Parallel Genetic Algorithms

Master's Thesis
Laurens Jan Pit
30/08/95
Department of Computer Science
Leiden University

Preface

Can.

Leiden, 30/08/95

Laurens Jan Pit

Abstract

In 1992 Boers and Kuiper designed a method that automatically searches for optimal modular artificial neural network architectures. For this they used modular techniques based on biological systems. A genetic algorithm imitating evolution is used to find recipes which are used by an L-system which imitates natural growth. From the recipes a modular artificial neural network is constructed which imitates the human brain on a very small scale. Preliminary results indicated that modular networks can be found that outperform standard solutions. One drawback is the large amount of computing power needed to evaluate each network architecture.

The goal of this research was to implement the algorithms on a parallel computer (CM-5) in order to allow for larger simulations to investigate the real potential of the methods proposed. The main theme is therefore parallel genetic algorithms imitating evolution on a number of separate islands which once in a while exchange individuals.

Contents

Chapter 1

Introduction

Everything around us is part of some system. The goal of research is always to understand and describe how a system works. One of the most fascinating systems to study is the brain-system. It is however also one of the most incomprehensible ones, with its parallel activity of a myriad of neurons.

With the introduction of computers researchers have tried to model the system ‘brain’ (or simple brain-like functions) into the system ‘computer’. However, this modelling has not been very successful in the fifties and sixties because the models were not complex enough to solve interesting problems and also because the computers were not powerful enough. The seventies and eighties solved this to some extent. The models could achieve solutions for simple problem areas, but more complex problems couldn’t be solved within a feasible period of time. Thus the models were not practical.

This thesis is aimed to solve some practical problems. The brain will be modelled on a small scale by artificial neural networks which can be characterized as being heterogeneous, modular and asynchronous. The design of these networks is a complicated matter for which no standard rules exist. In 1992 Boers and Kuiper developed a method that leaves the design of modular neural networks to the computer [4]. To this end they also used two systems that are, like artificial neural networks, models of systems from nature. They are a genetic algorithm for the evolutionary search towards better neural networks, and an L-system for the growth and development of network architectures. The evolutionary search for architectures of modular neural networks is a very time-consuming process and is during this research implemented using parallel computing.

I support Minsky and Papert’s view stated in their epilogue [40]: “We maintain that the scientific future of neural networks is tied not to the search for some single, universal scheme to solve all problems at once, but to the evolution of a many-faceted technology of brain design. It encompasses good technical theories about the analysis of learning procedures, of useful architectures and of organizational principles to use when assembling these components into larger systems.”

1.1 Problem Solving

In principle the brute force method, i.e. generate and test every possibility, can be used to solve every problem of which a representation of the search space is known and where the solution can be recognized. However, this is often hardly practical when solving real-world problems. Even with the most powerful computers it can take ages to seek all possible solutions. Even with a seemingly simple task to build a house from ten wooden blocks there are more possibilities than an individual could ever try in its whole life.

An improvement to this blind search by generating and testing is when a method can be used with which progression can be signalled. For instance, we could find the top of a hill by following the path of someone who is climbing a hill in the dark — every time, with each step, to climb as steep as possible.

For simple problems this is a useful tactic, but when the problem gets more complicated it could be as difficult to recognize progression as it is to solve the actual problem. The hill-climber will end up on a hill, but without an overview probably on one that is not the real top in the area. There is no full-proof method to prevent this.

The most powerful method we could use to solve complex problems is one where the problem is split into several simpler problems, which can be solved independently from each other. Much of the research in the Artificial Intelligence area is focused on this subject of splitting problems into smaller subproblems, which in turn also could be split into even smaller subsubproblems. Thus we seek for *modularity.*

In some parts of the Artificial Intelligence world, research is done by incorporating knowledge into the computer programs. This has proven to be an effective method for a lot of problems that humans find difficult to solve. However, for every new problem new knowledge has to be built into the program. Therefore it would be more efficient if we had programs appropriate for a wide range of problems, and not for one specific task. Also, problems which humans find relatively easy to solve, for example building a house with 10 wooden blocks, seem too difficult to solve with programs which have built-in knowledge. In this thesis I will concentrate on those seemingly simple problems.

1.2 Artificial Life

One approach that turned out to be very effective for solving many problems is the *reverse engineering* approach. This means that one looks for something that works, tries to understand it, and then rebuild it. In this thesis it means that I will look at nature, because of its excellent results. All the methods used are kept as *biologically plausible* as possible. With these methods a system will be composed of the simplest underlying mechanisms of natural evolution in order to *synthesize* life-like behaviour, or *Artificial Life*.

Specifically, I will focus on bottom-up artificial life simulations of evolving populations, because evolution is the most important property of natural life, and is responsible for the diversity, adaptation and ecological complexity that we see today and know from fossil record.

Many think evolution takes care of life, i.e. that evolution has the goal to make sure that organisms survive and reproduce. They only see the survivors, and not that most mutated animals die before they can give birth to children. The species we see are those whose predecessors developed effective surviving mechanisms. That's why it *seems* that their behaviour is to serve their life, but only in the environments in which their predecessors evolved.

Evolutionists hold that random changes are responsible for the development of living species. Yet a lot of people think in a theological way which means they hold that evolution is orientated towards certain goals. This last idea is probably based on the correct insight into solving problems and the incorrect insight into the way evolution develops. The correct insight is that humans don't invent aeroplanes by coincidence or without certain goals. The human mind *is* goal orientated. The incorrect insight however is to think that nature is out to solve problems, such as how to build flying animals.

Because we are in the engineering domain we do have certain goals we set for ourselves: we want to solve certain problems. For this we use *genetic algorithms* which are inspired by natural evolution, but where little effort has been done to model it in detail.

1.3 Genetic Algorithms

Genetic algorithms can be viewed as a biological metaphor of Darwinian evolution. The fitter an individual is, the more chance it will have to survive in its environment and produce offspring.

In GAs (*genetic algorithms*) a population of strings is used, where each string can be viewed as the genotype of an individual with a set of *chromosomes* each consisting of a number of *genes*. Each gene represents a parameter of the problem to be solved. Depending on how well an individual is at solving the given problem it is given an award in the form of a *fitness* value. Individuals which do well get a high fitness, those who do poorly get a low fitness. As with Darwinian evolution the genes of individuals with higher fitness will have a higher chance to survive than those with lower fitness. Those with low fitness will be replaced by new individuals having high fitness. A new population, or *generation*, is created by using selection and recombination mechanisms on the strings based on their fitness. The most commonly used operators are *selection, crossover, inversion* and *mutation*. A more thorough treatment of GAs is given in chapter 2.

1.4 L-systems

L-systems can be viewed as a metaphor of biological growth and development of living organisms.

The growth of living organisms is governed by genes. Each living organism starts from one cell which contains all the genetic information (the *genotype*) necessary to develop into its final form (the *phenotype*). The genetic information is not a blueprint of how the organism will look in the end, but can be seen as a *recipe* [12]. This recipe is followed by each cell, not by the organism as a whole. The way a cell will behave depends not only on the genes from which information is extracted but also on the information read in the past and its cellular environment. Thus the development of a cell is a local matter.

The biologist Aristid Lindenmayer modelled this development in plants with an *L-system* [38]. An L-system consists of a *rewriting mechanism*, so that a string can be rewritten into another string by rewriting all the characters in the string *in parallel* into other characters. Which rule is used on a character (or set of characters) depends on the character itself and its neighbouring characters. The resulting string can be interpreted in many ways. Here a G2L-system will be used which is a special case of an L-system . A G2L-system consists of a special rewriting mechanism and interpretation where a string is translated into a graph [6]. L-systems and G2L-systems will be described in more detail in chapter 6.

1.5 Artificial Neural Networks

The human brain is considered to exhibit *intelligent* behaviour. Although the meaning of intelligence is somewhat vague, maybe even an illusion we have of ourselves, humans have tried for decades to create computer programs which can be called intelligent.

Many attempts couldn’t satisfy the expectations. Deterministic programs of which the behaviour could in principle be predicted (for example the class of rule based systems) were not satisfying. When we use the *reverse engineering* approach again, we could take a look at what we know about the way the brain works. Then we could view that intelligent processing can emerge from a large number of simple computational units (neurons), each sending excitatory and inhibitory signals to each other. Because these neurons are relatively easy to model it is in principle possible to build an artificial neural network (ANN) that can compete with the complexity of a human brain. Here we face a practicality problem though, because in a human brain the amount of neurons and connections between them are huge. The current computing powers are too low to store this complexity. Therefore large simplifications have to be made in ANN.

Thus an ANN can be viewed as a biological metaphor of the human brain. A neuron in the brain is represented by a *node*. An ANN is built up from a number of nodes which are connected to each other. Each node receives some input, does some computation on this input and sends the result of this computation as output to other nodes. For the ANN to communicate with the outside world some nodes are designated as *input* nodes (senses) and some nodes as *output* nodes (expression). All other nodes that have no direct link with the outside world are called *hidden* nodes.

As with humans, an ANN has to be *trained* in order to perform a certain task. In this thesis *Back Propagation Networks* (BPN) are used as ANNs, and *supervised learning* as the method to train a network. In supervised training the network is repeatedly offered so-called *input/output pairs* until the network has learned the problem or until the teacher gives up. Each input/output pair specifies the input the network will get and the desired output the network should produce. The input values are propagated through the network, until some output is produced. The difference between the produced output and the desired output is used to calculate an error for each output node. With these errors the internal connections between nodes are adjusted. These errors are back propagated through the net, until it reaches the input nodes. This way the whole internal representation of the specific problem is changed.

1.6 GA, G2L and ANN combined

Boers and Kuiper designed a method that automatically searches for optimal modular neural networks [4]. For this they used modular techniques based on biological systems. A GA was used to find recipes which are used by the G2L-system to construct a graph which represents the architecture for an ANN. Globally the method is summarized in the following 3 steps:

1. A GA generates a population of bit-strings, which are the chromosomes of the individuals. Individuals with high fitness will have a higher probability to produce offspring. The fitness of a newly created individual is determined in step 3.
2. A chromosome from step 1 is decoded into a set of production rules. Using the G2L-system, these rules are used to rewrite strings starting from an axiom, and the resulting string is then interpreted as a graph. This graph forms the specification of the architecture for an ANN.
3. The ANN from step 2 is trained to solve a particular task. Depending on how well the network could learn the task a fitness is determined. This fitness is returned to the GA, specifically to the chromosome which contained the production rules that resulted into the network.

1.7 Research Goals

Preliminary results from using the method of Boers and Kuiper suggested that the method does work, and finds modular neural networks that perform better than networks without the specific architecture. One of the problems they encountered, however, was the amount of computing power needed. The goal of this research was to port their code containing the algorithms to a parallel computer in order to get results faster.

Specifically, in this thesis an investigation is done into *parallel genetic algorithms*. The idea is to eventually let several GAs run simultaneously on many processors, getting more ANNs evaluated, so more complex real-world problems can be tackled. In order to get the application of Boers and Kuiper up and running on a parallel hardware machine several adaptations and rewriting of code had to be made.

Chapter 2 gives an extensive overview of genetic algorithms. Chapter 3 reviews the various parallel variants of GA which can be considered as new paradigms within the field of GA. In chapter 4 results from several experiments with GA and it's parallel variants are presented. Parallel GA was applied to the famous travelling salesperson problem, the Schwefel optimization function, and Walsh polynomials. Chapter 5 and 6 give minimal introductions into L-systems and ANNs respectively. Chapter 7 describes some implementational details of the software developed. Chapter 8 combines the theory of GA, G2L-system and ANN, and results from several experiments using these theories are presented. Finally this thesis is concluded in chapter 9 and recommendations for further research are given.

Chapter 2

Genetic Algorithms

In this chapter the sequential genetic algorithm (GA) is described which is used as the basis for parallel genetic algorithms. GAs are stochastic search and optimization techniques which were inspired by the analogy of evolution and population genetics. They have been demonstrated to be effective and robust in searching very large, varied spaces in a wide range of applications. GAs were first introduced by John Holland [29].

2.1 Overview

Looking at nature one sees living species, which grow and then interact with each other. A specie is characterized by its genotype, which is independent of the environment it lives in. Genetic operators function on the genotypic level while the selection mechanism operates on the phenotypic level. According to Darwin, the fitter a specie, the higher chance it has to survive and thus produce offspring [10]. The environment determines the phenotypic expression of the genotype and bias the system towards the selection of fitter species.

Evolution is a biological metaphor of genetic algorithms. Chromosomes are represented by bit-strings (zeros and ones). The bit-string contains multiple sub-strings, or genes. An individual is coded by one or more of these bit-strings. Genetic operators like crossover, inversion and mutation are applied to these bit-strings, each with a certain probability. Thus sometimes it could happen that a child is just a copy of one of its parents. Each individual gets a fitness value determined by an evaluation function. The higher the fitness, the better the coded representation of the individual. A group of individuals makes up a population. These individuals in the population are then going through an evolution process, in order to direct the search for new individuals with higher fitnesses which give better solutions to the optimization problem at hand. In pseudo-code this process looks as follows:

```
g = 0;
initialize population P(g);
evaluate P(g);
while not done do
   g = g + 1;
   P’(g) = select parents P(g-1);
   recombine P’(g);
   mutate P’(g);
   evaluate P’(g);
   P(g) = survive (P(g-1),P’(g));
od;
```

There are four fundamental differences between genetic algorithms and more traditional optimization and search procedures. These are in short [20]:

1. GAs work with a coding of the parameter set, not the parameters themselves.
2. GAs search from a population of points, not from a single point.
3. GAs use pay-off (objective function) information, not derivatives or other auxiliary knowledge.
4. GAs use probabilistic transition rules, not deterministic rules.

Each parameter of the problem to optimize is encoded in a bitstring, which can be viewed as a gene. The set of the parameters is encoded by concatenating all the bitstrings, which can be viewed as a chromosome. The devising of a suitable coding for a problem is done by the user of the genetic algorithm. The algorithm itself has no knowledge about the meaning of the coded strings. There are parameters for the genetic algorithm itself, but usually they remain fixed for a given problem. Looking at nature one can see unchanging parameters too, like the gravitation-force constant, the weight of an electron, etc. These parameters do affect the evolution, since the individuals in the environment will adapt themselves to this parameter space automatically.

Genetic algorithms don’t work from a single point in the search space, but from an entire population of points. Each point is assigned a fitness value, which gives an indication of how good the solution is. Take for example a look at figure 2-1. It visualizes the fitness as a function over the space of possible genetic combinations using 2 genes for one individual in a particular environment. Fitness should not be regarded as an attribute of any particular gene, but rather of the total combination of genes, therefore the entire genotype. Working from a single point, for example starting at the origin, traditional search procedures like hill-climbing would likely end up at peak P_1. At that point the algorithm has found an optimum and stops searching. However, it is a local optimum, and it has missed the global optimum at peak P_2.

Using a genetic algorithm one starts at many different random points in the search-space at the same time. A population is represented as a cloud of points on the fitness surface, one for each individual. The more diverse the population, the more scattered it is on the surface. Individuals with high fitnesses have a higher probability to be selected for contributing offspring than individuals with lower fitnesses. Therefore a population tends to move uphill in the fitness surface. Also, over many generations more points will concentrate around certain regions in the landscape. In this way it is more likely that one of the points in figure 2-1 will work its way up to peak P_2.

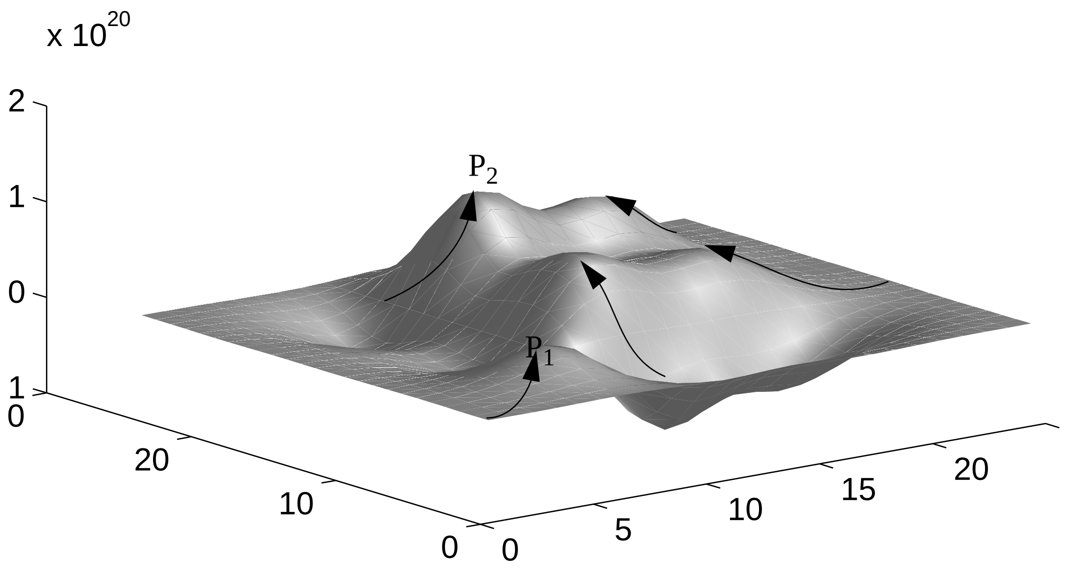


figure 2-1 *Example 2-dimensional landscape or fitness surface. The height of the surface is the fitness for each gene combination in a particular environment.*

The fitness value is calculated by an objective evaluation function. This means that the genetic algorithm does not need specific knowledge about the problem to solve. It only needs the fitness value, not the meaning of the value. Thus the genetic algorithm can be used on a wide range of problems without changing the code of the algorithm. However, because of this blindness to auxiliary information it can place an upper bound on the performance of the algorithm when compared to methods which are specially designed for the problem at hand.

Genetic algorithms use probabilistic transition rules to guide their search, instead of deterministic rules. This may sound as a simple random search, but it’s not. Chance moves the population in a random direction upon the fitness surface, but it is the selection mechanism that is used to direct the search towards regions in the search space where there is a probable improvement.

2.2 Selection

The selection mechanism of genetic algorithms operates on the phenotypic level of the individuals in the population. Every individual decodes the sought parameters of the problem, where each parameter is encoded as a gene on a chromosome. A gene is represented by a set of characters, and a chromosome by a string. According to the fitness-values of the strings those strings with higher values will have higher probability to be selected for contributing to one or more offspring in the next generation. This artificial operation can be viewed as a metaphor of natural Darwinian survival. In nature an individuals fitness is determined by its ability to survive all kinds of obstacles long enough in its environment where it becomes reproductive. In the artificial environment the objective evaluation function only cares about life or death of the string-individual.

Table 2-1 shows a sample population of 7 strings. Each string is a binary representation *x* of length 8 of a decimal value *y*. The evaluation function used is $F(y) = y^2$.

Table 2-1: Sample population

bitstring x	y	F(y) = y^2	rank	roulette wheel
11101011	235	55,225	1	0.387
10100101	165	27,225	2	0.191
10011100	156	24,336	3	0.171
10000010	130	16,900	4	0.118
01100101	101	10,201	5	0.072
01000110	70	4,900	6	0.034
00111110	62	3,844	7	0.027

In this thesis three different selection mechanisms have been tried: the *roulette wheel selection* described by Goldberg [20], the *rank based selection* described by Whitley [56], and the *tournament selection* as described by Goldberg and Deb [22].

With roulette wheel selection strings are assigned a probability proportional to their evaluation fitness. Many define the probability that individual *y* is chosen as a parent as:

$$P(y) = \frac{s_y}{\sum_{i=0}^{n-1} s_i}$$

where *n* is the population size, and s_j is evaluation score associated with the individual labelled *j*. Strings are entered into the mating pool which are selected according to those probabilities. This is like spinning a roulette wheel, see figure 2-2. Special care should be taken that the ratio between the highest and lowest probability is not too large. That may lead to the undesirable situation that certain individuals quickly dominate the population and thus cause premature convergence into a non-optimal solution. Goldberg suggests to scale the probabilities linearly by a ratio between 1.2 and 2 [20].

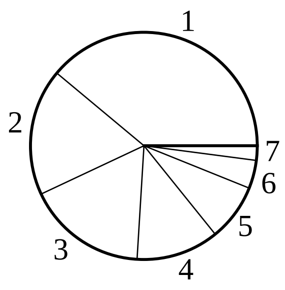


figure 2-2 *Roulette wheel for the sample population of table 2-1.*

With rank-based selection strings must be ordered by their fitness. Then each string is assigned a probability by a linear function according to its rank of the string in the population. Whitley used the following formula:

$$i = \left\lfloor \frac{\mu}{2(p-1)} \cdot \left(p - \sqrt{p^2 - 4(p-1) \cdot r}\right) \right\rfloor$$

where i is the rank of the selected string, μ is the number of individuals in the population, p is the selection pressure, and r a random value in the range $[0.0, 1.0)$. The value of the selection pressure determines the preference for the best individuals. If this pressure is between 0 and 2 all the individuals have a chance to be selected. When this pressure is for example 2.5 one third of the bad individuals are never selected. See also figure 2-3. The higher the pressure p the more the best individuals are preferred. Many choose $p = 1.5$ for good results. The main purpose of this scheme is to select as severely as possible without destroying the diversity of the population too much.

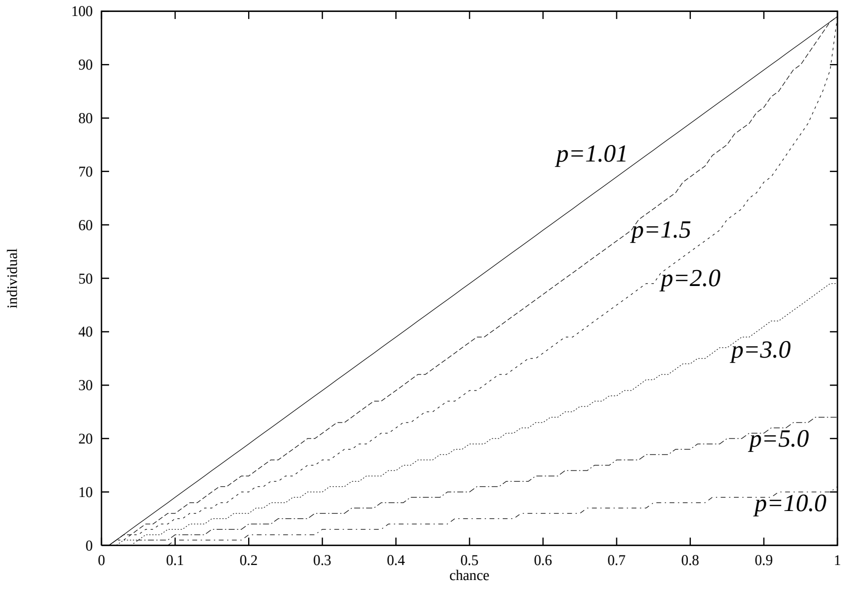


figure 2-3 *Influence of rank-pressure p on rank based selection-probability.*

Another selection mechanism is the tournament selection. From a population p individuals are randomly chosen to contest to become a parent. The individual with the highest evaluation fitness score is chosen, the rest is discarded. Then this process is repeated, so that two parents are chosen. Again, the higher the pressure p the more the best individuals are preferred. In this case $p = 2$ is usually chosen.

2.3 Replacement

One not only has to consider how to select individuals to be parents but also how to put the resulting offspring back into the population. The selection phase during the execution of a GA is applied to the *current population*. The selected strings are put into an *intermediate population* (also referred to as the *gene pool*). Then the genetic operators crossover, inversion and mutation are applied to the intermediate population, which creates the *next population*. The next population can then be evaluated. The process of going from the current population to the next population forms one generation in the execution of a GA.

Mainly there are two ways to create a next generation. One is to replace the whole current population by the next population. The sizes of the current, intermediate and the next population have to be equal in this case. This *generational replacement* was used by Goldberg in his Simple Genetic Algorithm (SGA) [20].

The other option is the so called *steady-state replacement* as described by Whitley [56]. A newly created individual replaces the worst individual of the current population when the new individual has a higher evaluation fitness. This can be seen as a process where the size of the intermediate population is 1. The next population is almost an exact copy of the current population, except for maybe 1 individual which came from the intermediate population. Therefore this model is often referred to in literature as the *static population model*. The best solutions always stay within the population and therefore the best fitness value increases monotonously. Steady-state replacement is always better than creating a whole new population when it can be shown that a global optimum can be reached from a string in the population without passing any local optima during the process [58].

To place these two replacement methods in the light of mother nature again, when we look at certain short-living individuals the parents are dead before their offspring are able to reproduce (e.g. fruitflies). This is what the generational model is based on. With longer living individuals (e.g. mammals) parents and children live at the same time. Children and parents compete together for selection. The static population model is based on this.

An important difference between a generational replacement GA, and steady-state GA is that in the latter new offspring are immediately available for reproduction. Such a GA has therefore the opportunity to exploit the promising individuals as soon as they are created.

In this thesis some changes are made to these two standard algorithms. Together with the generational method the *elitist method* is used. With the elitist method the best individual of the current population is saved into the next population. Then the best fitness value increases monotonously, just as in the steady-state model. When using the steady-state method the size of the intermediate population is chosen somewhat larger than 1.

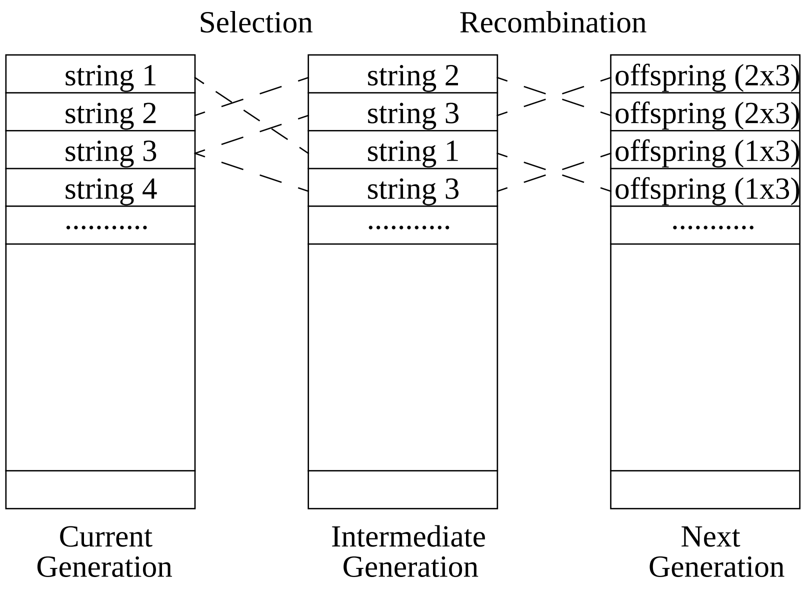


figure 2-4 *One generation is divided into a selection phase and a recombination phase. Strings selected are copied to an intermediate population. Recombination is then performed on pairs of strings. Mutation can be done after recombination.*

The selection phase of the GA produces two parents for each new individual that is to be included in the intermediate generation. Three genetic operators are used to create the offspring: crossover, inversion and mutation. These mechanisms operate on the genotypic level of the individuals in the population. They are explained next.

2.4 Crossover

The crossover operator takes different parts from two parents from the mating pool and creates a new individual by combining the parts. So the offspring contains genetic information from both parents. Only *reciprocal crossover* is considered, where equivalent length sub-strings are exchanged. Crossover points (usually two, and they have to be between two genes) are chosen randomly. The new individual, or child, will then consist of alternate parts of the parent strings.

Though randomized, this exchange of information gives genetic algorithms much of their power. Good parts from one parent could replace bad parts from the second parent, and thus create a child with only partial good parts. See figure 2-5 for an example of crossover.

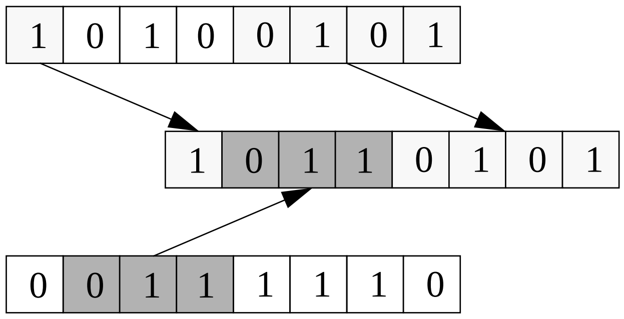


figure 2-5 *Crossover of the members ranked 2nd and 7th from table 2-1.*

2.5 Inversion

Inversion takes two points in a chromosome at random and inverts the sequence of the genes between these two points. The genetic algorithm has to keep track of the positions of each gene within a chromosome, because the genes have to retain their intended meaning regardless of their position. Therefore inversion has no immediate effect on the fitness of the individual. The reason to use inversion is to make it possible that good string arrangements are built which are less likely to be destroyed by crossover. An example is shown in figure 2-6, using strings with genes containing only 1 bit.

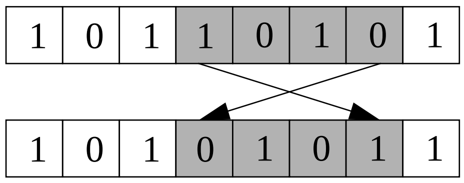


figure 2-6 *Inversion of the crossover result from figure 2-5.*

2.6 Mutation

Mutation is a very simple genetic operator. It changes at random the value of a gene in a newly created individual. As in natural populations the mutation rate used is very small, in the order of one mutation per thousand genes transfers. Thus mutation is considered to be a secondary mechanism of genetic algorithms. Yet it is still used to introduce new solutions into the population and to protect the algorithm from premature loss of important genetic material by reintroduction of genes.

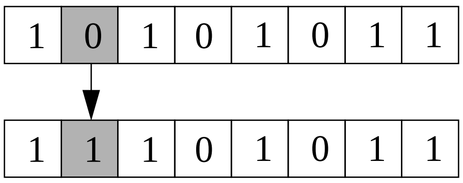


figure 2-7 *Mutation of the inverted string from figure 2-6. Bit 2 has been mutated.*

2.7 Schema Theorem

At this point one could wonder why a process as described above should result in an effective form of optimization to find (near-)optimal solutions. Although there is no accepted theory which explains exactly why GAs perform well, some hypotheses have been introduced which can partially explain this success.

Holland's *schema theorem* [29] was the first hypothesis to explain how GAs work. A schema is a similarity template describing a (sub)set of chromosomes. If the population consists for example of binary strings, we can define a ternary alphabet {*0,1,#*}. With this we can build a schema to represent a pattern of gene values. A chromosome is part of the schema when at every location a *0* in the schema matches a *0* in the chromosome, a *1* matches a *1*, and a *#* (*wild card* or *don't care* metasymbol, which is never explicitly processed by the genetic algorithm) matches either a *0* or a *1*. So for example the schema *#0#1* matches the subset {*0001, 0011, 1001, 1011*}. The idea of a schema is thus to give a powerful and compact way to talk about all the well-defined similarities among finite-length strings over a finite alphabet.

Let's first look at an example, using the strings from table 2-1 as the starting situation. Suppose we want to maximize the function $F(x) = x^2, x \in [0, 255]$. The GA doesn't know anything about the fitness function *F*. The only information the GA has access to are the strings and their fitness values. When you look at the strings separately you will only see ten pieces of information. But when you consider the strings, their fitness values, and the similarities among the strings in the population you admit a wealth of new information to help directing the search for the maximum. In our example it seems like a good idea to put a *1* in the first position of new strings. This idea of using similar (small) parts of highly fit strings to create a new string can be explained more precisely using the concept of *schemata* and *building blocks*.

It is useful to begin formalizing the potential sampling rate of a schema, *H*. In a population of *n* bitstrings of length *l* there are 3^l schemata (every bit-position can be *0,1* or *#*). One bitstring in the population belongs to 2^l-1 schemata because each position in the string can be either the bit value contained in the bitstring or the # symbol (the string of all # symbols represents the search space itself and is not counted as a partition of the space [29]). The defining length of a schema *H*, denoted by $\Delta(H)$, is the difference between the indexes of the position of the left-most respectively the right-most occurrence of either a *0* or *1*. The defining length is a direct measure of how many possible crossover points fall within the significant portion of a schema. Let *m(H,t)* be the number of strings representing schema *H* in a population at a certain generation *t*. Let *f(H)* be the average fitness of the strings representing schema *H*, and f_{avg} the average fitness of the total population. The effect of the reproduction operator on schema *H* can be expressed by:

$$m(H, t+1) = m(H, t) \cdot \frac{f(H)}{f_{avg}}$$

Thus schemata which have higher fitness values than the average fitness of the population increase (exponentially) in number of samples in the next generation. This is done for all schemata in the population in parallel. Of course, by reproduction alone no new sampling of schemata are actually occurring since no new strings are generated. Crossover and mutation provide the means for this. When we define p_s as the probability that a schema stays intact after crossover, and p_c as the probability that crossover will occur, then we can express the following:

$$p_s \geq 1 - p_c \cdot \frac{\Delta(H)}{l-1}$$

Furthermore, let p_m be the mutation probability. Every position in a schema which has value *0* or *1* has a chance of p_m to be flipped. Mutation on a position with a *#* symbol has no effect on a schema. The order o of a schema *H* is defined as the number of bits in the representation of *H* that have value *0* or *1*, and is denoted by o*(H)*. Then the probability that mutation affects the schema representing *H* is $(1-p_m)^{o(H)}$. For small mutation rate p_m<<1 this can be estimated by $1 - p_m \cdot o(H)$.

Combining the effects of reproduction, crossover and mutation leads to the following expression of the *schema theorem* :

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{f_{avg}} \cdot \left[1 - p_c \cdot \frac{\Delta(H)}{l-1} - p_m \cdot o(H)\right]$$

It can be seen that schemata with high fitness have a higher chance of surviving into the next generation. Also schemata with short defining length have better chances of survival than schemata with long defining length (schemata with short defining length are also called *building blocks* [20]). So the schema theorem learns that schemata with short defining length and high fitness propagate exponentially throughout the population.

