Genetic Algorithms: Theory and Applications

Lecture Notes

Second Edition — WS 2001/2002

by
Ulrich Bodenhofer

Institut für Algebra, Stochastik und
wissensbasierte mathematische Systeme
Johannes Kepler Universität
A-4040 Linz, Austria

FLLL
Fuzzy Logic Laboratorium Linz - Hagenberg

Preface

This is a printed collection of the contents of the lecture *"Genetic Algorithms: Theory and Applications"* which I gave first in the winter semester 1999/2000 at the Johannes Kepler University in Linz. The reader should be aware that this manuscript is subject to further reconsideration and improvement. Corrections, complaints, and suggestions are cordially welcome.

The sources were manifold: Chapters 1 and 2 were written originally for these lecture notes. All examples were implemented from scratch. The third chapter is a distillation of the books of Goldberg [13] and Hoffmann [15] and a handwritten manuscript of the preceding lecture on genetic algorithms which was given by Andreas Stöckl in 1993 at the Johannes Kepler University. Chapters 4, 5, and 7 contain recent adaptations of previously published material from my own master thesis and a series of lectures which was given by Francisco Herrera and myself at the Second Summer School on Advanced Control at the Slovak Technical University, Bratislava, in summer 1997 [4]. Chapter 6 was written originally, however, strongly influenced by A. Geyer-Schulz's works and H. Hörner's paper on his C++ GP kernel [18].

I would like to thank all the students attending the first GA lecture in Winter 1999/2000, for remaining loyal throughout the whole term and for contributing much to these lecture notes with their vivid, interesting, and stimulating questions, objections, and discussions.

Last but not least, I want to express my sincere gratitude to Sabine Lumpi and Susanne Saminger for support in organizational matters, and Peter Bauer for proof-reading.

Ulrich Bodenhofer, February 2000.

Contents

List of Figures

Chapter 1

Basic Ideas and Concepts

Growing specialization and diversification have brought a host of monographs and textbooks on increasingly specialized topics. However, the "tree" of knowledge of mathematics and related fields does not grow only by putting forth new branches. It also happens, quite often in fact, that branches which were thought to be completely disparate are suddenly seen to be related.

Michiel Hazewinkel

1.1 Introduction

Applying mathematics to a problem of the real world mostly means, at first, modeling the problem mathematically, maybe with hard restrictions, idealizations, or simplifications, then solving the mathematical problem, and finally drawing conclusions about the real problem based on the solutions of the mathematical problem.

Since about 60 years, a shift of paradigms has taken place—in some sense, the opposite way has come into fashion. The point is that the world has done well even in times when nothing about mathematical modeling was known. More specifically, there is an enormous number of highly sophisticated processes and mechanisms in our world which have always attracted the interest of researchers due to their admirable perfection. To imitate such principles mathematically and to use them for solving a broader class of problems has turned out to be extremely helpful in various disciplines. Just briefly, let us mention the following three examples:

Artificial Neural Networks (ANNs): Simple models of nerve cells (neurons) and the way they interact; can be used for function approximation, machine learning, pattern recognition, etc. (e.g. [26, 32]).

Fuzzy Control: Humans are often able to control processes for which no analytic model is available. Such knowledge can be modeled mathematically by means of linguistic control rules and fuzzy sets (e.g. [21, 31]).

Simulated Annealing: Robust probabilistic optimization method mimicking the solidification of a crystal under slowly decreasing temperature; applicable to a wide class of problems (e.g. [23, 30]).

The fourth class of such methods will be the main object of study throughout this whole series of lectures—*Genetic Algorithms (GAs)*.

The world as we see it today, with its variety of different creatures, its individuals highly adapted to their environment, with its ecological balance (under the optimistic assumption that there is still one), is the product of a three billion years experiment we call evolution, a process based on sexual and asexual reproduction, natural selection, mutation, and so on [9]. If we look inside, the complexity and adaptability of today's creatures has been achieved by refining and combining the genetic material over a long period of time.

Generally speaking, genetic algorithms are simulations of evolution, of what kind ever. In most cases, however, genetic algorithms are nothing else than probabilistic optimization methods which are based on the principles of evolution.

This idea appears first in 1967 in J. D. Bagley's thesis "The Behavior of Adaptive Systems Which Employ Genetic and Correlative Algorithms" [1]. The theory and applicability was then strongly influenced by J. H. Holland, who can be considered as *the* pioneer of genetic algorithms [16, 17]. Since then, this field has witnessed a tremendous development. The purpose of this lecture is to give a comprehensive overview of this class of methods and their applications in optimization, program induction, and machine learning.

