them for reasons of a religious nature. In scientific circles, however, the basics of his theory are widely accepted.

A very important aspect of Darwin's theory is that an organism that is well fitted for survival has a greater chance of perpetuating its characteristics in future generations than an organism that is unfit for survival. Therefore, the traits that make an organism suitable for survival are likely to be passed on to future generations, while traits that constitute an inferior organism are likely to fade away because there is less chance that the organism will procreate. This mechanism is called "survival of the fittest" or, in Darwin's own words, *natural selection*.

In this way, there is a rapid development of new beings which are better adapted to their surroundings than their parents or grand-parents. Also, the mechanism enables organisms to adapt to changes in the environment (Dawkins 1986, Dawkins 1989).

1.2 The genetic algorithm

John Holland is a psychologist and computer scientist, who is the founder of a branch in computer science which is known as *complex adaptive systems*. In his book *Adaptation in Natural and Artificial Systems* (Holland 1992), the first edition of which was published in 1975, he describes an adaptive system as a system that changes constantly to make better use of its environment, and he develops a general theory of adaptive systems. Among other things he describes how *genetic operators* (operators which change the basic coding of a system) may be used to get a system into a desired state.

Holland's book has been the basis for a number of practical developments in the area of adaptive systems. One of the most important of these, for which Holland himself is largely responsible, is the *genetic algorithm* (GA). GAs are search algorithms based on the mechanics of natural selection and natural genetics. GAs are most commonly used for optimisation problems which are difficult to analyse. Since most problems can be formulated as an optimisation problem, GAs can be used with a wide range of problems.

Biological genetics

Many of the terms used with GAs are derived from biological phenomena. Some understanding of the meaning of these biological terms may help understanding these terms when used with GAs. Therefore, I'll first explain a bit in biological terms. I'll do this in a highly simplified form, as it is not the focus of this text.

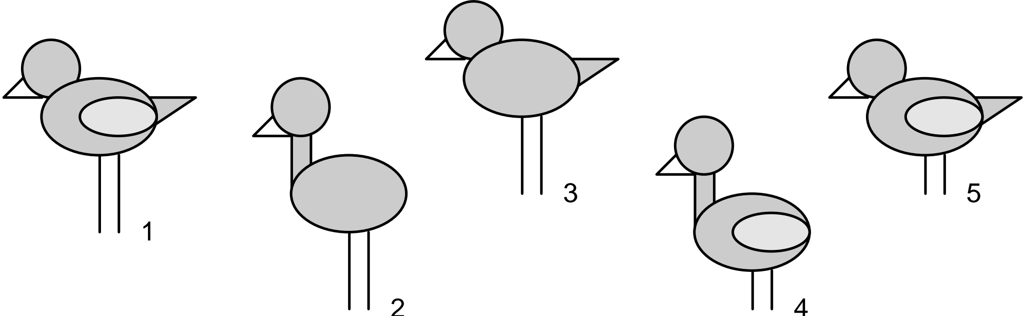


figure 2: A population of birds. We see only the phenotypes here.

A collection of beings belonging to the same race in a specific area is called a *population*. Each of the beings in the population is called an *individual*. Each individual develops according to a code laid down in its *chromosomes*. The chromosomes describe the natural characteristics of the individual. The set of chromosomes is called the individual's *genotype* or *genome*. The way the individual is constructed when it has developed, is called its *phenotype*.

A chromosome consist of bits and pieces of discrete information. Each of these bits of information is called a *gene*. A gene can take on a few different values, and each of the possible values is called an *allele*. The position a gene has on a chromosome is called its *locus*.

Consider the population of birds in figure 2. The birds look different, but they belong to the same race. Instead of saying "they look different", one could also say "their phenotypes differ". The reason their phenotypes differ is mainly because they have different genotypes. I say "mainly", because each bird's history may also account for features of its phenotype. For instance, a bird may lose its tail in a fight. Its phenotype will then no longer show a tail, although its genotype still says the bird should have a tail.

The genotypes, as specified by a bird's chromosomes, tell us what the bird should look like. In a population, the chromosomes of all birds have the same genes. This means, that if one bird has a gene that decides if it has long or short legs, all birds have this gene. The gene may take on other allele values with different birds. For instance, the gene for leg length may take on an allele value for long legs or an allele value for short legs. If we look at the population in figure 2, birds 4 and 5 seem to have the short legs allele, while the other birds have the long legs allele.

In our population of birds, we recognise a gene for having a long neck or not, for having long legs or not, for having a tail or not, and for having wings or not. Suppose there is one chromosome, call it chromosome *C*, in the set of chromosomes of a bird which contains just these genes. This chromosome is shown in figure 3.

Suppose each of these genes may take on the value *Y* or the value *N*. If the value is

Long neck	Long legs	Tail	Wings

figure 3: Chromosome *C*. It has four genes.

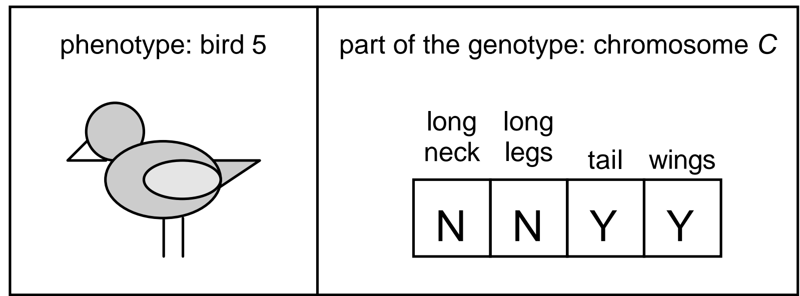


figure 4: The phenotype and part of the corresponding genotype of bird 5.

Y, the corresponding feature is activated for the bird, and if *N*, it isn't. We can now state, for instance, that "the gene at locus 1 on chromosome *C* determines the neck size; *Y* is the long neck allele and *N* is the short neck allele." One of the birds, bird number 5 in the population, with its chromosome *C* is shown in figure 4. Summing up the bird in figure 4 in a table gives table 1.

The phenotype of a bird determines its chance for survival. If a bird lacks wings, it can't fly, and so it will be an easy prey. It seems wings are a boon to the bird's chance of survival. A tail helps the bird balancing, so a tail is also a positive feature. Short legs seem to be less breakable than long legs, especially during landing, so we may add short legs to our list of survival promoting features. Let's also suppose a long neck helps the bird to survive, since it helps the bird to get food from places that are difficult to reach. All these features together determine a bird's *survival value*. The bird with the highest survival value would have a long neck, short legs, a tail and wings, or, its chromosome *C* should be *YNYY*. This bird is not in the population in figure 2.

From this point on, I will use terms and express ideas which are only in a very remote way linked with biological reality. If we would like to rate the survival value of a bird with a number (something a biologist would find distasteful, I think), we could, for instance, give the bird a point for each allele it has in common with the ideal bird. We add the points, and call this number the bird's fitness ratio, or just *fitness* for short. This means the ideal bird has a fitness of 4. In figure 5 the same population as in figure 2 is shown, but now chromosome *C* and the fitness ratio of each bird is also shown.

If we let nature have its way, the birds with the lowest fitness values won't survive very long, and therefore will have less chance to procreate. Fitter birds will dominate the group of procreating birds. This is what is meant by "the survival of the fittest".

When two birds produce a child, its alleles will be a mixture of the alleles of the parents. This mixture will be constructed by some *genetic operators*. A genetic

gene	locus	allele	phenotype
long neck	1	N	the bird has a short neck
long legs	2	N	the bird has short legs
tail	3	Y	the bird has a tail
wings	4	Y	the bird has wings

table 1: Description of bird 5 in genetic terms.

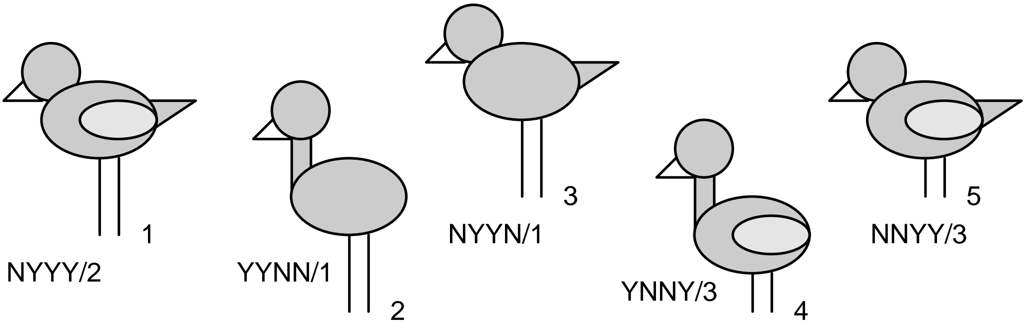


figure 5: The same population of birds, now with genotype and fitness values.

operator uses the chromosomes of both parents to construct a new genotype for the child. The child's genotype contains features of the genotypes of both the parents. Since chances are that only the fittest birds procreate, we may expect a new generation of birds to be at least as fit than the previous generation, and generally more so.

Suppose bird 4 and 5, which are the fittest and therefore most likely to survive, mate. Since the parents both have short legs and have wings, or rather, since the parents both have the allele for short legs and the allele for wings, the child will almost certainly also have these features. So the child will have at least a fitness of 2. This means that because its parents' genotypes are quite good, the chances are that the genotype of their offspring will also be quite good. The possibilities for the new bird are summed up in table 2.

If every possible combination of alleles is as likely to arise as every other combination, we have a 50% chance that the new bird is as fit as each of the parents, a 25% chance that the bird is less fit than each of the parents, and a 25% chance that the bird is more fit than each of the parents. Note that these parents may produce the ideal bird as a child.

Although the average fitness of the children will be the same as the average fitness of the parents, the procreating part of this next generation will have a higher average fitness than the procreating part of the present one, because only the fitter children will procreate. Therefore later generations will have a higher average fitness than earlier generations.

Genetic algorithms

The basis for a GA is a collection of alternative solutions for a problem. This collection

chromosome C	origin	fitness
YNNY	neck from bird 4, tail from bird 4	3
YNYY	neck from bird 4, tail from bird 5	4
NNNY	neck from bird 5, tail from bird 4	2
NNYY	neck from bird 5, tail from bird 5	3

table 2: the possibilities for a child from bird 4 and bird 5.

is called a *population*, and each alternative solution is called an *individual*. In most cases the population has a fixed size.

An individual is represented by a structure, which is a collection of *strings*. This structure is called the *genotype* of the individual, or the *genome*, while each string in the genotype is called a *chromosome*. Most commonly the genotype consists of only one string, which makes the chromosome equal to the genotype. This is what we will assume if not explicitly stated otherwise. So, each individual is represented by one chromosome, which describes the individual in coded form. In most cases a chromosome is of a fixed length.

If we decode the chromosome, we get the alternative solution the chromosome

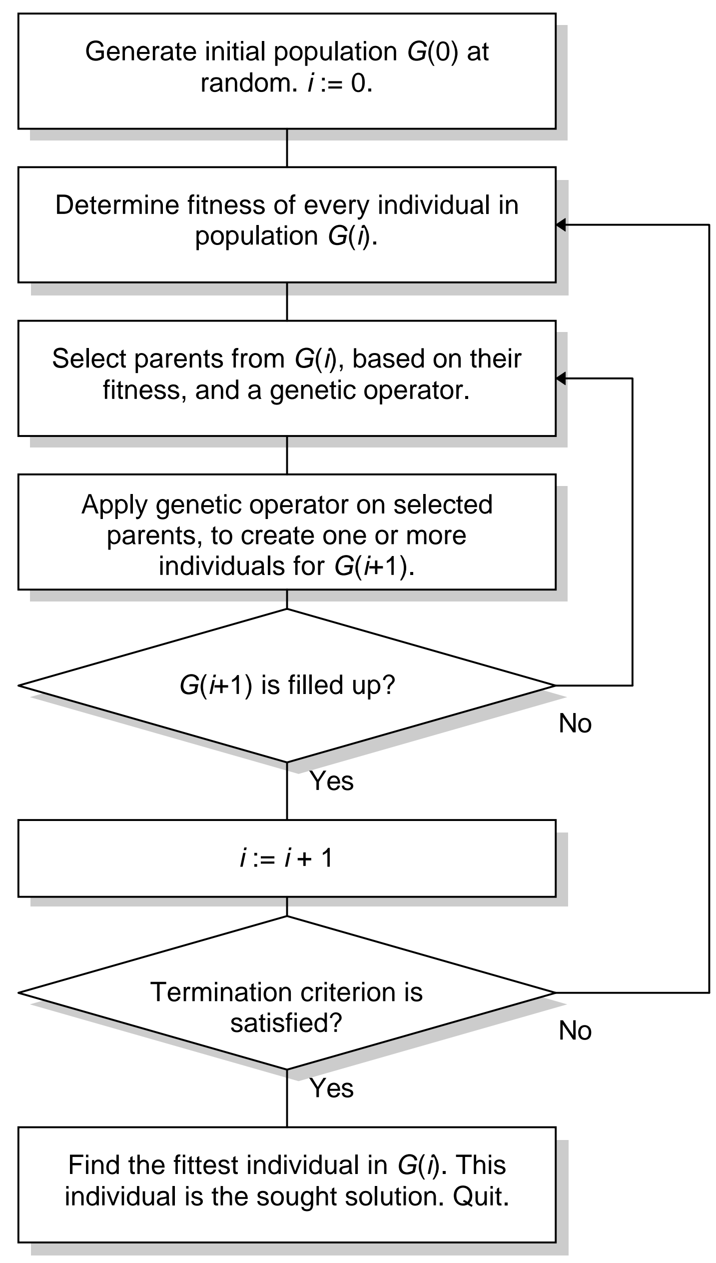


figure 6: The basic Genetic Algorithm.

represents, which is called the *phenotype*. The collection of all possible alternative solutions is called the *solution space*.

A chromosome consists of characters. Each character has a specific meaning, which is connected to its position in the string. The characters are called *genes*. A position in the string is called the *locus*, so the locus of a gene determines what a gene means. If we talk about a gene with a specific locus and a specific value, we use the term *allele*.

Added to each individual is a value which indicates how well the individual "performs" in relation to other individuals in the population. This value is called the *fitness* of the individual. The higher the fitness, the better an individual performs. The fitness is determined by evaluating a *fitness function* which works on an individual's phenotype.

Based on the fitness of the individuals in the population and some *genetic operators* a new population is created from the existing one, which is called the next *generation*. The genetic operators aim towards a generation with a higher average fitness than the last generation while also creating some new individuals. Since the average fitness rises, every next generation contains "better" individuals than the generation before.

The GA continues to create new generations until some termination criterion has been fulfilled. Most commonly this criterion is a maximum number of generations. When the process has terminated, the individual with the highest fitness in the last generation is taken as the best approximation of the required solution.

Figure 6 gives a general schema, often encountered in standard literature, of how a GA works. An example of a complete GA-run is presented in paragraph 1.3.

Genetic operators

Three commonly used genetic operators are *reproduction*, *crossover* and *mutation*.

Reproduction is the artificial equivalent of the principle of survival of the fittest. With this operator an individual from the current generation is simply copied to the next generation. The selection process is based on the fitness of the individual, that is, an individual with high fitness has a greater chance to be selected than an individual with low fitness. Reproduction makes sure individuals with high fitness stay in the population, while those with lower fitnesses are removed.

Crossover is the artificial equivalent of sexual reproduction. Two different individuals are selected from the last generation, again in accordance to their fitness as with reproduction. These two individuals are the *parents* of two new individuals in the next generation. A *crossover point* is selected at random on the chromosomes. Each

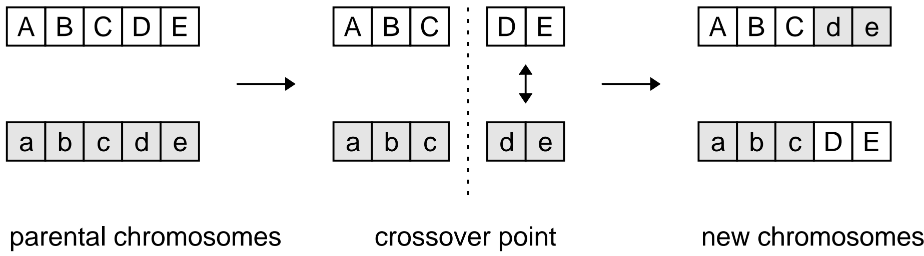


figure 7: The crossover operator.

parent chromosome is cut in two pieces at this crossover point. Then the first part of the first chromosome is added to the second part of the second chromosome, to construct one of the new individuals. The remaining parts are also added together to make the second new individual. If we need just one new individual, we can select one of the two new ones at random.

For instance, if the parent chromosomes are “ABCDE” and “abcde”, and the crossover point is “between the third and fourth gene”, the new chromosomes are “ABCde” and “abcDE”. This is shown graphically in figure 7. Crossover creates fresh, never-before-seen individuals for the next generation, while trying to give these individuals characteristics needed for high fitness.

Note that, although crossover is seen as the GA equivalent of sexual reproduction, there is no concept of gender in this operator. Also, sexual reproduction in nature has a very different way of combining characteristics from both parents.

Mutation is the artificial equivalent of biological mutation. Every allele of every chromosome in the next generation has a slim chance to get mutated, that is, to get changed into a random new value. Mutation gives alleles which no longer occur in a population a chance to return in a future generation.

1.3 The simple genetic algorithm

The most basic GA found in literature has a fixed population size, a fixed chromosome length, a fixed number of generations, and uses only reproduction, crossover and mutation to create new generations. This GA is known as the *simple genetic algorithm* (SGA).

The SGA is not a very useful form of the GA. This is largely due to the fact that the basic forms of reproduction and crossover suffer from a number of weaknesses. For instance, the SGA often converges in just a few generations to a number of chromosomes which are very much alike, and so it searches just a small portion of the solution space. Diversity in the population is crucial for a GA to work reliably.

However, the SGA is a splendid example to demonstrate how a GA works, which is what we’ll do now. The program used to create it is based on a program described in Goldberg’s book *Genetic Algorithms in Search, Optimization & Machine Learning* (Goldberg 1989).

It should be noted that Goldberg’s program is just one way of implementing the SGA. The characteristics of the SGA are formulated vaguely enough to be open for interpretation.

The function

The function we’re going to maximise is:

$$(1) \qquad f(x) = \left(4\sqrt{x} - x\right)^4 \qquad \text{for} \qquad x \in [0,16]$$

$f(x)$ has positive values everywhere on its domain. It is unimodal (has only one

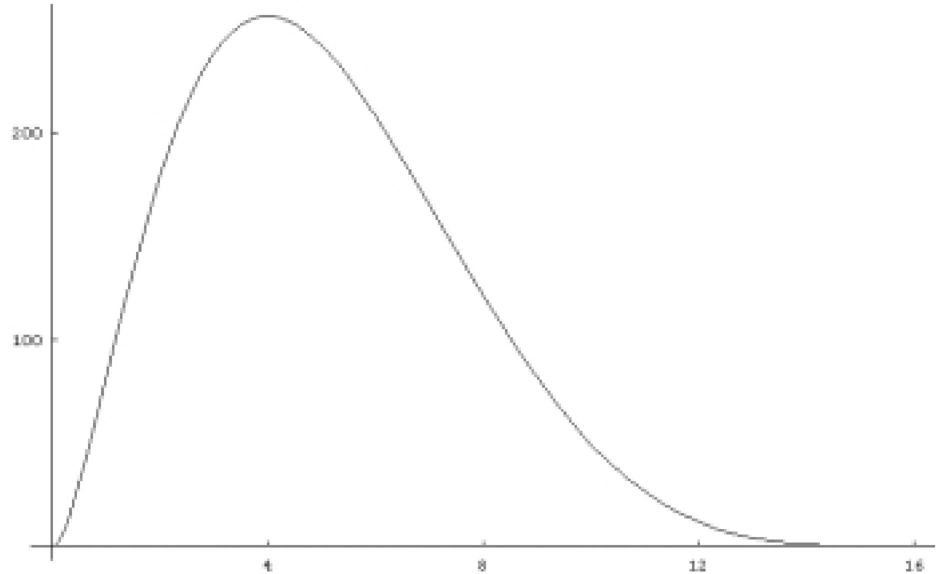


figure 8: Function *f(x)*.

optimum) and continuous. $f(x)$ finds its maximum for $x = 4$, where it is 256. $f(x)$ is displayed in figure 8.

The coding

Since we're going to find the maximum of $f(x)$ with the SGA, we need a coding to represent the different values of x. We choose a binary notation, so every gene can only get the values 0 and 1, or, the string *alphabet* is {0, 1}. We don't want to be restricted to whole numbers, but because of the fixed length of the chromosome we are not able to express real numbers. We are, however, capable of expressing fractions.

The chromosome length is L. In whole numbers and in binary notation, the chromosome can express all numbers from 0 to 2^L-1. This translation of the chromosome to a whole number gives a number, which we call B (for binary). The translation of the chromosome to a fraction on the interval [0,16] is expressed by the formula:

$$x = \frac{16B}{2^L} \qquad (2)$$

For instance, for $L = 6$, x can take on the values 0, ¼, ½, ¾, 1, 1¼, ... , 15½, 15¾. By varying the chromosome length, we can make x as precise as needed.

The parameters

We choose the following parameters for the SGA:

The population size is 6. That's very small for a population. GAs work better on

bigger populations. Populations of 100 or more individuals are normal practice. For this example, however, we like to keep things simple.

The chromosome length is 8. With this length the domain gets divided in 256 fractions.

The number of generations is 8. This is not much, but the solution space is not very big, and even with a small population it gets searched very fast. Besides, a small population converges quite fast, so a large number of generations wouldn't be very useful.

The crossover probability is 0.7. This means, that the chance that a new individual is generated by means of crossover is 0.7. Since the only two ways to create a new individual are crossover and reproduction, consequently the reproduction probability is 0.3.

The mutation probability, also called the *mutation rate*, is 0.001. This means that every allele in a generation has a chance of 0.001 to get mutated. This doesn't seem very high. Too high a mutation rate, however, would affect the GA process adversely. In a fit population a mutation will create, most of the time, individuals which are weak compared to the average member of the population. A high mutation rate would introduce many weak individuals into a fit population and the population would then degenerate. The number of mutated individuals in each population should be small, and therefore a low mutation rate is preferred.

For the fitness function we choose simply the value of $f(x)$, divided by 256 for normalisation. The only reason we're able to normalise in this way is the fact that we already know the maximum of $f(x)$. Normally this won't be the case. There are, however, other ways to normalise a function.

Since $f(x)$ reaches its maximum for $x = 4$, the most fit individual to be found has a chromosome 01000000. Another highly fit but very different chromosome is 00111111, which equals 3.9375 and has a fitness of 0.9998.

The first generation

The first generation is generated at random, that is, every allele of every chromosome in

Generation 1			
Index	Chromosome	Phenotype	Fitness
1	00010101	1.3125	0.4467
2	00001001	0.5625	0.1379
3	10111101	11.8125	0.0548
4	11010111	13.4375	0.0088
5	11010000	13.0000	0.0160
6	00000100	0.2500	0.0366
Minimum fitness:			0.0088
Maximum fitness:			0.4467
Average fitness:			0.1168

table 3: The first generation.

this generation is determined by a coin toss. Because there is a high fitness on a large part of the domain (fitness is at least 0.8 on more than 23 percent of the interval), there will be an individual with high fitness in almost every population generated this way.

Since for this example it is illustrative to see how the maximum fitness increases with the generations, a limitation is placed on the individuals in the first generation. Whenever a chromosome is generated for the first generation, its fitness is tested, and if the fitness is more than 0.5, the chromosome is rejected and a new chromosome is generated. This is, of course, not standard practice for GAs and it is in fact not at all advisable, since it is very well possible that specific combinations of alleles are withheld from the first generation and thus unlikely to appear in future generations.

Not all experiments which start with such a limited first generation and such a small population generate a highly fit individual at the end of the run. We will follow one of the experiments that succeeded. It starts with the generation displayed in table 3.

The second generation

The second generation is generated as follows:

Two different individuals are chosen from the previous generation. This selection is probabilistic, but the individuals in the previous generation do not have an equal chance to get selected. The chance of each individual to get selected is in accordance to its fitness. More specifically, if the sum of all the fitnesses in the previous generation is F_{sum}, and the fitness of an individual i is F_i, the chance of this individual to be selected as the first of the two different individuals is F_i / F_{sum}. This is like a roulette wheel selection with slot sizes according to the fitness of the individuals. So, individual 1 with its fitness of 0.4467 has a chance of 0.4467 / (0.4467 + 0.1379 + ... + 0.0366) = 0.4467 / 0.7008 = 0.6374 to be chosen as the first individual.

The second individual should be different from the first (it may have the same chromosome, but it should be a different individual). The selection goes the same way, but if the individual already chosen as the first individual is chosen again, the selection is repeated until we have two different individuals. If the population were larger, we wouldn't have to bother with this reselection, since the chance of the same individual being selected twice would be small and of no consequence (in Goldberg's version of the SGA this concept isn't implemented). With such a small population, however, the restriction is useful.

When the two individuals are selected, an operator is chosen, either reproduction or crossover, also in a probabilistic manner. Since the crossover probability is 0.7, crossover has a chance of 0.7 to be chosen, and reproduction a chance of 0.3.

If reproduction is chosen, both individuals are copied to the next generation. If crossover is chosen, a crossover point is determined, also at random. Since there are seven possible crossover points on a chromosome of length 8, a seven-sided die is thrown to select the crossover point. The two chosen individuals now become the parents of two new individuals in the next generation by crossing them at the selected crossover point.

This process is repeated until the new population has as many individuals as the previous population. After that, every allele in the new population gets a chance for mutation. In our example we set the mutation probability to 0.001, so the chance is very

Generation: 2							
Index	Chromosome	Phenotype	Fitness	P1	P2	CrP	Mut
1	0**0**0010**01**	0.5625	0.1379	2	1	7	0
2	0**0**0101**01**	1.3125	0.4467	2	1	7	0
3	0**0**0101**01**	1.3125	0.4467	1			0
4	1**0**1111**01**	11.8125	0.0548	3			0
5	0**0**0111**01**	1.8125	0.6364	1	3	4	0
6	1**0**1101**01**	11.3125	0.0821	1	3	4	0
Minimum fitness:			0.0548				
Maximum fitness:			0.6364				
Average fitness:			0.3008				

table 4: The second generation. The bold genes have converged to a fixed allele value.

slim. However, in the case that we find an allele is to be mutated, the allele is changed into a random legal value that is not equal to its original value. Since in our example there are only two legal values for an allele, 0 and 1, the allele is changed to the other value.

It should be noted that the selection of two individuals, followed by the selection of the genetic operator, is a personal interpretation of Goldberg of the SGA. More commonly the operator would be chosen first, followed by the choice of either one or two individuals, according to the needs of the operator.

The second generation in our experiment is shown in table 4. The columns with the captions P1 and P2 show the parents of the individual. If the individual was created by means of reproduction, column P2 is empty. The column CrP shows, in the case of crossover, the crossover point. The column Mut shows the number of mutations the chromosome has undergone.

Because of the high fitness of individual 1 in relation to the other individuals in the first generation, it is highly likely that this individual will occur as a parent for many individuals in the second generation. As we can see this is indeed the case. Individual 1 is the parent of no less than five individuals in the second generation. On the other hand, individuals 4 and 5 in the first generation have extremely low fitnesses, and as expected they had no influence on the second generation. Individual 6 also befell this fate, in spite of the fact that its fitness is comparable to the fitness of individual 3.

As we can see from the minimum, maximum and average fitness of the second generation, the population as a whole is a lot better than the first generation. We can also see that some convergence has taken place. All six individuals in the second generation have the same allele in the second, seventh and eighth locus. Therefore, unless one of these alleles is mutated, the fittest individual in the final generation will have a zero at loci 2 and 7, and a 1 at locus 8.

Further generations

Just like the second generation is constructed from the first, the third generation is

Generation: 3							
Index	Chromosome	Phenotype	Fitness	P1	P2	CrP	Mut
1	00001001	0.5625	0.1379	1			0
2	00011101	1.8125	0.6364	5			0
3	10011101	9.8125	0.2130	6	5	1	0
4	00110101	3.3125	0.9680	6	5	1	0
5	00011101	1.8125	0.6364	5	2	5	0
6	00010101	1.3125	0.4467	5	2	5	0
Minimum fitness:			0.1379				
Maximum fitness:			0.9680				
Average fitness:			0.5064				

table 5: The third generation.

Generation: 4							
Index	Chromosome	Phenotype	Fitness	P1	P2	CrP	Mut
1	10011101	9.8125	0.2130	3	4	5	0
2	00110101	3.3125	0.9680	3	4	5	0
3	00011101	1.8125	0.6364	2			0
4	00110101	3.3125	0.9680	4			0
5	00011101	1.8125	0.6364	2	5	4	0
6	00011101	1.8125	0.6364	2	5	4	0
Minimum fitness:			0.2130				
Maximum fitness:			0.9680				
Average fitness:			0.6764				

table 6: The fourth generation.

Generation: 5							
Index	Chromosome	Phenotype	Fitness	P1	P2	CrP	Mut
1	**00**1**1**0**101**	3.3125	0.9680	5	2	1	0
2	**00**0**1**1**101**	1.8125	0.6364	5	2	1	0
3	**00**0**1**0**101**	1.3125	0.4467	5	2	4	0
4	**00**1**1**1**101**	3.8125	0.9978	5	2	4	0
5	**00**0**1**1**101**	1.8125	0.6364	3			0
6	**00**0**1**1**101**	1.8125	0.6364	6			0
Minimum fitness:			0.4467				
Maximum fitness:			0.9978				
Average fitness:			0.7203				

table 7: The fifth generation. The chromosomes have converged to six fixed allele values, indicated by the bold columns.

Generation: 6							
Index	Chromosome	Phenotype	Fitness	P1	P2	CrP	Mut
1	00110101	3.3125	0.9680	1			0
2	00010101	1.3125	0.4467	3			0
3	00111101	3.8125	0.9978	4			0
4	00011101	1.8125	0.6364	2			0
5	00011101	1.8125	0.6364	5			0
6	00011101	1.8125	0.6364	2			0
Minimum fitness:			0.4467				
Maximum fitness:			0.9978				
Average fitness:			0.7203				

table 8: The sixth generation.

Generation: 7							
Index	Chromosome	Phenotype	Fitness	P1	P2	CrP	Mut
1	00011101	1.8125	0.6364	6	4	3	0
2	00011101	1.8125	0.6364	6	4	3	0
3	00110101	3.3125	0.9680	1	5	6	0
4	00011101	1.8125	0.6364	1	5	6	0
5	00111101	3.8125	0.9978	3			0
6	00011101	1.8125	0.6364	5			0
Minimum fitness:			0.6364				
Maximum fitness:			0.9978				
Average fitness:			0.7519				

table 9: The seventh generation.

Generation: 8							
Index	Chromosome	Phenotype	Fitness	P1	P2	CrP	Mut
1	00011101	1.8125	0.6364	4	5	6	0
2	00111101	3.8125	0.9978	4	5	6	0
3	00110101	3.3125	0.9680	3			0
4	00111101	3.8125	0.9978	5			0
5	00011101	1.8125	0.6364	2	4	7	0
6	00011101	1.8125	0.6364	2	4	7	0
Minimum fitness:			0.6364				
Maximum fitness:			0.9978				
Average fitness:			0.8121				

table 10: The eighth and final generation.

constructed from the second. Every new generation is created from the previous one, just like generation 2 was created from generation 1. In the third generation we find a highly fit individual.

If we compare the chromosome of the highly fit individual 4 to the other chromosomes in the population, we find that a distinguishing feature of this chromosome is that its first three alleles are 001, while almost all others (except for one very weak individual) have 000 as the first three alleles. If this feature is crucial to the fitness of individual 4, we may expect this feature to return in following generations.

In fact, if we analyse the problem, we find that the fittest chromosome to be found which starts with three zeroes, which is 00011111, is still less fit than the weakest chromosome to be found which starts with 001, which is 00100000. Because GAs are all about favouring fit individuals, we may and do indeed find more occurrences of chromosomes which start with 001 in the fourth generation, displayed in table 6, and later generations.

In the fifth generation, displayed in table 7, we see a high rate of convergence. There are only four different chromosomes in this generation, and all the chromosomes are equal in six loci. The fittest individual in this generation has indeed the best possible chromosome which has these six loci fixed at the specific allele values which are common to all the chromosomes. Unless mutation helps us out, we cannot expect a better individual to arise. Indeed, in the eighth and final generation we don't find a fitter individual. The minimum and average fitness have got a little better, but that is not our primary concern. The last three generations are shown without further comment.

The development of fitness

It's interesting to see the development of the minimum, average, and maximum fitness. This is presented graphically for our experiment in figure 9. As we can see all three

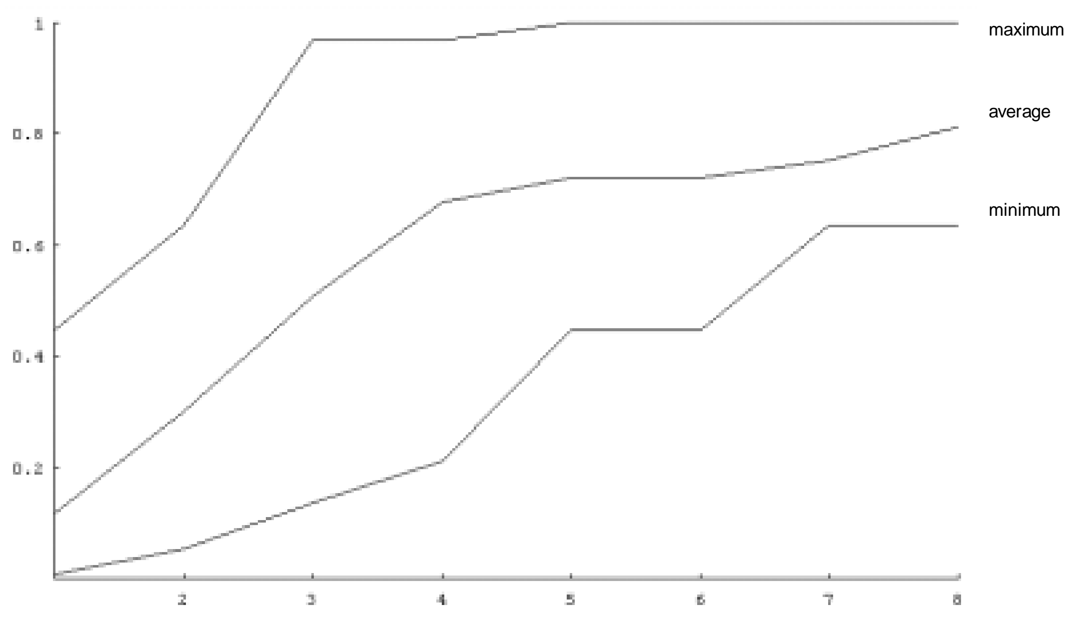


figure 9: The development of minimum, average and maximum fitness for eight generations.

fitnesses have a tendency to get better. It's not necessarily the case that they will always get better, especially not in the case of the minimum fitness, but this tendency is typical for GAs. In fact, if this wouldn't be the case, GAs wouldn't work.

It should be noted that the optimal fitness of 1.0000 isn't reached in this run. We *do* find a fitness that is close to the optimum. Since GAs are probabilistic in nature, we cannot expect to find the optimum, and it's not smart to rely solely on GAs when your goal is to find *the* optimum. However, if your goal is to find an "acceptable" approximation of the optimal solution, GAs can be of great help.

1.4 GAs and conventional search algorithms

There are a number of differences between GAs and conventional search algorithms like "hillclimbing" (which starts at a random point on a function and then climbs the function in the steepest possible direction until a local maximum is reached) or "simulated annealing" (which chooses an initial random point on a function and then makes small changes in this point until the system seems to be "frozen", meaning that the small changes lead to no better solutions). The most eye-catching differences are the following:

- *GAs are blind*. Conventional search algorithms change function parameters. GAs work with coded parameters, and change the coded parameters, without any concern for the meaning of the code.
- *GAs are inherently parallel*. Conventional search algorithms work with just one alternative solution, while GAs work with a population of alternative solutions.
- *GAs don't use additional information*. Conventional search algorithms often use additional information, like the derivative of the function to be maximised in order to determine in which direction the function increases. GAs use no additional information. The only information a GA uses is the fitness of the chromosomes in the current population.
- *GAs are probabilistic in nature*. Conventional search algorithms are deterministic, while GAs are probabilistic in nature. However, if we compare GAs with a purely probabilistic search, GAs work far better, because they use the implicit knowledge which is found in the fitness of the chromosomes.

Because of their characteristics, GAs work in many different environments and circumstances, with many different kinds of problems. Another way of phrasing this, is to say that GAs are *robust*. Most conventional search algorithms are not robust, and therefore can only be used with the problems for which they specifically have been designed.

There are many good points about GAs, but we shouldn't close our eyes for the weaknesses. The most notable weak points about GAs are the following:

- There is no thorough theoretical framework for the evolutionary process, as opposed to, for instance, statistical algorithms.
- There is no guarantee that a solution will be found, not even a mediocre one, which

is not the case with, for instance, exhaustive searches.

- Often GAs are slow in comparison with conventional techniques, since whole populations are examined instead of separate solutions.

Balancing the advantages and weaknesses of the use of GAs, we may conclude that GAs can be of greatest value with those problems for which no good dedicated search algorithm as yet exists or can be found.

1.5 Schemata

To analyse the inner workings of GAs, John Holland introduced the concept of *schemata*. A schema is a *similarity template*, defining a subset of strings in a population which are similar at certain string positions. A schema is presented as a string, which has the same length as all the other strings in the population, and uses the same alphabet, to which a "don't care" symbol is added, usually an asterisk. A schema defines those strings in the population that are equal to the schema in all those positions that are *fixed*, that is, that don't have the "don't care" symbol.

For instance, suppose the strings are of length 8, and the string alphabet is {0,1}. Then the schema alphabet is {0,1,*} and all schemata are also of length 8. The schema 000*01*0 describes the subset {00000100, 00000110, 00010100, 00010110}, the schema 00****** describes the subset consisting of all those strings that start with two zeroes, and the schema ******** describes the entire population.

Holland explains the fact that GAs "work", by pointing out that the fitness of a string in the population is not only applied to the string itself, but also, indirectly, to all the schemata which define a subset that contains the string. When a schema describes a subset that contains mostly fit strings, GAs may favour this schema in future generations, and the population will get more and more strings which belong to that particular subset.

For instance, in the SGA example earlier in this chapter, I pointed out that strings which start with 001 are generally more fit than strings that start with 000. This is the same as pointing out that the schema 001***** defines a subset of strings which are generally more fit than the strings that are contained in the subset defined by the schema 000*****.

Call the size of a population n, and the length of a string L. Each position in a string can be replaced by a * to get a schema, so each string defines 2^L schemata which are part of the population, and therefore the entire population contains between 2^L and $n2^L$ schemata. However, not all of these schemata are processed in a useful manner by the SGA process.

For instance, the string 10000010 is, among others, defined by the schema 1*****1*, but when crossover operates on this string, there is a 6/7 chance that the crossover point will fall between the two 1's and the schema will get disrupted. A schema that gets disrupted between generations isn't processed usefully, since it doesn't survive between generations even if it is fit.

Holland calculated the number of schemata that are processed in a useful manner by the SGA to be $O(n^3)$, which is a lot better than the number of individual strings that are

processed, which is no more than the size of the population, n.

The Schema Theorem

If we have a schema H, the *order* of the schema, denoted by $o(H)$, is defined as the number of fixed positions present in the schema. The *defining length* of the schema, denoted by $\delta(H)$, is defined as the distance between the first and the last fixed string position.

For instance, o(*11**0**) = 3, since there are three fixed positions, and δ(*11**0**) = 4, since the last fixed position is 6, he first fixed position is 2, and 6 minus 2 is 4. If a string has only one fixed position, it's defining length is zero. For the schema that has no fixed positions at all, there is no defining length.

There is an *occurrence* of a particular schema in a population if there is a string in the population which is part of the subset of strings defined by this schema. The number of occurrences of a schema in a population is the number of strings in that population that are defined by that schema. The fitness of the schema in that particular population is defined as the average fitness of all the strings in the population that are defined by that schema. This means that the fitness of a schema can change in time.

This warrants an example. Observe the first generation of the SGA example. There are two occurrences of schema ***101** in this population, namely the chromosomes 1 and 4. This means that the fitness of this schema in the first generation is equal to the average fitness of these chromosomes, or (0.4467 + 0.0088)/2 = 0.2278. The second generation shows three occurrences of the same schema, namely chromosomes 2, 3 and 6. The fitness of this schema in the second generation is (0.4467 + 0.4467 + 0.0821)/3 = 0.3252.

Let $m(H,t)$ be the number of occurrences of schema H in a population at time t. Let $f(H,t)$ be the fitness of schema H at time t. Let $f_{\text{pop}}(t)$ be the average fitness of the entire population at time t. Let L be the string length, p_{c} be the crossover probability and p_{m} be the mutation probability. Holland has calculated that for the expected number of occurrences of a schema H at time t+1, $m(H,t+1)$, when processed with the SGA and disregarding a few very small cross-product terms, holds that:

$$(3) \qquad m(H,t+1) \geq m(H,t) \cdot \frac{f(H,t)}{f_{\text{pop}}(t)} \cdot \left[1 - p_{\text{c}} \cdot \frac{\delta(H)}{L-1} - o(H) \cdot p_{\text{m}}\right]$$

Those interested in how this formula is derived can look it up in Holland's book (Holland 1992) or, for a less mathematical approach, Goldberg's book (Goldberg 1989). Goldberg also tells us what we should learn from this formula, namely that short, low-order, above average schemata receive exponentially increasing trials in subsequent generations, when the principle genetic operators are reproduction, crossover and mutation. This is called the *Schema Theorem*, or the Fundamental Theorem of Genetic Algorithms.

Going back to the SGA example, it was stated that we would expect chromosomes that started with 001, one of which appeared for the first time in generation 3, to return in future generations. According to formula (3), this is true, as we calculate the expected number of occurrences of schema 001***** in the fourth generation, by

substituting in the formula the values from the third generation:

$$(4) \qquad m(001{*}{*}{*}{*}{*},4) \geq 1 \cdot \frac{0.9680}{0.5064} \cdot \left[1 - 0.7 \cdot \frac{2}{7} - 3 \cdot 0.001\right] = 1.5235$$

So, the Schema Theorem states that the number of occurrences of schema 001***** in the fourth generation should be at least 2. And indeed, there were two chromosomes represented by this schema in generation 4, so the number of occurrences of this schema was 2.

Because of the very small population size, we shouldn't regard this as too significant a result. If we check the same schema in later generations, the results are not according to the Schema Theorem. The larger the population, the more accurate the Schema Theorem will be. This is because the Schema Theorem only tells us what we *statistically* may expect to happen, and the larger the population, the more trust we may place on statistics.

The Schema Theorem places emphasis on highly fit schemata of short defining length. Such schemata are called *building blocks*. Instead of trying to create highly fit strings from scratch, by using the crossover operator GAs combine fit, short substrings (building blocks) to form strings of potentially higher fitness.

Of course, this is only an explanation of why GAs *can* work. It is not a guarantee that they *will* work. Indeed, it is possible to define problems which are provably misleading to GAs, for instance, where most of the building blocks which lead to a local optimum have a higher fitness than the building blocks which lead to the global optimum. These problems are called *GA-deceptive*. Typical for these problems is that in the solution space, the best solutions are surrounded by the worst, while the next-to-best solutions are surrounded by solutions which are also quite good. However, these problems are difficult to solve for almost all search techniques, and in practice they are rare.

A problem that is *GA-hard* is a problem where not only the optimal solution is difficult to find for a GA, but where a GA actually diverges the search away from the global optimum. It should be noted that not all GA-deceptive problems are GA-hard, so even with GA-deceptive problems we may find that a GA can still solve them.

A very simple example of a GA-deceptive problem can be found in Goldberg's book (Goldberg 1989, pp. 46-48), but this example is too long and too technical to repeat here.

1.6 Making GAs work

As stated before, the SGA is not the most efficient or the most effective way to use a GA, although there are problems that can be handled with the SGA. For instance, a few reasonably fit strings from the initial population may easily flood the entire population in just a few generations, and convergence will occur at a very early stage. This means only a small part of the solution space is searched, and possibly the best solutions are missed. Also, the crossover and reproduction operators are not suitable for many problems, and other operators may be needed, either as an addition to or in exchange

for these operators.

This paragraph will explore some, but certainly not all, of the possible adaptations of the GA process and alternatives for genetic operators. Not all will be explored in detail, but a fair amount of attention is given to the processes of scaling, selection and replacement, and to the genetic operator-classes inversion and permutation. They should serve as an indication of what different techniques may be used to create a GA which is suitable for a specific problem.

Scaling

Premature convergence is recognised as one of the greatest fallacies of the SGA. There are many methods to avoid it. One such method is *scaling*.

In the early stages of a GA, a few very fit individuals may tend to dominate the population. In this case, we would like to have their fitness value scaled down, so that less fit individuals also get a chance to contribute to the gene pool of the next generation. In the latter stages, the population may have converged quite a bit, and the fitness of the individuals may be more evenly spread. In this case we would like to single out those individuals that are just a little bit better than the rest of the population, so we would like to scale up their fitness values.

There are different scaling methods. The most basic of these is *linear scaling*. Most others are much like this type of scaling. Linear scaling works as follows. First, we calculate the fitness of all individuals just like we did before. This fitness is called the *raw fitness*. We calculate the average raw fitness f_{avg} and we determine the maximum raw fitness f_{max} and minimum raw fitness f_{min}. Then we calculate for every individual a new fitness value which is called the *scaled fitness*. The scaled fitness is the fitness we're going to use as the fitness for the GA process.

The scaling process is illustrated in figure 10. The scaled fitness f' is determined from the raw fitness f as the result of the following linear relationship:

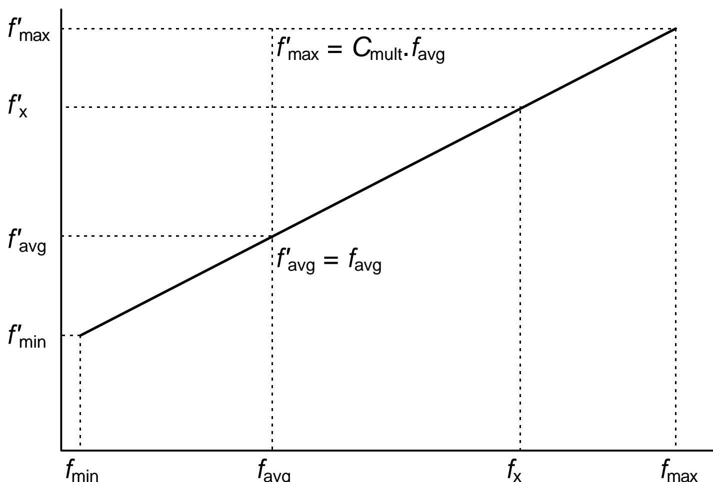


figure 10: Linear scaling of the fitness. C_{mult} is chosen to be 2. The scale is determined by the calculation of f'_{avg} and f'_{max}. After that, it's easy to determine the scaled version of any fitness f_x.

$$(5) \qquad f' = af + b$$

To determine the constants a and b in this equation, we make use of the following relationships for the average scaled fitness f'_{avg} and the maximum scaled fitness f'_{max}:

$$(6) \qquad f'_{avg} = f_{avg}$$

$$(7) \qquad f'_{max} = C_{mult} \cdot f_{avg}$$

The constant C_{mult} is chosen as the number of expected copies in the next generation of the individual with the highest fitness. For instance, if we choose C_{mult} to be 2, we would expect the fittest member of the current population to be selected only twice to be used for some genetic operator in creating the next generation. Since we keep the average scaled fitness equal to the average raw fitness, average individuals may be expected to contribute one offspring to the next generation.

However, these formulas could lead to a negative fitness for very weak individuals, which isn't acceptable. Therefore, if we find that the minimum raw fitness leads to a minimum scaled fitness f'_{min} that is negative, we scale "as much as possible" by using the following, alternative equation instead of equation (7), this time for the minimum scaled fitness f'_{min}:

$$(8) \qquad f'_{min} = 0$$

Selection

Many mechanisms exist to select individuals for some genetic operation. In the SGA, the selection is a simple roulette wheel selection. This selection has some weak points. For instance, it is very well possible that the best individual in a population is not selected for the next generation, especially when there are several quite fit individuals in the population.

Elitist selection takes care that this best individual is always selected for reproduction at least once, simply by adding this individual to the next generation if it doesn't appear in this next generation when the next-to-last individual is produced.

Expected value selection tries to minimise stochastic errors with the selection process. A maximum is placed on the number copies an individual may contribute to the next generation, which is equal to the individual's fitness divided by the sum of all the fitnesses, rounded up.

Replacement

It has proved to be very useful not to replace an entire population by a new generation. Instead, just a part of the population is replaced by new individuals. This leads to a second kind of selection, namely, the selection of the individual to be replaced.

Crowding is a much-used method to implement this replacement mechanism. It needs two parameters: the *generation gap* and the *crowding factor*. The generation gap

indicates how big a fraction of the current population is to be replaced. The crowding factor indicates how many individuals will be selected to choose an individual from that will be replaced. For instance, if the crowding factor is 2, and there is a new individual generated, two individuals will be selected purely at random from the current population. One of these will be replaced by the new individual, namely the one which differs the least from the new individual. This difference is a character-by-character comparison, like a Hamming distance.

Another method is *preselection*. This method also uses a generation gap. With preselection, however, the individual to be replaced is not selected randomly. It is one of the parents of the new individual that will be replaced, namely the parent with the smallest fitness.

With both methods mentioned, the individual to be replaced and the new individual will be somewhat alike. This helps keeping diversity in the population and therefore slows convergence down.

Inversion

Until now the only operators mentioned to create individuals for a new generation were reproduction and crossover. A third group of operators are the *reordering* operators. Reordering operators change the ordering of the characters in a string. Basically, there are two kinds of reordering operators: those that change the loci and those that change the allele values.

Inversion is the most common operator that changes the loci on a chromosome. Two points are selected on the chromosome, the chromosome is cut at these two points, the part in the middle is reversed and then the chromosome is reassembled. This operation *only* changes the loci, so the new chromosome expresses exactly the same phenotype as the old chromosome. For example, take the following chromosome over the alphabet {0,1} (the numbers above the alleles are the loci):

```
1 2 3 4 5 6 7 8
1 0 0 0 1 1 1 1
```

If we use the inversion operator on this chromosome, and the cutting points are chosen as “between the second and third allele” and “between the sixth and seventh allele”, this chromosome would become:

```
1 2 6 5 4 3 7 8
1 0 1 1 0 0 1 1
```

At first, this seems to contribute little to the GA process. However, suppose there exists a very fit schema that has two fixed positions that are far from each other. Such a schema would be likely to get disrupted by the crossover operator and therefore not be usefully processed by the GA. Inversion could bring these fixed positions closer to each other, without disturbing the meaning of the chromosome or the schema. Of course, in combination with inversion the crossover operator becomes much more difficult to implement, since inversion is applied to specific individuals and not to the population as a whole.

Permutation

There are several reordering operators that change the allele values. These are used when a permutation of a string is needed to generate a solution for a problem. For instance, in the *travelling salesman* problem, we are to determine the shortest circular route between a number of cities that visits each city exactly once. If we give each city to be visited a unique number, a natural expression of a solution would simply be a string which contained each city number once, and the order of the string would indicate the order wherein the cities would be visited. Whatever changes we make in the string, it should always contain each city number once and only once. However, we would also like to implement something like crossover, to make fit strings reinforce each other, but normal crossover wouldn't result in a permutation of the original string.

A basic operator that could be used is the *partially matched crossover* (PMX). Two parent strings are selected, and two crossover points on the strings are selected. We cut out the substrings between the crossover points, and exchange them between the two strings. The problem is that the chances are great that the two strings now no longer contain all different values, and that some values will be doubled. This is solved by replacing the double values outside the substring with the disappeared values, exchanging them for the value that was replaced by their double. For example, we have the following two strings:

7 5 3 1 2 4 6 8
1 5 2 6 3 7 4 8

We cross them so that the middle two characters are exchanged, which gives us:

7 5 3 **6** **3** 4 6 8
1 5 2 **1** **2** 7 4 8

The first string now has twice a 6 and twice a 3. Since the new 6 replaced a 1, the old 6 at locus 7 now gets replaced by a 1. The new 3 replaced a 2, so the old 3, at locus 3, gets replaced by a 2. The same happens to the second string, which leaves us with the following two strings:

7 5 **2** 6 3 4 **1** 8
6 5 **3** 1 2 7 4 8

PMX is not the only permutation operator. Others are *order crossover* (OX) and *cycle crossover* (CX). OX is very much like PMX, but it has a different way of removing the double values. CX works somewhat different and has a habit of perturbing a chromosome far more than the other two operators, but on the other hand, every allele value of the child chromosome is always found in the same locus as in one of the parents, which can be a boon with some problems.

Other mechanisms

I will briefly mention a few other mechanisms.

Multiparameter optimisation means the natural expression of our problem asks for more than one parameter, for instance, if we must solve a set of equations with more than one variable. We could solve this by defining the genotype as a collection of chromosomes instead of one chromosome. However, very often the parameters are just glued together to make one large chromosome.

Multiobjective optimisation or *multicriteria optimisation* means we have more than one fitness criterion for our problem. It is sometimes impossible or unwise to combine these criteria. There are ways to use GAs with more than one fitness function. One way is to rank individuals by *dominance*: an individual dominates another individual if it scores better on all fitness criteria. Nondominated individuals, individuals that aren't dominated by any other individual, are ranked at the top. The rest of the population is again examined, and the individuals that are now nondominated are ranked second. This continues until all individuals are ranked. Selection is then done according to rank.

The absolute value of the fitnesses may have an impact on the performance. For instance, if the fitnesses of the chromosomes in a generation vary between 0.5 and 1.0, we may get different results than if the fitnesses vary between 99.5 and 100.0. A technique called *windowing* is often used to reduce the impact of the absolute values of the fitnesses. Windowing subtracts the same value from all the fitnesses, so that the lowest fitness ends up at a selected value, often zero.

Another fitness-adapting technique is *linear normalisation* or *ranking*. The fitnesses are sorted in decreasing order. The highest fitness is changed in a constant value. This value gets decreased linearly to give the other fitnesses a new value.

Diploid chromosomes are chromosomes that have at least two alleles at each locus. One of the alleles is the dominant allele, and the phenotype is only influenced by this allele. In time, however, the other allele could become dominant. In a diploid chromosome therefore inactive allele values are preserved for future use. This could be useful in a changing environment. If some allele combination is quite fit, but the environment changes and the combination is no longer fit, it would be unwise to remove this combination entirely from the population, since the circumstances which made it fit could arise again in the future. The opposite of the word diploid is *haploid*. Normally we work with haploid chromosomes.

Sometimes it is useful to combine a conventional search method and a GA. For instance, when we have a local search method that performs very well, we could use a GA to quickly point out the areas where the optimum is most likely found, and then let local search pick out the local optimum at these areas. These are called *hybrid techniques*.

The term *hybrid techniques* is also used in situations where we apply a GA-like technique, which is not exactly like Holland designed it. For instance, sometimes real numbers are used instead of binary strings as chromosomes in the population. This means the crossover and mutation operation need to be implemented differently.

Knowledge augmented operators are operators which use not only the fitness function to select the individuals to work upon, but also use some problem-related information. The GA can no longer be called blind, but this can work quite well with certain problems.

The *Breeder Genetic Algorithm* (BGA) is a technique whereby not every individual can be selected for reproduction. Only the best individuals, a certain percentage of the population, may produce the next generation. These are crossed randomly.

Random Respectful Recombination (R^3) produces a child by giving the child all the alleles that the parents have in common, and then decides on the remaining allele values by making random, legal choices. There are situations where this is surprisingly effective.

The crossover operator is found in many different forms. For instance, there is *multiple point crossover*, which splits the chromosomes in more than one piece, *multiple parents crossover*, in which more than two parent chromosomes are used to produce a child, or a combination of these techniques. There is *uniform crossover* (UX), whereby two parents produce one child, and for each child-allele one parent is randomly chosen, and the allele is copied from the allele in the same place on that parent. There is *half-uniform crossover* (HUX), which is the same as UX, but afterwards each of the children is again UX'd with one of the parents, so they will be more alike to the parents.

The crossover operator is a very difficult operator to implement in such a way that it works well for the problem under consideration. It is therefore no surprise that so many different varieties exists. Many researchers in GAs use tailor-made crossover operators on their problems.

1.7 Parallel GAs

Because GAs work with a population of alternative solutions, GAs are seen as inherently parallel. This in itself doesn't make a GA easy to implement on a computer which supports parallel processes. In Holland's version of the GA, we select the parents of a member of the next generation from the complete current population. If we want to generate new members on a parallel system, each processor needs to be able to access the complete current population. It's not standard practice on a parallel system that the processors have access to a large common database.

When a population is viewed as one large group of individuals, which interbreeds without restriction, we call it a *panmictic* model. To solve the problem of implementing a GA on a parallel system, we need to replace this panmictic model with another model. The study of parallel GAs has become a separate branch in the study of GAs. This branch is known as parallel genetic algorithms (PGAs).