Holland [28] argued that since each string contains a great many schemata, the number of schemata which are effectively being processed in each generation is of the order n^3. However, this argument does not hold in general for any population of size *n*. If *n* is chosen equal to 3^l then at most *n* hyperplanes can be processed. Therefore, *n* must be chosen with respect to the stringlength *l* to make the n^3 argument reasonable. Still, the argument is important because it says that despite the disruption of long, high-order schemata by crossover and mutation, genetic algorithms inherently process a large quantity of schemata while processing a relatively small quantity of strings. This property of genetic algorithms is known as *implicit parallelism*[1].

2.8 Deception

An important question for GA research to answer is “What problems pose challenging optimization tasks for a GA?’’. One of the major approaches to this has been the study of *deception* [57].

For a GA to be successful chromosomes need to be built, via the process of crossover, from optimal schemata (especially *building blocks*) which are contained in the global optimum. However, when schemata which are not contained in the global optimum increase in frequency more rapidly than schemata which are contained in the global optimum, a GA is misled, so that the GA converges towards a local optimum instead of towards the global optimum. The GA is misled because the best points tend to be surrounded by the worst.

Imagine a person is looking for the highest point in an area as drawn in figure 2-8. A person (or GA) at point A must determine which direction to take, and all the information the person has got is on a signpost (or schemata) at that point. The signpost says for example: “The land to the west is generally higher than the land to the east”. This is correct information, and the person will probably head off west. However, the word ‘generally’ makes the signpost deceptive. The west may have high ground and the east overall low ground, but the peak is east.

Practically speaking, there is usually some regularity in the problems encountered in the real world, so those problems usually don’t suffer from deception. Still, it is important to keep the behaviour of deception in mind.

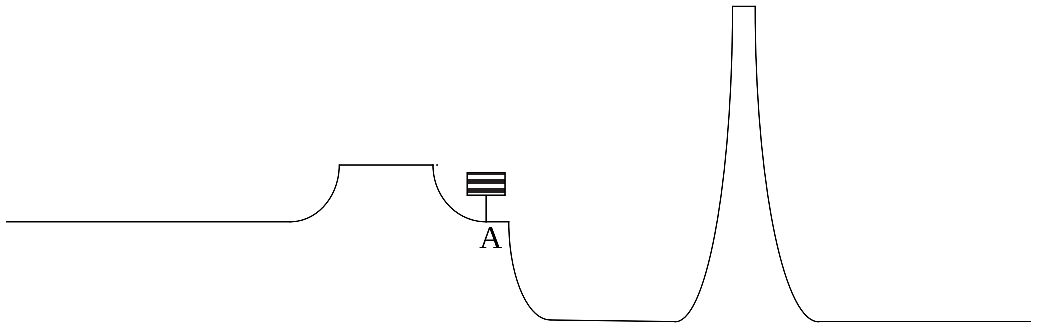


figure 2-8 *1-dimensional landscape*

1. Implicit parallelism does not refer to the potential for running genetic algorithms on parallel hardware, although genetic algorithms are generally viewed as highly parallelizable algorithms, as will be presented in chapter 3.

Chapter 3

Parallel Genetic Algorithms

During recent years the area of genetic algorithms in general and the field of *Parallel genetic algorithms* in particular has matured up to a point, where the application to a complex real-world problem in some applied science is now potentially feasible and to the benefit of both fields. Due to increasing demands placed on genetic algorithms, such as searching large search spaces with costly evaluation functions and using large population sizes, there is an ever growing need for fast implementations to allow quick and flexible experimentation. Parallel processing is the natural route to explore. Furthermore, some of the difficulties that face standard GAs (such as premature convergence) may be less of a problem for parallel variants. Parallel GAs can even be considered as creating new paradigms within this area and thus establishing a new and promising field of research. In this chapter various variants of parallel GA are described, together with short reviews of related work. First an overview is given of the various parallel hardware architectures available which affect the way a GA can possibly be parallelized.

3.1 Parallel Hardware

Parallel computing, or the concurrent operation of separate processing units, has for a long time been used as a technique to derive better performance from a given technology. Over the years various types of concurrency within a computer system evolved to enable several operations to occur at once. Flynn made an early attempt to divide the various types of computer systems into four major categories based on the number of instruction and data streams that are processed simultaneously [16, 17]. Flynn’s categories are as follows:

- Single instruction stream, single data stream machines (SISD).
- Single instruction stream, multiple data stream machines (SIMD).
- Multiple instruction stream, single data stream machines (MISD).
- Multiple instruction stream, multiple data stream machines (MIMD).

The SISD architecture executes a single program as one sequential procedure using a single set of data and is still today’s most used type of computer (e.g. Intel 80486). SIMD machines may invoke a plurality of data operations to occur concurrently during one instruction cycle. Generally a SIMD machine is massively parallel, with hundreds or thousands of processor elements. These processor elements are generally not based on 8- or 16-bit microprocessor chips but on even simpler bit-organised, special purpose chips. They are relatively slow, cheap, and the amount of local memory on the chip is low. Processor elements can communicate with neighbouring processor elements. Systems with this architecture are restricted in the problems that they can solve, but are particularly suited to operations on large matrices (e.g. MasPar MP-1). MISD machines allow actions within an instruction cycle to overlap with different actions of consecutive instruction cycles on a single data stream to achieve a higher rate of instruction execution. However, machines with these architectures are very rare (e.g. CRAY-205). Lastly, the MIMD organization allows multiple sets of possibly different instructions to execute separately and concurrently on multiple sets of data. Processors can communicate with each other via messages. Generally a MIMD machine is transputer-based, with 8 to 64 processors. The processors are relatively fast, and the amount of local memory high. The processors may share a common memory (e.g. CM*).

It has become common to study the performance of parallel systems in terms of *speed-up* and *efficiency*. Let T_p be the time required to perform some calculation on a system with p processors. Then the speed-up factor over a uniprocessor is

$$S_p = \frac{T_1}{T_p}$$

and the efficiency in theory is

$$E_p = \frac{S_p}{p} \le 1$$

In theory, when p processors working concurrently on a task take $1/p$ of the time that it takes for one processor to complete the same task then the efficiency E_p will be unity. Usually communication and synchronisation problems are amongst the factors which prevent this from being achieved. The arguments behind the definitions of speed-up and efficiency appear very reasonable and simple, so they are widely used, but there is reason to doubt their validity when applied to today’s real multiprocessor systems. Parallel computing can introduce paradigms, that are not so simple to implement in serial computing, which can cause the effect of super-linear speed-up. It has also been shown possible that program code running on 8 processors can produce the same results more than twice as fast as when the same code is run on 4 processors (e.g. [31]), which also implies super-linear speed-up.

Genetic algorithms lend themselves excellently for effective parallelization, giving their inspiring principle of evolving in parallel a population of individuals. However, the classical Holland’s algorithm is not straightforwardly parallelizable because of its need (in the selection step and in crossover for some of its many variations) of global control that causes the need to serially execute a piece of code and to force non-local interactions between the processors, thus requiring high communication costs.

Several approaches have been proposed to overcome these limitations yielding excellent results. The different models used for distributing the computation and to ease parallelization, cluster around three main approaches which will be explained in the next sections. First, a parallel GA similar to the traditional GA model will be reviewed. Next two approaches are described which decompose the population as a particular model of a GA. The first decomposition approach explained is the *island model* which divides a population into several equal subpopulations, each of which runs a GA independently and in parallel in respect to the other subpopulations. Occasionally individuals are migrated to another subpopulation. The second decomposition approach explained are the *cellular genetic algorithms*. This parallel variant of GA divides a population in such a way that each processor typically only has one individual. Each processor then uses GA operators only on individuals in a bounded region. Fit individuals propagate throughout the whole population because all regions overlap each other.

3.2 Global Populations with Parallelism

In these models one global population is kept, on which the traditional genetic operators are performed. Selection is done on the global population, and then the selected individuals undergo crossover and mutation in parallel.

Grefenstette mentions a *synchronous master-slave* model [24]. Here a single master process keeps the whole population in its own memory, performs selection, crossover and mutation on the individuals, but leaves the calculation of the fitness of the new individuals to *k* slave processes. The problem with this model is that a lot of time is wasted if some slave processes finish their evaluation significantly quicker than others. Another problem is that much is depending on the master process. Grefenstette then mentions the *semisynchronous master-slave* model, where an individual is inserted when a slave process is done, and that same process gets a new individual as soon as possible. Again the problem of dependency on one master process is still there. Lastly Grefenstette considers the *asynchronous concurrent* model. Here the individuals of the population are kept in a common shared memory, which can be accessed by *k* concurrent processes. Each process performs function evaluations, but also genetic operations. Each process works independently of the others[1]. Whitley states that the only difference between this last model and the traditional (serial) genetic algorithm is a change in the selection mechanism [58]. Selection can best be done by tournament selection. Assume that one has a population of ***N*** individuals and *N/2* concurrent processors. Twice each processor randomly

1. This model was actually used by Boers and Kuiper [4]. At one time they used for a specific problem 11 Sun Sparc4 workstations. Each workstation selected several individuals from a population which was maintained in a global shared file. Then each workstation performed genetic operators on the selected individuals, evaluated them, and then wrote the newly created individuals plus their fitnesses back to the global file.

selects two individuals from the shared memory, and keeps the best. The two selected strings are then subjected to crossover, mutation and evaluation. The resulting children are put back in the population in the shared memory. This all goes in parallel.

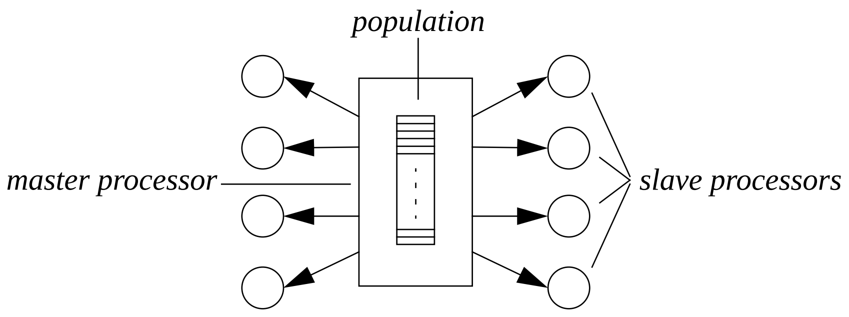


figure 3-1 *Global population with parallelism*

3.3 Island Models

Looking at nature again one could say that the world of humans consists of one big population. Another view is to say that it's actually a collection of subpopulations which evolve independently from each other on isolated continents or in restricted regions. Once in a while some individuals from one region migrate to another region. This migration allows subpopulations to share genetic material. The idea is that isolated environments, or competing islands, are more search-effective than a wider one in which all the members are held together.

If we apply this to genetic algorithms we could parallelize it by letting each processor run its own (sequential) genetic algorithm with its own subpopulation, but each trying to maximize the same function. If a neighbourhood structure is defined over the set of subpopulations, and once in a while each subpopulation sends its best individuals to its neighbours, we say we're running a *distributed genetic algorithm.* If no swapping of individuals to neighbours is done we have a special case of the distributed model, which we call the *partitioned genetic algorithm.* These models are most suited for MIMD machines.

Since each processor starts with a different initial population *genetic drift* will tend to drive these populations into different directions. By introducing migration the island model is able to exploit differences in the various subpopulations; this variation represents a source of genetic diversity. However, migrating a large number of individuals too often may drive out any local differences between islands, thus destroying global diversity. On the other hand, if migration occurs not often enough, it may lead to premature convergence of the subpopulations. When dealing with an island model the main issues to be considered are:

- the processors with which each processor exchanges individuals with.
- migration frequency, or how often a processor exchanges individuals.
- migration rate, or the number of individuals exchanged between processors.
- the individuals chosen to exchange.
- the individuals deleted after having received individuals from others.

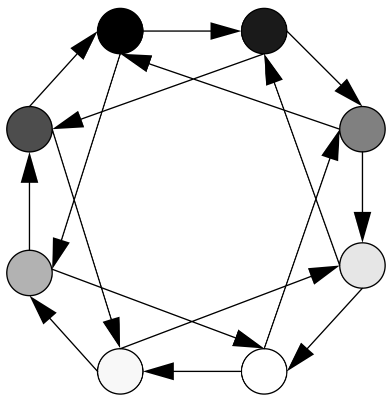


figure 3-2 *Example of an island model*

One of the first island models was introduced by Pettey, Leuze and Grefenstette [46]. Their problem was that they had individuals with long bit-representations and the evaluation of the fitness relatively took a lot of time. They tried parallel evolution of several subpopulations where each process followed the same procedure.

Cohoon, Hedge, Martin and Richards investigated the effects on the evolution process of having a diversity of environmental characteristics across the populations [7]. Here they are influenced by the theory of *punctuated equilibria* [14]. In short this theory holds that the emergence of new species can be associated with very rapid evolutionary development after a geographically separation. Thus they proposed a distributed genetic algorithm where each subpopulation evolves until it reaches equilibrium in a stable environment (*stasis*), after which the environment is changed by merging (previously isolated) subpopulations together. Later they investigated the use of different control parameters (such as crossover and mutation rates, or population sizes) per processor as another way to differentiate the subpopulations [8].

Tanese tried to answer the question whether a distributed GA can achieve near-linear speed-up without compromising its performance, or better yet, whether it can obtain even better performance than the traditional version [53]. In her experiments she used a population size of 256 for the traditional GA, and for the distributed GA various subpopulation sizes while maintaining the total population size to 256. Her first experiment used the partitioned GA studying the effect of dividing a large population into a number of small subpopulations. Tanese's result to this was that the partitioned GA consistently found better individuals than the traditional GA (even with relatively small subpopulations), but worse average fitness of the total population.

Her second experiment used the distributed GA studying the effect of various migration rates and frequencies. Results showed again that the distributed GA consistently found better individuals than the traditional GA, but because of migration it also could achieve higher average fitness of the total population. This was best achieved with a moderate migration rate, for example migrating 20% of each subpopulation every 20 generations or so (when using generational replacement). In both experiments near-linear speed-up was achieved.

Mühlenbein, Schomisch and Born implemented the island model with rank-based selection to act as function optimizers [42]. Starting from solutions generated by a GA they used an additional greedy local search algorithm. They tested their system on the five de Jong functions (F1-F5) [35], the highly multimodal Rastrigin's function (F6), and two other functions proposed by Schwefel (F7) and Griewangk (F8). They showed that the time needed to reach the optimum and the number of function evaluations compare favourably with previously published results on the same problems obtained with standard GAs. They also argued that some problems can be too easy on parallel hardware, which could explain why these problems showed little speed-up. On Schwefel's function F7 they showed superlinear speed-up when increasing the number of processors from 4 to 8. They concluded that parallel search pays off only if the search space is large and complex, and that it is able to solve complex problems that have not been previously solved.

Pettey et al. also tested the five de Jong functions on a parallel GA, but using randomly communicating subpopulations [46]. Because they exchanged individuals every generation the results showed an increased likelihood of premature convergence. They also concluded that an increase in the number of processors improves convergence speed, but not the quality of the solution.

Norman also tested a parallel GA using randomly communicating subpopulations [44]. Individuals were accepted when their fitness was better than the fittest individual. Individuals which were just as fit as the least fit individual were accepted with probability p_0. Individuals with a fitness between the best and least fit individual were accepted with probability p_1, where p_1 is a linear interpolation between p_0 and 1. The migration frequency was variable. Norman then showed improved convergence speed, and also an improvement in the quality of the solution.

3.4 Cellular Genetic Algorithms

Genetic algorithms implemented on a SIMD machine typically have one individual string residing at each processor element (cells). Individuals select mates and recombine with other individuals in their immediate neighbourhood (e.g. north, south, east and west). This class of genetic algorithms are in fact a subclass of cellular automata. Thus, the term *Cellular Genetic Algorithm* is proposed to describe this class of parallel algorithms.

Parent selection is not used as proposed by Holland, because it relies on a global ranking of all individuals. Global ranking of all individuals introduces an unnecessary central control and the amount of communication overhead would become too costly. Instead, each individual selects an individual in its local neighbourhood as its mate. This can be done by selecting the best individual from among the neighbours or by some local random selection scheme.

The selected individual is then mated with the individual residing in the cell. One offspring is produced and may or may not replace the individual in the cell depending on the replacement scheme chosen. The model is thus fully distributed with no need of any central control.When dealing with a cellular model the main issues to be considered are:

- the neighbourhood structure
- the selection scheme
- the replacement scheme

Although there are no islands in this model, when one assumes to have a 64x64 2-dimensional grid of processor elements and one neighbourhood is say 30 moves away from another neighbourhood, then these neighbourhoods can be viewed as isolated as two subpopulations in the island model. This kind of separation is referred to as *isolation by distance* [41].

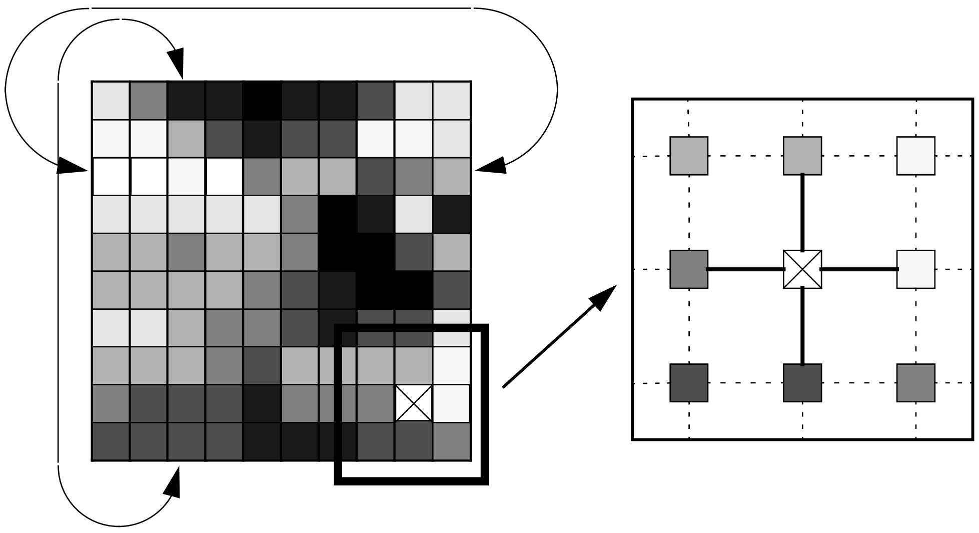


figure 3-3 *A cellular model, together with a neighbourhood relation.*

The colouring of the cells in figure 3-3 represents genetically similar material that form virtual islands isolated by distance. The arrows indicate that the grid wraps around to form a torus. On the right side of figure 3-3 an example of a neighbourhood structure is given. The processor with the cross can communicate with its south, east, north and west neighbours. Communication between the cells only takes place during the selection phase. In the example of figure 3-3 all processors concurrently send their individual to their north neighbour, while at the same time all the processors concurrently receive an individual from their south neighbour. This is repeated for all existing directions in the neighbourhood structure. Thus in four steps all processor elements will have all the information they need to continue the GA process locally. Because of synchronisation of the processors the evaluation of one individual should take about the same time for all individuals. If for example on a 4096-processor system one processor needs *100t* clock cycles to finish evaluating its individual while all the others only need *t* clock cycles, then 4095 processors are *99t* clock cycles being idle.

Several people have done research using cellular GA, among them are Baluja [3], Collins and Jefferson [9], Davidor [11], Spiesens and Manderick [50, 51] and Gorg-Schleuter [23]. The results obtained were generally that on hard problems the cellular GA provided better solutions compared to the standard one. It was less prone to get stuck in local optima, could find several optima in the same run, the diversity of the genes was greater, and it was more robust concerning the parameter settings.

Chapter 4

Parallel GA Experiments.

This chapter describes some experiments using parallel genetic algorithms on a variety of problems. These include:

- Travelling Salesperson Problem
- Schwefel Optimization Problem
- Walsh Polynomials

4.1 Travelling Salesperson Problem

Introduction

In the travelling salesperson problem (TSP) , a hypothetical salesperson must make a complete tour of a given set of cities in the order that minimizes his total distance travelled. He should return to the starting point and no city may be visited more than once. This problem may seem easy, but it belongs to the class of NP-complete problems, which means it is currently not solvable in deterministic polynomial time. If there are n cities to be visited, the search space is n!. So for instance, even a seemingly simple problem with 30 cities would need $8 \cdot 10^{18}$ years to solve if a supercomputer could be used that can evaluate 1 million cities per second. Consequently, heuristic methods should be used to deal with this problem. They may find solutions that are only approximations of the optimum, but they will do it in a reasonable amount of time. In this case a GA is used.

First an important question connected with the chromosome representation is addressed. Usually chromosomes are represented as binary strings, which allows for binary mutation and crossover. For example, when applying these operators to the table 2-1, that yielded into legal offspring, i.e., offspring within the search space. This is not the case for the travelling salesperson problem. For example, for a TSP with 10 cities, a city needs 4 bits to be represented within a chromosome. However, some 4-bit sequences do not correspond to any city (e.g. 1101). This can be solved in two ways. We could use a so-called *repair algorithm* which repairs a chromosome so that it is moved back into the search space. Or we could use a different chromosome representation, and instead of using repair algorithms we could incorporate the knowledge of the problem into the genetic operators.

The search space for the TSP is the set of permutations of the cities. The most natural way to represent a tour is through a path representation, where the cities are listed in the order in which they are visited. As an example of a path representation, assume there are 6 cities {1,2,3,4,5,6}. Then the tour (1 2 3 4 5 6) means the salesperson visits city 1 first, city 2 second, city 3 third, ... , returning to city 1 from city 6.

Although it seems natural enough, there are at least two drawbacks to this representation. First, it is not unique. For example, (2 3 4 5 6 1) and (3 4 5 6 1 2) actually represent the same tour as (1 2 3 4 5 6) (i.e., representation is unique only up to direction of traversalclockwise or counterclockwise and originating city). This representational ambiguity generally confuses the GA. Second, for the tour representation, a simple crossover operator could fail to produce legal tours. For example, the following strings with cross site 3 fail to produce legal tours.

```
before crossover     (1 2 3 4 5 6)
                     (4 5 2 3 6 1)
crossover site             ^

after crossover      (1 2 3 3 6 1)
                     (4 5 2 4 5 6)
```

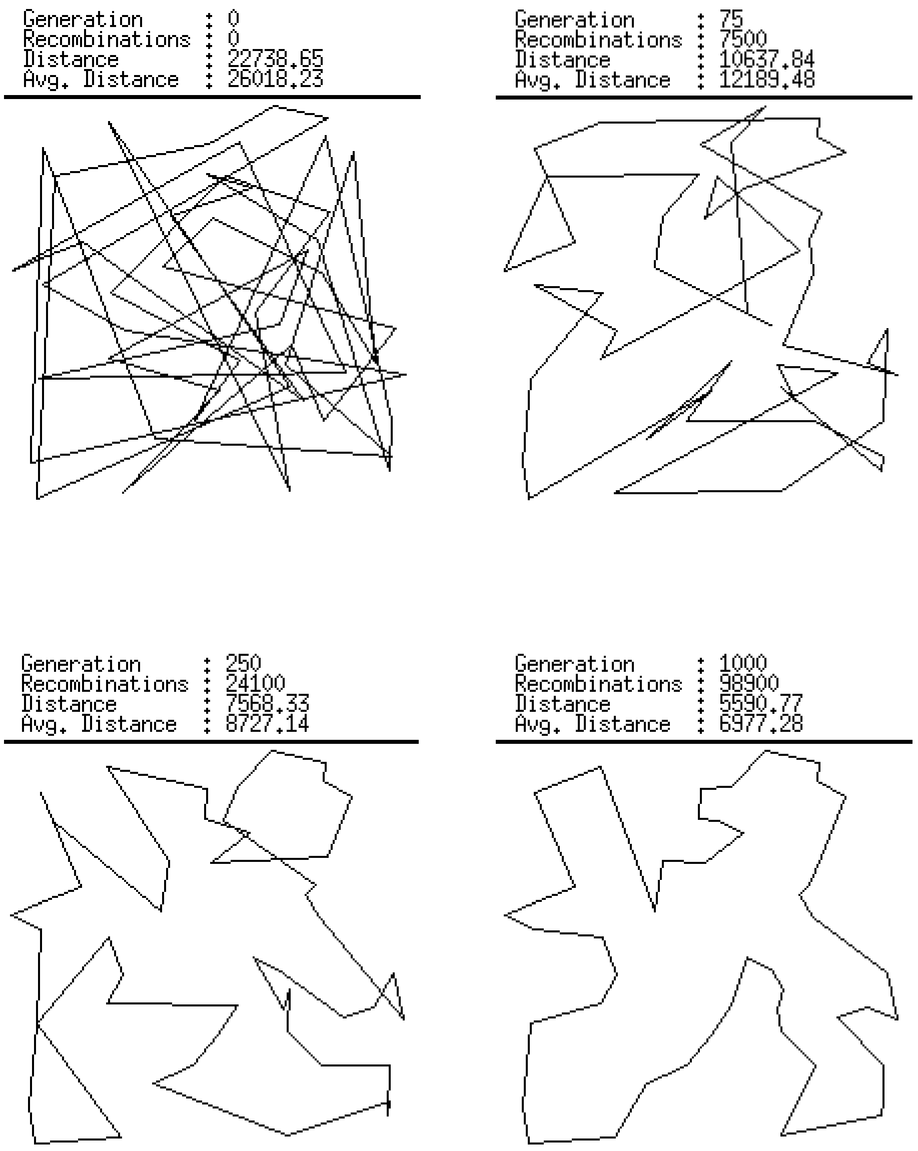
Later in this section it will be shown how to overcome this problem.

The TSP has an extremely easy evaluation function. For any potential solution, we can refer to the table with distances between all cities to calculate the total length of the tour. Thus, in a population of tours, we can easily compare any two of them.

Tournament selection is used with a pressure of 2. When the new generation is full the old is replaced by the new in one step, i.e. generational replacement is used. In addition, the GA uses the elitist strategy. The elitist strategy guarantees that the best individual of a population survives to the next generation.

The mutation operator is specifically made for this TSP. It takes a chromosome, randomly selects two points and randomly scrambles the cities between the two points. The length of the mutation segment shouldn't be too large, so as not to change too much in the original chromosome. The inversion operator used randomly chooses two cities of a tour, and inverts the subpath between these two cities inclusively.

To illustrate the working of a genetic algorithm on the travelling salesperson problem I made a program that can be run under an X-window environment. Figure 4-1 shows a simple example using only mutation and inversion performed on a 50-cities problem. The population consisted of 100 individuals. Mutation rate was 0.01 and inversion rate was 0.6.



```
Generation     : 0
Recombinations : 0
Distance       : 22738.65
Avg. Distance  : 26018.23
```

```
Generation     : 75
Recombinations : 7500
Distance       : 10637.84
Avg. Distance  : 12189.48
```

```
Generation     : 250
Recombinations : 24100
Distance       : 7568.33
Avg. Distance  : 8727.14
```

```
Generation     : 1000
Recombinations : 98900
Distance       : 5590.77
Avg. Distance  : 6977.28
```

figure 4-1 *Some best tours found during a simulation with 50 cities.*

The above figure shows some of the best individuals of subsequent populations found during the simulation. The top-left tour is the starting best individual, the tour at the bottom-right is the found optimum. With each generation the number of recombinations (in this case the number of inversions) is given that were done that lead to the shown tour, the distance of the best tour (the length of the path), and the average distance of all the individuals in the population. The genetic algorithm quickly halves the best distance to about 10,000. After that it slowly converges to the found optimum. Figure 4-2 shows the distance of the best individual and the average distance of the population against the number of generations.

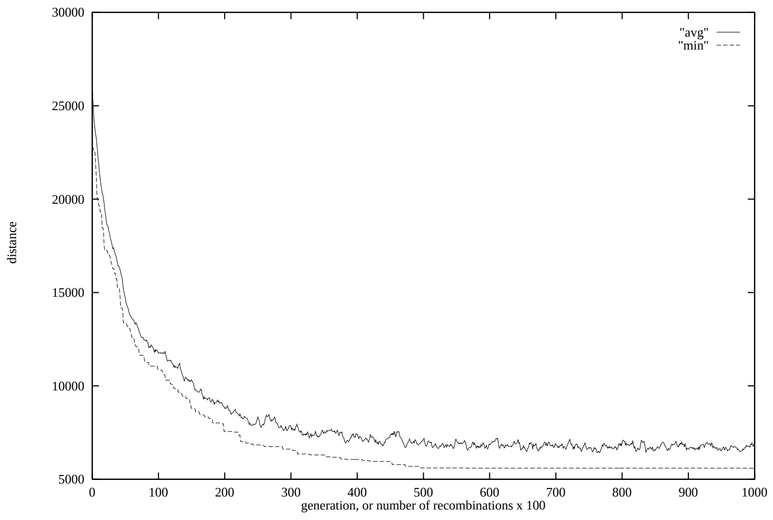


***figure 4-2** Average and best tourlength against number of generations.*

The MX operator

In the next experiment crossover is used. For this the matrix crossover operator is chosen as described by Homaifar et al. [31]. This approach is based on the matrix representation of the tours.

Two parents are chosen from the population which are then transformed into a matrix representation, where entry (*i*,*j*) from the matrix is set to 1 when there exists an edge from city *i* to city *j*, otherwise the entry is set to 0. This representation is thus unique up to traversal direction. Two crossover points are randomly chosen which cut the matrices vertical. Then the offspring are made from the parents by exchanging the columns as determined by the crossover points. Figure 4-3 shows an example of his operation.

It could be that the resulting offspring contain more than one 1 in a row or contain cycles. Therefore we need some sort of a *repair algorithm*; such an algorithm would *repair* a chromosome, moving it back into the search space. The first step of the repair algorithm moves a “1” from each row with duplicate “1”s into another row that has no “1” entries. For example, in the first offspring child1 from figure 4-3, the algorithm may move entry (1,7) into (3,7), and the entry (6,4) into (8,4). Similarly, in the second offspring child2, the algorithm may move entry (3,4) into (1,4), and (8,3) into (6,3). After the completion of the first step of the repair algorithm, the first offspring represents a (legal) tour,

```
(1 3 7 5 8 4 6 2),
```

and the second offspring represents a tour which consists of two subtours,

```
(1 4 2 8 7 5) and (3 6).
```

```
parent1: (5 8 3 4 6 2 1 7)        parent2: (3 6 4 2 8 7 5 1)

 | 1 2 3 4 5 6 7 8                  | 1 2 3 4 5 6 7 8
-+----------------                 -+----------------
1| 0 0 0 0 0 0 1 0                 1| 0 0 1 0 0 0 0 0
2| 1 0 0 0 0 0 0 0                 2| 0 0 0 0 0 0 0 1
3| 0 0 0 1 0 0 0 0                 3| 0 0 0 0 0 1 0 0
4| 0 0 0 0 0 1 0 0                 4| 0 1 0 0 0 0 0 0
5| 0 0 0 0 0 0 0 1                 5| 1 0 0 0 0 0 0 0
6| 0 1 0 0 0 0 0 0                 6| 0 0 0 1 0 0 0 0
7| 0 0 0 0 1 0 0 0                 7| 0 0 0 0 1 0 0 0
8| 0 0 1 0 0 0 0 0                 8| 0 0 0 0 0 0 1 0
      ^     ^                            ^     ^
     MX sites                           MX sites


child1:                            child2:

 | 1 2 3 4 5 6 7 8                  | 1 2 3 4 5 6 7 8
-+----------------                 -+----------------
1| 0 0 1 0 0 0 1 0                 1| 0 0 0 0 0 0 0 0
2| 1 0 0 0 0 0 0 0                 2| 0 0 0 0 0 0 0 1
3| 0 0 0 0 0 0 0 0                 3| 0 0 0 1 0 1 0 0
4| 0 0 0 0 0 1 0 0                 4| 0 1 0 0 0 0 0 0
5| 0 0 0 0 0 0 0 1                 5| 1 0 0 0 0 0 0 0
6| 0 1 0 1 0 0 0 0                 6| 0 0 0 0 0 0 0 0
7| 0 0 0 0 1 0 0 0                 7| 0 0 0 0 1 0 0 0
8| 0 0 0 0 0 0 0 0                 8| 0 0 1 0 0 0 1 0
```

figure 4-3 *Two intermediate offspring after the first step of the MX operator.*

The second step of the repair algorithm should be applied to the second offspring only. During this stage, the algorithm cuts and connects subtours to produce a legal tour while preserving as many of the existing edges from the parents as possible. For example, the edge (4 6) is selected to connect these two subtours, since this edge is present in the first parent. Thus the complete tour (a legal second offspring) is

```
(1 4 6 3 2 8 7 5).
```

To prove that this MX operator really works a simulation was run of a GA trying to solve the same 50-cities problem given earlier, but this time only using the MX operator with a rate of 0.65 (thus no mutation and no inversion operator are used). Figure 4-4 shows the result, where the best distance found is plotted against the number of MX operations performed.

After a certain amount of generations the GA has converged and therefore doesn't improve any more. The best distance isn't that good, compared to the best found individual from figure 4-1. When other genetic operators, such as mutation and inversion, are used together with the crossover operator, the algorithm finds better solutions.