1.2 Definitions and Terminology

As a first approach, let us restrict to the view that genetic algorithms are optimization methods. In general, optimization problems are given in the

following form:

$$\text{Find an } x_0 \in X \text{ such that } f \text{ is maximal in } x_0, \text{ where } f : X \to \mathbb{R} \text{ is an arbitrary real-valued function, i.e. } f(x_0) = \max_{x \in X} f(x). \tag{1.1}$$

In practice, it is sometimes almost impossible to obtain global solutions in the strict sense of (1.1). Depending on the actual problem, it can be sufficient to have a local maximum or to be at least close to a local or global maximum. So, let us assume in the following that we are interested in values x where the objective function f is "as high as possible".

The search space X can be seen in direct analogy to the set of competing individuals in the real world, where f is the function which assigns a value of "fitness" to each individual (this is, of course, a serious simplification).

In the real world, reproduction and adaptation is carried out on the level of genetic information. Consequently, GAs do not operate on the values in the search space X, but on some coded versions of them (strings for simplicity).

1.1 Definition. Assume S to be a set of strings (in non-trivial cases with some underlying grammar). Let X be the search space of an optimization problem as above, then a function

$$\begin{array}{rccc} c: & X & \longrightarrow & S \\ & x & \longmapsto & c(x) \end{array}$$

is called *coding function*. Conversely, a function

$$\begin{array}{rccc} \tilde{c}: & S & \longrightarrow & X \\ & s & \longmapsto & \tilde{c}(s) \end{array}$$

is called *decoding function*.

In practice, coding and decoding functions, which have to be specified depending on the needs of the actual problem, are not necessarily bijective. However, it is in most of the cases useful to work with injective decoding functions (we will see examples soon). Moreover, the following equality is often supposed to be satisfied:

$$(c \circ \tilde{c}) \equiv \mathrm{id}_S \tag{1.2}$$

Finally, we can write down the general formulation of the encoded maximization problem:

$$\text{Find an } s_0 \in S \text{ such that } \tilde{f} = f \circ \tilde{c} \text{ is as large as possible}$$

The following table gives a list of different expressions, which are common in genetics, along with their equivalent in the framework of GAs:

Natural Evolution	Genetic Algorithm
genotype	coded string
phenotype	uncoded point
chromosome	string
gene	string position
allele	value at a certain position
fitness	objective function value

After this preparatory work, we can write down the basic structure of a genetic algorithm.

1.2 Algorithm.

$t := 0$;
Compute initial population $\mathcal{B}_0$;

WHILE *stopping condition not fulfilled* **DO**
BEGIN
select individuals for reproduction;
create offsprings by crossing individuals;
eventually mutate some individuals;
compute new generation
END

As obvious from the above algorithm, the transition from one generation to the next consists of four basic components:

Selection: Mechanism for selecting individuals (strings) for reproduction according to their fitness (objective function value).

Crossover: Method of merging the genetic information of two individuals; if the coding is chosen properly, two good parents produce good children.

Mutation: In real evolution, the genetic material can by changed randomly by erroneous reproduction or other deformations of genes, e.g. by gamma radiation. In genetic algorithms, mutation can be realized as a random deformation of the strings with a certain probability. The positive effect is preservation of genetic diversity and, as an effect, that local maxima can be avoided.

Sampling: Procedure which computes a new generation from the previous one and its offsprings.

Compared with traditional continuous optimization methods, such as Newton or gradient descent methods, we can state the following significant differences:

1. GAs manipulate coded versions of the problem parameters instead of the parameters themselves, i.e. the search space is S instead of X itself.

2. While almost all conventional methods search from a single point, GAs always operate on a whole population of points (strings). This contributes much to the robustness of genetic algorithms. It improves the chance of reaching the global optimum and, vice versa, reduces the risk of becoming trapped in a local stationary point.

3. Normal genetic algorithms do not use any auxiliary information about the objective function value such as derivatives. Therefore, they can be applied to any kind of continuous or discrete optimization problem. The only thing to be done is to specify a meaningful decoding function.

4. GAs use probabilistic transition operators while conventional methods for continuous optimization apply deterministic transition operators. More specifically, the way a new generation is computed from the actual one has some random components (we will see later by the help of some examples what these random components are like).