The island model

The *island* model, also called the *coarse-grained* model, works with a number of panmictic populations. Each of these populations develops like the standard GA, but on its own, so they can be processed in parallel with one processor for each population.

Sometimes the populations exchange a few individuals. This is called the *migration* phase. Migration makes it possible that the populations get access to the fit allele combinations of the other populations.

Migration can be implemented in a number of ways. Sometimes the individuals which migrate, and the islands they migrate to are chosen randomly, sometimes deterministically. It can be done in synchronisation or asynchronously, which gives a better performance if there are slow processors in the cluster. Sometimes the islands are

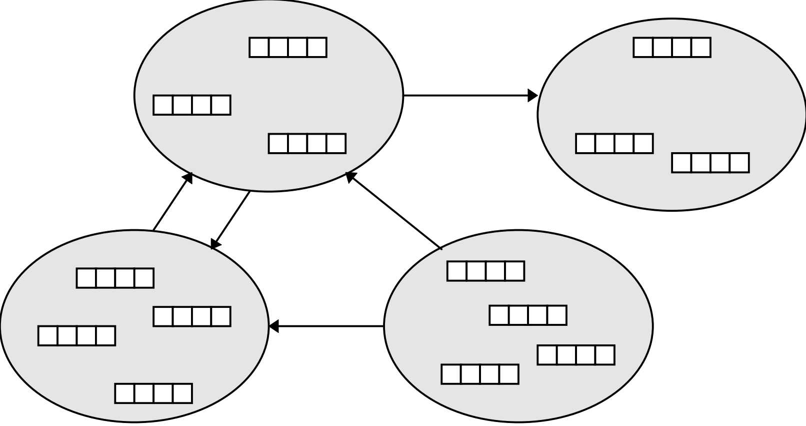


figure 11: The island model. The arrows show some possibilities for migration.

placed in a ring, and migration occurs only between neighbouring islands, which is known as the *stepping-stone* model.

Because the islands develop independently, this encourages *niching*. Different islands converge to different solutions. This approach is therefore suitable for multimodal problems, which are problems that have more than one optimum, as opposed to unimodal problems which have only one optimum. Because the only communication needed between the islands is the transmission of migrating individuals, this model is suitable for many architectures, for instance, a network containing several computers, a distributed system, or a MIMD (Multiple Instruction Multiple Data) machine.

The cellular model

In the *cellular* model, also known as the *fine-grained* or *diffusion* model, each individual has a unique co-ordinate in some space, typically a grid. The grid can be one, two or even more dimensional. The grid can have boundaries or be cyclic. Each

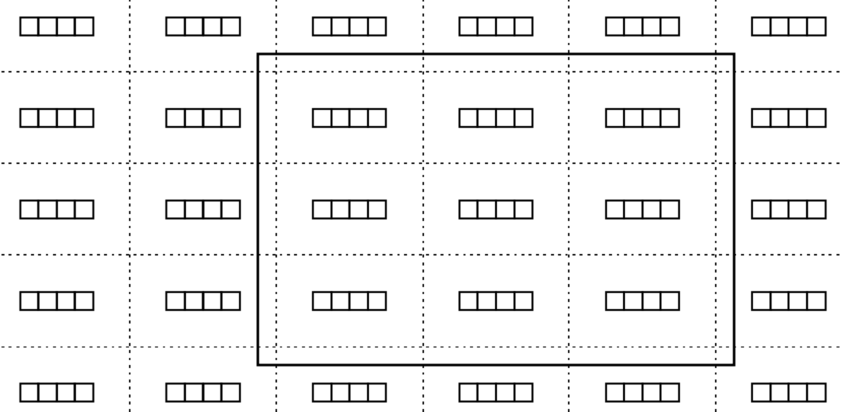


figure 12: A cellular model, in this case a two-dimensional grid. If demes are considered to be the individuals lying right next to a specific individual, the rectangle selects the individuals belonging to the deme for the individual in the middle.

individual only mates with the individuals within some specific radius. This group of possible mates is called a *deme*. New individuals replace existing individuals, also within the deme.

Demes overlap each other, and thus fit solutions tend to spread slowly over the grid. Inbreeding within the demes leads to the development of related solutions in a small area, but different solutions may develop in different places on the grid. This makes the cellular model, like the island model, suitable for multimodal problems.

This model can be implemented on a parallel computer wherein the processors are also placed in a grid. The processors only have to communicate with their neighbours, and only the individuals lying on the boundary between two processors need to be transmitted, and then only if chosen for mating or replacement. The model is especially suitable for SIMD (Single Instruction Multiple Data) machines.

1.8 Designing a GA

When we design a GA to solve a particular problem, we should try to exploit what we know about them to create such codings and such fitness functions that the GA will indeed be able to solve our problem. There are a few practical rules we may take into consideration. Remember, however, that adhering to these rules is no guarantee for success and that there are many problems for which it is impossible to follow all of them. As observed before, there is no firm theoretical basis for GAs and so most of these rules are purely practical observations which may or may not work out to the advantage of the problem under consideration.

First of all, the choice of fitness function is crucial. The fitness function should not only recognise the optimum, it should also be able to reward partially correct solutions. For instance, if our problem consists of solving the equation $x^2 = 1$, we cannot be satisfied with just allotting high fitness to values for x which solve the equation. We should in some way also give fairly high fitness to values for x which are near a solution. So, we have to design a fitness function that can distinguish between values for x which are near a solution and values which are far from a solution. If it is possible that non-legal chromosomes are created, the fitness function should give a penalty to chromosomes which aren't legal. It can be very effective to start with a low penalty, and to increase this penalty with the generations. This filters out non-legal chromosomes gradually, but not before they are allowed to make a contribution to the gene pool.

We should also be very attentive when designing the coding. Since building blocks are helpful to a GA, we should make sure that building blocks can be found in the coding. This leads to the *principle of meaningful building blocks*: we should select a coding so that short, low-order schemata are relevant to the underlying problem and are relatively unrelated to schemata over other fixed positions. A coding which observes the principle of meaningful building blocks is said to have low *epistasis*. Epistasis means that the influence of a gene on the fitness of an individual depends on the allele values of other genes elsewhere on the chromosome. If those other genes are near to the gene they influence, they can form a building block, and the epistasis effect has very little influence on the success of the GA. Unfortunately, there are many problems in which

the epistatic effect on a coding is unavoidable, but one should always attempt to make the epistatic effect as low as possible.

A GA is better off with a coding which employs a small rather than a large alphabet. With a small alphabet, it is easier for a GA to respond to similarities between fit strings, that is, to recognise the building blocks. This is expressed in the *principle of minimal alphabets*: we should select the smallest alphabet that permits a natural expression of the problem.

Very important is the choice of the operators. Reordering operators are aimed at another kind of problems than simple crossover operators. Some problems benefit of inversion, for instance, when we deal with multiparameter optimisation problems where we don't know the relation between the parameters.

Almost all operators need a selection mechanism. Selection is done mostly according to fitness, but even then there is a number of ways to implement selection. For instance, scaling with the right parameters often helps.

We should make balanced choices in determining the parameter values for the chosen operators, for the size of our population and for the number of generations. For example, it's no use to choose a very high mutation rate, since, as we can deduce from the Schema Theorem formula, a high mutation rate works against highly fit strings. Another choice has to be made for the population size. GAs work best with large populations, but we also want our GA to be fairly quick and efficient, which promotes a rather small population.

The designing of a GA is a process that needs tuning a bit and which becomes easier when one has some experience with GAs. One shouldn't count on choosing the right coding, the right fitness function and the right parameters on the first try. It is also advisable to run a GA with specific parameters more than once. If the results are rather erratic over the runs, one should mistrust the design.

1.9 Summary

Genetic algorithms (GAs) are search algorithms based on the mechanics of natural selection and natural genetics. GAs are most commonly used for optimisation problems, but since many problems can be formulated as an optimisation problem, GAs are widely applicable.

GAs works with a population of strings. Each string encodes an alternative solution to a problem. With each string a fitness value is associated, which indicates how well the string performs in relation to the other strings in solving the problem.

Genetic operators are used to combine strings to produce a new generation of strings. The operators favour strings with a high fitness value. The most common operators are reproduction, which simply copies a string; crossover, which takes two strings, cuts them at a random point, and glues the first part of the first string to the second part of the second string to produce a new string; and mutation, which changes a random position in a string.

The new population is used to create yet another population in the same way, which is in its turn used to create yet another population, and so forth, until a certain termination criterion is fulfilled. At that point, the fittest string is presented as the

solution to the problem. The procedure described here is also known as the simple genetic algorithm (SGA).

GAs distinguish themselves from conventional search mechanisms, in that they are blind, inherently parallel, probabilistic and that they don't use additional information about the solution. On the downside it should be pointed out that there is no solid theoretical basis for GAs, and that there is no guarantee a solution will be found. In practice GAs often work very well.

One explanation for the success of GAs is the Schema Theorem. A schema is a template for a collection of strings, which is defined by fixing some string positions, and leaving other string positions free. It can be argued that GAs favour schema's that define fit strings. For the SGA it is calculated that in a population of n strings, every cycle $O(n^3)$ schema's are evaluated. This enables the GA to quickly find the points in the solution space where the fittest strings reside.

Besides the operators found in the SGA, there are many other genetic operators. Some of them are slightly different versions of reproduction, crossover and mutation, while others are totally different. There are also variants of the population design, some of which are necessary to make GAs suitable to run on parallel systems. It is the task of the user to select the operators and other parameters which are needed to make a GA successful in solving a certain problem.

2 Genetic Algorithm Applications

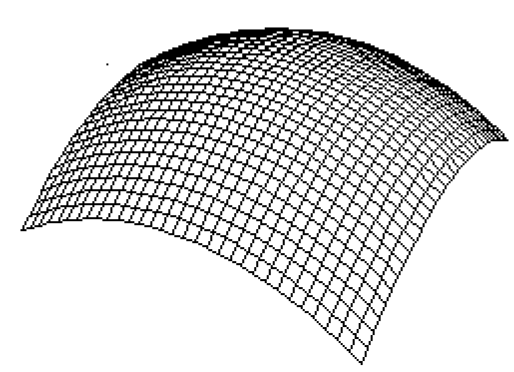
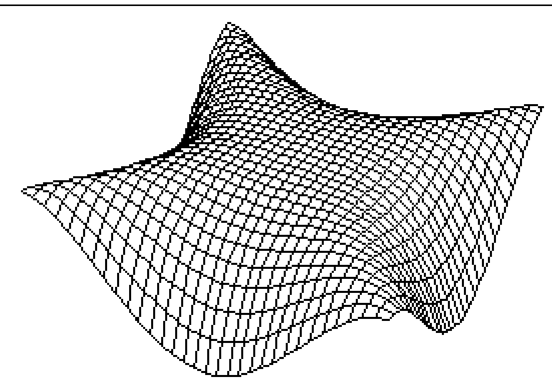
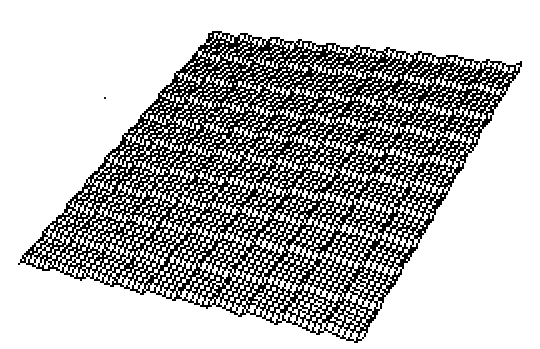
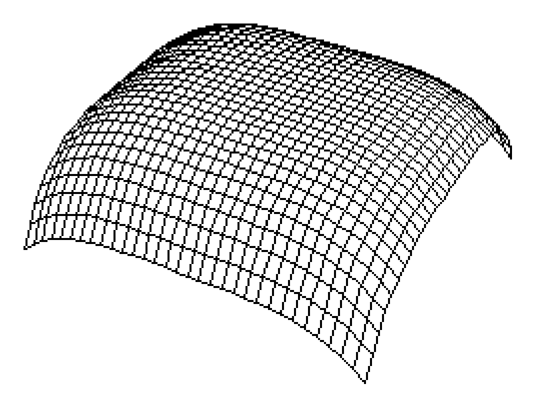
GAs are no more than a technique, a tool in the toolbox of the computer scientist. The most direct application of GAs is in the field of function optimisation. GAs are in their purest form only suitable for this purpose, since the direct result of a GA run is a maximised fitness function. However, lots of problems can be defined as an optimisation problem, and therefore GAs can be applied to a wide range of problems. Besides that, GAs are also used as the basis for or as a part of some other techniques.

This chapter will explore some of these GA applications. It will talk about generalised systems for solving problems, and also about some specific problems which can be tackled with GAs. A general survey of function optimisation with GAs is given (2.1), followed by two subfields of GAs: classifier systems (2.2) and genetic programming (2.3). An example is given of a way to combine GAs with conventional search techniques (2.4). Some detailed real-world examples of GAs are presented, namely the design of gaspipe networks (2.5), the design of a business model on a large and fuzzy database (2.6) and the design of an artificial neural network (2.7). This is followed by a short list of other fields where GAs have been applied successfully (2.8) and some examples of available tools for the application of GAs (2.9).

2.1 Function optimisation

Function optimisation is the most natural application for GAs, and since the arrival of GAs a lot of people have researched this subject. Most noted amongst them is K.A. de Jong, who published his influential doctoral dissertation *An Analysis of the Behavior of a Class of Genetic Adaptive Systems* in 1975, of which a condensed version can be found in Goldberg's book (Goldberg 1989). De Jong was very interested in the application of GAs in other fields than function optimisation, but he recognised the importance of careful examination of the behaviour of GAs in a laboratory setting before they were applied to more esoteric domains.

De Jong designed five test functions, with different characteristics in, among others, the areas of continuity, modality (the number of local optima) and dimensionality. He

	figure 13: De Jong's test function F1. $f_1(x_i) = \sum_{i=1}^{3} x_i^2$ $x_i \in [-5.12, 5.12]$ A two-dimensional, inverted version of F1 is displayed here. This is a simple parabolic function. It is continuous, convex, unimodal, quadratic, deterministic and low-dimensional.
	figure 14: De Jong's test function F2. $f_2(x_i) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2$ $x_i \in [-2.048, 2.048]$ An inverted version of F2 is displayed here. This is a two-dimensional quadric curve. It is continuous, non-convex, unimodal, quadratic, deterministic and low-dimensional.
	figure 15: De Jong's test function F3. $f_3(x_i) = \sum_{i=1}^{5} \text{floor}(x_i)$ $x_i \in [-5.12, 5.12]$ A two-dimensional version of F3 is displayed here. The function rises like a staircase to the optimum. It is discontinuous, non-convex, unimodal, non-quadratic, deterministic and high-dimensional.
	figure 16: De Jong's test function F4. $f_4(x_i) = \sum_{i=1}^{30} i x_i^4 + \text{Gauss}(0,1)$ $x_i \in [-1.28, 1.28]$ A two-dimensional, inverted version of F4 is displayed here (disregarding the Gaussian component). It is continuous, convex, unimodal, quadratic, stochastic and high-dimensional.

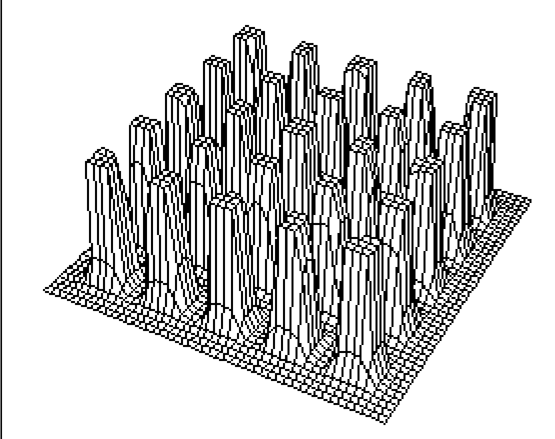


figure 17: De Jong's test function F5.

$$f_5(x_i) = 0.002 + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2} (x_i - a_{ij})^6}$$

$$x_i \in [-65.536, 65.536]$$

This function has up to 25 peaks, one for each pair (a_{1j}, a_{2j}). The function is continuous, non-convex, multimodal, non-quadratic, deterministic and low-dimensional.

called these function F1 through F5. They are displayed in the figures 13 to 17, with a characterisation. It should be noted that I've decided to call a function "high-dimensional" if its dimension is higher than 3.

He tried different implementations of GAs, and measured their convergence rate and their performance, which he called respectively the *off-line* and *on-line performance*. These names refer to the emphasis which different kind of applications put on different aspects of GAs. An on-line application is interested in reaching results quickly, and therefore performance is the most important aspect of a GA for this kind of application. An off-line application is more interested in the convergence rate.

The GAs he used were designed according to, what he called, a *reproductive plan*. The first reproductive plan he studied was the SGA, with a generation gap. Of course this is not one individual plan, but rather a family of plans according to the value of parameters like population size, operator probability and the size of the generation gap, which he varied to study their effects. After that he studied some other plans, using techniques like elitist selection, expected value selection, crowding and combinations of these.

Amongst the conclusions reached by De Jong were the following observations:

- Large populations score better than small populations in off-line performance, while worse in on-line performance. This comes as no surprise.
- A high mutation rate is often used to battle allele loss. De Jong found that although it reduces the loss of alleles, it does so in a very limited way, while it severely decreases both off-line and on-line performance. Therefore, mutation is a very questionable technique. It should be noted that even now mutation is still under debate. The stands vary from no mutations at all to a mutation rate of about one mutation for each individual in every generation.
- Off-line performance benefits most from a generation gap of 1, which means non-overlapping populations. Since generation gaps which are less than 1 take a bit more work in replacing an individual, we would expect on-line performance to degrade with smaller generation gaps. De Jong found that this degradation is very limited. This is important, since generation gaps are a technique often used with applications which "search while performing", which are typically on-line applications.

- Elitist selection improves local search at the expense of the global perspective.
- Expected value selection is a good remedy against allele loss. It might be a viable alternative for mutation.
- Most functions benefit in one way or another from the adaptations of the SGA which are put forth in the other reproductive plans. Multimodal functions like De Jong's test function F5, however, suffer in both on-line and off-line performance from all plans except for the crowding model. Since multimodal functions are daily practice in the areas where we would like to use GAs, crowding should be regarded as an important tool.

De Jong also concluded that a crossover probability of 0.6 seemed to be a good trade-off between on-line and off-line performance. Later studies showed that a crossover probability of 1.0 worked a lot better, if stochastic errors with the selection process were reduced, for instance with expected value selection.

2.2 Classifier systems

Machine learning systems are systems which learn a specific task by being changed under the influence of their environment. Best-known in this area are artificial neural networks. Classifier systems (CSs) are an architecture within the domain of *Genetics-Based Machine Learning* (GBML) systems. GBML systems use genetic search as their primary discovery mechanism.

A CS is a machine learning system that learns string rules, called *classifiers*, to guide its performance in an arbitrary environment. The three main components of a CS are a *rule and message* system, an *appointment of credit* system, and a genetic algorithm.

A CS is a black box, which gets information from the environment through *detectors*, and which performs some action in the environment through *effectors*. The information from the environment comes in the form of *messages*, which are placed on a fixed-size *message list*. Messages are fixed-length strings which contain characters from a specific alphabet. The messages may activate classifiers from the *classifier store*.

A classifier is a rule, consisting of a *condition* and a message. When the condition is fulfilled, the classifier may place its message on the message list. This is called '*posting* a message'. A condition looks like a schema, that is, it is of the same length as the messages and it uses the same alphabet as the messages use, to which a "wild card" symbol is added. The hash-mark (#) is a commonly used wild card. The condition is fulfilled when there is a message on the message list which is equal to the condition in all positions except for the positions containing wild cards.

Let's clarify this with an example of a CS which contains only a rule and message system. Suppose our messages have length 4, and our alphabet is {0,1}. The message list can contain two messages. The classifier store contains the following classifiers:

classifier 1: `11##:0001`
classifier 2: `1#00:1010`
classifier 3: `#000:1110`
classifier 4: `##01:0110`

For instance, classifier 1 states: “if there is a message on the message list which starts with double 1, message 0001 may be posted on the message list”.

The environment now delivers the message 1100 to the CS through the detectors. This message is posted. The CS checks if there is any message on the list which activates a classifier. 1100 is the only message on the list, and it can activate two classifiers, numbers 1 and 2.

Both classifiers 1 and 2 are activated. The message list is cleared, and the messages with classifiers 1 and 2 are posted. The message list now contains the messages 0001 and 1010. In the next iteration, only classifier 4 can be activated, namely by message 0001. The message list is cleared and the message with classifier 4, 0110, is posted. Since there are no classifiers activated by this message, the process terminates. Note that classifier 3 is never activated. Schematically:

Step 1:	The environment posts message 1100.
Step 2:	Message 1100 activates classifiers 1 and 2.
	Classifier 1 posts message 0001.
	Classifier 2 posts message 1010.
Step 3:	Message 0001 activates classifier 4.
	Classifier 4 posts message 0110.
	The process terminates.

What we try to do with a CS, is deciding on a set of classifiers which solve a specific problem for us. The way this is done, is by manipulating the classifier store and by using an appointment of credit system to reward classifiers which we need to solve our problem. First, let’s discuss the appointment of credit system.

Each classifier in the classifier store maintains a record of its net worth, called its *strength*. The higher the strength of a classifier, the better it performs and the more we seem to need it. So, while the CS is active, it has to change the strength of the

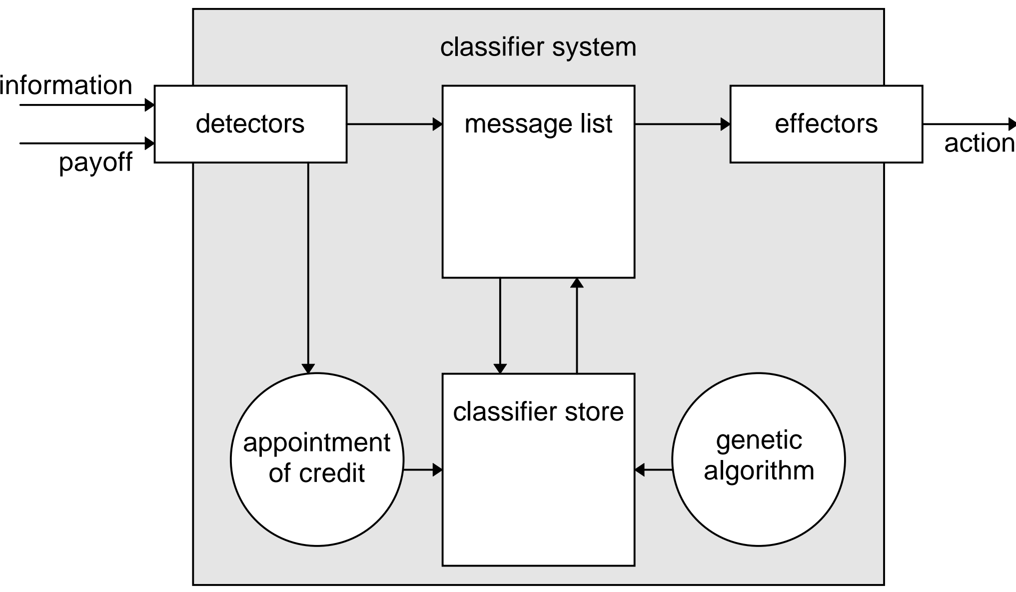


figure 18: A classifier system.

classifiers according to their effect on the environment. One prevalent method to do this is the *bucket brigade*.

The bucket brigade makes use of an *auction* and a *clearinghouse*. Every iteration an auction is held. All classifiers make a bid to post their message. This bid is according to their strength. If the message list is too short to contain all messages which could be posted, the highest bidding classifiers are activated. The activated classifiers post their message.

Now the clearinghouse phase is entered. All activated classifiers pay their bid to the classifier that posted the message which activated them, possibly to the environment itself. For instance, suppose in the above example all classifiers start out with a strength of 100. Suppose their bid is always one-fifth of their strength. In step 2, classifiers 1 and 2 are activated, so they both pay 20 to the environment. The environment's strength is increased by 40, while the strength of both classifiers 1 and 2 is decreased by 20. In step 3, classifier 4 is activated by the message posted by classifier 1. Classifier 4 now pays one-fifth of its strength to classifier 1. The process terminates.

We can now see how the strength of the classifiers changes. This is only useful if they change according to their success in solving our problem. This is done by rating the action the classifier system performs through the effectors on the environment. This action is mostly based on the current message list. For instance, the action can be the delivering of one of the messages on the list to the environment. The environment then sends a payoff to the classifier system, according to the success of the action. This payoff is distributed among the strengths of the classifiers which were responsible for the action.

The action can be performed every cycle, but we can also decide to run the system for a number of cycles before we perform the action. This can be a fixed number of cycles, or it can be based on the contents of the message list, or possibly something else.

A very simple implementation of a CS gives the classifiers a rule and an action instead of a rule and a message. The classifiers do not place new messages on the message list, but the classifier with the highest bid immediately performs its action through the effectors on the environment. Such a one-level CS is very easy to implement and can be surprisingly effective.

Since a classifier's bid is according to its strength, strong classifiers pay higher rewards to the classifiers which posted the messages that activated them than weak classifiers, so a chain of classifiers which leads to high payoffs from the environment strengthens itself, while classifiers which don't get rewards from the environment, and which don't indirectly activate well-performing classifiers, get weakened. Often some kind of taxing system is also used, by which every iteration classifiers pay a standard amount to the environment. This is to get rid of classifiers which are just sulking in the store without doing much.

Lastly, we need to manipulate the classifier store to get new, possibly better classifiers. This is done by viewing the classifier store as a population of chromosomes, and by using a GA to change this population, using the strength as the classifier's fitnesses. We have to design this GA so that it works well for a classifier system. This means, for instance, that we don't want the population of classifiers to change too rapidly. Use of the crowding model is a solution. We also need an extra parameter to decide at what moment the GA is activated.

It’s easy to see the parallel between artificial neural networks (ANNs) and CSs. In both cases, a black box system is optimised by training it with inputs for which the output is known. By rewarding correct answers and penalising errors, well-working links are strengthened and bad links are weakened or even removed.

The big difference between ANNs and CSs is that after the system is trained, a CS gives a definite set of rules that solves our problem, while a ANN just *is*. You cannot see what rules are implemented by the net, unless it’s really trivial. In a ANN the rules are *implicit*, while a CS gives *explicit* rules. On the other hand, a CS can generate *only* production rules, while a ANN can show more complex behaviour.

2.3 Genetic programming

In 1992 John R. Koza’s book *Genetic Programming* (Koza 1992) was published. It immediately found a great interest from many researchers in the field of GAs, a lot of whom started to work in Koza’s field. In 1994, not only Koza published a sequel to his 1992 book, but also a collection of papers from some of Koza’s followers was printed by the same publishing company, under the title *Advances in Genetic Programming* (Kinnear 1994), which showed genetic programming (GP) to be applicable in many different areas.

GAs manipulate data structures, namely chromosomes. GP starts with the idea that a computer program is no more than a data structure, albeit of another nature than the classic data structures used with GAs. Koza’s idea is to manipulate programs in the same way chromosomes are manipulated with a GA, to create a program which excels in performing a given task.

GAs are used to find the solution to a specific problem, for instance, finding the peak of some specific function. GP is also used to find the solution to a specific problem, only in this case the problem is finding an *algorithm* to perform a *specific task*. For instance, that task can be the sorting of an array. We don’t want to sort a specific array, but we want to discover an algorithm that sorts *any* array.

GP has some benefits over GAs. Solutions derived with GP are algorithms and therefore more flexible than solutions found by using a GA. More important, once an algorithm is found we have a reusable component to develop more complex programs. In fact, the focus of Koza’s 1994 book is the automatic discovery of programs which use components derived with GP. However, it is obvious GP is far more difficult to use than GAs are.

GP works with a population of computer programs, like GAs work with a population of chromosomes. Computer programs consist logically of a set of functions and terminals appropriate to a specific domain. For instance, if our domain is calculating with natural numbers, the set of functions may be a selection of mathematical operators like addition and multiplication, and the set of terminals may be the natural numbers combined with some variables. When solving some problem with GP, the *alphabet* used is a set of functions and terminals.

The program itself is expressed as a specific combination of functions and terminals. For instance, suppose we have the following two functions:

`add(.,.)` which takes two parameters, adds them together, and returns the result.
`mult(.,.)` which takes two parameters, multiplies them, and returns the result.

Furthermore, we have two terminals `x` and `y` which are variables and which can both contain only natural numbers. We can now create the following program, which takes a number and results in a number twice as big:

```
add(x,x)
```

Another program, which takes two numbers, adds them together, multiplies the result of this addition with the second number and then returns the result, is:

```
mult(add(x,y),y)
```

The syntax of our programs should be well-defined, and we should take care not to create badly constructed programs. For instance, according to the rules defined earlier the program:

```
mult(x)
```

violates the rule that `mult` should get precisely two parameters and is therefore not acceptable. We should make it impossible to construct such a program from a population of syntactically correct programs. Since we construct new programs with genetic operators, we should take care that we use syntax-preserving operators.

The crossover operator is an operator that's not obviously syntax-preserving. If we just cut two correct programs in half, and reassemble them with their tails exchanged, we can't expect to yield two syntactically correct programs. The crossover operator with GP is therefore implemented differently from the crossover with GAs. Programs are viewed as tree-like structures, wherein each node contains a function and each leaf a terminal. For instance, the two syntactically correct aforementioned programs are expressed in the trees in figure 19.

If we use the crossover operator on these two programs, we remove one of the subtrees from each program and exchange these subtrees. For instance, we could

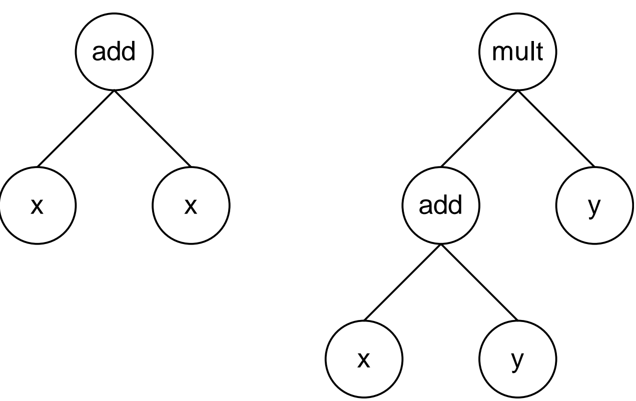


figure 19: the programs `add(x,x)` and `mult(add(x,y),y)`.

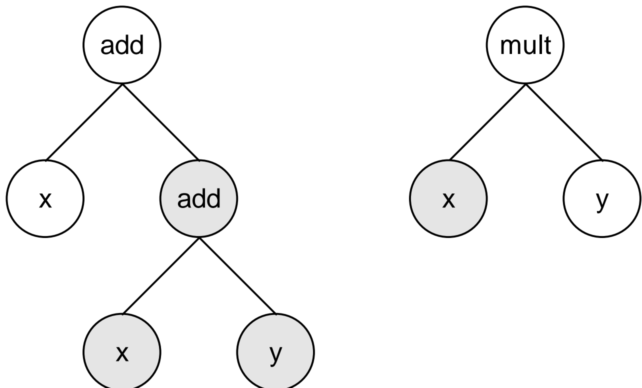


figure 20: the two programs from the previous figure after a crossover operation. The two shaded parts are exchanged.

exchange the subtree containing only the rightmost `x` from the first program with the subtree which has the function `add` as its root from the second program, to create the two programs in figure 20. In this way we always get syntactically correct programs if we start out with syntactically correct programs.

Now it's cleared up how programs are coded so that genetic operators can work on them, and how those genetic operators work, we're still lacking on crucial element for GP: a fitness function. The fitness function has to be defined so that it rates a program according to its success in performing a given task. A much-used technique is the application of *fitness cases*.

The effect of a program is the conversion of some input to specific output. The collection of fitness cases is a variety of inputs for which the desired output is known. The program is run with each of these inputs, and according to how close the output comes to the desired output, a fitness measure is determined for each case. The average of these fitness measures is the fitness function.

Summarising: just like with GAs, GP starts with an initial, randomly created population of individuals, in this case programs. In each iteration a new population of individuals is created, by using genetic operators like reproduction, crossover or mutation, which use a fitness function defined by a number of fitness cases. At the end of the run, the program with the highest fitness is presented as the solution or approximate solution to our programming problem.

The differences between GAs and GP are the following:

- With GAs, the genetic material is mostly structured linear, while with GP the genetic material is almost always structured tree-like.
- With GAs, the genetic material mostly has a fixed length, while with GP the genetic material is almost always of a variable length.
- With GAs, the individuals in the population are just data, while with GP the individuals are executables.
- With GP, we make use of syntax-preserving crossover. With GAs syntax-preservation is mostly of no concern.

Koza's choice for a computer language to use with his GP experiments is LISP. LISP is a language constructed specifically to make programs equal to the data they process, and LISP programs have the needed tree-like structure. This makes LISP an ideal choice for GP.

2.4 Hybrid systems

The most common interpretation of the term *hybrid system* when talking about GAs, is when a GA is combined with a conventional search technique to solve an optimisation problem (the other interpretation is when a GA is used with techniques which are non traditionally coupled with GAs, like real-valued chromosomes - however, that use of the term hybrid systems is only encountered in early GA literature). Often a GA is used to scan the solution space quickly, and then the conventional technique finds the optima in the regions that were selected by the GA. This two-phase optimisation is schematically presented in figure 21.

Besides the choices that have to be made for the hybrid system concerning the parameters of the GA and the local search technique, an important decision that has to be made is the moment that the global search is halted and the local search begins. Good results are achieved by starting the local search at the moment a cycle of the global search fails to improve the maximum fitness by a certain, small percentage. Other possibilities are simply the number of generations or the development of isolated clusters of individuals.

The REsearch Model Optimization package (REMO) is a tool designed by Michael Syrjakow and Helena Szczerbicka which implements such a hybrid system (Syrjakow 1995). Besides a GA, the global search can be done with a Monte Carlo technique or with simulated annealing. The local search is done with pattern search.

REMO also supports a multiple-stage optimisation, in which several optima are searched by running the system multiple times. To make sure the system cannot end up at the same optimum twice, the fitness is adjusted negatively around optima already found. The system can make use of populations already developed in earlier stages. The system keeps looking for new optima, until no new optimum can be found anymore within a certain number of cycles.

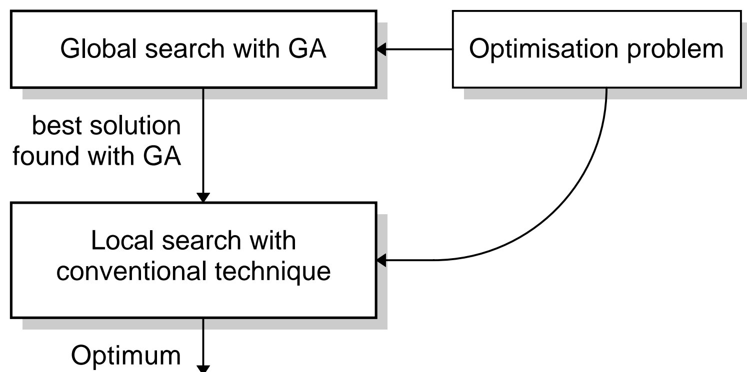


figure 21: Two-phase optimisation using both a GA and a conventional search technique.

2.5 Gaspipe networks

In this and the following two paragraphs I will focus on some examples of real-world applications of GAs. The first is an example of a fairly straightforward optimisation problem: the design of an efficient gaspipe network. This example is taken from a paper by Nicholas J. Radcliffe and Patrick D. Surry (Radcliffe 1994).

A gaspipe network is a network of gaspipes connecting several nodes. There are source nodes, which supply gas to the system, and demand nodes which distribute it. There are two aspects to such a network: the route that the pipes follow and the diameters of the different pipes in the network. The route is no problem, since it normally follows the streets in a city because easy access is needed. The pipe diameters to be used, however, are of some concern.

Thinner pipes, which have smaller diameters, are preferred because they are cheaper. The network as a whole should be able to withstand pressures at or above a minimum required level. Thicker pipes can withstand higher pressures, and may therefore be needed. An engineering constraint is that every pipe in the network has at least one pipe of equal or larger diameter upstream. The problem is to find a valid network that is as cheap as possible.

A standard heuristic for solving the gaspipe network problem is by assuming a constant pressure drop over the whole network, and guessing some initial pipe sizes which may give a valid network. The network is locally optimised by reducing pipe sizes while keeping the network valid, until no further pipe diameter reduction can be achieved. This approach has been programmed on small computers, and leads to quite good network designs. The time such a program needs to solve the problem is typically just a few seconds.

British Gas had to design a gaspipe network for a collection of 25 nodes, two of which were source nodes and 23 were variable demand nodes. There were six diameter sizes possible for each pipe, and 25 pipes in the network. This leads to a solution space

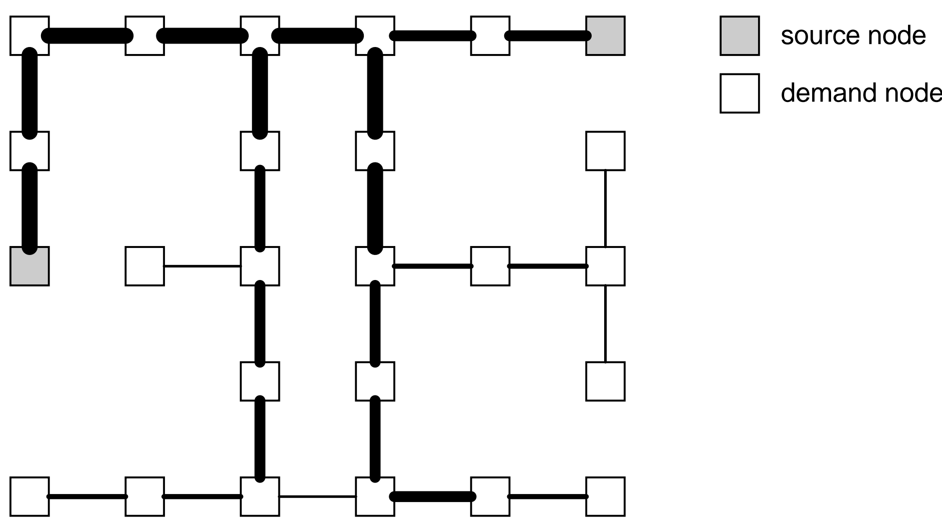


figure 22: The gaspipe network as designed with a heuristic approach. The thickness of the lines represents the diameter of the pipe used. The three pipe sizes with the largest diameter are shown as the thickest lines

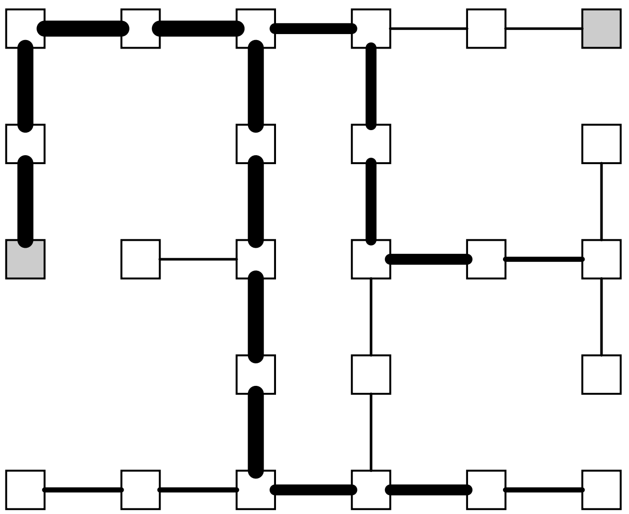


figure 23: The gaspipe network designed by a GA. It scores about 4% better than the network designed with a heuristic approach.

of $25 \cdot 10^6$ possible configurations (of which the largest part consists of non-valid networks). The heuristic approach used by British Gas lead to the network shown in figure 22. This network was actually installed.

At the same time, British Gas was working with the Edinburgh Parallel Computer Centre (EPCC) on a genetic approach for this problem. They used the Reproductive Plan Language 2 (RPL2), a language designed to facilitate the design and use of evolutionary techniques. RPL2 is a product of EPCC.

The chromosome used had 25 genes representing the 25 pipes, and each gene had six different allele values, representing the pipe diameters. To determine the fitness of an individual, a cost function was used, which simply summed the costs of the pipes in the network. The smaller the fitness value, the cheaper and therefore the better the network. The constraints for minimum pressure and upstream pipe size were introduced in the form of a penalty. This penalty was gradually increased as a function of the generation number. A fairly conventional GA was used, with elitism to preserve the best network. Population size was about 100, and the number of generations also. Parallel approaches were tried, but the results were found not to be significantly different from the standard, panmictic approach. Each run of the GA took a few minutes. This is considerably more than the heuristic, but not much if you consider that about 10000 networks are evaluated, while the heuristic evaluates just a few.

The GA was run several times, and produced consistently good results, although it did not always provide the same solution. Almost always the network which resulted at the end of the run was valid, and almost always it was better than the network designed with the heuristic approach. The best network found, shown in figure 23, was about 4% cheaper than the network that was actually installed.

This project convinced British Gas that GAs could be a valuable technique to be used on real business problems. They intend to use them on a number of other problems in the nearby future.

2.6 Business modelling

This paragraph describes an example of the use of GAs in business modelling. The example is taken from a paper by E.W. Haasdijk et al. (Haasdijk 1994).

A *model* is an abstract relationship between a current and a future state of affairs. Therefore, a model has the power to predict the future under the condition that the current state of affairs is known. Not all models perform reliably; in fact, very few models manage to predict the future without error. This is no surprise, since a perfect model for something would be equal to that something. Usually a model needs to be a simplification. Who wants a map on a 1:1 scale?

In business models are very important. For instance, if someone wants to do a direct-mail campaign, and there is a reliable model which predicts who will respond positively to this campaign, one might save a lot of money by just mailing the people that will respond, skipping those that just consider it junk mail. In banking, models are used to predict if someone who applies for a loan, will pay it back in time. If too many loans are given to people who'll never be able to repay, the bank has a serious problem, and that's why it's essential the bank uses a reliable model for trustworthy loan applicants.

There are many techniques to generate models, for instance neural networks and statistical methods like regression analysis. These techniques use a bottom-up approach: the model is constructed on the basis of examples. Statistical methods are the most common, and have the advantage of a solid theoretical basis. There are, however, serious problems if the data is incomplete, or if the best model to be constructed is not in accordance with some presupposed, for instance polynomial, relationship.

Models are constructed from data. Databases available to businesses are not ideal. Some common problems are too few or too many cases and inconsistent, incomplete, dynamic or even distorted data. This means models which use these databases have to be robust to deal with the inconsistencies, blank spots and errors in the database. Also, a model should be constructed and evaluated in a very short period of time, a matter of days instead of months, in order to deal with dynamic data.

Suppose we want to design a model that selects data from a set that falls in a certain category, for instance the category of those people that will respond to a direct mail campaign. Every data element in the set that falls into the selected category is called *good*, while every other element in the set is called *bad*. The model rates all data elements as good or bad. After the model has rated the data, we have four kinds of elements: Gg are those elements that are good and that are rated good by the model, Gb are those elements that are good but that are rated bad by the model, Bg are those elements that are bad but that are rated good, and Bb are those elements that are bad and that are rated bad.

One common way of measuring the success of a model is *accuracy*. Accuracy tells what percentage of the rated data is placed in the correct category. Accuracy is a scoring percentage and is simply the result of the following formula:

$$\text{(9)} \qquad \text{Accuracy} = \frac{Gg + Bb}{Gg + Gb + Bg + Bb} \cdot 100\%$$

Although at first glance accuracy seems to be a pretty good rating mechanism, it is in

fact very misleading. A database often does not contain 50% bad and 50% good cases. Suppose the database contains 95% good cases. The model "always rate as good" scores 95%, although this model uses no input whatsoever from the data. If we use this model on another database in which only 5% of the cases are good, the accuracy drops immediately to 5%.

An alternative measuring rate is the *Coefficient of Concordance* (CoC). It tells how well the model *distinguishes* between good and bad cases. The formula is:

$$\text{(10)} \qquad \text{CoC} = \left(\frac{Gg}{Gg + Gb} + \frac{Bb}{Bg + Bb} \right) \cdot 50\%$$

The model "always rate as good" scores 50% according to the CoC *on any database*. This is the same as a model which assigns good and bad ratings randomly. The model is in fact random: it selects the rating of a case randomly to be good or bad, with a chance of 1.0 to select good, and 0.0 to select bad. 50% is sensibly the lowest possible score, because if a model scores less than 50%, we could reverse the model to get a score higher than 50%. Such a model is called *cross-wired*.

In the course of the ESPRIT III project PAPAGENA (PArallel environments for PArallel GENetic Algorithms) the OMEGA system has been developed to deal with the problem of designing business models. OMEGA is currently available for several business applications, but it's aimed mainly at marketing, credit and insurance management.

OMEGA is a very flexible tool. Besides evolutionary methods it also encompasses statistical methods, to which it switches over if the evolutionary methods don't produce good results (although it is said that this almost never happens). It supports several different evolutionary techniques, database design, data analysis and model analysis. The end result is that OMEGA will propose a set of promising models. It should be noted that OMEGA translates non-numerical data automatically into numerical data.

OMEGA has been used in several real-world cases. One of those is the design of a response scoring model for use in marketing loans for a British bank. The client wished to reduce mailing costs by 20%. Since they used to mail every customer in the database, it meant they needed to select 80% of the customer base to be mailed, and these 80% should contain virtually all respondents.

The database used contained typical data on the bank's existing customers, like account balance, account history, services rendered to the customer, demographic location and household type. Account balance and history are, for a bank, clear and exact data, while most other data are not known for every client, sometimes out-of-date and sometimes simply incorrect. This makes conventional techniques difficult to use and therefore a GA the logical choice to analyse the problem.

The chromosome used needed to present a formula, which works with a selection of the data to get to a cut-off between good and bad cases. Koza's tree-like structure was used, with operators in the nodes and numerical data in the leaves. The CoC was chosen for the fitness function. OMEGA supports many different configurable GA parameters, and supposedly the default selection was chosen.

In a single run, OMEGA developed a model that was in every aspect better than the model developed by the bank specialists with conventional means. It had a better

predictive power, it was more robust and it took only minutes to develop, which is significant since it takes weeks to develop a model with conventional methods. This was one of the cases which proved OMEGA's usefulness. Currently OMEGA is used in several banking projects in Great Britain.

2.7 Neural network architectures

This paragraph describes an example whereby GAs are used to generate a suitable artificial neural network (ANN) architecture. The example is from a paper by Steven A. Harp and Tariq Samad (Harp 1991).

An ANN is a simulation of a very simple brain. The network consists of a series of interconnected neural nodes. Some of these nodes are input nodes, some are output nodes, and there may be hidden nodes, which are internal to the system. The input nodes can be used to present some data to the network. This data is processed by the network. The output nodes show the result of the process.

The network is trained by presenting data to the input nodes for which the expected output is known. If the result is close to the expected result, the connections in the network which lead to this output are strengthened. If the result is far from the expected result, the responsible connections are weakened. This way, the network is trained to solve some problem.

There is an endless number of possible configurations for an ANN architecture. It depends on the problem which architectures are suitable and which aren't. The architecture design is a complex and noisy affair, which makes GAs a logical choice to help the designing process.

Harp and Samad have developed a system called NeuroGENESYS to synthesise appropriate network structures and values for learning parameters. Besides the obvious goal of designing ANN architectures, they also hope to achieve some insight in how suitable networks are constructed.

NeuroGENESYS starts with a population of randomly generated networks. These networks are automatically trained with a set of training stimuli. After the training, a second set of input and output parameters is used to test the resulting, trained network. This leads to an evaluation of the performance of each network, which is used as a fitness criterion. A GA is used to create a new population of networks from the old population, and the cycle resumes with the training of the new population.

The biggest problem in this project is the representation of the network structures as strings, to be used with the GA. Since a simple network needs less description than a complex one, it's obvious that the strings can't be of fixed length. However, crossover shouldn't create nonsense strings.

They decided let the strings consist of "areas". Each area describes a segment, for instance a layer, in the network. An area has a fixed-length header describing the construction of the area, containing data like the number of nodes. The header is followed by zero or more fixed-length "projection specification fields" (PSFs), each PSF describing the connections from the area to some specific other area. There are markers placed between the areas, and other markers between the PSFs.

The genetic operators used are reproduction, crossover, mutation and elitism.

Crossover is only allowed to cross strings on positions which are the same distance from equivalent markers, to preserve a valid network description. However, a valid structure does not necessarily describe a valid network. For instance, it is possible to get a network that consists of several separate parts, or without a path leading from the input to the output nodes. Because the training mechanism used is backpropagation, recurrent paths in the network are also not allowed. The invalid networks were filtered out after they were generated.

The performance of a network can be geared towards a number of parameters. Accuracy may be deemed the most important, but the minimisation of the fan-out of each node, the number of nodes in the network, the learning speed or a trade-off between some of these parameters can also be selected.

Three example problems used to test NeuroGENESYS were the training of a neural network to recognise a visual 8x4 representation of digits, to perform an exclusive OR, and to approximate a sine function.

For the recognition of digits, much to the surprise of the creators, NeuroGENESYS managed to come up with an architecture without any hidden layers, which solved the problem perfectly. To solve the exclusive OR several small, well-performing networks were found. For the sine function, when moderate accuracy was selected for the fitness measure, the resulting networks typically had one hidden layer with just two nodes. This is, according to literature, the minimal configuration which can give a crude approximation of a sine. If high accuracy was selected, very intricate, multilayer networks, that gave a surprisingly good approximation of a sine, were built.

This example is not the only way to approach the design of ANNs with GAs. The second part of this thesis will be dedicated entirely to this subject.

2.8 Other applications

GAs have been used for applications in many areas. Some examples are:

- *Biology*. GAs have been used to simulate populations, to investigate niche theory (Goldberg 1989) and to determine the structure of DNA samples (Davis 1991).
- *Computer science*. GAs have been used to develop clustering algorithms and to learn difficult boolean functions (Goldberg 1989). A very interesting application is in the field of data mining, the design of search functions to get specific information from a large and unstructured database (Stender 1994).
- *Game playing*. GAs have been used to play the iterated prisoner's dilemma and simple games, like hexapawn and poker (Goldberg 1989).
- *Modelling*. GAs have been used for geographical modelling, for instance finding the best locations for retail services, for the searching of favourable protein conformations, for economic predictions (Stender 1994), for missile evasion (Davis 1991) and for the evolving of 3D models for model-based object recognition systems (Kinnear 1994).
- *Engineering*. GAs have been used to solve graph-colouring problems, to design a job scheduling procedure, to design circuit layouts, to optimise pipeline systems (Goldberg 1989), to solve scheduling problems, to design networks, to steer a robot-

arm, to interpret sonar data, to optimise aircraft designs (Davis 1991), to control robot insects and to plan robot paths (Kinnear 1994).
- *Sensory processing*. GAs have been used for binary pattern recognition, image classification (Goldberg 1989) and natural language processing (Kinnear 1994).
- *Mathematical problems*. GAs have been used to examine the blind knapsack problem, the two-armed bandit problem, the optimisation of noisy functions (Goldberg 1989), the travelling salesman problem (Michalewicz 1994) and the cracking and co-evolving of random number generators (Kinnear 1994).

2.9 Tools

For those who wish to experiment with GAs, or use them in business, many tools are available. Basically, there are two different kinds of tools: *application oriented* tools, which are complete, workable, configurable systems, and *algorithm oriented* tools, which are libraries or programs delivered in source code form, which can be used to build a special purpose system. Here are a few examples.

GA Workbench

GA Workbench, designed and built by Mark Hughes from Cambridge Consultants Ltd., is an MS-DOS based program which is no more than a demonstration of the power of GAs. It can be used to give people a "feeling" of what a GA is and what it can and can't do. It encompasses a configurable GA, which is used to evaluate a "function". This function can simply be drawn on the screen by the user. GA Workbench is freeware.

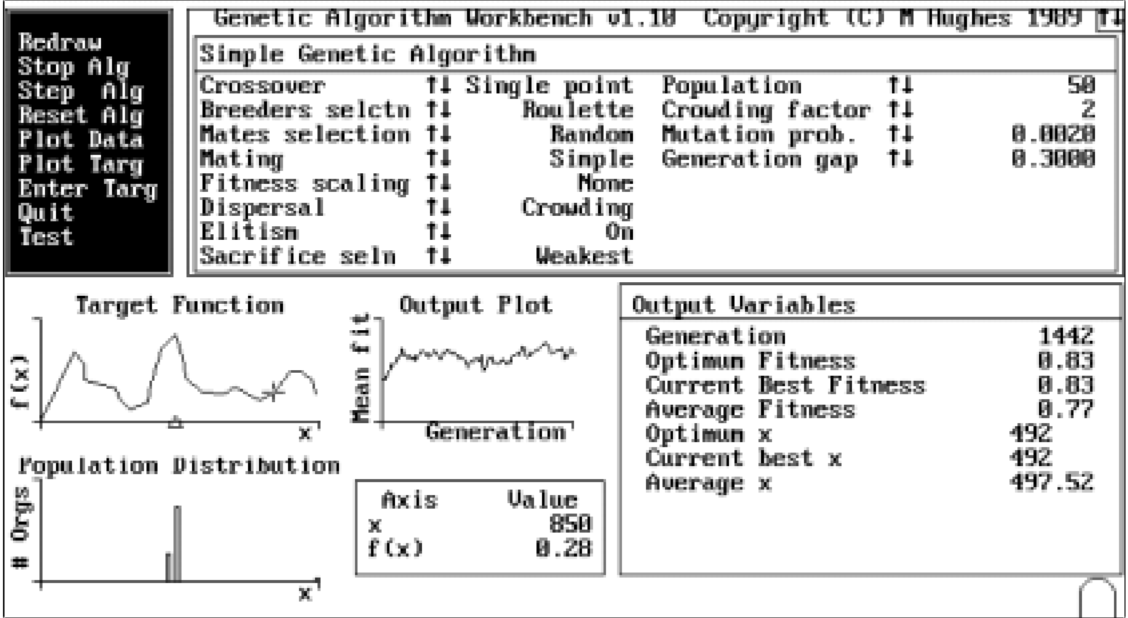


figure 24: The GA Workbench screen.

OOGA

OOGA, which stands for "Object Oriented Genetic Algorithm", is a well-known algorithm library which can be used to develop new genetic techniques. It is created by Lawrence Davis, mainly as a support for his *Handbook of Genetic Algorithms* (Davis 1991).

GENESIS

One of the first algorithm oriented tools was John Grefenstette's program GENESIS. GENESIS is currently available in the public domain in source code format, and can be used without any modifications on Unix and MS-DOS computers. The program is a general purpose GA. It incorporates the SGA, crowding, scaling, elitism, Gray codes and rank based selection. Virtually every GA parameter is configurable. GENESIS can be used to build a mature GA application, and has been used for this purpose in the past.

GAME

GAME stands for "Genetic Algorithm Manipulation Environment". It is a basic system developed in the course of the ESPRIT III project PAPAGENA. It's a flexible framework for the development and analysis of GAs and PGAs. It is meant to incorporate a wide range of genetic techniques, including SGA techniques in many varieties, inversion, breeding and parallelism. Another goal of the GAME system is to offer a way of parallelising existing GA systems, like Grefenstette's GENESIS.

GAAF

GAAF stands for "Genetic Algorithm for the Approximation of Formulas". It is based on the GAME system. It consists of a GA that searches for a formula that accurately represents the relationships in a set of examples, in short, which builds models. GAAF is a generic tool, which can be used to develop more specific applications.

OMEGA

OMEGA, marketed by KiQ Limited, is a full-grown GA application, geared towards the financial market and currently in use by several UK banks. OMEGA is based on the GAAF system. It is used to develop predictive models, and supports many steps in the design process, including the analysis of the data and the evaluation of the results. An example of an OMEGA application is given in paragraph 2.6.

2.10 Summary

There are numerous problems which can be approached with GAs. Foremost are function optimisation problems, which were examined intensively by De Jong. He found that many different kinds of functions can be optimised with a great variety of GAs, but that multimodal functions needed crowding to get good results.

Holland introduced classifier systems to use GAs to generate a collection of rules to solve a problem. The inner workings of CSs are a lot like neural networks, except for the fact that a CS provides us with explicit rules, while the rules in a neural network are hidden.

Koza introduced genetic programming, a technique to evolve programs with the same means as GAs use to solve optimisation problems. To be able to use crossover with programs, the chromosomes need a tree-like structure. A fitness function is defined on a set of inputs, on which the generated programs are tested.

Hybrid systems can be used to combine the power of GAs with the power of conventional optimisation algorithms. GAs are used to scan the solution space quickly, while conventional algorithms can be used to get to the optimum in an area selected with a GA in an efficient way.

Some example applications for GAs are the optimisation of gaspipe networks, the creation of business models and the design of neural network architectures. There are many other application, for instance in the areas of biology, computer science, game playing, modelling, engineering, sensory processing and mathematics.