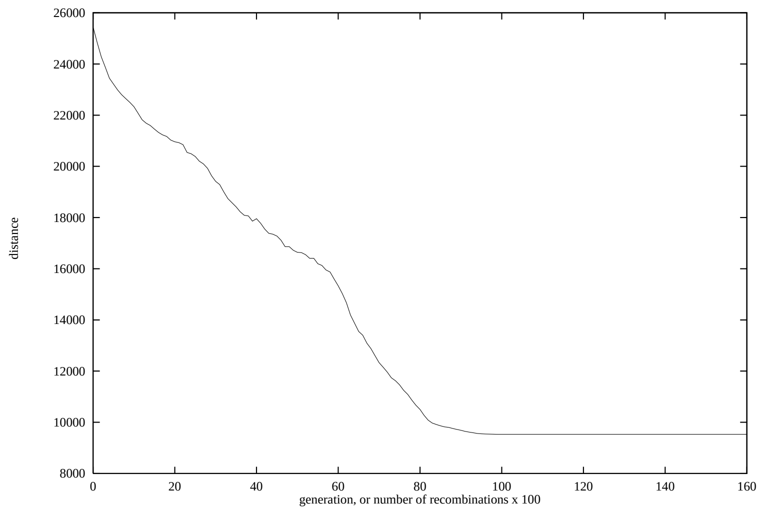


***figure 4-4** Length of best tour against number of MX operations.*

Partitioned and distributed GA performed on the TSP

Most performance results reported for the TSP heavily depend on many details (population size, number of generations, size of the problem, etc.). Moreover, many results were related to relatively small sizes of the TSP (up to 100 cities); as observed in [34]: “It does appear that instances as small as 100 cities must now be considered to be well within the state of global optimization art, and instances must be considerably larger than this for us to be sure that heuristic approaches are really called for.” Therefore, in the following, experiments will be done using parallel genetic algorithms, and only on problem sizes of 100, 318 or 442 cities.[1] These are sufficiently difficult to tackle with parallelism.

Tanese found that the partitioned and distributed GA outperformed the canonical serial GA on a class of difficult, randomly-generated Walsh polynomials [53]. This left open the question whether they would also hold for other functions that were more amenable to optimization by a GA. To this end, the performance of a partitioned and a distributed GA was compared to that of the canonical GA on the travelling salesperson problem.

In the first experiments I tried the partitioned GA on a 100-cities problem. The GA was run a CM-5 with 16 processors. Remember from section 3.3 that in a partitioned GA no communication between the processors takes place. I did 16 simulations using a population size of 800 (i.e. 16 simulations of the canonical serial GA), 16 simulations using 4 subpopulations of 200 individuals each, and 16 simulations

1. The city problems used are from the publicly available collection of cities (partly with optimal solutions), compiled by Gerhard Reinelt (Institut für Mathematik, Universität Augsburg). The collection is available through ftp from titan.rice.edu (128.42.1.30) as /public/tsplib.tar.Z. The cities problems used in this research are contained in the files kroA100.tsp, lin318.tsp and pcb442.tsp.

using 16 subpopulations of 50 individuals each (thus making the total population size 800 in each simulation). Figure 4-5 plots the average of the best found (minimum) distances over these 16 simulations against the number of generations.

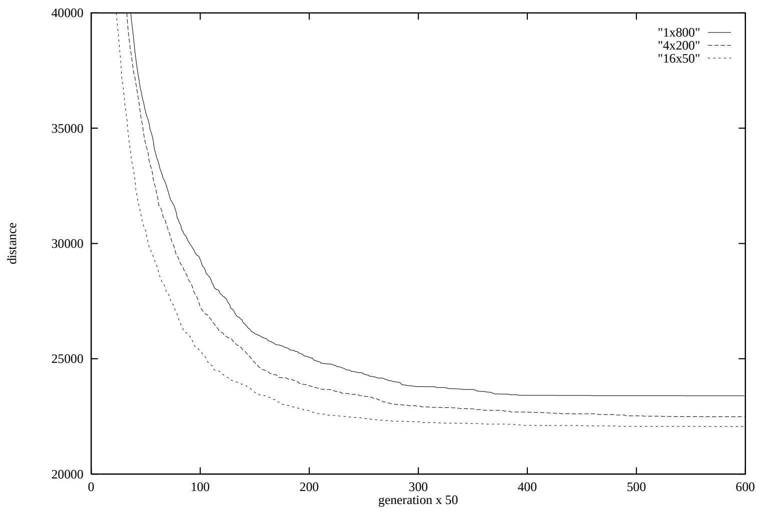


figure 4-5 *Average minimum distances plotted for a 100-cities problem.*

Several observations can be made. First, it seems that the partitioned GA outperforms the canonical serial GA. It can be seen that the more subpopulations are used the better the end result gets. So instead of running a GA with one large population it is more worthwhile to run the GA a number of times with a smaller population size and take the best result from those runs. Using the partitioned GA has another advantage compared to the canonical serial GA: because generational replacement is used, less recombinations and evaluations are done per generation. To explain why the partitioned GA gives better results in this case, take a look at figure 4-6 which shows the average distance of all the individuals in the total population plotted against the number of generations for the different simulations done.

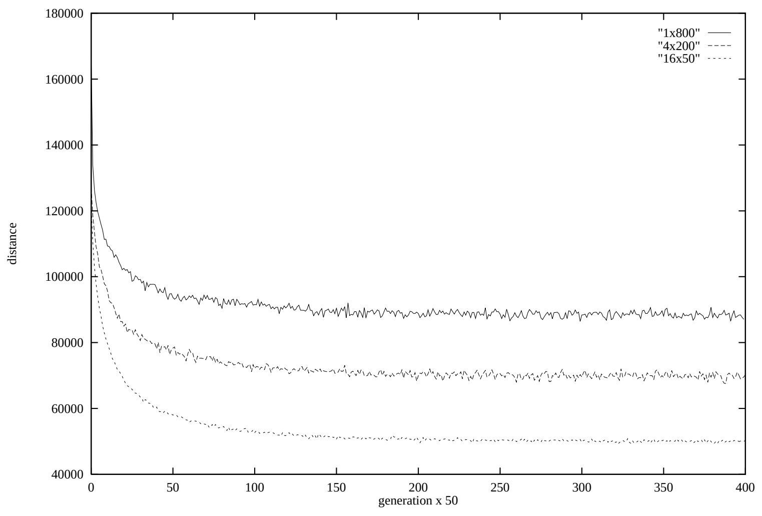


figure 4-6 *Average distances plotted for a 100-cities problem.*

As can be seen, the larger the (sub)population size, the more ‘garbage’ there is in the population. It is probably the tournament selection mechanism used in combination with generational replacement that is responsible for keeping larger amounts of ‘bad’ tours in the subsequent populations as the (sub)population size increases.

Then I tried the distributed GA. I ran 16 simulations using 16 subpopulations of 50 individuals each, and communication took place every 750 generations. The processors were arranged ladderlike as shown on the right side of figure 4-19. The results of the distributed GA were then plotted against the results of the partitioned GA, see figure 4-7.

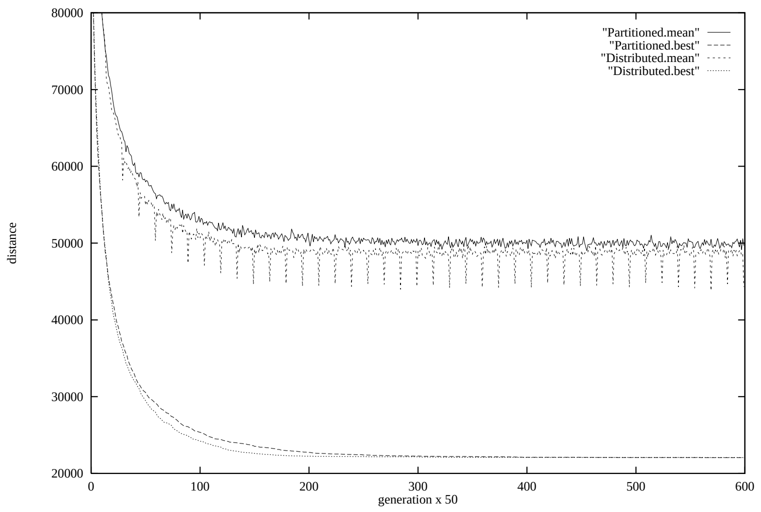


figure 4-7 *Best and average distances from the partitioned and distributed GA for a 100-cities problem.*

The distributed GA slightly outperforms the partitioned GA for some generations with respect to the best found tour and the average tourlengths, but eventually they converge to equal best distances. Notice the downward peaks in the average tourlengths for the distributed GA. At those points communication took place. Of course it would be preferable if the GA could keep the average low when it dropped after communication. Instead it jumps back to worse averages. It is most likely that the MX operator disrupts the GA process, making it impossible to improve any further.

Lamarckian evolution vs. Baldwin effect performed on the TSP

So far I have used tournament selection, elitist strategy, generational replacement, inversion, mutation and the MX operator together. I’ve done a lot of simulations with the 318- and 442-cities problems too, which I won’t present here, but generally it turned out that none performed as well as I would’ve liked (often the best found optimum was about 10% or more away from the best known optimum). It seemed that the MX operator on its own can’t reach the results Homaifar et al. presented in their article (they reported results within 2% of the best known optimum) [31].

However, Homaifar et al. specialized their GA variant by using 2-opt iteratively as a deterministic hill climber. In path representation 2-opt is nothing more than inversion. This *hybrid GA* thus combines local optimization with the simple GA. The strings produced by the genetic recombination operators are improved by local search, after which these resulting improvements are coded back onto the strings processed by the GA. This is equivalent to a form of *Lamarckian evolution*.

Local search in this context can be thought of as being analogous to the learning of useful adaptations by organisms during their lifetime. Not only the population as a whole is evolving, but also the individuals themselves are given a chance to evolve. Most biologists now accept that the Lamarckian hypothesis is not substantiated. Since in that case all that was learned by the phenotype is not communicated back to the genotype, some infer that learning, or evolving of individuals during their life-time, has no effect on the evolutionary search.

There is however another way in which learning can guide the evolutionary search, which has recently received new attention. Instead of coding the improvement of an individual back to its genetic encoding as in Lamarckian evolution, the fitness value of an evolved individual is determined as a function of its improvements. This has the effect of changing the fitness landscape, but the nature of this form of evolution is still Darwinian. This effect is known as the *Baldwin effect*, after Baldwin who first proposed the idea a hundred years ago that learned behaviour of organisms could influence evolution [2].

Figure 4-8, taken from Gruau and Whitley [26], illustrates how local optimization can alter the fitness landscape. N steps of local optimization deepens the basin of attraction, therefore making the landscape flatter around the local optima (minima for this example). When each individual always learns until it fully converges to a local optimum, then the landscape becomes flat in each basin of attraction. Each basin of attraction has a potentially different evaluation corresponding to the evaluation of the local optimum. Since hyperplane sampling is the basis for the claim that genetic algorithms globally sample a search space, changing the fitness landscape in this way has the potential for increasing the likelihood of allocating more samples in certain basins of attraction.

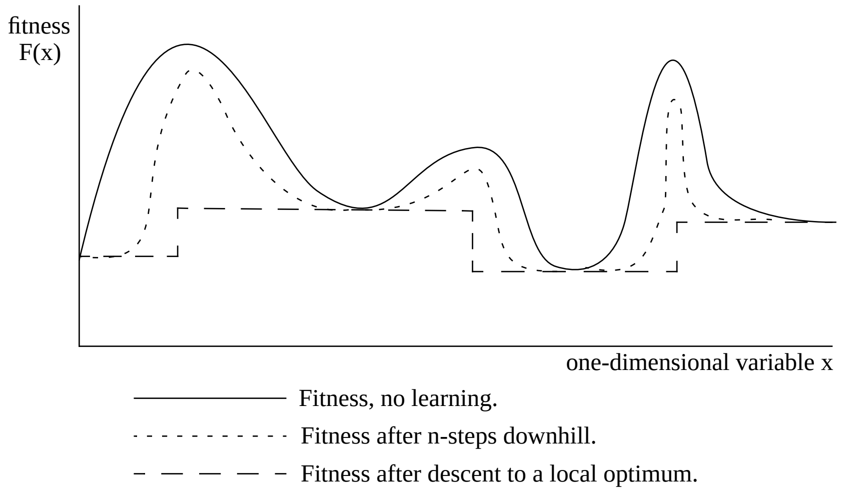


figure 4-8 *The effect of local search (or learning) on the fitness landscape of a one dimensional function. Improvements move downhill on the fitness landscape.*

The use of a hybrid GA, a combination of local optimization with simple GA, has been used by different researchers to solve the TSP. But I have not come across any article that uses the Baldwin effect on the TSP. So I thought I would try that out. I used the distributed GA as described earlier, and as local optimization of an individual the best inversion possible was done. First I tried this out on the 100-cities problem with 16 simulations. Figure 4-9 shows the average best tours found plotted against the number of generations for the plain distributed GA (where no local optimization was used), for the distributed GA when using the Baldwin effect, and when using Lamarckian evolution.

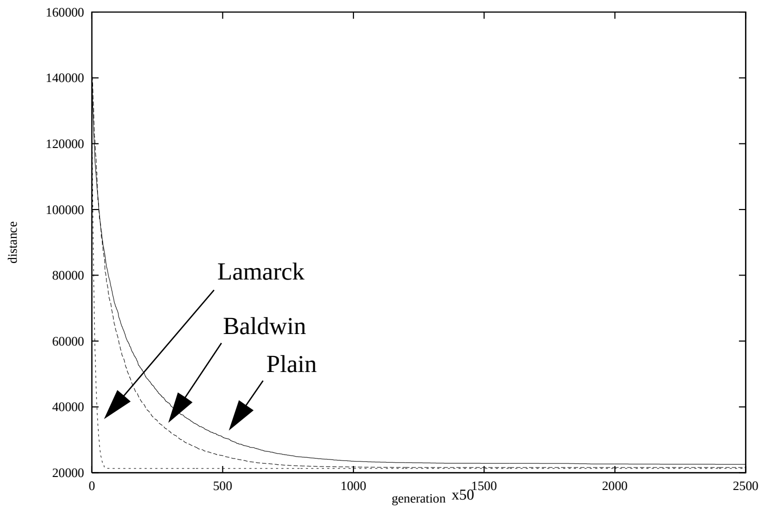


figure 4-9 *Performance of the plain PGA, Baldwin effect and Lamarckian evolution on a 100-cities problem.*

All eventually converge to equal minimum distances. However, the Lamarckian variant gets there within a few generations, while the two other variants need a lot more. During the converging process, the Baldwin effect gives somewhat better results compared to the effect of not using local optimization of the individuals.

I tried the same experiment using the 442-cities problem. However, this turned out to be very impractical, especially when using the Baldwin effect or Lamarckian evolution. I had to wait for days for one simulation to converge. This was due to the costly local optimization operator.

Therefore I changed it into doing the first N inversions that lead to improvement. Then I tried the 442-cities problem again. The variable N was chosen 15, the subpopulation size was chosen to be 16 individuals making the total population consisting of 256 individuals. For each variant I did 5 simulations. Figure 4-10 shows the average best tours found plotted against the number of generations for the plain distributed GA, for the distributed GA when using the Baldwin effect, and when using Lamarckian evolution.

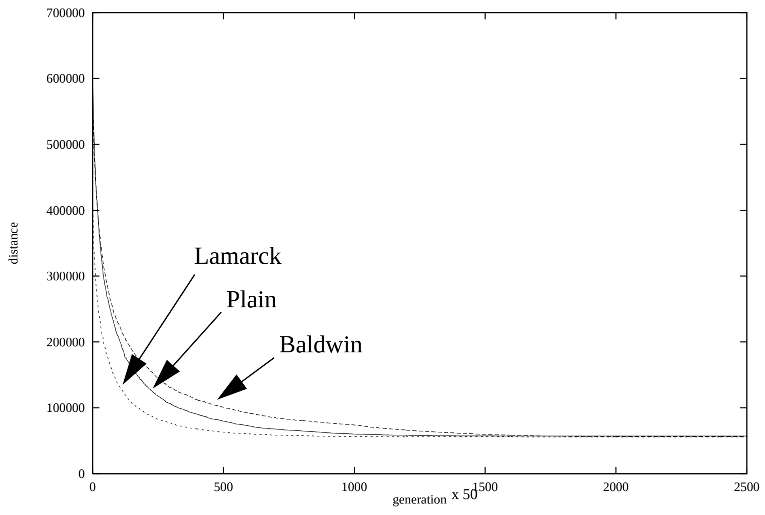


figure 4-10 *Performance of the plain PGA, Baldwin effect and Lamarckian evolution on a 442-cities problem, using 15-steps local improvement.*

Again, all eventually converge to equal minimum distances. Note however that with the use of a different local optimization operator the Lamarckian variant isn't as quick in converging any more compared to the previous experiments. Also note that now the plain distributed GA gives somewhat better results compared to the Baldwin effect during the converging process. Because of these two results I thought that maybe I could get the Baldwin effect and Lamarckian evolution perform better if I increased the variable N. Figure 4-11 gives an indication of what happens then, using the 318-cities problem this time.

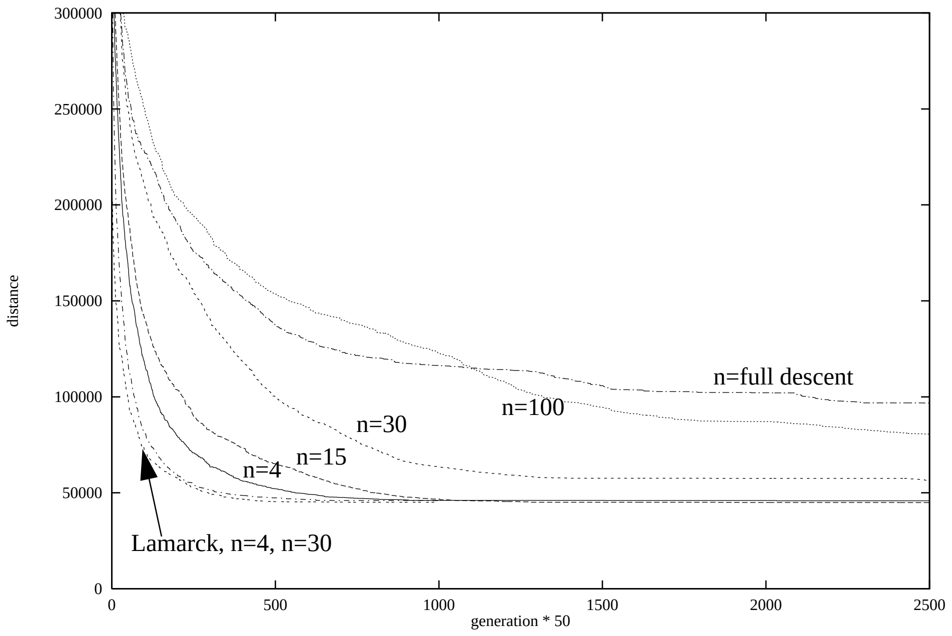


figure 4-11 *Performance of the Baldwin effect and Lamarckian evolution on a 318-cities problem, using various n-steps local improvements.*

The two lines the arrow points at are results from a Lamarckian evolution using N=4 and N=30; they perform equally well. The other lines are all from a Baldwinian simulation. The higher the value N is chosen the worse the results get. Clearly the Baldwin effect is not useful for the TSP. It is hypothesised that the Baldwin effect is only effective when the problem at hand is sufficiently deceptive in nature. The travelling salesperson problem isn’t deceptive at all.

Lastly, I want to present an optimal tour found for the 442-cities problem. This problem was originally designed by Grötschel. It represents 442 points non uniformly distributed in a square of area 11.4 square inches and issuing from a real world drilling problem of integrated circuit boards. The best known solution to date is 51.21 inches. Figure 4-12 shows an optimal tour found; it has a tourlength of 51.41 inches, which is 0.4% from the best known solution. Considering that no use was made of operators specially designed for this problem, and considering that Grötschel’s own solution was 51.45 inches, this is an excellent result.

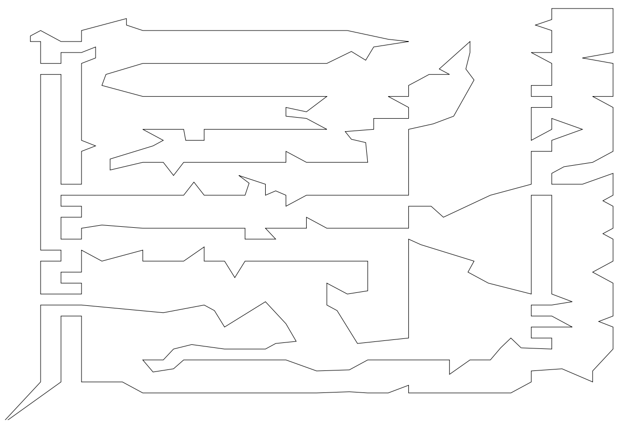


figure 4-12 *Optimal tour found for the 442-cities problem with length 51.41 inches.*

4.2 Schwefel's Optimization Problem

I also tested the effect of the Baldwinian and Lamarackian search strategies when applied to the numerical minimization problem of the following function proposed by Schwefel [52]:

$$F(x) = \sum_{1}^{n} -x_i \sin\left(\sqrt{|x_i|}\right) \qquad -512 \le x_i \le 512$$

This function is plotted in 1 dimension in figure 4-13. The global minimum is at x_i = 420.9687, i = 1,…,n. The local minima are located at the points $x_k \approx ((0.5+k)\pi)^2$, k = 0,2,4,6 and $x_k \approx -((0.5+k)\pi)^2$, k = 1,3,5. The second best minimum is at x_i = 420.9687, i=1,…,j-1,j+1,…,n and x_j = -302.5232, $j \in \{1,…,n\}$, which is far away from the global minimum. Therefore the search algorithm may be trapped in the wrong region.

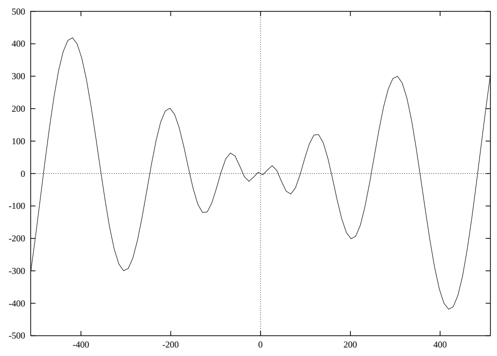


figure 4-13 *The Schwefel function plotted in 1 dimension.*

I first tried experimenting with a single population of 50 individuals. Each individual was coded with 20 genes (n = 20) stored in 16 bits each, of which the last 6 bits were discarded when de- and encoding. Each gene represented an x_i, and was gray-coded. Tournament selection was used with a pressure of 2. Two-point crossover on gene-level was used with probability set to 0.7, mutation rate was 0.004. No inversion was used. Generational replacement was used, and the elitist method was not used. A next generation is defined when 50 new individuals are created. When using local search the learning algorithm worked as follows: for x_1 calculate the one-dimensional Schwefel functions $F(x_1)$, $F(x_1 - \Delta)$ and $F(x_1 + \Delta)$ where Δ << 1.0. Determine which has the lowest value, and take that as the direction for further steps, which is to take at the most 4 times steps of 8 in the x dimension downhill in the y dimension. When a step does not result in an improvement local search is stopped (it has then arrived near a local minimum). This process is then repeated for all x_i, i = 2,…,n. Figure 4-14 shows the results until generation 3000. Each line shows the error of the best individual in the population, averaged over 48 runs.

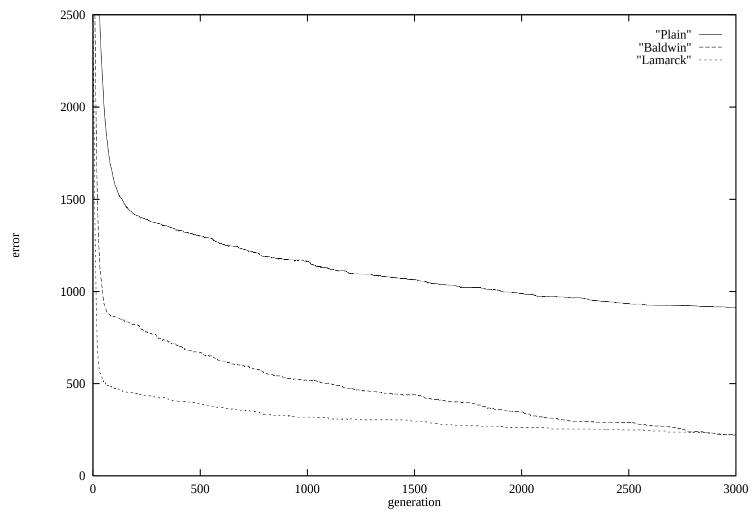


figure 4-14 *Performance of the Baldwin effect and Lamarckian evolution on the Schwefel function when using a population of 50 individuals.*

Figure 4-15 shows the same experiments for a population with 128 individuals. Here the mutation rate was set to 0.005, and the local search algorithm did at the most 2 times steps of 5.5 in the *x* dimension downhill in the *y* dimension for each *x*.

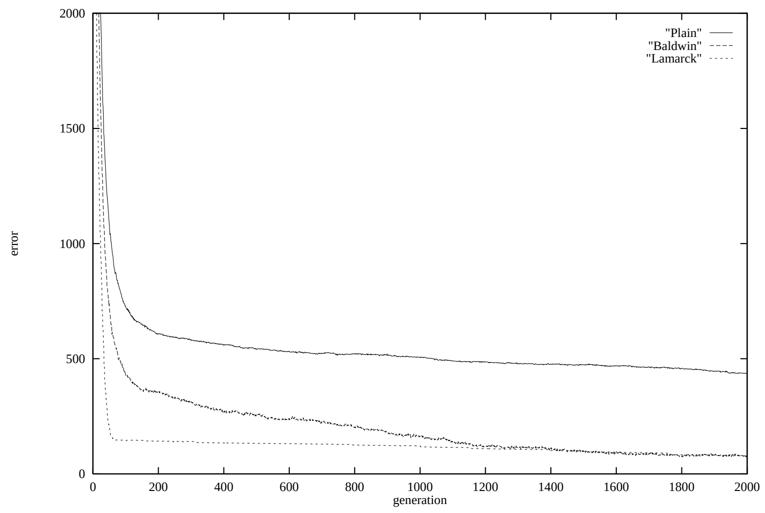


figure 4-15 *Performance of the Baldwin effect and Lamarckian evolution on the Schwefel function when using a population of 128 individuals.*

Then I did the same experiments as those from figure 4-15, but this time I used a parallel genetic algorithm. All the settings from the previous experiment were kept the same. The population was divided into 16 subpopulations of 8 individuals each, which makes a total of 128 individuals for the total population. The communication structure used was the ladder-like arrangement from figure 4-19. Migration period was set to 15 generations and the migration rate was 2 individuals. Figure 4-16 shows the result with again average best errors out of 48 runs. Note the saw-teeth in the error-lines. At the points the teeth go drastically down communication took place, after which the algorithm tried to settle down again in its usual behaviour.

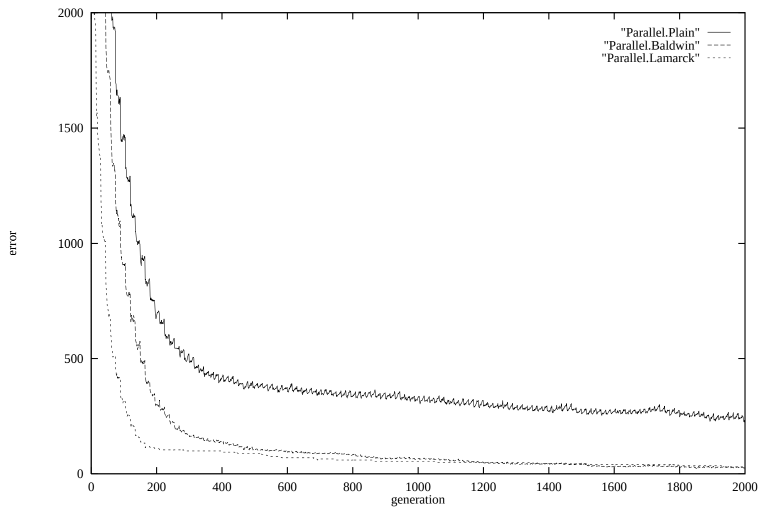


figure 4-16 *Performance of the Baldwin effect and Lamarckian evolution on the Schwefel function when using a population of 128 individuals and a parallel genetic algorithm.*

Figure 4-17 shows a blown-up picture of figure 4-15 but with the results from the parallel variant with the Baldwin effect. The latter cuts each error lines of the singular variants and converges to an error of almost zero.

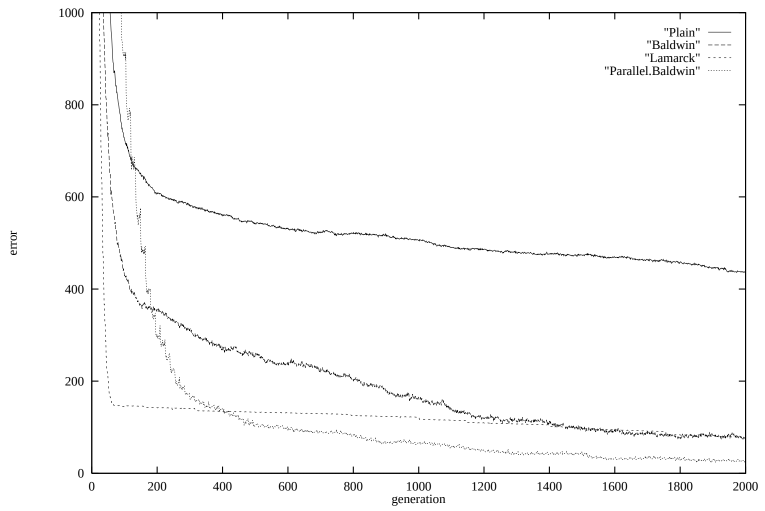


figure 4-17 *The Baldwin effect when using a parallel genetic algorithm plotted against the Baldwin effect and Lamarckian evolution when using a canonical genetic algorithm.*

For completeness, figure 4-18 plots all the results from the parallel genetic algorithm against the results from the singular genetic algorithm. Note that each parallel variant has a longer start-up time compared to its singular variant, but eventually always outperforms it with the error halved.

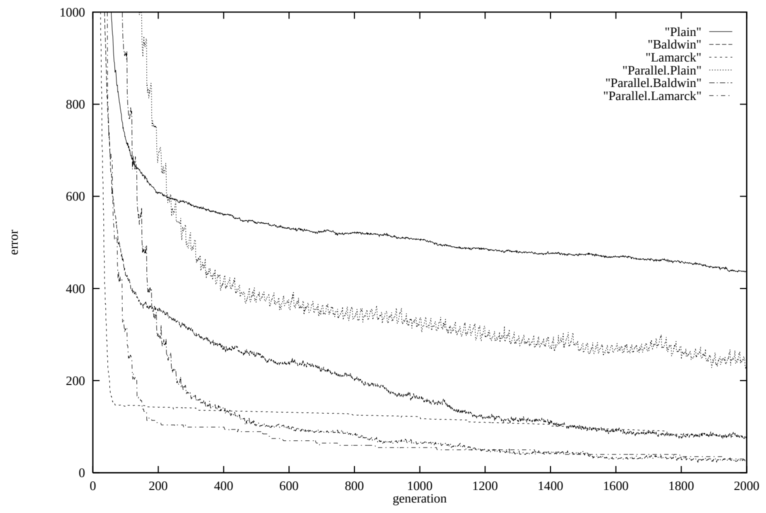


figure 4-18 *All results from the parallel genetic algorithms plotted against the results from the canonical genetic algorithms.*

Lastly, table 4-1 shows performance data for each algorithm, indicating the number of times the global optimum was found. Again, different local search algorithms were used for the populations with 50 and 128 individuals.

Table 4-1: Number of times the global optimum is found at different generations, out of 48 runs.

pop. size + method	50 GA			128 GA		128 PGA	
generation	1000	2000	3000	1000	2000	1000	2000
Plain	0	0	0	0	0	1	5
Baldwin	1	3	8	19	37	30	42
Lamarck	1	2	6	13	22	28	37

Note that the parallel genetic algorithm could find the global optimum without any local search in a number of cases after 2000 generations, whereas its singular variant couldn't. In fact, in all cases the parallel variant outperforms the singular variant. Also note the peculiar fact that although the error surface of the Lamarckian variant remains more or less flat after generation 1000, it still improves in terms of finding the global optimum. It is also interesting to see that even though the error surface of the Baldwin effect is above the Lamarckian error surface (except at the very end), Baldwin scores better in terms of finding the global optimum.

As a last note I have to say that the selection of the parameters, replacement scheme, genetic operators and local search algorithm to get the Baldwin effect perform better than Lamarckian evolution is somewhat of a black art. For instance, using the cross-over operator on bit level severely damages the Baldwin effect. Or when inversion is used, or 6-point crossover is used instead of the 2-point crossover, then the plain genetic algorithm is just as capable of finding the global minimum as any result I've shown above. Having a local search algorithm that learns too much or too little has the same effect on Baldwin as could be seen with the travelling salesperson problem in the previous section, thus making the choice of how much a local search algorithm should learn of importance. Also, when using Whitley's one-at-a-time replacement instead of generational replacement makes the Baldwin effect virtually ineffective. I've encountered all these side-effects, but due to time-limitations of writing this thesis I will not discuss these results in detail.

4.3 Communication Schemes

I wanted to know whether there is any neighbourhood relation between processors that leads to favourable results. To test this I needed a fitness function that was quick in its evaluation (which left out the travelling salesperson problem), and the fitness surface should be fairly complex (which left out the Schwefel function).

In this section a large class of functions called *Walsh polynomials*, which are based on Walsh functions [55] , are used for optimization. These can be used to design highly deceptive functions for a GA and to calculate the average fitness values of schemata. As a basis, these Walsh functions, which take values ± 1, are more practical than the traditional trigonometric basis like the Fourier functions. A good introduction to using Walsh functions in a GA context is given by Goldberg [18, 19, 20].