Chapter 2

A Simple Class of GAs

> *Once upon a time a fire broke out in a hotel, where just then a scientific conference was held. It was night and all guests were sound asleep. As it happened, the conference was attended by researchers from a variety of disciplines. The first to be awakened by the smoke was a mathematician. His first reaction was to run immediately to the bathroom, where, seeing that there was still water running from the tap, he exclaimed: "There is a solution!". At the same time, however, the physicist went to see the fire, took a good look and went back to his room to get an amount of water, which would be just sufficient to extinguish the fire. The electronic engineer was not so choosy and started to throw buckets and buckets of water on the fire. Finally, when the biologist awoke, he said to himself: "The fittest will survive" and went back to sleep.*
>
> Anecdote originally told by C. L. Liu

In this chapter, we will present a very simple but extremely important subclass—genetic algorithms working with a fixed number of binary strings of fixed length. For this purpose, let us assume that the strings we consider are all from the set

$$S = \{0, 1\}^n,$$

where n is obviously the length of the strings. The population size will be denoted with m in the following. Therefore, the generation at time t is a list of m strings which we will denote with

$$\mathcal{B}_t = (b_{1,t}, b_{2,t}, \ldots, b_{m,t}).$$

All GAs in this chapter will obey the following structure:

2.1 Algorithm.

$t := 0$;
Compute initial population $\mathcal{B}_0 = (b_{1,0}, \ldots, b_{m,0})$;

WHILE *stopping condition not fulfilled* **DO**
BEGIN
  FOR $i := 1$ **TO** m **DO**
    select an individual $b_{i,t+1}$ *from* $\mathcal{B}_t$;

  FOR $i := 1$ **TO** $m - 1$ **STEP** *2* **DO**
    IF $\mathsf{Random}[0, 1] \leq p_{\mathbf{C}}$ **THEN**
      cross $b_{i,t+1}$ *with* $b_{i+1,t+1}$;

  FOR $i := 1$ **TO** m **DO**
    eventually mutate $b_{i,t+1}$;

  $t := t + 1$
END

Obviously, selection, crossover (done only with a probability of $p_{\mathbf{C}}$ here), and mutation are still degrees of freedom, while the sampling operation is already specified. As it is easy to see, every selected individual is replaced by one of its children after crossover and mutation; unselected individuals die immediately. This is a rather common sampling operation, although other variants are known and reasonable.

In the following, we will study the three remaining operations selection, crossover, and mutation.

2.1 Genetic Operations on Binary Strings

2.1.1 Selection

Selection is *the* component which guides the algorithm to the solution by preferring individuals with high fitness over low-fitted ones. It can be a deterministic operation, but in most implementations it has random components.

One variant, which is very popular nowadays (we will give a theoretical explanation of its good properties later), is the following scheme, where the

probability to choose a certain individual is proportional to its fitness. It can be regarded as a random experiment with

$$\mathsf{P}[b_{j,t} \textit{ is selected}] = \frac{f(b_{j,t})}{\sum_{k=1}^{m} f(b_{k,t})}. \tag{2.1}$$

Of course, this formula only makes sense if all the fitness values are positive. If this is not the case, a non-decreasing transformation $\varphi : \mathbb{R} \to \mathbb{R}^+$ must be applied (a shift in the simplest case). Then the probabilities can be expressed as

$$\mathsf{P}[b_{j,t} \textit{ is selected}] = \frac{\varphi(f(b_{j,t}))}{\sum_{k=1}^{m} \varphi(f(b_{k,t}))} \tag{2.2}$$

We can force the property (2.1) to be satisfied by applying a random experiment which is, in some sense, a generalized roulette game. In this roulette game, the slots are not equally wide, i.e. the different outcomes can occur with different probabilities. Figure 2.1 gives a graphical hint how this roulette wheel game works.

The algorithmic formulation of the selection scheme (2.1) can be written down as follows, analogously for the case of (2.2):

2.2 Algorithm.

$x := \mathsf{Random}[0, 1]$;
$i := 1$

WHILE $i < m$ & $x < \sum_{j=1}^{i} f(b_{j,t}) / \sum_{j=1}^{m} f(b_{j,t})$ **DO**
  $i := i + 1$;

select $b_{i,t}$;

For obvious reasons, this method is often called proportional selection.

2.1.2 Crossover

In sexual reproduction, as it appears in the real world, the genetic material of the two parents is mixed when the gametes of the parents merge. Usually, chromosomes are randomly split and merged, with the consequence that some genes of a child come from one parent while others come from the other parents.