Several tools are available to experiment with GAs. There are libraries and programs which can be used to build a new GA system, and also full-blown GA systems which are created for a specific purpose, like the evolving of financial models.

Conclusion of Part I

Overview

The first part of this thesis presents an introduction to the workings and possibilities of genetic algorithms (GAs). It is based on several books and papers.

GAs are search algorithms which are based on the principles of natural selection and natural genetics. A population of alternative solutions for a problem is processed through several generations, ultimately leading to a generation which contains highly fit solutions, that is, solutions which solve the problem at hand or which are at least a very good approximation of the required solution.

GAs are in their purest form applicable to optimisation problems. A lot of problems can be formulated as an optimisation problem and so GAs can be used to approach them. The GA principles can be applied in many other ways, for instance in combination with conventional search techniques. GAs have been used in almost every imaginable field, and often with success.

However, there is not just one GA, which can be applied to every possible problem. A GA has to be designed, using a selection of many different genetic operators and many different parameters. Also it's necessary to design a suitable coding for the alternative solutions to the problem, which is certainly not trivial. Therefore, the use of GAs is often a difficult matter, for which there are some guidelines but for which there is no recipe which always leads to success. In fact, the principles of GAs are such that success is never guaranteed. However, for problems for which there is no conventional technique to solve them, GAs can be a suitable approach.

Personal views

I'd like to conclude this part with a few personal thoughts and ponderings about this material. Note that these are my own, personal views, which are based mainly on the preliminary experiences I got with GAs after my exploration of the GA literature.

If you study GA literature, you might get the impression that there isn't any area

wherein GAs can *not* be applied. Just name your problem, and a GA specialist can design a GA which attacks it. Because of the generality of GAs, this is not surprising. However, in my opinion it is also cause for concern.

I'm afraid people tend to forget about the weaknesses of GAs. The most blatant of these weaknesses are the facts that GAs are terribly slow in comparison with conventional search techniques, and worse, that in most cases you cannot be sure that the solution found with a GA is indeed close to an optimal solution.

As soon as a problem is defined, it is possible to run a GA for this problem. Without analysis of the problem beforehand, however, it will be impossible to see if the GA solution is any good. This reminds me of the proverb, "Give a child a hammer and it will see every problem as a nail". Most problems are *not* nails, and to approach those problems with a GA-hammer will lead to inferior solutions, *without people realising that the solutions are inferior*.

How, in my opinion, should GAs be used? It may sound a bit corny, but I think the best approach is, as always, to start with an analysis of the problem. It must be shown specifically that GAs are a suitable approach to the problem at hand. This means that it should have many, if not all of the following characteristics (this list is built by me personally, and it may not be complete, but it's as complete as I can get it):

- If the GA works with data, the data domain should be very large, noisy, incomplete, inconsistent and/or divers. If not, an exhaustive search of the domain will often be more applicable.
- The solution space should either be very large, or, if a general technique is required (as is often the case), should increase exponentially with the size of the problem. If not, an exhaustive search of the solution space will often be more applicable.
- No suitable (efficient) conventional technique to attack the problem should be known. If there is, there is no reason to use a GA.
- It should be possible to algorithmically decide, when presented with two random different solutions, which one is better, and even "how much" better. If not, it won't be possible to define a fitness function.

Furthermore, I think GAs should not be used thoughtlessly. This means that not only the problem should be analysed according to the above-mentioned characteristics, but we should also have some expectations of the performance of the GA and a way of judging the results. The following (admittedly rather trivial) list contains what I think are important aspects of the GA designing process:

- We must know what our objective is. Do we want to generate an acceptable solution quickly, or do we want an optimal solution? Do we want just one solution, or a whole class of them? If we don't decide this beforehand, we can't decide if the GA results are acceptable.
- With the solution characteristics in mind, we should carefully design the GA (this sounds trivial, but I have the distinct impression this is in practice not always done, since badly designed GAs often also work, although much slower). We must decide on GA parameters, like population size and number of generations. We must decide on the genotype, like the number of chromosomes, the structure (linear or a tree-like) and the alphabet. We must define a fitness function which can be calculated

from the genotype. We must find genetic operators which work well on the genotype. Important is that this designing process should be done after we've decided on our objectives.

- If in any way possible, we should find a way to test the success of the GA, for instance by running it on a problem with the same characteristics for which the answer is known. If we don't do that, we cannot judge the success of the GA and therefore it's impossible to say if we've solved our problem or not.

After the GA has run, we should analyse its performance. If we didn't get the results we expected, we should try to find out what went wrong, and only then redesign the GA and try again. Probably our thoughts about the original problem were in error, and we should reconsider the use of GAs.

Some experimentation with genetic operators should be allowed in the first stages of a GA design, because there's still a lot of uncertainty about the effect of many of them. Some of them just "seem to work". Goldberg says that therefore the GA design is a trial-and-error process (Goldberg 1989). My personal opinion is that this may be true, but it shouldn't be used as an excuse to make the GA design a random process.

Precisely because there is no sound theoretical basis for GAs, should we use GAs in a disciplined manner. There are hundreds of different genetic operators, and an infinite number of combinations of operators, chromosome configurations and GA parameters. If we just try some of these combinations without thinking, a random search through the solution space may get replaced by a random search through the GA configuration space, which won't lead to better results but which will take a lot more time.

Only by analysis of the problem beforehand, a defensible design of the GA and by analysis of the results after the GA has run, can we be confident that our solution is acceptable and therefore can we apply GAs in a useful manner and expand our knowledge of GAs.

Part II

Genetic Algorithms and Neural Networks in Control

"What we have to learn to do, we learn by doing."
--- Aristotle

"When results are all that counts, the best solutions survive."
--- E.W. Haasdijk, *Genetic Algorithms in Business*

3 GAs and Neural Networks

This chapter is about the usage of GAs to design artificial neural networks (ANNs). ANNs are a widely known technique in artificial intelligence, and because of that it seems to be unnecessary to delve deep into the theory of ANNs. However, a short overview of the most important aspects of ANNs is given at the start of this chapter (3.1). This is followed by a survey of the areas where GAs have been applied in the design of ANNs (3.2). The problem of competing conventions, which most GA designs struggle with, is explained (3.3), followed by a general discussion of the design of a GA for ANN evolution (3.4). After that, some examples of applications are given (3.5).

3.1 Artificial neural networks

The technological field of artificial intelligence (AI) aims towards building machines (which almost always means the programming of computers) which have the ability to process knowledge and to reason about it. The three main approaches are expert systems, artificial neural networks (ANNs, sometimes NNs) and evolutionary systems.

Expert systems are rule-based systems, for which the rules are mostly generated by interviewing experts in a certain field. They won't play a role in this thesis. Evolutionary systems are rather new. Foremost of these are GAs, which are extensively examined in the first part of this thesis.

ANNs can be viewed as the simulation of a very simple brain. This brain is trained to perform a certain task, by exposing it to test-inputs, examining the output it generates, and rewarding it or penalising it according to the rate of success of the output. The purpose of this training is to get the ANN into a configuration wherein it can generate highly successful responses to all kinds of inputs, even inputs for which it has not been specifically trained (although the inputs cannot differ too much from those in the training set).

The biological basis of ANNs, or rather, the idea behind the theory of ANNs, is the structure of a brain. Oversimplified, a brain consists of cells known as *neurons*, which are interconnected. Each neuron can get 'input' from a number of other neurons, or

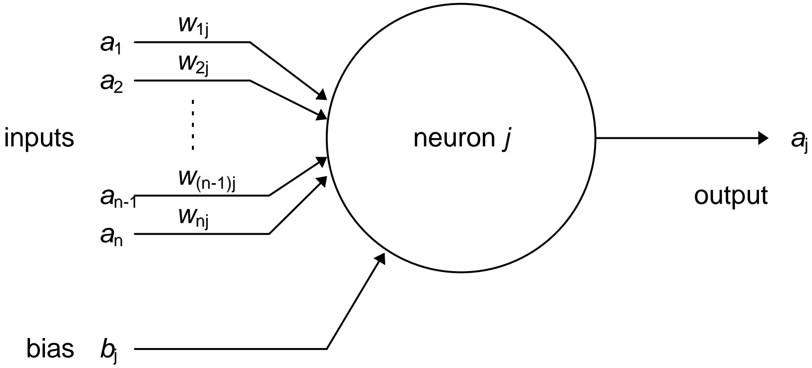


figure 25: An artificial neuron, used in artificial neural networks.

from external stimuli. If the total input is stimulating enough, the neuron 'fires', sending a signal to the neurons for which itself is one of the inputs. A neuron may also be connected, in combination with several other neurons, to an 'output device', for instance, a part of the body to which the brain belongs. This way, the output device can be controlled.

An ANN consists of a network of *artificial neurons* or *neural nodes*. A basic model of one such neuron, devised by McCulloch and Pitts (Aleksander 1990), is shown in figure 25. Neuron *j* receives inputs a_1 to a_n over connections with weights w_{1j} to w_{nj}. It also receives one extra input, known as the *bias*, which is a constant. It calculates the output a_j for this neuron, using a function *f*, by way of the following formula:

(11) $$a_j = f\left(\left(\sum_{i=1}^{n} w_{ij} a_i\right) + b_j\right)$$

The function *f* is known as the *activation* or *transfer function*, and it normally is the same for every neural node in the network. A common activation function is the *sigmoid*, which is shown in figure 26. The formula for the sigmoid is:

(12) $$f(t_j, \alpha) = \frac{1}{1 + e^{-\alpha t_j}}$$

where t_j is the total input to neuron *j* (which is the parameter given to *f* in formula 11), and α is a constant, which determines the slope of the sigmoid and therefore the *learning rate*.

An example of an ANN is shown in figure 27. For clarity, the biases are left out. The net consists of *layers*. The leftmost group of neurons, which get their input from the environment and not from other neurons, is known as the *input layer*. The example shows an input layer which has three neurons. The rightmost group of neurons, which present their output to the environment and not to other neurons, is known as the *output layer*. The example shows an output layer which has two neurons. Between the input and output layer, there may be other layers known as *hidden layers*. The example

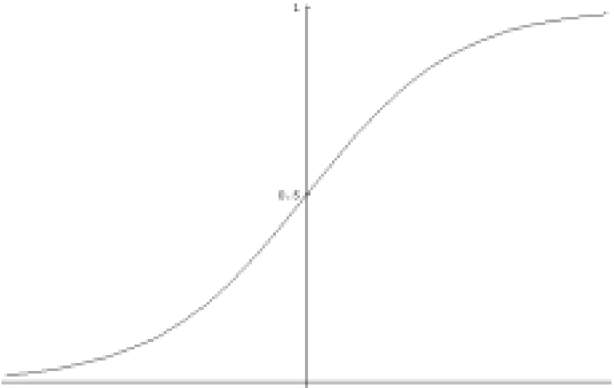


figure 26: The sigmoid activation function.

shows two hidden layers, one of which has three and one of which has four neurons.

In the example, every neuron in a layer is connected to every neuron in the next layer. We say that this network is *completely connected*. Also, every neuron is connected only to successive layers, but not to preceding ones. This network is therefore called a *feedforward network*. If there are only connections from the preceding layer and to the next layer (and not from *any* preceding layer and to *any* successive layer) it is called a *layered feedforward network*. This is the case with the neural network in the example. Note that the name “feedforward network” is often used when in fact a layered feedforward network is meant. If the ANN *does* have connections to preceding layers, it is called a *feedback* or *recurrent network*.

The ANN is no more and no less than a *mapping* from all the possible inputs to certain outputs. Of course, we want this mapping to be useful in solving a problem. We present the problem parameters to the ANN, and we expect the output to be a representation of some sensible response to these problem parameters. To get the ANN into a state wherein it represents such a useful mapping, it has to be *trained*.

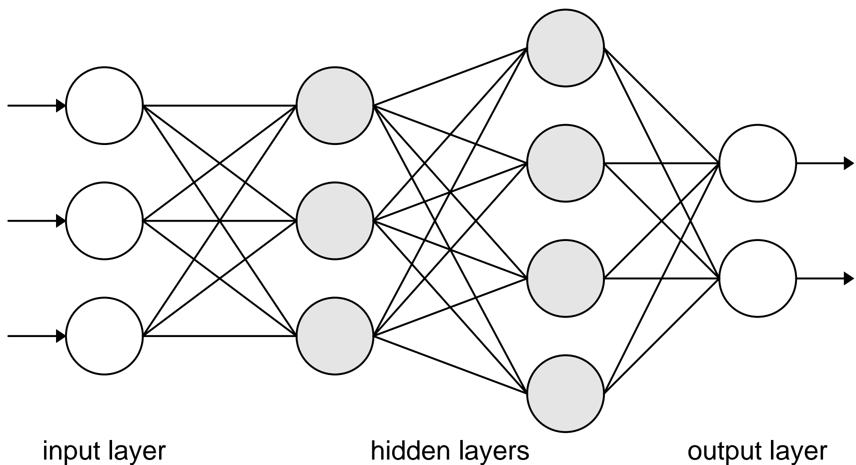


figure 27: An example of an artificial neural network. The hidden nodes are shaded.

One way of training an ANN is by presenting it with sample inputs for which the expected output is known, examining the output, and adjusting the connection weights according to the success of the output. For instance, if the response is far off the expected response, we could weaken the connections which were mainly responsible for this response. Or, if the response is good, we could strengthen the weights responsible. This is known as *supervised learning* or *supervised training*.

The most common procedure for weight adjustment in the net is the *backpropagation* algorithm. The error between the outputs achieved and the expected outputs, is propagated backwards into the network, and the weights are adjusted according to the error. Backpropagation is based on the *gradient descent* of the error function. Those interested in the exact workings of this algorithm are referred to standard literature, for instance Aleksander and Morton's book (Aleksander 1990).

Backpropagation works, but the following problems should be noted:

- Backpropagation is terribly slow. The samples have to be presented many times to the network before an acceptably small error is reached (if it is reached at all). The first few cycles, the error will normally be reduced a lot, but then the diminishing of the error quickly slows down to an almost imperceptible rate, and sometimes the error will even increase again.
- Backpropagation is based on gradient descent search. This makes it expensive, especially when gradient information is not cheaply available. The weight adjustment in the network takes a lot of time, and it has to be done for many weights, many times.
- Backpropagation will often end up with a network that is just a *local* minimum in the error space, while we desire to reach the *global* minimum.
- The success or failure of the backpropagation algorithm is very dependent on the initial weights in the network, the set of training samples and the chosen network architecture (larger is not necessarily better).
- Backpropagation only works well on feedforward networks (whether layered or not).
- Backpropagation can only determine the weights and not the architecture of the network (to be fair: architecture design has never been a task of the backpropagation algorithm).
- The activation function has to be continuously derivable.

The search for a successful ANN for a certain problem is a search for the global minimum in the space of the errors generated by all possible ANNs working on a set of training samples for this problem. There is an infinite number of possible ANNs we can take into consideration. The error space is noisy, since small changes in the weights of a network may affect the error enormously. The error space is multimodal, since for most mappings, there exist many different ANNs which represent them (more about this will follow later in this chapter, when discussing the competing conventions problem). There are methods to find useful ANNs, but these methods are expensive, don't work in every situation, and don't guarantee success. These are the arguments to take a look at GAs, to see if they can be of use in the design of ANNs.

3.2 Areas of GA application in ANN design

There are a number of areas where we might try to apply GAs in the design of ANNs. A fairly complete overview is presented by Xin Yao in his article *Evolutionary Artificial Neural Networks* (Yao 1995). Here is a list of the most important of them:

Connection weights determination

GAs can be applied in the *determination of the connection weights* in an ANN. As stated in the previous paragraph, finding a suitable set of connection weights by training is time-consuming, does not always succeed, and not all training algorithms are suitable in every situation. The application of GAs for the weight determination process may still take a long time, but GAs offer greater possibilities of finding the global minimum of the error function, and since GAs only need to know the relative fitnesses of the ANNs under consideration, we are less restricted by the network configuration.

Most often, GAs are used to find a set of connection weights for which there is a very small error. However, sometimes a GA is used to select a suitable *initial* set of weights, that is, a set of weights which leads to a successful ANN after training with some standard training routine. In the former case, the fitness of a network will normally be determined solely by the error. In the latter case, training time may also play a role.

GAs are competitive with other approaches in weight determination, especially where gradient information is difficult to obtain or the network architecture is very complex. However, when gradient information is cheaply available, other approaches may be more effective.

More about connection weight training will follow later in this chapter.

Network architecture design

The *design of the network architecture* or *topology* is another common area of GA application with ANNs. The network architecture is of great importance for the success of an ANN. For some problems, a big network is unavoidable, while for others smaller networks are more suitable. While one hidden layer is enough for most problems, sometimes a number of smaller hidden layers are helpful in getting good results. Some problems need recurrent networks. Not always a completely connected network is needed. The space of all possible networks is infinite, and as yet there is little or no theory about what architecture works well for what problem. This makes GAs a viable approach.

The fitness of a certain architecture is dependent on the success of network after being trained, and sometimes of the characteristics of the training process, like the time it takes. This means that a certain amount of training is necessary in determining the fitness of an architecture, which can take a long time if you realise that it has to be done for every individual in a population, in every generation.

It should be noted that the design of a network architecture is often combined with the search for a suitable set of connection weights.

More about network architecture design will follow later in this chapter.

Other areas

A few experiments have been done in the usage of GAs in the following areas:

- Node activation function design.
- The selection of inputs, for instances, with problems for which there is a large number of possible inputs, and for which it is unknown which inputs are needed by the ANN to solve the problem.
- The evolution of learning rules.
- The evolution of algorithmic parameters, like the learning rate.
- The analysis of ANNs.

A Framework

In his overview, Xin Yao (Yao 1995) presents a framework for viewing the different levels where GAs can be applied in ANN design. This framework is shown in figure 28. At the lowest design level (note that “learning tasks” is not a *design* level), on the fastest time-scale, there is the evolution of the connection weights. Connection weights can only be determined when we already have decided on an architecture and a learning rule.

Above the level of the connection weights is the evolution of the architecture, and above that the evolution of the learning rules. If we like, we may place the evolution of the learning rules on the second level and the evolution of the architecture above that. It depends on the prior knowledge we have of the problem. For instance, if we already know what learning rule would be suitable, we would place the learning rules on the highest level.

Either way, to work on any level, we need to have the results of the levels above. For instance, with the scheme of figure 28, before we can seek a suitable architecture,

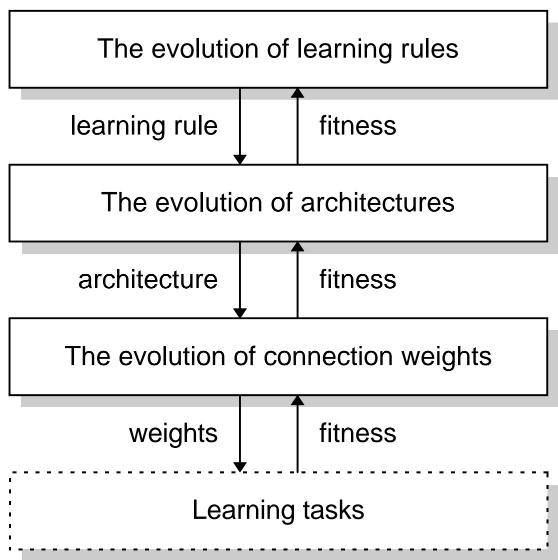


figure 28: A framework for the design of ANNs with GAs.

we need to have decided on a learning rule. Furthermore, to test the fitness of our solutions derived on a level, we need to examine the results when employing this solution on the levels below. For instance, to determine the fitness of an architecture, we have to seek the connection weights, since our success in finding these weights determines the fitness of the architecture. Therefore, the evolution of an architecture or a learning rule is generally more expensive than the evolution of connection weights.

3.3 Competing conventions

The problem

Before talking more in depth about the design of a GA suited for ANN evolution, a problem that always arises with the evolution of ANNs should be explained. This problem is known as the *permutation problem*, or, the *problem of competing conventions* (some authors call it the *problem of isomorphism*, or, the *structural/functional mapping problem*).

An ANN implements a mapping from an input to an output. To have competing conventions means, that for any specific mapping, there are several structurally different ANNs which implement exactly this mapping. In practice, there can be a huge number of equivalent ANNs for every mapping. Thierens *et al.* (Thierens 1993) distinguish the following two symmetries:

- *Hidden node redundancy*. If, which is often the case, a neuron sums the weighted inputs and applies an odd activation function (for instance linear threshold or a hyperbolic tangent) to the sum to produce the output value, the output of the total network doesn't change if we flip the signs of all the incoming and outgoing weights (including the bias, if any). An example is shown in figure 29. Since for every node in the ANN there are two possibilities, for the ANN as a whole, if there are n hidden nodes, there are 2^n different combinations.
- *Hidden layer redundancy*. If a hidden neuron, with all its incoming and outgoing connections, is exchanged (in the representation, of course) with another neuron with all its incoming and outgoing connections, we have a different *structural* representation of the ANN, but *functionally* the ANN stays exactly the same. An example is shown in figure 30. In a network with n hidden nodes, there are about $n!$ different combinations of these hidden nodes.

Since these two transformations are independent of each other, for a network with n hidden nodes, there are $2^n n!$ functionally equivalent but structurally different representations, if the activation function is odd, and otherwise $n!$ different representations.

What makes competing conventions such a big problem? First, the existence of competing conventions dramatically increases the size of the solution space, which means it will take a lot more time to converge to an acceptable solution. More important, the existence of competing conventions almost completely destroys the

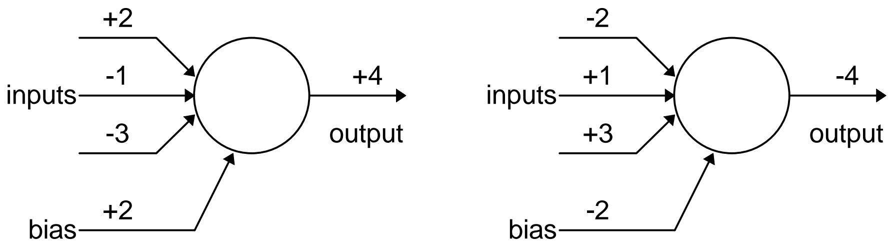


figure 29: Hidden node redundancy: two different but functionally equivalent neurons (in case of an odd activation function).

usefulness of the crossover operator. Since the crossover operator is arguably the most important operator in GAs, it is vital that we find a way to deal with this problem.

To see why the crossover operator gets impaired by competing conventions, suppose we have a population of ANNs with some quite fit individuals. These individuals are coded as strings. Since competing conventions can be in the population, it is possible that two very different strings represent the same ANN. This means that the diversity in the population is less than it appears. But that's not the worst part of it. If we cross two rather fit but not equivalent strings, we may, and will often, get two new strings which represent ANNs that are not at all fit.

For instance, suppose that each character in the string codes one hidden node, and the string “abcdef” codes the fittest possible ANN. In the population are the strings “abcdeg” and “hbcdef”. If we cross these two strings, we might get the desired ANN. However, the second string may reside in the population as, for example, “fedcbh”. Crossing may now leave us with something like the strings “abccbh” and “feddeg”, which may very well be not fit at all.

So, if the nodes that fulfil an equivalent function on two ANNs are not found in the same place on the chromosome, the features that made these ANNs fit won't find their way to the offspring when the crossover operator is used.

Solving the problem

Several ways of dealing with the competing conventions problem have been proposed. Here follows a list of some of them.

First, it should be noted that some researchers just ignore the problem of competing conventions, and simply use the crossover operator as always. There are reports that

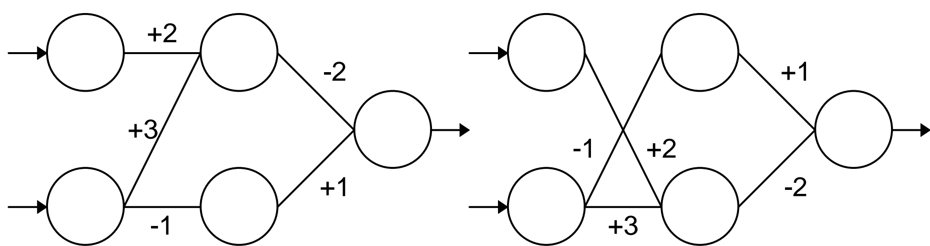


figure 30: Hidden layer redundancy: two structurally different but functionally equivalent ANNs.

this works fine, and that attempts to remove competing conventions lead to a vast decrease of performance and not to better results (Hancock 1992).

I've encountered only one method of dealing with hidden node redundancy: Thierens *et al.* (Thierens 1993) simply flip the signs of all the incoming and outgoing weights if there are more negative than positive weights. A small problem with that, which isn't addressed by the researchers, is that the outgoing weights of one node are the incoming weights of another node. If the first node has more positive than negative weights, and is already checked, the flipping of the signs of the incoming weights of the second node may change the signs of the outgoing weights of the first node, and therefore may leave the first node with more negative than positive weights.

To deal with hidden layer redundancy, many researchers have just left the crossover operator out, and evolve ANNs with reproduction and mutation only. Since that way the mutation operator is the only operator that can make changes in a chromosome, the mutation rate has to be set quite high. The GA process then becomes a kind of random hillclimbing (the change is in the neighbourhood and random, but the ANNs that are climbing uphill are more fit and therefore more likely to survive). This is sometimes called *genetic hillclimbing* (Schaffer 1992).

Another simple way of dealing with the problem of hidden layer redundancy, is using small populations. If the population is small, there can't be many competing conventions. However, crossover is of no use anymore, and this 'solution' is therefore often used in combination with the use of only a high mutation rate, strengthening the hillclimbing features (Schaffer 1992).

Some researchers only use the crossover operator if the parents are not too different. This is sometimes called *restrictive mating*. It is an easy way out, and the use of the crossover operator, which is designed to combine features from fit but totally different parents, is doubtful (Alba 1993).

The best way of dealing with hidden layer redundancy and the crossover operator, is to rearrange the hidden nodes in the parent ANNs, so that nodes with equivalent functionality are placed at the same positions on both the parents. Thierens *et al.* (Thierens 1993) suggest that the functionality of a node is determined foremost by the signs of the incoming and outgoing weights. They reshuffle the hidden nodes of one of the parents so that the signs of the nodes of the second parent match those of the first parent in the same position as closely as possible. This is done with a greedy algorithm, whereby the first nodes examined have a much larger chance of being matched with equivalent nodes than the latter ones. This suboptimal matching is justified by the great speed of the algorithm and the fact that it introduces some diversity in the population.

Other ways of locating functionally equivalent hidden nodes are similarities in the number of connections (if the network is not completely connected), using the Hamming distance between the coding of the nodes (if the coding is binary), and spin-offs of these criteria (Hancock 1992).

Cascade correlation is a well-known constructive algorithm of building ANNs by adding hidden nodes one at a time. If this is combined with the genetic evolution of the node to add, we have a GA for the evolution of ANNs which is free of competing conventions, since the part of the ANN that is already built is not changed by the GA, and competing conventions can only arise if more than one hidden node is involved. This is known as *genetic cascade learning*. Preliminary studies show it to perform quite well (Karunanithi 1992).

The ENZO (Evolutiver NetZwerk-Optimierer) system introduces a complex way of insuring that each functional mapping is implemented by the structural mapping with the shortest connection lengths, thereby hoping that only a limited number of configurations for some mapping can arise (Braun 1993).

Summing up, these are the categories of ways of handling the problem of competing conventions:

- Ignoring it.
- Abstaining from using the crossover operator, or using it a very limited way.
- Rearranging the hidden nodes in the parent individuals to place functionally equivalent hidden nodes in the same position on both parent chromosomes.
- Genetic cascade learning.
- Introducing a mechanism for forcing certain implementations of a mapping, for instance the mapping with the shortest path.

Rearranging the hidden nodes seems to be the most effective: it leaves all the aspects of GAs intact and is relatively easy to analyse and understand. The reason it is not always used is that it depends on the theory behind the determination of equivalent nodes of ANNs how difficult the method is to implement and how much time it will cost. Most methods are very time-consuming and quite difficult to implement.

3.4 The design of a GA for ANN evolution

GAs for the evolution of ANNs are often somewhat different from GAs in other fields. There is also some special vocabulary introduced with these differences. This paragraph will examine GAs for ANN evolution, including the coding, the fitness determination, the genetic operators, the GA parameters, an overview of the complete process, and special techniques.

This paragraph will focus on GAs which evolve connection weights and architectures, since these are the most common and most studied.

Encoding

If we want to code an ANN as a string, we can decide to code every aspect of the ANN, or just a few characteristics of the ANN. The first variety, which is known as *direct encoding* (sometimes called *strong* or *low-level encoding*), leads to large strings, in which every connection and every weight is coded. The second variety, which is known as *indirect encoding* (sometimes called *weak* or *high-level encoding*), just encodes a few aspects of an ANN, like the number of layers, the level of connectivity between the layers, and the weight-ranges. Another kind of weak encoding is related to GP, namely the coding of some rules which can generate an ANN.

It's difficult to get good results with indirect encoding, since small differences between ANNs can lead to very different results, and small differences aren't accounted for with indirect encoding. It can, however, be used to study classes of ANNs in general

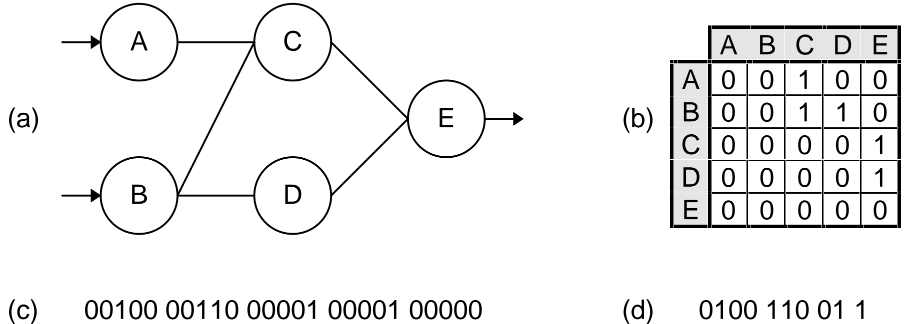


figure 31: A feedforward ANN architecture and its coding. (a) shows the network. (b) shows the matrix which codes this network. The vertical axis shows the nodes from which the connections come, and the horizontal axis the nodes to which the connections lead. The cells of the matrix contain the connection status, zero meaning that a connection is absent and 1 meaning that a connection is present. (c) shows a straightforward representation of this matrix as a string: the rows of the matrix are placed side by side. (d) shows another string, which codes the same matrix, taking into account that the network is feedforward. The lower half of the matrix, including the diagonal, needn't be represented in the string. With some knowledge of the layers, the code can be even shorter.

problem areas.

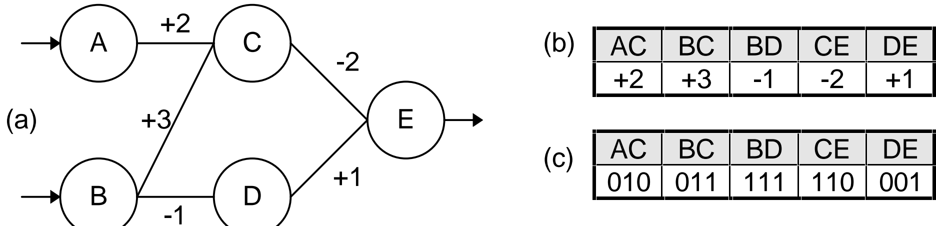
The greatest flaw of direct encoding is the fact that it leads to very large chromosomes. The larger the chromosome, the longer it will take for a GA to converge to a solution, and the more expensive the algorithm will be in terms of time. Most experiments which use direct encoding therefore concentrate on quite small networks. In practice large ANNs are more common and needed. It is therefore important to try to minimise the length of the chromosome as well as possible.

In case we want to code just an architecture, the most common direct way this can be implemented is as a matrix. If the ANN has n nodes, an nxn matrix is needed to express all possible connections (including connections from a node to itself), whereby each element of the matrix indicates if a connection is present or absent. This matrix can be represented by a string with n^2 bits. For an example, see figure 31.

A great advantage of this representation is that all kinds of ANNs can be expressed. There's no limit to the topological configurations. Of course, there are a lot of configurations which don't express viable networks. These are often discouraged by placing a penalty on the fitness. If we want to limit the possibilities, the matrix can often be simplified and thus the strings can be shorter. For instance, if we only want to create feedforward networks, we can immediately eliminate more than half of the elements of the matrix.

When we're searching for some architecture which is suitable for our problem area, we're not just interested in the presence or absence of connections; we're also interested in the number of nodes we need. Therefore the strings shouldn't be of a fixed length. This, of course, places some limits on the use of the crossover operator. An easy solution is to place a maximum limit on the number of nodes, and always use strings which can represent that many nodes.

Weight encoding takes a lot more space than architecture encoding. There are two ways of weight encoding: using binary values or using real values. If we use real values, we need only one position in the string to encode one weight, but real values are not suitable to change with the crossover operator. If we use binary values, we need more positions in the string to encode one weight. The number of different weights we can



(b)

AC	BC	BD	CE	DE
+2	+3	-1	-2	+1

(c)

AC	BC	BD	CE	DE
010	011	111	110	001

figure 32: Weight encoding. (a) shows the ANN with the connection weights. For pure weight encoding, the architecture is fixed. The strings therefore need only encode the weights of the connections which are present, and don't have to account for absent connections. (b) shows a real value weight encoding of this ANN. The string has five positions. (c) shows a binary value encoding of the string, using three bits to encode each weight, in a standard 2-complement binary system. The string has fifteen positions.

encode depends on the *granularity* we set, which means, how many different bits we use to encode a weight. For example, if we just need to encode the weights -1.0, -0.5, 0.0 and +0.5, we need no more than two bits, and therefore two positions, for each weight in the string. If we need to be able to encode all real values between -1.0 and +1.0 up to four decimals, we need at least fifteen bits to encode one weight.

A problem with this is, that we often don't know up front how fine a granularity we are going to need. If we choose a granularity which is too fine, we may converge to an acceptable solution, but it may take far too long. If we choose a granularity which is too coarse, we may not be able to find a solution at all.

An example of weight encoding is found in figure 32.

When combining weight optimisation with architecture design, a normal way of coding adds to each present connection the weight that is associated with it. This can lead, in the case we use real values for the weights, to a chromosome which combines real values with binary values.

Looking at the chromosome as a whole, the coding can be as simple as in the examples above, or very complex indeed. For instance, in the structured GA (sGA, not to be confused with SGA, which is the simple GA), the researchers use a multilevel coding, whereby genes at a high level can be turned on or off to activate or deactivate groups of genes at a lower level. Dasgupta and McGregor use the sGA to optimise both architecture and weights. The higher level, which forms the first part of the chromosome, codes the connectivity, and the lower level, which forms the second part, codes the weights and biases. They report good results (Dasgupta 1992). Other authors, however, hold that the existence of a connection and the weight belonging to the connection are functionally grouped, and should therefore be near to each other on the chromosome.

Sometimes researchers code extra information in the chromosome, like the length of the binary weight coding, the chance of crossover or the learning rate, with the objective to let these parameters also evolve.

Fitness determination

For weight determination, the most common way of determining the fitness is simply by decoding the chromosome and building the ANN, and then calculating the mean square

error (MSE) of the network by trying it on a sample training set. It's not different from determining the success of a network with the backpropagation algorithm. Of course, the smaller the MSE, the fitter the ANN. The training set which we use to calculate the MSE should be chosen carefully, which is exactly the same as when we're using the backpropagation algorithm.

If we're using a GA to generate an ANN architecture, or a combination of an architecture and the connection weights, again the MSE is taken to determine the fitness, but often this is not the only ingredient. The complexity and size of the network will be a factor, the needed training time will sometimes play a role, and if it is possible that networks arise that are not viable, a penalty function will be introduced.

All this, of course, can only work if it is possible to generate a training set. In neurocontrol problems this is most often *not* the case. This will be addressed in the next chapter.

Fitness calculation is pretty straightforward, but one should note that this part of the GA will normally take by far the most time of the whole process.

Genetic operators

When whole populations are replaced by a new generation, reproduction, the copying of existing individuals to the next generation, is often applied. However, sometimes new individuals are brought into the population by crowding or a technique reminiscent of that, for example, the replacement of the weakest individual of the population if the produced child scores better. Elitist selection is often used.

Mutation is almost always used, sometimes divided in several kinds of mutation, for instance one kind of mutation for the connection weights, and another kind of mutation for the connections. It is quite normal to use very high mutation rates. Some authors even rely solely on mutation to generate new ANNs, and in that case the mutation rate needs to be high.

These high mutation rates are, certainly in the case of the absence of any other genetic operator, in my humble opinion an admission of the authors that they couldn't get GAs to work well with ANNs. Relying exclusively on high mutation rates is normal in using another kind of technique, called evolutionary programming (EP) (Heitkoetter 1995). Since for the application of the mutation operator an individual doesn't need to be encoded, EP does not request encoding, and I think that researchers who use only mutations should first study EP. However, if real values are used to code weights, mutation is the simplest technique of changing the weights, and a mutation rate somewhat higher than normal with GAs should be accepted.

The problems with the crossover operator have been intensely discussed in paragraph 3.3. After rearranging the coding so that functionally equivalent nodes are located in the same position, there is no reason not to use crossover, at least, if the strings are of the same length. If they're not, they should first be extended to the same length (not only physically, but also functionally - the same number of nodes, the same number of bits for each weight, etcetera), before crossover can take place.

Whitley *et al.* suggest to use a form of adaptive crossover (Whitley 1993). They state that crossover works best in the earlier stages of the GA process, when there aren't that many fit ANNs in the population. Each individual gets an indication of the

chance to crossover, which is encoded in the chromosome. If the individual is used as a parent, this chance determines if the crossover operator will be used. If it is used, and the child produced if fitter than the parent, the chance to crossover gets decreased. If not, it gets increased. The aim is to lower the chance to crossover, which in the opinion of the researchers is bad on accord of the competing conventions problem, and increase the chance to mutation in latter stages of the process.

Strangely, there haven't been many experiments with reordering operators like inversion and PMX. The only reports I have found suggest they don't work well in this problem field (Alba 1993). One might think they could serve to bring nodes together that form a functional group.

Of course, many genetic operators have been designed especially for the evolution of ANNs. Some of these are also useful in other areas, while others are dedicated to ANN evolution. An example is the GA-simplex operator, introduced in the ANNA ELEONORA system (Maniezzo 1993). Montana and Davis have designed a whole range of operators suitable for ANN evolution (Montana 1989). These operators are discussed in more detail in paragraph 3.5.

GA Parameters

There's not a lot to say about GA parameters, because they are problem-dependent. A few general observations can be made.

Because of the competing conventions problem, and the long evaluation time of each individual, populations tend to be on the small side. In practice, populations of more than fifty individuals are rarely encountered.

The maximum number of generations criterion is often not specified, since there is a target error to be reached. In any case, even if a number of generations is specified, it should be possible to continue the process from the last generation onwards.

The process

Now the individual aspects of GAs for ANN evolution are examined, an overview of the process as a whole can be given. This is not very different from the standard GA process that is explained in the first chapter, and it will hold no surprises, but for completeness sake it is presented in figure 33.

Note that this is only a basic schema for the process of GA design of ANNs. For instance, in the schema only the error on the training set is used to find the fitness. As explained under the heading "Fitness determination" in this paragraph, the error is normally the main ingredient of the fitness, and very often the only ingredient, but there are other fitness criteria which might be used.

The biggest difference with the GA presented in figure 6 in the first chapter is the explicit decoding of every individual into the corresponding ANN for fitness determination, and the fact that the fitness is determined by testing the ANNs on a training set. This is not exclusive for GAs which design ANNs, since it is very usual that a chromosome gets decoded and tested for fitness determination (we even did it with the SGA in the first chapter). In this case, however, it may be a complex process, so it should be stressed that this is a separate and very important step.

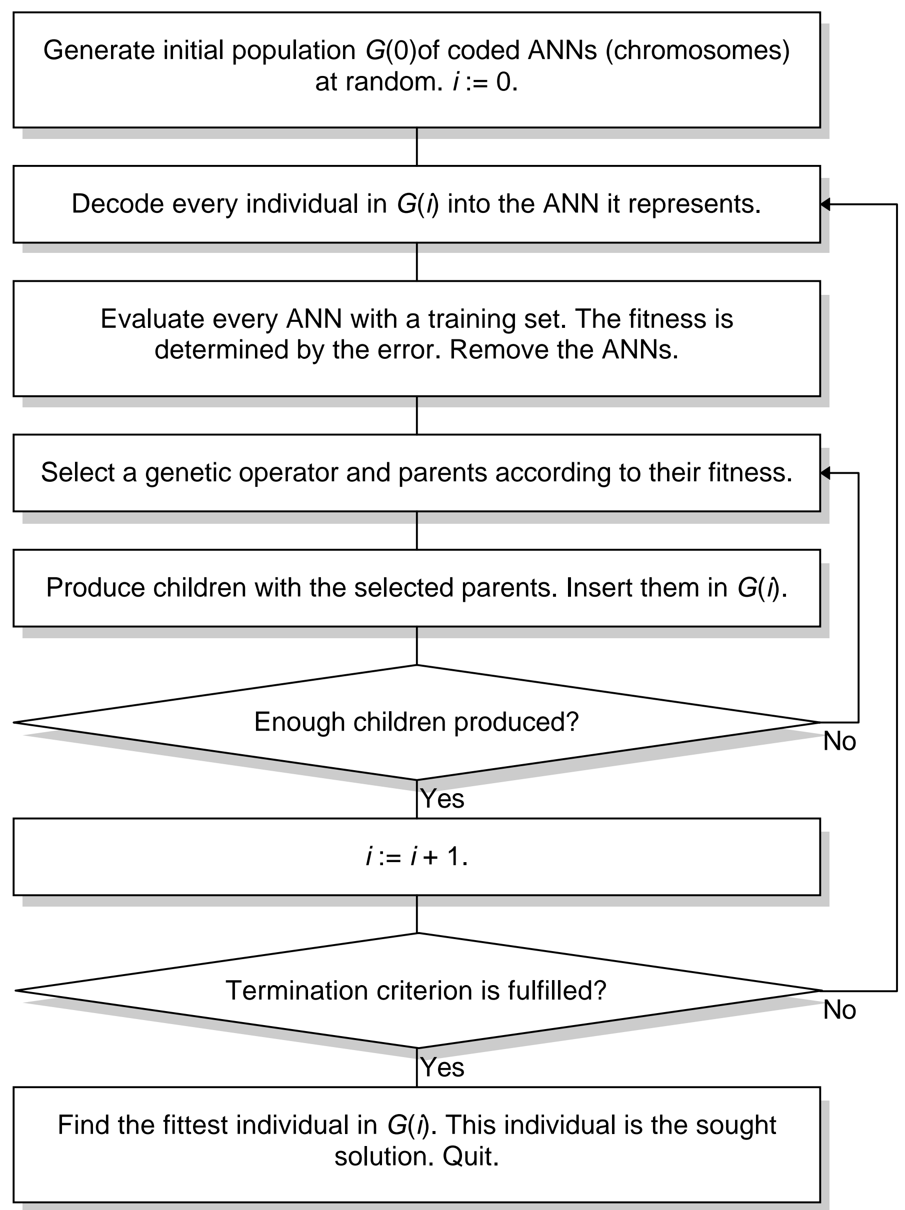


figure 33: A basic GA for the design of ANNs.

It should be noted that this schema can be much improved upon by taking into account the characteristics of the problem and the GA implementation. For instance, many genetic operators used with this kind of GAs use the decoded chromosomes, the ANNs, to generate a child. Examples of these are given in the next paragraph, with the experiments of Montana and Davis. It may therefore be useful to keep the decoded chromosome with the chromosome at all times, instead of removing it after evaluation.

Also, since the evaluation step is very time-consuming, and not every individual will be replaced in the next generation, we should consider skipping the step of calculating the fitness of *every* individual in the next generation. If the individual was also present in the previous generation, we still have the fitness of this individual, unless the training

set changes between generations, which will most often not be the case.

Special techniques

Very often the evolution process with a GA is combined with another technique, thus leading to a *hybrid algorithm*. An example is the use of a GA to globally explore the search space, and when promising ANNs have been found, the use of a local optimisation procedure to find the optimum. Backpropagation can be used as a local optimisation algorithm.

It is also possible to interchange GA runs with runs of a backpropagation algorithm. This will lead to faster convergence, which is sometimes advisable, but which is mostly harmful.

For local optimisation, instead of the backpropagation algorithm, other techniques can be used, like simulated annealing.

3.5 Evolution of ANNs in practice

Many experiments have been done with the evolution of ANNs with GAs. Just to get a taste of this field, some examples are related here.

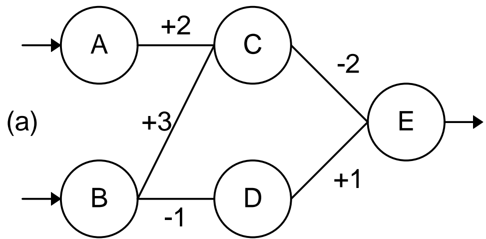
Maniezzo's ANNA ELEONORA

Vittorio Maniezzo has designed ANNA ELEONORA (Artificial Neural Networks Adaptation: Evolutionary LEarning Of Neural Optimal Running Abilities - this one is surely chosen solely for the beautiful acronym) to optimise both the weights and the architecture of ANNs (Maniezzo 1993). The idea behind this system is not only to try to evolve ANNs, but also to try to discover the most optimal representation of the ANNs. The most optimal representation is the shortest chromosome that contains enough detail to generate an ANN with an acceptable error.

To this end, Maniezzo employs a binary coded ANN. For every possible connection, a chromosome contains a 1 or a 0, indicating that the connection is present or not. For every connection which is present, a weight is encoded in a fixed number of bits. This fixed number is also part of the chromosome: it contains a weight length indicator, the granularity, as the first four bits. An example of a coded ANN with this system is presented in figure 34.

Maniezzo employs a fairly standard GA. There are four operators used, reproduction, crossover, mutation and GA-simplex.

Maniezzo ignores the effects of competing conventions. Still, the standard crossover does not work, since the chromosomes can be of different length, because the granularity may differ. A simple implementation trick solves this problem. Chromosomes are actually implemented with maximum length, and with weight values even where no connection is. Since the size of the search space is determined by the interpretation of the strings, it is still constrained by the granularity and the connectivity, and it still isn't as large as it would be when the string were all interpreted with their



(b)

length	AC	AD	BC	BD	CE	DE
0011	1 010	0	1 011	1 111	1 110	1 001

figure 34: The coding system of ANNA ELEONORA. (a) shows the ANN to be coded. It has six possible connections, five of which are present. The one absent is the one between nodes A and D. (b) shows the coding. The first four bits contain the number of bits which are needed to code one weight, in this case 3. Then all possible connections are coded. An absent connection is coded with a 0, a present connection with a 1 followed by the weight, coded in the number of bits indicated by the length indicator. In this case a standard 2-complement system is used to code the weights.

maximum size. The crossover is implemented as a standard crossover on the physical strings. This process is illustrated in figure 35.

The mutation operator is also a bit special. It is split into three parts: a granularity mutation operator, a connectivity mutation operator, and a weight mutation operator. This was done because of the different interpretation of the bits. Not only are the consequences of each of these mutations different, but also, because of the different interpretations, Maniezzo felt that a different mutation probability for each of the types was warranted.

GA-simplex is devised by Maniezzo himself. It is meant for local optimisation, and is based on linear programming techniques. Local optimisation is often done with the backpropagation algorithm, which is very expensive. GA-simplex is cheap, but doesn't guarantee anything, which is not surprising with GAs. It works as follows:

Three parent chromosomes are selected, totally at random. Call the fittest parent x_1,

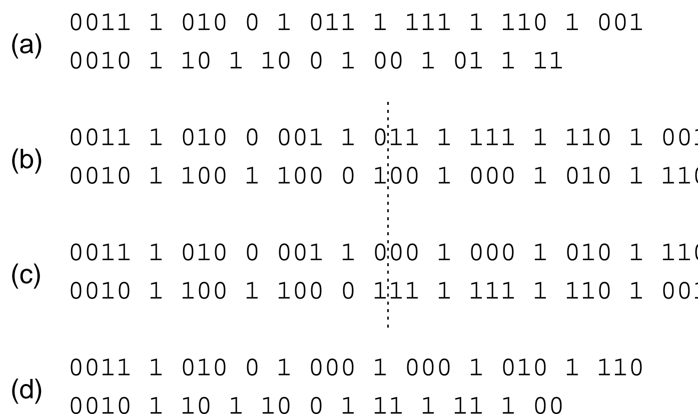


figure 35: Crossover according to ANNA ELEONORA. (a) shows the strings as they are interpreted by the system. (b) shows the physical strings, with values for absent connections and extended weights for the string with the lower granularity. The line shows where the crossover will take place. (c) shows the physical strings after crossover has been performed. (d) shows the strings, after the crossover has been performed, as they are interpreted by the system.

the fittest-but-one x_2, and the least fit parent x_3. Now a child x_4 is generated according to two rules. First, every allele that x_1 and x_2 have in common is copied to x_4. Second, for all the other alleles, the negation of the allele in that position with x_3 is copied to x_4. For instance, suppose x_1 is 1010, x_2 is 1100 and x_3 is 0110. Since x_1 and x_2 have the first and the fourth allele in common, x_4 gets those alleles and is therefore defined by 1**0. For the remaining two alleles, the negation of those alleles is taken from x_3. Since x_3 is defined by *11*, x_4 is defined by *00*, and so, in the end, x_4 is 1000. The idea behind GA-simplex is, that since x_1 and x_2 are the fittest parents, what they have in common is probably quite good. But for those alleles where they disagree, we decide *not* to take what x_3 presents, since x_3 isn't fit and can only serve as a negative guideline.

Maniezzo's tests were fairly limited. He tried a lot of different problems, but all on rather small networks. The largest ANNs he reported about were had two hidden layers, with four nodes in each layer. His populations contained about 100 individuals, and the GA ran for about 3000 generations, although in most cases a few dozen generations were enough to find a good solution. His crossover probability, to be most effective, was rather low, because, as he said, it didn't seem to work all that well. He suspected it was because he ignored competing conventions. Most important, his tests showed that the optimum granularity could indeed be found by his system.

My comment on ANNA ELEONORA is, that the system has some nice features I haven't encountered in other systems (like the granularity mutation and the much criticised use of binary chromosomes). The rather simple extension of doing something about competing conventions, coupled with the more complex extension of allowing networks with different numbers of layers and nodes in one population, should make this system very useful indeed.

The experiments of Montana and Davis

Davis Montana and Lawrence Davis have in the early days of ANNs evolution with GAs experimented with several genetic operators aimed at this field. They have reported on their experiments in an often referred to article (Montana 1989).

In all their experiments they used real-value encoding, and fitness determination based on the MSE on a training set. They only tried to determine the weights of the network, not the architecture. The initial weights were generated randomly with a probability distribution given by $e^{-|x|}$. The operators they tested are the following:

- *Unbiased-Mutate-Weights* (UMW). This operator replaces with a probability p an allele with a random value chosen from the initialisation probability distribution.
- *Biased-Mutate-Weights* (BMW). This operator adds with a probability p to an allele a random value chosen from the initialisation probability distribution.
- *Mutate-Nodes* (MN). This operator randomly selects n nodes from an ANN, and adds to all the incoming weights of these nodes a random value chosen from the initialisation probability distribution.
- *Mutate-Weakest-Nodes* (MWN). The *strength* of a node is defined as the difference between the evaluation of the ANN, and the evaluation of the ANN with all the output links of the node set to zero. Therefore, the strength of a node indicates how much the node affects the ANN performance. MWN selects the m weakest nodes

from an ANN, and performs a mutate-weights operator on all the incoming and outgoing weights of these nodes. If the strength of the node is negative, the mutation is biased, otherwise unbiased.

- *Crossover-Weights* (CW). This operator is equivalent with uniform crossover (UX), which is discussed at the end of paragraph 1.6.
- *Crossover-Nodes* (CN). This operator is a UX which works on nodes. If a node is copied to a child, it means that all the incoming connections to this node are copied to the child.
- *Crossover-Features* (CF). By presenting several inputs to two parent ANNs and comparing the responses of the nodes in these parents, equivalent nodes on the parents are localised. One of the parents gets reorganised so that equivalent nodes are found in equal positions. Then a CN is performed. CF is a crossover operator which can be used to circumvent the competing conventions problem.
- *Hillclimb*. This operator calculates on a parent ANN the gradient for each member of the training set and sums them together. After normalisation the resulting gradient is used to obtain a child from the parent by taking a step in the direction determined by the gradient.

Their experiments tested these operators on the design of an ANN with two layers, the first of seven nodes and the second of ten nodes, for a difficult classification problem. The genetic algorithm was allowed to run for 10,000 iterations. They came to the following conclusions:

- MN works better than BMW. This is no surprise. MN takes into account the structure of the ANN, or, more specific, which weights belong to which nodes, while BMW and UMW work directly on the chromosomes, which don't express that kind of information.
- BMW works better than UMW. Again, this is no surprise. BMW retains more of a fit parent than UMW, so it's obvious that it will be more efficient as a crossover operator.
- There is little difference in the performance of CW, CN and CF. We would have expected CF to perform better than the other two, since it doesn't suffer under the problem of competing conventions. Montana and Davis expressed no comments on this, but it should be noted that their original article, which is relatively old, didn't focus on the problem of competing conventions at all. It is possible they weren't aware of its existence.
- MWN performs very well at the start of a run, but not in the middle or at the end of a run. According to Montana and Davis, this may be a result of the definition of node strength.
- Hillclimb doesn't work too well. It often forces convergence to a local minimum instead of the global minimum. Therefore this operator should only be used on ANNs which are near the global minimum.
- In comparison with backpropagation, a GA with MN and CN as operators was found to perform a lot better. It converged quicker to the global minimum, and it found a better solution than the backpropagation algorithm.

In general

Many experiments have been done with GAs in the design of ANNs. The references in literature I have found deal mainly with small networks. This is a pity, since most ANNs that are of practical use are quite big. I seems researchers shrink away a bit when it comes to larger networks.

The *International Conference of Artificial Neural Networks and Genetic Algorithms* was held for the first time in 1993 in Innsbruck, Austria. Many of the examples in this and previous paragraphs come from the *Proceedings* of that conference and can be found in the bibliography. It was held again in 1995 in Alès, France. I thought that probably some researchers had taken the next step, and used GAs to optimise *large* ANNs. I was astounded to find that in the *Proceedings* of that conference, there were *very* few articles about the design of ANNs with GAs, and none was about the optimisation of large ANNs (Pearson 1995).

It is possible that there have been experiments with large ANNs, but that these experiments failed miserably and the researchers didn't want their failures published (although that doesn't seem to be a very scientific attitude). I personally think it would be worthwhile to do more experiments in this direction, since, if successful, it would elevate this subject from the realm of research to the level of practicality.

3.6 Summary

The search for an artificial neural network (ANN) which implements a certain mapping is a search in a large, noisy, multimodal search space, and may therefore benefit from the use of GAs. The most common areas where GAs can be applied are in the design of the network architecture and in the determination of the connection weights.

An important problem in the use of GAs to optimise ANNs, is the problem of competing conventions. This means that there are many ways to implement a certain mapping in an ANN. The existence of competing conventions dramatically increases the size of the search space, and it reduces the usefulness of the crossover operator.

To solve the problem of competing conventions, the best approach is to reorder the neural nodes of the parent ANNs before the crossover takes place, to insure that functionally equivalent nodes are located in the same position on both the parent ANNs. Other approaches are possible, and a lot of researchers just ignore the problem or refrain from using the crossover operator.

When coding an ANN as a string, we can use direct encoding, meaning we code every relevant detail about the ANN, or indirect encoding, meaning we just code some general pattern from which the ANN can be generated. Direct encoding is used most often.

In coding architectures, direct encoding is normally implemented as a matrix in which every present connection is indicated with a 1, and every absent connection with a zero. In coding weights, the weights can be coded as real values or binary values. If both the weights and the architecture are to be optimised, a combination of these techniques needs to be used.

Standard genetic operators can be used. The crossover operator has to be adjusted

to take care of competing conventions. Mutation rate is often set quite high. Some tailor-made genetic operators exists, like the GA-simplex operator, which uses a technique taken from the field of linear programming to perform local optimisation.

There have been many attempts to use GAs for designing ANNs in practice, but most of them focus on small networks. Experiments on larger ANNs need to be done, to see if this approach can work in practical situations.

4 GAs and Neurocontroller Design

Neurocontrol is about the use of ANNs in controlling plants. A plant is a mechanism which accepts some input, and presents some output as a reaction to that input. The output is not only dependant on the input, but also on the internal state of the plant. This chapter is about the use of GAs to design ANNs for neurocontrol. It begins by telling something about the backgrounds of plant control (4.1), followed by the backgrounds of the use of AI in plant control (4.2). Of the different AI-techniques used in plant control, neurocontrol is explained in more detail (4.3). It is explained how and why GAs may be applied in designing neurocontrol networks (4.4) and an example is given of what has been achieved in this field (4.5).

4.1 Plant control

A *plant* or *process* is a system which has an input and an output. Between the input and output there exists a cause-and-effect relationship. Controlling this system means we can change the inputs, thereby achieving different outputs. The aim of this exercise is to get the system to produce specified, desired outputs. Therefore, *plant control* is about the manipulation of the inputs of a plant, in order to achieve and maintain a desired output of this plant.