Walsh polynomials formalized

Walsh functions form an ordered set of rectangular waveforms taking one or two amplitude values: +1 and -1. They form a complete orthogonal set of functions and can thus be used as a basis. Consequently, any fitness function depending on *l* bits can be thought of as being made up of 2^l Walsh functions of various strengths. Whatever the function is, the rectangular waves are the same, so the only thing needed to describe the function are 2^l Walsh coefficients (strengths). Thus there are two equally valid ways of viewing a fitness function on *l* bits: 2^l string fitnesses, or 2^l Walsh coefficients.

A string of binary digits $x_l x_{l-1} \ldots x_1$ can be thought of as being the binary representation of a decimal number *x* where $x = \sum_{i=1}^{l} x_i 2^{i-1}$. A Walsh function of index *j* over a binary string *x* of length *l* is defined as follows:

$$\Psi_j(x) = \prod_{i=1}^{l} (-1)^{j_i x_i}$$

where $x_l x_{l-1} \ldots x_1$ and $j_l j_{l-1} \ldots j_1$ are the binary representations of *x* and *j*. Thus $\Psi_j(x)$ is +1 when $x \wedge j$ has even parity, otherwise -1. The set $\{\Psi_j(x) : j = 0, 1, 2, \ldots, 2^l-1\}$ forms a basis for the fitness functions defined on $[0,2^l)$. So formally a Walsh polynomial is a function *f* defined as follows:

$$f(x) = \sum_{j=0}^{2^l-1} w_j \Psi_j(x)$$

where w_j is the Walsh coefficient. Or in simple terms: to find the fitness of a string add up the effects of all the waves on that string.

A Walsh function of index j is of *order k* when the number of 1's in the binary representation of j is equal to k. There are $\binom{l}{k}$ order k Walsh functions. A Walsh function of order k defines a set of 2^k schemas. An *order k Walsh polynomial* is a Walsh polynomial in which the only non-zero Walsh coefficients are all associated with order k Walsh functions.

Experimental setup

In the experiments of this section binary strings of length $l = 32$ were used, and order 8 Walsh polynomials. Each order 8 Walsh polynomial was built up in the following way: choose randomly a point x_{opt} to be the global optimum, choose randomly 32 different indexes j of order 8 Walsh functions, and then assign to each of them a Walsh coefficient w_j as follows:

$$w_j = \Psi_j(x_{opt}) \cdot rnd(0, 5)$$

where *rnd*(0,5) is a random real value between 0 and 5. With a chromosome length of 32 it is enough to have 32 non-zero Walsh coefficients to ensure the existence of precisely 1 global optimum and to make the problem sufficiently hard to solve.

Fifty order 8 Walsh polynomials were generated randomly and were tested 5 times with a GA using the island model with different communication structures. The distributed GA was implemented on a CM-5[1] with 16 subpopulations of size 16, thus making the total population size 256.[2] Each run started with a different initial population. 2-Point crossover was used with a crossover rate of 0.7. Mutation rate was set to 0.005. Whitley-selection was used with a pressure of 1.5. As replacement scheme the steady-state model was used, where the size of the intermediate population was varied. Gray-coding was used, which has the advantage that a mutation of 1 bit has little effect on the value [20]. Each run did 5,000 evaluations.

The variables per testrun were:

- size of intermediate population; {2, 4, 8, 16}.
- migration rate; {2, 4, 8}.
- migration period; {50, 250, 500, 1000}.
- communication scheme {partitioned, ladder, circle, full, split, random}.

Communication scheme

Six different communication schemes were tested. They were:

1. *partitioned*: the GA runs on 16 processors, but there is no communication between the processors.
2. *ladder*: the processors are arranged ladderlike as shown on the right side of figure 4-19. Mühlenbein et al. suggested that this was a 'promising population structure' [42].

1. See chapter 7 for a description of the CM-5 with 16 nodes.
2. Because an order 8 Walsh function defines a set of $2^8 = 256$ schemas, each schema should have on average one instance in a population of 256.

3. *circle*: the processors are connected to each other so that they form a closed circle. Thus each processor is connected with only two other processors.
4. *full*: the processors are fully connected with each other. This means that if for example the migration rate is 4 and the migration period is 50, then after each processor has evaluated 50 strings it sends its 4 best strings to every other processor, but every processor accepts only the 4 best of all the strings received. Thus using the fully connected arrangement means that the 4 best of the total population are spread over all the subpopulations.
5. *split*: this neighbourhood relation is shown on the left side of figure 4-19. The processors are connected as shown. The number besides each connection indicates the migration period. So in the figure, processor A communicates with processor B after they've each evaluated 150 strings, and processor A communicates with processor C after they've each evaluated 2,375 strings. This arrangement can be seen as a form of islands within islands model.
6. *random*: when it's time for a processor to send its best strings, it chooses the processor to which it is going to send its individuals to randomly.

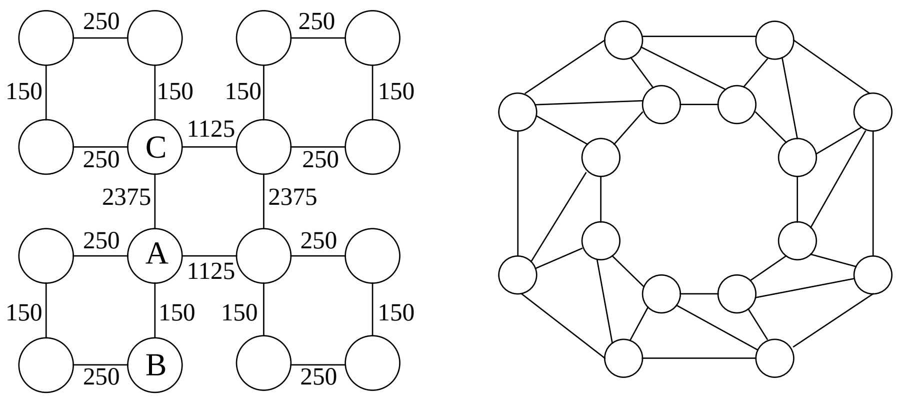


figure 4-19 *Islands within islands model, and the ladder-like arrangement.*

Results

With each experiment 50 order 8 Walsh polynomials were tested 5 times. Of each order 8 Walsh polynomial the best found solution of the 5 times was compared to the global optimum. Per experiment the average error and the standard deviation was calculated from the 50 differences. In total 208 runs were performed, taking five full days on a CM-5 with 16 processors. Table 4-2 shows the ten best and fifteen worst results, ordered by the average error[1]. The first column shows the neighbourhood relation used, the second the average of the errors. S.D. indicates the standard deviation. Members indicates the size of the intermediate population used, and the last two columns show the migration rate and migration period respectively.

1. The complete table can be found in Appendix A.

Table 4-2: Results from experimenting with different neighbourhood relations.

Type	Mean	S.D.	Members	Rate	Period
ladder	1.89	0.13	8	4	250
ladder	1.93	0.11	8	4	1000
ladder	1.94	0.13	16	4	50
ladder	1.95	0.12	4	2	250
random	1.97	0.10	4	4	250
circle	1.97	0.11	8	4	1000
circle	1.97	0.11	4	2	500
ladder	1.97	0.12	8	8	250
ladder	1.98	0.10	8	2	500
random	2.00	0.11	8	4	500
. . .	. . .	. . .	. . .	. . .	. . .
random	2.49	0.10	4	2	50
full	2.49	0.12	8	2	50
partitioned	2.51	0.10	2	n.a.	n.a.
partitioned	2.51	0.11	8	n.a.	n.a.
full	2.51	0.12	8	4	50
circle	2.52	0.09	2	4	50
random	2.52	0.13	2	4	50
partitioned	2.52	0.11	16	n.a.	n.a.
full	2.54	0.12	8	8	250
full	2.54	0.12	2	8	250
full	2.56	0.11	4	4	50
full	2.63	0.14	4	8	50
partitioned	2.64	0.13	4	n.a.	n.a.
full	2.66	0.14	2	4	50
full	2.68	0.12	2	8	50

Table 4-3 shows results obtained from using a canonical serial GA. In this case one global population was maintained consisting of 256 individuals. The size of the intermediate population was chosen from the set {2, 16, 32, 64, 128, 256}. All other variables and operators used were exactly the same as the ones used for the parallel GA. Six runs did 5,000 evaluations, six other 80,000 evaluations. Again the average error and the standard deviation is given for each run.

Table 4-3: Results from a canonical serial GA.

Max. Generation	5.000		80.000	
Members	Mean	S.D.	Mean	S.D.
2	5.38	0.18	3.13	0.17
16	5.71	0.18	3.07	0.15
32	5.64	0.17	3.04	0.17
64	5.80	0.16	3.07	0.16
128	5.71	0.16	3.12	0.15
256	5.37	0.19	2.90	0.19

Discussion

Several observations can be made from the tables 4-2 and 4-3. First, all 208 runs with the parallel GA significantly outperform the 12 runs done with the canonical serial GA, even when no communication takes place as in the partitioned model. A division of a population into several subpopulations therefore seems once again the better road to travel.

Also, of the 208 runs the 4 runs done with the partitioned model all end up in the worst 15 list. Thus, although the partitioned model significantly outperforms the serial model, in turn the distributed model significantly outperforms the partitioned model.

Looking at the list of the worst 15 runs one could think that the ‘full’ model isn’t good enough. This is not quite true. The reason those runs are the worst is because of the combination of low migration period, low intermediate population size and a high migration rate. Because of this combination the diversity of the total population is often decreased.

The ‘split’ model didn’t perform as well as I had expected. Probably the two connection weights of 2,375 between the two ‘continents’ of 8 processors was too high, thereby given the GA little chance to take advantage of this separation within 5000 evaluations.

The ‘ladder’ model seems the most promising, but the ‘random’ and ‘circle’ model can be just as good. When the migration period is set high, it is best to set the size of the intermediate population low. When the migration period is set low, the size of the intermediate population is best set to a high value.

Chapter 5

L-Systems

5.1 Overview

Patterns resulting from the growth and development of many living organisms can be simulated by a mathematical system called *fractals*. A fractal was defined by Mandelbrot to be any curve or surface that is independent of scale [39]. Thus a fractal is *self-similar*, meaning that when any portion of the curve is blown up in scale it would appear identical to the whole curve.

Growth involves the repetition of the same simple processes (e.g. branching) so that self-similarity arises on different scales. These repetitive processes that model growth can be captured as sets of rules and symbols, called *L-systems* (short for *Lindenmayer-systems*, after Aristid Lindenmayer [38]).

5.2 Simple L-systems

L-systems are sets of rules and symbols (a formal grammar) that model growth processes. A simple L-system contains three elements:

1. *Alphabet*, symbols denoting elements that can be replaced or remain fixed.
2. *Axiom*, a string of characters from the alphabet defining how the system should begin.
3. *Production rules*, define how the characters from the alphabet are to be replaced by other characters.

In formal grammars production rules are normally applied one-by-one sequentially. In L-systems however all symbols are rewritten in parallel to form a new expression. Consider the following simple L-system:

```
alphabet:  {A, B}
axiom:     A
rules:     A → B
           B → AB
```

The left-hand-side of the production rule (the part in front of the →) is called the *predecessor* and describes the string to be replaced. The right-hand-side is called the *successor* and describes the string with which it should be replaced. The process of replacing symbols stops when no production rule can be applied, or by placing an upper-bound on the number of steps. The simple L-system given above produces the following sequence of strings:

```
step 0:    A
step 1:    B
step 2:    AB
step 3:    BAB
step 4:    ABBAB
step 5:    BABABBAB
step 6:    ABBABBABABBAB
step 7:    BABABBABABBABBABABBAB
```

If we count the length of each string, we obtain the famous Fibonacci sequence of numbers:

```
1 1 2 3 5 8 13 21 34 ...
```

The power of L-systems comes when a predefined interpretation is assigned to the symbols. For instance a symbol might represent physical parts of a modelled plant, such as leaves and internodes, or it could represent local properties such as the magnitude of a branching angle.

Visualising expressions generated by an L-system requires that the symbols in the model refer to elements of a drawing on paper or on a computer screen. A common tool used for this purpose is *Turtle Graphics*, which appeared in the popular language LOGO [1]. A turtle provides a vivid mental model. It plays the part of the pen in a penplotter. Relative moves and draws of the pen are specified in terms of direction and a distance. Thus each symbol in an L-system can be interpreted as a turtle command.

Take for example the Koch-Graph, as proposed by Helge von Koch in 1905 [36]. It is one of the oldest examples of a fractal and can be described with the following L-system:

```
alphabet: {F, +, -}
axiom:     F
rules:     F → F-F++F-F
```

Suppose the following interpretations are attached to the symbols:

```
F : let the turtle draw a line for a distance d.
+ : let the turtle change direction by δ degrees to the left.
- : let the turtle change direction by δ degrees to the right.
```

The distance d and angle δ are variables. Figure 5-1 shows the result after 5 stages using an angle of δ=80°.

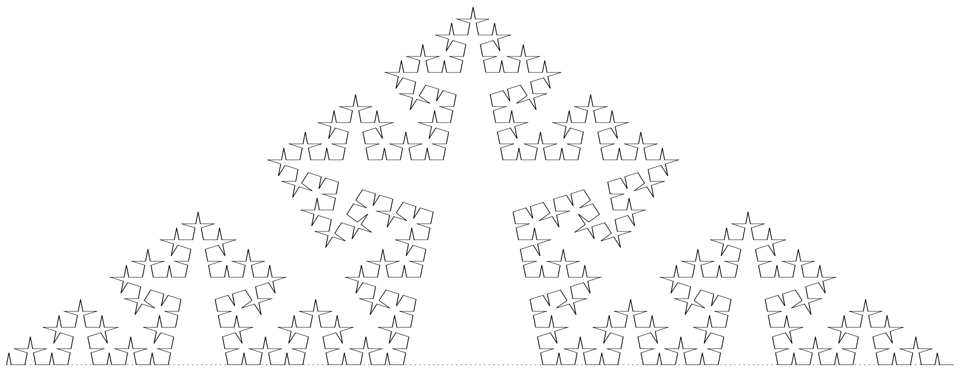


***figure* 5-1** *Koch-graph after 5 stages,* δ*=80°.*

To obtain branching as happens in real life plants two more symbols can be introduced with the following interpretation:

```
[ : Push the current position and direction of the turtle on a stack.
] : Pop a position and direction from the stack and let the turtle
    continue from there.
```

Matching pairs of the brackets [and] delimit branches.

5.3 Context-sensitive L-system

In the previous section information was transferred from parent to child symbols at the time of child creation. This is called *lineage*. In a growing plant information is also exchanged between coexisting adjacent cells, such as phytohormones, nutrients, or water. To model this *interactive* property the simple L-system is extended to *context-sensitive L-systems*.

Context is introduced to make sure certain conditions are met before a module is rewritten. A production rule in a context-sensitive L-system has the following form:

```
L < P > R → S
```

P is the predecessor and S the successor, both are mandatory. L and R (the *left*- and *right-context*) may be absent. P is replaced by S if P is preceded by L and followed by R. If two or more production rules apply for a certain module, the one with the longest context is used. When both contexts are of equal length, the first one in the list is considered. L-systems without context are called 0L-systems. When one of the production rules has one-sided context it is called a 1L-system. 2L-systems have one or more production rules with two-sided context.

Look for example at the following 2L-system:

```
alphabet:  {A,B,C}
axiom:     A
rules:         A     → ABC
           A < B     → A
           B < C     → BCB
           B < C > A → B
```

```
step 0:   A
step 1:   ABC
step 2:   ABCABCB
step 3:   ABCABABCABCBB
```

The plant in figure 5-2 was created from the 2L-system described below. The symbols 0 and 1 are not interpreted by the turtle. During context matching the turtle commands (-,+ and F) are ignored. So for example the string F1F1F1 is rewritten to F1F0F1. The production rules were constructed by Hogeweg and Hesper [30]. As can be seen from this picture context-sensitive L-systems lead to more natural looking plants.

```
alphabet:     {0,1,F,[,],-,+}
axiom:        F1F1F1
rules:        0 < 0 > 0 → 0
              0 < 0 > 1 → 1[-F1F1]
              0 < 1 > 0 → 1
              0 < 1 > 1 → 1
              1 < 0 > 0 → 0
              1 < 0 > 1 → 1F1F
              1 < 1 > 0 → 1
              1 < 1 > 1 → 0
                  +     → -
                  -     → +
```

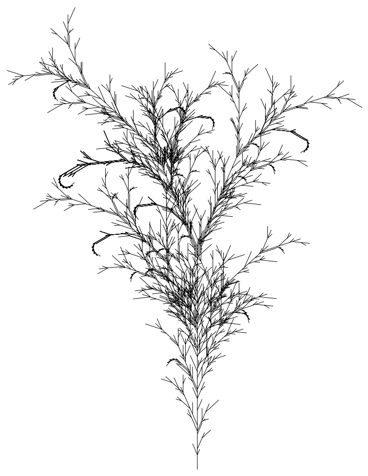


figure 5-2 *Axiom F1F1F1, production rules as above, δ=16°, 30 rewriting steps.*

5.4 Graph construction with a G2L-system

In this research a GA is used to manipulate a population of sets of production rules. Each member of the population is a binary string consisting of one or more production rules for an L-system[1]. To determine the fitness of the string an axiom is rewritten using the production rules the string contains. Because L-systems were

1. How these production rules can be extracted from a binary string will be shown later in section 8.1.

originally constructed to model biological growth, it is logical to use them when trying to describe the growth of the brain, where the brain can be seen as a modular neural network[1]. The resulting string is therefore interpreted as a modular neural network. A modular neural network is represented by a directed graph, see figure 5-3.

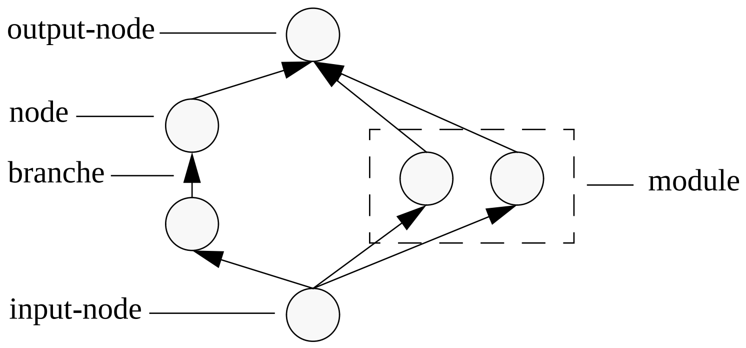


figure 5-3 *A directed graph*

Instead of using a LOGO-based interpretation of a string from an L-system, a suitable interpretation has been constructed to generate a directed graph directly from an L-system, the G2L-system [6].

The strings used are made of characters from the alphabet {A-Z,1-9,[,]}. The string is interpreted by reading it from left to right. A *node* in the graph is represented by a letter (A-Z). Nodes between square brackets form a *subnetwork* (or *module*). A digit *x* following a node or module is interpreted as ‘‘jump *x* nodes and/or modules to the right and make a directed connection to that node and/or module’’. Two adjoining modules do not have to be fully connected. *Output nodes* from the first module are connected to all *input nodes* from the second module. An input node has no input from other nodes within the module, and an output node has no output to other nodes within the module.

As an example the string A13[BC]1D1E represents the graph drawn in Figure 5-4. The string consists of 4 layers: the nodes A, D and E, and module BC. Node A is connected to module BC, and to node E which is the third layer after node A. Module BC is connected to the next layer which is node D. And node D is connected to node E. Note that module BC is connected to the same nodes outside the module, and that the nodes within the module are not connected to each other.

Not every string over the alphabet {A-Z,1-9,[,]} is allowed. The predecessor and successor may only contain nodes or complete subgraphs. The number of left and right brackets must therefore be equal to each other and have to be in the correct order. For example, the string A]BC[D is not correct because of the bracket’s order. Empty subgraphs ([ ]) are not allowed. The predecessor must contain at least one node. Thus, unlike in normal 2L-systems where only single symbols may be rewritten, in G2L-systems it is allowed to replace complete substrings. The successor may be omitted, which can be used to remove the predecessor from the string when the

1. Neural networks will be explained in chapter 6.

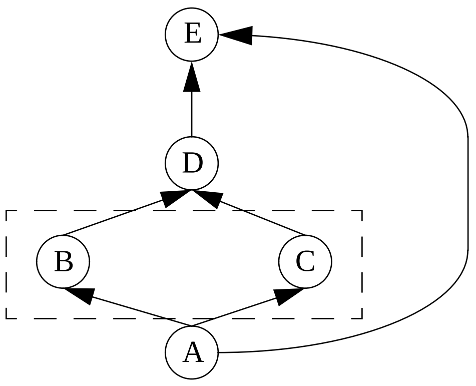


figure 5-4 *The graph from string A13[BC]1D1E visualized.*

production rule is applied. The left- and right-context of a production rule may also be omitted. But if one is present then the same restrictions hold for it as for the predecessor, with the added restriction that every digit must follow a node or a module. For example, the string 1A[BC] is not allowed as a context.

The definition of *context* in a G2L-system is also different from a 2L-system. In a normal 2L-system when the predecessor is rewritten into the successor the context is directly on the left and right side of the predecessor. In a G2L-system context is determined after the complete string has been interpreted.

A production rule with context is matched when the left context has edges *to* the predecessor and when the right context has edges *from* the predecessor in the graph interpretation of the string.

Look for example at the following G2L-system:

```
alphabet: {A,B,C,D,1,2,[,]}
axiom:    A
rules:        A     → B1B1B
              B > B → [CD]
              B     → C
          C < D     → C
              D > D → C2

step 0:   A
step 1:   B1B1B
step 2:   [CD]1[CD]1C
step 3:   [CC2]1[CC]1C
```

After step 3 no more rules apply. If a digit is contained in a module and the skip goes beyond the closing bracket, the skip is continued after the]. Figure 5-5 shows the result of each successive rewriting step. Note that the first D in the string from step 2 is replaced by C2 because it is connected to the second D in the string, and the second D in the string is replaced by C because it gets input from the first C in the string.

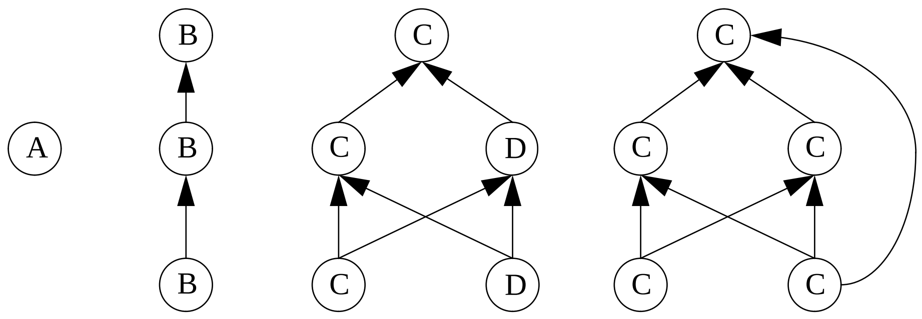


figure 5-5 *The successive rewriting steps in the growth of a graph.*

Chapter 6

Artificial Neural Networks

6.1 Overview

One of the most fascinating and challenging tasks in the history of humanity is probably to understand how the brain works, particularly in humans. A lot of effort is put into finding ways to make machines which show intelligent behaviour. The question "What is intelligence?" however seems unanswerable, or at least to be in the domain of philosophy.

Nonetheless scientists and engineers need criteria to determine whether a machine may be called intelligent or not. In 1950 Alan Turing proposed a test procedure to determine whether a machine shows intelligent behaviour [54]. This test has become known as the *Turing test*. To conduct this test a person who will play the role of an interrogator is situated in a room with two terminals, known as A and B. One terminal is connected to a person in another room, the other terminal is connected to the machine to be evaluated. The interrogator doesn't know whether A or B is the machine. It's his job to determine which of them is the machine by typing questions to the terminal. The machine's job is to fool the interrogator into believing it to be the person. If the machine succeeds at this, then Turing is willing to accept that the machine can think intelligently.

No computer has ever passed the Turing test. However, when we narrow down the domain of a complete imitation of a person to a more restricted domain some computer programs do pass the Turing test. These *expert systems* are able to compete with the best specialists in very narrow areas. For example, on the domain of chess-playing, computer programs have rankings comparable to the best chess-players of the world. In the medical field some expert systems make better diagnoses than specialists.

Expert systems designed to show intelligent behaviour usually make use of sets of rules. This however poses some practical difficulties: no person could ever supply a complete set of such rules, it would take too long and could certainly not be done without human mistakes; no program could easily handle all those rules, the response-time is too slow and there is a storage problem; there are areas lacking 'absolute knowledge' which can not be described by a definite set of rules. Therefore researchers look for other approaches.

One is to look at nature again, see the brain, understand it's basic workings and try to simulate that in a computer. The basic building block of a brain is the *neuron* which is a single nerve cell from the nerve system. Artificial neural networks are based on the way these neurons work. The advantage of artificial neural networks is that they acquire knowledge on their own. They do not have to be told what to do, but instead *learn* it by themselves. They are capable of autonomously discovering similarities, generalize automatically to novel situations, extract knowledge from examples in complex task domains, and they are tunable to changing environments.

6.2 Artificial Neural Networks

An artificial neural network (ANN) is based on the workings of a nerve system. To understand the functioning of an ANN it is therefore useful to understand the functioning of a single neuron.

The size and shape of neurons vary, depending on their type and their location in the nerve system. However, all neurons are basically similar in structure. As shown in figure 6-1 each neuron consists of a cell body, a single axon, and one or more dendrites. The neuron functions by means of electro-chemical signals, or impulses. When a sense organ or other receptor is stimulated, an electro-chemical impulse is set up in the dendrites of a neuron. The impulse passes rapidly in an electrical wave to the cell body and from there on along the length of the axon. After the passage of the impulse the neuron returns to normal, its resting state. Impulses from the axon of one neuron are transmitted to the dendrites of the next neuron at a point called the *synapse*. When an impulse reaches the end of an axon, the axon liberates chemical substances called *neurotransmitters*. The neurotransmitters cross the synapse, initiating an impulse in the dendrites of the next neuron. The neurotransmitters are then quickly inactivated by an enzyme, which prevents them from continuing to stimulate the neuron. It is believed that *changes* in the activity of neurotransmitters being released causes the processes of *learning* and *developing.*

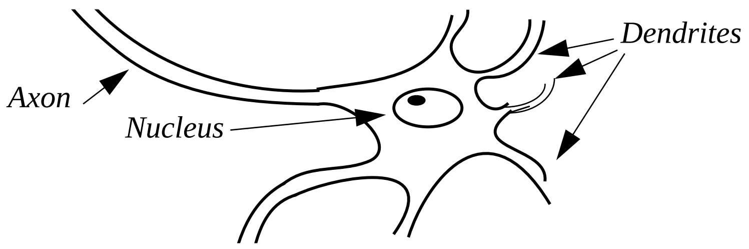


figure 6-1 *A Neuron*

In the artificial case, the artificial neuron takes its input as real numbers, and the output is calculated as a function of its input. Every input x_i has a corresponding weight w_i. A weight can be positive or negative, which is analogous to real neurons which give either excitatory or inhibitory signals respectively at their synapses. The stimulation of an artificial neuron is usually simply the sum of all its inputs multiplied by their weights, defined as:

$$stim = \sum_{i=1}^{n} w_i x_i + \theta$$

where θ is a bias term to shift the sum relative to the origin. The actual activation, or output, of the artificial neuron is usually defined as a function of the stimulation:

$$act = f(stim)$$

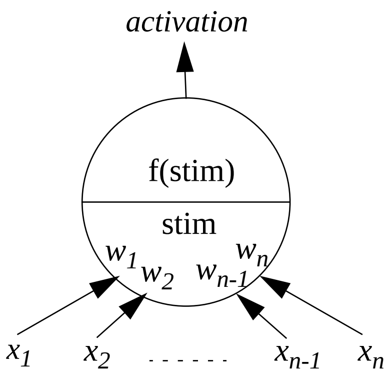


figure 6-2 *An artificial neuron*

Now that the functioning of an artificial neuron is defined and understood we can interconnect a number of them together in order to build an ANN. There are different ways to do this, but a standard one is the *feedforward* network paradigm. A standard feedforward network consists of 3 layers: an input, an output and a hidden layer. Connections between different layers go in one direction. In figure 6-3 for example there is a connection from node (or artificial neuron) A to B, which means that the output of node A is propagated as input for node B. A node can have any number of input- and/or output connections. A layer of nodes with no input-connections is called an input layer; this layer gets the stimulus from the ‘‘outside world’’. A layer of nodes with no output-connections is called an output layer; this layer gives the response to the ‘‘outside world’’. A layer of nodes with both input- and output-connections is called a hidden layer.

Recurrent loops are not allowed in feedforward networks. The output of a node is not allowed to return to the same node, or to a node in the same layer, or to a node in a previous layer. The input of a node always comes from a previous layer (except of course when it’s part of the input layer).

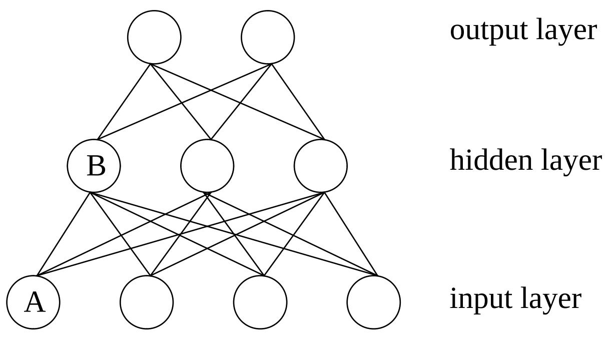


***figure 6-3** An artificial neural network* [1]

As with the human brain, an ANN has to be *trained* in order to perform a certain task. Rumelhart and McClelland described extensively how they used *supervised learning* as the method to train a feedforward network, which they then called a *backpropagation* network (BPN) [49]. In supervised training the network is repeatedly offered so-called *input/output pairs* until the network has converged toward a state that allows all the training patterns (the input/output pairs) to be encoded, or until the teacher gives up. Each input/output pair specifies the input the network will get and the desired output the network should produce. The input values are propagated through the network, until some output is produced. The difference between the produced output and the desired output is used to calculate an error for each output node. With these errors the internal connections between nodes are adjusted. These errors are back propagated through the network, until it reaches the input nodes. This way the whole internal representation of the specific problem is changed. When all input/output pairs have been offered to the network once, one training cycle has been done or one *epoch.*

Note that the representation of an item is not located at a particular place in computer memory as is the case in the conventional way. The representation of an item corresponds to a certain pattern of activity. Different items correspond to different patterns activity over the same group of nodes. A new item is ‘stored’ by modifying the interactions between the nodes so as to create a new stable pattern of activity. The main difference from a conventional computer memory is that patterns which are not active do not exist anywhere. They can be re-created because the connection weights between units have been changed appropriately, but each connection weight is involved in storing many patterns, so it is impossible to point to a particular place where the memory for a particular item is stored. For a more detailed description of how a BPN works the reader is referred to appendix B.

A fully connected BPN is of course a large simplification compared to a real brain, and some problems can arise because of this. A BPN may not converge to the global minimum error of the weight-space. Whether the BPN reaches this global minimum may depend on the random initialization of the weights. But even when the BPN

1. Throughout this thesis, all connections in the figures point upwards. The input nodes are at the bottom, the output nodes at the top.

does reach the global minimum the network may not be very useful, because it may be unable to inter- or extrapolate responses to input not seen during the training process. Another problem that may arise happens when the network is supposed to learn several unrelated problems at the same time. Even when the network is in principle large enough to learn all the problems at the same time, the problems seem to be in each other's way, causing *interference* throughout the weight-space. Not only BPNs suffer from these problems, most other neural network paradigms do too. There seems to be a solution to these problems, which is found in *modularity.*

6.3 Modular Artificial Neural Networks

Let's once again take a look at nature, at the human brain in specific. The brain is made up of around 10^{11} neurons. These neurons are not fully connected with each other. Instead the brain is divided in different modules at several levels, which in turn are divided in a number of functional areas. For example, the brain is divided in a left and a right half, which function for a large part independently from each other. The two parts are connected to each other through the *corpus callosum*, a module with a relatively small amount of connections. On a lower level there are a number of functional areas, like the visual, auditory and motory areas, between which relatively small number of connections exist. At a smaller scale than the functional areas more clear divisions can be made to show areas that each have their own specialism. All this suggests the brain is highly modular, meaning the brain can be divided in identifiable parts, each with its own purpose or function. Of course modularity is not only found in the brain, but almost everywhere in nature (e.g. leaves of a tree, hairs on the skin, scales of a fish, etc.). A good reason to also incorporate this feature in a BPN.

To allow modularity in feedforward networks, the full connectivity between adjacent layers has to be discarded. Instead of full connectivity between two layers, some connections are left out. Although a network with only one hidden layer can compute any (mathematical) function, more hidden layers can be added to allow for faster learning. Layers can be split into sub-layers, leaving specific connections out. This way different nodes see different things. The amount of computing is decreased, several local minima are removed which increase the speed of convergence, and mutual interference between the simultaneous learning of different tasks is minimized. Figure 6-4 shows a visualized example of a modular artificial neural network. A rectangular box is a module, a group of unconnected nodes, each connected to the same set of nodes. The number within the box shows the numbers of nodes in the module. Connected modules are fully connected. All feedforward networks can still be built with these components, when one considers that each node is a module on its own.

Studies so far have shown promising results when using modular artificial neural networks. It is stated that modular networks learn faster, generalize better, have a clearer architecture and are more suitable when there are hardware limitations to be considered. See also Boers and Kuiper [4], Rueckl, Cave and Kosslyn [48], Jacobs, Jordan and Barto [33], Murre [43], and Happel and Murre [27].