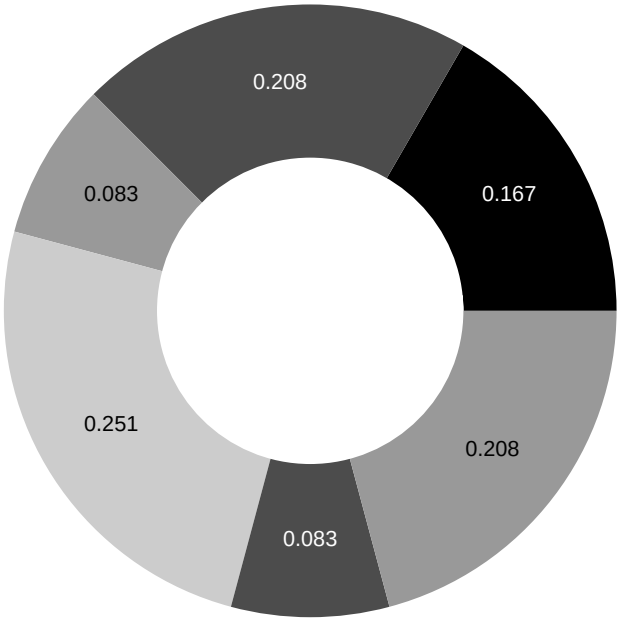


Figure 2.1: A graphical representation of roulette wheel selection, where the number of alternatives m is 6. The numbers inside the arcs correspond to the probabilities to which the alternative is selected.

This mechanism is called crossover. It is a very powerful tool for introducing new genetic material and maintaining genetic diversity, but with the outstanding property that good parents also produce well-performing children or even better ones. Several investigations have come to the conclusion that crossover is the reason why sexually reproducing species have adapted faster than asexually reproducing ones.

Basically, crossover is the exchange of genes between the chromosomes of the two parents. In the simplest case, we can realize this process by cutting two strings at a randomly chosen position and swapping the two tails. This process, which we will call one-point crossover in the following, is visualized in Figure 2.2.

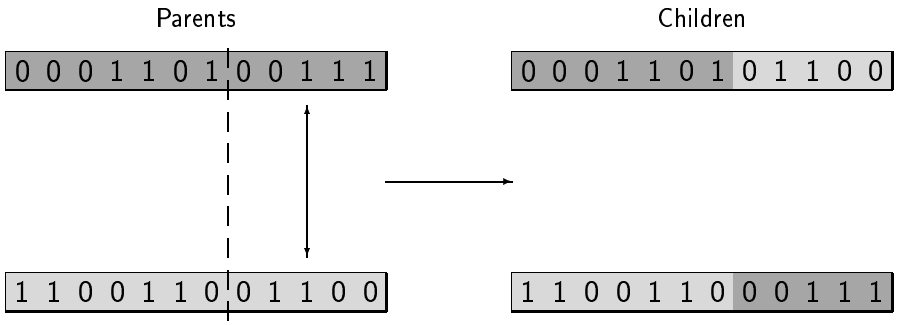


Figure 2.2: One-point crossover of binary strings.

2.3 Algorithm.

$pos := \mathsf{Random}\{1, \ldots, n-1\};$

FOR $i := 1$ **TO** pos **DO**
BEGIN
  $Child_1[i] := Parent_1[i];$
  $Child_2[i] := Parent_2[i]$
END

FOR $i := pos + 1$ **TO** n **DO**
BEGIN
  $Child_1[i] := Parent_2[i];$
  $Child_2[i] := Parent_1[i]$
END

One-point crossover is a simple and often-used method for GAs which operate on binary strings. For other problems or different codings, other crossover methods can be useful or even necessary. We mention just a small collection of them, for more details see [11, 13]:

N-point crossover: Instead of only one, N breaking points are chosen randomly. Every second section is swapped. Among this class, two-point crossover is particularly important

Segmented crossover: Similar to N-point crossover with the difference that the number of breaking points can vary.

Uniform crossover: For each position, it is decided randomly if the positions are swapped.

Shuffle crossover: First a randomly chosen permutation is applied to the two parents, then N-point crossover is applied to the shuffled parents, finally, the shuffled children are transformed back with the inverse permutation.

2.1.3 Mutation

The last ingredient of our simple genetic algorithm is mutation—the random deformation of the genetic information of an individual by means of radioactive radiation or other environmental influences. In real reproduction, the probability that a certain gene is mutated is almost equal for all genes. So, it is near at hand to use the following mutation technique for a given binary string s, where $p_{\mathbf{M}}$ is the probability that a single gene is modified:

2.4 Algorithm.

FOR $i := 1$ **TO** n **DO**
  IF Random$[0,1] < p_{\mathbf{M}}$ **THEN**
    invert $s[i]$;

Of course, $p_{\mathbf{M}}$ should be rather low in order to avoid that the GA behaves chaotically like a random search.