A plant may also have some internal states. The output is not only dependant on the inputs, but also on the internal states of the plant. The internal states can change in time, sometimes as a reaction to the input.

A simple example of plant control is the thermostat which regulates the temperature in a house. The plant that the thermostat controls is the heater. The input of the heater can be an electrical signal, which can turn the heater on and off. The thermostat controls this signal. The output of the heater is heat, which is observed by measuring the current temperature in the house (often only in the vicinity of the thermostat itself). This measurement is given to the control element of the thermostat.

We can specify a desired temperature by entering it into the thermostat. The thermostat calculates the difference between the desired and the current temperature,

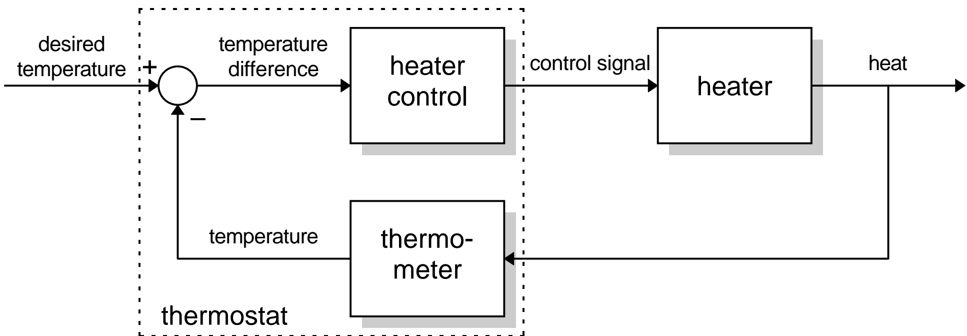


figure 36: An example of plant control: a thermostat that controls a heater. The thermostat elements are found within the dotted rectangle.

and uses that to control the electrical signal. This control won't be continuous. The thermostat will contain some kind of timer, which indicates when it is time for another measurement. This timer can be seen as an internal state.

The logic which the thermostat uses can be as simple as: "If the difference between the desired and the current temperature is positive, send a signal to turn the heater on, otherwise send a signal to turn the heater off. Wait for one minute and repeat." The thermostat example is shown in figure 36.

The thermostat control is an example of a *closed-loop control*. The controller always knows of the current output of the plant, and so it can examine the effect of the control signal on the plant, and it can find out when the desired output is reached. In an *open-loop control*, the controller has no information on the plant output, and in such a case the controller needs to have very detailed knowledge of the plant that is controlled. In practice, open-loop controls are rare.

There are many traditional, non-AI techniques of plant control. Foremost of these is, of course, a human being who observes the plant and makes the necessary adjustments. Simple mechanical controls are also common. A good example is a steam valve, which

```
I := 0;                  { initialize the integral }
E_old := 0;              { initialize last error }
T := step_size;          { step size for approximation of the integral and derivative }
K_p := proportional_parameter;
K_i := integrating_parameter;
K_d := differentiating_parameter;
while not_finished do
begin
    E := desired_plant_output - current plant output;        { initialize error }
    I := I + E·T;                                  { approximation of the integral }
    D := (E - E_old) / T;                          { approximation of the derivative}
    controller_output := K_p·E + K_i·I + K_d·D;    { calculation of controller output }
    E_old := E;
end;
```

figure 37: A simple algorithm for a PID controller.

opens automatically if the pressure gets too high, and closes again when the pressure is safe again.

In recent years, computer controls have been used with varying degrees of success. A common controller for simple plants is the *Proportional Integrating Differentiating* (PID) *controller*. The PID controller mainly uses the error, defined as the difference between the desired output of the plant and the current output of the plant, multiplied by a specific plant-dependant constant, to generate a signal for the plant. The signal is adjusted using the integral and the derivative of the error to compensate for side effects. A simple implementation of the PID controller algorithm is given in figure 37. The PID controller can be made more sophisticated by adding, for instance, integral windup protection or noise reduction techniques.

The PID controller is not suitable for every type of plant. It works best on *linear* systems. For a *non-linear* system, it can maybe be used for part of the control range. Most control theory is about linear systems, for which, after analysis, very good controllers can be built. In practice, however, such controls may work less well, because of disturbances and changes in the plant and noise in the measurements. There is also some theory for the control of non-linear systems, but general solutions do not exist. In practice, most plants are non-linear, at least to a certain degree (Jarmulak 1994a).

4.2 AI in plant control

The more complex the plant, the more difficult it will be to design a good controller. Because of the lack of general conventional methods for controlling non-linear plants, attention has turned to AI methods for plant control. The two main ways in which AI control is applied are *direct intelligent control* and *indirect intelligent control* (Verbruggen 1992).

Direct intelligent control is similar to closed-loop control, where the controller is replaced with an AI controller. Indirect intelligent control is also similar to closed-loop

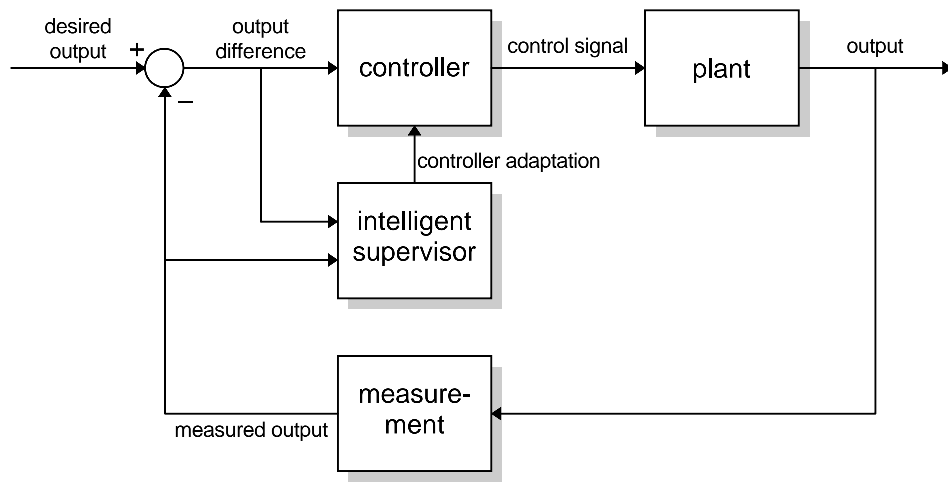


figure 38: Indirect intelligent control.

control, but an extra element is added: an intelligent supervisor directs or adapts the controller, guided by some information from the environment. An example of a common indirect intelligent controller, whereby the supervisor uses the current plant output and the difference with the desired output as input, is shown in figure 38. Other configurations are possible.

There are several AI methods which can be used to implement intelligent controllers. Expert systems are used to replace a human controller, by capturing the human control knowledge in rules. This approach is close to conventional programming. The biggest difference is that the programming language used, for instance LISP or PROLOG, is specifically designed for the purpose of expressing rules. Expert systems are used often for plant control.

ANNs have the ability to learn and are therefore a suitable approach for plant control, because, if we cannot design a system which can control a plant, we might try to get a computer to simply learn how to control the plant. The use of ANNs in plant control is called neurocontrol, and it is the subject of the next paragraph.

The use of GAs in plant control is, as far as I can see after an extensive search through literature, at present limited to the design of other kinds of controllers. GAs can be used to determine a suitable set of parameters for a conventional controller, to design a set of rules for an expert system, or to optimise a neural network for a neurocontrol approach. A GA can be used as the supervisor in an indirect intelligent control system. The little work that has been done in the field of using GAs in plant control has been directed mostly to the design of neurocontrol systems.

4.3 Neurocontrol

Neurocontrol is the name given to the use of neural networks to control a process or a plant. This means that an ANN is trained to control the input of a plant. The ANN is in fact a mapping from some input to an output. The output of the ANN is the input of the plant. A logical input for the ANN is either the difference between the desired output and the current output of the plant, or both the desired and the current output of the plant. A very straightforward way of neurocontrol is therefore the closed-loop control, as shown in figure 39 (the measurement element is left out, to make things clearer; we just assume the plant delivers its output in a form that doesn't need to be changed to another format).

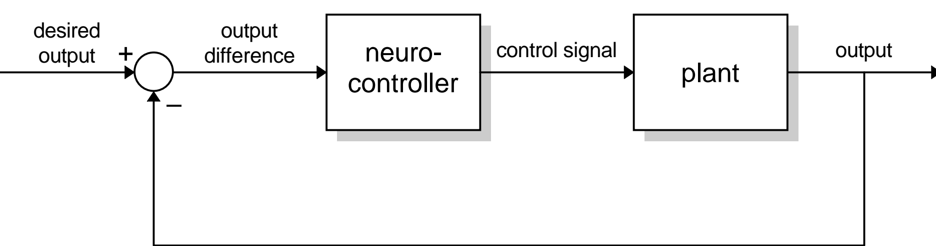


figure 39: A straightforward neurocontrol configuration.

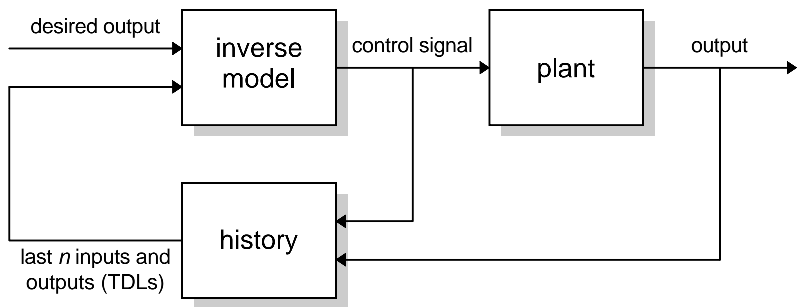


figure 40: The idea of direct-inverse model control.

Modelling

Many implementations of a neurocontrol system use a *model* or an *inverse model* of the plant to be controlled. An inverse model can be presented with the desired output of the plant, and results in the input that needs to be given to the plant in order to get the plant to deliver the desired output. Since the inverse controller needs to know the state of the plant, it may also be presented with, besides the desired output, additional information. This information is often the recent history of the control process, that is, the last *n* plant inputs and outputs. The idea of this so-called *direct-inverse model control* is shown in figure 40. The feeding of recent control history to the plant is done through so-called *tapped-delay lines* (TDLs).

The inverse of a model is often very difficult to create. It may be unattainable, or, if it does exist, may be unstable. An ANN can be used to create such an inverse model. To train an ANN to be an inverse controller solely on the basis of knowledge of plant inputs and outputs is, according to literature, quite difficult (Jarmulak 1994a). It is usually far more difficult than the training of an ANN to be a *forward model*, also called an *identifier*, of the plant, which is a model that reacts in the same way to inputs as the plant itself.

A plant identifier can be used to train an ANN to be an inverse model. The to be trained inverse model is linked to the identifier, and the whole is trained to map an input on itself, whereby the identifier part remains fixed. Even if the inverse model can't be reached, the ANN may converge to a good controller. This is called a *forward-model inverse controller*.

There are controller configurations in which the identifier itself plays a central role. In a *model-based predictive controller* an identifier is used to predict future plant outputs resulting from a sequence of plant inputs. The predicted values are used to find an input sequence which leads to the desired plant output.

These and other neurocontrol configurations have been studied by Jacek Jarmulak with his NeuroControl Workbench (NCWB) (Jarmulak 1994a). All these configurations use either an identifier, or an inverse model of the plant. The inverse model of the plant is best derived by using the plant identifier, and so the plant identifier is used in all cases.

Training an ANN to be a model

To train an ANN to be a plant identifier, we can use on-line training. However, this is not standard practice, since in an on-line run we may not encounter enough different situations fast enough, and so the training may take a very long time and can easily go astray.

For off-line training we need a training set. This training set can be generated by presenting the plant with a set of inputs, and by examining the outputs generated by the plant. However, the plant may not always generate the same output when presented with the same input, since the internal state of the plant may change. Even if we consider the input of the plant to be the input to examine *plus* the current output of the plant, we may get different results at different times, since the current output of the plant need not represent the complete state of the plant.

With off-line training, it may not always be possible to get a good identifier for the plant. If we cannot get a good identifier, the only way to get an ANN to be a plant controller is by training it directly to be a controller. We have to get back to the configuration shown in figure 39, the straightforward neurocontrol configuration, wherein no model is used.

Training a neurocontroller

The controller must guide a plant towards the desired output by presenting the right input. It would be best if the plant immediately reacts to the input by presenting the desired output, which is called *dead-beat control*. In practice dead-beat control is rare. It may not be possible to get the plant to the desired output by presenting a single input. The plant may not be able to react quickly enough to the inputs, there may be limitations to the input, or the desired output may need a sequence of inputs which cannot be reduced a single input.

The obvious way to train an ANN is by using a training set. But, to generate a training set to train an ANN to be a controller, we need to know how we want the plant to react. Unless dead-beat control is possible, there will usually be countless ways of getting to the desired output. Only if we choose between these ways, we can create a training set. However, it is tremendously difficult to make the right choices.

For instance, suppose our plant is a small cart which should be moved to a certain position. The controller may apply force to the cart, and thus can speed it up or slow it down. One way of moving the cart is by applying maximum force in the right direction, and when the cart is at a certain distance from the desired position, applying maximum force in the opposite direction for just enough time to get it to stop in the right spot. Another way of moving the cart is by calculating the distance between the desired and the current position of the cart, and by applying force in proportion to the distance.

Which way is better? The first way will get the cart to the desired position the quickest. The second way seems to me to be more stable; even if the calculation is not very good, the cart will probably end up somewhere near the desired position, since its speed will be reduced when it gets near the target. Also, I think the second way can be implemented in a simpler ANN (less nodes and less connections) than the first way.

In this example, we will often prefer a quick way of getting the cart in the desired spot, but it should also be stable and if possible, the ANN shouldn't be too complex. It

will always important that the ANN results in a working controller (of course) and that it will be able to handle situations for which it not specifically has been trained. Since we can only guess which implementation will have which characteristics and how the different implementations compare to each other, we will have a hard time finding the best implementation, even if we know what our preferences are. And besides that, if it is possible for us to make the right choice between all possible ways to reach a desired output, it won't be that difficult anymore to directly program a controller instead of training or evolving one.

We may conclude that it is usually impossible to create an ideal training set for controller training. We have to accept a suboptimal training set and therefore a suboptimal, or maybe even quite unacceptable controller (still, most existing methods for training a neural controller use such a generated training set), or we have to find another way to train the ANN. One possibility is *reinforcement control*.

Reinforcement control

We talk about reinforcement control when there is no neural model used to train a controller (neither a forward nor an inverse model), but the controller is adapted according to a criterion function which evaluates the behaviour of the controller in relation to the desired behaviour. Since no model is used, the knowledge of the system is eliminated from the controller. The only way to evaluate the behaviour of the controller is to observe its performance in a runtime situation. The evaluation of the behaviour of the controller is used to adapt the controller.

The distinguishing characteristics of reinforcement control are therefore the absence of a model and the usage of an evaluation of the practical performance of the controller to adapt the controller.

It looks like GAs, which are based on adaptation according to fitness (which we find in the evaluation of the behaviour of the controller), and which even without knowledge of a system (which indeed we don't have, for the lack of a model) can lead that system efficiently to a desired state, are a suitable approach for implementing reinforcement control.

Conclusion

Summing up, there are three distinctly different forms of neurocontrol:

- Inverse control, using an inverse model of the plant.
- Model-based control, using an identifier, a forward model of the plant.
- Reinforcement control, using no model of the plant.

The first two forms of neurocontrol are based on the existence of some form of model. If there is no such model available, or if it can't be generated, we are left with reinforcement control. GAs seem to be a suitable choice to design a controller in a reinforcement control environment.

4.4 GAs in neurocontrol

Based on the preceding text, we can see two places where GAs may be used in neurocontrol: to design a (forward or inverse) model of the plant to be controlled to use in a control configuration, or to design a complete controller without a model, which is the case with reinforcement control.

Design of a plant model

If we can generate a training set, which can include TDLs, a GA can be used to design an ANN which has a minimal error on this training set. If the training set represents a plant, the ANN becomes a model of that plant. If the training set is inverted the ANN becomes an inverse model of the plant. This presumes that it is possible to generate a model (forward or inverse) of the plant. Otherwise, either the GA won't succeed in minimising the error, or the model will not function properly.

If it is indeed possible to generate a training set, and we want to design the neural model with a GA, we can use the methods described in the previous chapter. However, the need for a GA is not so great in this case, because there are many other methods known to create a neural model if there is a suitable training set.

It is possible to use a GA in support of these other methods. For instance, we could use a GA to examine the possible architectures for the model, since most other methods, like backpropagation, presume a fixed architecture. We should realise, however, that the evolving of an architecture is a *very* time-consuming process, since every possible architecture in every generation of the population will be trained to see if it works well, so we should be reluctant to attempt such an enterprise.

If it looks like the conventional methods cannot create a good model, this could be a result of the fact that these methods are often limited to feedforward ANNs. We could use a GA to examine more esoteric architectures, to see if it is possible to create a neural model using, for instance, a recurrent network. Since no good conventional methods exist to train every possible network configuration, it seems GAs are a viable approach in this situation. However, we should first try the conventional methods, because if we would immediately start using a GA, we would add just another level of complexity, another level of randomness, and it may be unnecessary to do that.

If there is no training set available, and none can be generated, the training of a model should be done on-line, whereby the model gets adapted according to the differences with the output of the plant. Conventional training algorithms don't work in this case, since they rely on certain aspects of a training set which cannot be found in on-line training, like a great variety in inputs which are presented in a random fashion. Again, a GA might work in this case, but I'm wondering why we should first generate a neural model with a GA, and then use this model in a control configuration, and not immediately generate a complete neurocontroller with a GA. Only if the model would be much simpler than the complete controller, we would benefit from generating a model before generating the complete controller. Since the complexity of the controller is determined mainly by the complexity of the model, this usually won't be the case. It should be noted that, as far as I can see, there is at this time no literature about this subject.

Concluding, it seems to me that in the design of a neural model, if there is a training set available, we should not immediately think of GAs. We should first exhaust the conventional methods, and only if this leads to no success, we could consider trying a GA. If there is no training set available, we should forget about the model completely.

So, in the end we must conclude that GAs do not seem a logical first design choice when confronted with a model-based control configuration.

Reinforcement control

Supervised learning uses the difference between the desired and current output of a plant to train the ANN. Essential in this respect is that the feedback is immediate: the input should generate the correct output at once, and not after a few cycles of the same input, or maybe after a sequence of other inputs. In plant control this is not desirable. It would mean that a dead-beat control for the plant should be possible, and this is usually only possible for really simple, theoretical plants.

Reinforcement learning, also known as *graded learning*, is similar to supervised learning, except for the fact that the ANN only receives a score that tells it how well it has done over a sequence of multiple training trials. We can see this as a fitness value that the plant gets awarded. Reinforcement learning is often used in process control, for which it particularly applicable. We name this type of application *reinforcement control*. There is not much literature about reinforcement control, because currently the only useful learning laws are commercially proprietary, and therefore a well-guarded secret (Hecht-Nielsen 1991).

Reinforcement control is naturally close to GAs. It works with a fitness function. It only observes the performance of the ANN under consideration, without using additional knowledge. The ANN should be adapted according to the fitness value it receives. GAs are particularly suited for effective and efficient (at least in the long run) adaptation. So, reinforcement control provides what a GA needs, and a GA can provide what is needed for reinforcement control. If we use a GA with reinforcement control, we call this *genetic reinforcement control* (GRC).

In conventional reinforcement learning just one ANN gets trained. After a number of cycles, a training phase is entered whereby the ANN gets adapted, followed again by running the ANN for a number of cycles, followed by a training phase, etcetera. The fitness value of the ANN is determined just before each training phase.

With GRC, we have a population of individuals which are examined. We need to determine the fitness of each individual only once, since the individuals don't get adapted themselves. Adaptation comes from the children that are produced, and that replace the inferior individuals in the population.

With GRC, there are many decisions we have to make. For instance, how long should a test run be? If it is too short, the fitness value may be unreliable, and the training will fail. If it is too long, the training may take far more time than necessary. Maybe the test run should be short in the first generations, and get longer when convergence takes place. If there is more than one possible target situation for a plant, how many target situations should the plant reach in the test run? Should these situations be the same for every test run? If not, shouldn't the fitness of every individual be redetermined every generation, even if the individual is copied from the previous

generation?

The intricacies of GRC will be discussed later in this text. For now, it would seem that GRC is the most viable application for GAs in neurocontrol. On a side note, I would like to point out that there is an additional benefit of the use of GAs in reinforcement control over conventional methods in reinforcement control: GAs can be used to generate a wide range of different types of networks, even those that cannot be trained with conventional methods. If the best solutions are those that cannot be gained with conventional methods, GAs may lead us to them.

4.5 An experiment with GAs in neurocontrol

Currently only a handful of researchers has given any attention to GAs in neurocontrol. Foremost is professor Darrell Whitley, who has done some work in this field with various associates. Besides some papers by Whitley, very few publications specifically about GAs in neurocontrol can be found.

Whitley is one of the best known researchers in the field of evolutionary algorithms for the optimisation of ANNs. His has published some papers on this subject, and has recently written an overview of the field (Whitley 1995). One of his research projects is in the optimisation of ANNs in neurocontrol problems. He and his fellow researchers used the GENITOR algorithm to deal with this problem area.

The GENITOR algorithm is a general GA, which Whitley uses so often that he normally doesn't explain it anymore. It uses the ranking mechanism to order the individuals in the population. The genetic operators select two parents from the population at random, according to rank, and produce two children. One child is selected randomly and discarded. The other child replaces the lowest ranking individual in the population, and the child is ranked according to its fitness.

Whitley found that genetic diversity helps GENITOR, so there is always some mechanism to introduce genetic diversity into the population. He mostly uses adaptive mutation, which means that the rate of mutation gets higher if the population starts to converge. A spin-off of GENITOR, named GENITOR II, uses a parallel approach with island populations, which also helps to introduce genetic diversity and which can be used to battle genetic drift (Whitley 1990).

Whitley has used the GENITOR algorithm to optimise the connection weights of an ANN to be used for neurocontrol (Whitley 1993). The chromosomes contain the weights of all the connections, plus a special allele which holds the crossover chance. The algorithm starts with a randomly generated population, wherein every weights gets initialised to a random value between -2.5 and +2.5, and the crossover chance to a random value between 0.0 and 1.0. The fitness is a function of the error, and as is usual with GENITOR, is used to rank the individuals in the population.

In each generation cycle, two individuals are selected using a linear bias according to the ranks. The crossover allele of the first individual determines the chance that crossover will be used. If chance rules crossover out, the second individual gets rejected, and the first individual produces a child by mutation: every allele of the first individual gets changed by adding a random value between -10.0 and +10.0. The generated individual is inserted into the population according to its rank, removing the

lowest ranking individual.

If crossover is chosen, the parents are checked. If they differ by less than three alleles, no new individual is generated ("incest prevention"). Otherwise, a simple one-point crossover is performed. The two children get the crossover probability from the first parent. If this first parent is fitter than a child, the child's crossover probability gets increased by 0.10, to a maximum of 0.95. Otherwise, its crossover probability gets decreased by 0.10, to a minimum of 0.05. Again, the children replace the lowest ranking individuals and get inserted in their rightful position in the population.

The iteration phase ends when an individual is generated which confirms to the goal set for the run, or when a maximum number of iterations is performed.

So far, there's not much that makes Whitley's GENITOR especially suited for neurocontrol. The crux is in the fitness determination. Whitley experimented with a *pole-balancing system* (also called an *inverted pendulum* or *cart-centering system*). In such a system, the controller can apply force to a small cart on which a pole rests on one end. The cart should be moved by the controller so that the pole remains balanced. The controller fails when either the pole falls, or when the cart runs off the track it's on.

The fitness is determined by the period the controller succeeds in keeping the pole-balancing system from failing. The controller is considered to be perfect if it can keep the pole balanced for 40 minutes of simulated time (120,000 timesteps). The emergence of such a controller is also the signal for GENITOR to stop the evolution process.

Whitley reports encouraging results from the use of GENITOR in his (admittedly quite simple) experiments. The GENITOR algorithm was used earlier by other researchers. However, Whitley made some basic changes to the algorithm, which gave it, according to him, the success it achieved. The three characteristics which were essential to its success were the use of a *small population* (of about 50 individuals), the use of *real value-encoded weights*, and the use of an *extremely high mutation rate*. Whitley's version of GENITOR is essentially mutation-driven. The competing conventions problem is circumnavigated by Whitley, by using a small population and giving a very minor role to the crossover operator.

I would like to comment on Whitley's use of GENITOR. Whitley largely skips one of the most important aspects of GAs, namely the crossover operator to effectuate changes. As soon as there are some quite fit parents in the population, mutation will get more and more influence, until nothing else is used. I think Whitley's mutation rate is set *extremely* high. *Every* allele is mutated, within a *gigantic* range. This range is so large, that two individuals which are mutations from one and the same parent are likely to differ more from each other than any two individuals which were generated in the starting population (Whitley doesn't neglect to point this out in his paper, and in personal correspondence has admitted that the success also surprised him).

It looks like Whitley is randomly generating new individuals, until a fit one is found purely by chance. I can't believe this would lead to good results on large ANNs. Unfortunately, Whitley only tested it on rather small networks, and it is possible that the success would be less on larger configurations.

4.6 Summary

Plant control is about the controlling of a process to get it to deliver a certain output by applying the right input. There are many different ways of plant control. Conventional ways include human control, mechanical control and computer control by conventional means, like PID control.

AI is brought into the field of process control where it is difficult to get the computer to be a good controller by conventional means. Examples are the use of an expert (rule-based) system and neurocontrol. In neurocontrol a neural network is trained to be a controller.

Two distinctly different ways of implementing neurocontrol are model-based control, using either an inverse or a forward model of the plant, and reinforcement control. Reinforcement control gets its feedback solely from the performance of a neural controller over a period of time.

If we want to use GAs in neurocontrol, two obvious possibilities are to use a GA to generate a model for model-based neurocontrol, or to use a GA to generate a controller by some means of reinforcement control. If we want to generate a model, the only viable situation to apply a GA is when conventional methods don't seem to work, and we use the GA to generate a complete model (not only the architecture).

For reinforcement control GAs are a natural approach, since the reinforcement evaluation can be used as a fitness measure, since reinforcement control is used in those situations where no information is known except for the behaviour of the plant, and since GAs provide a method of adaptation which is needed in reinforcement control configurations. If we use GAs in reinforcement control, we call this *genetic reinforcement control* (GRC).

We may conclude that the use of GAs in process control is, at the moment, best applied to reinforcement control situations. This is a subject in which still a lot of work needs to be done, judging by the fact that there is currently only a handful of papers about it available.

Conclusion of Part II

Overview

The second part of this thesis presents an overview of the application of GAs in the design of neural network, and specifically in the design of neurocontrollers. It is based on a large number of papers, which can be found in the bibliography.

A lot of research has been done in the design and training of neural networks with GAs. Important questions of the GAs which are used for this purpose are:

- How is the network coded? Should the network be encoded with all its details (direct encoding) or should just some aspects of the network be encoded (indirect encoding)? Should real or binary values be used for the weights? Should only weights be encoded, or the architecture, or a combination of them? In what order are the connections and the weights placed on the chromosome? Can the chromosome be of a variable length?
- What genetic operators should be used? Are standard genetic operators enough? How high should the mutation rate be?
- How is the fitness measured? Should complexity of the network be taken into account?
- Should competing conventions be considered? And if so, how should they be treated?

Although there is research about this subject matter published in abundance, only very few of the questions raised above have a universally accepted answer.

A neurocontroller is a neural network which is used to control a plant. A plant is a process which converts input into output, and which has internal states which direct the input-output conversion.

Training of neural networks by conventional methods mostly requires a training set. It is very difficult, if not impossible, to create a good training set for neurocontrol training. A second way to train a neurocontroller is by using a plant model. However, it is not always possible to create a plant model. A third way is to use reinforcement

control. Reinforcement control adapts a neurocontroller according to the success it has during the actual application of the neurocontroller.

GAs are especially suited to design neurocontrollers which need reinforcement learning to be trained. This is because a GA can use the performance rating of a neurocontroller as a fitness measure, and a GA doesn't need more information than that. Besides that, GAs can be used to design esoteric network configurations. This application of GAs is called genetic reinforcement control (GRC).

Very little GRC research has been done, but the approach seems promising.

Personal views

I have already concluded that the application of GAs on neurocontrol design is most suitable for reinforcement control problems, since in other situations there are conventional methods to be tried first, and GAs just seem to make the problems more complex.

This conclusion doesn't come from literature. In literature, sometimes control problems are used as test problems when GAs are applied to the design of neural networks, but the reinforcement aspects of control problems are seldom mentioned. The only researcher I've encountered who directs his research specifically to reinforcement control problems is Darrell Whitley (Whitley 1993).

I've already commented on Whitley's GENITOR in paragraph 4.5. I will repeat these comments here: Whitley's GENITOR approach to GRC is essentially mutation-driven, thereby neglecting the arguably most important operator of GAs, the crossover operator. Maybe Whitley is right, maybe his GENITOR is the best way to approach GRC. However, I see no reason why the approaches other researchers use to standard neural network design problems wouldn't work just as well for neurocontrol problems, except that they need to be extended with some reinforcement parameters.

If Whitley says real values work better than binary values, this may be true for his configuration, but it might not be true in other situations. Whitley has not attempted to solve the competing conventions problem, and just uses a small population so that the problem doesn't get too big. These are not complaints about Whitley's research. It's just an indication that a lot more research needs to be done on the many different aspects of GRC.

What we can say, is that if GRC is found to be useful, this can be very worthwhile, since reinforcement learning is a technique which is often required in industry, but for which very few publications exist.

Doing the required research in GRC means that lots of experiments need to be performed. For that, a flexible software environment is needed. I've decided to finish my graduation project with the design and development of such an environment. This environment is the subject of the final part of this text.

Part III

Elegance

"The road to wisdom?
Well, it's plain and simple to express:
Err and err and err again
But less and less and less"
--- Piet Hein (quoted by Dennett in *Darwin's Dangerous Idea*)

"The art of progress is to preserve order amid change
and to preserve change amid order"
--- Alfred North Whitehead

5 The Design of Elegance

This chapter describes the software environment Elegance*, which has been implemented to experiment with the use of GAs in the design of controllers, especially neurocontrollers, in a reinforcement situation. It opens with an explanation of why a software environment is needed, and what the requirements for this software environment are (5.1). Then a functional design (5.2) and a technical design (5.3) are given. The complexity of the GA warrants a separate paragraph (5.4). This is followed by an overview of some implementation aspects (5.5), an idea of how the program is used in practice (5.6), and an evaluation of the performance and usability of the program (5.7). The chapter concludes with an indication of how the program can be improved and extended in the future (5.8).*

5.1 Purpose

In the previous chapter it was explained that the subject of the application of GAs in reinforcement control is, although promising, still very much in the initial research stages. Very few scientists have aimed their research at this particular subject, and the results of those that have are at least candidates for improvement (as far as I can see).

There are many different aspects of GAs that need to be tuned to reinforcement control problems, like the coding of the chromosome, the fitness function, the genetic operators and the general parameters. What combination of which parameters works well in what particular situation is something that can only be answered when it is possible to run a large number of different experiments. To be able to run those experiments, a flexible software environment is needed.

Elegance, which stands for “Engineering Laboratory for Experiments with Genetic Algorithms for Neural Controller Evolution”, is the name of the software environment I have created to run a large selection of those experiments. The main purpose of Elegance is to be able to test many different GA configurations on the same problem area to see which configuration runs best. Since this may depend on the problem type, Elegance offers not only flexibility in GA design, but also in plant design. I have found

in literature no reference to the existence of such a general tool in this particular research area.

The main requirements of Elegance are formulated as follows:

- Elegance should be flexible in GA configuration design.
- It should be easy to add new genetic operators.
- Elegance should support a number of different plants.
- It should be easy to add new plants.
- The fitness function of a GA should be based on an evaluation of the performance of a controller in an actual run (this is the reinforcement requirement).

Following these main requirements, there are a number of minor requirements which were kept in mind while implementing Elegance:

- Whitley's GENITOR (paragraph 4.5) principles should be supported, because it seems to me that they can be improved upon.
- Maniezzo's ANNA ELEONORA (paragraph 3.5) principles should largely be supported, because I think they sound promising and they are radically opposed to Whitley's GENITOR.
- Both weight and architecture design should be supported.
- Several types of neurocontrollers should be supported.
- There should be a way to deal with competing conventions.
- There should be support for tapped-delay lines (TDLs).
- Elegance should be user-friendly, self-explanatory and easy-to-use.

Flexibility is a beautiful thing, but it's also wise to place some limits on it in order not to overextend oneself. The following limits were placed on Elegance:

- Speed is deemed to be of less importance than user-friendliness and maintainability of code.
- Elegance aims itself specifically at neurocontroller design (although there are different types of neurocontrollers supported). PID controllers are added for comparison and test purposes.
- Of course, it would be nice if other controller types could be added easily, but since the controller type has a lot of influence on the GA implementation, only controller types which are a lot like the controller types already implemented can be added with only minor modifications.
- The size of the neurocontroller is limited to some maximum number of nodes, which makes the neurocontroller implementation a bit easier (this limitation will very likely be removed in a future version of Elegance, and the current limit is more than enough for the current plants).
- The chromosome structure design is fixed.
- Only direct encoding is supported.
- Either the weights of an ANN can be determined, or the weights in combination with the architecture, but not the architecture by itself.
- There is no combination with a conventional local optimisation technique supported.

- Newly generated individuals are always inserted into the original population, replacing other individuals.
- To extend the program with new plants, new controllers or new genetic operators, one needs to have access to the source code and the compiler.

As basis for both the functional and the technical design of Elegance, I've used the program NCWB by Jacek Jarmulak (Jarmulak 1994a). Jarmulak's program is also used to experiment with the design of neurocontrollers, but his focus is on model-based control. His design could easily be adapted for Elegance. Besides the necessary changes to his design, I have also tried to improve upon certain aspects, especially on the user interface and the usability of the program.

5.2 Functional design

The control environment which is the basis of Elegance is shown in figure 41. This control environment is called the *control loop*. The control loop consists of:

- A *plant*, which is the process to be controlled.
- A *controller*, which directs the input to the plant.
- A *setpoint generator*, which produces the desired plant output.
- A *history* entity, which remembers some recent outputs of the plant and optionally sends them to the controller.

This control loop is a simulation of a practical situation. It is meant to show how the controller directs the plant, and also to test the performance of a controller during an evolution run. So, the evolution run uses the control loop to determine the fitness of a controller. This is schematically shown in figure 42.

The GA contains a population of controllers encoded as chromosomes. Using genetic operators, the GA generates new chromosomes. A new chromosome is decoded and installed as the controller in the control loop. The GA then runs the control loop for some time. During the run, the GA collects the results of the run which are used to rate the controller performance, and so the fitness of the chromosome. After the fitness has been determined, the new chromosome is added to the population, replacing an existing chromosome.

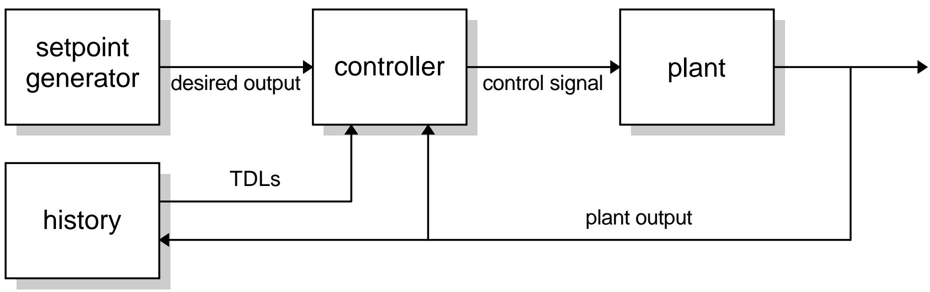


figure 41: The basic Elegance control loop.

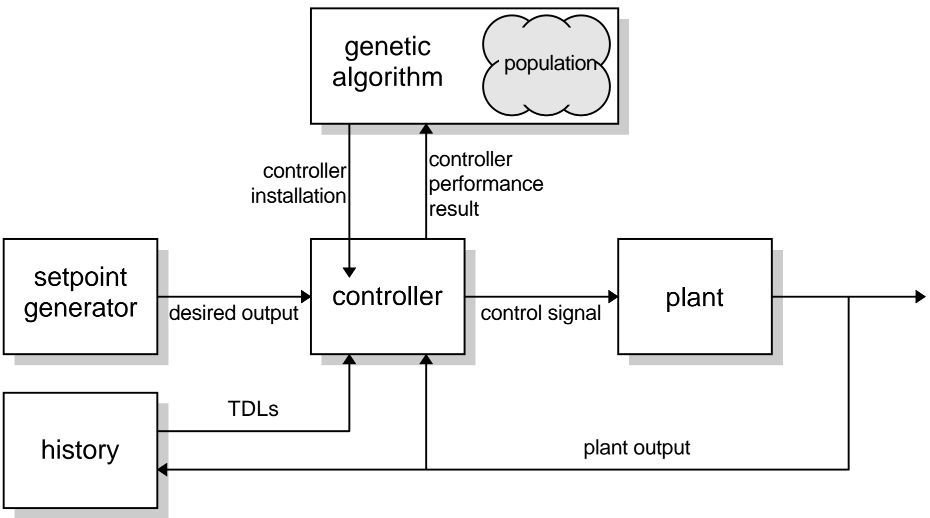


figure 42: The GA and the control loop.

Elegance has an object-oriented design, and distinguishes several basic objects, which are shown, with their relationships, in figure 43. In this figure, if there is an arrow from one object to another, it doesn't only mean that the first object can access the second object, but also that the first object can access all those objects that the second object can access. The objects are:

- *Project*: The project is a container for all the objects needed for an experiment.
- *Plant*: The plant is the process that needs to be controlled by the controller. The plant also remembers the values for the TDLs.

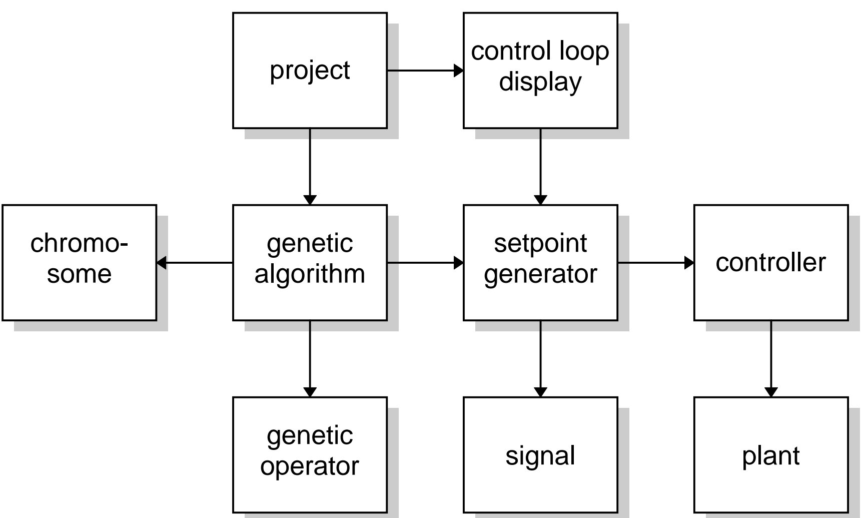


figure 43: The basic objects of Elegance.

- *Controller*: The controller is the process that generates the input for the plant. If it's an ANN, the controller contains a complete description of that ANN, including the architecture, the weights and the activation function.
- *Setpoint generator*: The setpoint generator generates desired outputs, or *setpoints*, which the controller should use as a target for the plant output.
- *Signal*: For every possible plant output, the setpoint generator has a signal which directs the setpoint for that plant output.
- *Control loop display*: The control loop display, or *display* for short, contains a definition of the charts the user wants to see during a control loop run.
- *Genetic algorithm*: The GA contains all parameters which define how the evolution is performed, like the encoding parameters, the fitness definition, the handling of genetic operators, the population size and the reinforcement parameters.
- *Genetic operator*: The GA has access to a collection of selected genetic operators, which it can use to generate new chromosomes.
- *Chromosome*: The GA has access to a population of chromosomes, each of which encodes a possible controller configuration.

The main functions implemented in Elegance are:

- Project maintenance.
- Plant maintenance.
- Controller maintenance.
- Setpoint generator maintenance.
- Control loop display maintenance.
- GA maintenance.
- Running an evolution.
- Running a control loop.

5.3 Technical design

In this paragraph, I will focus upon the individual basic objects of Elegance, especially on their attributes, and, if applicable, on the different implementations that the objects can have.

Project

The project is meant as a container for all the objects an Elegance experiment needs. Because of that, Elegance saves all the objects of a project when the project is saved. The main attributes of a project are:

- The *project name*, which is simply a line of text which can be used to describe the experiment.
- The *filenames* of all the files that are used by the project to store the project objects.
- The *timestep* size, which is the granularity of the clock used by the control loop. The

control loop is, for implementation reasons, a discrete process, and that's why a timestep size is needed.

Plant

The plant is the process that is controlled in the control loop. The plant in itself has no attributes of its own, but depending on the plant implementation there may be internal parameters and an initial plant state to define. The initial plant state is equal to the initial state of the plant outputs. The plant also stores the values for the TDLs.

Elegance contains the following plant groups:

- *Test plants*. Elegance contains three plants which are mainly used for testing purposes. These are the *SISO plant*, which has just one input and one output, the *DIDO plant*, which has two inputs and two outputs, and the *titration*, which is a highly chaotic, discretised version of a titration process. The SISO and titration plants are copied from NCWB (Jarmulak 1994a) and the DIDO plant is designed specifically for Elegance.
- *Trolley*: A trolley is a small cart which is driven by an electrical signal. The purpose of the controller is to use the signal to direct the trolley to specified positions. There are two versions of the trolley found in Elegance. The basic *trolley*, which is copied from NCWB, is a one-dimensional version of the trolley, in which the cart is placed on a rail of half a meter in length. The *two-dimensional trolley* places the cart in a small room, and the trolley gets directed by two input signals, the electrical signal and the direction in which the control force is applied, to move the trolley to specified spots in the room. The two-dimensional trolley is designed specifically for Elegance.
- *Pole balancing system*: The *pole balancing system*, also called the *inverted pendulum* or *cart centering system*, consists of a cart on a piece of rail, which can be moved by a control force. On the cart a pole is placed on one end, with the bottom end of the pole fixed to the cart. The pole can fall to the left and to the right. It's the purpose of the controller to move the cart in such a way that the pole will stay balanced. There are two versions of the pole balancing system found in Elegance. The basic *pole balancing system*, copied from NCWB, is as described. The *bang-bang pole balancing system* has two differences with the basic pole balancing system. First, the control force can only be +10N or -10N. Second, the controller output is converted by the plant to the range [0,1] by using a sigmoid with learning rate 1 (equation 12). This value is considered to be the chance that the control force will be +10N. If it is not +10N, it's -10N. So, there is an element of randomness in the way the plant works. This implementation of the bang-bang pole balancing system is copied from Whitley's GENITOR experiments (Whitley 1993). The main difference between the pole balancing system and the other plants in Elegance is the fact that with the pole balancing system the controller's aim is to keep the plant from failing, while with the other plants the controller should direct the plant output to a certain value.
- *Bioreactor*: The *bioreactor* is a constant volume, isothermic, continuous flow, stirred, tank reactor. It contains water, biological cells and nutrient. Water

containing nutrient is continuously added to the tank. The volume is kept constant by draining contents of the tank equal to the incoming rate. The bioreactor is controlled by means of the flow rate. The state of the tank is described by the cell mass and the amount of nutrient in the tank. It's the objective of the controller to get the bioreactor to stabilise the cell mass in the tank at a certain level. The bioreactor is copied from Jarmulak's NCWB, and it's the most complex plant in Elegance.

The various plants are described in more detail in appendix B. To add a plant to Elegance, the source code the programmer needs to provide is only a list of the input, internal and output parameters needed by the plant, and a function which performs a plant cycle (see appendix C.1).

Controller

The controller is the object that needs to be designed by the genetic algorithm. The controller's job is twofold: to direct the plant to a certain desired output and to keep the plant from failing. If the first goal is most important, which it is for most plants, than the success of the controller is mainly defined by the mean difference between the desired and the current plant output. If the second goal is most important, as it is, for instance, with the pole balancing system, the success of the controller is mainly defined by the length of the period it can keep the plant from failing.

Elegance supports two kinds of controllers: PID controllers and neural controllers. For both types of controllers, we need to specify which of the plant outputs will be used as *setpoint outputs*. Setpoint outputs are those outputs for which the setpoint generator tells the controller the target value, which the controller then tries to get the plant to produce.

For each setpoint output, a factor needs to be specified. This factor indicates the weight of the error (the difference between the setpoint and the actual plant output) on this particular setpoint in the calculation of the global controller error. This means, that the error on a setpoint with a factor of 2 is counted twice as much as the error on a setpoint with a factor of 1. For each plant, at least one setpoint output should be specified. For PID controllers, one setpoint output is also the maximum.

There are two types of PID controllers in Elegance:

- The *basic PID controller*: This PID controller is implemented conforming to the algorithm specified in figure 37.
- The *PID controller with integral windup protection*: This PID controller is equal to the basic PID controller, with an extension which protects the integral in the algorithm from becoming too big when the controller output approaches the limits of the plant input. This version of the PID controller is copied from NCWB (Jarmulak 1994a).

For PID controllers, we need to specify the proportional, integrating and differentiating parameter. These are also the parameters which the GA can optimise when a PID controller is chosen as the controller element of the project. PID controllers are explained in detail in paragraph 4.1.

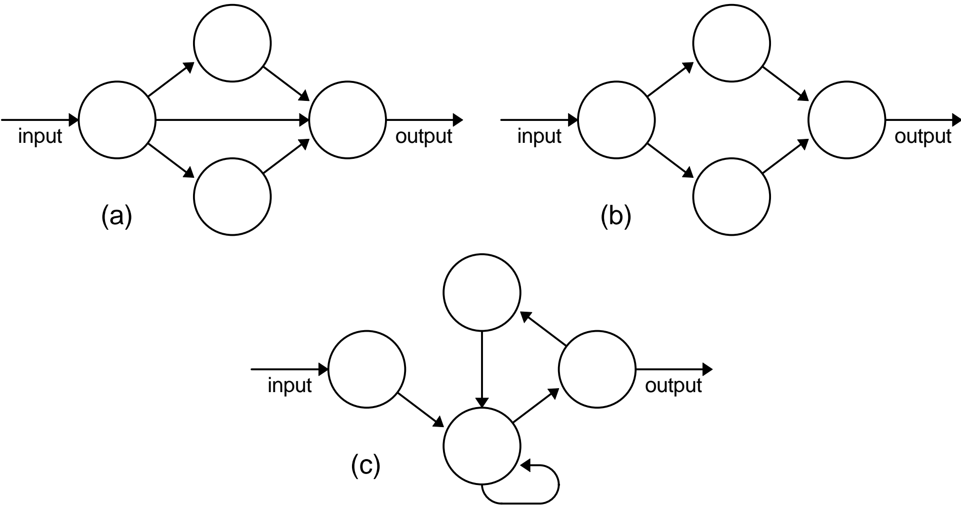


figure 44: Examples of the three types of ANNs supported by Elegance as neurocontrollers: (a) the feedforward neurocontroller; (b) the layered feedforward neurocontroller; (c) the recurrent neurocontroller.

There are three types of neurocontrollers in Elegance (examples of which are shown in figure 44):

- The *feedforward neurocontroller*. This type of ANN is described in paragraph 3.1. It is an ANN in which the connections are always from nodes in one layer to nodes in successive layers, although it's not compulsory that the links are always connected to *the* next layer.
- The *layered feedforward neurocontroller*. This is a feedforward neurocontroller in which the links are always connected to nodes in *the* next layer, and not to nodes in *any* of the successive layers.
- The *recurrent neurocontroller*. This type of ANN is rather different from the feedforward controllers. Connections between any of the nodes are allowed. It's not useful to have connections to the input nodes, however, since they are *clamped*, which means that they have a fixed value that cannot be changed from inside the neural network. All other connections are possible, however. Since this means that it is possible to have loops in the ANN, a different approach to the flow in the network is needed. The implementation is thus: nodes retain the value they got in the last cycle (during the last timestep). The next cycle, all nodes synchronously send the value they have over the outgoing connections to the nodes they are connected to. Then they synchronously receive the values from other nodes through their incoming connections, and retain the new value they now calculate.

For neurocontrollers, several parameters are needed:

- The *weight minimum and maximum*, which determine the limits of the weights. If the GA changes a weight so that it gets larger than the maximum, the weight takes on the maximum. If the GA changes a weight so that it gets smaller than the minimum, the weight takes on the minimum.

- The *maximum number of hidden nodes*, which determines how many hidden nodes the neural network can maximally have. The sum of the input nodes, hidden nodes and output nodes cannot exceed a predefined value (in the first version of the software this value is 75, which is more than enough for the plants in Elegance). If the ANN is a layered feedforward network, the maximum number of nodes needs to be specified per layer, with a maximum of five layers.
- The *activation function*, which is a transfer function used by the hidden nodes. If present, this function can be linear, an arctangent, or a sigmoid. The linear function is simply a limitation of the hidden nodes output. The arctangent is also used in NCWB (Jarmulak 1994a). The sigmoid is the most common activation function. For an activation function, the user may specify the minimum value, maximum value and learning rate (although the learning rate will be overwritten by the genetic algorithm). The fact that the activation function is used only on hidden nodes is also copied from NCWB.
- A *network output limitation*, which specifies whether or not the controller should make sure that the controller output falls within the limits of the plant input. Normally this should be the case.

Besides these parameters, each neurocontroller has an architecture and weights. The architecture specifies which connections are present in the neurocontroller. While Elegance itself limits the connections in accordance with the network type chosen, the user may indicate which of the remaining connections should always be absent, which should always be present and which are optional. Only optional connections will be subject to change by the GA. The weights of the connections in the neurocontroller can be specified by the user, but all weights are subject to change by the GA.

Setpoint generator and signals

The setpoint generator generates the desired plant output values according to the current control loop cycle. The user may specify for each plant output which is indicated to be a setpoint output with the controller, what kind of signal defines the desired plant output. For each of the signals some signal-dependant parameters need to be specified. The possible signals are:

- A *constant wave*. This is simply a constant value.
- A *square wave*. This is a signal that oscillates between two fixed values.
- A *sinusoid wave*: This is a signal that implements a sinusoid.
- A *random wave*: This is a signal that each timestep has a chance of changing to another random value.
- A *random wave with a fixed interval*: This is a signal that changes to another random value after a fixed number of timesteps has passed.

For each signal, it can be specified that the signal should be error proportional. This means that the controller does not get the actual signal value, but the difference between the signal value and the actual plant output. This can only be indicated with neurocontrollers, since PID controllers already calculate this difference internally.

Control loop display

The control loop display specifies the charts that are shown on the screen during a control loop run. Since for each configuration the user might wish to see different things on the display, this display is made user-definable.

A display consists of *charts*, and each chart consists of a number of parameters and a collection of *lines*. The chart parameters are:

- The *chart name*, which appears at the top of the chart.
- The *chart height*, which is a percentage of the screen height.
- The *colour of the background and the axes*.
- The *chart width*, which indicates how many simulated seconds are displayed on the chart.
- The *minimum and maximum Y-value*, which determines what line values can be displayed.
- The *X- and Y-marks*, which indicate where marks are placed on the axes.

A chart contains lines. Each line is defined by the following parameters:

- The *line type*, which can be a *plant input line*, a *plant output line*, a *setpoint line*, or an *error line*. The user actually selects *which* input, output, setpoint or error he wants to see displayed on the chart.
- The *line name*, which is displayed in the chart *legend* (a little box with a specification of the lines on the chart).
- The *line colour*.

Chromosome

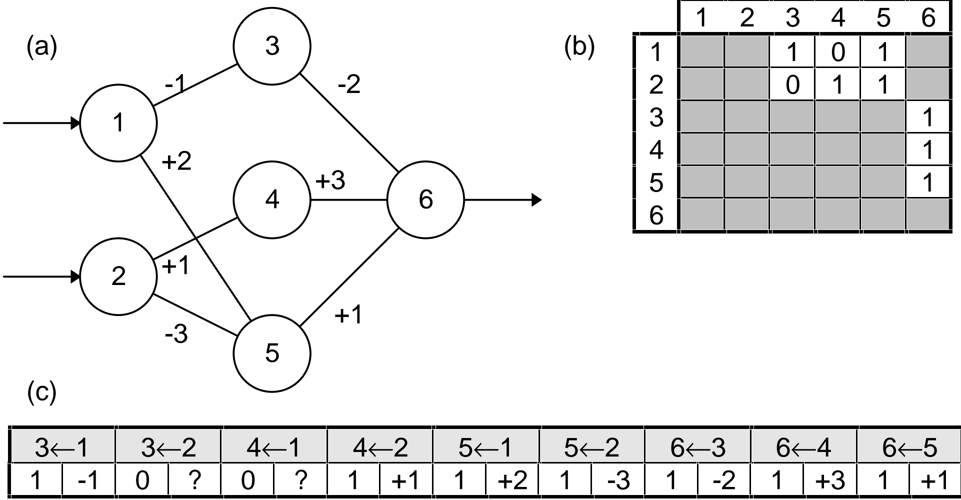
The chromosome is an encoded representation of a controller. The chromosome structure is designed specifically to hold an ANN, but if the controller is a PID controller, the same chromosome structure is used. The coding of the chromosome is based on the coding used by Maniezzo, with his ANNA ELEONORA (Maniezzo 1993), which is discussed in paragraph 3.5.

The chromosome structure is consists of groups of data. Each group of data codes one of the possible connections in the network, including the weight belonging to this connection. So, each group of data contains a bit, which indicates if the connection is present (if the bit is 1, the connection is present, otherwise it's absent), followed by the weight, which can be either a real value or a number of bits, depending on the encoding specification of the GA.

The collection of bits which indicate the status of the connections are called the

connection	weight	connection	weight	connection	weight	

figure 45: The Elegance chromosome structure.



	1	2	3	4	5	6
1			1	0	1	
2			0	1	1	
3						1
4						1
5						1
6						

3←1		3←2		4←1		4←2		5←1		5←2		6←3		6←4		6←5	
1	-1	0	?	0	?	1	+1	1	+2	1	-3	1	-2	1	+3	1	+1

figure 46: An example of the encoding of an ANN as a chromosome. (a) shows the ANN in question. Since I have decided that this is a layered feedforward neural network, there are nine different connections possible. (b) shows the nine possible connections in a matrix. On the vertical axis the nodes from which the connections come are indicated, and on the horizontal axis the nodes to which the connections lead. The shaded cells indicate connections which are not possible. A zero in a cell indicates that that connection is absent, and a 1 that the connection is present. (c) shows the encoded chromosome. The top row shows which connection is coded, and the bottom row shows the groups of data. The weight for two of the possible connections cannot be determined from the network, since the connections are absent, but still something is coded for these weights in the chromosome. This is indicated with a question mark.

connectivity bits. If a connectivity bit indicates that a connection is absent, there is still a weight coded for this connection. This eases implementation and works as a kind of memory for weights which are temporarily shut off.

The groups of data are placed together in one long string. This is presented graphically in figure 45.

The nodes in the ANN are numbered. The order of the string is as follows: first the incoming weights of the first node are coded, then the incoming weights of the second node, etcetera. The incoming weights are ordered according to the node numbers of the nodes from which the connections come. An example of the coding of an ANN is shown in figure 46.

If the weights are encoded binary, a weight length is needed to determine the number of bits available for each weight. The weight length is fixed for a chromosome, but it may vary in the population. The weight length is therefore an attribute of a chromosome, but this attribute is not coded into the chromosome.

A second attribute of a chromosome can be a crossover probability, which is used when adaptive mutation is applied, like Whitley does in GENITOR (see paragraph 4.5). This attribute is also not coded into the chromosome.

A third attribute can be the learning rate of the activation function of a neurocontroller, again not coded into the chromosome.

For a PID controller, the chromosome holds three groups of data. The three connectivity bits are all 1. The first weight is the proportional parameter, the second weight is the integrating parameter, and the third weight is the differentiating parameter.

Genetic algorithm

The genetic algorithm defines how the evolution of new controllers is executed. There are many parameters, which basically fall in these group:

- *Encoding parameters*, which are used to define the details of the chromosome encoding.
- *Fitness parameters*, which are used to define how the fitness calculation takes place.
- *Genetic operator parameters*, which contain a list of the genetic operators used by the GA, and also indicates how they are handled.
- *General parameters*, like population size and maximum number of generations.
- *Initialisation parameters*, which are used to indicate how a new population should be initialised.
- *Reinforcement parameters*, which are used to take care of the reinforcement aspects of the fitness determination.

Because of the large number of parameters, the details of the GA are discussed in a separate paragraph (5.4).