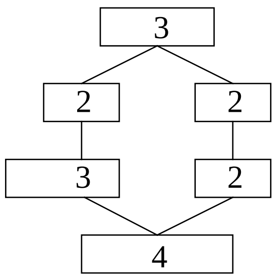


figure 6-4 *A modular artificial neural network.*

Chapter 7

Implementation

Because the amount of computing time needed to find good neural network architectures with the method of Boers and Kuiper, an implementation on a supercomputer was preferred. First I tried to port their code on a Parsytec transputer system (MIMD) with 16 processors, but this was not a simple task. First I had to translate C++ code to C, and even then it didn't work very smoothly. The communication capabilities on the Parsytec were not adequate enough. I also considered implementing the code on a MasPar MP-1 (SIMD) with 1024 processors, but this was not practical due to hardware limitations. Finally I got access to a CM-5 parallel computer with 16 processors, which has a very nice programming environment and the communication capabilities between processors are sublime.

First this chapter shortly describes the environment of the CM-5. Then some functions provided by CM are briefly described. After that the software developed is described, with some details highlighted.

7.1 Environment of CM-5

CM-5 stands short for Connection Machine and is produced by Thinking Machines Corporation. The CM-5 is a highly scalable parallel processing computer. The number of computational processors (or nodes) on a CM-5 ranges from fairly small to very large. For this thesis I had access to a CM-5 with 16 processors, each had 32Mb local RAM, and communication speed was about 10Mb/sec/node. A CM-5 provides for both space-sharing and time-sharing. The CM-5 provides both SIMD and MIMD capabilities. For this thesis use was made of the MIMD capabilities. Nodes on a CM-5 are assigned to partitions. A partition, or host, has access to all UNIX facilities that normally form part of the SunOS version of UNIX. Also access was given to special tools and utilities provided by CM software to facilitate parallel programming. The source code for a CM-5 message-passing program depends on the programming model in use:

- For a *hostless* program there is a single set of source code files for the nodes. In the hostless programming model, the host merely initiates execution of the node program, and thereafter acts as an I/O server for the nodes.
- For a *host/node* program there are two sets of source code files, one for the host and one for the nodes. The host program must explicitly start and monitor the execution of the node programs.

In this thesis the hostless mode as recommended by CM was used. Single node programs were written which ran on all the nodes. The program did all computation and communication; it did not communicate explicitly with the host. Each node executed its code asynchronously, fetching data and instructions from its local memory. It synchronizes with other nodes only when required to do so for message-passing purposes (e.g., to send or receive a synchronous message, or to participate in a global instruction, to do I/O, etc.). In the hostless mode an internal server program is run on the host. This program downloads the source code to the nodes, which then begin executing it, and goes into a polling loop as an I/O server, so that it can communicate with I/O devices on behalf of the nodes. This allows node programs to do I/O. The code that runs on the nodes performs all the normal tasks of an application program. Computation on each node is written normally. Communication among nodes uses special function calls. Communication can be point-to-point, when one node sends a message to a second node; or it can be global, with all nodes contributing to the message (and, usually, with all nodes receiving the result). Global communication synchronize all the nodes; point-to-point communications can be either synchronous or asynchronous. Hostless programming is supported in C, C++, Fortran 77, CM Fortran and C*; for this research C is used as the programming language.

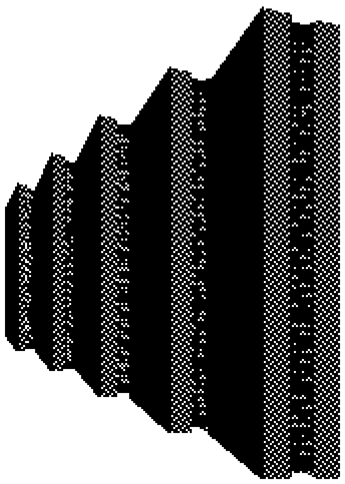


figure 7-1 *CM-5 from Thinking Machines Corporation*

7.2 CMMD library functions

CMMD is a library of message-passing routines for the Connection Machine CM-5 system. CMMD allows to send messages from one processing node to another in a number of different ways, depending on the need of the application. Here I will briefly sum up the minimal set of CMMD function calls I used. More details can be found in the CMMD Reference Manual.

Node information.

```
int CMMD_self_address (void);
```

returns an integer that represents the node identifier (the logical address) for the calling node. The system I used contained nodes 0 –15.

```
int CMMD_partition_size (void);
```

returns the number of processors in the current partition. For the system I used this was 16.

Input/Output.

```
int CMMD_set_open_mode (CMMD_file_mode_t io_mode);
int CMMD_set_io_mode (int fd, CMMD_file_mode_t io_mode);
int CMMD_fset_io_mode (FILE *stream, CMMD_file_mode_t io_mode);
```

These are the CMMD I/O functions I used. Most UNIX system and library calls are supported, but the nodes need to know how to treat the I/O. `CMMD_set_open_mode` changes the behaviour of UNIX `open()` and `fopen()` calls. The other two functions change the manner in which a global file is treated. The possible `io_modes` ares:

`CMMD_local`	`Default. Treats each file operation as purely local to the calling node.`
`CMMD_independent`	`Globally opens the file but allows each node to perform I/O operations independently on the file.`
`CMMD_sync_seq`	`Globally opens the file and synchronizes all I/O operations. Read operations spread data across all nodes in processor order. Write operations ouput data from all nodes in processor order.`
`CMMD_sync_bc`	`Gloablly opens the file and synchronizes all I/O operations. Read operations broadcast data to all nodes, write operations take data as if from a single node.`

To make debugging easy I used for example `CMMD_set_open_mode(CMMD_local)` so that each node could open its own files and `CMMD_fset_io_mode(stdout, CMMD_independent)` to be able to monitor single nodes on the standard output. When reading a global file onto all the processors this could easily be done by setting `CMMD_set_open_mode(CMMD_sync_bc)` so that there is no need, as on other parallel machines, to read the file on one node and then take care that the data is sent around the nodes correctly.

Synchronization.

```
int CMMD_set_global_or (int value);
int CMMD_get_global_or (void);
void CMMD_sync_with_nodes (void);
```

These are the three synchronization functions which serve to synchronize all nodes at a given point in a program. When using `CMMD_sync_with_nodes()` each node waits until all nodes have made this call. Only then all the nodes continue. In addition, two asynchronous global logical OR routines allow nodes to signal each other by contributing to a global logical OR and reading its results. The global OR mechanism can be used as a non-blocking method of determining when all processors have reached a given state. I used this to signal all the nodes when to stop running the genetic algorithm and start collecting data.

Asynchronous message passing.

```
int CMMD_send_noblock (int dest_node, int tag, void *buffer,
                       int buffer_desc);
int CMMD_msg_pending (int node, int tag);
int CMMD_msg_sender (void);
int CMMD_receive_block (int source_node, int tag, void *buffer,
                        int buffer_desc);
```

Once in a while a node sends a couple of individuals from it's population to another node. Because a node might be busy training a network, which takes a lot of time, sending nodes don't want to wait for a confirmation from a destination node that it has received the buffer. For this the `CMMD_send_noblock` is used. The function always returns immediately, having queued the buffer for later transmission when the destination node has declared its readiness to accept. For a node to check whether there is a message waiting to be received it can use the `CMMD_msg_pending` function; as parameters one can use `CMMD_ANY_NODE` and `CMMD_ANY_TAG`. When there is a message waiting to be received a node calls `CMMD_msg_sender` to determine from which node the message was sent. Then it can receive the message with `CMMD_receive_block`. This last function only returns when the message has been received and copied into the specified buffer.

7.3 Software used

Original software

The original software to search for networks consisted of a main program and three subprograms. They were:

- *genalg*, the main program which manipulates a population.
- *chr2gram*, translates the chromosome of an individual from the population into a set of production rules.
- *lsystem*, rewrites strings using the production rules for a number of iterations, resulting in an adjacency matrix representing a network.
- *backprop*, trains a network using backpropagation.

Backprop was written in C++. At the time I started this project I first had to translate *backprop* from C++ to C because the Parsytec didn't support C++. I also had to put the four different programs together into one program, therefore directing output from the different programs not to files but to memory.

Datastructures

The main program *pga* uses the library *Extended GenLIB*. The two main datastructures are described:

```
typedef struct            /* contains information for each member */
{
    float    fitness;     /* fitness of the member                */
    unsigned *genPos;     /* pointer to array of genpositions     */
    BYTE     **genValue;  /* pointer to array of chromosome       */
} MEMBER;
```

An individual from a population is described by a `MEMBER` structure, which contains the fitness of the individual, an array of genpositions necessary for the inversion operator to be able to switch genes, and an array of bytes containing the chromosomes. Each byte consists of 8 bits.

```
typedef struct            /* contains population info             */
{
    unsigned popSize,     /* nr of members in this population     */
             nrGenes,     /* nr of genes in each member           */
             genSize;     /* size of gene in bit                  */
                          /* (must be multiple of 8)              */
    MEMBER   *member;
} POPULATION;
```

Each population is described by a `POPULATION` structure, which contains the number of inidividuals in the population, the number of genes each individual has, the size of each gene in the individuals, and an array of individuals. To make computation easy only gene sizes which are a multiple of 8 bits are allowed. In this research `nrGenes` is used to indicate the number of chromosomes of an individual, and `genSize` is used to indicate the length of a chromosome.

Population maintenance

```
POPULATION *DefinePopulation (unsigned popSize,
                              unsigned nrGenes,
                              unsigned genSize,
                              BOOLEAN  initialize);
void FreePopulation (POPULATION *p);
```

A population is allocated memory by a call to `DefinePopulation`. The `initialize` parameter indicates whether the newly created population should be filled with randomly initialized individuals or not. Freeing the allocated memory is done by a call to `FreePopulation`.

```
Cmmd_LoadPopulation (POPULATION *p, char *popfile,
                    unsigned long *generation, int PID);
Cmmd_SavePopulation (POPULATION *p, char *popfile,
                    unsigned long generation, int PID);
Cmmd_LoadPopulation1 (POPULATION *p, char *popfile,
                     unsigned long *generation, PID);
Cmmd_SavePopulation1 (POPULATION *p, char *popfile,
                     unsigned long generation, PID);
```

Because one run may take days, it is useful to be able to save a population to disk once in a while. There are two ways to write and load a population from disk. The first two functions load/save each subpopulation from/into a separate file; this can be done asynchronously. The other two functions load/save subpopulations into/from the appropriate nodes from a single file; for this synchronisation of the nodes is necessary.

Genetic operators

```
unsigned Select (POPULATION *p);
unsigned RankSelect (POPULATION *p, double pressure);
unsigned TournamentSelect (POPULATION *p, unsigned pressure);
```

All selection methods described in chapter 2 are available; roulette wheel, rank based and tournament selection. The higher the pressure, the greater chance individuals with high fitness have to be selected.

```
RankReplace (POPULATION *p, POPULATION *newp, unsigned newmember);
```

This function is useful when the population needs to stay sorted all the time used in the steady-state model as described by Whitley.

```
CopyMember (POPULATION *p, unsigned member,
            POPULATION *newp, unsigned newmember);
```

To avoid the disruption of genetic information when performing genetic operators on the parents it is necessary to copy them into an intermediate population. This can be done by using the above function.

```
void Mutate (POPULATION *p, unsigned member,
             int variance, double pMut);
int BitCrossover (POPULATION *oldpop,
                  unsigned parent1, unsigned parent2,
                  POPULATION *newpop, unsigned child,
                  double pCross, unsigned point);
int BitInvert (POPULATION *p, unsigned member, double pInv);
```

These are the actual genetic operators on the chromosomes as described in chapter 2. `Mutate()` flips every bit in a chromosome with a chance of `pMut`. For the programs used in the experiments of chapter 4 functions `Crossover()` and `Invert()` were used. After crossover one offspring is produced. `Crossover()` supports a maximum of one crossover point per gene. `Invert()` swaps genes. For this program only one gene is used, where the gene is acted upon as a chromosome consisting of 1-bit long genes. For this special case the functions `BitCrossover()` and `BitInvert()` were written.

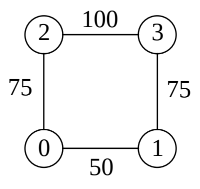
Communication

```
int Cmmd_ReadNetwork (char *network_file);
int Cmmd_Connected (int from_node, int to_node);
```

The first function reads a file into the memory of each node (processor). The file contains a specification of how the various nodes are connected to each other. Each connection is directed. With each connection a weight can be given. For Example:

```
0 1 50
0 2 75
1 0 50
1 3 75
2 0 75
2 3 100
3 1 75
3 2 100
```

gives the following network structure:



The function `Cmmd_Connected` () returns the weight between two nodes. When there is no connection it returns 0.

```
int Cmmd_SendBest (POPULATION *p, int migration_rate);
int Cmmd_SendRandom (POPULATION *p, int migration_rate);
int Cmmd_SendBest2Random (POPULATION *p, int migration_rate);
int Cmmd_SendRandom2Random (POPULATION *p, int migration_rate);
int Cmmd_RecvIndividuals (POPULATION *p, int migration_rate);
```

These functions take care of sending and receiving of individuals. `Cmmd_SendBest` () sends the first `migration_rate` best individuals from the population to the other nodes it is connected to. `Cmmd_SendRandom` () randomly chooses `migration_rate` different individuals from the population to be sent. `Cmmd_SendBest2Random` () sends the best individuals to one other randomly chosen node, and `Cmmd_SendRandom2Random` () sends randomly chosen individuals to one other randomly chosen node. Lastly, `Cmmd_RecvIndividuals` checks the buffer to see if there are individuals waiting to be received. If not, the function returns immediately, otherwise it will receive the individuals. At most `migration_rate` individuals are allowed to be received.

Miscellaneous

```
int Chr2Gram (POPULATION *p, unsigned member, char **production);
int Lsystem (char **production, unsigned nrofprod,
             char *axiom, unsigned steps, matrixType **adjMatrix);
float Backprop (matrixType **adjMatrix, unsigned nrNodes);
```

These are the main functions used to determine the fitness of a chromosome. `Chr2Gram` translates the chromosome into a set of production rules, and returns the number of production rules found. `Lsystem` () rewrites the `axiom` using the set of production rules for a maximum of `steps` steps, and translates the resulting string into an adjacency matrix. It returns the number of nodes of the matrix. `BackProp` () is the neural network simulator, and returns a fitness indicating how good the network is.

```
int GetProblem (char *problem_file);
```

Problem information for the neural network simulator was programmed hardcoded. With the use of libraries from Matt White it is now possible to read the problem specification from file. One call to `GetProblem` () in the main program is enough.

7.4 Main program

The basic structure of the program is given below. `May_send_individuals` () is a function which determines whether it is time for a node to send some of its own individuals to other nodes.

```
#define PSIZE           32    /* size of population on one node    */
#define NRMEMBER        10    /* size of intermediate population   */
#define NRGENES         1     /* number of genes                   */
#define CHROMSIZE       1024  /* chomosome length                  */
#define PRESSURE        1.5   /* selection pressure                */
#define PINV            0.5   /* inversion rate                    */
#define PMUT            0.01  /* mutation rate                     */
#define PCROSS          0.65  /* crossover rate                    */
#define SITES           2     /* number of crossover sites         */
#define MIGRATION_FREQ  50    /* migration period                  */
#define MIGRATION_RATE  6     /* migration rate                    */

float Fitness (POPULATION *p, unsigned member)
{
   nrofprod = Chr2Gram (p, member, production);
   nrNodes  = Lsystem (production, nrofprod, AXIOM, STEPS, adjMatrix);
   fitness  = Backprop (adjMatrix, nrNodes);
   return (fitness);
}

void main (void)
{
   POPULATION*  p;           /* population on this node            */
   POPULATION*  np;          /* new parents for crossover          */
   POPULATION*  localPop;    /* intermediate population            */
   unsigned     p1, p2;      /* the chosen parents                 */

   CMMD_set_global_or (0);
   p = DefinePopulation (PSIZE, NRGENES, CHROMSIZE, TRUE);
   np = DefinePopulation (2, NRGENES, CHROMSIZE, FALSE);
   localPop = DefinePopulation (NRMEMBERS, NRGENES, CHROMSIZE, FALSE);

   while (!CMMD_get_global_or())
   {
      /* Create new intermediate population                        */
      for(i=0; i<NRMEMBERS; i++)
      {
          p1 = RankSelect (p, PRESSURE);
          p2 = RankSelect (p, PRESSURE);
          CopyMember (p, p1, np, 0);
          CopyMember (p, p2, np, 1);

          BitInvert (np, 0, PINV);
          BitInvert (np, 1, PINV);
          BitCrossover (np, 0, 1, localPop, i, PCROSS, SITES);
```

```
              Mutate (localPop, i, 10, PMUT);

              localPop->member[i].fitness = Fitness (localPop, i);

              if (CMMD_get_global_or ())
                 break;
           }

           /* Put new individuals in the population sorted by fitness    */
           for (i=0; i<NRMEMBERS; i++)
              RankReplace (p, localPop, i);

           generation = generation + 1;

           /* Determine if it’s time to send individuals to other nodes  */
           if (May_SendIndividuals (generation, MIGRATION_FREQ))
              Cmmd_SendBest (p, MIGRATION_RATE);

           /* Check for individuals to be received from other nodes      */
           if (Cmmd_RecvIndividuals (localPop, MIGRATION_RATE) > 0)
              for (i=0; i<MIGRATION_RATE; i++)
                 RankReplace (p, localPop, i);

           /* If this node has found an optimum then signal all          */
           /* nodes to stop                                              */
           if (p->member[0].fitness > MAXFITNESS)
              CMMD_set_global_or (1);
         }

         Cmmd_SavePopulation (p, popfile, generation, PID);
         FreePopulation (p);
         FreePopulation (np);
         FreePopulation (localPop);
   }
```

The next chapter will present some results of experiments done with the software on the CM-5. After that a chapter is dedicated to a number of suggestions and ideas on how to improve and/or expand the software further on the CM-5.

Chapter 8

Seeking Architectures

8.1 Overview

Determining suitable neural network architectures for a given problem is most often like seeking a needle in a haystack. Instead of searching by hand, Boers and Kuiper used a genetic algorithm to automatically find these architectures [4]. Other researchers have tried this too, but most used the chromosomes of the genetic algorithm as *blue-print* representations for the neural networks. They could obtain good results for small networks but got stuck on the harder problems, which need larger networks, because the number of possible connections to be coded in the chromosomes grew exponentially. Boers and Kuiper designed a method that does not suffer from this problem. They used the metaphor of a *recipe*, where not the network itself is coded, but a set of rules that produce the network. However, they have not tested this method on any large problems yet. A combination was made of genetic algorithms, L-systems and neural networks.

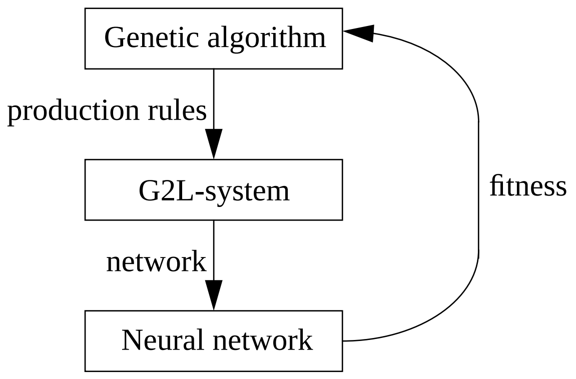


figure 8-1 *The combination of the three systems.*

The genetic algorithm generates a population of bit-strings, which are the chromosomes of the individuals. A chromosome codes recipes, or production rules. The set of production rules decoded from a chromosome are applied to an axiom for a number of iterations and the resulting string is then transformed into a neural network architecture by the G2L-system. The resulting neural network is then trained on a specific problem for a number of times. Depending on how well the network could learn the problem at hand a fitness value is determined. Low errors in the network results in a high fitness value. The fitness value is returned to the genetic algorithm, specifically to the chromosome in the population which coded the recipe that resulted in the network.

The G2L-system uses production rules of the form L < P > R → S, where every part of these rules is a string made of characters from the alphabet {A-H,1-6,[,]}. To separate constituent parts of the production rules within a chromosome a special character was used (the asterisk), therefore making the total number of characters that can be coded 17. In principle 5-bit strings can be used to code these 17 characters, but instead 6-bit strings were used. Thus 17 characters are related to 64 6-bit strings, which shows similarity with biological genetic codes where 20 amino-acids are related to 64 triples with four different bases. Table 8-1 shows the translation table used. The first two bits of a string need to correspond with the bits in the left column, the second two bits with the bits in the upper row, and the last two bits with the bits in the right column. For example, the genetic codestring 110100 corresponds to the first character B in the table.

Table 8-1: The translation table

	00	01	10	11	
00	4	[	D	]	00
	4	[	D	]	01
	*	[	3	3	10
	*	[	3	6	11
01	*	2	E	]	00
	*	2	E	]	01
	*	2	F	]	10
	*	2	F	]	11
10	3	A	G	[	00
	3	A	G	[	01
	3	A	H	]	10
	5	A	H	]	11
11	1	B	*	C	00
	1	B	*	C	01
	1	B	[	C	10
	1	B	[	C	11

Furthermore, a chromosome can be read in twelve different ways. By starting at one of the first 6 bits different production rules can be obtained by reading forward. The other 6 ways to read a chromosome are done by starting at any of the last 6 bits and reading backwards. This feature provides the genetic algorithm again with a much higher level of implicit parallelism than in traditional applications. Figure 8-2 shows an example from Boers and Kuiper where a chromosome of length 48 is decoded[1]. For clarity's sake only four of the twelve possible decodings are shown. The four decodings result in the following four rules:

```
**A*BB*D*
[1]1H][*
*D]B1[5*
1[*[HAE[1
```

Only the first of these four is a complete production rule. Rewritten in the usual notation it is read as:

```
A > BB → D.
```

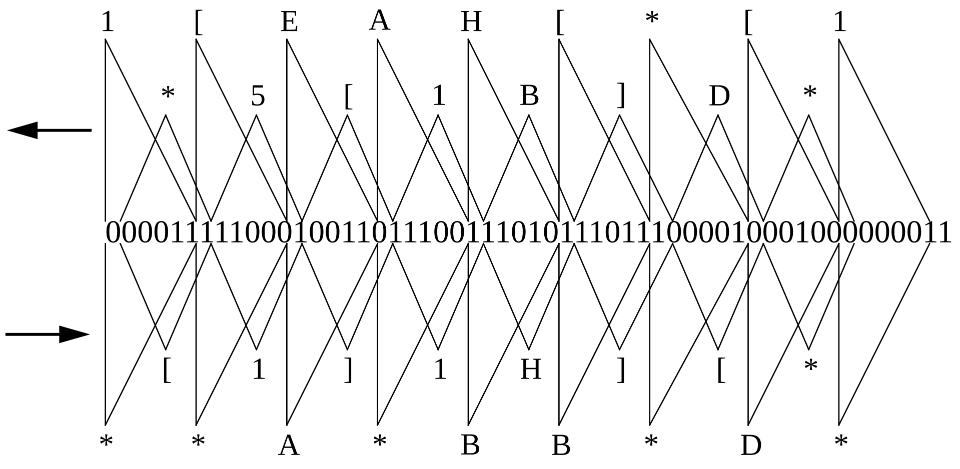


figure 8-2 *Extracting production rules from a chromosome.*

Finally, the software developed by Boers and Kuiper also contains several functions capable of repairing faulty strings. The functions remove strings with extraneous brackets, useless commas, succeeding digits, etc. in order for the strings to meet the restrictions of useful production rules. Repair mechanisms can also be found in nature, in living cells to correct mistakes in the replication of DNA.

1. In the experiments done in this research chromosome length is usually 1024 or longer.

8.2 Experiments

The XOR problem

The first experiment is a small problem where a network has to be found that can learn the logic exclusive OR function. The logical XOR function is a function of two boolean variables:

Table 8-2: the XOR function

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

In the late sixties Minsky and Papert proved that it is impossible to construct a network without a hidden layer that can learn the XOR function [40]. Because of this proof and since at that time it was unknown how to train a network with a hidden layer research in neural networks was for a long time thought to be not worthwhile. Even at the present the XOR problem is still used as the benchmark. Although I believe that nowadays the XOR problem receives too much attention in learning-speed studies I will present some results using this problem to show the potential of the methods described so far.

One of the parameters in the program is the variable *nrofiter*. This variable is used as the stop-criterion when training a network. It indicates how many epochs a found network should be trained, after which the error of the network is determined (and subsequently the fitness for the GA). The lower this value, the faster the evaluation of a network for the GA. Since more than 80% of the program is spent on training found networks I tried an experiment varying this parameter to see the effect on the speed of convergence of the GA and the quality of the XOR networks found. The value of *nrofiter* was varied between 75 and 1,000 epochs. The other parameters used in these experiments were:

```
#psize          50
#nrmembers      10
#nrgenes        1
#chromsize      1024
#pressure       1.5
#pinv           0.5
#pmut           0.03
#pcross         0.65
#sites          2
#steps          6
#axiom          ABC
#times_train    5
#neighbourhood  circle
#migration_freq 25
#migration_rate 4
#minimum_freq   1
```

The *times_train* variable indicates how often a found network is trained, after which the fitness is determined by averaging the last found errors. This makes the chance of a chromosome coding for that network receiving an erroneous fitness small. The *neighbourhood* variable indicates the neighbourhood relation used. The possible neighbourhood relations I used are described in section 4.3. In these experiments each processor is connected to two other processors to form a circle. Individuals are sent to neighbouring processors at time *g* when *g* is greater than *minimum_freq* and is a multiple of *migration_freq* and the best fitness in the subpopulation has changed during the time *g* - *migration_freq* and *g*. When the best fitness has not changed during that time and no individuals have been sent at time *g* - *migration_freq*, then individuals will be sent at time *g* + *migration_freq* anyway. Lastly, there is a parameter called `SUPERSNOEI` in the program which removes extraneous nodes (e.g. unused output nodes, chains of nodes) from a found network. In all experiments done this parameter was set to on.

Table 8-3: Results of varying *nrofiter* for the XOR problem.

#nrofiter	evaluations	trained	time (est. min.)
75	45,000	7,700	5
125	18,600	3,740	12
250	6,200	475	2
500	2,400	90	0.5
1,000	1,900	20	0.4

Table 8-3 shows the results. The values are averages of 5 runs, and should therefore only be considered as estimations. The column *evaluations* indicates the number of string-evaluations done by the GA, *trained* indicates the number of networks that were actually trained. The time shown is an estimation in minutes execution time on the CM-5. As one can see there's a big difference between the number of string-evaluations and the number of networks that were trained. This is explained as follows: A chromosome of 1,024 bits contains on average 50 production rules. But of these 50 rules none could be applicable to the chosen axiom. When this happens the fitness for the chromosome is set to 0. Even when it so happens that a production rule is used it often generates a string which translates into a network architecture which is beforehand considered to be not a good one (e.g. it has too few or too many nodes, or too few output-nodes for the task to learn). These last networks are also evaluated to fitness 0. Eventually at the most 15% of all the generated individuals are evaluated to a non-zero fitness. This percentage is calculated over an entire simulation. Usually the first couple of generations no valid networks are produced at all. The number of evaluated individuals increases with time.

In all cases good neural networks were found. Figure 8-3 shows some of the best neural networks found during the different runs. The number below a network indicates the value of *nrofiter* that was chosen during that particular run. The dark shaded nodes at the bottom are the input nodes, the dark shaded nodes at the top are the output nodes. In the right-most network the node at the bottom with the arced pattern is also an input node, one which always gets a default input-value of 0.0. For the three networks shown the error is plotted against the number of epochs in figure 8-4. It can be seen that lower *nrofiter* values result in more complex neural networks which learn the problem faster in terms of epochs needed to converge. It is important to take this result into consideration when trying larger problems. For larger problems the value *nrofiter* may need to be so high in order to give a network a chance to converge, that is not *practical* any more to use this value in the program. To make it practical, low values of *nrofiter* will have to be chosen. But this may then result in complex architectures, while it may be preferable that simple modular architectures are found that can learn the problem, even if it takes more epochs to learn the task. Simple architectures usually need more epochs to converge, but they also have less connections which means that they do one complete epoch faster. So it is a trade-off. Either you want complex architectures that converge fast in terms of epochs but maybe slow in terms of time per epoch needed, or either you want simple architectures that converge slow in terms of epochs but maybe fast in terms of time per epoch needed. In this thesis I do not care about how complex or how simple an architecture found is. I am after *practicality* and after network architectures that can *learn* the task after extended training. That means I will choose the value *nrofiter* low enough to be able to get a good enough idea about how well a found network converges, and therefore have a (hopefully) fair indication of how well it might be after extended training. It has to be noted though that larger networks increase the number of (local) optima, therefore decreasing the chance of good generalisation. So on the training set the neural network might learn the problem perfectly, but because of overgeneralisation it could fail on the test set.

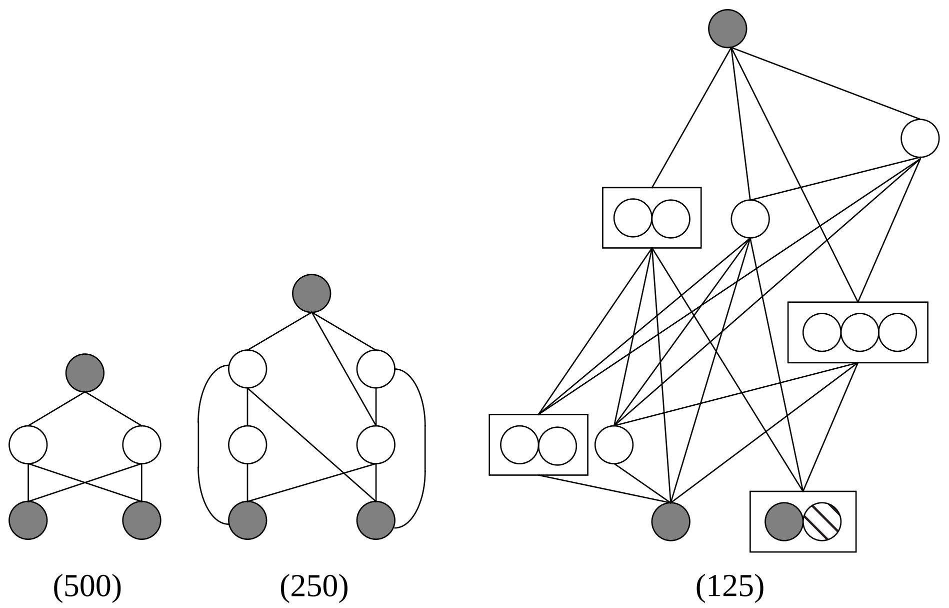


figure 8-3 *Some XOR networks found.*

The fitness of a network was simply the inverse of the summed squared error of the output node during the last epoch. The GA stopped when a network was found with fitness higher than 25 (which means an average error of 0.1 in the output node), or when a processor had done a maximum of 3,000 string-evaluations. When *nrofiter* was chosen above 500 the program quickly stopped with the left-most network from figure 8-3 as the best found. When *nrofiter* was chosen below 125 the GA stopped because the maximum number of evaluations was reached. It had then found networks comparable to the ones found when *nrofiter* is equal to 125; complex and with a fast converging error towards zero. But no network was found that could get the error low enough within 75 epochs. Note that the time needed when *nrofiter* equals 75 is considerably less compared to when it equals 125, even though more networks were trained.[1]

A closer look at the networks obtained from runs with *nrofiter* equal to 125 or less reveals that there are 6 layers of nodes in the best found networks. I tried another run with *nrofiter* equal to 75, but this time with the parameter *steps* set to 16 in order to allow networks with a maximum of 16 layers. In one run several networks were found that could indeed always learn the XOR function with an error less than 0.2 within 75 epochs. These networks contained from 70 up to 200 nodes. Although it seems ridiculous to use a network with 200 nodes to solve a simple XOR function, when this can also be done with just 5 nodes, these experiments show the potential power of searching for good neural network architectures using a genetic algorithm whose chromosomes encode production rules that make up the network.

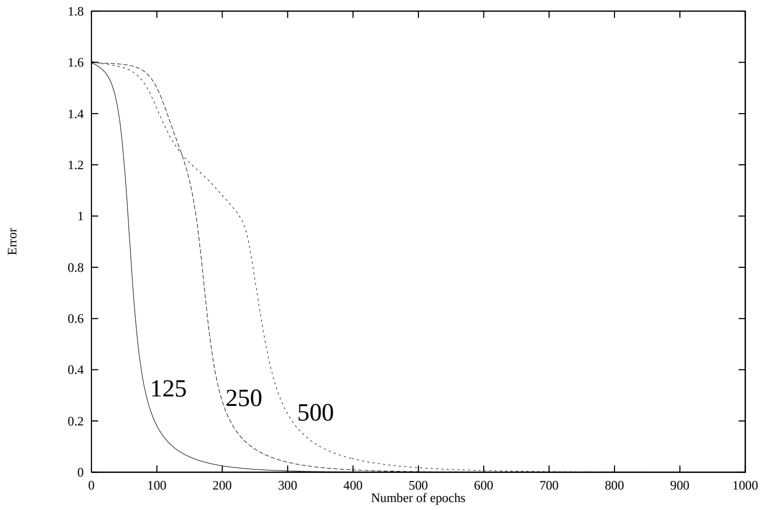


figure 8-4 *Convergence in the error of the three different XOR networks from figure 8-3. The numbers next to the curves indicate the value of* #nrofiter *with which the network was found.*

1. I also tried *nrofiter* equal to 50 and 100, which gave the same results as that for 75.

The TC problem.

With the TC problem, a neural network should be able to recognize the letters T and C in a 4x4 grid [49]. Each letter, consisting of 3x3 pixels, can be rotated 0°, 90°, 180° or 270° and can be anywhere on the 4x4 grid. The total number of input patterns is therefore 32 since there are 4 positions to put the 8 different 3x3 patterns. Figure 8-5 shows the possible 3x3 patterns and a sample 4x4 grid that would be presented as input to the network.