Again, similar to the case of crossover, the choice of the appropriate mutation technique depends on the coding and the problem itself. We mention a few alternatives, more details can be found in [11] and [13] again:

Inversion of single bits: With probability $p_{\mathbf{M}}$, one randomly chosen bit is negated.

Bitwise inversion: The whole string is inverted bit by bit with prob. $p_{\mathbf{M}}$.

Random selection: With probability $p_{\mathbf{M}}$, the string is replaced by a randomly chosen one.

2.1.4 Summary

If we fill in the methods described above, we can write down a universal genetic algorithm for solving optimization problems in the space $S = \{0,1\}^n$.

2.5 Algorithm.

$t := 0$;
Create initial population $\mathcal{B}_0 = (b_{1,0}, \ldots, b_{m,0})$;

WHILE *stopping condition not fulfilled* **DO**
BEGIN

(∗ proportional selection ∗)

FOR $i := 1$ **TO** m **DO**
BEGIN
$x :=$ Random$[0, 1]$;

$k := 1$;
WHILE $k < m$ & $x < \sum_{j=1}^{k} f(b_{j,t}) / \sum_{j=1}^{m} f(b_{j,t})$ **DO**
$k := k + 1$;

$b_{i,t+1} := b_{k,t}$
END

(∗ one-point crossover ∗)

FOR $i := 1$ **TO** $m - 1$ **STEP** *2* **DO**
BEGIN
IF Random$[0, 1] \leq p_{\mathbf{C}}$ **THEN**
BEGIN
$pos :=$ Random$\{1, \ldots, n - 1\}$;

FOR $k := pos + 1$ **TO** n **DO**
BEGIN
$aux := b_{i,t+1}[k]$;
$b_{i,t+1}[k] := b_{i+1,t+1}[k]$;
$b_{i+1,t+1}[k] := aux$
END
END
END

(∗ mutation ∗)

FOR $i := 1$ **TO** m **DO**
FOR $k := 1$ **TO** n **DO**
IF Random$[0, 1] < p_{\mathbf{M}}$ **THEN**
invert $b_{i,t+1}[k]$;

$t := t + 1$
END

2.2 Examples

2.2.1 A Very Simple One

Consider the problem of finding the global maximum of the following function:

$$\begin{array}{rccc} f_1: & \{0, \ldots, 31\} & \longrightarrow & \mathbb{R} \\ & x & \longmapsto & x^2 \end{array}$$

Of course, the solution is obvious, but the simplicity of this problem allows us to compute some steps by hand in order to gain some insight into the principles behind genetic algorithms.

The first step on the checklist of things, which have to be done in order to make a GA work, is, of course, to specify a proper string space along with an appropriate coding and decoding scheme. In this example, it is near at hand to consider $S = \{0,1\}^5$, where a value from $\{0,\ldots,31\}$ is coded by its binary representation. Correspondingly, a string is decoded as

$$\tilde{c}(s) = \sum_{i=0}^{4} s[4-i] \cdot 2^i.$$

Like in [13], let us assume that we use Algorithm 2.5 as it is, with a population size of $m = 4$, a crossover probability $p_{\mathbf{C}} = 1$ and a mutation probability of $p_{\mathbf{M}} = 0.001$. If we compute the initial generation randomly with uniform distribution over $\{0,1\}^5$, we obtain the following in the first step:

Individual No.	String (genotype)	x value (phenotype)	$f(x)$ x^2	pselect$_i$ $\frac{f_i}{\sum f_j}$
1	0 1 1 0 1	13	169	0.14
2	1 1 0 0 0	24	576	0.49
3	0 1 0 0 0	8	64	0.06
4	1 0 0 1 1	19	361	0.31

One can compute easily that the sum of fitness values is 1170, where the average is 293 and the maximum is 576. We see from the last column in which way proportional selection favors high-fitted individuals (such as no. 2) over low-fitted ones (such as no. 3).

A random experiment could, for instance, give the result that individuals no. 1 and no. 4 are selected for the new generation, while no. 3 dies and no. 2 is selected twice, and we obtain the second generation as follows:

Set of selected individuals	Crossover site (random)	New population	x value	$f(x)$ x^2
0 1 1 0|1	4	0 1 1 0 0	12	144
1 1 0 0|0	4	1 1 0 0 1	25	625
1 1|0 0 0	2	1 1 0 1 1	27	729
1 0|0 1 1	2	1 0 0 0 0	16	256

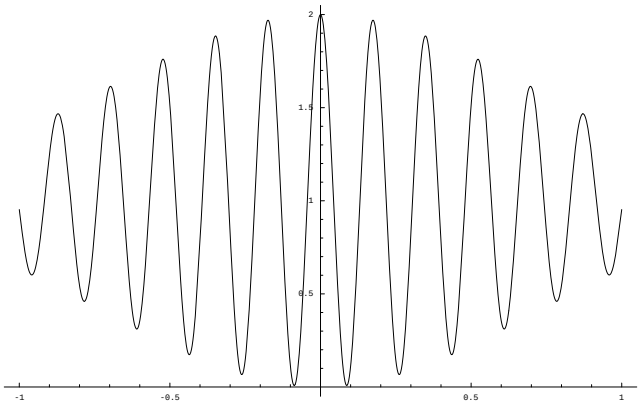


Figure 2.3: The function f_2.