Genetic operators

Elegance contains a large number of genetic operators. Some of these operators ask the user to specify parameters. Also, each genetic operator which is added to the GA needs the following attributes:

- A *name*, which is used to refer to the genetic operator. Normally this name will simply be the genetic operator name, but in case one genetic operator is added more than once to the GA, the user might wish to distinguish between them.
- A *factor*, which indicates how often the genetic operator should be selected in comparison with other genetic operators. For instance, a genetic operator with a factor of 2 will be selected about twice as much as a genetic operator with a factor of 1.
- If the genetic operator uses random values between some minimum and maximum, the user might indicate if these random values should be generated by using an *inverse exponential division* over the domain. This means that most random values will be found near the centre of the domain. This probability distribution is also used by Montana and Davis (Montana 1989), as discussed in paragraph 3.5.

  The genetic operators found in Elegance are:

- *Binary mutation*. This is the standard mutation operator, as found with the SGA (see paragraph 1.2).
- *Binary weight mutation*. This is equal to binary mutation, with the exception of the fact that the connectivity bits are not touched. This genetic operator is used by Maniezzo (see paragraph 3.5).
- *Biased weight mutation*. This is the BMW operator defined by Montana and Davis

(see paragraph 3.5). It mutates weights by adding a random value to them.

- *Unbiased weight mutation*. This is the UMW operator defined by Montana and Davis (see paragraph 3.5). It mutates weights by changing them into a random new value.
- *Biased nodes mutation*. This is the MN operator defined by Montana and Davis (see paragraph 3.5). It selects a number of nodes and changes all the incoming connections from these nodes by adding a random value to the weights.
- *Unbiased nodes mutation*. This is almost equal to the biased nodes mutation operator. The differences is that the random value is not added to the weights, but that the weights are changed into a random value.
- *Granularity mutation*. This genetic operator is used by Maniezzo (see paragraph 3.5). It changes the weight length of a chromosome.
- *Connectivity mutation*. This genetic operator is used by Maniezzo (see paragraph 3.5). It flips some of the connectivity bits of the chromosome.
- *Node existence mutation*. This genetic operator either removes an entire node from the chromosome by setting the connectivity bits for all incoming and outgoing connections to zero, or it activates all connections to and from a node by setting the connectivity bits to 1.
- *Learning rate mutation*. This genetic operator changes the learning rate which determines the slope of the activation function to a random value between a minimum and maximum set by the user. All learning rates start on a random value between a user-specified minimum and maximum, but can be changed by this operator.
- *One-point crossover*. This is the standard crossover operator, as found with the SGA (see paragraph 1.2).
- *Two-point crossover*. This is equal to the one-point crossover operator, except for the fact that two crossover points are chosen and that only the middle parts of the chromosomes are exchanged.
- *Uniform crossover*. This is the standard uniform crossover (see paragraph 1.6, under "other mechanisms").
- *Half-uniform crossover*. This is the standard half-uniform crossover (see paragraph 1.6, under "other mechanisms").
- *Nodes crossover*. This is the CN operator defined by Montana and Davis (see paragraph 3.5). It is a uniform crossover which selects a parent to copy a node from, and then copies to the child all the incoming connections to that node.
- *GA-simplex*. This is the GA-simplex operator defined by Maniezzo (see paragraph 3.5).

To add a new genetic operator to Elegance, the source code the programmer needs to provide is a list of the parameters needed by the operator, a specification of the types of controllers and the types of chromosome encoding the operator works with, and a function which generates the children from a set of parents which is given to the function. The maximum number of parents a genetic operator can have is three, and the maximum number of children it can produce is two (see appendix C.2).

5.4 The genetic algorithm

The GA is the most complex object of an Elegance experiment. Because of the large number of parameters, and because the parameter settings influence the way the evolution takes place, the GA description is placed in a paragraph of its own.

As stated in the previous paragraph, the GA parameters can be broadly divided into six groups. Each of these groups will be discussed now.

Encoding parameters

The encoding parameters specify how an ANN is encoded into a chromosome. The connectivity bits are always specified binary, since there are only two possible values for each connection: it is either present or it is absent. For the weights, the encoding can be done in three different ways:

- A weight can be encoded as a *real value*.
- A weight can be encoded as a *binary value with a fixed length*.
- A weight can be encoded as a *binary value with a variable length*.

Real-valued weights have the advantage that all possible weight values can be coded into the chromosome. Furthermore, the chromosome complexity is reduced since each weight is contained in exactly one allele. A disadvantage of the use of real-valued weights is that a number of genetic operators doesn't work with real-valued weights, and that the weight values can only be changed by mutation operators.

The opposite holds for binary-valued weights: only a limited number of different weights can be encoded, and the chromosome complexity is increased because each weight is contained in a number of alleles, but, on the other hand, every genetic operator will work on binary-valued weights, and crossover operators can be used to change weights.

If binary-valued weights are used, a *weight length* needs to be specified, which indicates how many bits are used to code a weight. If the weight length is fixed, it's the same for all chromosomes in the population. If the weight length is variable, the weight length is still fixed for all the weights within a chromosome, but it may vary across the chromosomes in the population. In that case, the user needs to specify the *minimum and maximum weight length*.

For binary-valued weights, also a *lower and upper boundary for the weights* must be specified. These are used to decide which weight value is encoded by a binary string. The formula to calculate the weight is:

(13) $$W = W_L + N\left(W_U - W_L\right)/2^L$$

where W is the weight value, W_L is the lower boundary, W_U is the upper boundary, L is the weight length, and N is the number you get when you make a straightforward translation of the binary string to a natural number (for instance, 001 binary is 1 decimal, 010 binary is 2 decimal and 111 binary is 7 decimal).

For example, suppose that $L = 2$. There are only four possible weights which can be

coded then. Suppose that W_L = -1 and W_U = +1. Then the four values are -1, -0.5, 0 and +0.5, and the binary weight translations are 00 = -1, 01 = -0.5, 10 = 0 and 11 = +0.5. Note that the upper boundary can never be reached.

Fitness parameters

The basic raw controller fitness can be determined in two different ways: either the run length until a plant failure is used to generate the fitness, or the mean square error (MSE) over the run is used, that is, the mean difference between the setpoints and the actual plant output. If *time-until-failure* is the base for the fitness measure, the user must specify a target number of timesteps, that is, the number of timesteps a controller should make a plant run without failure to be considered perfect. If *MSE* is the base for the fitness measure, the user must specify a target MSE, that is, a value for the MSE which the controller should stay below to be considered perfect.

With time-until-failure, the basic raw fitness F is calculated as follows:

$$(14) \qquad F = \begin{cases} 0 & \{t = 0\} \\ f(t)/f(T) & \{t > 0\} \end{cases}$$

where t is the number of timesteps the plant was able to run without failure, T is the target number of timesteps, and the function f is either simply $f(x) = x$, or the natural logarithm. The basic raw fitness always falls in the interval [0,1].

With MSE, the basic raw fitness F is calculated as follows:

$$(15) \qquad F = 1 - \frac{\sum_{t=1}^{T} E(t)}{T \cdot E_{\max}}$$

where $E(t)$ is the controller error at timestep t, E_{max} is the maximum possible controller error, and T is the maximum number of timesteps that the trial run takes.

The controller error at timestep t is calculated as follows:

$$(16) \qquad E(t) = \sqrt{\sum_p \left[W_p \left(\frac{S_p(t) - O_p(t)}{O_{p\max} - O_{p\min}} \right)^2 \right]}$$

where W_p is the factor, or weight, for setpoint p (as explained with the controller in the previous paragraph), $S_p(t)$ is the value for setpoint p at timestep t, $O_p(t)$ is the value for plant output p (to which setpoint p belongs) at timestep t, O_{pmax} is the maximum value for plant output p, and O_{pmin} is the minimum value for plant output p.

Because of this definition for $E(t)$, the calculation for E_{max} becomes simply:

$$(17) \qquad E_{\max} = \sqrt{\sum_p W_p}$$

This definition also places the MSE-based fitness in the interval [0,1].

The raw fitness can be adjusted by using a complexity penalty and/or a premature failure penalty. The calculation for the adjusted raw fitness F_a is:

$$(18) \qquad F_a = P_f \cdot P_c^L \cdot F$$

where F is the basic raw fitness, P_f is the penalty for premature failure (this penalty is only relevant for MSE-based fitness), P_c is the complexity penalty, and L is the chromosome length. The *chromosome length* is defined as the number of active alleles in the chromosome, where each connectivity bit is counted as an active allele, and the alleles for the weights which belong to present connections are also considered to be active alleles.

F_a is the raw fitness as Elegance reports it. The raw fitness may be scaled to get a scaled fitness which is used in the program. Scaling can be done by performing scaling or ranking (both these techniques are discussed in paragraph 1.6).

The technique known as *scaling* adjusts the fitness so that the average fitness stays the same, but the maximum fitness becomes a user-defined multiple of the average fitness. All other fitnesses are remapped using a linear equation that is explained in paragraph 1.6.

The technique known as *ranking* sorts the population in order of decreasing fitness, and then remaps the fitnesses to a user-defined interval. If the interval maximum is M and the interval minimum is m, and there are N individuals in the populations, the ranked fitness R_n of the nth individual is:

$$(19) \qquad R_n = M - (n-1)\frac{M-m}{N-1}$$

Since the control loop may contain some randomising elements, the calculated fitness may vary a bit for a controller. This is not very important, unless it concerns the fitness calculation of a ‘winning’ controller, that is, a controller that succeeds in reaching the target. In that case, the user may choose that the fitness calculation is repeated once again, to make sure we have indeed a winner.

The only fitness-associated question that remains is, what should be done when the fitness is MSE-based but the plant fails before the control loop run is finished (the formula assumes an error for each step of the run)? This is considered to be a concern of the reinforcement aspect of the GA, and it will be answered when the reinforcement parameters are discussed, later in this paragraph.

Genetic operator parameters

The genetic operators parameters are concerned with the handling of the genetic operators, which lead to the production of new individuals for the population.

The user may specify if he wishes to use *adaptive mutation*. Adaptive mutation is a technique devised by Whitley (Whitley 1993), which adapts the chance that crossover is selected according to a crossover probability which is kept with each chromosome (see paragraph 4.5 for a detailed description of this technique). Whitley has used adaptive

mutation with just one mutation and one crossover operator. In Elegance the number of genetic operators is variable, and the crossover probability determines whether the choice is between the crossover operators or between the mutation operators. A genetic operator is considered to be a mutation operator if it works with just one parent chromosome; otherwise it's a crossover operator. If adaptive mutation is used, the user needs to specify the minimum and maximum crossover probability, and the rate of change of the crossover probability.

Incest prevention can be used to enforce that parents which are selected for crossover differ at least by a specified number of alleles. If adaptive mutation is used, the effect of incest prevention is not only that incestuous parents will never be used for crossover, but also that the crossover will then be replaced by a mutation of the first parent.

The user may specify if he wishes to use *elitism*, that is, if he wishes to ensure that the best chromosome in the population will not be replaced by a new individual. If elitism is enforced, and the GA finds that it still needs to replace the best individual, instead the worst individual in the population is chosen for replacement.

If the user selects *duplicate checking*, it means that the GA will ensure that no duplicates will find their way into the population. Only if it seems to be impossible to create anything but duplicates, duplicates will be allowed. In that case the algorithm obviously has converged far too much to have any hope of finding a better individual anyhow.

The user may select to perform *viability checking*. This is only useful for neurocontrollers, not for PID controllers. Viability checking means that the GA will make sure that for every output node there is a flow which begins at at least one of the input nodes.

After mutation, only one child will be produced. After crossover for most operators two children will be produced. The *after crossover* selection specifies what should be done with the children: either both children will be placed in the population, or the fittest of the children will be selected, or a random child will be selected.

If a layered feedforward neural network or a recurrent neural network is used as controller, something can be done about the occurrence of *competing conventions* (see paragraph 3.3). The method implemented in Elegance is the method defined by Thierens (Thierens 1993), which is explained in paragraph 3.3. The reason that feedforward neural networks (without layers) are not supported by the GA for the competing conventions treatment, is that the method shuffles nodes in the network without changing the network architecture, and in a feedforward neural network normally there are not that many nodes that can be shuffled, and it is difficult to find out which nodes can.

General parameters

There are just a few general parameters in the GA. There is the *population size*, and the *maximum number of generations* which will be generated if the target is not reached. Since new individuals replace existing individuals in the population, the standard definition of a generation can not be used. Instead, a generation is defined to be the period wherein as many new individuals are generated as there are individuals in the

population.

The *replacement policy*, which indicates how new individuals are inserted in the population, is also a general parameter. The user may choose one of the following:

- The least fit individual is replaced.
- A random parent is replaced.
- The least fit parent is replaced.
- *Crowding* is applied.

With crowding, introduced in paragraph 1.6, a user-defined number of individuals is selected randomly from the population, and either the least fit of these individuals, or the individual which is closest to the new individual is replaced.

Initialisation parameters

The initialisation parameters specify how the individuals of the first generation are generated.

The user must specify a weight minimum and maximum, and the chance that an optional connection is present. The division of the weights across the interval defined by the minimum and maximum can be either linear or inverse exponential. The latter is also applied by Montana and Davis (paragraph 3.5).

For the learning rate, the user must also specify a minimum and maximum. If a fixed learning rate is required, these must be set to the same value.

Reinforcement parameters

The reinforcement parameters of the GA are concerned with the reinforcement aspects of the evolution run, that is, with the length and evaluation of the control loop test run which is used to determine the fitness.

At the start of the control loop, the *initial plant state* can be *randomised*. The user may specify which percentage of the legal interval of the plant output parameters values should be used to determine a random value for the output value, and also if this random value should be spread either linear or inverse exponential across the interval.

Since due to random characteristics in the control loop run the determined fitness value may be not quite correct, the user may also indicate that the *fitness value* should be *redetermined* after the individual for which the fitness value has been calculated has been in the population for a certain number of generations. This ensures that individuals which got a fitness value which was higher than they actually deserve are periodically re-evaluated.

A value can be set for the *total error* during an evolution run for which a failure will occur. Note that an MSE-based fitness can be replaced with a time-until-failure-based fitness by setting the value of this total error to the target number of timesteps times the target MSE (for instance, if the target MSE is 0.01, time-until-failure can be used with a target number of timesteps of 10000 and a total error of 100).

For time-until-failure based fitness, there are no more reinforcement parameters, since the length of the run until a failure is all that matters. For MSE-based fitness, there

are several more parameters which should be specified. I have found no literature about MSE-based fitness, so these parameters are something I thought up myself.

We could decide that every control loop run should be of the same length. It is difficult to decide how long that run length should be. If we choose a value that is too big, the control loops will take more time than necessary (and evolution is already a slow process). However, if we choose a value that is too small, the fitness determination will be unreliable. Therefore, the following method of determining the control loop run length is implemented:

Each control loop starts by running a *fixed number of timesteps*. After that, the *run length* is *increased* again and again by a user-definable number of timesteps, until since the last increase the *change in MSE* falls below a user-defined minimum, or a *maximum number of timesteps* is reached. This seems a reasonable implementation, since it prevents the run from continuing when the MSE doesn't change that much anymore, and it also prevents this criterion from extending the run length for a virtually endless period. If the run doesn't continue for the maximum number of timesteps, the controller error for the rest of the timesteps is considered to be the final MSE.

A second decision which the user must take is what should be done when the plant fails before the control loop finishes on one of the previous criteria. Elegance offers these possibilities:

- Awarding the maximum possible controller error for all missed timesteps. This means that the controllers that make plants fail will have a quite a low fitness, and probably *ranking* as a fitness scaling technique would be a wise choice.
- Awarding the MSE until the failure for all missed timesteps. In that case a premature failure penalty would be in order.
- Resetting the setpoint generator, controller and plant and continuing the run. This obviously will take a lot of time.

5.5 Implementation

Tools

As stated before, most important for the implementation of Elegance were maintainability of code and a user-friendly interface. Speed was considered to be less important, although it should play some role during the implementation process since speed is always quite a problem with GA evolution.

As implementation platform Microsoft Windows was chosen. This choice was not based on the qualities of this environment, since, in my personal experience and opinion, it is slow, unstable, badly designed and inconsequent. However, despite its weaknesses Windows is probably the best-known and most widely accepted environment with a graphical user interface on the PC-market. The design of Elegance is based on Jarmulak's program NCWB which runs under MS-DOS, and, although Jarmulak has tried his best to make the program as user-friendly as possible, the user still needs to

study a user manual and to practice for an hour or two with NCWB before he can use it with some ease. Windows offers more possibilities for making the program easy to use.

As development tool the choice fell on Borland Delphi 1.0. Delphi was first available in the spring of 1995, and it immediately became very popular, because it is easy to use, flexible, fast, truly object oriented and very powerful. Since I needed to build some experience with Delphi for my daytime job, and my first encounters with it showed it to be a really excellent tool with exactly the power I needed to build Elegance, the choice was obvious. Now the first version of Elegance is finished, I can say that I am not disappointed in this tool. In fact I am outright enthusiastic about it.

Hardware requirements

The machine I used for development was a Pentium 90 PC with 16Mb of internal memory, and also a portable 486DX/25 PC with 8Mb of memory. Furthermore, a 486DX/33 PC with 8Mb of memory was used for a large number of the simpler tests.

In my experience the minimum PC configuration which runs Elegance reasonably smooth is a 486DX/33 PC with about 4Mb of memory, running Windows 3.1. Processing power is very important, and since almost every calculation uses floating-point numbers a mathematical coprocessor is an absolute must. As for the screen, Elegance will run fine with a minimum screen resolution of 640x480, which is the Windows minimum.

Elegance doesn't require that much memory, although the user should make sure that the program doesn't start swapping in virtual memory. Windows itself will use between 1 and 2 megabytes of memory. The memory requirements of Elegance depend on the length of the chromosomes and the size of the population. For not-too-complex experiments 2Mb of free internal (not virtual) memory, probably less, should be more than enough for the program.

The more calculations the program needs to make to determine the fitness of a chromosome, the slower the program will be. The number of calculations is very much dependant on the size of the neural network which is trained. If the neurocontroller contains 15 hidden nodes or more, I would definitely recommend using nothing less than a Pentium PC. And even then, count on an evolution run that will take the better part of a day, if not more.

Of course, the number of cycles the evolution needs to run also depends on the GA parameters chosen. A wrong choice will dramatically increase the time needed to finish the run successfully, if it will finish successfully at all. And since the purpose of Elegance is to determine what exactly *is* a good configuration, the user should still fear long evolution runs for the bigger problems, even on a fast machine.

Software requirements

Elegance needs at least Windows 3.1 to run. It will also run under other Windows-compatible environments, like Windows 95, Windows NT and WIN-OS/2. Delphi 1.0 source code can be recompiled under Delphi 2.0 to get native Windows 95 executables, but I discovered that Delphi source code is not entirely upwards compatible. Therefore, the first version of Elegance is only available as a 16 bit application. In the near future, I

intend to make the Elegance source code suitable for the Delphi 2.0 compiler to create a 32 bit version of the program.

Interface

The interface of programs which run under Windows is varying a lot from program to program. There are extensive books which prescribe how a Windows interface should look and work. However, many design problems are not covered by those books, and even Microsoft solves certain problems differently in different programs.

In the design of the Elegance interface, I have tried to conform to the interface designs of the most common Windows programs, like Word and Excel. I haven't always succeeded in that (mostly in minor details), but I've tried to at least make Elegance itself consequent in its interface. Users who are familiar with standard Windows programs shouldn't have any trouble with Elegance.

The two main diversions I made from the common expectations a user has of a Windows program are:

- The Elegance menu bar cannot be reached from program windows, because that would offer to the user the possibility to open more than one maintenance window at a time, and for implementation reasons that is not desirable.
- The Elegance windows are not resizable, simply because that wouldn't be very useful in the program, and if it was allowed a lot more code would be needed to make it a useful feature.

5.6 Using the program

The previous paragraphs gave an overview of the functional design, technical design and implementation aspects of Elegance. This paragraph should give an idea of how the program is used in practice.

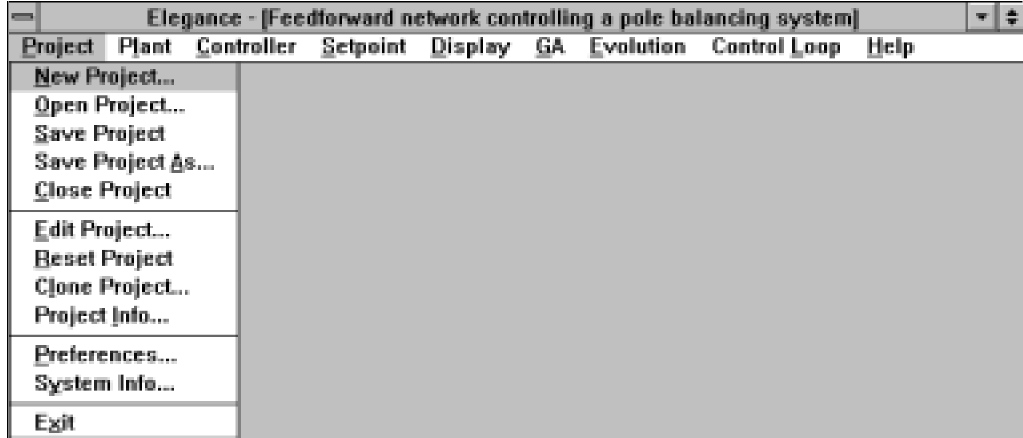


figure 47: The Elegance menu bar. The first six item are the maintenance menu's for the program's basic experiment element, the next two (evolution and control loop) are the main processes, and the last choice offers global access to the help section.

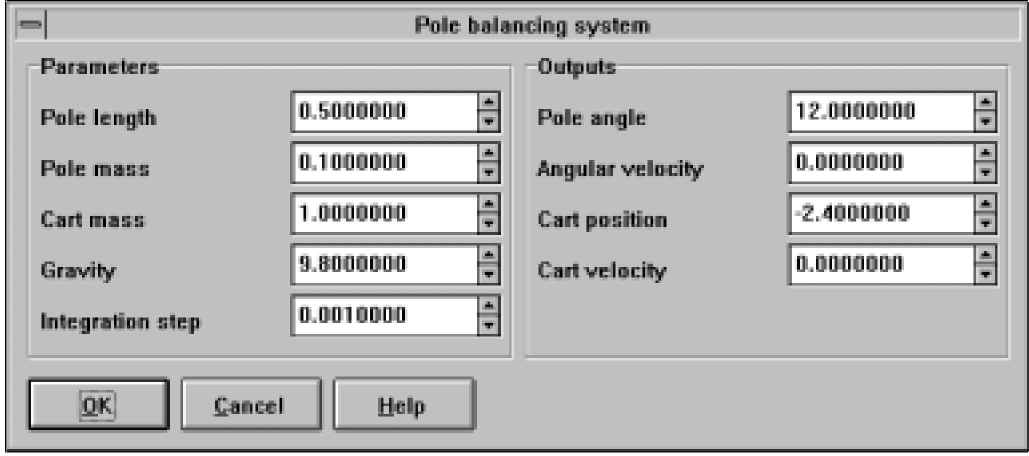


figure 48: An example of an Elegance editing screen: the pole balancing plant.

When the program is started, an empty screen with only a menu bar on it is shown (figure 47). Normally, the user will work through the menu’s from left to right. Choices will become available at the right time.

The first six choices on the menu bar are the maintenance menu’s for the basic objects of an experiment: the project, plant, controller, setpoint generator, control loop display and GA. With the *New* choices, the user can create a new instance of such an object, which is immediately followed by a screen on which the newly created instance

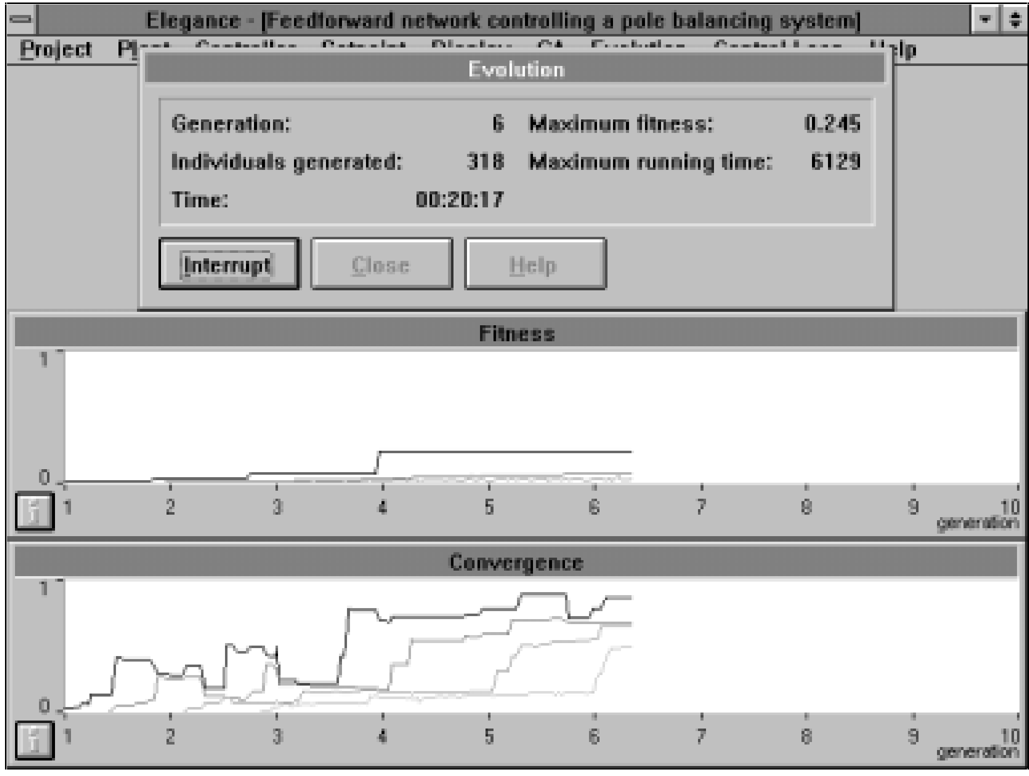


figure 49: The Elegance evolution run. A feedforward neural network is evolved to control a pole balancing system.

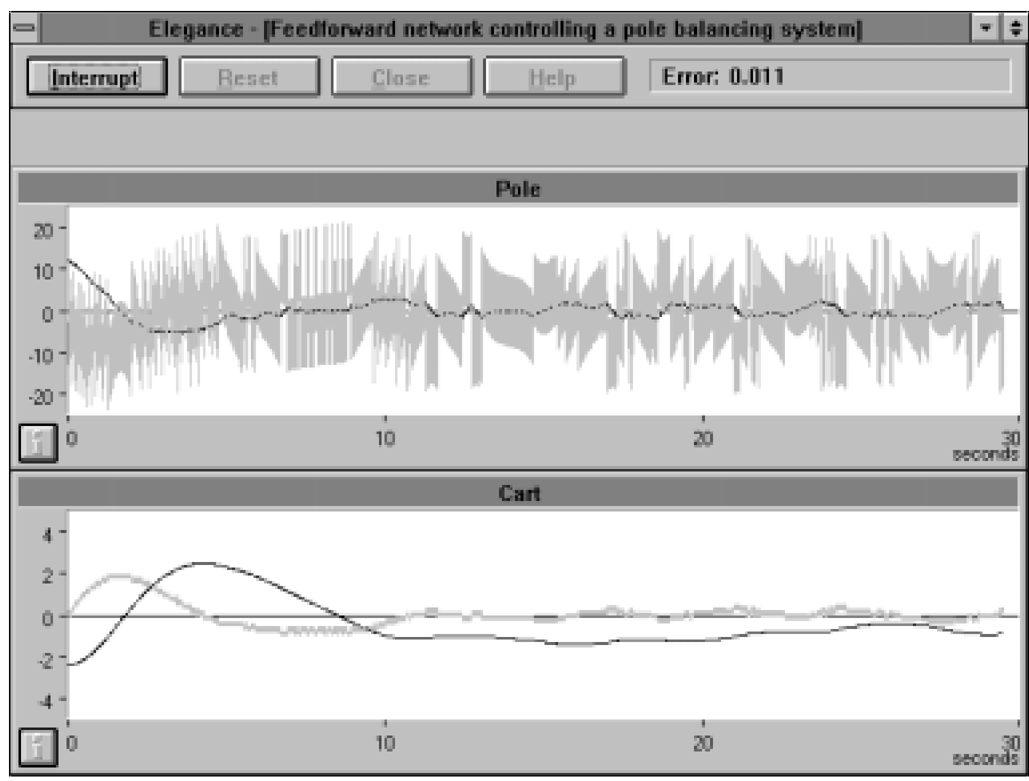
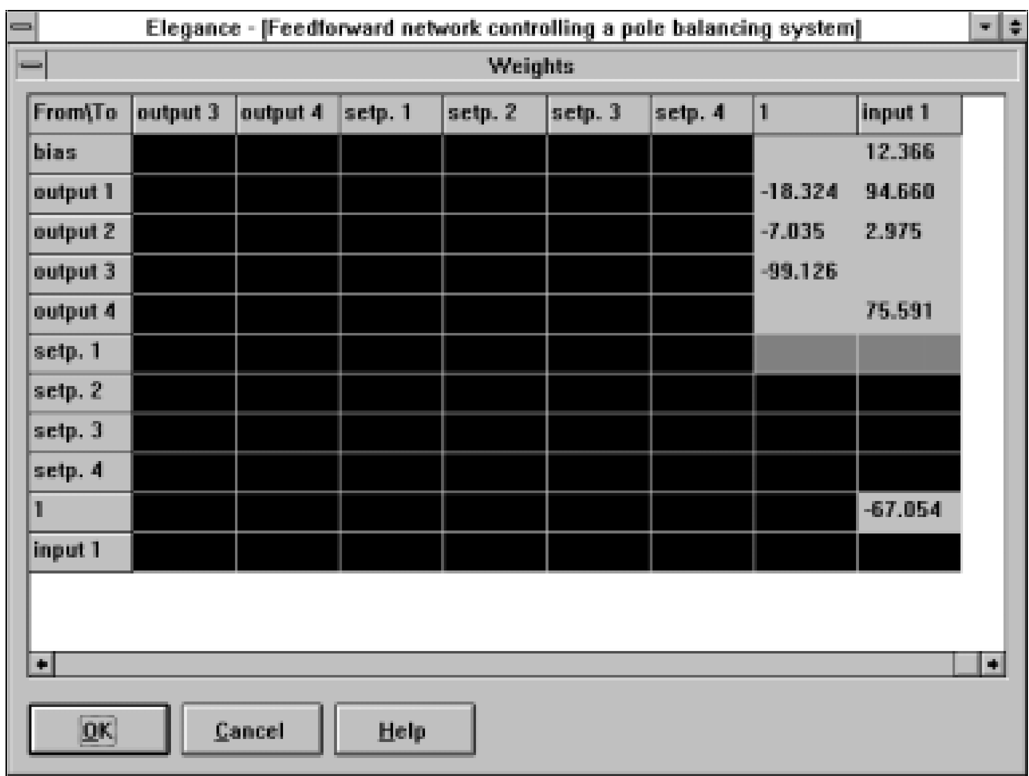


figure 50: The Elegance control loop. Two user-defined charts are shown, which present the performance of a neurocontroller on a pole balancing system. The upper chart shows the pole angle in degrees as the black line, and the grey, wildly oscillating line, is the angular velocity of the pole. The lower chart shows the cart position as the black line and the cart speed as the grey, mildly oscillating line.

can be edited. This edit screen is also accessible with the *Edit* choice. If a user wants to re-use an object already created with another project, he can use the *Open* choice to get it. Each menu may contain several more choices, for instance for testing an object instance. The project menu also contains some general program options.

Editing an object works the same way every time. A screen is presented, with on it the attributes of the element. The user may edit these attributes, and after he is finished, he may either accept his changes by pressing *OK*, or reject the changes by pressing *Cancel*. As an example, the screen belonging to the pole balancing plant is shown in figure 48.

When all elements are created, the evolution menu can be accessed. This menu contains choices to initialise and deinitialise a population, to run an evolution, to examine a population, and to update the controller with the values specified by one of the chromosomes in the population. During the evolution run, a screen is shown which presents information about the way the evolution is going (figure 49). The information on this screen can be very instructive. The text screen informs us about the length of time the evolution is running, how many individuals have been generated, and the exact status of the best individual currently found. The fitness graph shows the development of the fitness over the generations, and the convergence graph shows how much the convergence is among several sections of the population. If the bottom line of the convergence graph, which shows the convergence among all the individuals in the



From\To	output 3	output 4	setp. 1	setp. 2	setp. 3	setp. 4	1	input 1
bias								12.366
output 1							-18.324	94.660
output 2							-7.035	2.975
output 3							-99.126	
output 4								75.591
setp. 1								
setp. 2								
setp. 3								
setp. 4								
1								-67.054
input 1								

figure 51: This matrix shows all the connections of a pole balancing neurocontroller with one hidden node, developed by Elegance. On the vertical axis, the nodes are shown *from* which the connections come, and on the horizontal axis the nodes *to* which the connections lead. Node *output 1* is the pole angle, node *output 2* is the angular velocity, node *output 3* is the cart position and node *output 4* is the cart speed. Node *1* is the sole hidden node. Node *input 1* is the control force which is the input for the plant. The black cells are connections which are always absent, because of the network structure chosen. The dark grey cells are connections which the user has marked "always absent" (in this case, the setpoint input is removed this way, since this setpoint is always zero and as such the setpoint node isn't useful). The grey cells without a weight value are connections which are possible, but which the evolution process has removed. The grey cells with a weight value in them are present connections.

population, gets too high, the user knows it won't be of much use to let the evolution continue its run.

When the user thinks the time is right (mostly when the evolution has finished successfully), he can update the controller with one of the chromosomes (usually the best) and observe the performance of this controller by running a control loop, during which he can examine the screen he himself has defined (figure 50).

The exact controller structure can be examined in the *Controller* menu. The controller is presented here as a matrix, which shows all present connections with their weights (figure 51).

This straightforward way of working makes the program seem rather simple, and basically it is just that, but it has many features which make it easier to use. For instance, the user can start a run of a range of experiments, statistics on an evolution run can be saved to a file, the user can specify a number of preferences and there is an extensive context-sensitive help system.

5.7 Evaluation

User-friendliness

At the moment of this writing, the only person who has had any experience with Elegance am I. Since I, being the author of the program, know every nook and cranny of it, I am not a very good judge on the user-friendliness of Elegance. Based on my years as a professional programmer, however, I feel that the program is pretty simple and self-explanatory to use, and I think that those are the most important aspects of user-friendliness.

Usefulness

A more important matter is, is Elegance useful? My trust in evolution and GAs is so big, that I didn't doubt the fact that Elegance would be able to develop an acceptable neurocontroller. However, I had serious doubts if the performance of the program would make it a serious rival to conventional techniques like backpropagation. When implementing the program, being confronted with the complexities of the implementation, those doubts started getting bigger and bigger. I began imagining that the evolution of even a simple neurocontroller would take days.

Much to my relief, my doubts weren't confirmed. After most of the bugs had been removed, the program started evolving very good neurocontrollers for the simpler plants in just a few minutes time. True, that was on a Pentium PC, but I found that even on a 486DX/33 PC the evolution process most of the time finished successfully within an hour.

I've done a whole bunch of preliminary experiments with Elegance, some of which are discussed in the next chapter. The flexibility of the program is such, however, that I could only research a very limited number of ideas and situations. Elegance can obviously be used for many more experiments.

Elegance has shown me that at least genetic reinforcement control is something that's worth looking into, and that makes Elegance a program that certainly can have value in the research process. More about this will be discussed in the next chapter and in the conclusion to the third part.

Stability

Concerning the stability of the program, I've found it runs very stable on many different PCs under Windows 3.1. I haven't had a lot of access to PCs running other versions of Windows, so I can't comment very well on those, but I trust that it should work just as well on those PCs.

To make sure that even in unstable situations work doesn't get lost, Elegance contains an autosave option which saves the population during evolution every generation, or even, if the user desires so, after every change in the population.

5.8 Future versions

Preliminary experiments show Elegance to be a program with some promising features. If it will be used some more, there definitely will be one or more new versions of the program. Commentary by users will be used to enhance the program, fix bugs and upgrade the program's possibilities. Some features I'd like to add are the following:

- Parallel processing support, as soon as it can be done on a PC, should be incorporated in the program. GAs are very suitable for parallel processing, and it could lead to vast increases in speed, which the program sorely needs.
- I have made virtually no attempts to speed up the code. Several procedures could be redesigned to run faster.
- The current maximum of neural nodes in a controller, set to 75, should be increased or even be eliminated.
- Other methods of combating competing conventions can be implemented.
- A conventional technique for local optimisation could be very useful.
- A wizard should be added to completely create a new project, including the GA, based on a few user-defined selections.
- A few simple features, like drag-and-drop in list boxes, can be added.
- Printing of an ANN matrix should be added.
- Extra setpoint functions can be added (more about this in the next chapter).
- Extra ranking features can be added (only linear ranking is implemented at this time).
- Jarmulak's NCWB contains several 'general' plants, which can be used to define a wide range of plant functions. Those general plants would form a nice addition to Elegance.

5.9 Summary

This chapter discusses Elegance, a program developed to experiment with the use of GAs for the design of neurocontrollers in a reinforcement control situation.

Elegance works with projects, and each project consists of a number of objects, of which the most important are:

- A plant that is to be controlled.
- A controller, either a PID controller or a neurocontrolle, that is to be designed by the program.
- A setpoint generator, which generates the targets of the plant output during the control loop run.
- A control loop display, which shows some user-defined charts during the control loop run.
- A GA, which contains a specification of all aspects of the evolution process, including the population and the genetic operators.

The GA is quite flexible and has many features. Some of the configuration

possibilities it offers are:

- Flexible encoding: real-valued and binary valued chromosomes and chromosomes of fixed and variable length.
- Flexible fitness: fitness based on time-until-failure and fitness based on mean square error (MSE).
- Flexible genetic operator handling: many different genetic operators, adaptive mutation, treatment of competing conventions, duplicate checking, viability checking and incest prevention.
- Flexible reinforcement parameters: varying control loop run lengths and initial plant state randomisation.

Elegance has been implemented under Windows using Borland Delphi. A fast PC is recommended for running experiments with it.

First experiences with Elegance show that GA evolution of neurocontrollers is potentially a serious rival for conventional techniques which are used to design neurocontrollers. Elegance offers enough speed and flexibility to use it for further research in this subject area.

6 Elegance Experiments

This chapter will discuss some of the preliminary experiments done with Elegance. Two goals were aspired with these experiments. They should show that Elegance works, and it would be nice to show some interesting results. First, the environment and the premises for these experiments are discussed (6.1). This is followed by a description of an attempt to find something that works better than Whitley's GENITOR (6.2). As an example of MSE-based fitness, a trolley-controller is developed (6.3). As an example of a more complex problem, a neurocontroller for a bioreactor is developed (6.4). The chapter finishes with an overview of the experiments and an indication of directions for further research (6.5).

6.1 Preparing the experiments

This chapter describes some of the preliminary experiments performed with Elegance. The main purpose of these experiments was to show that the program indeed works. To obtain some interesting results would be a nice boon.

Two computers were used to do the experiments which are discussed in this chapter: a 486DX/33 PC with 8Mb of internal memory, and a Pentium 90 PC with 16Mb of internal memory. Both computers ran Windows 3.1. The 486 PC was used for the simpler experiments, which are found in paragraph 6.2 and 6.3. The bioreactor experiments were done on the Pentium.

I've tried to perform each experiment at least ten times, to make sure that extreme results would not influence the experiments too much. Unless stated otherwise, at least ten run of each experiment were performed. Also, unless explicitly stated differently, all genetic operators had a factor of 1.

6.2 A pole balancing controller

The bang-bang pole balancing system

A pole balancing system consists of a cart on a rail of half a meter long. On the cart, on one end, stands a pole, which can fall to the left and right. A controller can move the cart by applying a force. The aim of the controller is to keep the pole from falling. A detailed description of the pole balancing system is found in appendix B.6.

The version of the pole balancing system that is used for the experiments in this paragraph, is the bang-bang pole balancing system, which is mentioned in appendix B.7. In the bang-bang pole balancing system only two different forces can be applied to the cart: either -10N or +10N (even the absence of force is not possible). Also, the input signal to the plant is interpreted by the plant as the chance that +10N will be applied. This last aspect is not very common, but it is copied from Whitley's implementation (Whitley 1993).

The pole balancing system is often used in neurocontrol research. It is one of the simpler systems to be controlled, and is therefore very suitable for preliminary experiments.

Whitley's GENITOR

Darrell Whitley has experimented with the evolution of a neurocontroller for a pole balancing system (Whitley 1993). His experiment is described in detail in paragraph 4.5. Whitley uses a genetic algorithm based on the GENITOR algorithm. The characteristics of this algorithm are:

- Ranking is used as a fitness conversion technique.
- The population is rather small (50 individuals).
- Real-valued chromosomes are used.
- The genetic operators are one-point crossover and biased weight mutation, the last with an extremely high mutation rate and an enormous interval.
- After crossover, one of the children is randomly selected and the other one is discarded.
- New children replace the lowest ranking individual in the population.
- Adaptive mutation is used.

Some of the reasons Whitley gives for this configuration are:

- A small population means the competing conventions problem won't have much influence.
- Real values can span a greater range and can be more precise than binary values.
- A high mutation rate promotes diversity in the population.
- Adaptive mutation promotes mutation when the population starts to converge.

I would comment on that by pointing out the following:

- Whitley's mutation rate is so high, that he is in fact generating random new chromosomes with it. This means that the power of GAs, which is trying to preserve well-working strategies coded in the chromosomes, is weakened.
- If random individuals are generated, chances are high that they are not very fit. Since this means they get a place low in the population, and since GENITOR replaces the least fit individuals, and since fitter individuals will procreate more often, doesn't this also mean that such a randomly generated individual will very likely be removed from the population before it gets a chance to procreate? If that is true, the effect of promoting diversity is lessened.
- A small population inhibits diversity. Maybe if the population is larger, the mutation rate need not be so high. The competing conventions problem can be solved in another way.
- Real values may be more exact, but neurocontrollers do not need very precise values for the weights to work well. There's a lot of elbow-room in solution space in the neighbourhood of a good controller, judging from the fact that small changes in the weight values of a good controller often have no impact on the success of that controller.
- Adaptive mutation promotes mutation when conversion takes place, but is that really necessary? If the chance on crossover is equal to the chance on mutation, in 50% of the cases a random new individual is created. Do we really need more mutation than that?

Elegance now offers us a way to experiment with Whitley's GENITOR, to see if it indeed works well, if it can be improved upon, and if the comments above hold water.

The experiment parameters

The controller I tried to evolve for controlling the bang-bang pole balancing system was a feedforward neurocontroller with five hidden nodes, with a sigmoid as activation function with a learning rate of 1 and a range of [-1,+1]. TDLs were not used and the connections from the setpoint input node were removed (these connections are superfluous, since the input from this node is always zero). Five hidden nodes is rather a lot, since it can be done with just one node, but five nodes may develop into a better controller.

The starting situation of the plant was the cart positioned unmoving at -2.4m, and the pole at an angle of 12 degrees. The rest of the plant parameters were default. The timestep size was 0.02.

The controller was considered to be perfect if it could keep the plant from failing for 20,000 timesteps. Whitley set this value to 120,000, but preliminary experiments showed that if a controller could do it for 20,000 timesteps, it could almost certainly do it for an unlimited time. Since it costs a lot of time to test a controller for 120,000 timesteps, I decided that I would just leave it at the lower value.

Furthermore, I decided that the controller should be evolved within the generation of 350 individuals after the initial population had been generated. For Whitley's

GENITOR, with a population size of 50, this meant that eight generations should be sufficient. This was because I found that GENITOR seldom needs more than that, and other configurations shouldn't do worse than GENITOR. However, it also meant that sometimes an experiment would fail that would possibly have succeeded if a few more individuals could have been generated.

General remarks

During the experiments, I found that it often would happen that after the building of the initial population a perfect controller would have been generated. This happened on no less than eight occasions in 90 experiments. This means that about 0.2 percent of all randomly generated chromosomes already encode a perfect pole balancing controller. This, in my opinion, indicates that the pole balancing system is maybe a bit too simple to be a good subject for evaluating the validity of the application of GAs in neurocontroller building. Simply generating a few hundred random controllers would also work. It is therefore no surprise that Whitley's approach, which is, as already indicated, for a large part a purely random process, works well.

I also found, during the experiments, that results of most configurations would vary a lot over the experiments. Only a few were consistently bad, and none of the configurations was consistently good ('good' meaning that a perfect controller would be generated within the generation of 200 individuals). Because of the random effects, ten experiments with each configuration is evidently not enough to draw any firm conclusions about a specific method. It is, however, enough to indicate which techniques are promising.

Testing GENITOR

The first series of experiments concerned, of course, GENITOR. The parameters of these experiments were:

- *Encoding*: Real values.
- *Fitness*: Time-until-failure, target 20,000 timesteps. Ranking with interval [1,50].
- *Operators*: Adaptive mutation with interval [0.05,0.95] and a change rate of 0.1. Incest prevention with three alleles. Selection of a random child after crossover.
- *Parameters*: Population size 50, replacement of least fit individual.
- *Initialisation*: Weight initialisation on interval [-2.5,+2.5], connection existence chance 0.5.
- *Reinforcement*: Default.

and genetic operators:

- *Biased weight mutation* within an interval of [-10,+10], with a chance of 1.0 for each allele.
- *One-point crossover*.

There is no question that Whitley's GENITOR works, at least on this simple configuration. Of the ten experiments, six finished successfully within the generation of 200 individuals, and only one of the experiments failed. Convergence was kept low.

To test the effect of several of the parameters, I changed some of them and ran new series of experiments. As I stated before, it seems to me that the effect of adaptive mutation in this configuration is negligible. So, a logical test is to leave every parameter the same as before, but to remove adaptive mutation.

The result of this second series of ten experiments was that five finished successfully within the generation of 200 individuals and just one failed. Convergence was kept quite low. This result confirms my expectation that adaptive mutation does not add much to the success of GENITOR. It may be, however, that in more complex experiments it does help, especially if the mutation operator is not so much a randomisation operator as a real mutation.

To see if maybe one of the genetic operators was superfluous, I ran two series of ten experiments, both equal to the second series, but one with only the crossover operator and one with only the mutation operator.

The experiment with only mutation finished six times within the generation of 200 individuals, and failed twice. Convergence was kept very low. At first, this success seems unwarranted. Simply generating random individuals seems to be a successful strategy. The explanation for this result is already given above: pole balancing is too simple to need more than a random search. The result of this experiment confirms that.

The experiment with only crossover three times finished successfully within the generation of 200 individuals, and failed three times. Convergence couldn't be kept

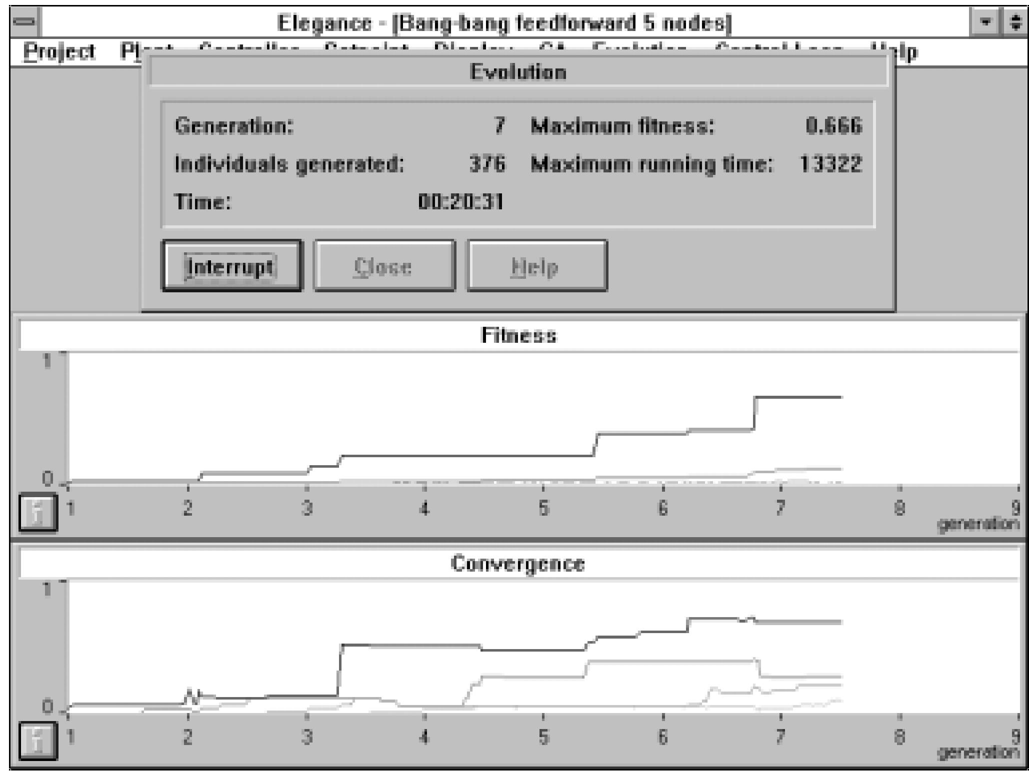


figure 52: An evolution run of a five-hidden-nodes neurocontroller for a bang-bang pole balancing system.

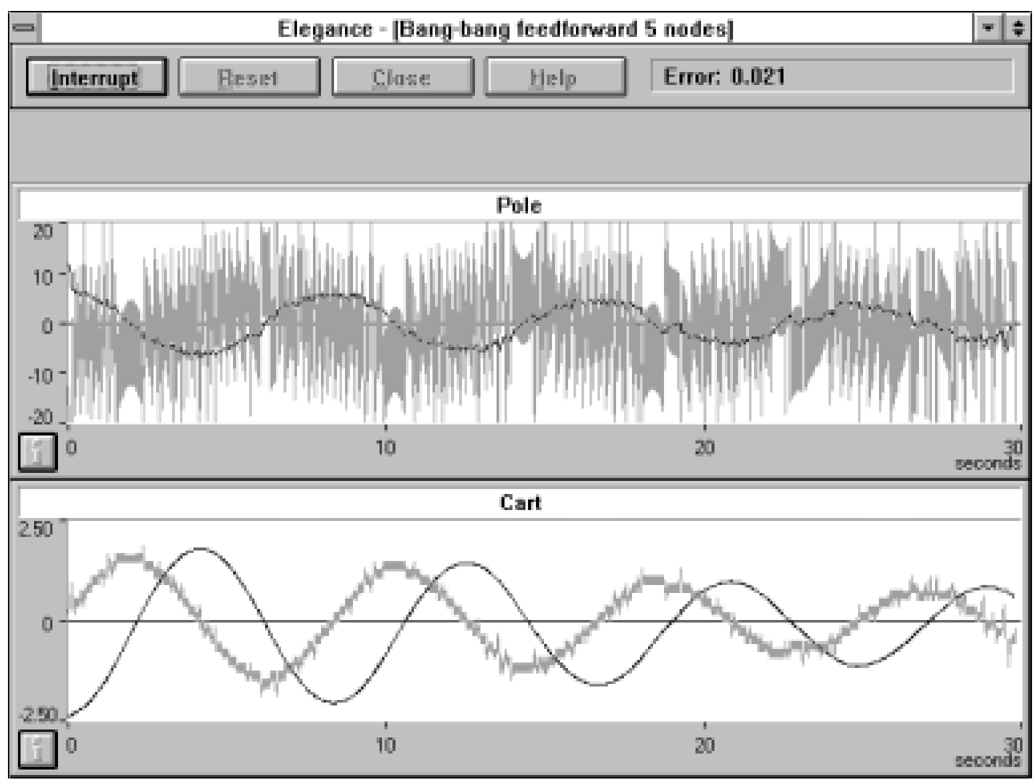


figure 53: A bang-bang pole balancing control loop. In the *Pole* chart, the black line shows the pole angle and the grey, wildly oscillating line the angular velocity. In the *Cart* chart, the black line is the cart position and the grey, mildly oscillating line the cart speed. This is certainly not the best bang-bang pole balancing controller possible. It is very well possible to get the pole to stabilise within a range of one or two degrees within a few seconds.

low. This result shows that a configuration with one-point crossover as the sole genetic operator might work, but only by chance. This is noteworthy, because in the standard genetic algorithm (see, for example, the simple genetic algorithm in paragraph 1.3) the crossover operator is considered to be far more important than the mutation operator.

I thought that maybe, if I replaced real-valued chromosomes with binary-valued chromosomes, one-point crossover as the only operator might work better, since the crossover can cause changes in the weights that way. This series of experiments failed no less than five times, however, so the slight weight changes invoked by the crossover operator in this case were not enough to battle the bad sides of this configuration.

Testing Maniezzo's ideas

Maniezzo's ideas (see paragraph 3.5) were not targeted at neurocontrol evolution, but his principles are incorporated in the following configuration:

- *Encoding*: Binary values, variable weight length between 1 and 16 bits. Weights within an interval of [-100,+100].
- *Fitness*: Time-until-failure, target 20,000 timesteps. Scaling with a scaling factor of 2, and logarithmic fitness remapping.
- *Operators*: Elitism, duplicate checking. Incest prevention with three alleles. Keep all

children after crossover.
- *Parameters*: Population size 100, crowding with a crowding factor of 2 and replacement of the least fit of the selected individuals.
- *Initialisation*: Weight initialisation on interval [-100,+100], connection existence chance 0.5.
- *Reinforcement*: Default.

and genetic operators:

- *One-point crossover* with a factor of 0.04.
- *Binary weight mutation* with a factor of 0.18, with a chance of 0.0005 for each allele.
- *Granularity mutation* with a factor of 0.18.
- *Connectivity mutation* with a factor of 0.18, with a chance of 0.1 per allele.
- *GA-simplex* with a factor of 0.9.

This configuration succeeded six times within the generation of 200 individuals, but it also failed three times. Convergence didn't seem to be any worse than with GENITOR. The failures can be explained by the fact that GA-simplex is in Maniezzo's algorithm by far the most executed operator. Since GA-simplex is meant for local optimisation, the effects will be only beneficial for chromosomes which are near but not at an acceptable configuration. With pole balancing, such an optimisation operator is not needed, since random jumps seem to work very well. The effect of GA-simplex in this configuration is probably not very beneficial, and therefore may slow the process down, unless the initial population already contained a good chromosome.

We may conclude that, at least with pole balancing, GENITOR works better than Maniezzo's ideas. However, Maniezzo's ideas may work better with more complex experiments.

Testing other ideas

I've tested a few ideas of my own. I've added complexity penalties and tried other selection mechanisms, scaling mechanisms, and genetic operators. It's not very useful to discuss all these experiments. A very successful one was the following configuration:

- *Encoding*: Real values.
- *Fitness*: Time-until-failure, target 20,000 timesteps. Ranking with a range of [1,100].
- *Operators*: Elitism, duplicate checking, viability checking. Incest prevention with three alleles. Keep all children after crossover.
- *Parameters*: Population size 50, replace least fit individual.
- *Initialisation*: Weight initialisation interval [-100,+100], connection presence chance 0.5.
- *Reinforcement*: Default.

and genetic operators:

- *Biased weight mutation* with a chance of 0.1 per allele within a range [-10,+10].
- *Connectivity mutation* with a chance of 0.1 per allele.
- *One-point crossover*.
- *Nodes crossover*.

This configuration finished successfully seven times within the generation of 200 individuals, and failed just once. The results seemed to be a bit better than those of GENITOR. However, for the failure and the two experiments which ended in more than the generation of 200 individuals, convergence had started to rise more than seemed healthy. It is to be expected that for more complex experiments, convergence might normally rise too high to finish successfully. This convergence problem was in fact the case with every of the ideas tried.

Ten nodes experiments

To see if the configurations could also be used in experiments which were a bit more complex, I tried some of them on neurocontrollers with ten hidden nodes. Also, I let the process run until an acceptable controller was reached.

GENITOR still performed very well: eight experiments finished within the generation of 200 individuals, and only one needed more than 300 (it needed 740 to be exact). This indicates that, as far a GENITOR is concerned, the complexity of the neural network doesn't make much difference.

Other experiments showed results comparable to those of GENITOR. Only one of them was exceptional, although not in a positive sense. This configuration was equal to the configuration specified above, under the heading "Testing other ideas", except that a population of size 100 was chosen, and crowding with a factor of 2 and 'least fit'-replacement was used as a replacement policy. This configuration needed very long training times. Only two experiments finished within the generation of 200 individuals, and six needed more than 300. Something in this configuration makes it a very bad design, although to me it isn't clear what. More experiments are needed to sort this out.

Conclusions

The pole balancing system is too simple a plant to show if the application of GAs in neurocontroller evolution is a viable approach. The experiments with this system, however, do indicate some general rules:

- Keeping diversity in the population is important, otherwise convergence may lead to very long evolution sessions, which may be unsuccessful even in the long run.
- GENITOR works well, at least on simple configurations, but many other configurations work equally well.
- Crossover and mutation both are important to the effects of a good GA.
- Real-valued chromosomes seem to work a bit better than binary-valued chromosomes. The advantages of binary-valued chromosomes don't seem to be that influential.