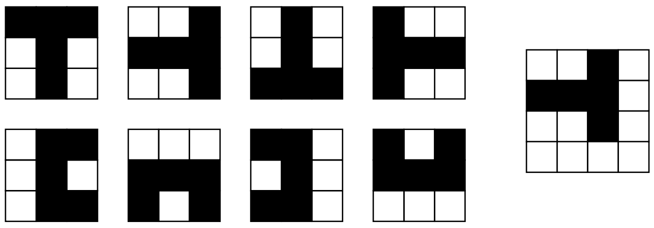


figure 8-5 *The 8 possible letters and one sample input grid of 4x4.*

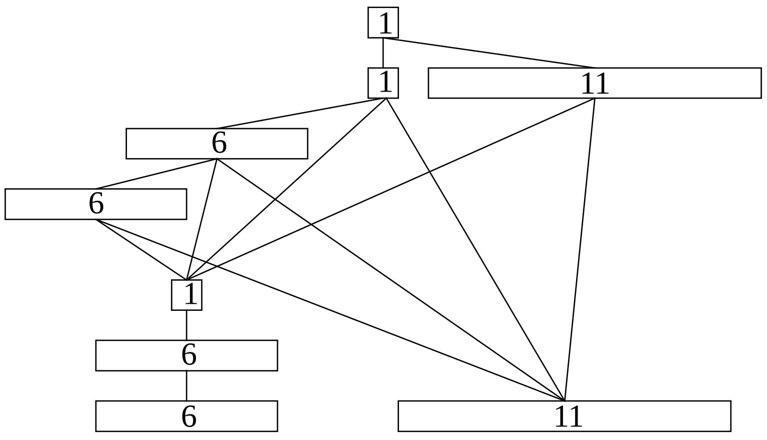
A black pixel was represented with an input value of 0.9, and white pixels with an input value of 0.1. The output node was trained to respond with 0.1 for a T and with 0.9 for a C. An approximation of the maximum possible error used to calculate the fitness of the network is:

$$error = \sqrt{0.9^2 \cdot 32} \approx 5.1$$

The network was trained for a number of epochs after which the network was tested to see how many of the possible 32 patterns were recognized correctly. When *k* patterns were recognized correctly, the fitness was calculated as 100 + *k* - *error*. So low errors resulted in a high fitness. The following parameters were chosen:

```
#psize           100
#nrmembers       20
#nrgenes         1
#chromsize       1024
#pressure        1.5
#pinv            0.5
#pmut            0.01
#pcross          0.65
#sites           2
#steps           6
#axiom           ABC
#times_train     1
#nrofiter        250
#neighbourhood   circle
#migration_freq  50
#migration_rate  10
#minimum_freq    70
```

I did two runs. Both ended with somewhat similar results. Figure 8-6 shows the best found network during the first run, together with the production rules responsible for the network architecture. It evaluated to a fitness of 128.73. The best found network in the other run had a fitness of 128.63. This means that 2 or 3 out of the 32 patterns were not learned correctly within 250 epochs.



```
          B → D[H[B]BFCB]2
     BC < D → A
```

figure 8-6 *The best network found during the simulation.*

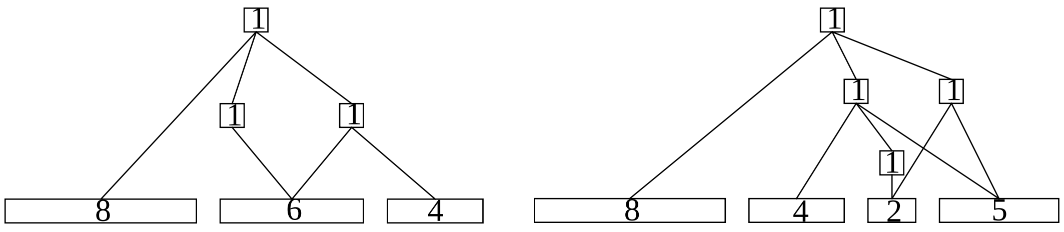
It is interesting to note that during the first simulation the following rules were also found:

```
B → D[H[B]BFCB]2
A → 3
```

It produced the same network as the one shown in figure 8-6. Since the axiom is `ABC` the rule `A → 3` is used once here, but doesn't really contribute to the end form of the network. In an earlier generation the following production rule was used:

```
A → D[H[B]BFCA]2
```

which shows great similarity with the previous production rule. It produced a simple network with an input layer of 36 nodes all directly connected to 1 output node. It evaluated to a fitness of 116.77. Although the two production rules look very similar to each other, the produced network architectures are totally different, as are their fitness values.



```
A → FHD[A]3FC                    A → FH[D[A]]3FC
```

figure 8-7 *Two other networks found during the early stages of the simulation.*

In the above two production rules the difference was in two letters. During the same simulation the two networks shown in figure 8-7 were found in an early stage. They too have production rules with great similarity; here the difference is in two brackets. Both networks evaluated to low fitnesses (112.95 and 112.68 respectively). In this case there is some similarity in the network architectures as well as in the fitness. In nature too, one simple change in the chromosome structure of a new individual can have no influence or a major influence.

Introducing *patience* variables.

During a typical run solving the TC problem, after 33,000 evaluations were performed, the times spent on the separate parts of the program were as follows[1]:

Table 8-4: Times spent per part of the program.

part of program	CPU time (sec.)	CPU time (perc.)
GA operators	140	0 %
Decoding from chromosomes to production rules	4,300	12 %
G2L-System	1,500	4 %
Training and evaluation of ANN	30,000	83 %
Communication operators	60	0 %

Of the 33,000 evaluations only 10,000 networks were actually trained. The rest was evaluated to fitness zero because the production rules did not produce valid networks. About 80% of the time was spent on training and evaluating valid networks. To train a network it was given a set of training input/output pairs. These pairs were presented to the network for a given number of times, which should be enough to give the network a chance to decode the problem correctly. When the network has been trained, the network is tested, and a fitness value is determined. Because the resulting weight-space of the network depends on the random initialization of the network, the network is reset, trained again, evaluated again, and a new fitness is determined. This process is repeated a number of times. After that the average fitness from the different runs is calculated, which will be the ultimate fitness given back to the GA.

To reduce the amount of time needed to train a network a *patience* variable was introduced. The idea was that a network could converge long before the stop-criterion was reached, which was a fixed number of times presenting the training set. The new stop-criterion was that the network would stop training when the change in the summed squared error of the output nodes did not increase or change more than a certain low percentage (*error_change*) for a number of epochs (*patience*). This

1. The time is rounded off, and shown in seconds of *execution* time on the CM-5. It does not include time during which the program was swapped out. The actual time is the execution time shown divided by 16 (since the CM-5 I used had 16 processors).

was tested on a problem to find good network architectures solving the TC problem, where we would stop when the error didn't change more than 1% for 20 epochs, or when the upper bound of 500 epochs was reached (i.e. *patience* = 20, *error_change* = 0.01 and *nrofiter* = 500). The result was that the amount of time for the GA to converge decreased dramatically. However, the resulting networks performed badly. Changing the *change_error* variable to 0.1% or even 0.01% and the *patience* variable to 100 did not change this poor result significantly. The question was how it was possible that only networks were found that perform badly, even after extended training, when the summed squared error of the output-nodes does not change in any significant way for a long period of time. To answer this it is a good idea to look at the development of the errors of a network. I compared the networks that I found with the *patience* variable set with networks I found during previous and next runs (without the *patience* variable set) that performed well after extended training. I just picked some found networks at random, trained them once, and plotted the error; see figure 8-8.

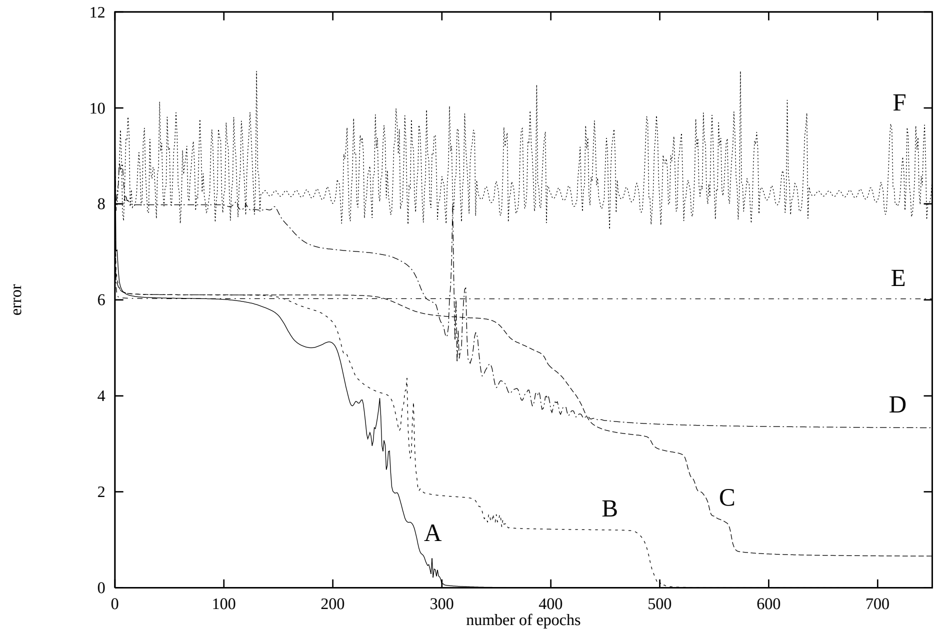


figure 8-8 *Error plotted against number of epochs for some networks found.*

Lines E and F show typical examples of networks found with the *patience* variable set; either the error is heavily oscillating at high values, or the error drops within 10 to 20 epochs to a certain (high) error, and from there on the line remains flat during the whole training. Lines B, C and D show error lines for networks that were found halfway through a simulation which had the *patience* variable not set. Line A shows the error development of a best found network with the *patience* variable not set. Looking at these lines it now becomes clear why the chance of finding networks that converge like A, B, C or D is low when the *patience* variable is set. After the error has initially dropped within 10 to 20 epochs all invariably remain at a somewhat constant error for over 100 epochs. What's more, they have during that period an error comparable to the worst converging networks. So when for instance network A is found, and the *patience* variable is set, then it gets evaluated to the same fitness as for example network E. This means that the GA will have no clue that network A is really better than network E, and therefore there's a chance that network A will be lost because it will not be selected as a parent for the next generation, and more rubbish like networks E and F are generated. It is also clear now why the results do not change when higher values of the *patience* variable are chosen.

Since it seemed that especially during the first number of epochs the error is not changing much, I tried another test where each network was forced to learn for a minimum number of epochs (*miniter*) before allowing the *patience* variable to be applicable. It improved the results a bit, but still the resulting networks performed mediocre. The reason for this is now also clear from the error lines in figure 8-8. When for example *miniter* = 200, *patience* = 30 is chosen then networks like C and D will still get bad fitness values. Network B will get a better fitness value, but it will be based upon the error around epoch 300. Even if it gets past this point it will stop at the plateau around epoch 400, and it will miss the evaluation of being a very good network that can learn the problem completely correct. Because networks don't get the credit they deserve they have a chance of being lost in subsequent selection phases of the GA. Choosing even higher values for the *miniter* and *patience* variables does not add anything new, since it will only mean that learning will stop at the original threshold of *nrofiter* we had at first.

Calculating the fitness of the initial population.

In the original version of the program the population was initialized with random chromosomes. These initial individuals were however not evaluated and their fitness values were simply set to zero. This meant that useful information that happened to be present in the initial population might get directly lost. In the program I made the addition that each initial individual is evaluated before the genetic process starts, therefore beginning with parents with correct fitness values. Practically speaking though, this did not seem to have any effect since most of the time all individuals get evaluated to zero anyway (because of the number of non-applicable production rules and/or invalid networks generated).

Making the primeval soup period as short as possible.

To give an idea of what the *primeval soup* theory entails I will quote some text from the thesis from Boers and Kuiper [4]:

> "The atmosphere of the earth at the time when there was no life on earth, contained no oxygen but plenty of hydrogen and water, carbon dioxide, and very likely some ammonia, methane and other simple organic gases. The 'primeval soup' theory states that this environment, under influence of lightning and ultraviolet light (there was no ozone layer), after thousands of millions of years spontaneously created some molecules that were able to replicate themselves, and subsequently started evolution [45]. Once started, evolution could slowly give rise to more and more complex creatures."

This is something that can be found in the simulations I did as well. It always took tens of generations before a valid network was found. As I already mentioned, of the newly created individuals only 15 to 20 percent at the most are considered as valid networks and get evaluated to a non-zero fitness. However, these percentages were calculated over an entire simulation. The start-up process is much worse. When there is nothing it takes a lot of time to create something spontaneously out of this nothing. Once there's something it gets easier to create more somethings each generation, i.e. more useful new chromosomes. Only then when there are individuals with non-zero fitnesses has the genetic algorithm really started with its schema processing. Boers and Kuiper presume that the results found by them so far can be compared only to the period just after the start of evolution, and that much longer simulations on much larger tasks perhaps require the full potential of their methods.

There is a problem with this however when using their methods together with a parallel genetic algorithm. It is very well possible (I've seen it happen each time) that one subpopulation after a small number of generations finds a valid network, after which an increasing number of better networks are found and put within that subpopulation. Meanwhile, all the other subpopulations are still busy in their primeval soup period searching for their first valid network. After some more generations the subpopulation that has already developed some good networks decides to send some of its individuals to neighbouring subpopulations. These neighbouring subpopulations, which still haven't found valid networks on their own (i.e. each individual still has a fitness of zero), are shocked by the amount of good networks they suddenly receive, and they start searching for better networks from these networks they have just received. But this is not what is desired. It only means that more subpopulations are now searching from the same subspace, i.e. this means that they are doing much of the same work and there is hardly any diversity between the subpopulations. Evolution could then get stuck on all subpopulations with one reasonable set of production rules, that was accidentally found on one specific subpopulation, and which can not evolve any further anywhere.

Therefore it seemed like a good idea to make the primeval soup period as short as possible. This could be done by making sure that the initial population, before starting the GA process, has individuals with non-zero fitness. I.e. only insert the next initial individual into a subpopulation when it has non-zero fitness. This way the simulation will start with a high amount of randomly found production rules that, when combined, may quickly result in good solutions.

I tried this, but in my judgement this was terrible and far from practical. When it takes for instance 1,500 randomly generated chromosomes to create one something out of nothing[1], and a subpopulation of say 100 individuals has to be filled, then it

1. This was in fact an average when applied to the TC problem. With an intermediate population size (*nrmembers*) of 20 it took around 70 generations on average before a network with a non-zero fitness was found. This was also the reason why I chose *minimum_freq* equal to 70 (with *migration_freq* = 50 this meant that a subpopulation would start its first communication at generation 100).

would take 150,000 evaluations before the genetic process can be *started* (and I'm only talking about one subpopulation here, so it would mean around 2.5 million evaluations in total when using 16 subpopulations). In the meantime I could have found some good working neural networks ten times when I just started the GA process with an initial population of individuals with zero fitness. I would rather take this last route and keep the best found network out of the 10 runs, even if this means that the end-result might have been a local optimum in evolution. Limiting the number of initial individuals with a non-zero fitness to say 10 still makes the start-up time excruciatingly long (in the order of hours).

To keep the primeval soup period as short as possible it is better to find ways where something can be created out of nothing faster. I'll come back to this issue in a later paragraph.

Introducing a database.

Since more than 80 to 90 percent of the program is spent on training neural networks it seemed like a good idea to create a database containing already trained network architectures together with their fitnesses. This would probably save a lot of computing power. Instead of storing the network architecture, a lot of space is saved when only the used production rules that evolved into the architecture are stored.

Setting up an effective and efficient database requires more time than I can afford to spend on this research. Therefore I decided to set up a database in a simple and crude form just to get a rough indication of the implications on the speed. In the program I kept track of which production rules were used during the rewriting steps in the G2L-system. At the beginning of each new generation the database is emptied and each individual of the best half of the current subpopulation is decoded into production rules. These are given to the G2L-system, which returns a string containing the production rules used. This string together with the fitness of the individual was put into a small database. Then when the intermediate population was being filled the production rules used by a newly created individual were first looked up in the database. If a match was found, the new individual got the corresponding fitness. If no match was found, the network architecture was trained.

Using a database in this form means that at each time it only contains network architectures equal to the best half of a current subpopulation. Network architectures that might have been trained in the past can be lost. Also, since each subpopulation maintains its own database it is possible that one subpopulation is training a network that is contained in the database of another subpopulation.

When I first tried the program with the database a new problem introduced itself. An individual with high fitness can spread itself throughout the population, which means that when no database is used this individual is evaluated over and over again. So for instance, when the *times* variable is set to 1 as I did when solving the TC problem, one particular network could be trained more than once. This means that this network gets more chances to get a higher fitness, since other initial random weights of the network might be directed towards lower errors in the network. When a database of evaluated production rules is used a particular network is evaluated once and only once. This means that if the network started with bad initial random weights it gets a lower fitness than it deserves. When using the database the *times* variable should therefore be set to higher values to average out the error of a network. This of course means an extra large amount of computing time.

In the experiments I will present hereafter I will each time give three values, which I call: *invalid*, *database* and *trained*. In order to discuss results pertaining to the addition of a database I will start with presenting some example numbers, see tables 8-5 and 8-6 (the values are rounded off). These are results from a rerun of the TC problem with the same parameters as I started with, but with variable *times* set to 4 and the database installed. The GA had fully converged. A newly created individual is marked as *invalid*, *database* or *trained*, depending on how the fitness was determined. It is marked as *invalid* when the individual was evaluated to a fitness of zero (i.e. no network architecture could be produced with the production rules, or the network architecture produced was considered invalid), and it is marked as *trained* when the network architecture resulting from the individual's production rules was trained and therefore was evaluated to a non-zero fitness. An individual is marked as *database* when it got its fitness from the database, thereby bypassing the need to train the network.

Table 8-5: Values showing how the fitness was determined.

invalid	365,000	73 %
database	111,000	22 %
trained	25,000	5 %
total evaluations	501,000	100 %

Table 8-6: Times spent per part of the program.

part of program	CPU time (sec.)	CPU time (perc.)
GA operators	550	0 %
Decoding from chromosomes to production rules	38,000	6 %
G2L-System	36,000	6 %
Training and evaluation of ANN	445,000	76 %
Communication operators	160	0 %
Database operators	66,000	12 %

As can be seen from table 8-5 about 75% of all created individuals is marked as invalid. Around 20% got its fitness from the database, and only 5% actually got trained. This 5% takes about 75% of the time of the program, as can be seen from table 8-6. Had there not been a database it can be deduced that the total time for the GA to converge would take about 4 to 5 times longer, and about 95% of the time would've been spent on training networks. Note in the above the time needed to create the database each time. Since I used just a crude form of a database this time needed to perform database maintenance could easily become substantially less when more efficient database operators are installed. However, for my purposes it served me enough since the time is not really significant in comparison with the time to train a network. Of course the database can also be made more effective, which means cutting down even more on the time needed to train networks.

The reader should also note the following: when the variable *times* is set to 1 and no database would have been used, and this is compared to the above where the variable *times* was set to 4, then it could be deduced from the above tables that the total time spent on the program would be somewhat equal for the two runs. It is therefore doubtful whether a database will be of real help.

Adapting the translation table.

During one generation newly created chromosomes produce on average 50 valid production rules, but very often none is applicable to the axiom. Even when some rules are used during the rewriting steps of the G2L-system, often network architectures are produced that will not even get trained, due for instance to too few nodes, too many nodes, too few output nodes, etc. Wiemer researched to increase the number of applicable production rules by adapting the translation table 8-1 [59]. She used a different evaluation function though. Produced network architectures were not trained. Instead the fitness was simply a function of the average number of nodes in a module, the number of modules and the average number of connections between modules.

One of the experiments she tried was to replace some numbers and letters in the translation table by asterisks. It turned out that this resulted in an improvement in her experiments. I also tried this for the TC problem. I replaced 5 numbers and 2 letters in the translation table by asterisks. This did not seem to improve much[1] (best found networks had fitnesses of 125.41 and 126.56).

Wiemer also introduced a new character, the point. In the successor the point was ignored and filtered out of the string. In the left part of the production rule the point was interpreted as an asterisk. This way the chance for matching production rules was increased and the successor wouldn't be too small. The best results were produced when she combined adding more asterisks and more points to the table, as the one shown in table 8-7.

1. Each time I tried a new translation table I did two runs. This took a complete day. It is hard and maybe even unjustifiable to base assumptions on just 2 runs. Nevertheless, I have no choice in this due to time limitations, and therefore I can only give observations biased by my subjective interpretation. However, even with 2 runs I think I can make reasonable judgements based on the results about how promising a translation table may be. This is based upon a number of factors, such as the best found network and its fitness, the time of the primeval soup period, the average number of production rules that are applicable, the number of valid networks found per generation, and the convergence speed of the GA.

Table 8-7: Translation table used by Wiemer [59].

4	4	*	*	[	[	[	[	D	D	3	3	]	]	*	6
.	*	*	*	.	2	2	2	E	E	F	F	]	]	]	]
*	3	3	5	.	A	A	A	G	G	H	H	[	[	]	]
1	1	1	1	.	B	B	B	*	*	[	[	C	C	C	C

The shaded symbols show where changes have been made to the original translation table. These changes had the effect of increasing the convergence speed of the GA, and the number of fitness evaluations turned out to approximate a normal distribution. Because the fitness function used was an alternative one, I tried to experiment with the given translation table to see its effect on the original fitness function which trains the networks. The results did indeed show a slight improvement. The number of applicable production rules clearly increased, and the time of the primeval soup period decreased on average on the subpopulations by several generations (originally it took about 70 generations before the first valid network was found; with the addition of the points it took about 20 to 30 generations). The fitnesses of the best found networks were somewhat higher (129.89 and 130.11). Still, the number of valid networks produced that could be trained did not increase as much as I would've expected considering the increase in applicable production rules, and the convergence speed also didn't increase that much. This can perhaps be explained by the fact that Wiemer accepted every network architecture created, while I only consider network architectures with a certain minimum and maximum amount of input- and output-nodes. It seems that the adaptations of the translation table used by Wiemer are just not enough yet to overcome the constraints I put on the form of a network architecture.

To increase the number of applicable production rules even more I tried another experiment where I replaced the two letters `'H'` by the letter `'A'` (the axiom chosen was `'ABC'`). This didn't show any further significant improvement. Then I replaced in addition to all previous adaptations the two letters `'G'` by the letters `'A'` and `'B'`, and replaced the numbers `'5'` and `'6'` by a point thereby making the maximum jump over modules 4; see also table 8-8.

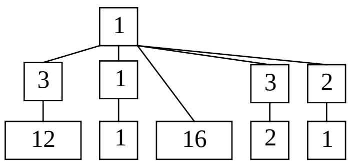
Table 8-8: Translation table used.

4	4	*	*	[	[	[	[	D	D	3	3	]	]	*	.
.	*	*	*	.	2	2	2	E	E	F	F	]	]	]	]
*	3	3	.	.	A	A	A	B	A	A	A	[	[	]	]
1	1	1	1	.	B	B	B	*	*	[	[	C	C	C	C

The number of applicable production rules now increased even more. The primeval soup period shortened in such a way that some subpopulations started with valid networks in their first generation. The convergence speed of the GA seemed to increase, but I stopped the simulation halfway, since due to this convergence speed the migration period now seemed too high. I decreased the variable *migration_freq* from 50 to 25, and decreased the variable *minimum_freq* from 70 to 0, and did again two complete runs. The parameters then looked as follows:

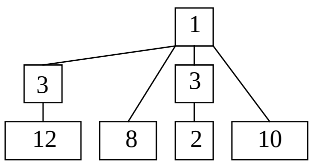
```
#psize           100
#nrmembers       70
#nrgenes         1
#chromsize       1024
#pressure        4.0
#pinv            0.5
#pmut            0.01
#pcross          0.65
#sites           6
#steps           6
#axiom           ABC
#times_train     4
#nrofiter        250
#neighbourhood   circle
#migration_freq  25
#migration_rate  20
#minimum_freq    0
```

Both simulations performed equally well. To give an indication of the performance I will show some results from one of them. During a simulation a subpopulation once in a while saves its best individual. When this particular simulation ended a total of 45 individuals were saved in files. Of these 45 individuals 13 individuals had the curious property of carrying a production rule of the form `A` → `N[ADFC]S`, where `N` is a number and `S` a substring. Below some of these networks are shown, together with their fitness value they got and the production rules that were used. They are sorted by their fitness value, from top-left to bottom-right. With the exception of one network, they are also in order of time of (file) creation.



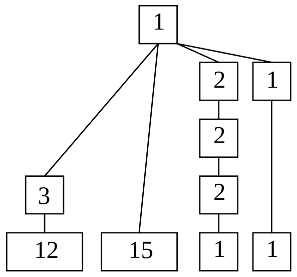
117.90

```
C < C     →
    C > C →
    A     → 3[ADFC]A[CFE]CACC
```



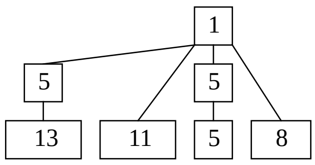
121.48

```
     AA    → C
DF < C     → AAA
     D     →
     C > C →
     A     → 3[ADFC]A[CFE]CAC
     B     →
     C     → B
```



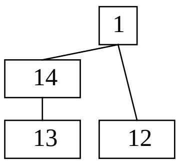
124.60

```
C < C       →
    CA > E  →  DCAC4B[[B31BD]2[FA]3]
                BE[DA]221CF3A
    A       →  3[ADFC]A[CFE]CA
    F       →
    BD      →  A
```



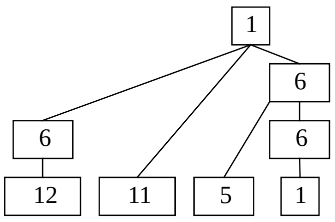
126.82

```
    F  > C →
F < C      →  433
F < E      →  B
    AC > F →  FFF
    F      →  B
    B  > C →
    A      →  3[ADFC]A[CFEAC]A
    C  > A →  F
```



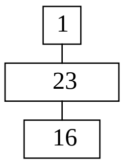
129.07

```
     D > C  →  F
D <  C      →  F
     B      →  [ECE]4C
FE < EA     →
     A      →  4[ADFC]A[CFEEA]
```



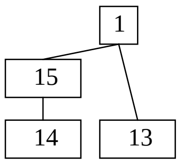
129.77

```
C      →
F > C  →
A      →  3[ADFC]A[C[EAC]AC]
B      →
```



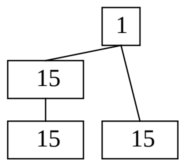
130.06

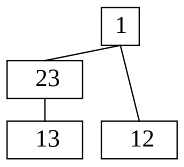
```
         B              →  23CAA1C3ACAAAB
         C              →  [[D]FAEC]
         23CAA          →  1C3ACAAAB
CACA  <  AAB  >  A      →
AAB   <  A              →  234AD[C]1B
  C   <  D              →  F3B
         F              →  CA
         A              →
```



131.42

```
      C      →  BC
DCD < B      →  BE2
      B > E  →
FE  < EA     →
     A      →  4[ADFC]A[CFEEA]
      B      →
```





131.82

```
B → 2CCAA1AC2ACAEAABE
C → [ECFAEC]A
E →
A →
```

131.88

```
    C →
    E →
C < D →
    A → 4[ADFC]A[CAFEAC]
```

figure 8-9 *Example of the development of production rules and their corresponding network architectures. Sorted by fitness, from top-left to bottom-right.*

The last and best network was found after about 50.000 evaluations had been performed. The way the networks seem to develop is by keeping 12 to 16 input nodes on the left-side, increasing the number of nodes in the hidden layer that is connected between the left-most input-part and the output node, and by making the right-side of the network simpler with every step. Also note from the examples shown above the number of production rules used to develop the network architectures. In the original setup mostly 1 (if at all) was used, and with any luck 2 production rules were used. Because production rules are now more easily applied to the axiom the convergence speed of the GA increases. This can also be seen from table 8-9 when compared to table 8-5.

Table 8-9: Values showing how the fitness was determined when using the translation table 8-8.

invalid	97,000	80 %
database	11,000	9 %
trained	14,000	11 %
total evaluations	122,000	100 %

The total number of evaluations needed for the GA to converge has decreased by about 75%. Relatively speaking though, the number of invalid produced network architectures increased a bit. The ratio between *database* and *trained* has changed significantly. In the old situation the number of individuals that got their fitness from the database was about 4 times the number of individuals that got actually trained. In the new situation more individuals get trained than individuals getting their fitness from the database. This can very well be explained as follows: in the old situation where mostly only one production rule was used the genetic operators had little chance of making changes in that, so that the offspring was often like one of its parents. In the new situation the genetic operators have a higher chance to mix different production rules, therefore creating more differences in the offspring produced. Since the offspring are now more often different from their parents less use is made of the database. Still, the database has its use as can be deduced from tables 8-9 and 8-10, since otherwise the program would probably have spent about 275,000 seconds more, while the time that was needed to maintain the database is about 1,500% less.

Table 8-10: Times spent per part of the program.

part of program	CPU time (sec.)	CPU time (perc.)
Decoding from chromosomes to production rules	9,000	2 %
G2L-System	14,000	4 %
Training and evaluation of ANN	350,000	90 %
Database operators	17,000	4 %

Figure 8-10 shows the fitness of the best individual of the total population, the average of the best individuals from all subpopulations and the average fitness of the total population throughout the simulation. Again it has to be noted that this can only be viewed as an indication of how the fitness developed; it is not the actual development since the asynchronisation of the processes makes it hard to keep track of this. With this last simulation for example one subpopulation stopped at around generation 160 while another stopped at around generation 60. Each subpopulation writes its fitness values to a separate file each generation, so the figure was created simply by adding all the values from the separate files.

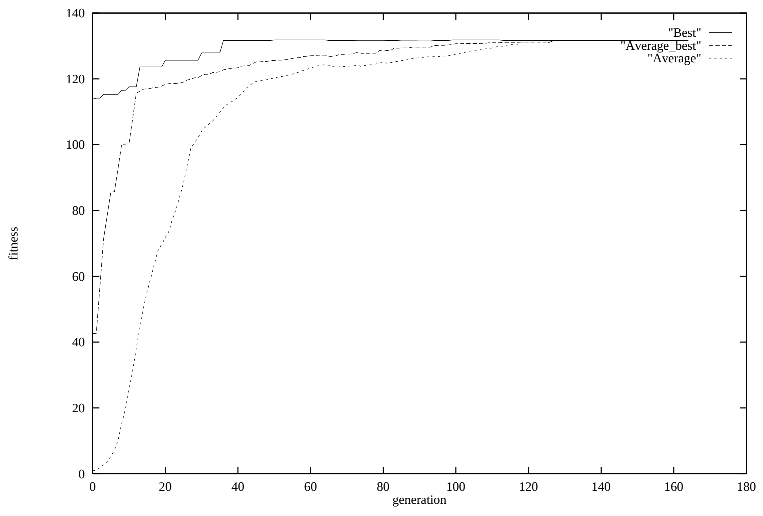


figure 8-10 *The fitness of the total population during one simulation.*

Finally the networks found during the simulation are compared to standard backpropagation networks with one hidden layer. The learning rate was 0.4, and the momentum was 0.9. The number of nodes in the hidden layer for the standard networks was varied from 4 to 10. The 32 patterns were presented 250 times to each network. Each network was tested 50 times. Table 8-11 shows the results.

Table 8-11: Performance of standard networks on the TC problem.

nr of hidden	4	5	6	7	8	9	10
Error	3.62	1.21	1.27	1.12	0.84	0.66	0.75
Correct	1	12	9	13	20	29	26
Fitness	120.81	125.57	125.01	127.41	128.34	129.16	128.73

The first row shows the average error, and the second row shows the number of times the network classified all 32 patterns correctly out of the 50 runs. The last row shows the fitness the network would've gotten. Table 8-12 shows an identical table, but for the networks found during the simulations.

Table 8-12: Performance of found networks on the TC problem.

network	figure 8-6	16 hidden	23 hidden	figure 8-9
Error	1.26	0.28	0.34	0.34
Correct	8	44	42	45
Fitness	126.49	131.28	130.47	130.70

The more nodes are added to one hidden layer, the better it gets. Note that the fitness of the network from figure 8-6 was higher than it deserved according to this last table. This was probably because during that simulation no database was installed and the variable *times* was set to 1, which meant that the network could have gotten a lot of chances to improve its fitness due to one good set of initial random weights with which it started the training.

Remember from the paragraph about the primeval soup theory that originally it took a lot of time to create something out of nothing. Adapting the translation table is a way to create something out of nothing faster. Since initial individuals with non-zero fitnesses can now be created within reasonable time I decided to re-install this feature in the program.

Lastly, I tried to solve the TC problem using a smaller population size. Instead of 100 individuals per subpopulation I tried 50. With other problems tackled with a GA this is already considered a 'large' subpopulation size. However, for this application it produces undesirable results, such as premature convergence of the GA on the subpopulations. I could not solve this by decreasing the migration period. Therefore subpopulation sizes of 100 seem to be minimally needed.

‘What’ / ‘Where’ problem.

The ‘what’/‘where’ problem was proposed by Rueckl et al. [48]. They wanted to explain why ‘what’ and ‘where’ are processed by separate cortical visual structures. Like the TC problem there are a number of patterns consisting of 3x3 pixels, see figure 8-11. In this problem there are 9 patterns and they are placed on a 5x5 grid. The network should not only recognize the *form* of the pattern, but also the *place* of the pattern on the grid.