So, we obtain a new generation with a sum of fitness values of 1754, an average of 439, and a maximum of 729.

We can see from this very basic example in which way selection favors high-fitted individuals and how crossover of two parents can produce an offspring which is even better than both of its parents. It is left to the reader as an exercise to continue this example.

2.2.2 An Oscillating One-Dimensional Function

Now we are interested in the global maximum of the function

$$\begin{array}{rcl} f_2 : [-1,1] & \longrightarrow & \mathbb{R} \\ x & \longmapsto & 1 + e^{-x^2} \cdot \cos(36x). \end{array}$$

As one can see easily from the plot in Figure 2.3, the function has a global maximum in 0 and a lot of local maxima.

First of all, in order to work with binary strings, we have to discretize the search space $[-1,1]$. A common technique for doing so is to make a uniform grid of 2^n points, then to enumerate the grid points, and to use the binary representation of the point index as coding. In the general form (for an arbitrary interval $[a,b]$), this looks as follows:

$$\begin{array}{rcl} c_{n,[a,b]} : \quad [a,b] & \longrightarrow & \{0,1\}^n \\ x & \longmapsto & \mathrm{bin}_n\left(\mathrm{round}\left((2^n-1)\cdot\frac{x-a}{b-a}\right)\right), \end{array} \tag{2.3}$$

where bin_n is the function which converts a number from $\{0, \ldots, 2^{n-1}\}$ to its binary representation of length n. This operation is not bijective since information is lost due to the rounding operation. Obviously, the corresponding decoding function can be defined as

$$\begin{array}{rccl} \tilde{c}_{n,[a,b]}: & \{0,1\}^n & \longrightarrow & [a,b] \\ & s & \longmapsto & a + \text{bin}_n^{-1}(s) \cdot \frac{b-a}{2^n-1}. \end{array} \tag{2.4}$$

It is left as an exercise to show that the decoding function $\tilde{c}_{n,[a,b]}$ is injective and that the equality (1.2) holds for the pair $(c_{n,[a,b]}, \tilde{c}_{n,[a,b]})$.

Applying the above coding scheme to the interval $[-1, 1]$ with $n = 16$, we get a maximum accuracy of the solution of

$$\frac{1}{2} \cdot \frac{2}{2^{16} - 1} \approx 1.52 \cdot 10^{-5}.$$

Now let us apply Algorithm 2.5 with $m = 6$, $p_{\mathbf{C}} = 1$, and $p_{\mathbf{M}} = 0.005$. The first and the last generation are given as follows:

```
Generation   1 max. fitness 1.9836 at -0.0050
#0       0111111101010001 fitness: 1.98
#1       1101111100101011 fitness: 0.96
#2       0111111101011011 fitness: 1.98
#3       1001011000011110 fitness: 1.97
#4       1001101100101011 fitness: 1.20
#5       1100111110011110 fitness: 0.37
Average Fitness: 1.41

...

Generation  52 max. fitness 2.0000 at 0.0000
#0       0111111101111011 fitness: 1.99
#1       0111111101111011 fitness: 1.99
#2       0111111101111011 fitness: 1.99
#3       0111111111111111 fitness: 2.00
#4       0111111101111011 fitness: 1.99
#5       0111111101111011 fitness: 1.99
Average Fitness: 1.99
```

We see that the algorithm arrives at the global maximum after 52 generations, i.e. it suffices with at most $52 \times 6 = 312$ evaluations of the fitness function, while the total size of the search space is $2^{16} = 65536$. We can draw the conclusion—at least for this example—that the GA is definitely better than a pure random search or an exhaustive method which stupidly scans the whole search space.