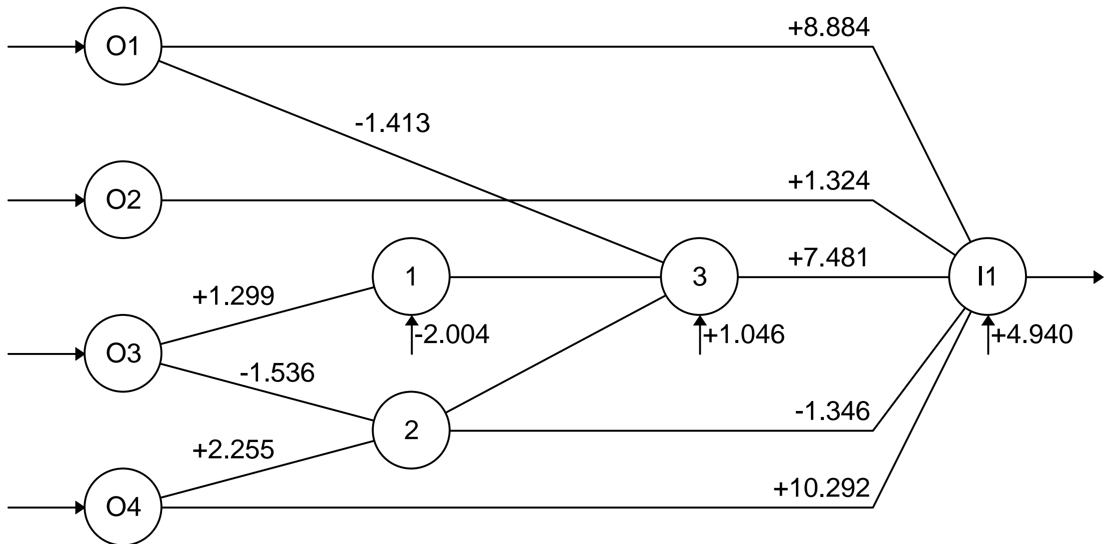


figure 54: A bang-bang pole balancing neurocontroller with three hidden nodes evolved by Elegance. *O1* is the pole angle, *O2* is the angular velocity, *O3* is the cart position, *O4* is the cart speed, and *I1* is the input to the plant. This input gets transformed by a sigmoid with learning rate 1 to a value between zero and 1, and is interpreted as the chance of applying force +10N to the plant. Otherwise, -10N is applied. This particular neurocontroller performs quite well: although the cart takes a long time to get to the centre of the rail (which, incidentally, isn't a requirement, but it often works out that way), the pole gets stabilised within a few timesteps. The hidden nodes 1 and 3, and the output node *I1*, have a bias indicated with the arrow which points to them from below.

- The effect of adaptive mutation does not seem to be of much benefit to the GA.

This does not answer all the comments I stated on GENITOR above. However, it is far too easy to develop a good neurocontroller for a pole balancing system to draw any conclusions about GENITOR from these experiments. More complex experiments need to be done with GENITOR to find out if it really works better than other configurations.

6.3 A trolley controller

The trolley

The trolley is a cart on a piece of rail of half a meter long, which is driven by an electromotor. The controller sends signals to the motor and that way can steer the trolley to specified positions on the rail. The trolley is described in detail in appendix B.4.

During the control loop, failing is not allowed, but *not* failing in itself does not make the controller acceptable. The trolley is therefore an example of a system for which the controller must be evolved using MSE-based fitness.

Our aim is to develop a trolley controller which can direct the trolley to any position on the rail, within a range of [-0.2,+0.2]. The positions the rail offers fall in a range of

[-0.25,+0.25], but if we wouldn't allow for a bit of leeway, it seems that the PID controller, which will, in theory, never make the cart cross the target position, is the best solution to the problem.

The experiment parameters

Since our goal is to develop a controller which can direct the cart to *any* position within a range of [-0.2,+0.2], the most suitable setpoint function seems to be a random wave with a fixed interval, with a minimum of -0.2, a maximum of +0.2, a period of 1 (so that every second another setpoint is indicated) and a minimum change of 0.05.

For the fitness calculation a target MSE must be set. This poses a small problem. If a training set is used to train a neural network, in theory the network can become perfect, with an MSE of zero. In practice, neural nets are trained to get the MSE below a small value, for instance 0.01. If the network has a suitable architecture, in principle any small value for the MSE can be reached.

With reinforcement learning, there is a certain minimum value for the MSE which can not be reached. Observe, for example, the trolley. The MSE is the mean difference between the setpoint and the trolley position. At the moment the setpoint changes, the error gets large, and therefore the MSE increases.

Suppose we would use as a setpoint function a square wave with a minimum of -0.2, a maximum of +0.2, and a period of 2 (so that every second another setpoint is indicated), and that the timestep size is 0.02. Even if the trolley could move immediately to the target setpoint and stop dead there, every second there would be a timestep with an error of 0.8 (if the weight for the setpoint is 1 - for a calculation of this error, see formula 16). The MSE would thus become 0.8 / 50 = 0.016. Note that this is the MSE for a perfect controller, one which needs no more than one timestep to get the trolley to its target setpoint. We haven't even considered the possible physical impossibilities of this feat: since the trolley has a maximum speed of 10 meters per second, it can drive no more than 0.2 meters in 0.02 seconds, meaning that it takes at least two timesteps to make the trolley cross 0.4 meters, which is the difference between the minimum and maximum setpoint. And then there are things to consider like the time it takes for the trolley to reach its maximum speed, the time it needs to apply the brakes, etcetera.

When testing with the trolley system, I found that a target MSE of 0.02 is reachable (with a random wave with a fixed interval), although it can take a very long time. A target MSE of 0.025 is a lot easier to reach. During the trolley experiments I set the target MSE to 0.025, and the maximum number of generations to 50, so that the evolution should have enough time to get to this target.

Because I found that it was very difficult to evolve a neural network with five hidden nodes as a trolley controller, while it was easy to do this with a neural network with one hidden node, I decided to run a series of experiments starting with one-hidden-node networks, and then increasing the number of hidden nodes until a satisfying configuration had been found. Because I used so few hidden nodes, the neural network configuration chosen was a simple feedforward neural network, with a sigmoid with a learning rate of 1 and a range of [-1,+1] as activation function.

The genetic algorithm

I ran three series of experiments, the first with one hidden node for the neurocontroller, the second with two hidden nodes, and the third with three hidden nodes. In all cases, I used the same genetic algorithm:

- *Encoding*: Real values.
- *Fitness*: MSE-based, target 0.025. Failure penalty 0.9. Winning chromosome gets recalculated.
- *Operators*: Elitism, viability checking, duplicate checking. After crossover all children are kept. Incest prevention with three alleles.
- *Parameters*: Population size 100. Replacement policy crowding, with a crowding factor of 2 and replacement of the least fit of the selected individuals.
- *Initialisation*: Weight initialisation in a range of [-10,+10], and a chance of 0.5 that a connection is present.
- *Reinforcement*: Basic run length 500 timesteps, increase factor 100, until the change in MSE drops below 0.02 or the run length exceeds 5000 timesteps. Re-evaluation of an individual after five generations. In case of premature failure, MSE until the failure is used for the rest of the run.

with genetic operators:

- *Randomisation*: This is a biased weight mutation with a chance of 1.0 for each weight, and a range of [-10,+10]. This is the same biased weight mutation that Whitley uses in GENITOR (Whitley 1993). Since I also use a less aggressive biased weight mutation, I named this one "randomisation", since that is in my opinion the effect.
- *Biased weight mutation* with a chance of 0.1 per weight and a range of [-5,+5].
- *Connectivity mutation* with a chance of 0.1 per allele.
- *Node existence mutation* with a chance of 0.75 to remove a node.
- *Nodes crossover* with equal chances for both parents.
- *Half-uniform crossover* with an alikeness factor of 0.5 to the selected parent (this is a standard half-uniform crossover).

The choices for this genetic algorithm were made with the experiences of the pole balancing experiments in mind. The recalculation of the winning chromosome was added because a random setpoint function was used, so a lucky streak could lower the MSE of a chromosome. The randomisation genetic operator was added to battle diversity, and it worked quite well. Because I didn't want to replace the chromosomes generated by randomisation too soon, even if they weren't very fit, I used crowding as replacement policy.

The experiments

The evolution of a feedforward network with only one hidden node was just a preliminary stage, in my idea. I knew it could work, but I didn't think it would be very

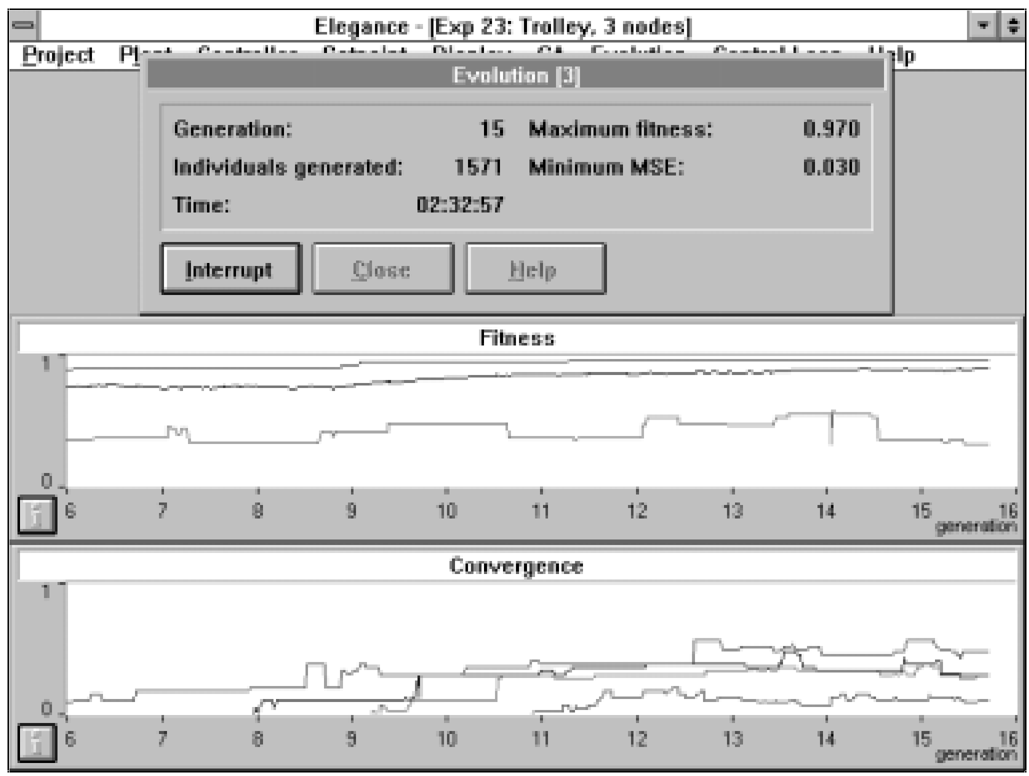


figure 55: A trolley evolution run. A three-hidden-nodes network is being evolved.

good. After I ran three experiments, I decided to continue with the second series of experiments. The results of the three one-hidden-node experiments were that they reached the target in 1262, 1471 and 1844 generated individuals, and that the experiments took 1:26, 2:06 and 2:26 hours respectively. This is a mean run length of about 1525 individuals, and a mean time of about 2 hours.

The second series, using a feedforward network with two hidden nodes, did not do better than the first series. In fact, it did a lot worse. The mean run length was about 2500 individuals, and the mean time needed was about 4:45 hours. However, there were a few experiments which scored quite good. One even ended within the generation of 850 individuals. My first guess was that perhaps a better configuration of the genetic algorithm could speed up the convergence, but I still decided to try the same algorithm on a third series, with three hidden nodes in the neural network.

The third series performed ghastly. I did only nine experiments in this series, but of those nine experiments, three didn't even succeed in reaching the target within 50 generations (which is the equivalent of the generation of 4900 individuals). The mean run length was about 3350 (and this even leaves the failing experiments at 4900 individuals - if they would have been allowed to continue until they would have succeeded, the mean run length would have been even higher), and the mean time needed was 7:40 hours.

In the meantime, it had occurred to me that something very different than the genetic algorithm could be the problem here, so I started to save the best controller which was generated after an evolution run. The results were clarifying. I saved six

neurocontrollers from the experiments wherein I used three hidden nodes. Of these six, only three came from experiments which didn't fail. Those three had all ended up with a neurocontroller configuration *without any hidden nodes*.

This indicates that not only it is possible to use no hidden nodes in a neurocontroller for trolley control, but also that the best configurations (or at least the configurations that can be reached easily) seem to use no hidden nodes at all. If this is indeed true, it explains why the experiments with more hidden nodes didn't work well: since the evolution has to get rid of the hidden nodes, using a configuration with more hidden nodes obstructs the evolution process.

A square wave setpoint generator

I now developed a neurocontroller without any hidden nodes, with the same genetic algorithm. This worked fine. However, when testing the result of this evolution run, I found that it had a slight chance to fail, if two consecutive random setpoints were close to (different) ends of the allowed range.

This exposed a second problem with the genetic algorithm configuration: because the setpoint function was random, a neurocontroller could be generated which had the ability to fail, but which wouldn't during the test run. It might be that some offspring which was very like this controller would not be so lucky, but that wouldn't matter, since the parent still had enough chances to spread its genes in the population.

I replaced the random wave setpoint generator with a square wave setpoint generator, which used a minimum of -0.2, a maximum of +0.2 and a period of 2. I then

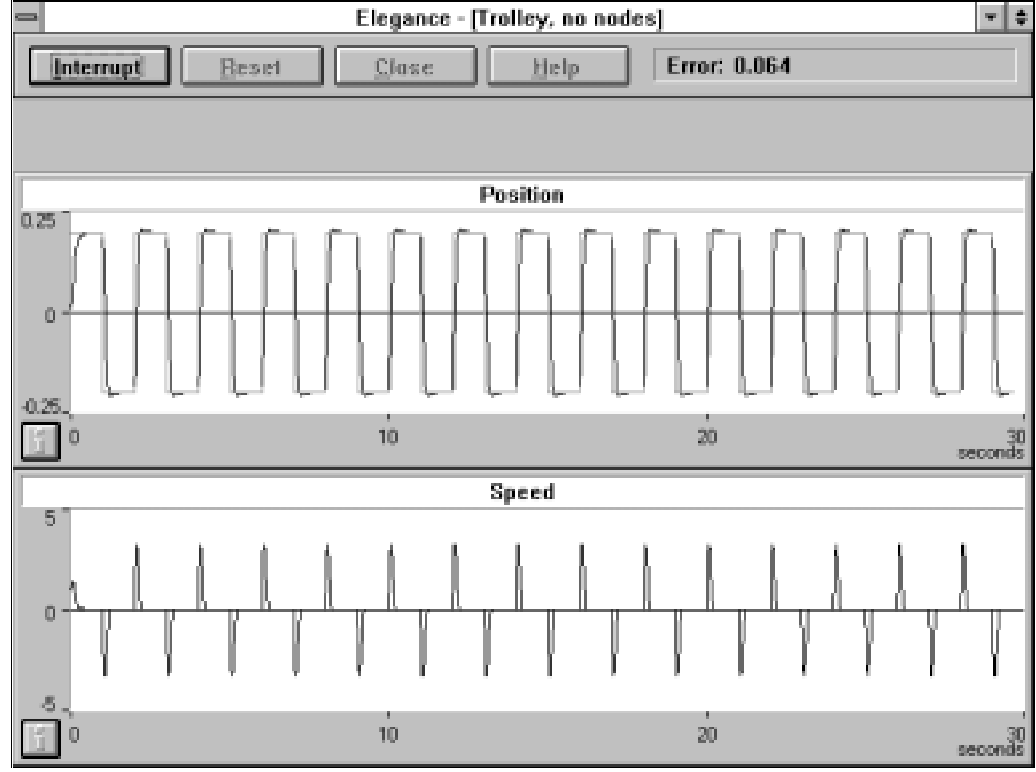


figure 56: A square wave setpoint generator control loop for a trolley neurocontroller.

recalculated the entire population with this new setpoint generator.

Indeed, most controllers in the population failed on this function, which wasn't such a surprise. I let the evolution continue its run, and it finally settled on a controller which would keep the error at 0.62.

When observing the so generated controller in the control loop, it seemed to me that the real error was slightly higher than that. This was, of course, due to the fact that the MSE calculation would stop when the MSE over the run would not change more than 0.02. This can happen quite early in the evolution run.

To check what the real MSE was, I set the MSE change to 0.001 and recalculated again the fitness for the population. Indeed, the MSE now settled on 0.65, which was as I had observed in the control loop.

For comparison, I developed a new neurocontroller, this time with one hidden node, and set the MSE change factor to 0.005. This lead to a neurocontroller which would keep MSE at 0.64, even after recalculation with an MSE change of 0.001.

Both the resulting neurocontrollers are shown in figure 57. The difference between them is too small to be of any significance.

Conclusions

The trolley is perhaps even less suitable a system to test the viability of GAs in neurocontroller evolution than a bang-bang pole balancing system. An ANN with no hidden nodes at all develops into a good neurocontroller, and although it is imaginable that an ANN with more hidden nodes would develop into a better neurocontroller, it seems this would take far too much time, and that the evolution would probably settle for a configuration which has no hidden nodes (or maybe one), which is far more easy to optimise.

Still, there is a lot to learn from these experiments. First of all, the random wave with a fixed interval seems to be not a very reliable setpoint generator. It offers too much opportunity to the luck factor to decide which controller will score well, and it may lead to a controller which has the possibility to fail. The square wave works in this case as a good alternative, but I think that in more complex situations, probably another setpoint generator is needed, for instance one which moves through a number of predefined setpoints. This can be seen as a possible future enhancement of Elegance.

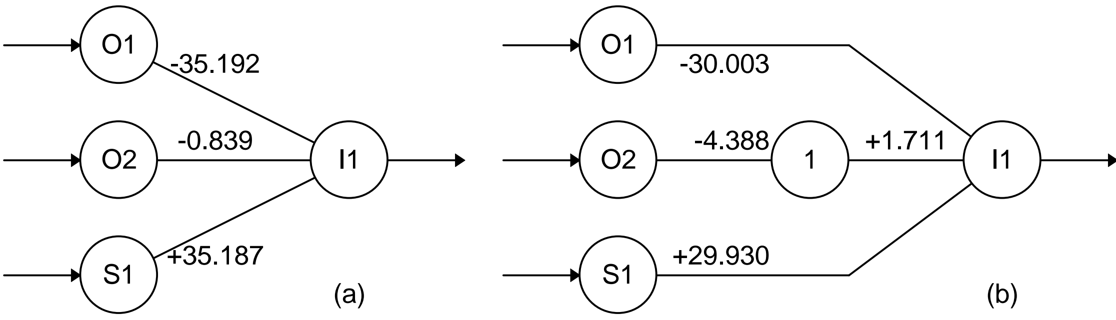


figure 57: Two neurocontrollers developed to control a trolley. *O1* is the trolley position input, *O2* is the trolley speed input, *S1* is the setpoint and *I1* is the plant input. Neurocontroller (a) reaches an MSE of 0.65 on a square wave setpoint function with a minimum of -0.2, a maximum of +0.2 and a period of 2. Neurocontroller (b) reaches an MSE of 0.64 on the same setpoint function.

Second, the fitness determination stops if the change in MSE falls below a certain value. If that value is too low, the fitness determination takes a long time. If it is too high, the value the MSE settles on might not be the value which is the actual value we're looking for. What should we do?

To answer that question, think about the situations wherein we would use a GA. These are often situations in which we are satisfied with an acceptable solution to a problem, and not strive for the optimal solution. Therefore, it isn't of much importance if the fitness determined by the evolution process isn't exactly right. It just serves as an indication of how well the controller performs. That indication may be slightly off.

Of course, if we set the MSE change factor very low, we can be quite sure that the best controller will indeed be at the top of the population. However, I think that the time this takes does not justify the results. The top controller will be quite good, even with a more lenient MSE change factor, and that's all we need.

If we still want to see if we can get to the best controller, we might decide to first run an evolution with a rather high minimum MSE change factor, and, at the end, set this factor to a low value, recalculate the fitness for the entire population, and then continue for a few more runs. We could also, which is in my opinion a better option, switch to a dedicated local optimisation technique. This isn't incorporated in Elegance at this time, but it would be a valuable extension.

Lastly, I'd like to point out one of the powers of the evolution techniques used, which shows itself in this series of experiments. When using backpropagation, the weight values are adapted, but connections are never removed. By using the right genetic operators, the evolution process could remove certain connections, thereby showing that a small neural network could be developed more easily, and still reliably, into a good trolley neurocontroller. In this case, the possibility of architecture design indeed worked out.

6.4 A bioreactor controller

The bioreactor

The bioreactor is tank which is filled with water. The water contains biological cells and nutrient. The cells feed on the nutrient, which makes the cell mass increase. If there is not enough nutrient, the cell mass will decrease. Water containing nutrient can be added to the tank, while water is drained from the tank in an equal rate.

It is the objective of the controller to stabilise the cell mass and amount of nutrient at certain levels. The controller can adjust the flow rate to reach that goal. The bioreactor is a highly chaotic system, and there are few stable combinations of cell mass and amount of nutrient, that is combinations for which the flow rate can be constant.

In theory, with the default values, there are stable setpoints at $(C,N) = (0.1207,0.8801)$ for $w = 0.75$ and at $(C,N) = (0.2107,0.7226)$ for $w = 1.2$, where C is the cell mass, N the amount of nutrient and w the flow rate.

A detailed description of the bioreactor is found in appendix B.8.

The setpoint function

There are basically two ways to evolve a bioreactor controller: Either it is developed with a setpoint function which switches between two stable setpoints, or it is developed with a random setpoint function. If a controller can be developed with a random setpoint function, this would make the controller perhaps more 'general' than when we use fixed setpoints.

Still, I chose to evolve the bioreactor with fixed, stable setpoints. The reasons behind this decision were the following:

- The trolley experiments showed that evolving a controller with random setpoints introduces an inordinate amount of chance in the development. The bioreactor is far more difficult than a trolley, and I think we can't allow that amount of chance in this case.
- Since the bioreactor reacts very chaotic to unstable setpoints, and since most setpoints (especially random setpoints) are unstable, it would be virtually impossible for the controller to stabilise the flow rate on such a random setpoint function. It would therefore be unlikely that the result would be a controller which stabilises the flow rate.
- I tried it for a medium-length evolution run, and the results were not very good.

The use of a setpoint function with just two stable setpoints may result in a controller which can only switch between those two setpoints, and won't be able to switch to perhaps a third stable setpoint. This is not necessarily the case, however. A general solution may be more simple than a solution which can distinguish exactly two predefined setpoints, and therefore a general solution may very well be the result of an evolution run which uses only stable setpoints. I have done no research in the subject matter in the process of writing this thesis, but it might be an interesting follow-up subject.

In short, I used a square wave setpoint function, with the two setpoints mentioned above, and a period of 25 simulated seconds. This period seems to be quite long, but the bioreactor cannot react as fast as, for instance, a trolley. Besides, the project timestep was set to 0.1.

The neural network

What ANN configuration could best be used to evolve into a bioreactor neurocontroller? A lot depends on how complex the problem area is. Since Jacek Jarmulak had already trained an ANN as identifier for a bioreactor, and I decided to use his configuration as guidance. This seemed to be a good idea, since I presume that the identifier aspect of a model-based controller is by far the most complex part of that controller.

Jarmulak said that a layered feedforward neural network, with two layers and twenty nodes in each layer, and as input, besides the current plant state and the setpoints, two TDLs, could develop into a successful identifier. This is what I decided to use also. I also copied Jarmulak's arctangent activation function, with a range of [-1,+1]. In the

first experiment, I set the learning rate to a fixed 1.0.

This neurocontroller is quite complex.. It contains no less than 580 possible connections, each of which can be either present or absent, and each of which has no restrictions on the weight value.

The genetic algorithm

For the first experiment, I used a genetic algorithm with the following parameters:

- *Encoding*: Real values.
- *Fitness*: MSE-based, target 0.01. Ranking in a range [1,100]. Premature failure penalty 0.9.
- *Operators*: Elitism, duplicate checking, viability checking and treatment of competing conventions. After crossover keep all children. Incest prevention with three alleles.
- *Parameters*: Population size 100, crowding with a crowding factor of 2 and replacement of the least fit of the selected individuals.
- *Initialisation*: Weight initialisation in range [-5,+5], and chance that connection is present 0.5.
- *Reinforcement*: Basic run length 500 timesteps, increase factor 100, until change in MSE gets below 0.02 or run length exceeds 5000 timesteps. In case of premature failure usage of MSE until failure for the rest of the run.

with genetic operators:

- *Randomisation*: Biased weight mutation with a chance of 1.0 per allele, and a range of [-10,+10].
- *Connectivity mutation* with a chance of 0.1 per allele,
- *One-point crossover*.
- *Nodes crossover*.

After the 1200th individual had been produced, I added two extra genetic operators:

- *Biased weight mutation* with a chance of 0.1 and a range of [-5,+5].
- *Node existence mutation* with a chance of 0.75 on removal.

The target MSE was set far too low to ever be reached. I had decided to let the evolution run until the convergence got too high or, in my opinion, an acceptable neurocontroller had been evolved.

The evolution

The evolution of the bioreactor neurocontroller was done on a Pentium 90 computer. This is no luxury. The fitness determination of one controller would take between 1.5 and 2 minutes. While this doesn't seem to be a lot, it means that in 24 hours no more

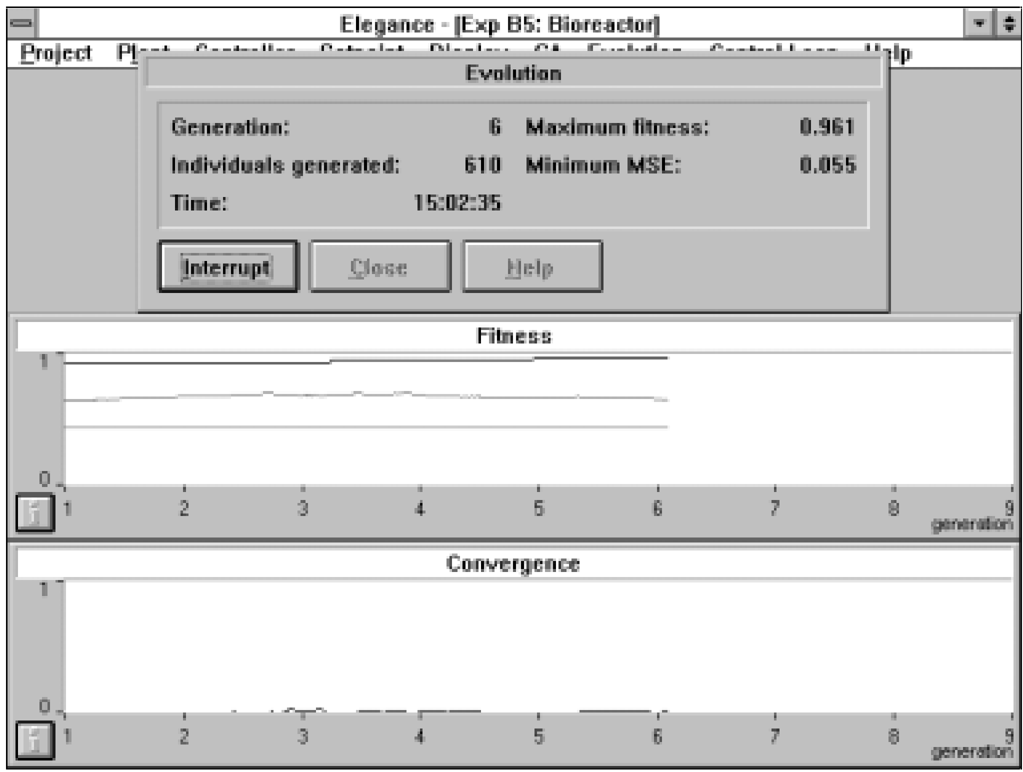


figure 58: A bioreactor neurocontroller evolution run. The ANN configuration used is eight inputs (two states, two setpoints, and two TDLs each consisting op two inputs), two layers of twenty nodes each, and one output node controlling the flow rate.

than 900 controllers can be tested. Since I also needed to use this computer to write my thesis, the first experiment was done over the course of a few days.

The evolution run looked different from the previous experiments I had done. The convergence seemed to rise never more than a pixel or two above zero, and the minimum and maximum fitness seemed to be almost stable from the very first evolved individual on (figure 58).

I think the explanation for the minimum fitness can be found in the fact that the bioreactor in theory can never fail. Therefore, however bad the neurocontroller is, its fitness will always be larger than a certain minimum fitness larger than zero. Experimentally, I found that minimum fitness to be about 0.44.

Furthermore, the fact that the maximum fitness starts quite high and increases very little, might be explained by the fact that it's not too difficult to design a controller which is not too bad. Something like "if amount of nutrient is below nutrient setpoint, set flow rate to 2, otherwise set flow rate to zero" is easy to implement and will have a relatively high fitness. It is to be expected that such a controller will be in the initial population, so the maximum fitness has a reasonable chance to start in the top ten percent of the fitness chart. And if the maximum fitness starts high, it can't rise very much.

I found that after a few hours of evolution, a controller gets developed which results in a control loop as shown in figure 59. This obviously isn't a very good controller. Although it succeeds for the most part in keeping the cell mass and the amount of nutrient somewhere in the neighbourhood of the setpoint values, they fluctuate a lot,

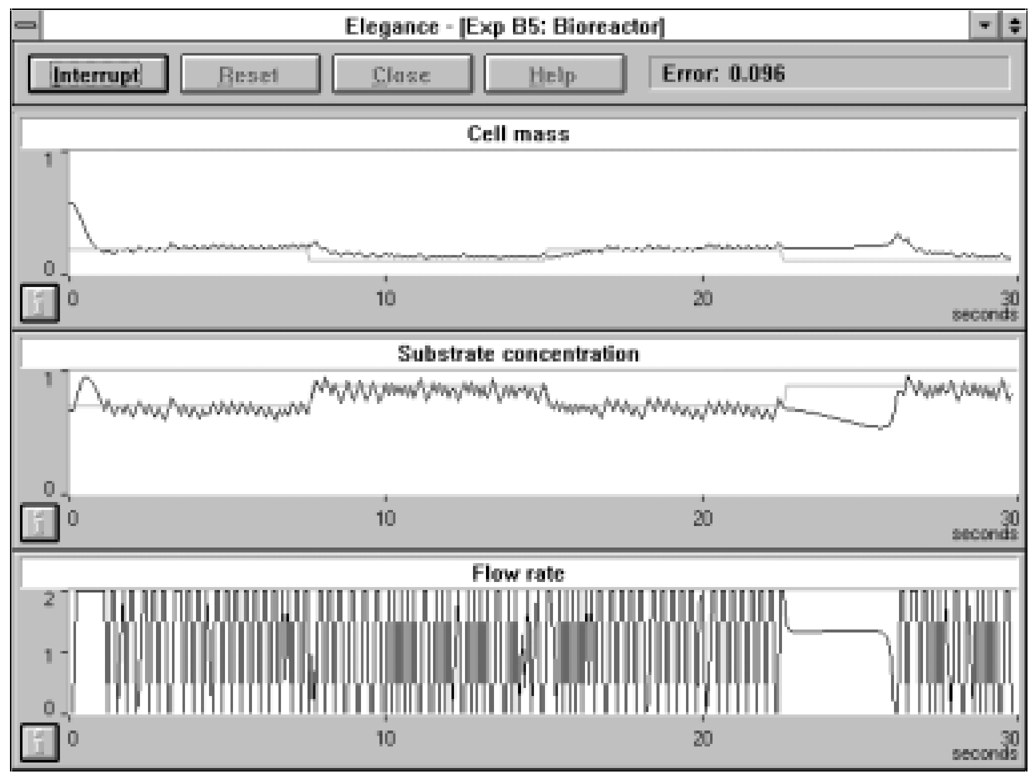


figure 59: A bioreactor neurocontroller control loop somewhere half-way an evolution run. The MSE stabilises around 0.55.

there are sometimes large diversions from the setpoint, and the flow rate oscillates extremely wildly.

I had considerable fears that the evolution process wouldn't end in a good controller, because I saw no obvious way in which a neurocontroller which moves the flow rate between the extremes would get to a neurocontroller which could keep the flow rates constant.

However, it's not for nothing that genetic algorithms are considered to be a subject of artificial intelligence. If there is a path in the solution space which leads from one of the controllers in the population to an acceptable controller via small steps, which mostly increase the fitness and never decrease it too much, in theory the acceptable controller can be found by the genetic algorithm. If I cannot see an obvious path, it doesn't mean that there isn't one.

After about 50 hours of evolution and the generation of twenty generations, the MSE had dropped to 0.25 and I decided to take a look at the result. Much to my delight, that result was a very good bioreactor controller. The control loop for this neurocontroller is shown in figure 60.

Notice that the target setpoints are not quite reached. In fact, the bioreactor stabilises on setpoints $(C,N) = (0.1178,0.8828)$ and $(C,N) = (0.2013,0.7425)$ with flow rates respectively 0.7348 and 1.2083.

I have as yet no solid explanation for this small diversion from the actual setpoints. It may be that the fact that the bioreactor implementation is an approximation of the theoretical bioreactor, leads to these setpoints (perhaps in combination with the fact

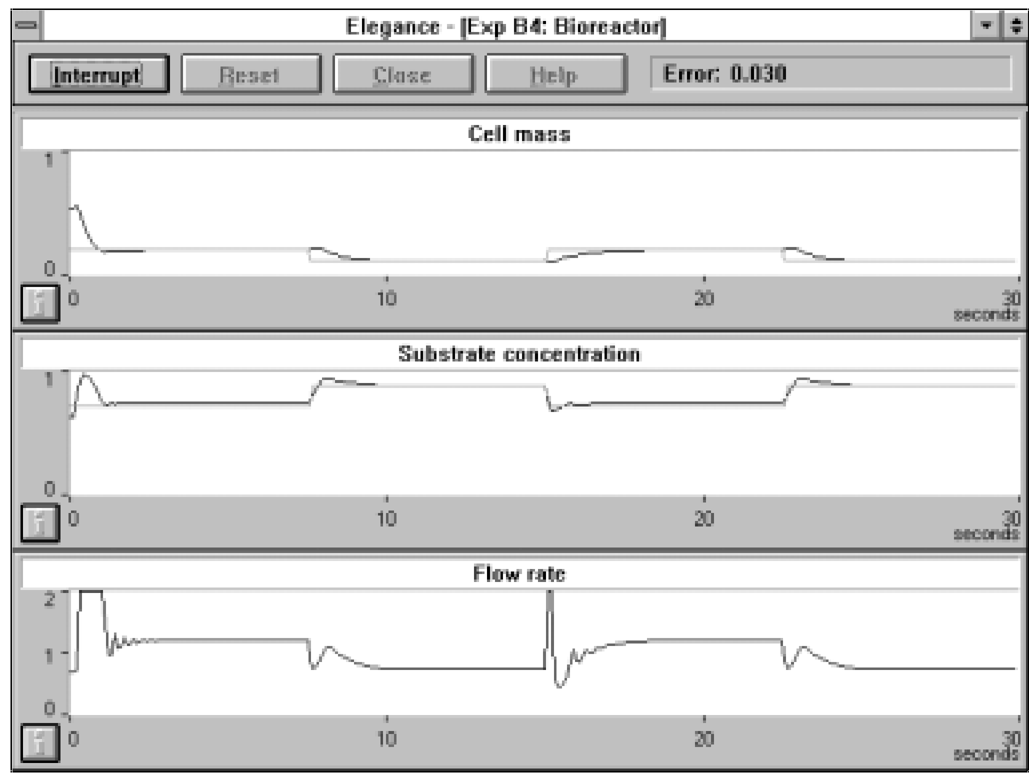


figure 60: A control loop of a bioreactor neurocontroller evolved with Elegance. This controller reaches an MSE of about 0.025 if the period of the setpoint function is 25 (in this figure, the period is 15 so the MSE is slightly higher).

that the timestep size is quite large). It may also be that the target setpoints will eventually be reached, but slowly. Or, the approximation is stable within a small range of the target setpoints, and the genetic algorithm is satisfied with that. Or maybe these are simply different stable setpoints, which are close enough to the target setpoints for the controller to accept them.

Be that as it may, this experiment has shown that a complex neurocontroller as one for a bioreactor can be evolved by Elegance within a relatively short time. This is a very firm indication that the application of GAs for neurocontroller design in a reinforcement situation is a viable approach.

A more complex experiment

After a second attempt to evolve a bioreactor neurocontroller with the same genetic algorithm had shown comparable results, I tried a different approach to the problem. This time, besides a few small changes, I set the MSE change factor (which indicates when a fitness determination run can end) to 0.01 and the maximum run length to 10,000. Also, I added learning rate mutation with a range of [0.5,5.0].

This experiment didn't do very well. Not only increased the time needed to test one controller by a factor of more than 2, also did the experiment seem to get stuck on a minimum MSE of 0.050, which is too high for a really good neurocontroller.

The explanation for the increase in test time is the fact that I asked for a more

precise fitness determination. This was not a smart move, because I had already concluded that a minimum MSE change factor of 0.02 would produce good results, so why would I halve this factor? The reason was, of course, that I hoped to get slightly better results, but as I have already indicated with the trolley experiments, as soon as a reasonably reliable indication of the fitness is found, the fitness determination process can be halted because a reasonable indication is all we need. More precise results are best derived after the GA has explored the solution space, and then preferably with a local optimisation technique.

The explanation for the bad results of the run, even after more than 100 hours of evolution, are, I believe, found in the learning rate mutation operator. I found that the population contained many different learning rates. If a crossover operator is executed on two parents, the genetic algorithm gives the child the learning rate of one of those parents. Since the learning rate determines the effect of the nodes and weights, only the nodes and weights from just one of the parents are copied to the child while keeping the effect they have in that parent. From the other parent, effectively a bunch of random effects is copied. Therefore, the goal we aspire with the crossover operator, that a child gets beneficial characteristics from both parents, will seldom be reached. With the pole balancing system experiments, we have already concluded that the crossover operator is an important aspect of the evolution process. This can very well be the reason that this experiment didn't work out.

The learning rate mutation operator should be implemented differently if it has to work. A possibility is to implement a learning rate for each node in a network, instead of one learning rate for the entire network, combined with the nodes crossover operator, and mutate the learning rates in a biased manner. This does mean that the ANN implementation in Elegance has to change.

A simpler bioreactor controller

Since Jarmulak had also been able to build a bioreactor model with a layered feedforward neural network with two hidden layers with only fifteen nodes in each layer, I attempted to develop a neurocontroller with this configuration with the same GA I used for the first experiment, except that the weight initialisation was within a range of [-10,+10]. The setpoint function I used had a period of 15 simulated seconds.

This experiment succeeded with even better results than I had found in the first experiment. The MSE it ended up with was 0.023, while the first experiment reached an MSE of 0.025, even with a longer period for the setpoint function (with a period of 15 simulated seconds the MSE for the first experiment would have been higher).

It occurred to me that a controller for a plant might be far more simple than a model of that plant. So I now used the same GA on a feedforward network (not layered) with five hidden nodes and no TDLs. As activation function the sigmoid was used, with a learning rate of 1 and a range of [-1,+1].

After no more than 7 hours of evolution and the generation of 2500 individuals, this experiment ended up with a quite good bioreactor controller, with an MSE of 0.024. The controller with this fitness had been generated around individual 2000, and at the end of the run convergence had risen too high to expect better results if the evolution would have continued.

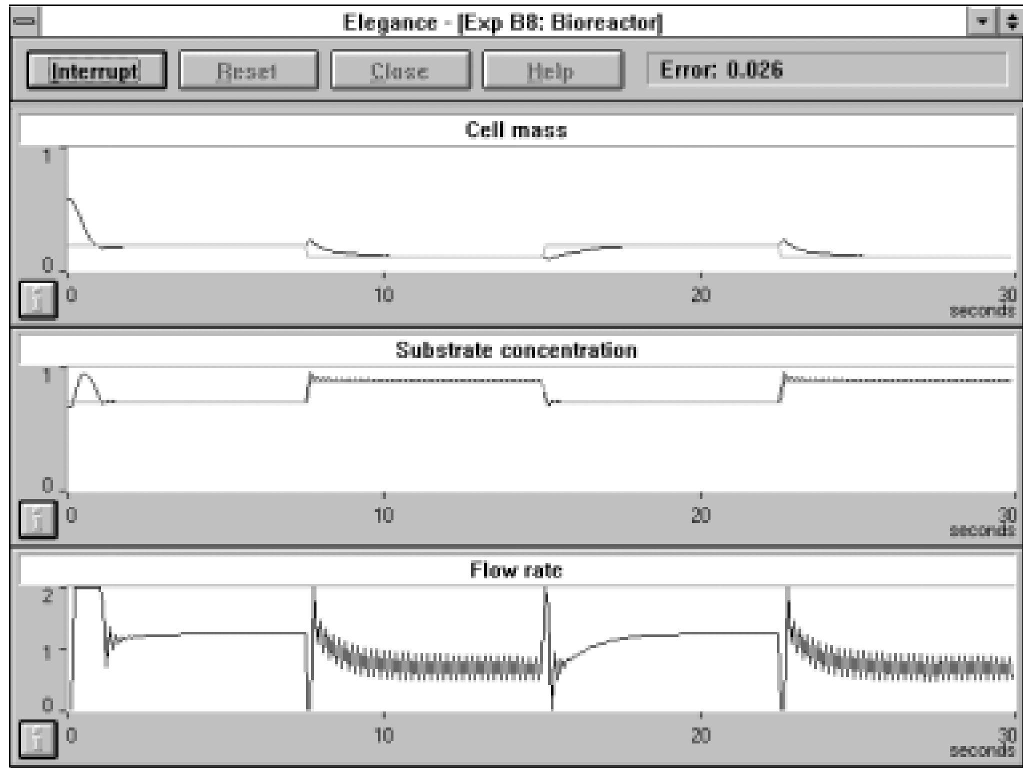


figure 61: The control loop for a bioreactor neurocontroller with five hidden nodes.

The control loop for this neurocontroller is shown in figure 61. Note that this is certainly not a perfect controller. For half of the period, it cannot reach a stable setpoint. If the timestep would not have been set to 0.1, but to 0.5 as Jarmulak did with his experiments, this neurocontroller would probably have scored a lot worse, since there would have been more time for the plant to diverge the actual output from the setpoint (such divergence does not occur if a stable setpoint is reached).

But still, this five-hidden-nodes neurocontroller is not at all bad, and indicates that simpler controllers might be evolved faster and probably with better results than the more complex controllers used in the first experiments. To test this theory, I developed a feedforward neurocontroller with ten hidden nodes and again no TDLs and a sigmoid as activation function, with a learning rate of 1 and a range of [-1,+1]. I used the same GA as I used for the previous experiment, which is as follows:

- *Encoding*: Real values.
- *Fitness*: MSE-based, target 0.005. Ranking in a range [1,100]. Premature failure penalty 0.9.
- *Operators*: Elitism, duplicate checking and viability checking. After crossover keep all children. Incest prevention with three alleles.
- *Parameters*: Population size 100, crowding with a crowding factor of 2 and replacement of the least fit of the selected individuals.
- *Initialisation*: Weight initialisation in range [-10,+10], and chance that connection is present 0.5.

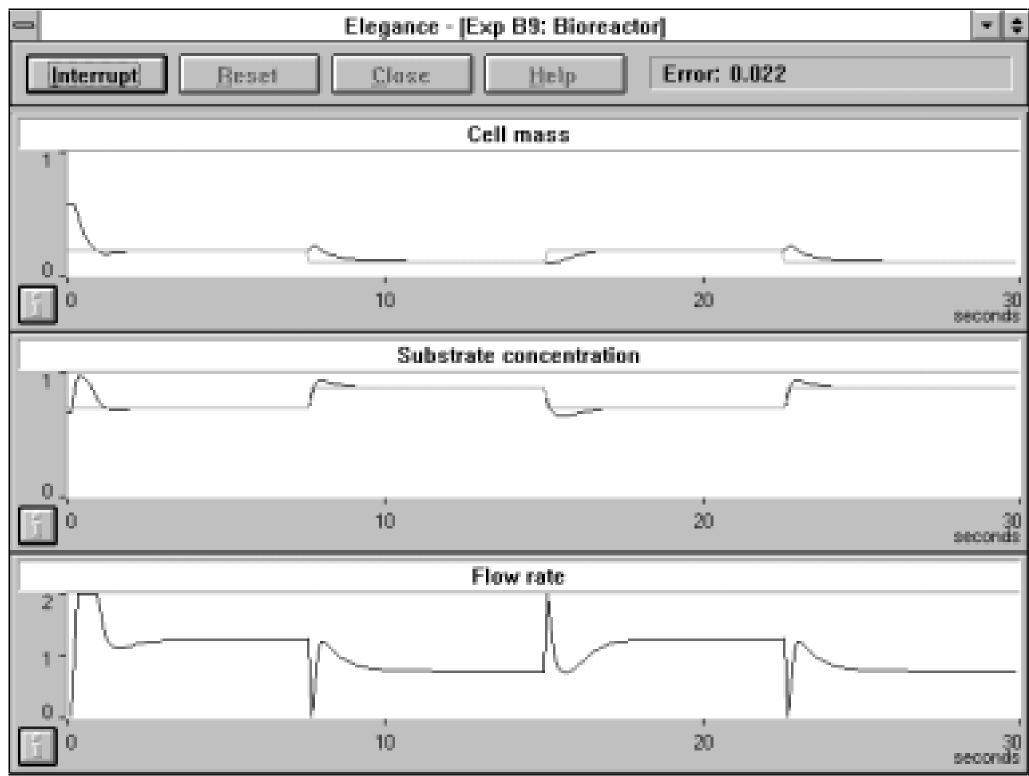


figure 62: The control loop for a bioreactor neurocontroller with seven hidden nodes.

- *Reinforcement*: Basic run length 500 timesteps, increase factor 100, until change in MSE gets below 0.02 or run length exceeds 5000 timesteps. In case of premature failure usage of MSE until failure for the rest of the run.

with genetic operators:

- *Randomisation*: Biased weight mutation with a chance of 1.0 per allele, and a range of [-10,+10].
- *Connectivity mutation* with a chance of 0.1 per allele,
- *One-point crossover*.
- *Nodes crossover*.
- *Biased weight mutation* with a chance of 0.1 and a range of [-5,+5].
- *Node existence mutation* with a chance of 1.0 on removal.

The results of this experiment were very good. After 8 hours of evolution and the generation of about 1500 individuals, the MSE for the best controller found had dropped to 0.019. I let the run continue until 4000 individuals had been generated, which took about 20 hours for the complete run, but the fitness didn't get visibly better (it got slightly better, but only after the fourth decimal). The control loop for this controller is shown in figure 62. The ANN has been reduced by Elegance to seven hidden nodes. Table 11 shows the resulting neurocontroller.

These last two experiments indicate that the controller for a bioreactor can not be as

	1	2	3	4	5	6	7	I1
bias	1.140	-3.462	-5.850	-0.138	-4.434	-4.513		
O1	-7.148	-8.502	-8.727		-4.977	-2.420	-3.876	
O2	-8.818	4.596		-2.971			1.235	6.935
S1		-1.131	2.816					-0.441
S2	-5.157	12.254		-1.462	-7.109	3.961		-7.597
1							8.181	8.453
2				5.578				
3						-9.694	18.643	
4					-6.636	-1.866		
5						8.733		
6								-1.969
7								-8.548

table 11: This is a very good bioreactor neurocontroller with seven hidden nodes. *O1* is the bioreactor cell mass output, *O2* is the bioreactor amout of nutrient output, *S1* is the cell mass setpoint, *S2* is the amount of nutrient setpoint, *I1* is the bioreactor flow rate input, *bias* is the node bias and the numbered nodes are hidden. The vertical axis contains the nodes from which the connections come, and the horizontal axis the nodes to which the connections lead. The cells of the matrix contain the connection weights. An empty cell is an absent connection. A shaded cell indicates a connection which is impossible because of the ANN configuration chosen. This neurocontroller uses a sigmoid as activation function, with a learning rate of 1 and a range of [-1,+1].

simple as the controller for, for example, a trolley, since five hidden nodes is not enough, but that the controller can be much more simple than a bioreactor identifier. Ten hidden nodes seems to work fine for the development.

Conclusions

The most important conclusion from the bioreactor experiments is that GAs are arguably a viable approach in neurocontroller design in a reinforcement situation. The fact that it works in itself is reason enough for optimism. This combined with the fact that a mere 8 hours of evolution on a Pentium 90 is enough to evolve an ANN with ten hidden nodes into a good bioreactor neurocontroller may turn this optimism into enthusiasm.

Jarmulak needed one week to develop an ANN with two hidden layers of twenty nodes into a bioreactor identifier (Jarmulak 1995). Comparing this with the time Elegance needed to develop a neurocontroller with two hidden layers with twenty nodes each, corrected for the speed difference of the machines used, we find that the time needed is about the same for both programs.

However, we also found that the development of a model seems to be a more complex assignment than the development of a controller. For a bioreactor, a simple feedforward ANN with ten hidden nodes evolves in a very good neurocontroller, while an ANN for a bioreactor identifier must be, according to Jarmulak, considerably larger. The model-based approach Jarmulak uses leads to more general results, since a good model can be used to develop several different kinds of controllers. But, as Elegance has shown, if the goal is just to develop a controller, this can probably be done with a simpler ANN configuration.

6.5 Conclusions and further work

Conclusions

For reasons of time, only three experiments have been performed with Elegance while writing this thesis. These experiments were:

- *The development of a bang-bang pole balancing controller*. This experiment was performed specifically to see if an alternative for Whitley's GENITOR could be found. It was concluded that the problem is too simple to draw any firm conclusions about the performance of any technique, because even purely random techniques work in designing a good pole balancing controller. Still, it was found that Elegance could quickly develop such a controller, proving that the program works.
- *The development of a trolley controller*. This problem was also found to be too simple to draw conclusions about genetic reinforcement control, because very simple ANNs, those with no hidden nodes at all, or at most one hidden node, were found to be best in solving the problem. Still, since this conclusion was reached by Elegance itself, which reduced more complex networks to simpler ones, it showed that the architecture design capabilities of Elegance are a potentially powerful feature.
- *The development of a bioreactor controller*. This is a complex problem which Elegance could solve in a reasonable amount of time, proving that genetic reinforcement control is a viable technique. Elegance also showed that an ANN configuration can be used with no more than ten hidden nodes, while, according to Jarmulak, for a model of the bioreactor a more complex ANN is needed.

Far too few experiments were performed to draw firm conclusions, but a few preliminary results, which indicate the possibilities for research with Elegance, are the following:

- Keeping diversity in the population is important. This can be accomplished by using a kind of biased randomisation operator. If such an operator is used, care should be taken that the resulting chromosomes are not removed from the population before they get a chance to procreate.
- Both crossover operators and mutation operators are important for genetic reinforcement control.
- The effect of adaptive mutation does not seem to have much influence.
- Random setpoint functions are not very suitable for fitness determination, because they give the chance factor too much influence in the evolution process, which will result in a non-reliable population ordering.
- With MSE-based fitness, fitness determination is halted when the change in MSE gets below a certain value. This value should not be set too low, because this increases the needed length of time enormously while contributing little to the process, which only hinges on a fitness indication and not on exact results.
- Learning rate mutation, as implemented by Elegance, does not work well, probably because it distorts the effects of the crossover operator.

- ANN configurations for a controller can probably be simpler than ANN configurations for a plant model.
- And, perhaps most important: genetic reinforcement control is a rival for conventional neurocontroller design techniques.

Further work

I can think of many areas of research which can be immediately entered with Elegance. To give some ideas, here is a list of some of the questions Elegance might help to answer:

- Does GENITOR work as well on more complex plants as it does on the pole balancing system?
- What combination of genetic operators works best in genetic reinforcement control (both on simple plants as on more complex plants)?
- Can Elegance develop recurrent neural networks as neurocontrollers, and how does such a recurrent neurocontroller compare to a feedforward neurocontroller?
- Is the effect of Thierens' competing conventions treatment useful?
- Is a deterministic setpoint function capable of developing a general neurocontroller, for instance for a bioreactor?
- Can some general rules be determined for the settings of the test run length for MSE-based fitness?

Besides questions which Elegance can answer without making any changes to the program, some of the extensions which could generate interesting results are the following:

- Elegance could use a square setpoint function which contains more than two setpoints, or it could use a random setpoint function which would be the same for every test run. This could be used to examine the possibility of generating a general neurocontroller with a fixed setpoint function.
- Elegance could certainly be used to develop neurocontrollers for more complex plants than are currently implemented. For instance, a bioreactor with a time-delay factor where the flow rate change is concerned would be, I think, a nice challenge.
- Elegance could use a plant which needs time-until-failure fitness which is more complex than the pole balancing system, for instance a two-pole balancing system in which a second pole is placed on top of the first.
- Elegance could use a more flexible ranking technique, since ranking seems to be quite important but at the moment is only implemented in Elegance as a linear function. This could be used to examine the, possibly very beneficial effect of more complex ranking functions (Chabot Stadhouders 1994).
- Elegance could be extended with a few other genetic operators. I have encountered many more genetic operators in my exploration of GA literature, and most of them haven't been implemented for reasons of complexity. Some of them could be very useful, though.
- Learning rate mutation as it is implemented at this time in Elegance does not seem to

work well. A different implementation could produce interesting results.

- Elegance could be extended with competing conventions treatments which are more complex but possibly more useful than Thierens' technique. This is especially important if Thierens' technique proves not to be too beneficial to the evolution process.
- Elegance could be extended with totally different controller types, like LISP programs or mathematical functions. Although this is a difficult extension to program, it could open up a totally new area of research.

6.6 Summary

This chapter described some of the preliminary experiments done with Elegance. The experiments used a bang-bang pole balancing system, a trolley, and a bioreactor as plants.

Both the bang-bang pole balancing system and the trolley are plants which are too simple to be of real significance in the research for the validity of the genetic reinforcement control. However, the experiments with these two plants showed that Elegance works. They also indicated a few practical rules which might be of value in the design of suitable GAs, like the fact that random setpoint functions are probably not very useful in the evolution run.

The bioreactor is a very complex plant, and Elegance managed to evolve a good neurocontroller for this plant. This result is significant, because it shows that GRC might very well be a sound technique for neurocontroller evolution. The bioreactor experiment also showed that a neurocontroller for a bioreactor can be much simpler than an identifier for a bioreactor.

These first experiments have shown that further research in GRC is certainly warranted.

Conclusion of Part III

Overview

The subject of the third part of this text is the program Elegance. This program has been developed to experiment with the application of GAs in the evolution of neurocontrollers in a reinforcement situation.

To achieve that goal, Elegance has a very flexible design. Not only does it contain a wide range of GA parameters which can be set by the user, and large number of genetic operators, but it also contains various different, configurable plants. Because of the object oriented approach of the program, it is very simple to add new plants, and thus new experiment domains, provided one has access to the source code.

Preliminary experiments with Elegance show that the program works, that it can be used to obtain knowledge about genetic reinforcement control (GRC), and, most important, that GRC seems to be a viable approach to the problem of developing neurocontrollers in a reinforcement environment.

Personal views

The process I went through during the research for this thesis were of a kind often encountered when one works on a project in a hitherto unknown domain. First, I became very enthusiastic about the possibilities of GAs. Then, when I began really working on the problem, serious doubts started to rise and a certain amount of pessimism entered my mind. As the joke goes, this should be followed by a third phase of PANIC (followed by a search for the guilty parties), but, a bit to my surprise, this was not the case. Elegance for a large part restored my initial trust in GAs, even in such a complex problem domain as reinforcement control.

This does not mean that I do not still have doubts about GAs. I think they can easily be misused, and that they are far too slow to be used in problem domains which can also be attacked with conventional techniques. But there are many problem domains which are too complex, or too chaotic, for conventional techniques, and this is where

GAs may prove their worth.

I think neurocontroller design is such a problem domain. True, there are conventional techniques that can be used with these problems. However, these techniques place limitations on the solutions (like the fact that backpropagation, for instance, only works with feedforward networks), are often not a lot faster than GAs, do not always work (backpropagation may very well converge to a local optimum), and are almost always less powerful than GAs (backpropagation cannot design an architecture).

Elegance has not only shown that it works, and that GRC is a viable technique, but it also already brought up some interesting preliminary research results, like, for instance, the following:

- Both mutation and crossover operators are important for genetic reinforcement control.
- Random setpoint functions are not very suitable for fitness determination.
- For MSE-based fitness, the fitness determination process should be interrupted quite early, because a rough fitness indication is enough for the process to work. A long fitness determination only increases the time the process takes without any relevant added value.
- A neurocontroller for a plant may very well be far more simple than a neural identifier for that plant.

It seems to me that Elegance has already proved its worth in the research of GRC. Working with the program not only showed me the validity of some of my ideas, but also stimulated me to form new ones, and offered me ways to test a large number of these new ideas. And that’s exactly the purpose of an experimenting environment.

Is evolution elegant?

It is common business in the GA-world to give programs a nice acronym for a name. I decided to stick to this practice, and I wanted to use the letters GA and NC (for neural controller) in the acronym for my program. “Elegance” was the only word that I could think of, and the rest of the acronym was easily filled in.

I had some initial doubts about this name. At first glance, GAs are far from being ‘elegant’. In fact, they look more like a brute-force method, and I think that’s exactly what they are.