Rueckl et al. tried a number of experiments with different networks. Each network had 25 input, 18 hidden and 18 output nodes. Nine output nodes were used for encoding ‘what’ and nine were used for encoding ‘where’. Their results showed that a network learned faster and better when the hidden layer was split where each part processed the ‘what’ and ‘where’ separately. Of importance was the number of nodes allocated to the ‘what’ and ‘where’ systems. Figure 8-12 shows the optimal network found. Four nodes are dedicated to the processing of ‘where’, and fourteen nodes to ‘what’ which is a more complex task. Rueckl et al. also stated that when the processing of ‘what’ and ‘where’ was done with one hidden layer of 18 nodes then interference would occur.

Boers and Kuiper reported in their thesis that when they tried to find a network that could solve this problem they found a simple network without a hidden layer, thus one with 25 input nodes directly connected to the 18 output nodes. They also reported that this simple network without a hidden layer could very well learn this problem [4].

Before describing some experiments I did with this problem it is necessary to introduce a new network learning-algorithm I incorporated into the program.

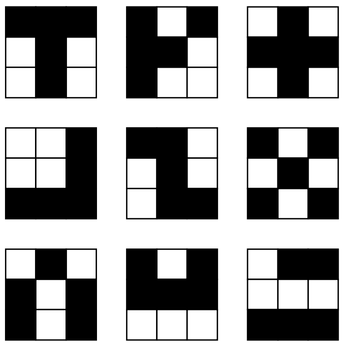


figure 8-11 *The nine patterns.*

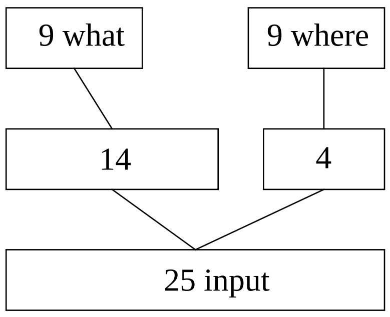


figure 8-12 *The network from Rueckl et al. [48].*

The Quickprop Algorithm

The greatest obstacle in the application is the slow speed at which the neural network simulator works. This is a problem for which most applications with a back-propagation algorithm suffer. Also, back-propagation learning scales up poorly as tasks become larger and more complex. Therefore I looked for another simulator to install, and found the fastest available in the form of the Quickprop program developed by Fahlman.[1] It has a new learning algorithm that is faster than standard back-propagation by an order of magnitude or more and it appears to scale up very well as the problem size increases.

Everything in the Quickprop algorithm proceeds as in standard back-propagation, but for each weight a copy of the error derivative computed during the previous training epoch is kept, along with the difference between the current and previous values of this weight. Two assumptions are made: first, that the error vs. weight curve for each weight can be approximated by a parabola whose arms open upward; second, that the change in the slope of the error curve, as seen by each weight, is not affected by all the other weights that are changing at the same time. For each weight, independently, the previous and current error slopes are used as are the weight change between the points at which these slopes were measured to determine a parabola; a jump is directly made to the minimum point of this parabola.

In order to keep the jump within limits a new parameter is introduced, called *maxfactor*. No weight step is allowed to be larger in magnitude than *maxfactor* times the previous step for that weight. Another new parameter is called *hypererr*. When the error of an output unit is calculated usually the error is simply defined as the squared difference between the actual and the desired output. It was suggested that a non-linear error function might speed up learning. The idea was that for small differences between the output and the desired output, the error should behave linearly, but as the difference increased, the error function should grow faster than linearly, heading toward infinity as errors approach their maximum values. One function that meets these requirements is the hyperbolic arctangent of the difference. When *hypererr* is set to true the non-linear error function is used, otherwise the usual linear error function is used.

There are some more new parameters included in the Quickprop algorithm, but they are outside the scope of this thesis (and not changed during the simulations here). For a more in-depth description of how the Quickprop algorithm works the reader is referred to Fahlman's article [15].

To get an impression of the difference in speed between standard backpropagation and the Quickprop algorithm I did some experiments using the 'What'/'Where' problem using the network from Rueckl et al. as shown in figure 8-12. The learning rate *alpha* was chosen 0.4 and the momentum rate *beta* was chosen 0.9. Using the standard backpropagation simulator it took about 8 minutes to do 1000 training epochs. Doing 1,000 training epochs using the Quickprop simulator took about 1 minute. After 1,000 epochs the networks hadn't converged yet. When I used a training set with input and output values from the set {0.1, 0.9}, and *hypererr* set to false, it took about 1250 epochs on average for the Quickprop algorithm to converge. Changing the training set to input from the set {0.0, 1.0} and output from the set {-0.5, 0.5} it took about 200 epochs on average for the network to converge.

1. The original LISP version and a translation into C of the Quickprop program can be found at http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/systems/0.html.

This means that in this case the symmetrical sigmoid activation function performs significantly better than the asymmetrical sigmoid activation function which is usually used. When in addition the parameter *hypererr* is set to true it took about 100 epochs on average to converge, or about 5 to 10 seconds. Thus it can be seen that the Quickprop algorithm computes faster *and* converges faster. This is definitely advantageous compared to the original neural network simulator used in the application.

As already mentioned in the previous paragraph, Rueckl et al. stated that when the processing of ‘what’ and ‘where’ was done with one hidden layer of 18 nodes then interference would occur. I tried whether this was true with the last settings from above. It turned out that out of 50 runs the Quickprop algorithm could always learn the problem correctly, where 90% of the runs converged within 150 epochs. I also checked whether the problem could be learned with simple gradient descent. This too worked fine; usually within 200 epochs the network converged. These results suggest that the division that Rueckl et al. made in the hidden layer is not really necessary. The split does make the network converge faster though.

Also, Boers and Kuiper reported in their thesis that a simple network with 25 input nodes directly connected to the 18 output nodes (thus without a hidden layer) could very well learn the problem [4]. I’ve tried numerous simulations with different neural network simulators to see whether this was true. Not one experiment succeeded.[1] The ‘where’ problem could always be solved, but the more complex ‘what’ problem typically had after extended training 3 or 4 values wrong per output node of the 81 training patterns. Another simulator reported a maximum of 78% of the problem learnt[2]. These results suggest that this problem is *not* linearly separable. Results from new experiments with the application to see whether a network architecture can be found that is able to learn the ‘What’ / ‘Where’ problem will be presented in a later paragraph.

TC problem revisited.

With a new neural network simulator installed in the application I tried the TC problem again. Before that I looked at some error curves produced by the Quickprop algorithm. Figure 8-12 shows two typical error curves. The errors of the networks plotted were computed by adding all the squared errors on the output nodes. Although overall the errors converge nicely, there are sudden jumps. Since the fitness of a network was a function of the error at the last epoch it can be seen that it is possible that the error at the last epoch is as bad as when the network started training. I changed the fitness by making it a function of the lowest error found during the training of the network. This way a network can’t accidentally get a low fitness because it just happened to have a high error at the last epoch.

1. This difference in results could be explained by a bug that was found by M. Borst in the neural network code of Boers and Kuiper.
2. This was the general purpose backpropagation NevProp simulator, also available from http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/systems/0.html.

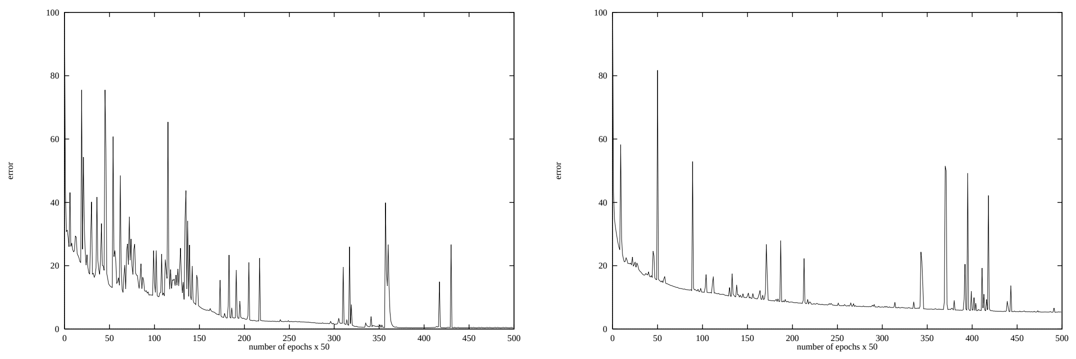


figure 8-12 *Example error lines plotted when using Quickprop.*

The application was run again trying to find a network architecture that could solve the TC problem using the Quickprop algorithm. The variable *nrofiter* was chosen as tight as possible, namely 60 epochs (compare this with the old value of 250 epochs that was minimally needed). The parameters were as follows:

```
#psize           100
#nrmembers       70
#nrgenes         1
#chromsize       1024
#pressure        1.5
#pinv            0.5
#pmut            0.01
#pcross          0.625
#sites           6
#steps           6
#axiom           ABC
#times_train     4
#nrofiter        60
#neighbourhood   circle
#migration_freq  5
#migration_rate  10
#minimum_freq    0
#alpha           0.4
#beta            0.9
#hypererr        true
#maxfactor       1.75
#scorethreshold  0.35
#maxnodes        60
```

The variable *scorethreshold* is used for a “threshold and margin” criterion: if the absolute difference between the actual and the desired output is below this threshold, then the output is counted as correct. The variable *maxnodes* denotes the maximum number of nodes in the network that is allowed for training.

Within 7500 evaluations (12 minutes) a network was found with a fitness of 130.54, which is already pretty good. I let the program run until the GA converged. Eventually it found the network as shown in figure 8-13, with a fitness of 131.75. This network *always* learns the TC problem, usually within 40 epochs. Table 8-13 shows how the fitness was determined. Compared to table 8-9 one can see another dramatic decrease in the number of total evaluations needed for the GA to converge. The total time spent on training the networks was 80,000 seconds.

Table 8-13: Values showing how the fitness was determined when using Quickprop.

invalid	36,500	76 %
database	3,500	7 %
trained	8,000	17 %
total evaluations	48,000	100 %

131.75

```
  A < A > A → C
      A > A → AAC1[AB]
  A < A     → AAC1[AB]
      A     →
      C     → [FC]
  A < B     →
      B     → B[ACB4C4B]2
BFF < F > C → 4AC4BABD
```

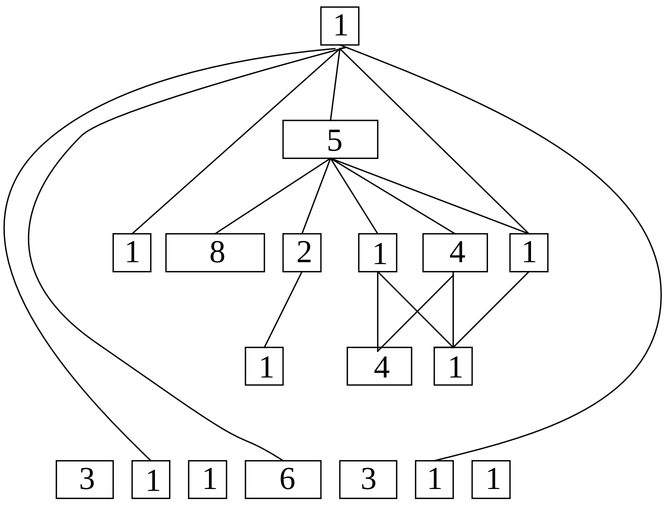


figure 8-13 *Network that solves the TC problem within 40 epochs on average using the Quickprop algorithm. For clarity sake the connections from the input nodes to hidden nodes have been omitted.* ***All input nodes are fully connected to all hidden nodes****. 8 input nodes are also directly connected to the output node. The fitness and the production rules that produced the network architecture are shown on the left.*

'What' / 'Where' problem revisited.

I did two runs trying to find an architecture that can solve the 'What' / 'Where' problem. The parameters were as follows:

```
#psize          100
#nrmembers      70
#nrgenes        1
#chromsize      1024
#pressure       1.5
#pinv           0.5
#pmut           0.01
#pcross         0.625
#sites          6
#steps          6
#axiom          ABC
#times_train    3
#nrofiter       150
```

```
#neighbourhood   ladder
#migration_freq  15
#migration_rate  10
#minimum_freq    0
#alpha           0.4
#beta            0.9
#hypererr        true
#maxfactor       1.25
#scorethreshold  0.35
#maxnodes        120
```

The fitness was calculated as 2000 - *MinErrorBits* - *MinError*, where *MinErrorBits* is the minimum number of bits of one epoch that were wrong and *MinError* is the square root of the minimum summed squared error of the output nodes of one epoch during a training.

Unfortunately, the genetic algorithm converged after about 300,000 evaluations to network architectures with fitnesses 1,962 and 1,958, which is bad. Why the application was unable to find a good network architecture for this problem is unclear. Maybe the variable *nrofiter* was set too low. Or it could be that it is hard to find production rules that expand the axiom into a network architecture suitable for this problem.

The mapping problem.

The last experiment presented is called the mapping problem. Standard backpropagation has difficulty with this when using one or no hidden layer. Van Hoogstraten used this problem to investigate the influence of the structure of a network upon its ability to map functions [32]. A two-dimensional map is created with an input space of $(0,1]^2$. On this map 100 (10 x 10) points are assigned to four classes (here the symbols ▲ ■ ● ✕ are used). Van Hoogstraten constructed two mappings, where the second was derived from the first by ‘misclassifying’ three of the 100 points. The second mapping is shown in figure 8-14. The misclassified points are (0.5,0.6), (0.6,1.0), (0.8.0.4) and can be seen as *noise*. Although Van Hoogstraten wanted networks that were not fouled by that noise (and therefore ignored them), the interest here is in networks that are able to learn *all* points correctly.

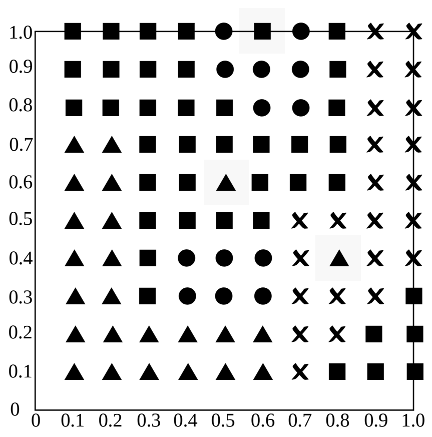


figure 8-14 *The input grid for the mapping problem.*

A network is presented with two input values (*x* and *y* respectively) and four output values (one for each symbol). Van Hoogstraten tried networks with hidden layers containing 6 and 15 nodes, both of which were more or less able to learn the first mapping without the noise, but failed to learn the second mapping with the three misclassified points. Another network with 100 nodes in the hidden layer was able to learn one of the misclassified points, but failed on the other two. Only when he used a network with three hidden layers of 20 nodes each all three misclassified points were learned correctly. This last network has 920 connections.

Boers and Kuiper tried to find a *small, modular* network that was able to learn the second mapping correctly. They needed three days and 11 SUN Sparc4 workstations to converge to a network with 23 nodes and with 82 connections. It took three more days for them to reach the network shown in figure 8-15, which has 24 nodes and 153 connections. According to Boers and Kuiper this network had a consistently higher fitness in comparison with the 2-20-20-20-4 network used by Van Hoogstraten.

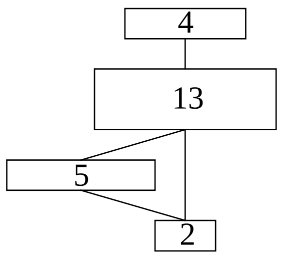


figure 8-15 *Network found by Boers and Kuiper [4].*

I doubted however whether the network from figure 8-15 was really that good, especially since the error curves plotted in the thesis of Boers and Kuiper showed that the 2-20-20-20-4 network converged to an error of 74 and their own network to an error of 14. Although it is not stated how the errors were computed, it is presumed that they were calculated as usual, i.e. by simply adding the squared errors of the output nodes. An error of 74 for the 2-20-20-20-4 network would then be very peculiar since that would mean an average error of 0.4 in each output node, which means that surely a lot of patterns were not being recognized correctly, while Van Hoogstraten states that the network can learn *all* patterns correctly. An error of 14 also seems quite high (average error of 0.2 in each output node).

First I tried to see what parameters to use for the 2-20-20-20-4 network. The usual learning rate *alpha* of 0.4 seemed too high; 0.05 produced better results. Then I tried experimenting with the activation function and the *hypererr* parameter to see which combination leads to learning the problem correctly. Figure 8-16 shows the results of averages of 5 runs.

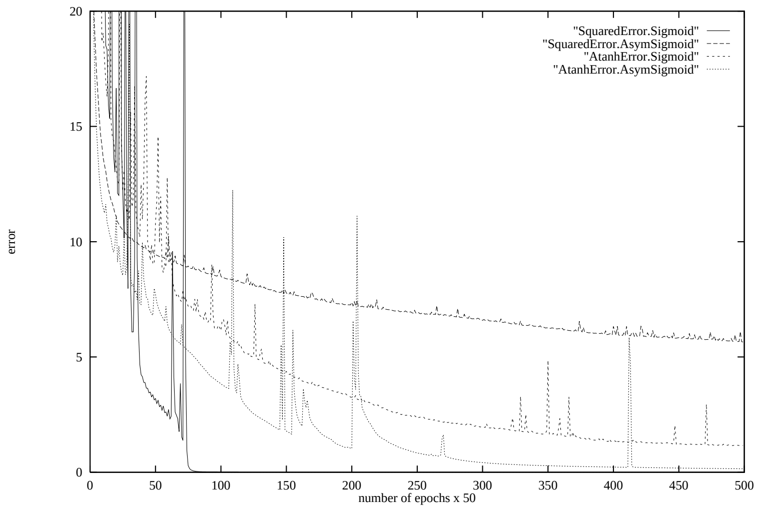


figure 8-16 *Errors plotted for the 2-20-20-20-4 network using different error and activation functions.*

The errors plotted are always the summed squared errors of the output nodes. The worst curve in the figure uses the asymmetrical sigmoid activation function and the linear error function; i.e. the same combination that Boers and Kuiper used. After 25,000 epochs it still misclassified 2 to 4 points. Better results are produced when the non-linear error function is used, but still not all points are classified correctly every time. When using the non-linear error function together with a symmetrical sigmoid activation function the error converges towards zero within 20,000 epochs and learns all points correctly. When using the linear error function again, together with the symmetrical sigmoid activation function the error converges within 5,000 epochs towards zero and classifies *all* points correctly including the noise.

I also tried varying the learning-rate *alpha*, when using the last combination of error and activation functions. Higher values of the learning rate also converged towards zero, but were more erratic ('jumpy') in their error curves, especially at start-up. Values of *alpha* above 0.3 started to give problems in convergence towards zero.

Lastly I tried the network found by Boers and Kuiper as shown in figure 8-15. The best results were again produced when I used a low learning rate, the linear error function and the asymmetrical sigmoid activation function. Figure 8-17 shows the convergence of their network, with the errors averaged over 5 runs. It ends with an error of 5.6, at which point it still misclassified 6 or 7 points.

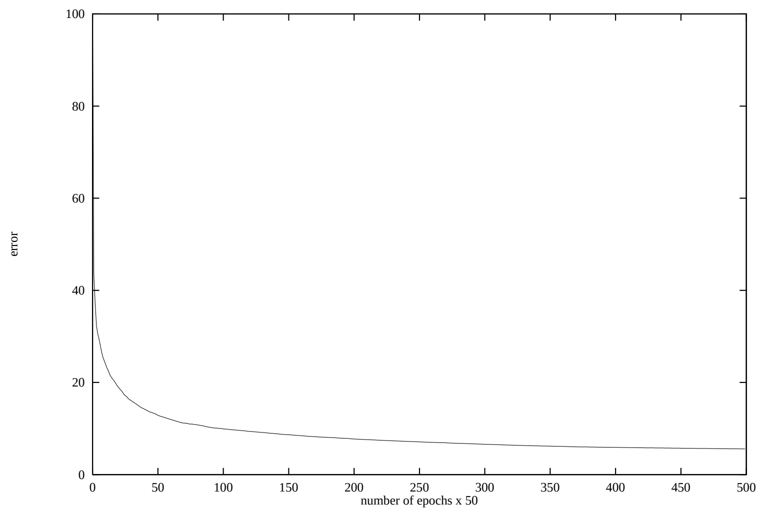


figure 8-17 *Error plotted for the network from Boers and Kuiper.*

I tried experimenting with the application using the symmetrical sigmoid activation function and the linear error function in order to find network architectures that can classify all points correctly. I did three experiments, each taking a couple of days. The fitness was calculated as 400 - *MinErrorBits* - *MinError*, where *MinErrorBits* is the minimum number of bits of one epoch that were wrong and *MinError* is the square root of the minimum summed squared error of the output nodes of one epoch during a training. The results of the experiments are presented next.

Experiment 1

The parameters used for the first experiment were as follows:

```
#psize           100
#nrmembers       70
#nrgenes         1
#chromsize       1024
#pressure        4.0
#pinv            0.5
#pmut            0.01
#pcross          0.625
#sites           6
#steps           6
#axiom           ABC
#times_train     1
#nrofiter        1000
#neighbourhood   ladder
#migration_freq  10
#migration_rate  20
#minimum_freq    0
#alpha           0.05
#beta            0.9
#hypererr        false
#maxfactor       1.25
#scorethreshold  0.35
#maxnodes        60
```

The best found network had 55 nodes with 552 connections, and had received a fitness of 391.12. The architecture of the network is too complicated to draw, therefore the matrix representation of the network is given in appendix C, together with the production rules responsible for the architecture. Figure 8-18 shows the average error out of 5 runs plotted for the network. The lowest average error is about 3.5. In 3 runs all but the 3 misclassified points were learned correctly, and in 2 runs all but 2 were learned correctly.

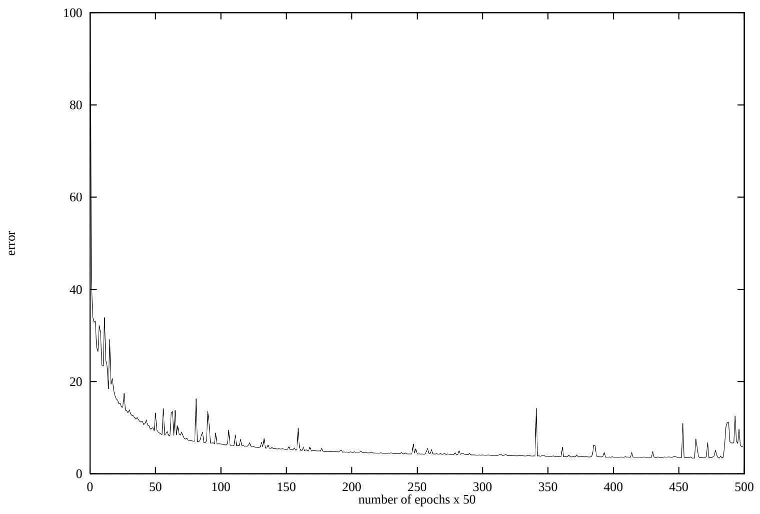


figure 8-18 *Average error out of 5 runs plotted for best found network from experiment 1.*

Experiment 2

In the second experiment I doubled the population size and the size of the intermediate population, and used a higher selection pressure (only the best 10% had a chance of getting selected). The parameters were as follows:

```
#psize           200
#nrmembers       140
#nrgenes         1
#chromsize       1024
#pressure        10.0
#pinv            0.75
#pmut            0.007
#pcross          0.625
#sites           4
#steps           6
#axiom           ABC
#times_train     1
#nrofiter        1000
#neighbourhood   ladder
#migration_freq  10
#migration_rate  40
#minimum_freq    0
#alpha           0.05
#beta            0.9
#hypererr        false
#maxfactor       1.25
#scorethreshold  0.35
#maxnodes        60
```

The best found network had 56 nodes with 683 connections, and had received a fitness of 390.08. The matrix representation of the network is given in appendix D, together with the production rules responsible for the architecture. Figure 8-19 shows the average error out of 5 runs plotted for the network. The lowest average error is about 1.9. In 2 runs *all* points were learned correctly. In 2 other runs all but one, and in one run all but 4 points were learned correctly.

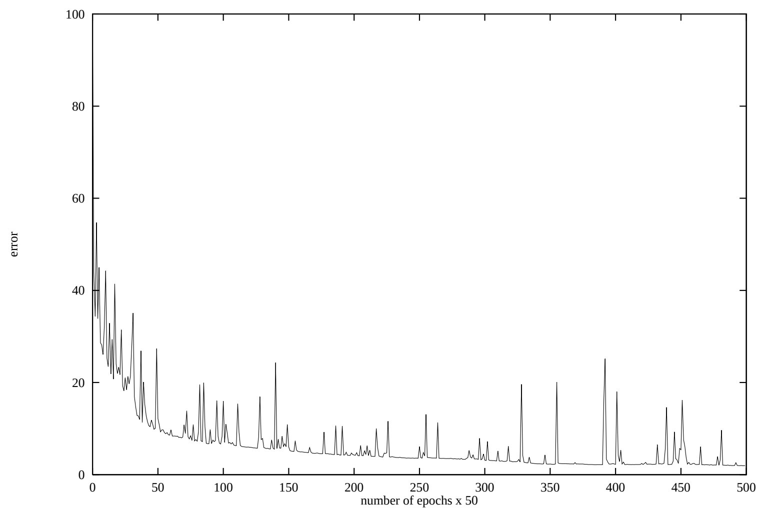


figure 8-19 *Average error out of 5 runs plotted for best found network from experiment 2.*

experiment 3

In the last experiment for the mapping problem the population size was chosen low, the selection pressure was set high again, inversion always took place, the migration period was halved, and the variable *nrofiter* was doubled. The parameters for this third experiment looked as follows:

```
#psize           75
#nrmembers       50
#nrgenes         1
#chromsize       1024
#pressure        10.0
#pinv            1.0
#pmut            0.01
#pcross          0.625
#sites           6
#steps           5
#axiom           ABC
#times_train     1
#nrofiter        2000
#neighbourhood   ladder
#migration_freq  5
#migration_rate  16
#minimum_freq    0
#alpha           0.05
#beta            0.9
#hypererr        false
#maxfactor       1.25
#scorethreshold  0.35
#maxnodes        60
```

The best found network had 48 nodes with 635 connections, and had received a fitness of 392.72. The matrix representation of the network is given in appendix E, together with the production rules responsible for the architecture. Figure 8-20 shows the average error out of 5 runs plotted for the network. The lowest average error is between 7.0 and 14.0. In 1 run *all* points were learned correctly. In 1 other run all but 3, in 2 runs all but 4 and in 1 other run all but 7 were learned correctly.

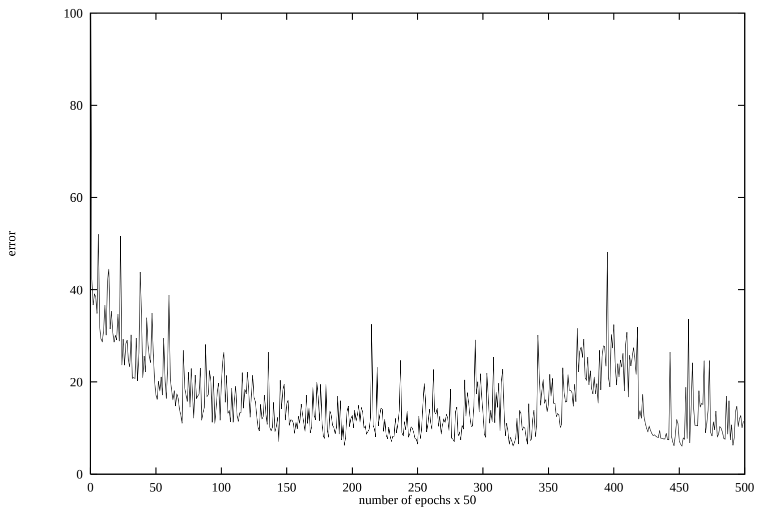


figure 8-20 *Average error out of 5 runs plotted for best found network from experiment 3.*

This last network is very instable in that it has problems in converging. The second and third best networks found in this experiment suffered from the same problems. Two example error plots of these second best networks are given in figure 8-21. As can be seen these networks got a high fitness because they can drop their error quickly to a low point, but when doing extended training they can not continue decreasing the error. Unfortunately, this side effect can not be helped without setting the variable *nrofiter* to higher values which would mean even more time spent on training networks. Overall it can also be seen that what Boers and Kuiper hoped for did not happen in these experiments: the networks produced are hardly small, nor simple, nor stable in their convergence. They are modular though.

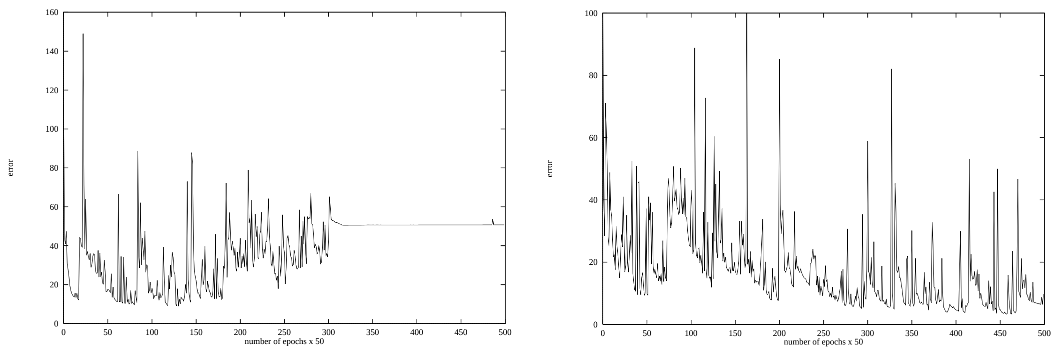


figure 8-21 *Error plots of two other 'good' network architectures found.*

Tables 8-14 and 8-15 show the usual statistics obtained during the third experiment. The ratio of the number of trained networks is quite high compared with the other experiments, while the database is hardly used. Nearly all time is spent on training networks.

Table 8-14: Values showing how the fitness was determined when solving the mapping problem.

invalid	14,800	65 %
database	300	1 %
trained	7,500	33 %
total evaluations	22,600	100 %

Table 8-15: Times spent per part of the program.

part of program	time (sec.)	time (perc.)
Decoding from chromosomes to production rules	1,800	0 %
G2L-System	900	0 %
Training and evaluation of ANN	1,285,000	99.9 %
Database operators	2,500	0 %

This concludes the experiments done with the application. In the next chapter conclusions are drawn, and some ideas are given for further research.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

The main theme that runs throughout this thesis is simulation of evolution using genetic algorithms on separate islands of subpopulations which communicate with each other once in a while. With multiple processors available, more points per generation can be sampled which means a higher rate of schema processing.

Working with a GA is a delicate matter and can even be viewed as black art. A certain combination of parameters, local search, selection-, replacement- and migration schemes may result in excellent results for a given problem, but it cannot be extended to other problems that easily.

What is required before deciding what building blocks to use for a GA is a deeper understanding of the problem faced to optimize. One could argue that this is the same problem faced when designing network architectures, which was solved by using a genetic algorithm to find good network architectures. Likewise, the question has now presented itself whether we should use a genetic algorithm to optimize a genetic algorithm?[1]

The total system is now beginning to become so complex and messy that some functionings aren't understandable any more. This might seem to be a problem, however it is better suited than deterministic approaches. Evolutionary computation *is* a messy business, and might in the future be the only search strategy able to generate the desired complexity needed. It has come to a point where the *combination* of parameter settings becomes important regarding the effect it will have on the whole.

Although the introduction of parallelism has increased the speed of the program, it has not made it possible to investigate the real potential of the methods proposed by extending the problem-space that is solvable within a feasible amount of time. The

1. In fact, this is precisely what Wiemer has done in her research project [59].

bottle-neck remains the time needed to train one network. Therefore the conclusion is that the methods proposed do seem to have potential, but the program has still not made it practical yet.

Unlike the results of Boers and Kuiper, who usually found small simple modular architectures, during this research large complex modular architectures were found. This difference is probably due to the use of a different translation table, and the newly installed neural network simulator.

In chapter 4 I showed the effect that learning of individuals can have on evolution. On the travelling salesperson problem Lamarckian evolution is more effective and more efficient than Darwinian evolution with Baldwin effect. When using N-steps local descent with the Baldwin effect N must not be too large, otherwise it will severely disrupt the genetic search. Also it should not be too small, otherwise no significant gain can be made compared to using no learning. On the Schwefel optimization problem the most effective form of evolution is when the Baldwin effect is tried. But also for this problem the local search algorithm had to be carefully chosen. From this it can be concluded that for the advantages of the Baldwin effect to appear the problem must have some degree of deceptiveness in its fitness surface. Whether the Baldwin effect is also advantageous when the fitness surface is highly noisy is unclear and is left for future research.

It also seems that a good evolution program for the TSP should incorporate local improvement operators, based on algorithms for local optimization, together with the usual genetic operators, which would incorporate heuristic information about the problem. The quest for an evolution program for the TSP, which would include ‘the best’ representation and genetic operators to be performed on them, is still going on.

I also showed in chapter 4 the effect of using different communication schemes. In general it turns out that the partitioned GA outperforms the canonical serial GA, and likewise the distributed GA outperforms the partitioned GA. Which communication scheme for the distributed model is best is dependable on the sort of problem, but generally speaking the ladderlike arrangement seems to be the most promising variant.

9.2 Future Work

Somewhere midway this research I delved more into parallel genetic algorithms to understand their workings. I’ve also put a lot of time into parallelizing the program of Boers and Kuiper on a CM-5. But there has to come a point when a research project has to be concluded, even when this leaves improvements and ideas open that I would have liked to implement, if only to quench my desire for making things faster and create better quality. In this section therefore, a number of directions are indicated for future research concerning the work of Boers and Kuiper, that appear to be potentially important and fruitful.