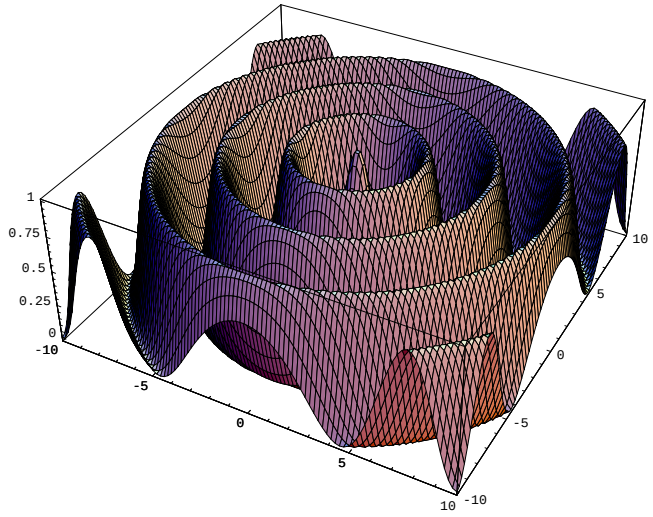


Figure 2.4: A surface plot of the function f_3.

Just in order to get more insight into the coding/decoding scheme, let us take the best string `0111111111111111`. Its representation as integer number is 32767. Computing the decoding function yields

$$-1 + 32767 \cdot \frac{1-(-1)}{65535} = -1 + 0.9999847 = -0.0000153.$$

2.2.3 A Two-Dimensional Function

As next example, we study the function

$$\begin{array}{rcl} f_3 : [-10,10]^2 & \longrightarrow & \mathbb{R} \\ (x,y) & \longmapsto & \frac{1-\sin^2(\sqrt{x^2+y^2})}{1+0.001\cdot(x^2+y^2)}. \end{array}$$

As one can see easily from the plot in Figure 2.4, the function has a global maximum in 0 and a lot of local maxima.

Let us use the coding/decoding scheme as shown in (2.3) and (2.4) for the two components x and y independently with $n = 24$, i.e. $c_{24,[-10,10]}$ and $\tilde{c}_{24,[-10,10]}$ are used as coding and decoding functions, respectively. In order to get a coding for the two-dimensional vector, we can use concatenation and

splitting:

$$\begin{aligned} c_3: \quad [-10,10]^2 &\longrightarrow \{0,1\}^{48} \\ (x,y) &\longmapsto c_{24,[-10,10]}(x)|c_{24,[-10,10]}(y) \end{aligned}$$

$$\begin{aligned} \tilde{c}_3: \quad \{0,1\}^{48} &\longrightarrow [-10,10]^2 \\ s &\longmapsto \left(\tilde{c}_{24,[-10,10]}(s[1:24]), \tilde{c}_{24,[-10,10]}(s[25:48])\right) \end{aligned}$$