But consider the problems for which GAs are used. These are problems for which people could not find a good method of solution. For instance, for the travelling salesman problem there is no simple solution known, and since it is NP-complete, it is very unlikely that such a simple, ‘elegant’ solution will ever be found. Neurocontroller design, especially for larger neurocontrollers, is also such a problem domain.

Neurocontroller design for reinforcement control can at the moment, and maybe always, only be approached with brute-force methods. And as brute-force methods go, GAs are a very elegant one. Close-up, GAs just make random jumps through the solution space. From a distance, however, we see that GAs carve a path through the solution space, which (if the design is good and the solution space is not GA-hard) will

invariably end up at an acceptable solution. Not only that; the path to this acceptable solution can be, and is almost always, surprisingly short.

In this respect GAs are in my opinion elegant, and I think the program Elegance is therefore aptly named.

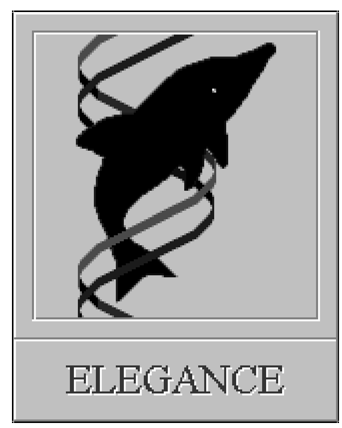
Summary

Genetic algorithms have in recent years become of major interest in the artificial intelligence community. Although they can be considerably misused, they have also proved to be a potential solution technique for problems for which no dedicated technique has been, as yet, developed.

Controller design, or, more particularly, neurocontroller design, in a situation where reinforcement control must be applied, is such a problem area. Genetic algorithms are a possible approach to this field, but the question wether the technique is useful for complex design situations is not settled at this time. The research which must determine this usefulness should start with a huge number of experiments. To perform a large selection of those experiments, the software environment Elegance has been created.

Elegance is very flexible, and is capable of comparing different genetic algorithm configurations in solving the same neurocontroller design problem. After an extensive search through literature, I have found no other environment which offers these possibilities. The first experiments done with Elegance show that genetic reinforcement control is indeed a viable technique, and a rival for conventional techniques used in this problem area. Therefore, more research in genetic reinforcement control is warranted. Elegance can help with that research.

Appendices



A Elegance user manual

In a Windows environment, it is often the case that even very complex programs are delivered with only a small booklet with instructions. The program itself contains all the information needed to use it, in the form of a context-sensitive help system. This means that at any point in the program, the user can press a Help button on the screen, or press the help key F1, and he gets a page with information on the current screen he is viewing, with hyperlinks to related subjects.

If such a help-system is designed well, the user needs nothing more, except for maybe a small introduction to the program, which guides him with his first, simple project. The benefits of this are mainly that the user always has the program manual on-line and will need no bulky books next to his computer, and that he always starts his search for help with information which is about the current state of the program.

Elegance also has an extensive help system with about 130 pages of text. The design of this system is very straightforward. For almost every screen in the program, there is one help page, which describes the entire screen, short and to-the-point. This screen is accessed through the Help button, which is found on the screen. Pressing F1 also takes the user to this help page, unless there's some more detailed information on the object which currently has the focus. The user is then taken there, but will have the option, through a hyperlink, to get to the information on the complete screen. The whole theory behind the program is also incorporated in the help file, but will only be accessed through hyperlinks or through the search function.

A small text is written as user manual to guide the user through his first steps with Elegance. That text starts on the next page.

Elegance user manual

Version 1.0, august 1996.

Introduction

Elegance, which stands for “Engineering Laboratory for Experiments with Genetic Algorithms for Neural Controller Evolution”, is a program designed to experiment with the use of genetic algorithms to design controllers, especially neurocontrollers, which need to work in a reinforcement situation.

Genetic algorithms (GAs) are search algorithms based on the principles of natural selection and natural genetics. A working knowledge of GAs is necessary to be able to work with Elegance. Neither this manual nor the on-line help system will tell the user much about GA theory. Those who wish to read about it, should get one of the many books about it. I heartily recommend Goldberg’s *Genetic Algorithms in Search, Optimization & Machine Learning* (Goldberg 1989).

Neurocontrollers are neural networks which are used to generate the input for a plant, so that the plant will produce a certain output. A plant is a process, which maps input to output and which may have internal parameters. More information about plants is found in the on-line help system. Neural network theory, however, is considered to be something the user is already familiar with. As a reference to neural networks, I recommend Alexander & Morton’s *An Introduction to Neural Computing* (Aleksander 1990).

Reinforcement control is control that does not make use of a model of the plant to be controlled. Instead, an evaluation of the success of the controller in a practical situation is used to adapt the controller to become more successful.

In industry, processes that cannot be modelled well are quite common, and reinforcement control is often the only acceptable option. However, since those companies which use reinforcement control most of the time consider the techniques they use industrial secrets, there is very little literature about the design of controllers in a reinforcement situation. Elegance offers a way to experiment with the possibility of applying GAs for this purpose.

Requirements

Elegance is developed for Microsoft Windows 3.1. One needs a PC which runs Windows 3.1 or a compatible environment, like Windows 95, Windows NT or WIN-OS/2. At least 4Mb of internal memory is recommended, more if the population gets large and/or the neurocontroller has many hidden nodes. Speed is very important, although with a well-designed GA the simpler controllers can be evolved within an hour on a 486DX/33 PC. For the more complex experiments, at least a Pentium is recommended. A mathematical coprocessor is an absolute must, because most of the calculations use floating point numbers. A coprocessor is found in all 486DX and Pentium computers. 486SX computers mostly lack a coprocessor, however.

Installation

The user needs just two files to work with the program: the executable ELEGANCE.EXE and the help file ELEGANCE.HLP. It is recommended that these two files are placed in a directory of their own. After that, the program can be started from the Windows desktop, by choosing *Run* from the *File* menu, and then selecting the ELEGANCE.EXE file. If the user wishes to install the program as an icon, he can use the option *New* from the *File* menu. This manual will not detail on that; please consult the Windows manuals if you need any help.

If you don't have a TEMP environment variable set to a directory for temporary files, it is recommended that you do that, since Elegance creates some temporary files which won't be removed if the program or Windows ends abruptly. It is normal in a Windows environment that such a TEMP environment variable is set.

On-line help

The on-line help system is considered to fulfil the function of a user manual, and is designed to be used that way, except for a small introduction to the program to help the user with his first experiment. This introduction is found in the next paragraph.

To use the on-line help system, the Windows configuration must give access to the standard Windows help utility. The default Windows installation will work OK.

The on-line help system can be accessed in several different ways:

- By pressing the *Help* button at the bottom of most screens.
- By pressing the *F1* key anywhere in the program.
- By using the *Contents* or *Search for Help On* menu choices in the *Help* menu.

If you need instructions on how to use the help system, please consult the standard Windows on-line help system manual, which can be accessed, if available, through the *How to Use Help* menu choice from the *Help* menu.

A simple experiment

Start Elegance. An empty screen is shown with a menu bar at the top. The easiest way to work is to go through the menu bar from left to right. Incidentally, if you wish to leave the program, choose *Exit* from the *Project* menu.

The first choice on the menu bar is *Project*. A project is considered to be the container of all the objects needed to run an Elegance experiment. Open the *Project* menu and choose *New Project*. A window is shown in which you can specify some characteristics of the project. Give the project a name in the first edit box, and press *OK*. You now return to the main screen. If you have made a mistake and wish to access the project attributes again, choose *Edit Project* from the *Project* menu.

As you can see, the *Plant* menu has become available. The plant is the process that is to be controlled by the controller we wish to develop. Open the *Plant* menu, and select *New Plant*.

You are now presented with a selection of several plant simulation types. One of

those is the *Trolley*. A trolley is a small cart on a rail, which can be moved by the controller by sending signals to the trolley's electromotor (more information about the trolley and all the other plant types can be found in the on-line help system). Select the *Trolley* by moving the reverse bar to it and pressing *OK* (as is usual with Windows programs, double-clicking also works).

The next screen shows some of the trolley parameters, which you can use to configure your particular brand of trolley. For now, just press *OK* to accept the defaults. These parameters can later be accessed through the *Edit Plant* menu choice.

Now, the next menu choice, *Controller*, has become available. Choose *New Controller* from this menu. You may now select a controller type from a selection menu. Choose the *Feedforward neural network*, and press *OK*.

Now a screen follows, which you can use to edit some of the current controller's parameters, and the setpoints you wish to use. For the parameters, specify a minimum weight of -100 and a maximum weight of +100. Notice that the output parameter *Position* is checked. This means that the controller will direct the trolley input to get the trolley position to a certain value. The factor, next to the setpoints, is the weight of that setpoint (that is, how much the controller should take this setpoint into consideration in relation to other setpoints). Since only one setpoint is checked, the factors are not important. Press *OK*. This screen can later be accessed from the *Edit Controller* menu choice.

Since you have selected a neurocontroller, now a neural parameters screen follows. Here you specify the characteristics of the neural network. Set the number of hidden nodes to 1, since one hidden node is all we need to get an acceptable trolley controller. Leave the rest of the parameters on the default values. There are two more neural parameter screens, which can be accessed through the *Architecture* and *Weights* buttons on this screen. The data on these screens will be designed by the GA, however, so just ignore them for now. Press *OK*. The neural parameters screen can be later accessed through the *Neural Parameters* menu choice from the *Controller* menu.

The *Setpoint* menu can now be accessed. This menu is used to maintain the setpoint generator. The setpoint generator generates signals which are used by the controller as the target values for the plant output. That is, if the setpoint generator tells the controller that the trolley should be moved to position +0.1, the controller will aim to do just that.

Choose *New Setpoint*. You may now choose for which plant output you will specify a setpoint. You can access this screen also from the *Edit Setpoint* menu. Since we have selected only the trolley position as setpoint output with the controller, we only need to specify a signal for the trolley position output. Select *Position*.

On the screen that follows, you can define which signal should be used. The signal type is presented as a combo box. Press on the small down arrow on the right side of this box, and choose the signal type *Random wave (fixed interval)*. You now get to see the parameters for this signal you need to specify. Since the rail length is half a meter, in the domain [-0.25,0.25], a minimum of -0.2 and a maximum of +0.2 would be in order. Enter those values, and specify 0.05 as the minimum change. Press *OK*. You now return to the plant output selection screen. Press *Close*.

Both the *Display* and the *GA* menu have now become available. We'll first do the display. This concerns the display that is shown during a control loop run, which follows later in the program.

Choose *New Display*. You are now presented with a screen on which the charts of the display are entered. You can also enter this screen by selecting *Edit Display*. At the moment, there are no charts. Press *Add* to add a chart. You get to the chart definition screen. The general chart parameters are entered on this screen, and at the bottom, the lines on the chart.

The chart we're going to specify will show the current trolley position and the position setpoint. We'll therefore name the chart "Position". Enter this in the top edit box. Since the values that the position can take will always be between -0.25 and +0.25 (otherwise the cart runs off the rail), enter these values in the minimum Y-value and maximum Y-value boxes.

Press *Add* to add the first line. A new window is opened, in which the line can be specified. Select "Output 0: Position" from the line type combo box and enter "Cart position" as description. Press *OK*. You return to the chart definition window, and the line description is now shown in the lines list box. Press *Add* again. Now select "Setpoint 0: Position" and enter "Position setpoint" as description. Press *OK*.

Press *OK* on the chart definition screen. You now return to the display screen. This one chart is enough for now, so press *Close*.

The last, but most complex object we need to define, is the genetic algorithm. Choose *New GA*. You get to a screen on which the GA is defined. This screen can be accessed later by choosing *Edit GA*.

The GA screen contains six pages, which you can access by choosing the tabs *Encoding*, *Fitness*, *Operators*, *Parameters*, *Initialisation* and *Reinforcement*. We'll work through them one by one.

On the *Encoding* page, the parameters are specified which concern the coding of the controller as a chromosome. In the first radio group box, choose *Real values*. Immediately the rest of the screen is dimmed. The rest of the parameters are not needed for real-valued chromosomes.

On the *Fitness* page, the parameters are specified which concern the fitness calculation. Since we wish to direct the trolley to certain positions, the difference between the setpoint and the current trolley position should be used as the basic fitness measure. You select this by choosing *Mean square error* (MSE) as *Basic fitness* (this is also the default). Select *Ranking* as *Fitness conversion* technique.

On the *Operators* page, the attributes which concern the genetic operators are set. Leave everything on the default values. We just select some genetic operators. Press the *Add* button, and select *Biased weight mutation*. Press *OK* to choose the defaults from the genetic operator screen. Also *Add* the *Connectivity mutation* operator and the *One-point crossover* operator.

On the *Parameters* page, set the population size to 50 and the maximum number of generations to 10.

On the *Initialisation* page, set the weight initialisation minimum to -25 and the maximum to +25.

On the *Reinforcement* page, leave everything on the default values.

Press *OK* to leave the *GA* screen.

Now all the objects of the project have been created. Before running the evolution, it is wise to save everything. Select *Save Project* from the *Project* menu. Because the project is considered to be the container of all the other objects, this saves everything. Every separate object is saved, and if it not already has a filename, you are asked to

supply one. Just select the default by pressing *OK*, unless you want to save it in another directory. Probably you have to save six different objects now.

The evolution consists of two parts. First, the population should be initialised. This is done by selecting *Initialisation* from the *Evolution* menu. This may take quite a while, depending on the speed of your computer (between a few seconds on a Pentium to a few minutes on a slow 486). Then, the evolution run is started by selecting *Run* from the *Evolution* menu.

While the evolution runs, you see some charts displayed on the screen. The upper chart shows the fitness development. The upper line is the maximum fitness, which is the one to watch. Don't get your hopes up when it gets 'quite high'. It should be *very* high to indicate success. The lower chart shows the population convergence. The fourth, bottom line (which may not be in view, if it is very low) shows the convergence for the whole population.

The text screen provides the most information. It gets updated every time a new controller is generated. This means that the fact that the timer remains frozen is not an indication that the program 'hangs'. You may notice a small red circle flashing in the bottom of the window. This indicates that the program is still running. During the fitness calculation of a generated controller, this is the only thing that moves.

It now depends on the success of the GA what happens. The evolution run may end with a 'winning' controller, or it ends when ten generations have been produced. It probably will be the last. This may take between a few minutes on a Pentium to about

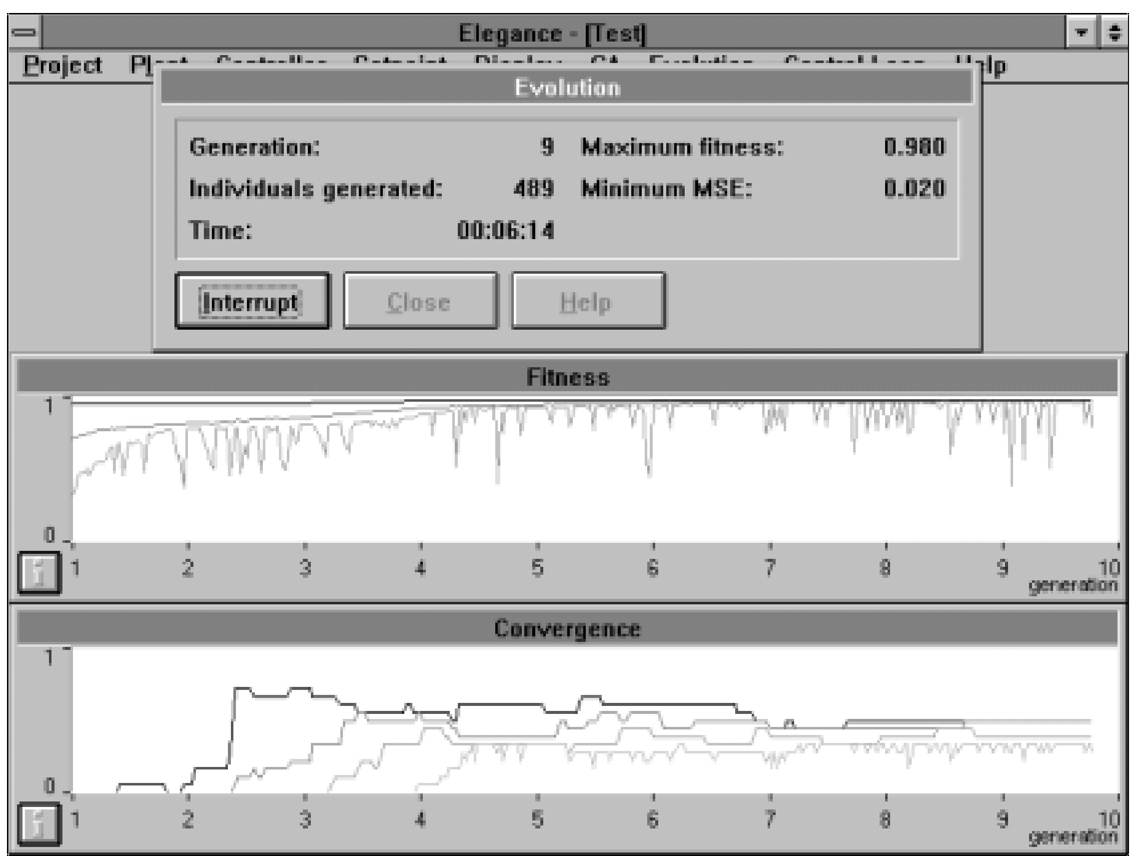


figure 63: An example evolution run.

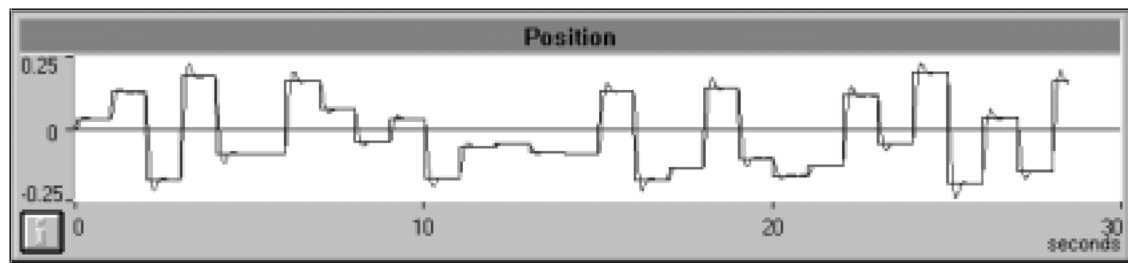


figure 64: An example of a control loop display.

an hour on a 486. You may also choose to interrupt the evolution by pressing the *Interrupt* button.

No matter how it ends, after the evolution has finished you can close the evolution text screen by pressing the *Close* button, and then activate the best controller found by choosing *Activate best controller* from the *Evolution* menu.

You are now ready to evaluate the performance of this controller in the *Control loop* menu. Choose *Run*. On the screen a small bar is shown at the top with some buttons, and at the bottom the chart you defined under the *Display* menu is shown. Press the *Go* button to start the run. The run will continue until you press the *Interrupt* button or the plant fails.

On the chart, you see two lines being drawn: the square wave is the setpoint, and the other line is the cart position. If the controller you generated is good, the actual cart position line will follow the setpoint line closely. If not, the line may do something completely different, or it may remain zero, or it may even happen that the cart runs off the rail. If the last thing happens, you can reset the simulation by pressing the *Reset* button, and start over by pressing the *Go* button.

If the controller is not very good, you may study the GA parameters and the help file, to design a more suitable configuration. You can also try again. The current configuration at least has the potential of generating a good controller, but you may have had bad luck.

This is all you need to know to start working with Elegance.

Suggestions

- Remember that the *F1* button always takes you to a context-sensitive help screen.
- The easiest plant to experiment with is, in my opinion, the *bang-bang pole balancing system*. This plant needs *time-until-failure* as basic fitness calculation.
- Large neural networks and large populations need long evolution times. Start with smaller networks (1-10 nodes) and smaller populations (50-100 individuals) to see if they work.
- If you do not have at least a Pentium computer, forget about the *bioreactor* plant.
- To get some ideas for well-working configurations, read about GENITOR and ANNA ELEONORA in the help file.
- When a population is initialised, several edit choices have become unavailable, like the number of nodes in the neural network and the encoding parameters. If you wish

to make changes to those parameters, you should first deinitialise the population.

Troubleshooting

Windows is well-known for its instability, so we can't expect Elegance to run always without problems (although I personally haven't experienced any problems with this version). Luckily, Elegance contains an autosave option, which saves, during an evolution run, the current state of the population every generation. This option is turned on per default. So, after a (hopefully very rare) crash, you can simply continue the evolution run.

For very unstable or time-consuming situations, the user might even indicate that he wants to save the population every time a new individual is generated. This can be done from the *Preferences* menu.

New versions

The latest version of Elegance can be obtained through the TU Delft, (probably) by anonymous ftp from *ftp.twi.tudelft.nl*, somewhere in the directory */TWI/TI*. The filename will (probably) be *elegxxx.yyy*, where *xxx* is a version number, and *yyy* is a compression extension, like *zip* or *arj*.

Commentary

If you have any comments on Elegance, you can contact me by e-mail. My e-mail address is, at the moment of this writing, *P.Spronck@inter.NL.net*. If that address is not available anymore, try to get the latest version of Elegance from the TU Delft, which will contain an e-mail address you can use to contact me or someone else who maintains the program.

Distribution policy

Elegance is freely available for non-commercial use. Copyrights remain with Pieter Spronck. For commercial use, please inquire at the TU Delft (contact professor E.J.H. Kerckhoffs, *eugene@kgs.twi.tudelft.nl*).

References

ALEKSANDER, IGOR & MORTON, HELEN (1990). *An Introduction to Neural Computing*. Chapman & Hall, London.

GOLDBERG, DAVID E. (1989). *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley Publishing Company, Inc.

SPRONCK, PIETER (1996). *Elegance: Genetic Algorithms in Neural Reinforcement Control.* Graduation thesis, TU Delft, Delft.

B Plants

This appendix specifies the characteristics of the plant types which are found in Elegance. This information is also found in the Elegance on-line help system. Most of these plants are copied from Jarmulak's NCWB (Jarmulak 1994b).

B.1 SISO system

The SISO system is the most simple plant found in Elegance, and it is mainly meant for testing purposes. It is copied from Jarmulak's NCWB (Jarmulak 1994b), who in his turn has it from Krijgsman.

SISO stands for Single Input Single Output. The plant is non-linear and discrete. Dead-beat control of the SISO system is possible. The timestep which is set with an Elegance object should be 1.0 for this plant.

The SISO system is controlled by the formula:

(20) $$y(k+1) = \frac{y(k)}{1 + y(k)^2} + x(k)$$

where $y(k)$ is the output of the plant at timestep k, and $x(k)$ is the input of the plant at timestep k. $x(k)$ falls in the domain [-2,+2]. This limits $y(k)$ to something like [-2.5,+2.5].

B.2 DIDO system

The DIDO system is one of the most simple plants offered by Elegance, and it is mainly meant for testing purposes. It is designed by myself.

DIDO stands for Double Input Double Output. The plant is non-linear and discrete.

Dead-beat control of the DIDO system is possible. The timestep which is set with an Elegance object should be 1.0 for this plant.

The DIDO system is controlled by the formula:

$$(21)\qquad \begin{cases} y_1(k+1) = \dfrac{x_1(k)\cdot y_1(k)}{1+y_2(k)^2} + x_2(k) \\ \\ y_2(k+1) = \dfrac{x_2(k)\cdot y_2(k)}{1+y_1(k)^2} + x_1(k) \end{cases}$$

where $y_1(k)$ and $y_2(k)$ are the outputs of the plant at timestep k, and $x_1(k)$ and $x_2(k)$ are the inputs of the plant at timestep k. $x_1(k)$ and $x_2(k)$ fall in the domain [-2,+2].

B.3 Titration

The titration system is one of the most simple plants in Elegance, and it is mainly meant for testing purposes. It is copied from Jarmulak's NCWB (Jarmulak 1994b), who in his turn has it from Krijgsman.

It is a discretised simulation of a titration curve. It is highly non-linear. Dead-beat control of the titration system is possible. The system is described by the formula:

$$(22)\qquad y(k+1) = y(k) + e^{-3\,|y(k)|}\cdot x(k)$$

where $y(k)$ is the output of the plant at timestep k, and $x(k)$ is the input of the plant at timestep k. Both $x(k)$ and $y(k)$ fall in the domain [-2,+2].

B.4 Trolley

The trolley, also called the 'cart positioning system', consists of a small cart which is controlled by an electrical signal. The signal can make the trolley move along a piece of rail. The controller should move the trolley to designated spots by manipulating the signal. A successful controller moves the trolley to the desired position as quickly as possible, without running it off the track.

The plant has just one input, the voltage, which falls in the range [-10V,+10V]. It has two outputs:

- The position of the trolley, in the range [-0.25,+0.25].
- The speed of the trolley, in the range [-10,+10].

The state of the system is described by the current position and the current speed of the trolley. The model of the trolley is defined by the differential equation:

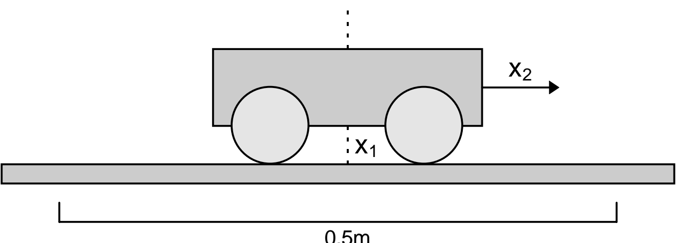


figure 65: The trolley.

$$(23) \qquad x' = \begin{bmatrix} 0 & 1 \\ 0 & -t^{-1} \end{bmatrix} \cdot x + \begin{bmatrix} 0 \\ k \cdot t^{-1} \end{bmatrix} \cdot u$$

where

x_1 is the position.
x_2 the speed of the trolley.
k is the system gain.
t is a time constant.
u the input voltage.

A negative speed indicates the trolley is moving to the left. *k* determines how fast the trolley can go. *t* determines how fast the trolley reacts to a change in the input. The input voltage *u* is limited to values between -10 and +10V.

The implementation of this plant in Elegance integrates the model by using the Runge-Kutta 4 method. When initialising the plant, the following internal parameters need to be specified:

- The system gain *k*, default 0.5.
- The time constant *t*, default 0.05.
- The integration step size, default 0.001.

The trolley is copied from Jarmulak's NCWB (Jarmulak 1994b).

B.5 Trolley 2-dimensional

The trolley 2-dimensional, designed specifically for Elegance, is a version of the trolley which moves in a two-dimensional space. Its implementation is equivalent to the implementation of the trolley, except that the input consists of two signals:

- A voltage, falling in the range [-10V,+10V], which is equivalent to the voltage of the trolley.
- The direction, in radians, falling in the range [-2π,+2π], in which the force, which is

the result of the voltage, is applied.

The implementation uses the two inputs to calculate a vector in two-dimensional space, which is the force applied to the cart. This force is split in a horizontal and a vertical force, and these are processed in the same way as the processing is done in the one-dimensional trolley.

The output of the plant consists of the following four parameters:

- The horizontal position of the cart, in the range [-0.25,+0.25].
- The vertical position of the cart, in the range [-0.25,+0.25].
- The speed of the cart, in the range [-10,+10].
- The direction in which the cart is moving, in the range [-2π,$+2\pi$].

B.6 Pole balancing system

The pole balancing system, also called the 'inverted pendulum', consists of a small cart on a piece of rail of 5 meters long, which is controlled by a force. A pole stands on one end on the cart, and it's the objective of the controller to keep the pole from falling by moving the cart. A successful controller can keep the pole from falling for an unlimited time, without running the cart off the track.

The plant has just one input, the force, and four outputs which describe the state of the system, namely:

- The pole angle, in the range [-75,+75].
- The pole angle velocity, in the range [-360,+360].
- The cart position, in the range [-2.5,+2.5].
- The cart velocity, in the range [-5,+5].

The input force can take on any value between -10N and +10N. There is a second version of the pole balancing system. The model of the pole balancing system is defined by the differential equation:

$$(24) \qquad \begin{cases} \ddot{\theta} = \dfrac{mg\sin\theta - \cos\theta\left[F + m_{\mathrm{p}} l\dot{\theta}^2 \sin\theta\right]}{(4/3)ml - m_{\mathrm{p}} l\cos^2\theta} \\ \ddot{\rho} = \dfrac{F + m_{\mathrm{p}} l\left[\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta\right]}{m} \end{cases}$$

where

ρ is the cart position.
ρ' is the cart velocity.
θ is the pole angle.
θ' is the angular velocity of the pole.

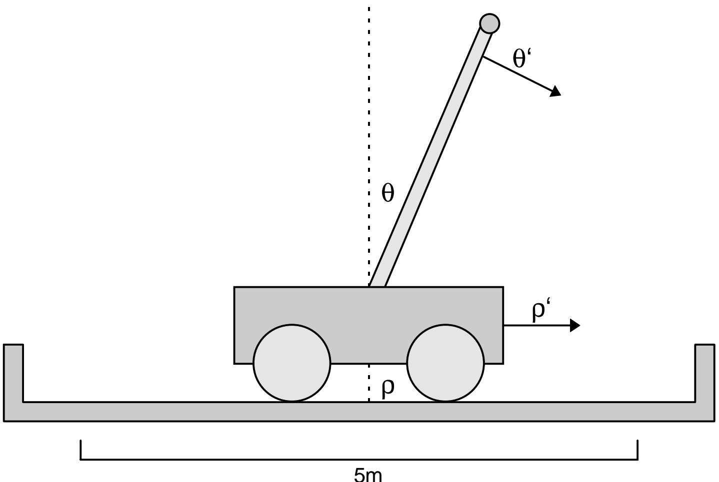


figure 66: The pole balancing system.

l is the length of the pole.
m_p is the mass of the pole.
m is the combined mass of the cart and the pole.
F is the control force.
g is the acceleration due to gravity.

The implementation of this plant in Elegance integrates the model using Euler's method. When initialising the plant, the following internal parameters need to be specified:

- The pole length, default 0.5.
- The pole mass, default 0.1.
- The cart mass, default 1.
- The gravity, default 9.8.
- The integration step size, default 0.001.

The pole balancing system is quite common and used by many researchers, like by Jarmulak in his NCWB (Jarmulak 1994b). However, this implementation, which is a copy from Jarmulak's implementation, differs from most implementation by allowing all values in the range [-10N,+10N] for the control force. It is more common to allow only -10N and +10N.

B.7 Bang-bang pole balancing system

The bang-bang pole balancing system is the same as the pole balancing system, except for two differences:

- The control force can only be -10N or +10N.
- The input to the plant, which is essentially unlimited, is remapped to the range [0,1], using a sigmoid with a learning rate of 1 (equation 12). The input is then considered to be the *chance* that the input force will be +10N. If chance decides that the input is not +10N, it's -10N.

The bang-bang pole balancing system is quite common and used by many researchers. The implementation used here, with the input considered to be a chance, is also used by Whitley in his GENITOR experiments (Whitley 1993).

B.8 Bioreactor

The bioreactor is a constant volume, isothermic, continuous flow, stirred, tank reactor. It contains water, biological cells and nutrient. Water containing nutrient is continuously added to the tank. The volume is kept constant by draining contents of the tank equal to the incoming rate. The bioreactor is controlled by means of the flow rate. The state of the tank is described by the cell mass and the amount of nutrient in the tank. It's the objective of the controller to get the bioreactor to stabilise the cell mass in the tank at a certain level. The following formula describes the bioreactor:

$$(25)\quad \begin{cases} \dfrac{dC}{dt} = -Cw + C(1-N)e^{\frac{N}{\gamma}} \\ \dfrac{dN}{dt} = -Nw + C(1-N)e^{\frac{N}{\gamma}}\dfrac{1+\beta}{1+\beta-N} \end{cases}$$

where

C is the cell mass.
N is the amount of nutrient.
w is the flow rate.
β is the cell growth rate.
γ is the nutrient consumption rate.

The model is integrated using the Runge-Kutta 4 method. The bioreactor is very chaotic and it's difficult to reach a setpoint. Most setpoints are not stable.

When initialising the plant, the following internal parameters must be specified:

- The cell growth rate, default 0.02.
- The nutrient consumption rate, default 0.48.
- The integration step size, default 0.001.

In theory, with the default values, there are stable setpoints at (C,N) = (0.1207,0.8801) for $w = 0.75$ and at (C,N) = (0.2107,0.7226) for $w = 1.2$.

The bioreactor is also found in Jarmulak's NCWB (Jarmulak 1994b).

C Extending Elegance

One of the requirements of Elegance was that it should be relatively easy to add new plants and new genetic operators. Provided one has access to the source code and the Borland Delphi compiler, this is indeed pretty simple. In this appendix, the process of extending Elegance is discussed.

C.1 Adding a plant

Adding a plant to Elegance is very easy. The process consists of three steps:

- The plant type should be defined. This is no more than the adding of two lines of code to the file *PLANTTYP.PAS*: a code for the new plant (for instance *PlantHeater* for a heater plant), and a description for the new plant (for instance “Heater”).
- The plant code should be written and saved in a new unit.
- A plant creation call should be specified. This is two lines of code in the file *PLANTCRE.PAS*: the unit name of the new plant should be added to the *uses* clause, and a call to the *Create* method of the plant should be added to the *case* clause which also contains *Create* calls for all the other plants.

The code for the new plant should either be derived from an existing plant, or from the abstract plant type *TPlant*. Both cases are treated in the same way, except that when a plant is derived from an existing plant, it may be that less code is required.

Only three methods need to be specified for the new plant:

- The *Create* method, which should specify the input-, state- and output parameters of the new plant, using the predefined methods *AddInput*, *AddState* and *AddOutput*. This is one line of code for each parameter.
- The class function *PlantType*, which returns the plant type. This is one line of code.
- The *DoCycle* method, which calculates one cycle of the plant. In this method the

plant input can be read from the *InputValue* array, and the outputs must be written to the *OutputValue* array. The method should call *CheckInput* at the start and *CheckOutput* at the end, which will make the plant fail if the value boundaries are crossed.

- A method *CheckParms* may be specified if the programmer wishes to do extra checks on the parameter values when the user enters them interactively.

```
unit Siso;

interface

uses
      Plant, PlantTyp, Value;

type
      TSiSo = class( TPlant )
      public
            constructor Create( const Step: TValuePtr ); override;
            procedure DoCycle; override;
            class function PlantType: TPlantType; override;
      end;

implementation

constructor TSiSo.Create( const Step: TValuePtr );
begin
      inherited Create( Step );
      {             Name          Value   Min     Max     Step    }
      AddInput(     'Input',      0,      -2,     2,      0.1     );
      AddOutput(    'Output',     0,      -2.5,   2.5,    0.1     );
end;

procedure TSiSo.DoCycle;
var
      x, y: TValue;
begin
      inherited DoCycle;
      CheckInput;
      x := InputValue[0];
      y := OutputValue[0];
      y := x + y / (1 + y * y);
      OutputValue[0] := y;
      CheckOutput;
end;

class function TSiSo.PlantType: TPlantType;
begin
      Result := PlantSiSo;
end;

end.
```

figure 67: The complete source code for the unit *SISO.PAS*.

As an example, the complete code for the SISO system is shown in figure 67. The SISO system is the most simple plant in Elegance, but the source code for the other plants isn't much longer, depending on the complexity of the plant function.

That's all. Everything else needed for the plant handling is done by Elegance.

C.2 Adding a genetic operator

Adding a genetic operator to Elegance is just a little bit more complex than the adding of a plant. The basic steps are the same:

- The genetic operator type should be defined. This is no more than the adding of two lines of code to the file *GENOPTYP.PAS*: a code for the new genetic operator (for instance *OperatorHillclimb* for a hillclimbing operator), and a description for the new genetic operator (for instance "Hillclimb").
- The genetic operator code should be written and saved in a new unit.
- A genetic operator creation call should be specified. This is two lines of code in the file *GENOPCRE.PAS*: the unit name of the new genetic operator should be added to the *uses* clause, and a call to the *Create* method of the genetic operator should be added to the *case* clause which also contains *Create* calls for all the other genetic operators.

The code for the new genetic operator should either be derived from an existing genetic operator, or from the abstract genetic operator type *TGeneticOperator*. Both cases are treated in the same way, except that when a genetic operator is derived from an existing genetic operator, it may be that less code is required.

The following methods need to be specified for the new genetic operator:

- The *Create* method, which should specify the parameters (if any) of the new genetic operator, using the predefined method *AddParameter*. This is one line of code for each parameter. Also, the number of parents should be specified in the variable *GONrOfParents*, and the number of children in the variable *GONrOfChildren*. The maximum number of parents is 3, and the maximum number of children 2. There can't be more children than parents.
- The class function *GeneticOperatorType*, which returns the genetic operator type. This is one line of code.
- The *Execute* method, which creates new children in the chromosome array *Child*, using the parents in the chromosome array *Parent*. Access to the parameters goes through the array *ParameterValue*. The method can make use of the predefined function *CreateCopy*, which creates a copy of a chromosome. There are several other handy predefined functions, which won't be discussed here. At the end, for each created child the *Legalize* method should be called, which makes sure that the child confirms to the specifications laid down in the controller and the genetic algorithm.

```
unit Binmut;

interface

uses
    Genop, GenopTyp, SysUtils, Chromo, Control;

type
    TBinaryMutation = class( TGeneticOperator )
    public
            constructor Create( const Contr: TControllerPtr ); override
            class function GeneticOperatorType: TGeneticOperatorType; override;
            procedure Execute( const Parent: array of TChromosome;
                    var Child: array of TChromosome ); override;
            class function WorksWithBinaryValues: Boolean; override;
            class function WorksWithVariableLength: Boolean; override;
            class function WorksWithFixedLength: Boolean; override;
            class function WorksWithPID: Boolean; override;
            class function WorksWithNeural: Boolean; override;
    end;

implementation

constructor TBinaryMutation.Create( const Contr: TControllerPtr );
begin
    inherited Create( Contr );
    GONrOfParents := 1;
    GONrOfChildren := 1;
    {              Name                     Value   Min     Max     Step    }
    AddParameter( 'Chance per allele',      0.1,    0,      1,      0.01    );
end;

class function TBinaryMutation.GeneticOperatorType: TGeneticOperatorType;
begin
    Result := BinaryMutation;
end;

procedure TBinaryMutation.Execute( const Parent: array of TChromosome;
    var Child: array of TChromosome );
var  I: Integer;
begin
    inherited Execute( Parent, Child );
    Child[0] := CreateCopy( Parent[0] );
    for I := 0 to Child[0].CLength - 1 do
            if Random < ParameterValue[0] then
                    if Child[0].Coding[I] = '0' then
                            Child[0].Coding[I] := '1'
                    else
                            Child[0].Coding[I] := '0';
    Child[0].Legalize( Controller );
end;

{ class functions WorksWith... are found here }

end.
```

figure 68: The source code for the binary mutation genetic operator.

- For each situation for which the genetic operator can be used, a class function needs to be specified which explicitly allows the genetic operator to be used in that situation. These class functions should return *True*. If a class function is not specified, it automatically is *False*. The class functions are: *WorksWithRealValues*, *WorksWithBinaryValues*, *WorksWithFixedLength*, *WorksWithVariableLength*, *WorksWithPID* and *WorksWithNeural*.
- If the operator works with random values, this can be indicated by specifying a class function *WorksWithRandomValues* which returns *True*. The user can then indicate that he wants to use inverse exponential random values.
- A method *CheckParms* may be specified if the programmer wishes to do extra checks on the parameter values when the user enters them interactively.

As an example, the code for the binary mutation operator is shown in figure 68. Only the class functions *WorksWith*... are left out (for simplicity - they all return *True*).

The biggest difficulty for the programmer comes in the specification of the more complex genetic operators. It will often be necessary that the programmer has a good understanding of the structure of the chromosome in the coding. For instance, the chromosome object as a whole can be accessed as a string, or the individual weights and connections can be accessed as an array. A chromosome with real values obviously has to be treated differently from a chromosome with binary values, and variable length chromosomes are again a special case. However, it leads too far to discuss these coding intricacies here.

C.3 Other extensions

The controller object *TController* and the signal object *TSignal* (which is used to generate setpoint signals) are constructed in the same way as the plant object and the genetic operator object. Adding a new signal type is therefore just as easy as adding a new plant.

Adding a controller, however, is more difficult, since the controller structure is intertwined with the structure of the genetic algorithm. If a new type of PID controller needs to be added, this can easily be done, simple by deriving it from an existing PID controller. Adding a new neurocontroller, which looks a lot like one of the existing neurocontrollers, should also be quite simple. Adding other controller types will lead to very radical changes in the genetic algorithm code, though.

Other changes will often require some studying of the code before they can be made, especially if they intervene in the evolution process. Small adaptations to existing processes shouldn't prove to be too difficult, since for the most part the code is rigidly structured. Testing of changes can often be done with the built-in test facilities in Elegance.

D A paper on Elegance

When I was almost finished with this thesis, an opportunity arose to write a paper on the application of GAs in neurocontrol development, as a follow-up on a paper by Jarmulak et al.*, titled* Universal Approach to Neural Process Control Illustrated on a Biomass Growth Model *(Jarmulak 1995). Professor Kerckhoffs suggested I should try to write a first version of this paper before I finished my thesis, and add it as an appendix, which is not uncommon practice at the TU Delft.*

Later on, the paper has been reworked, together with Jarmulak's paper mentioned above, and has been offered for publication in an international journal as a collaborated paper by Jarmulak, Kerckhoffs and me. At the moment of this writing, this paper is under review and will probably be published in 1997.

The original paper has been revised later on by professor Kerckhoffs, who thinks it can probably be published in the Proceedings of a 1997 congress. I have also added some revisions of my own. Professor Kerckhoffs is still not finished with his revisions, but since I think the version with his preliminary revisions is an improvement over the original paper, I have decided to replace the version of the paper in this appendix with the revised version.

Using Genetic Algorithms for Neural Reinforcement Controller Design in a Simulation Environment

P.H.M. Spronck, E.J.H. Kerckhoffs

Delft University of Technology
Faculty of Technical Mathematics and Informatics
Zuidplantsoen 4, 2628 BZ Delft, The Netherlands

Abstract: In production often highly non-linear processes are encountered, which need to be controlled. In this paper, attention is focused on neural reinforcement control of simulated plants. Reinforcement control uses an evaluation of the performance of the controller in a practical situation to adapt the controller to work better. The design of a neural controller in a reinforcement situation is not a trivial task. A method is discussed to design such neural controllers using genetic algorithms. To test the viability of this approach a specific software environment has been built, which enables performing experiments with many different genetic algorithm configurations. As an example, the design of a controller for a bioreactor simulation is discussed.

Keywords: adaptive control, reinforcement control, neural control, artificial intelligence, neural networks, genetic algorithms, software tools.

1. Introduction

In many situations in production industry and modern business there are processes, often called *plants*, which need to be controlled. Frequently, the controller function is taken by a human being or a mechanical controller. However, mechanical controllers are only suitable for simple situations, and human beings are expensive and cannot always react quickly enough in situations where large numbers of parameters or fast-changing processes are concerned. In modern control plants are computer controlled. The control software can be based on numerical procedures, tradional artificial intelligence techniques (reasoning systems), neural networks and fuzzy logic techniques.

In this paper we consider neural control, that is, control in which neural networks are involved. In neural control, it is tried to exploit the learning capabilities of neural networks. The design and training of neural networks is not an easy task. If a good conventional or neural model of the plant to be controlled is available, there are ways to create a good neural controller for the plant concerned. However, such a model can not always be built easily, and sometimes not at all. In that case one option that remains is to use reinforcement control, that is, to develop the neural controller according to the success, or the lack of success, of the performance of this controller in a practical situation. No plant model is needed in this case.

Genetic algorithms are search algorithms based on the principles of natural selection and natural genetics, which may be particularly suited to develop neural reinforcement controllers. This is studied in the research reported in this paper. There has not been much research in the application of genetic algorithms in the design of neural controllers, and the question of the viability of this approach is still not settled. The

approach is promising, though, so more research in this direction is certainly warranted. For this kind of research many experiments need to be done. A software environment has been created to perform, in a flexible way, a large number of different experiments in this respect.

The paper first gives a short introduction to genetic algorithms. This is followed by a discussion of neural control, and how genetic algorithms can be applied to the design of neural reinforcement controllers. Then the software environment created to do the necessary experiments is presented, and as an example the development of a neural controller for a bioreactor simulation is discussed.

2. Genetic algorithms

Genetic algorithms are search algorithms based on the principles of natural selection and natural genetics. They have been invented by John Holland, who in 1975 published his book *Adaptation in Natural and Artificial Systems* [1]. It took years before genetic algorithms became a major interest in the artificial intelligence community, but since the early '90s they have got a lot of attention and have become widely accepted as a powerful tool to handle complex optimisation problems.

While most optimisation techniques start with one potential solution to a problem, and adapt that solution to ultimately get it to an optimum, genetic algorithms work with a set of potential solutions to the problem at hand. This set is called a *population*, and the potential solutions in the set are called *individuals*. The individuals are not simply stored in the population; they are *encoded*. For instance, an individual may be translated by some method to a binary string. Such a coded individual is also called a *chromosome*. One character of a chromosome is called a *gene*, and a gene value is called an *allele*. Each individual in the population has been given a *fitness measure*, which indicates how well this individual performs in solving the problem, in relation to the other individuals in the population.

To get new and hopefully fitter individuals, the genetic algorithm applies *genetic operators* to individuals which are selected from the population. A genetic operator takes one or more *parent* individuals, and performs some action on them to produce

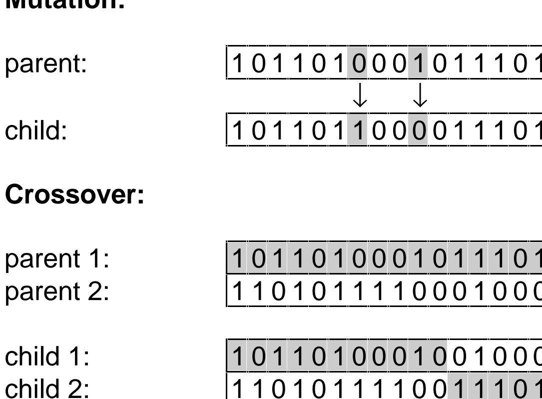


Figure 1: Examples of two genetic operators.

one or more new *child* individuals. For example, the *mutation* operator takes one individual as parent and makes a few changes in it to produce a child, while the *crossover* operator takes two parents, cuts them both in two parts, and exchanges the tails to produce two children (see figure 1). The selection of the parent chromosomes normally goes according to the fitness rates: the fittest individuals have a greater chance of being selected to procreate than the less fit individuals. Newly generated individuals either get inserted into the population, replacing other individuals, or are placed in a new population which will eventually replace the old population. The production of new individuals, called the *evolution* process, continues until some predefined goal is reached, most commonly until a maximum number of new individuals has been produced. At that point, the fittest individual in the population is considered to be the sought solution to the problem.

Genetic algorithms may seem to be a completely random process, but in practice they work quite well, providing the right configuration (population size, genetic operators, replacement policy, etcetera) is chosen. Strong points of genetic algorithms are that they are robust, meaning that they work well in many different environments and on many different problems; that they seek out a global optimum where most other techniques only lead to a local optimum; and that they need no more than a fitness measure to work. They are relatively slow, but since they are inherently parallel, it is easy to speed them up if more processors are available. A weakness of genetic algorithms is that they are not guaranteed to lead to an acceptable solution, not even to a mediocre one. Experience in the design of the genetic algorithm, and a sound analysis of the problem domain are needed to insure that the genetic algorithm will indeed do what it is required to do.

Those interested in more details of genetic algorithms are referred to the excellent introduction to the field by David Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning* [2].

3. Neural control

A *plant* is a process which maps an input to an output. The plant may have internal states, which influence the mapping, meaning that some specific input does not in general lead to one specific output. The input to the plant is presented by a *controller*, which should get the plant to produce a specific predetermined output. Such a target output is called a *setpoint*. The controller receives the plant output to guide the determination of the plant input needed to reach the setpoint. This is shown schematically in figure 2.

In the last decades, often straightforward computer programs have been employed as

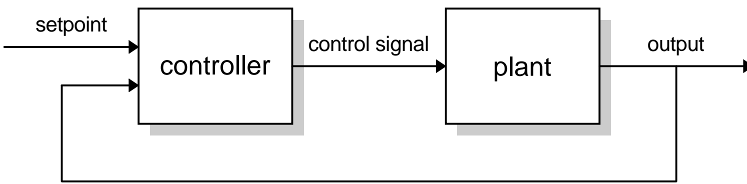


Figure 2: Plant control.

controllers. The more complex the plant becomes, the more difficult it is to design a good computer controller. Furthermore, really good computer controllers are mostly developed to control linear systems. In practice, however, plants are rarely linear. Of the AI techniques which have been used in control design, neural networks have shown some promise. Neural networks have the ability to learn a non-linear function, and can therefore be used to control non-linear plants. There are basically two ways in which neural networks can be employed in control systems. First, the neural network can be trained to be the plant controller itself. Second, the neural network can be trained to be a model of the plant (called a neural *identifier*), and this is then incorporated in the control system.

The model-based approach has been studied and experimented with by Jacek Jarmulak using his NeuroControl Workbench [3,4]. He considered two kinds of models: forward and inverse. Inverse models are presented with a plant output, and react with the plant input needed to get that output. Since that is exactly what we wish a controller to do, an inverse model is theoretically equal to a controller. However, a good inverse model of a plant is far more difficult to build than a good forward model. Moreover, a direct inverse controller is not stable in practice and therefore useless. Forward models can be used in several configurations, of which the most successful is the so-called model-based predictive control, in which a plant model is used to improve plant input and a neural controller is trained on-line with these improvements.

In Jarmulak's NeuroControl Workbench neural models are trained with backpropagation. Backpropagation works with a training set. A training set may consist of a number of plant inputs with the corresponding plant outputs. The neural network is then trained to produce the outputs when presented with the inputs. As stated before, the problem is that a plant will in general not react in one specific way to some input, while a training set can only attach one output to an input. To deal with this problem, the controller input consists of not only the desired plant output, but also of the current plant output and of some of the plant input and output's history. In this way a static backpropagation network can learn a dynamic input-output relation. This strengthens the approach considerably, but it is of course not guaranteed that an acceptable plant model can be designed.

An alternative is to use a neural network as the controller itself, and to train it with *reinforcement learning*. Reinforcement learning by definition does not make use of a plant model or a training set. Instead, observation of the performance of the controller in practical situations is used to adapt the controller to perform better. This is called *reinforcement control*. Reinforcement control is used by several companies, but the precise approach they use is confidential, and therefore literature about this subject is scarce. The design of a neural network as a controller is actually an optimisation problem. In reinforcement control the only information we have to adapt the neural controller concerns evaluations of the performance of the controller. Since genetic algorithms are used for optimisation, and since they need no more than a fitness evaluation to do the job, they seem to provide a viable approach to the design of neural controllers in a reinforcement situation.

Note that genetic algorithms could also be used to design a neural identifier which is then used in the controller. However, there are several good conventional techniques for this purpose, so genetic algorithms are not a logical first choice in this respect. Also note that genetic algorithms are only suitable for off-line training and are therefore

dependant on a good plant simulation.

4. Genetic algorithms and neural controllers

A lot of research has been done concerning the use of genetic algorithms to optimise neural networks. This certainly does not mean that researchers are unanimous about what makes a genetic algorithm suitable for this task. Virtually every aspect of these genetic algorithms is subject for debate. For instance, concerning the chromosome encoding, some researchers prefer the use of real values for the connection weight encoding, while others defend the use of binary encoded weights. A lot of the points of discussion stem from the problem of so-called *competing conventions* (more aptly named the *structural/functional mapping problem*). The problem of competing conventions is about the fact that one particular mapping from input to output can be encoded in several different ways. For instance, in a layered feedforward neural network, the nodes in one layer (including their connections) can be exchanged, leaving the neural network functionally the same, but structurally different. For an example, see figure 3.

The possibility of the occurrence of competing conventions leads to a vast increase in the size of the solution space, which may lead to an increase in the time needed for the evolution process. Moreover, competing conventions also virtually nullify the beneficial effect of the crossover operator. This is because for the crossover operator to be executed in a useful manner on two chromosomes functionally equivalent neural nodes should be encoded in the same location on those chromosomes. For example, suppose we have devised an encoding mechanism in which every neural node is encoded as one gene of a chromosome, and that the optimal neural network is encoded as *ABCDEF*. Suppose there are two quite fit neural networks in the population, encoded as *ABCDEX* and *XBCDEF*. Performing the crossover operator on these two chromosomes has a great chance of resulting in the optimal chromosome. However, if competing conventions are allowed in the population, the second neural network could, for instance, be encoded as *FEDCBX*. In this case, use of the crossover operator would not result in the optimal chromosome, but instead would very probably produce children which are less fit than their parents. To solve the problem of competing conventions, different researchers use different approaches. Some just ignore the problem, some advocate the use of a small population combined with a high mutation rate, some use special genetic operators, and some rearrange the structure of one of the chromosomes before the crossover is executed.

For neurocontroller evolution in a reinforcement control situation, another aspect is

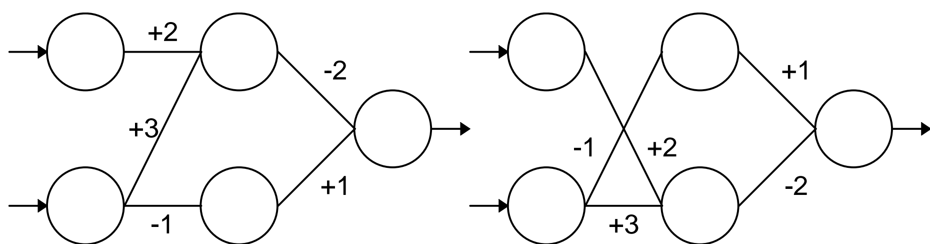


figure 3: Competing conventions: two structurally different but functionally equivalent neural networks.

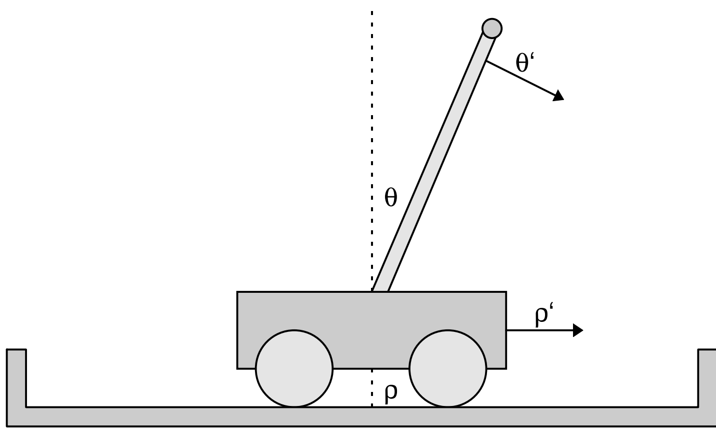


figure 4: The pole balancing system. The state of the system is described by the pole angle θ, the angular velocity θ', the cart position ρ, and the cart velocity ρ'.

added to the genetic algorithm design, namely how the test run, executed to determine the fitness of a neurocontroller, should be performed. Very little research has been done in this area. Darrell Whitley is one of the few researchers who has examined the subject of neurocontroller evolution in a reinforcement environment [5]. As a test plant he uses a *pole balancing system*. This is a system which consists of a cart on a rail, on which a pole rests. The pole can fall to the left and right. The cart is controlled by applying a force, and it is the objective of the controller to keep the pole from falling (see figure 4). The reinforcement aspects of the pole balancing controller evolution are easily determined. The plant fails if the pole falls, therefore the fitness is determined by the length of time the controller can keep the plant from failing. This kind of fitness is called *time-until-failure* based fitness. A maximum test run length needs to be set, and the controller is considered to be perfect if the pole remains balanced for that amount of time. The only remaining reinforcement aspect is the initial plant state used in the test run. Whitley starts by placing the cart at one end of the rail, and the pole leaning over at some specific angle to the other side of the rail.

There are many plant types for which the controller cannot be evolved with time-until-failure-based fitness. For instance, a *trolley* is a plant which consists simply of a cart on a rail, and the controller has to direct the cart to specific positions on that rail. The plant fails if the cart drives off the rail, but just keeping the plant from failing is not sufficient for the controller to be any good. The controller should also minimise the distance between the cart position and the setpoint position. A straightforward way of implementing fitness determination for controllers for plants like the trolley is to use the mean square error (MSE) over the test run, wherein the error is defined as the difference between the current plant output and the setpoint for this output. This is called *MSE-based* fitness determination, and in this case the inverse fitness is used, meaning that the controller which has the lowest MSE is considered to be the most fit. If the plant does not fail during the test run (that is, if the cart does not drive off the rail), the inverse fitness is simply the sum of the squares of the error for each time step, divided by the number of time steps. If the plant does fail, this situation can be handled in several ways. We can simply reset the plant and continue the run; or we can award the maximum error for all the remaining time steps; or we can award the MSE for the time steps until the failure for the rest of the run. In the latter case, it would be wise to

also set some penalty on premature failure.