Miscellaneous

Switch from hostless mode to host/node mode.

In this first implementation of the program the hostless mode as recommended by CM was chosen. In this mode there is a single set of source code files for the nodes, and the host only acts as I/O server for the nodes. The first suggestion is an implementational one, to now switch to host/node mode. Then there are two sets of source code files, one for the host and one for the node. A separate host program needs to be written, which explicitly must start and monitor the execution of the node programs. This is suggested because now the different GAs run asynchronously from each other, making it difficult to monitor the best found network, fitness, or the number of evaluations done at a particular time. Synchronisation of the processors is not desirable because a processor then runs the risk of being idle for a long time, for instance when another processor has just started training a network. With a host each processor can send its information whenever it wants, and the collection and assimilation of the information is left to the host. Another good reason to switch to host/node mode is to make it possible to implement a monitoring tool and a global database, which are explained next.

Make a tool to monitor the progression of each processor.

Using the hostless mode it is difficult to monitor what each processor is doing at a given time. Directing output to files is not convenient; for instance, saving the best found networks at 10 different stages of a run creates 160 files (since there's no synchronisation 16 processors each have to save 10 networks). With a global monitoring tool in the host/node mode at most 10 files can be saved if desirable. CM-5 provides for a nice X-windows environment, so best found networks could even be drawn on screen, as well as fitness progressions, number of evaluations, number of networks trained, time tables, etc.

Create a database containing sets of production rules already evaluated.

Chapter 8 already lightly touched on this subject. A set of production rules when applied to an axiom results in a network architecture, which is then evaluated. It was already shown that when the production rules that were used are stored in a database together with the fitness of the network that was produced from those rules then the amount of computing time can be reduced considerably. The way the database was programmed was however a basic one, and can be much improved.

Note that when the hostless mode is kept that this will mean that only local databases can be created. So a processor might start to learn a network, which has already been evaluated on another processor. Hence the need for a global database. This can easily be done in the host/node mode, where the database and its functions are maintained on the host. This also means that each processor has to send its intermediate population to the host *after* recombination and mutation but *before* evaluation of all the members, thus creating more communication overhead.

After a call to *lsystem* all the used production rules are stored alphabetically in the string *usedstring*. It would be best to add a new field in the MEMBER structure where *usedstring* can be stored. I haven't done this because I considered this a major change in a basic datastructure of the program which means that a lot of functions have to be changed accordingly. Also, I estimate that the creation of an efficient and effective database is a project in itself.

Parallel Genetic Algorithm

After hundreds of experimentations with genetic algorithms I've become wary of introducing programming tricks, tweaking parameters, and implementing ideas that logically should work but actually only cause unexpected undesirable side effects that overshadow the desired main effect. In this light I can now heartily agree with the following four principles of genetic algorithm design which David Goldberg brought to the attention in his article 'Zen and the Art of Genetic Algorithms' [21]:

1. Let nature be your guide.
2. Beware the frontal assault.
3. Respect the sieve of schemata.
4. Distrust central authority.

Therefore, the next suggestions concerning the genetic algorithm should be taken carefully. Do not fall into the trap of overcontrolling the genetic process, because that will very likely only result in failure.

Revisit the asynchronous concurrent model.

Boers and Kuiper ran the program using one population processed by several Sun Sparc4 workstations. The suggestion is made to investigate this model on the CM-5. For this the host/node mode is necessary, where one global population is maintained on a host and with the nodes performing the genetic operators and evaluations asynchronously. This violates the fourth principle given above because it reintroduces a degree of central control, but a principle is not a law; this model might turn out to be the best one suited for this case.

Speed up initial process.

It may be desirable to use the original translation table. It then takes a couple of generations before a set of production rules is found that can create a valid network. This need not be wasted time in terms of evolution, still it could be investigated if the initial process can be speeded up. This could be done by starting with a population with `useful' production rules created from previous simulations. Or by waiting for random members, but in such a way that when one processor has found enough valid network it first helps other processors which haven't found enough yet. Again, the host/node mode is then preferred where nodes can contribute useful chromosomes to a global buffer on the host. When the buffer is full, each node can then select a number of individuals.

Start with different parameters on each island.

Experiments could be done using different crossover and mutation rates on each island. These parameters could even be coded as a second chromosome in each individual.

Furthermore, experiments could be done using different genetic algorithms per island; differing subpopulation sizes, differing selection schemes, etc. (note: I do not recommend this last suggestion for the program of Boers and Kuiper, only for parallel genetic algorithms in general).

Change communication scheme.

Many different communication schemes can be chosen for the application. I suggest the following: A processor decides for itself when to send its individuals to another processor. This decision is based on the number of evaluations done, and/or whenever the fitness value of a best individual has increased. Instead of sending individuals it might be more effective for a processor to decide when it wants to *receive* individuals. For instance when a subpopulation has converged, it is desirable to receive new blood immediately from other subpopulations without having to wait for the decision of those other processors when they send their individuals. When implementing this scheme processors first need to send a message to the other appropriate processors indicating that they want to receive individuals. This makes it necessary to work with tags attached to the messages in order to differentiate between a request-message and a message sending individuals.

Genetically find optimal communication schemes.

In section 4.3 different communication schemes were investigated to see how well they performed. What was done was actually a search for a good architecture of connected processors. Better architectures could be found by using a genetic algorithm. I.e. use a genetic algorithm to find good architectures of connected processors that lead to favourable results when used by a parallel genetic algorithm.

Preserving initial diversity.

In the beginning a subpopulation should only allow to receive new individuals from another subpopulation when it has produced valid networks on its own. Otherwise the subpopulation might from the start just explore the same space as the other subpopulation is doing, therefore causing less diversity among the total population.

Store subpopulations in 2 dimensions.

I propose a new way to store a population, not necessarily for the work of Boers and Kuiper, but for the GA community in general. The idea is a combination of the island model and the cellular model. As in this work GAs are run separately on populations residing on islands. Once in a while individuals are migrated to other islands, preferably with different migration frequencies like the ratios of figure 4-19 in order to create islands within islands. A sequential GA is always performed on a population of which the members are stored linearly. Instead a single population should be stored in 2 dimensions, as in the real world. On this level it's like the cellular model. Selection schemes should be used accordingly, only within a local neighbourhood. So on each island a 2 dimensional population is stored. This total model should provide for high local inbreeding because of the 2 dimensional storage of subpopulations, and keep higher rates of diversity within the total population.

G2L-system

Experiment with different translation tables.

More experiments should be done with different translation tables since this has a major effect on the network architectures generated.

New experiments should be done with the original translation table, but with the Quickprop simulator installed, to compare with the results from section 8.2.

Do not use a fixed axiom.

Because the axiom is fixed potentially good production rules might get lost. To increase the number of valid network architectures produced by the G2L-system the following suggestions are made:

- The axiom could be added as a parameter in the chromosome.
- A list of axioms could be made, from which the G2L-system picks the first it encounters on which it can apply a production rule.
- An axiom could be determined from the valid production rules decoded from the chromosome. For example, when a chromosome decodes into the valid production rules:

```
A < A      → [CBC]1D
    CC > D →  C1[A2B]A
```

the fixed axiom could first be tried, but if no production rule apply, the axiom could be chosen as A1A, [CC]1D, a combination of both, etc.

Research building blocks.

The power of genetic search is based on schema processing. That means that there have to be similarities in the chromosomes that correlate to near equal fitnesses. There have to be exploitable similarities so that when using genetic operators the disruption of those similarities are minimized. When a disruption is minimized, but still this results in uncorrelated network architectures and fitnesses, then the conclusion must be that a genetic algorithm is not suitable for this problem (because it is too deceptive and/or noisy), or that the encoding by the G2L-system needs to be replaced by a different encoding scheme. *I suggest this should be researched thoroughly with high priority.*

Neural Networks

Until now only ‘small’ problems were presented to the program. During this research it became clear that the algorithms used would make it very difficult to approach the goal of learning larger problems such as the two-spiral problem. A neural network simulator several order of magnitude faster might permit a worthy attack.

Investigate Baldwin effect.

In chapter 4 I showed the effect of Lamarckian evolution and the Baldwin effect on Darwinian evolution for the travelling salesperson problem and for the Schwefel optimization function. The Baldwin effect could also be implemented in this program, by evolving the architecture of a neural network *while* it is learning a task. Networks learning in this way have already been researched at the group where this research took place, but it has not yet been implemented together with a GA. Evaluation of a chromosome may in this case take more time, but the genetic search may also be improved in convergence speed as well as in the quality of the best network found.

Before that, I recommend investigating the Baldwin effect on a problem with a quick evaluation function and with a highly complex fitness surface to get an idea of how well the effect is in that case (research with the Walsh polynomials is recommended). This because it seems the fitness surface of the program is highly complex, and it might be worthwhile to first gain insight into how the Baldwin effect behaves in complex fitness surfaces.

To implement Lamarckian evolution in the program it is necessary to translate the architecture of a network that evolved by itself back to production rules. At this moment that seems to be quite impossible (except of course for the trivial case $A \rightarrow X$ with A chosen as the axiom and X being the string representing the total graph).

Investigate faster learning procedures for neural networks.

A network simulator several orders of magnitude faster is needed. When presented to learn the two-spiral problem for instance, the time needed to evaluate one architecture with backpropagation takes a whole day.

No matter how many improvements one makes in the GA and/or G2L-system, and no matter how much the problem to learn is turned upside down, the fact remains that it takes hours for a network to learn the 2-spiral problem.

Therefore it is far from practical to use genetic search to find a good architecture that learns this problem with the usual backpropagation or the Quickprop algorithm. Some research might be done if it is possible to use some heuristics to determine the *potential* of an architecture and/or to *estimate* how well an architecture will do. If this is rated high enough the network is allowed to train.

Gruau is doing similar work to this one, but he uses genetic programming with cellular encoding [25]. According to Gruau his networks are evaluated typically within 100 milliseconds. He achieved this by coding the networks plus their weights, and then training the network for just one or two epochs. The weights are simply initialized to 0 or ±1 and his networks are of a deterministic nature. Gruau’s goal is to find a network plus given weights that can *solve* a particular problem in 1 epoch, while the goal of Boers and Kuiper is to find good network architectures that can *learn* the problem. Still, it is suggested to investigate coding the weights of a network. This can for instance be done by interpreting a character from the G2L-system as 0 or ±1 weight for a connection.

A Table 4-2 complete

Table 4-2: Results from experimenting with different neighbourhood relations.

Type	Mean	S.D.	Members	Rate	Period
ladder	1.89	0.13	8	4	250
ladder	1.93	0.11	8	4	1000
ladder	1.94	0.13	16	4	50
ladder	1.95	0.12	4	2	250
random	1.97	0.10	4	4	250
circle	1.97	0.11	8	4	1000
circle	1.97	0.11	4	2	500
ladder	1.97	0.12	8	8	250
ladder	1.98	0.10	8	2	500
random	2.00	0.11	8	4	500
circle	2.00	0.11	16	8	50
random	2.00	0.11	16	4	50
random	2.01	0.11	16	2	50
circle	2.01	0.13	4	2	250
circle	2.02	0.12	4	8	500
random	2.02	0.12	16	8	50
ladder	2.03	0.12	2	2	250
random	2.03	0.11	2	2	500
random	2.03	0.10	8	2	50
circle	2.03	0.11	4	4	250
full	2.04	0.11	4	2	1000
random	2.04	0.12	8	2	1000
circle	2.04	0.12	2	8	1000
random	2.04	0.11	2	4	500
ladder	2.04	0.10	16	8	1000
random	2.05	0.11	8	8	500
full	2.05	0.11	8	4	500
ladder	2.05	0.10	16	2	50
circle	2.05	0.13	4	4	500
random	2.06	0.10	16	2	1000

Table 4-2: Results from experimenting with different neighbourhood relations.

Type	Mean	S.D.	Members	Rate	Period
ladder	2.06	0.10	4	2	1000
random	2.06	0.10	4	8	500
circle	2.07	0.10	16	2	500
ladder	2.07	0.11	8	2	250
ladder	2.07	0.11	4	4	1000
circle	2.07	0.11	8	4	500
circle	2.07	0.10	16	8	500
full	2.07	0.11	8	4	250
random	2.07	0.12	4	2	1000
random	2.07	0.10	8	4	250
random	2.08	0.09	4	2	250
ladder	2.08	0.11	16	2	500
random	2.08	0.13	2	8	250
random	2.08	0.10	4	4	500
ladder	2.09	0.11	16	8	250
ladder	2.09	0.10	16	4	250
ladder	2.10	0.12	8	2	1000
circle	2.10	0.10	8	8	500
circle	2.10	0.11	4	2	1000
ladder	2.10	0.12	2	4	1000
circle	2.10	0.10	16	4	1000
random	2.10	0.11	2	8	500
random	2.11	0.09	2	2	1000
circle	2.11	0.09	8	2	500
ladder	2.11	0.10	16	4	500
ladder	2.11	0.10	2	2	500
random	2.11	0.11	4	8	250
split	2.11	0.13	2	2	n.a.
random	2.12	0.12	8	8	50
ladder	2.12	0.12	16	2	250
ladder	2.12	0.11	8	2	50
random	2.12	0.10	2	2	250
ladder	2.13	0.11	16	4	1000
random	2.13	0.11	2	4	250
full	2.13	0.13	2	2	1000
random	2.13	0.10	2	8	1000
circle	2.13	0.11	16	8	1000
random	2.14	0.09	8	2	250
circle	2.14	0.12	8	8	1000
full	2.14	0.10	8	2	500
circle	2.14	0.12	2	4	500
ladder	2.15	0.10	8	4	500
circle	2.15	0.11	8	2	1000
split	2.15	0.11	16	4	n.a.
circle	2.16	0.11	8	8	250
ladder	2.16	0.11	2	8	250

Table 4-2: Results from experimenting with different neighbourhood relations.

Type	Mean	S.D.	Members	Rate	Period
ladder	2.16	0.11	4	8	500
random	2.16	0.11	16	8	500
random	2.16	0.10	8	4	1000
circle	2.16	0.12	16	4	500
ladder	2.16	0.11	2	2	1000
random	2.16	0.10	4	2	500
circle	2.16	0.10	2	8	500
ladder	2.17	0.12	16	2	1000
ladder	2.17	0.10	8	8	50
random	2.17	0.12	16	2	500
circle	2.17	0.10	2	2	250
circle	2.17	0.12	4	8	1000
circle	2.17	0.10	4	4	50
circle	2.18	0.11	2	4	250
random	2.18	0.10	8	8	250
circle	2.18	0.12	16	4	50
full	2.18	0.10	16	8	1000
circle	2.18	0.11	8	2	250
random	2.18	0.09	8	8	1000
split	2.18	0.11	8	2	n.a.
circle	2.18	0.11	2	2	1000
full	2.18	0.10	8	2	250
split	2.19	0.12	16	2	n.a.
circle	2.19	0.10	8	4	250
ladder	2.19	0.12	4	2	500
split	2.20	0.11	4	4	n.a.
ladder	2.20	0.11	4	2	50
random	2.20	0.12	4	4	1000
circle	2.20	0.11	16	8	250
ladder	2.20	0.12	2	4	250
split	2.21	0.11	8	4	n.a.
full	2.21	0.13	16	8	250
ladder	2.21	0.11	2	2	50
full	2.21	0.11	16	8	500
circle	2.21	0.11	2	8	250
circle	2.21	0.11	8	2	50
random	2.21	0.11	8	2	500
full	2.22	0.11	16	2	50
ladder	2.22	0.12	2	4	500
full	2.22	0.12	16	4	500
circle	2.22	0.11	2	4	1000
ladder	2.22	0.11	4	4	500
random	2.22	0.10	4	8	1000
full	2.23	0.11	16	2	500
ladder	2.23	0.12	2	8	1000
split	2.23	0.11	16	8	n.a.

Table 4-2: Results from experimenting with different neighbourhood relations.

Type	Mean	S.D.	Members	Rate	Period
full	2.23	0.13	2	2	500
ladder	2.24	0.11	8	8	500
ladder	2.24	0.10	4	4	250
ladder	2.24	0.09	2	8	500
circle	2.25	0.12	8	4	50
ladder	2.25	0.11	16	8	50
circle	2.26	0.12	4	2	50
full	2.26	0.13	4	8	500
random	2.26	0.12	2	2	50
circle	2.27	0.11	4	4	1000
full	2.28	0.13	4	4	250
ladder	2.28	0.12	4	8	1000
random	2.28	0.12	8	4	50
ladder	2.29	0.12	8	8	1000
full	2.29	0.12	16	4	250
full	2.29	0.10	8	8	500
full	2.29	0.10	4	2	250
full	2.29	0.11	16	2	250
circle	2.29	0.10	2	2	500
circle	2.30	0.09	16	4	250
full	2.30	0.11	8	8	1000
ladder	2.31	0.10	16	8	500
split	2.31	0.11	4	2	n.a.
full	2.31	0.11	16	4	1000
random	2.31	0.11	2	4	1000
random	2.31	0.11	16	4	500
random	2.31	0.11	16	2	250
full	2.31	0.12	4	2	500
full	2.32	0.12	2	4	250
circle	2.32	0.11	16	2	50
split	2.33	0.11	8	8	n.a.
circle	2.33	0.09	4	8	250
ladder	2.33	0.12	2	8	50
circle	2.33	0.10	16	2	250
full	2.33	0.10	8	2	1000
full	2.33	0.10	8	4	1000
full	2.33	0.13	2	8	1000
full	2.33	0.12	2	2	250
full	2.34	0.12	16	2	1000
random	2.34	0.11	16	8	1000
ladder	2.34	0.11	4	8	250
random	2.34	0.11	16	4	1000
split	2.34	0.11	2	4	n.a.
circle	2.35	0.12	4	8	50
full	2.35	0.11	4	8	250
full	2.36	0.13	4	4	500

Table 4-2: Results from experimenting with different neighbourhood relations.

Type	Mean	S.D.	Members	Rate	Period
random	2.38	0.09	16	8	250
random	2.38	0.11	16	4	250
ladder	2.38	0.12	8	4	50
split	2.38	0.12	2	8	n.a.
full	2.39	0.14	2	8	500
ladder	2.39	0.11	4	8	50
ladder	2.39	0.11	2	4	50
circle	2.41	0.12	8	8	50
random	2.41	0.12	4	8	50
full	2.42	0.13	4	2	50
full	2.43	0.12	2	2	50
circle	2.44	0.11	2	2	50
full	2.45	0.11	16	4	50
ladder	2.45	0.12	4	4	50
full	2.46	0.13	16	8	50
circle	2.46	0.13	2	8	50
full	2.47	0.12	8	8	50
split	2.47	0.10	4	8	n.a.
random	2.48	0.12	4	4	50
random	2.48	0.10	2	8	50
full	2.48	0.09	4	8	1000
full	2.48	0.11	2	4	500
random	2.49	0.10	4	2	50
full	2.49	0.12	8	2	50
partitioned	2.51	0.10	2	n.a.	n.a.
partitioned	2.51	0.11	8	n.a.	n.a.
full	2.51	0.12	8	4	50
circle	2.52	0.09	2	4	50
random	2.52	0.13	2	4	50
partitioned	2.52	0.11	16	n.a.	n.a.
full	2.54	0.12	8	8	250
full	2.54	0.12	2	8	250
full	2.56	0.11	4	4	50
full	2.63	0.14	4	8	50
partitioned	2.64	0.13	4	n.a.	n.a.
full	2.66	0.14	2	4	50
full	2.68	0.12	2	8	50

B Backpropagation

This appendix gives a short description of the standard backpropagation algorithm for a network with *p* input, *q* hidden and *r* output nodes. A more extensive treatment on backpropagation networks can be found in Rumelhart and McClelland [49].

1. Initialize all the weights of the network with random values (e.g. between -1 and 1). We will denote the weights of the hidden layer and of the output layer as w_{ij}^{h} and w_{ij}^{o} respectively. The notation w_{ij}^{h} stands for the weight between input node i and hidden node j.
2. Choose an input/output pair $(\bar{x},\bar{y})$, where $\bar{x} = \begin{bmatrix} x_1 \\ \dots \\ x_p \end{bmatrix}$ and $\bar{y} = \begin{bmatrix} y_1 \\ \dots \\ y_r \end{bmatrix}$ are the inputvector and outputvector, and assign the inputvector to the corresponding input nodes.
3. Propagate the activation of the input layer to the hidden layer, and calculate the stimulation and activation of the hidden nodes. Often the bias θ of each node is implemented as an extra node 0 with a standard activation of 1, the weights from node 0 to the other nodes in the network are used as adaptive thresholds. The activation of the nodes in the hidden layer now becomes:

$$h_j = f(stim(h_j)) = \frac{1}{1 + e^{-\sum_{i=0}^{p} w_{ij}^{h} \cdot x_i}}$$

4. Propagate the activation of the *q* hidden nodes to the output layer.

$$o_j = \frac{1}{1 + e^{-\sum_{i=0}^{q} w_{ij}^{0} \cdot h_i}}$$

5. Calculate the deltas (the errors) of the output layer:

$$\delta_i^o = o_i (1 - o_i) (y_i - o_i)$$

6. Compute the deltas for the hidden layer:

$$\delta_i^h = h_i (1 - h_i) \sum_{j=1}^{r} \delta_j^o w_{ij}^o$$

7. Adjust the weights between the hidden layer and the output layer:

$$w_{ij}^o (t+1) = w_{ij}^o (t) + \alpha \delta_j^o h_i + \beta \Delta w_{ij}^o (t-1)$$

where

$$\Delta w_{ij}^o (t) = w_{ij}^o (t+1) - w_{ij}^o (t)$$

The last term is called the *momentum*, it tends to keep the weight changes (Δw_{ij}) going in the same direction by averaging the changes over the last few training cycles. Usually α, the *learning-rate parameter*, is chosen between 0.1 and 0.5 and β, the *momentum parameter* between 0.8 and 0.95.

8. Adjust the weights between the input layer and the hidden layer:

$$w_{ij}^h (t+1) = w_{ij}^h (t) + \alpha \delta_j^h x_i + \beta \Delta w_{ij}^h (t-1)$$

9. Repeat steps 2 to 8 until the *total error* of the network $E = \frac{1}{2} \sum_{i=1}^{r} (y_i - o_i)^2$ is small enough for each of the training-vector pairs in the training-set.

C Result Mapping Experiment 1

```
fitness: 391.12


production rules:

     BC → CE
     B  → 1
     A  → [BAA2]
     C  → AAC
     E  → CB


matrix file:
#nodes 55
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

D Result Mapping Experiment 2

```
fitness: 390.08


production rules:

     A → AD[A]
     B →
     D → 2


matrix file:
#nodes 56
0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

E Result Mapping Experiment 3

```
fitness: 392.72


production rules:

     AA < A → EE31D
          E → E31D
          A → B2B3
       B2B3 →
      E < C → [ECAB2A][D33A3]A
          B → 1B24[BA]
          C → C3


matrix file:
#nodes 48
0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

References

1. H. Abelson and A.A. diSessa; *Turtle Geometry*, Cambridge, Mass.: MIT Press, 1981.

2. M. Baldwin; ‘A new factor in evolution’. In: *American Naturalist*, 30:441-451. 1896.

3. S. Baluja; ‘A Massively Distributed Parallel Genetic Algorithm (mdpGA)’, Tech. Report CMU-CS-92-196, Carnegie Mellon University, School of Computer Science, 1992.

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

5. E.J.W. Boers, H. Kuiper, B.L.M. Happel and I.G. Sprinkhuizen-Kuyper; ‘*Designing modular artificial neural networks*’. Proceedings of computing Science in the Netherlands CSN’93, 87-96, 1993.

6. E.J.W. Boers; ‘Using L-systems as Graph Grammars: G2L-systems’. To appear.

7. J. P. Cohoon and S. U. Hegde and W. N. Martin and D. Richards; ‘Punctuated equilibria: a parallel algorithm’. In: *Proceedings of the 2nd International Conference on Genetic Algorithms and their application (ICGA)*, John J. Greffenstette (Ed.), Lawrence Erlbaum Associates Publishers, 1987.

8. J.P. Cohoon and W.N. Martin and D. Richards; ‘Genetic Algorithms on Punctuated equilibria’. In: *Proceedings of the 1st int. wks. on Parallel Problem Solving from Nature*, PPSN-90, 1990.

9. R.J. Collins and D.R. Jefferson; ‘Selection in massively parallel genetic algorithms’, In: *Proceedings of the 4th International Conference on Genetic Algorithms and their application (ICGA)*, R-K. Belew, L.B. Booker (Eds.), 249-256, San Diego CA, 1991.

10. C. Darwin; *The origin of species*, 1859.

11. Y. Davidor; ‘A Naturally Occuring Niche & Species Phenomenon: The Model and First Results’, In: *Proceedings of the 4th International Conference on*

Genetic Algorithms and their application (ICGA), R-K. Belew, L.B. Booker (Eds.), 257-263, San Diego CA, 1991.

12. R. Dawkins; *The Blind Watchmaker,* Longman 1986, Reprinted with appendix by Penguin.

13. M. Dorigo and V. Maniezzo; 'Introduction and Overview'. In: *Parallel Genetic Algorithms: Theory & Applications*', J. Stender (Ed.), IOS Press, Amsterdam, 1993.

14. N. Eldredge and S.J. Gould; 'Punctuated Equilibria: an Alternative to Phyletic Grdualism'; In: *Models of Paleobiology.* 82-115, Freedom, T.J. Schopf, 1972.

15. S.E. Fahlman; 'Faster learning variations on back-prop: An emperical study', In: *Proceedings of the 1988 Connectionist Models Summer School,* Morgan-Kaufman, San Mateo, 1988.

16. M.J. Flynn; 'Very high speed computing systems', *Proc. IEEE,* vol.54, 1901-1909, 1966.

17. M.J. Flynn; 'Some computer organizations and their effectiveness', *IEEE Trans. Comp.,* vol. C-21, 948-960, 1972.

18. D.E. Goldberg; Genetic Algorithms and Walsh Functions: Part I, A Gentle Introduction, *Complex Systems* 3, 129-152, 1989

19. D.E. Goldberg; Genetic Algorithms and Walsh Functions: Part II, Deception and Its Analysis, *Complex Systems* 3, 153-171, 1989.

20. D.E. Goldberg; *Genetic algorithms in search, optimization and machine learning.* Addison-Wesley, Reading, 1989.

21. D.E. Goldberg; 'Zen and the Art of Genetic Algorithms'. In: *Proceedings of the 3rd International Conference on Genetic Algorithms and their application (ICGA),* 80-85, J.D. Schaffer (Ed.), Morgan Kaufmann, San Mateo CA, 1989.

22. D.E. Goldberg and K. Deb; A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms,* ed. G. Rawlins, Morgan Kaufmann, 1991.

23. M. Gorg-Schleuter; 'Explicit Parallelism of Genetic Algorithms Through Population Structures', In: *PPSN 1: Proceedings of Parallel Problem Solving from Nature*, 1st Workshop, H.P. Schwefel, R. Manner (Eds.), 150-159, Springer-Verslag, Berlin, Germany, 1990.

24. J.J. Grefenstette; *Parallel adaptive algorithms for function optimization,* (Technical Report No. CS-81-19), Nashville: Vanderbilt University, Computer Science Department, 1981

25. F.Gruau; *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm.* PhD. Thesis, l'Ecole Normale Supérieure de Lyon, 1994.

26. F. Gruau and D. Whitley; 'Adding learning to the cellular development of neural network: evolution and the Baldwin effect'. In: *Evolutionary Computation 1*, 213-233, 1993.

27. B.L.M. Happel and J.M.J. Murre; 'The design and evolution of modular neural

network architectures'. In: *Neural Networks, vol. 7*, 985-1004, 1994.

28. J.H. Holland; *Hierarchical descriptions of universal spaces and adaptive systems*. University of Michigan Press, Ann Harbor, 1968

29. J.H. Holland; *Adaptation in natural and artificial systems.* The University of Michigan Press, Ann Harbor, 1975.

30. P. Hogeweg and B. Hesper; 'A model study on biomorphological description'. In: *Pattern Recognition*, 6, 165-179, 1974.

31. A. Homaifar, G.E. Liepins and S. Guan; 'A New Approach on the Traveling Salesman Problem by Genetic Algorithms'. In: *Proceedings of the 3rd International Conference on Genetic Algorithms and their application (ICGA)*, 116-121, J.D. Schaffer (Ed.), Morgan Kaufmann, San Mateo CA, 1989.

32. R.J.W. van Hoogstraten; *A neural network for genetic facies recognition.* Unpublished student report, Leiden, 1991.

33. R.A. Jacobs, M.I. Jordan and A.G. Barto; 'Task decomposition through competition in a modular connections architecture: The what and where vision task'. In: *Cognitive Science 15*, 219-250, 1991.

34. D.S. Johnson; 'Local optimization and the travelling salesman problem'. In: *Proceedings of the 17th colloquium on automata, languages, and programming*, Lecture notes in CS, Vol. 443, 446-461, M.S. Paterson (Ed.), Springer-Verslag, 1990.

35. K.A. de Jong; *Analysis of the Behavior of a Class of Genetic Algorithms*, Ph.D. dissertation, Univ. of Michigan, 1975.

36. H. von Koch; 'Une méthode géometrique élémentaire pour l'étude de certaines questions de la théorie des courbes planes'. In: *Acta Mathematica*, 30, 1905.

37. J.R. Levenick; 'Inserting introns improves genetic algorithm success rate: taking a cue from biology'. In: *Proceedings of the 4th International Conference on Genetic Algorithms and their application (ICGA)*, 271-278, R-K. Belew, L.B. Booker (Eds.), San Diego, CA., 1991.

38. A. Lindenmayer; 'Mathematical Models for Cellular Interaction in Development, parts I and I'. In: *Journal of Theoretical Biology*, 18, 280-315, 1968.

39. B.B. Mandelbrot; *The fractal geometry of nature*. Freeman, San Francisco, 1982.

40. M.L. Minsky and S. Papert; *Perceptrons*. MIT Press, Cambridge, MA, 1969.

41. H. Mühlenbein; 'Evolution in Time and Space -- The Parallel Genetic Algorithm'. In: *Foundations of Genetic Algorithms*, Gregory J. E. Rawlins, Morgan Kaufmann Publishers.

42. H. Mühlenbein, M. Schomisch, J. Born; 'The parallel genetic algorithm as function optimizer'. In: *Proceedings of the 4th International Conference on Genetic Algorithms and their application (ICGA)*, 271-278, R-K. Belew, L.B. Booker (Eds.), San Diego, CA., 1991.

43. J.M.J. Murre; *Learning and categorization in neural networks*. Hemel-Hampstead: Harvester Wheatsheaf (Hilldale, N.J.Lawrence Erlbaum), 1992.

44. M.G. Norman; 'A genetic approach to topology optimization for multiprocessor

architectures', Tech. Report ECSP-TR-7, Univ. of Edinburgh, Dept. of Physics, 1988.

45. L.E. Orgel; *The origins of life*. Wiley, New York, 1973.

46. C.B. Pettey, M.R. Leuze and J.J. Grefenstette; 'A parallel genetic algorithm'. In: *Proceedings of the 2nd International Conference on Genetic Algorithms and their application (ICGA),* John J. Greffenstette (Ed.), Lawrence Erlbaum Associates Publishers, 1987.

47. P. Prusinkiewicz and J. Hanan; *Lindenmayer systems, fractals and plants*. Spinger-Verslag, New York, 1990.

48. J.G. Rueckl, K.R. Cave and S.M. Kosslyn; 'Why are 'what' and 'where' processed by seperate cortical visual systems? A computational investigation'. In: *Journal of cognitive neuroscience, 1,* 171-186, 1989.

49. D.E. Rumelhart and J.L. McClelland (Eds.); *Parallel distributed processing. Volume 1: Foundations*. MIT Press, Cambridge, MA, 1986.

50. P. Spiessens and B. Manderick ; 'Fine-Grained Parallel Genetic Algorithms'. In: *Proceedings of the 3rd International Conference on Genetic Algorithms and their application (ICGA),* J.D. Schaffer (Ed.), Morgan Kaufmann, San Mateo CA, 1989.

51. P. Spiesens and B. Manderick; 'A Massively Parallel Genetic Algortihm. Implementation and First Analysis', In: *Proceedings of the 4th International Conference on Genetic Algorithms and their application (ICGA),* R-K. Belew, L.B. Booker (Eds.), 264-270, San Diego CA, 1991.

52. H.P. Schwefel; *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*, volume 26 of *Interdisciplinary System Research.* Birkhäuser, 1977.

53. R. Tanese; 'Distributed genetic algorithms'; In: *Proceedings of the 3rd International Conference on Genetic Algorithms and their application (ICGA),* J.D. Schaffer (Ed.), 434-439, Morgan Kaufmann, San Mateo CA, 1989.

54. Alan Turing; 'Computing Machinery and Intelligence'. In: *Computers and Thought*, E.A. Feigenbaum & J. Feldman (Eds.), McGraw-Hill, New York, 1963.

55. J.L. Walsh; *'A closed set of orthogonal functions'*, Ann. J. Math. 55, 5-24, 1923.

56. D. Whitley; 'The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best'. In: *Proceedings of the 3rd International Conference on Genetic Algorithms and their application (ICGA),* 116-121, J.D. Schaffer (Ed.), Morgan Kaufmann, San Mateo CA, 1989.

57. D. Whitley; 'Fundamental principles of deception.'. In: *Foundations of Genetic Algorithms,* G. Rawlins (Ed.), Morgan Kaufmann, 1991.

58. D. Whitley; *A Genetic Algorithm Tutorial,* (Technical Report CS-93-103), Colorado State University, 1993.

59. M. Wiemer; *Optimalisatie met een genetisch algoritme van een genetisch algoritme om neurale netwerken te ontwerpen.* Unpublished Master's thesis, Leiden University, Leiden, 1995.