Very little research has been done in the subject of genetic algorithm based evolution of neurocontrollers in a reinforcement environment, and most of it is concentrated on controllers with time-until-failure based fitness. The handful of papers published about the subject are, however, optimistic about the possibilities and results. Because this research is mainly of an experimental nature (there is little theory about what makes a genetic algorithm work well), it would benefit from the availability of a software environment in which in a flexible way large numbers of genetic algorithm configurations can be tested on several different plant types. Such a software environment has been developed at Delft University of Technology [6]. It is described in the next section.

5. The software environment “Elegance”

“Elegance”, which is an acronym for Engineering Laboratory for Experiments with Genetic Algorithms for Neural Controller Evolution, is a software environment constructed to experiment with genetic algorithm configurations for the design of neurocontrollers, particularly in a reinforcement control situation. Elegance’s view on the neurocontroller evolution process is shown in figure 5. The controller directs the plant with a control signal. This results in a plant output, which is sent back to the controller and to a history entity. The history entity can provide the controller with the last n plant outputs through so-called tapped-delay lines (TDLs). A setpoint generator indicates to the controller which desired plant output should be produced. The structure consisting of the plant, controller, setpoint generator and history entity is called the *control loop*.

The genetic algorithm contains a population of controllers. The fitness of these controllers is determined by placing them, one-by-one, into the control loop, making the control loop run for a certain length of time, and by examining the results of this test run. The genetic algorithm generates new controllers by applying genetic operators on

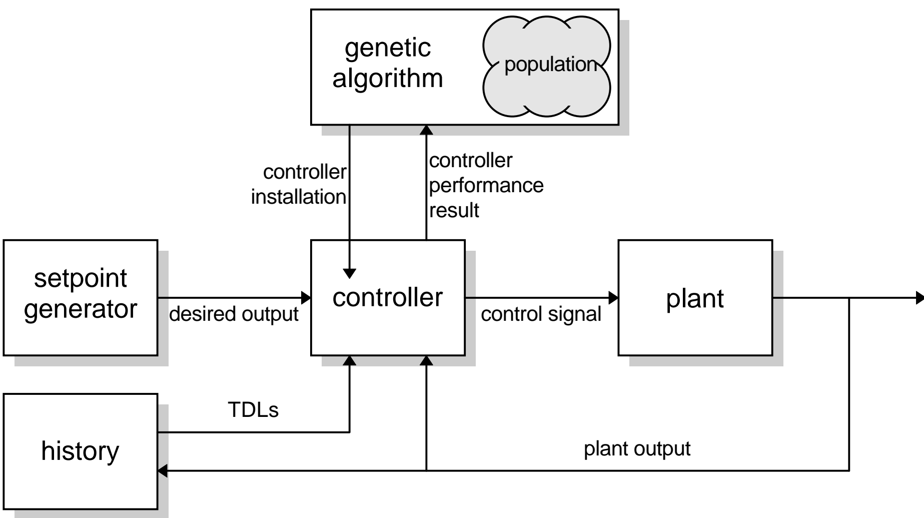


figure 5: The basic Elegance control structure.

parent controllers selected from the population, determining the fitness of these new controllers, and inserting them into the population, replacing existing controllers. Since the purpose of Elegance is to test many different genetic algorithm configurations, there is offered a great flexibility in the design of the genetic algorithm. In the following we give non-exhaustive list of Elegance's functionalities:

- Elegance supports both weight determination and the combination of architecture design and weight determination.
- Elegance supports both real valued chromosomes [5] and binary valued chromosomes of fixed and variable length [8].
- Elegance supports both time-until-failure-based fitness [5] and MSE-based fitness determination. MSE-based fitness can be configured to use a basic run length, which gets increased until the change in MSE falls below a certain threshold.
- Elegance supports several fitness scaling techniques, like ranking [2].
- Elegance supports a complexity penalty and a premature failure penalty.
- Elegance supports a wide range of genetic operator types, among which several kinds of weight mutation, several kinds of connection presence mutation, granularity mutation, several kinds of crossover operators and the GA-simplex operator (most genetic operators come from Montana and Davis [7], GA-simplex is designed by Maniezzo [8], and some operators have been designed specifically for Elegance).
- Elegance supports Whitley's adaptive mutation technique [5].
- Elegance supports Thierens' treatment of competing conventions [9].
- Elegance supports several kinds of replacement policies, like crowding (meaning that the individual to be replaced is selected deterministically from a randomly selected subset of the population) [2].
- Elegance supports feedforward neural networks, layered feedforward neural networks and recurrent neural networks as neurocontrollers. For comparison, it also supports conventional PID controllers.
- Elegance supports several different plant simulations.
- It is easy to add new plants and new genetic operators to Elegance.

For completeness, some of Elegance's limitations should also be mentioned:

- Elegance only supports direct encoding (meaning that a neural network is always encoded to all its details).
- The neural network size is in the first version limited to a maximum of 75 nodes (which is more than enough for controlling the plants currently implemented in the program).
- The chromosome structure design is fixed.
- As yet, there is no combination of genetic algorithms with a local optimisation technique supported. Such a combination would make a useful addition.

The design of Elegance has been inspired by the design of Jarmulak's NeuroControl Workbench [3]. There are six basic objects in the program, namely the project, which contains the five other objects: the plant, the controller, the setpoint generator, the control loop display and the genetic algorithm (the history entity is implemented as part

of the plant). These six objects can be defined and maintained separately. After they have been defined, an evolution run can be started, and when that run is finished, the results can be examined by running the best controller found. To make Elegance available to many users, it is implemented under Microsoft Windows 3.1. Because of the large number of floating-point calculations needed during the evolution process, a fast PC with a mathematical coprocessor is recommended, although the simpler experiments (like those with a pole balancing system) run smoothly on a low-end 486 PC.

6. Controlling a bioreactor simulation

The preliminary experiments done with Elegance were the design of a neurocontroller for a pole balancing system simulation and the design of a neurocontroller for a trolley simulation. The pole balancing system, however, was found to be so very simple to control, that randomly generating neurocontrollers with five hidden nodes would produce a good pole balancing neurocontroller within a few hundred trials. This means that almost any genetic algorithm configuration would produce good results, and therefore that the pole balancing system is not a very good test case to decide if the application of genetic algorithms in neurocontroller design is a promising technique. The trolley was found to be easiest to control with a neurocontroller with no hidden nodes at all, or at most one hidden node. Although the evolution techniques proved their worth by reducing neural networks with more hidden nodes to neural networks with at most one hidden node, this indicates that the trolley is also not a very suitable system to test these genetic algorithm based techniques.

Elegance does, however, contain more complex plants, from which the bioreactor simulation was selected to perform the first really challenging experiments. The bioreactor is a tank reactor containing a biological cell mass in water, which feeds on nutrient added to the tank, while water is drained from the tank at a rate equal to the nutrient input flow. The controller can adjust the flow rate. The objective of the controller is to stabilise the cell mass and the amount of nutrient in the tank at certain setpoints. The following formula describes the bioreactor:

$$\begin{cases} \dfrac{dC}{dt} = -Cw + C(1-N)e^{\frac{N}{\gamma}} \\ \dfrac{dN}{dt} = -Nw + C(1-N)e^{\frac{N}{\gamma}} \dfrac{1+\beta}{1+\beta-N} \end{cases}$$

where C is the cell mass, N the amount of nutrient, w the flow rate, β the cell growth rate and γ the nutrient consumption rate. The bioreactor is highly non-linear and chaotic, and there are few stable setpoints (which are setpoints for which the flow rate can remain constant).

It was decided to evolve a neurocontroller which could switch a bioreactor (with β = 0.02 and γ = 0.48) between two stable setpoints. This could mean that the resulting neurocontroller can *only* be used for those two setpoints. However, this is not necessarily the case. If a general solution to the control problem is easier to implement

than a solution which can only be used for the two selected setpoints, it is likely that this general solution will be the one found by the genetic algorithm. Also, using a random setpoint generator would introduce a large amount of chance in the fitness determination process, especially with such a chaotic plant, which might obstruct the evolution process. The two setpoints chosen were $(C,N) = (0.1207,0.8801)$, which is stable with a flow rate $w = 0.75$, and $(C,N) = (0.2107,0.7226)$, which is stable with a flow rate $w = 1.2$.

Elegance works with a predefined maximum number of nodes in the neural network configuration, from which it can remove hidden nodes, but to which it cannot add more nodes. The needed initial network configuration largely depends on the complexity of the problem. The choice of the maximum number of nodes should be made carefully. If it is too big, the evolution process will take much longer than necessary. If it is too small, the evolution process won't succeed. For the bioreactor, it was decided to choose a configuration for which Jarmulak had already found it would evolve into a good bioreactor identifier [4]. The reasoning behind this decision was that for a model-based controller the plant model is often by far the most complex part of the controller, so the needed identifier complexity could probably be about the same as the needed controller complexity for Elegance. The chosen neurocontroller configuration was therefore set to:

- 8 input nodes, consisting of two setpoint nodes (the desired cell mass and the desired amount of nutrient), two plant output nodes (again, the cell mass and the amount of nutrient), and two TDLs (totalling four nodes: the previous plant outputs, and the plant outputs before that).
- two hidden layers of 20 nodes each.
- 1 output node, the flow rate.

As activation function an arctangent was chosen with a range of [-1,+1]. This configuration has a total of 580 possible connections, which makes the solution space quite large indeed.

The genetic algorithm configuration was designed as follows:

- Encoding was done with real values for the weights.
- Fitness determination was MSE-based, with ranking as scaling technique. The basic run length was 500 time steps (one time step equalling 0.1 simulated seconds), which would be continuously increased with 100 time steps until the change in MSE would have dropped below 0.02. Premature failure is, at least in theory, not possible with the bioreactor.
- Population size was set to 100.
- Elitism was applied (meaning that the best individual in the population would never be selected for replacement).
- Duplicate checking was performed (meaning that the genetic algorithm would not allow duplicates in the population).
- Viability checking was applied (meaning that the genetic algorithm would not allow neural networks which have output nodes which are not connected via some path to at least one of the input nodes).

- Incest prevention was applied with three alleles (meaning that to be allowed to be used as parents, two chromosomes should differ by at least three alleles).
- A treatment for competing conventions was applied.
- As replacement policy crowding was applied.
- Weight initialisation was done in a range of [-5,+5], with a connection presence chance of 0.5 (meaning that about 50 percent of all possible connections would be initially activated in a neural network).

The following genetic operators were used:

- Randomisation, which in effect generates almost randomly new chromosomes, slightly based on an existing chromosome. This operator is equal to Whitley's mutation operator [5].
- Connectivity mutation, which removes some connections and adds some others [8].
- One-point crossover, shown in figure 1.
- Nodes crossover, which creates a child chromosome by copying repeatedly from a random parent a node with all its incoming connections [7].
- Biased weight mutation, which changes 10 percent of the weights of a parent chromosome within a small range [7].
- Node existence mutation, which either removes a node completely or activates all connections to and from a node [6].

The experiment was performed on a Pentium 90 PC. The evolution process needed between 1.5 and 2 minutes to generate one controller, including the fitness determination. This means that in a period of 24 hours about 900 controllers could be generated. After about 50 hours and the generation of 2000 controllers, the best controller generated had an MSE of 0.025. The evolution was interrupted at that point, and the controller was tested in the control loop. The result was found to be a good bioreactor neurocontroller, which could efficiently direct the bioreactor to the defined setpoints, and keep it there with a constant flow rate. The resulting neurocontroller had a configuration with two hidden layers, with 19 nodes in the first and 10 nodes in the second layer.

This was the result of the first experiment. More experiments were performed. Although the genetic algorithm used in the first experiment was found to be a good choice (small changes in the algorithm almost always harmed the evolution run), the initial neural network configuration was found to be less than optimal. In later experiments, a simple feedforward neural network (without the concept of layers) with a maximum of ten hidden nodes and no TDLs was used as initial neural network configuration. Surprisingly, within 8 hours and the generation of 1500 controllers this configuration could be evolved as a good bioreactor neurocontroller with no more than seven hidden nodes. The genetic algorithm used was the same as for the first experiment, except that the weight initialisation was in a range of [-10,+10] and no treatment of competing conventions was used. The control results are shown in figure 6. An initial neural network configuration with a maximum of five hidden nodes did not work well.

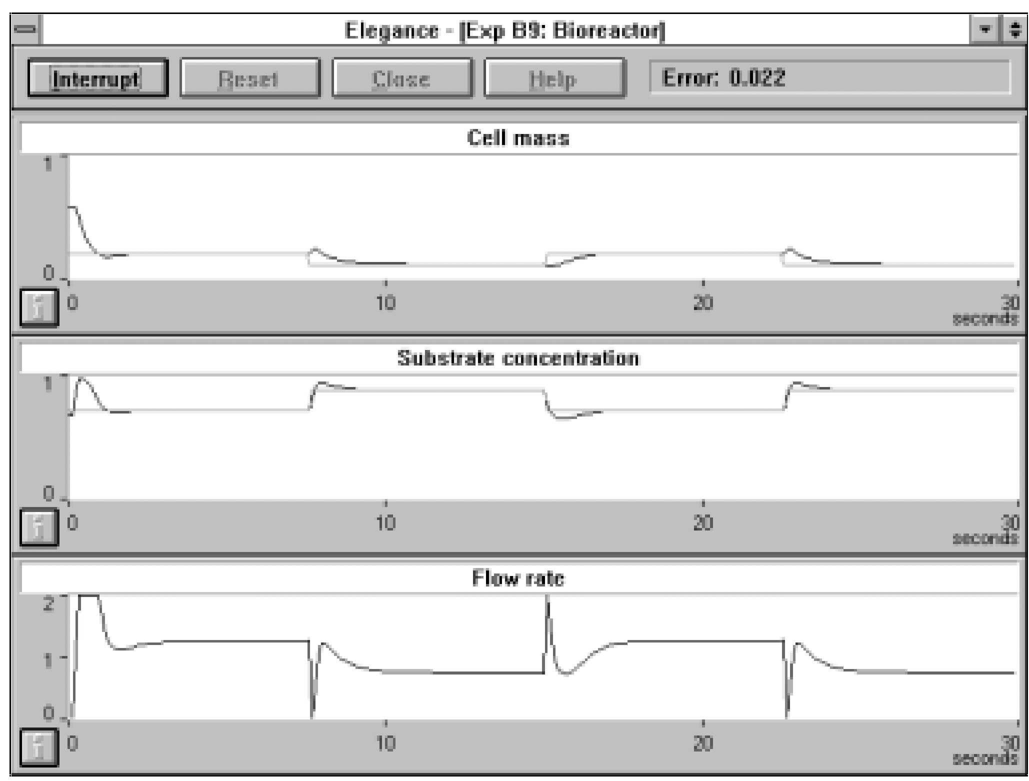


figure 6: A control loop of a bioreactor neurocontroller evolved with Elegance. This controller reaches an MSE of about 0.019. It consists of a feedforward neural network with only seven hidden nodes. In the first two graphs, the 'square' lines indicate the setpoints, while the curves indicate the actual values.

7. Conclusions

Comparing the results of Elegance with the results of the NeuroControl Workbench on a bioreactor simulation, we find that they need similar training times to reach similar (good) results on neural networks with equivalent configurations. This shows that the application of genetic algorithms for neurocontroller design is at least competitive with conventional techniques. Elegance also showed that in some cases the model-based approach might work less well than the genetic reinforcement approach, since a neurocontroller might need a simpler configuration than a neural model. Besides that, there are a number of advantages of using genetic algorithms:

- Genetic algorithms can be used to design neurocontrollers with any neural network configuration, while conventional techniques are often limited to specific configurations, for instance to feedforward networks.
- Genetic algorithms need nothing more than a controller performance evaluation to work. In principle, such an evaluation is always available. Conventional techniques are often dependant on extra information, like the derivative of the error function.
- Genetic algorithms can be used to not only determine the neural network connection weights, but also the architecture. Most conventional techniques are limited to weight determination.

Still, although it has been shown that the technique is viable, it isn't at all clear what

genetic algorithm configuration features lead to an efficient, successful evolution run, particularly in the design of neurocontrollers in a reinforcement situation. Elegance can be used to run a large number of experiments in this respect, and may therefore be a useful tool in this subject of research.

References

[1] HOLLAND, JOHN (1992). *Adaptation in Natural and Artificial Systems*, 2nd edition. MIT Press/Bradford Book Edition, Cambridge, Massachusetts.

[2] GOLDBERG, DAVID E. (1989). *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley Publishing Company, Inc.

[3] JARMULAK, JACEK (1994). *NeuroControl Workbench*. Master thesis, Delft University of Technology, Delft.

[4] JARMULAK, J.; KERCKHOFFS, E.J.H. & ROTHKRANTZ, L. (1995). *Universal Approach to Neural Process Control Illustrated on a Biomass Growth Model*. Published in *Proceedings of the 2nd IFAC/IFIP/EurAgEng Workshop on Artificial Intelligence in Agriculture*, Wageningen, the Netherlands.

[5] WHITLEY, DARRELL; DOMINIC, STEPHEN; DAS, RAJARSHI & ANDERSON, CHARLES W. (1993). *Genetic Reinforcement Learning for Neurocontrol Problems.* Published in *Machine Learning*, Kluwer Academy Publishers, Boston, Volume 13, pp. 103-128.

[6] SPRONCK, PIETER (1996). *Elegance: Genetic Algorithms in Neural Reinforcement Control*. Master thesis, Delft University of Technology, Delft.

[7] MONTANA, DAVID J. & DAVIS, LAWRENCE (1989). *Training Feedforward Neural Networks Using Genetic Algorithms*. Published in *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, San Mateo, California, pp. 762-767.

[8] MANIEZZO, VITTORIO (1993). *Searching among Search Spaces: hastening the genetic evolution of feedforward neural networks*. Published in *Artificial Neural Nets and Genetic Algorithms*, editors R.F. Albrecht, C.R. Reeves & N.C. Steel, Springer-Verlag, Wien, New York, pp. 635-642.

[9] THIERENS, DIRK; SUYKENS, JOHAN; VANDEWALLE, JOOS & DE MOOR, BART (1993). *Genetic Weight Optimization of a Feedforward Neural Network Controller*. Published in *Artificial Neural Nets and Genetic Algorithms*, editors R.F. Albrecht, C.R. Reeves & N.C. Steel, Springer-Verlag, Wien, New York, pp. 658-663.

Both the NeuroControl Workbench and Elegance are freely available for non-commercial use from Delft University of Technology.

Bibliography

ALBA, E., ALDANA, J.F. & TROYA, J.M. (1993). *Genetic Algorithms as Heuristics for Optimizing ANN Design*. Published in *Artificial Neural Nets and Genetic Algorithms*, editors R.F. Albrecht, C.R. Reeves & N.C. Steel, Springer-Verlag, Wien, New York, pp. 683-690. A description of the GRIAL tool, for the testing of genetic techniques.

ALEKSANDER, IGOR & MORTON, HELEN (1990). *An Introduction to Neural Computing*. Chapman & Hall, London. A basic book about the theory of neural networks.

BRAUN, HEINRICH & WEISBROD, JOACHIM (1993). *Evolving Neural Feedforward Networks*. Published in *Artificial Neural Nets and Genetic Algorithms*, editors R.F. Albrecht, C.R. Reeves & N.C. Steel, Springer-Verlag, Wien, New York, pp. 25-32. A description of the ENZO system, for the evolving of ANNs.

CHABOT STADHOUDERS, P.C.W. (1994). *Genetic Algorithms and Neural Networks*. Delft University of Technology, Delft. A graduation thesis on the design of Neural Networks with GAs. It focuses on weight optimisation in small networks.

DASGUPTA, DIPANKAR & MCGREGOR, DOUGLAS (1992). *Designing Application-Specific Neural Networks using the Structured Genetic Algorithm.* Published in *International Workshop on Combinations of Genetic Algorithms and Neural Networks*, editors L. Darrell Whitley & J. David Schaffer, IEEE Computer Society Press, Los Alamitos, California, pp. 87-96. A description of the sGA and of experiments in using it to evolve ANNs.

DAVIS, LAWRENCE (editor) (1991). *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York. This book is meant as a manual for those who want to use GAs without bothering too much about theoretical concepts. Besides a short, clear introduction to GAs, a large part of the book is devoted to application case studies.

DAWKINS, RICHARD (1986). *The Blind Watchmaker*. Penguin Books, London. This sequel to Dawkin's classic *The Selfish Gene* makes clear how evolution can create such

complex organisms as human beings. It also introduces computer-based evolution.

DAWKINS, RICHARD (1989). *The Selfish Gene*. Oxford University Press, Oxford. Second, revised edition (first edition 1976). This book describes the process of biological evolution for a non-specialist audience.

DEWDNEY, A.K. (1990). *The Magic Machine: A Handbook of Computer Sorcery*. W.H. Freeman and Company, New York. A fun collection of computer recreations. Some of the entries are about evolution on a computer.

GOLDBERG, DAVID E. (1989). *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley Publishing Company, Inc. This classic work about GAs is about the best introduction to the subject to be found. A large part of the first two chapters of this text is based on Goldberg's book.

GREFENSTETTE, JOHN J. (1990). *A User's Guide to GENESIS; Version 5.0.* Supplied with Grefenstette's program GENESIS. GENESIS is a GA system written in C, made available to the general public in source code format, to promote the study of GAs.

HAASDIJK, E.W., WALKER, R.F., BARROW, D. & GERRETS, M.C. (1994). *Genetic Algorithms in Business.* Published in *Genetic Algorithms in Optimisation, Simulation and Modelling* (Stender, 1994), pp. 157-184. This article contains some interesting examples of the application of GAs, especially in the field of business modelling.

HANCOCK, PETER J.B. (1992). *Genetic Algorithms and permutation problems: a comparison of recombination operators for neural net structure specification.* Published in *International Workshop on Combinations of Genetic Algorithms and Neural Networks*, editors L. Darrell Whitley & J. David Schaffer, IEEE Computer Society Press, Los Alamitos, California, pp. 108-122. Hancock investigates the competing conventions problem with the evolving of ANN architectures.

HARP, STEVEN A. & SAMAD, TARIQ (1991). *Genetic Synthesis of Neural Network Architecture*, published in *Handbook of Genetic Algorithms* (Davis 1991), pp. 202-221. This article describes an attempt to design a ANN architecture with a GA.

HECHT-NIELSEN, ROBERT (1991). *Neurocomputing*. Corrected edition (first edition 1990). Addison-Wesley Publishing Company, Inc. An extensive overview of neural network theory.

HEITKOETTER, JOERG & BEASLEY, DAVID (1995). *The Hitch-Hiker's Guide to Evolutionary Computation*. This is the excellent but huge FAQ (Frequently Asked Questions list) of the Internet newsgroup comp.ai.genetic. It contains introductory information on many evolutionary techniques, as well as references to books and software.

HOLLAND, JOHN H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Second, revised edition (first edition 1975). MIT Press/Bradford Book

Edition, Cambridge, Massachusetts. This work earned John Holland the title "father of Genetic Algorithms". It's a difficult and rather mathematical approach to evolutionary systems.

JARMULAK, JACEK (1994a). *NeuroControl Workbench*. Delft University of Technology, Delft. An award-winning graduation thesis on neurocontrol. It describes how neural networks can be used in process control, the implementation of a program to do some experiments with several kinds of neurocontrol, and the results of these experiments.

JARMULAK, JACEK (1994b). *NeuroControl Workbench version 1.20 User Manual*. Delft University of Technology, Delft. The user manual of Jarmulak's NeuroControl Workbench. It also contains a description of several types of plants and plant controllers used in this program.

JARMULAK, J., KERCKHOFFS, E.J.H & ROTHKRANTZ, L.J.M. (1995). *Universal Approach to Neural Process Control Illustrated on a Biomass Growth Model*. Published in *Proceedings of the 2nd IFAC/IFIP/EurAgEng Workshop on Artificial Intelligence in Agriculture*, Wageningen, the Netherlands. A description of model-based neurocontrol, NCWB and the training of a bioreactor plant.

JONGSMA, T. (1995). *(Parallel) Genetic Algorithms in Clarity, a visual functional database environment*. Delft University of Technology, Delft. A graduation thesis on GAs and the Clarity system.

KARUNANITHI, NACHIMUTHU, DAS, RAJARSHI & WHITLEY, DARRELL (1992). *Genetic Cascade Learning for Neural Networks.* Published in *International Workshop on Combinations of Genetic Algorithms and Neural Networks*, editors L. Darrell Whitley & J. David Schaffer, IEEE Computer Society Press, Los Alamitos, California, pp. 134-145. A preliminary investigation of the possibility of applying cascade correlation learning with the evolving of ANNs.

KINNEAR, KENNETH E. (editor) (1994). *Advances in Genetic Programming*. MIT Press/Bradford Book Edition, Cambridge, Massachusetts. A Collection of papers about genetic programming. One-half of the book describes innovative applications of GP, while the other half concentrates on papers about increasing the power of GP.

KOZA, JOHN R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press/Bradford Books Edition, Cambridge, Massachusetts. This is the classic introduction to one of the most promising subfields of GAs, genetic programming, and also contains a very good introduction to GAs themselves.

MANIEZZO, VITTORIO (1993). *Searching among Search Spaces: hastening the genetic evolution of feedforward neural networks*. Published in *Artificial Neural Nets and Genetic Algorithms*, editors R.F. Albrecht, C.R. Reeves & N.C. Steel, Springer-Verlag, Wien, New York, pp. 635-642. A description of the ANNA ELEONORA system, for the optimisation of both architecture and connection weights of ANNs.

MICHALEWICZ, ZBIGNIEW (1994). *Genetic Algorithms + Data Structures = Evolution Programs*. Second, extended edition (first edition 1992). Springer-Verlag, Berlin. A collection of articles, which present theoretical and practical views on many different areas in the field of GAs.

MONTANA, DAVID J. & DAVIS, LAWRENCE (1989). *Training Feedforward Neural Networks Using Genetic Algorithms.* Published in *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, San Mateo, California, pp. 762-767. Montana and Davis have devised a whole range of genetic operators for use in the optimisation of ANNs.

PEARSON, D.W., STEEL, N.C. & ALBRECHTS, R.E. (editors) (1995). *Artificial Neural Nets and Genetic Algorithms.* Springer-Verlag, Wien, New York. The Proceedings of the International Conference in Alès, France. This book contains a lot of articles about GAs, and a lot of articles about ANNs, but very few about the combination of these techniques.

RADCLIFFE, NICHOLAS J. & SURRY, PATRICK D. (1994). *The Reproductive Plan Language RPL2: Motivation, Architecture and Applications.* Published in *Genetic Algorithms in Optimisation, Simulation and Modelling* (Stender, 1994), pp. 65-94. This article contains some interesting examples of the application of GAs.

SCHAFFER, J. DAVID, WHITLEY, DARRELL & ESHELMAN, LARRY J. (1992). *Combinations of Genetic Algorithms and Neural Networks: A Survey of the State of the Art.* Published in *International Workshop on Combinations of Genetic Algorithms and Neural Networks*, editors L. Darrell Whitley & J. David Schaffer, IEEE Computer Society Press, Los Alamitos, California, pp. 1-37. This is a good introduction to the subject of combining the strengths of GAs and ANNs.

STENDER, J., HILLEBRAND, E. & KINGDON, J. (1994). *Genetic Algorithms in Optimisation, Simulation and Modelling.* IOS Press, Amsterdam. This collection of papers focuses on the development of Parallel GAs using the GAME system.

SYRJAKOW, MICHAEL & SZCZERBICKA, HELENA (1995). *Simulation and Optimization of Complex Technical Systems.* Published in the *Proceedings of the SCS'95*, editors Tuncer L. Ören & Louis G. Birta, pp. 86-95. This paper describes a theoretical and practical implementation of a hybrid system.

THIERENS, DIRK, SUYKENS, JOHAN, VANDEWALLE, JOOS & DE MOOR, BART (1993). *Genetic Weight Optimization of a Feedforward Neural Network Controller*. Published in *Artificial Neural Nets and Genetic Algorithms*, editors R.F. Albrecht, C.R. Reeves & N.C. Steel, Springer-Verlag, Wien, New York, pp. 658-663. This article discusses weight optimisation in an ANN, concentrating on the problem of competing conventions.

VERBRUGGEN, H.B., KRIJGSMAN, A.J. & BRUIJN, P.M. (1992). *Towards Intelligent control: Integration of AI in Control*. Published in *Application of Artificial Intelligence in Process Control,* editors L. Boullart, A. Krijgsman & R.A. Vingerhoeds, Pergamon

Press, Oxford, England. This article discusses several techniques of using AI in process control.

WALDROP, M. MITCHELL (1992). *Complexity: The Emerging Science at the Edge of Order and Chaos*. Penguin Books, London. A biographical story of the group of scientists who studied complexity and founded the Santa Fe Institute. John Holland was one of the most influential members of this group.

WHITLEY, D., STARKWEATHER, T. & BOGART, C. (1990). *Genetic algorithms and neural networks: optimizing connections and connectivity.* Published in *Parallel Computing*, Elsevier Science Publishers B.V., North-Holland, Volume 14, pp. 347-361. The GENITOR algorithm is introduced, and used to design the architectures of ANNs and to optimise the connection weights.

WHITLEY, DARRELL, DOMINIC, STEPHEN, DAS, RAJARSHI & ANDERSON, CHARLES W. (1993). *Genetic Reinforcement Learning for Neurocontrol Problems.* Published in *Machine Learning*, Kluwer Academy Publishers, Boston, Volume 13, pp. 103-128. A preliminary study in the application of GAs to evolve ANNs for neurocontrol problems.

WHITLEY, D. (1995). *Genetic Algorithms and Neural Networks.* Published in *Genetic Algorithms in Engineering and Computer Science*, editors J. Periaux & G. Winter, John Wiley & Sons Ltd. A preliminary study in the application of GAs to evolve ANNs for neurocontrol problems.

YAO, XIN (1995). *Evolutionary Artificial Neural Networks*. Published in *Encyclopedia of Computer Science and Technology*, editors A. Kent *et al.*, Vol. 33, pp. 137-170, Marcel Dekker Inc., New York. A fairly complete overview of the possibilities of optimising ANNs with GAs. The text concentrates on weight optimisation, network topology design, and learning rule development.

Index

see wild card symbol

* *see* don't care symbol

accuracy 57
activation function 72; 74; 76; 113; 117; 119; 121; 139; 146; 152; 157; 158; 203
  odd 77
adaptive mutation 102; 124–25; 135; 138; 139; 140; 141; 145; 161; 201
adaptive system 16
AI *see* artificial intelligence
Alba, E. 79; 84
Aleksander, I. 72; 74; 174
algorithm oriented tool 61; 62
allele 21; 36; 56; 122
  active 124
  biological 17; 18; 19
alphabet 23; 31; 42; 51
  schema 31
ANN *see* artificial neural network
ANNA ELEONORA 84; 86–88; 110; 118; 179
application oriented tool 61
appointment of credit 48; 49
architecture 72; 74; 75; 76; 77; 100; 105; 113; 117; 118; 125; 191
  design 59–60; 75; 151; 161
  encoding 81
  evolution 76; 80; 82; 86; 90; 100; 110; 201; 205
arctangent 117; 152; 203
Aristotle 69
artificial intelligence 10; 71; 155; 169; 195
  plant control 95–96
artificial neural network 48; 51; 57; 59; 119; 194
  analysis 76
  architecture *see* architecture
  biological basis 71
  completely connected 73; 75
  control *see* neurocontrol
  controller *see* neurocontroller
  design 105
  encoding *see* encoding
  evolution 77; 80–86; 102
  evolution pseudo code 85
  feedback *see* recurrent network
  feedforward *see* feedforward network
  inverse model 97
  large 90; 179
  layered feedforward *see* layered feedforward network
  model 98
  recurrent *see* recurrent network
  size 110; 128; 158; 159; 201
  theory 71–74
  training *see* training
auction 50
autosave 133; 180

backpropagation 60; 74; 83; 86; 87; 89; 100; 133; 151; 166; 197
bang-bang pole balancing system 114; 150; 161; 163; 179; 185–86
  evolution 138–45
BGA *see* breeder genetic algorithm
bias 72; 77; 82
biased nodes mutation 121
biased weight mutation 120; 138; 140; 144; 147; 153; 159; 177; 204
biased-mutate-weights 88; 89; 120
binary mutation 120; 191
  source code 190
binary weight mutation 120; 143
biological evolution 15
bioreactor 114; 137; 161; 162; 163; 179; 186; 194; 202; 203; 204; 205
  evolution 151–60
  identifier 160
blindness 30; 38
BMW *see* biased-mutate-weights
Borland 128; 135
Braun, H. 80
breeder genetic algorithm 38; 62
British Gas 55
bucket brigade 50
building block 33; 41
business modelling 57–59

cart centering system *see* pole balancing system
cart positioning system *see* trolley
cascade correlation 79
cause-and-effect 93
cellular model 40
CF *see* crossover-features
Chabot Stadhouders, P.C.W. 162
chart 113; 118; 177
CheckParms 188; 191
child 189; 196
chromosome 20; 38; 56; 58; 78; 82; 111; 113; 118–19; 120; 121; 122; 124; 125; 132; 135; 139; 195
  array 189
  binary 23
  biological 15; 17–18
  coding 41; 191
  design 105
  diploid 38
  haploid 38
  length 20; 22; 23; 24; 53; 81; 86; 124; 128
  non-legal 41
  structure 110; 201
chromosome C 17–18
classifier 48; 49
  fitness 50
classifier store 48; 49
classifier system 48–51; 63
clearinghouse 50
closed-loop control 94; 96
CN *see* crossover-nodes
coarse-grained model 39
CoC *see* coefficient of concordance
coding *see* encoding
coefficient of concordance 58
competing conventions 74; 77–80; 84; 86; 88; 89; 90; 103; 105; 106; 110; 125; 134; 135; 138; 139; 162; 163; 198; 201; 204
  solution 78–80; 198
condition 48
connection *see* architecture
connection weight *see* weight

connectivity 80; 82; 86
connectivity bits 119; 121; 122; 124
connectivity mutation 83; 87; 121; 143; 144; 147; 153; 159; 177; 204
constant wave 117
control
  closed-loop *see* closed-loop control
  dead-beat *see* dead-beat control
  direct-inverse model *see* direct-inverse model control
  forward-model inverse *see* forward-model inverse control
  model-based 101; 104
  model-based predictive *see* model-based predictive control
  open-loop *see* open-loop control
  reinforcement *see* reinforcement control
control loop 111; 114; 117; 124; 126; 127; 132; 134; 145; 150; 154; 155; 200; 202
  display 113; 118; 130; 134; 179; 201
  run length 127; 135
controller 94; 95; 96; 111; 113; 114; 115–17; 124; 127; 130; 132; 134; 138; 151; 174; 176; 178; 182; 184; 186; 189; 196; 197; 200; 201
  adding 191
  computer 95; 196
  conventional 96
  fitness 111
  human 94; 96; 104; 194
  mechanical 94; 104; 194
  performance 111; 179
  PID *see* PID controller
  winning 124
conventional search algorithm 30; 43; 54; 65; 66
convergence 26; 29; 33; 36; 47; 77; 82; 89; 101; 102; 125; 138; 141; 143; 144; 148; 153; 157; 178
  chart *see* convergence chart
  graph 131
  premature 34
convergence chart 178
Create 187; 189
CreateCopy 189
crossover 21–22; 25; 33; 39; 52; 53; 59; 78; 79; 80; 81; 83; 86; 90; 103; 144; 157; 161; 198; 201
  after 125
  and learning rate 157
  cycle *see* cycle crossover
  genetic programming 52
  half-uniform *see* half-uniform crossover
  Montana and Davis 89
  multiple-parents 39
  multiple-point 39
  nodes *see* nodes crossover
  one-point *see* one-point crossover
  only 141
  order *see* order crossover
  partially matched *see* partially matched crossover
  probability 24; 48; 82; 84; 88; 102; 119; 124
  syntax-preserving 53
  two-point *see* two-point crossover
  uniform *see* uniform crossover
crossover point 21; 25; 31; 37
crossover-features 89
crossover-nodes 89; 121
crossover-weights 89
cross-wired model 58
crowding 35; 47; 48; 50; 62; 63; 83; 126; 147; 201; 204
crowding factor 35
CS *see* classifier system
CW *see* crossover-weights
CX *see* cycle crossover
cycle crossover 37

Darwin, C. 16
Dasgupta, D. 82
data mining 60
data structure 51
database 57; 58
Davis, L. 60; 61; 62; 84; 85; 88; 89; 120; 121; 126; 201
Dawkins, R. 16
dead-beat control 98; 101; 181; 182
decoding 82; 84; 85
defining length 32
Delphi 128; 135
deme 41
Dennett, D.C. 13; 107
detector 48
Dewdney, A.K. 9
DIDO *see* double-input double-output
DIDO system 114; 181–82
diffusion model 40
direct encoding 81
direct intelligent control 95
direct-inverse model control 97
direct-mail campaign 57
discrete 181; 182
display 177 *see* control loop (display)
divergence from setpoint 158
diversity 22; 36; 78; 79; 102; 138; 139; 144; 161
DNA *see* chromosome (biological)
DoCycle 187
dominance 38
don't care symbol 31
double-input double-output 181–82
duplicate checking 125; 135; 203

Edinburgh Parallel Computer Centre 56
effector 48; 50
Elegance 165; 169; 205; 206
  a simple experiment 175–79
  acronym 200
  chromosome 118–19
  control loop display *see* control loop (display)
  controller 115–17
  design 109–36; 165; 201
  encoding 122–23
  enhancement 111
  evaluation 132–33
  experiment 132; 137–64
  extending 187–92
  fitness 123–24
  flexibility 109; 110; 134–35; 201
  functional design 111–13
  functions 113
  future 134
  genetic algorithm 120; 122–27
  genetic operator 120–21; 124–25
  hardware 128
  implementation 127–29
  initialisation 126
  installation 175
  interface 129
  limits 110–11; 201
  maintainability 110
  menu bar 129; 130; 175
  on-line help 132; 174; 175; 179
  parameter 125–26
  plant 114–15; 181–86
  preferences 180
  project 113–14
  purpose 109–11; 201
  reinforcement 126–27
  replacement policy 111
  requirements 110; 174
  setpoint generator 117
  software 128–29
  source code 115; 121
  speed 110; 127; 134
  stability 133
  technical design 111; 113–27
  usefulness 133
  user manual 132; 174–80
  user-friendliness 110; 127; 133
  using 129–32
  version 180
elitism 35; 47; 48; 56; 59; 62; 83; 125; 203
encoding 80–82; 83; 90; 105; 109; 113; 121; 122; 135; 177; 198
  binary 81; 86; 105; 106; 119; 122; 123; 138; 144; 191; 198; 201
  direct 80; 105; 110; 201
  fixed length *see* fixed length
  high-level *see* encoding (indirect)
  indirect 80; 105
  low-level *see* encoding (direct)
  parameter 120; 122–23
  real values 81; 83; 88; 103; 105; 106; 122; 138; 139; 144; 177; 191; 198; 201; 203
  strong *see* encoding (direct)
  variable length *see* variable length
  weak *see* encoding (indirect)
ENZO 80
EP *see* evolutionary programming
EPCC *see* Edinburgh Parallel Computer Centre
epistasis 41
error 74; 75; 84; 95; 100; 102; 115; 123; 150; 199
error proportional 117
error space 74
ESPRIT III 58; 62
Euler 185
evolution 103; 113; 122; 131; 147; 150; 178; 196; 198
  biological *see* biological evolution
  elegance 166
  run 111; 126; 128; 131; 153;

155; 159; 163; 178; 202
evolutionary programming 83
evolutionary system 10; 58; 71; 102
Execute 189
expected value selection 35; 47; 48
experiment 109
expert system 71; 96; 104

feedforward network 73; 74; 81; 100; 116; 125; 139; 146; 147; 148; 157; 158; 162; 176; 201; 204; 205
filename 113
fine-grained model 40
first generation 24–25; 126
fitness 18; 21; 26; 30; 41; 42; 56; 59; 75; 77; 81; 99; 101; 102; 104; 105; 113; 135; 154; 177; 195
  average 19; 21
  chart *see* fitness chart
  conversion 138; 143 *see* scaling
  criterion 38
  determination 82–83; 84; 88; 101; 103; 120; 126; 127; 151; 153; 177; 178; 203
  development 29–30
  genetic programming 53
  graph 131
  mean square error 123–24; 126; 145; 146; 161; 162; 199; 201; 203
  MSE-based *see* fitness (mean square error)
  parameter 120; 123–24
  raw *see* raw fitness
  recalculation 151
  re-evaluation 126
  scaled *see* scaled fitness
  scaling *see* scaling
  schema *see* schema (fitness)
  time-until-failure 123; 126; 179; 199
fitness case 53
fitness chart 178
fitness determination 199; 200
  precision 151; 157
fitness function 21; 24; 41; 58; 66; 101; 109; 110
  genetic programming 53
fixed length 53; 59; 81
forward model *see* identifier
forward-model inverse control 97
framework 76
function 51
function optimisation 45–48; 63
fuzzy logic 194

GA *see* genetic algorithm
GA Workbench 61
GAAF 62
GA-deceptive 33
GA-hard 33
GAME 62
GA-simplex 84; 86; 87–88; 121; 143; 201
gaspipe network 55–56
GBML *see* genetics-based machine learning
gene 21; 56; 82
  biological 15; 17–18
generation 21
  definition 125
  first *see* first generation
  maximum 125
  next *see* next generation
generation gap 35; 47
GENESIS 62
genetic algorithm 113; 115; 116; 120; 130; 134; 147; 158; 169; 174; 177; 178; 189; 191; 200; 201; 203; 205
  alternative training method 74
  and artificial neural networks 71–92; 105; 198
  application 45–64; 63
  artificial neural network evolution *see* artificial neural network (evolution)
  basics 16–22
  brute force 166
  characteristics 30; 196
  code 191
  definition 16; 42; 65; 174; 194; 195
  design 41–42; 67
  Elegance 122–27
  evolutionary system 10; 71
  goal 30; 66
  implementation 110
  in biology 60
  in classifier system 48; 50
  in computer science 60
  in engineering 60
  in game-playing 60
  in mathematics 61
  in modelling 60
  in sensory processing 61
  model design 58
  neurocontrol 93–104
  parallel *see* parallel genetic algorithm
  parameter *see* parameter
  personal view 65–68; 106; 165–68
  plant control 96
  problem characteristics 66
  pseudo code 20
  reinforcement control 99; 101; 109; 197
  result 67
  robustness *see* robustness
  theory 15–44
  tool 61–62
  weakness 30; 66; 196
genetic cascade learning 79
genetic drift 102
genetic hillclimbing 79
genetic operator 16; 21; 42; 67; 85; 90; 102; 105; 109; 111; 113; 120–21; 122; 135; 137; 143; 151; 177; 191; 195; 200; 204
  adding 110; 121; 189–91; 201
  artificial neural network evolution 83–84
  basics 21–22
  biological 18
  choice 42
  code 189
  creation 189
  crossover *see* crossover
  factor 120
  knowledge augmented *see* knowledge augmented operator
  Montana and Davis 88–89
  mutation *see* mutation
  parameter 120; 124–25
  random value 120
  randomisation *see* randomisation
  reordering *see* reordering operator
  reproduction *see* reproduction
  selection 25; 26
  syntax-preserving 52
  type 189
genetic programming 51–54; 63; 80
genetic reinforcement control 11; 101–2; 104; 106; 161; 163; 165; 166; 169
  and model-based control 157; 160; 161; 166
GeneticOperatorType 189
genetics *see* natural genetics
genetics-based machine learning 48
GENITOR 102–3; 106; 110; 114; 119; 138; 139; 140; 143; 144; 161; 162; 179; 186
  commentary 139
  testing 140–42
GENITOR II 102
genome *see* genotype
genotype 20
  biological 17; 19
global minimum *see* global optimum
global optimum 74; 75; 89; 196
Goldberg, D.E. 22; 26; 32; 45; 60; 61; 67; 174; 196
GP *see* genetic programming
graded learning *see* reinforcement learning
gradient descent 74; 75
granularity 82; 86; 88
granularity mutation 87; 121; 143; 201
Gray code 62
GRC *see* genetic reinforcement control
Grefenstette, J. 62

Haasdijk, E.W. 57; 69
half-uniform crossover 39; 121; 147
Hamming distance 79
Hancock, P.J.B. 79
Harp, S.A. 59
Hecht-Nielsen, R. 101
Hein, P. 107
Heitkoetter, J. 83
hidden layer 60; 203
hidden layer redundancy 77; 79
hidden node 59; 77; 139; 144; 161; 174
  adding 79
  functionally equivalent 79
  maximum 117
  rearrangement 79; 125
hidden node redundancy 77; 79
hillclimb 89
hillclimbing 79
  random 79
history 97; 111; 200; 201
Holland, J. 16; 31; 32; 39; 63; 195
Hughes, M. 61
HUX *see* half-uniform crossover
hybrid system 38; 54; 63; 86

identifier 97; 98; 99; 152; 197; 203

incest prevention 103; 125; 135; 204
indirect encoding 80
indirect intelligent control 95–96
individual 20
  biological 17
inferior solution 66
initial plant state 126; 135
initialisation 177
  parameter 120; 126
input node 59; 125; 203
input selection 76
integral windup protection 95; 115
intelligent supervisor 96
inverse exponential 88; 120; 126; 191
inverse model 97; 99; 100; 197
  training 97
inversion 36; 42; 62
inverted pendulum *see* pole balancing system
island model 39; 102
isomorphism *see* competing conventions

Jarmulak, J. 95; 97; 111; 114; 115; 117; 127; 134; 152; 158; 160; 161; 181; 182; 183; 185; 186; 197; 201; 203
Jong, K.A. de 45; 47; 48; 63
  test function 46–47

Karunanithi, N. 79
Kinnear, K.E. 51; 60; 61
KiQ Limited 62
knowledge augmented operator 38
Koza, J.R. 51; 54; 58; 63
Krijgsman, A.J. 181; 182

layer 72; 80; 117; 152
  hidden 72; 75; 88
  input 72
  output 72
layered feedforward network 73; 116; 117; 125; 152; 157; 198; 201
learning rate 72; 76; 82; 117; 119; 121; 126; 139; 146; 186
learning rate mutation 121; 156; 157; 161; 163; 201
learning rule 76; 77
  evolution 76
learning task 76
Legalize 189
line 118; 177
linear normalisation 38
linear programming 87
linear scaling 34
linear system 95; 197
LISP 54; 96; 163
local minimum *see* local optimum
local optimisation 86; 87; 110; 134; 151; 157; 201
local optimum 74; 89; 196
locus 21; 36; 37
  biological 17; 18

machine learning 48; 71
Maniezzo, V. 84; 86; 87; 110; 118; 120; 121; 142; 143; 201
mapping 73; 74; 77; 90; 96
  functional 80
  structural 80

McCulloch, W. 72
McGregor, D. 82
mean square error 83; 88; 123; 151; 177; 199
Mendel, G. 15
message 48
message list 48; 49
Michalewicz, Z. 61
Microsoft 127; 129; 174; 202
migration 39
MIMD *see* multiple-instruction multiple-data
MN *see* mutate-nodes
modality 45
model 57; 97–98; 99; 100; 111; 203
  absence 98; 99
  cross-wired *see* cross-wired model
  design 100–101
  forward *see* identifier
  inverse *see* inverse model
model-based predictive control 97; 197
Montana, D.J. 84; 85; 88; 89; 120; 121; 126; 201
Monte Carlo technique 54
Morton, H. 74; 174
MS-DOS 61; 62; 127
MSE *see* mean square error
multicriteria optimisation 38
multimodal 40; 41; 48; 74
multiobjective optimisation 38
multiparameter optimisation 38; 42
multiple-instruction multiple-data 40
mutate-nodes 88; 89
mutate-weakest-nodes 88; 89
mutation 22; 25; 53; 59; 79; 83; 86; 87; 103; 122; 125; 144; 161; 196; 201
  biased nodes *see* biased nodes mutation
  biased weight *see* biased weight mutation
  binary *see* binary mutation
  binary weight *see* binary weight mutation
  biological 22
  connectivity *see* connectivity mutation
  granularity *see* granularity mutation
  Montana and Davis 88–89
  node existence *see* unbiased node existence mutation
  only 141
  rate *see* mutation rate
  unbiased nodes *see* unbiased nodes mutation
  unbiased weight *see* unbiased weight mutation *see* unbiased weight mutation
  weight 83; 87; 121; 201
mutation rate 24; 47; 79; 83; 102; 103; 105; 138; 139
mutation-driven 103; 106
MWN *see* mutate-weakest-nodes

natural genetics 15; 16–18; 174
natural selection 16; 174
NCWB *see* neurocontrol workbench
neurocontrol 96; 98; 99; 102; 103; 104; 106
  theory 96–99
NeuroControl Workbench 97; 111; 114; 115; 117; 127; 134; 181; 182; 183; 185; 186; 197; 201; 205
neurocontroller 105; 110; 113; 115; 117; 118; 119; 125; 149; 174; 176; 205
  adding 191
  design 11; 105; 160; 166; 169; 174; 200
  evolution 133; 134; 138; 144; 163
  goal 98
NeuroGENESYS 59; 60
neuron 71
  artificial 72
Newton, I. 13
next generation 25–29
niching 40
NN *see* artificial neural network
node 72; 88; 125
  clamped 116
  hidden *see* hidden node
  input *see* input node
  output *see* output node
node existence mutation 121; 147; 153; 159; 204
node mutation 121
nodes crossover 121; 144; 147; 153; 157; 159; 204
nonlinear system 95; 181; 194; 202
normalisation 24
  linear *see* linear normalisation

object oriented 112; 128; 165
occurrence 32
off-line performance 47; 48
OMEGA 58; 62
one-point crossover 121; 138; 140; 143; 144; 153; 159; 177; 204
on-line performance 47; 48
on-line training 100
OOGA 62
open-loop control 94
optimisation problem 65; 195; 197
order 32; 41
order crossover 37
output node 59; 125; 203
OX *see* order crossover

Palmiter's Protozoa 9–10
Palmiter, M. 9
panmictic model 39; 56
PAPAGENA 58; 62
parallel 30; 39; 62; 102; 134; 196
parallel genetic algorithm 39–41; 56; 62
parameter 23–24; 66; 84; 109; 120; 125–26; 177
parent 21; 84; 121; 126; 189; 195; 201
partially matched crossover 37
payoff 50
Pearson, D.W. 90
penalty 41; 56; 81; 83
  complexity 124; 143; 201
  premature failure 124; 127; 200; 201
performance
  off-line *see* off-line performance
  on-line *see* on-line performance

permutation 37
permutation problem *see* competing conventions
PGA *see* parallel genetic algorithm
phenotype 21
biological 17; 18
PID *see* proportional integrating differentiating
PID controller 94–95; 104; 110; 115; 117; 118; 134; 201
adding 191
Pitts, W. 72
plant 93; 95; 97; 100; 110; 111; 112; 114–15; 127; 130; 133; 134; 174; 175; 179; 189; 196; 200; 201
adding 110; 165; 187–89; 201
code 187
creation 187
failing 114; 115; 124; 127; 188; 199
identifier *see* identifier
implementation 114
in Elegance 181–86
input 93; 96; 97; 98
internal state 93; 98
model 97–98
output 93; 94; 96; 97; 98; 100; 101; 113; 117; 200
test 114
type 187; 201
plant control 93–95–96; 101; 104; 105
plant controller *see* controller
plant model 105
plant output 95
PlantType 187
PMX *see* partially matched crossover
pole balancing system 103; 114; 147; 162; 184–85; 199; 202
bang-bang *see* bang-bang pole balancing system
evolution 138–45
population 20; 39; 111; 131; 195; 200
biological 17
initialisation 140; 154; 178
of programs 51
size 20; 22; 23; 42; 79; 84; 88; 113; 125; 128; 138; 174; 179; 203
small 103
posting a message 48; 49
predictive model 62
preselection 36
principle of meaningful building blocks 41
principle of minimal alphabets 42
process *see* plant
program 51
project 112; 113–14; 130; 131; 175; 177; 201
projection specification field 59
PROLOG 96
proportional integrating differentiating 95; 119
PSF *see* projection specification field

R^3 *see* random respectful recombination
Radcliffe, N.J. 55
random respectful recombination 39
random wave 117; 163; 203
fixed interval 117; 146; 150; 176
randomisation 147; 153; 159; 161; 204
randomness 100
ranking 38; 62; 102; 124; 127; 138; 162; 177; 201
raw fitness 34; 123; 124
recurrent network 73; 75; 100; 116; 125; 162; 201
reinforcement 124; 156; 177
parameter 113; 120; 126–27; 135
reinforcement control 99; 100; 101–2; 104; 106; 109; 134; 174; 194; 197; 200
reinforcement learning 101; 106; 146; 197
law 101
REMO 54
reordering operator 36; 37; 42; 84
replacement 35–36; 83
replacement policy 126; 201
reproduction 21; 22; 25; 33; 53; 59; 79; 83; 86
probability 24
reproductive plan 47
Reproductive Plan Language 2 56
restrictive mating 79
robustness 30; 57
RPL2 *see* Reproductive Plan Language 2
rule and message system 48
Runge-Kutta 4 183; 186

Samad, T. 59
scaled fitness 34; 124
scaling 34–35; 42; 62; 124; 201
linear *see* linear scaling
Schaffer, J.D. 79
schema 31–32; 41
defining length *see* defining length
disruption 31; 36
fitness 31
occurrence *see* occurrence
order *see* order
Schema Theorem 32; 42; 43
selection 42; 143
child 125; 138
elitist *see* elitism
expected value *see* expected value selection
for replacement 35; 102; 125; 126; 138
genetic operator *see* genetic operator (selection)
parent 25; 26; 35; 87; 102; 196
roulette wheel 25; 35
setpoint 113; 115; 123; 146; 151; 152; 155; 176; 186; 191; 196; 202; 203
factor 115
setpoint function 161
setpoint generator 111; 113; 117; 127; 130; 134; 149; 176; 200; 201
sexual reproduction 21
SGA *see* simple genetic algorithm
sigmoid 72; 114; 117; 139; 146; 157; 158; 186
signal 95; 113; 114; 117; 176; 183; 200
adding 191
electrical 182
SIMD *see* single-instruction multiple-data
simple genetic algorithm 22–30; 43; 47; 62; 84; 120; 121; 142
simulated annealing 54; 86
single-input single-output 181
single-instruction multiple-data 41
sinusoid wave 117
SISO *see* single-input single-output
SISO system 114; 181; 189
source code 188
software environment 106; 195
solution space 21; 24; 33; 43; 54; 55; 63; 66; 75; 77; 139; 155; 157; 166; 198; 203
square wave 117; 146; 149; 150; 152
statistical method 57; 58
Stender, J. 60
stepping-stone model 40
strength 49
structural representation 77
structural/functional mapping problem *see* competing conventions
structured genetic algorithm 82
supervised learning 101
Surrey, P.D. 55
survival of the fittest *see* natural selection
survival value 18
syntax 52
Syrjakow, M. 54
Szczerbicka, H. 54

tapped-delay line 97; 100; 110; 112; 114; 139; 152; 157; 158; 200; 203; 204
target 126
mean square error 123; 153
timesteps 123
taxing system 50
TController 191
TDL *see* tapped-delay line
temporary file 175
terminal 51
termination criterion 21; 42
test run
length 101
TGeneticOperator 189
thermostat 93–94
Thierens, D. 77; 79; 125; 162; 163; 201
timestep 103; 113; 123; 127; 139; 146; 152; 156; 158; 181; 182; 199
time-until-failure 123
titration 114; 182
topology *see* architecture
total error failure 126
TPlant 187
training 59; 71; 73; 74; 75; 97; 101; 105
neurocontroller 98–99
off-line 98
on-line 98
supervised 74
training set 83; 84; 88; 98; 99; 100; 101; 105; 146; 197
transfer function *see* activation

function
travelling salesman 37; 166
tree-like structure 52; 53; 58; 63
trolley 114; 152; 161; 163; 176; 182–83; 199; 202
evolution 145–51
trolley 2-dimensional 114; 183–84
TSignal 191
TU Delft 180
two-point crossover 121

UMW *see* unbiased-mutate-weights
unbiased nodes mutation 121
unbiased weight mutation 121
unbiased-mutate-weights 88; 89; 121
uniform crossover 39; 121
unimodal 22
Unix 62
Using Elegance 129–32
UX *see* uniform crossover

variable length 53; 81; 105; 191
Verbruggen, H.B. 95
viability checking 125; 135; 203

weight 72; 74; 75; 76; 77; 79; 82; 105; 113; 117; 118; 119; 191
binary encoding 122–23
determination *see* weight (evolution)
encoding 81; 122
evolution 75; 76; 80; 82; 86; 88; 90; 102; 110; 201; 205
incoming 77; 119
initial 75; 88; 102
length 119; 121; 122
maximum 116; 126
minimum 116; 126
mutation *see* mutation (weight)
outgoing 77
range 80
Whitehead, A.N. 107
Whitley, D. 83; 102; 103; 106; 110; 114; 119; 124; 138; 139; 140; 141; 161; 186; 199; 201; 204
wild card symbol 48
windowing 38
Windows 127; 128; 129; 133; 135; 137; 174; 175; 180; 202
qualities 127

Yao, X. 75; 76