If we apply Algorithm 2.5 with $m = 50$, $p_{\mathbf{C}} = 1$, $p_{\mathbf{M}} = 0.01$, we observe that a fairly good solution is reached after 693 generations (at most 34650 evaluations at a search space size of $2.81 \cdot 10^{14}$):

```
Generation 693 max. fitness 0.9999 at (0.0098,0.0000)
#0      000000001000000001000000000000000000000010000000 fitness: 1.00
#1      000001000000011001000110000000000000000010100010 fitness: 0.00
#2      000000001000000000100000000000000000000010000000 fitness: 1.00
#3      000000001000001001000000000000000000000010000000 fitness: 0.97
#4      000000001000001011001000000000000000000010000011 fitness: 0.90
#5      000000101000000001000010000100000000000010000000 fitness: 0.00
#6      000000001000000011000000000000001000000010000011 fitness: 0.00
#7      000000001000000001100000000010000000000110000000 fitness: 0.00
#8      000000001001000001000000000000000000000000100010 fitness: 0.14
#9      000000001000000001000000000000000000000010100010 fitness: 0.78
#10     000000001000011011000000000000000000000010000000 fitness: 0.75
#11     000000001000000001000000000000000000000010100000 fitness: 0.64
#12     000000001000001000010010000000000000000010001001 fitness: 0.56
#13     000000001000001011000000000000000000000010100010 fitness: 0.78
#14     000000001000000001000001000000000000000010000000 fitness: 1.00
#15     000000001000000001100000100000000000000010000000 fitness: 0.00
#16     000000001000001010001000000000000000000010100010 fitness: 0.78
#17     000000001000011011000000000000000000000010000011 fitness: 0.70
#18     000000001000001011001000000000000000000010000011 fitness: 0.90
#19     000000001000011001000010001000010000000010000010 fitness: 0.00
#20     000000001000000001000000000010000000000010100010 fitness: 0.00
#21     000000001000011001100000000000000000010010000000 fitness: 0.00
#22     000000001000000101100000000000000000010010000000 fitness: 0.00
#23     000000001000100001000000000000000000000010000111 fitness: 0.44
#24     000000001000000011000000000000000000000000000000 fitness: 0.64
#25     000000001000000001011000000000010000000010100010 fitness: 0.00
#26     000000001000000001001000000000000000000000100010 fitness: 0.23
#27     000000001000001011000010000000000000000010100010 fitness: 0.78
#28     000000001000001011100010000000000000000010101010 fitness: 0.97
#29     010000001000000011000000000000000010010010000000 fitness: 0.00
#30     000000001000001011000000000000000000000010000011 fitness: 0.90
#31     000000001000011011000000000000000000000011000011 fitness: 0.26
#32     000000001000001001100000000000000000000010000000 fitness: 0.97
#33     000000001001001011000110000000000000000011110100 fitness: 0.87
#34     000000001000000000000000000000000000000010100010 fitness: 0.78
#35     000000001000001011001000000000000000000010000010 fitness: 0.93
#36     000000001000011011000000000000000010000010000001 fitness: 0.00
#37     000000001000001011000000000010000000000010100010 fitness: 0.00
#38     000000001000001011000010010000000000000010000000 fitness: 0.00
#39     000000001000000001000000000010000000000010100010 fitness: 0.00
#40     000000001000001001000110000000000000000011010100 fitness: 0.88
#41     000000001010000001000000000000000000000010000000 fitness: 0.66
#42     000000001000001001100110000000000000000011010100 fitness: 0.88
#43     000000000000000000000000000000000000000010000011 fitness: 0.64
#44     000000001000001011001000000000000000000010100000 fitness: 0.65
#45     000000001000001011000110000000000000000011110100 fitness: 0.81
#46     000000000000000000000000000000000000000010000000 fitness: 0.64
#47     000000001000010001000110000000000000000010000000 fitness: 0.89
#48     000000001000001011000000000000000000000010100011 fitness: 0.84
#49     000000001000000111000000000000000000000010000001 fitness: 0.98
Average Fitness: 0.53
```

Again, we learn from this example that the GA is here for sure much faster than an exhaustive algorithm or a pure random search. The question arises, since f_3 is perfectly smooth, which result we obtain if we apply a conventional method with random selection of the initial value. In this example,

the expectation is obvious: The global maximum $(0, 0)$ is surrounded by a ring of minima at a radius of $\frac{\pi}{2}$. If we apply, for instance, BFGS (Broyden Fletcher Goldfarb Shanno—a very efficient Quasi-Newton method for continuous unconstrained function optimization [7]) with line search, it is likely that convergence to the global maximum is achieved if the initial value is inside that ring, but *only* in this case. If we take the initial value from $[-10, 10]^2$ randomly with uniform distribution, the probability to get a value from the appropriate neighborhood of the global maximum is

$$\frac{\left(\frac{\pi}{2}\right)^2 \cdot \pi}{10 \cdot 10} = \frac{\pi^3}{400} = 0.0775.$$

The expected number of trials until we get an initial value is, therefore, $\frac{1}{0.0775} \approx 13$. In a test implementation, it took 15 trials (random initial values) until the correct global optimum was found by the BFGS method with line search. The total time for all these computations was 5 milliseconds on an SGI O2 (MIPS R5000/180SC). The genetic algorithm, as above, took 1.5 seconds until it found the global optimum with comparable accuracy.

This example shows that GAs are not necessarily fast. Moreover, they are in many cases much slower than conventional methods which involve derivatives. The next example, however, will drastically show us that there are even smooth functions which can be hard for conventional optimization techniques.

2.2.4 Global Smoothness versus Local Perturbations

Consider the function

$$\begin{aligned} f_4 : [-2, 2] &\longrightarrow \mathbb{R} \\ x &\longmapsto e^{-x^2} + 0.01\cos(200x). \end{aligned}$$

As easy to see from Figure 2.5, this function has a clear bell-like shape with small but highly oscillating perturbations. In the first derivative, these oscillations are drastically emphasized (see Figure 2.5):

$$f_4'(x) = -2xe^{-x^2} - 2\sin(200x)$$

We applied the simple GA as in Algorithm 2.5 with $n = 16$, i.e. the pair $\tilde{c}_{16,[-2,2]}/\tilde{c}_{16,[-2,2]}$ as coding/decoding scheme, $m = 10$, $p_{\mathbf{C}} = 1$, and $p_{\mathbf{M}} = 0.005$. The result was that the global maximum at $x = 0$ was found after 9 generations (i.e. at most 90 evaluations of the fitness function) and

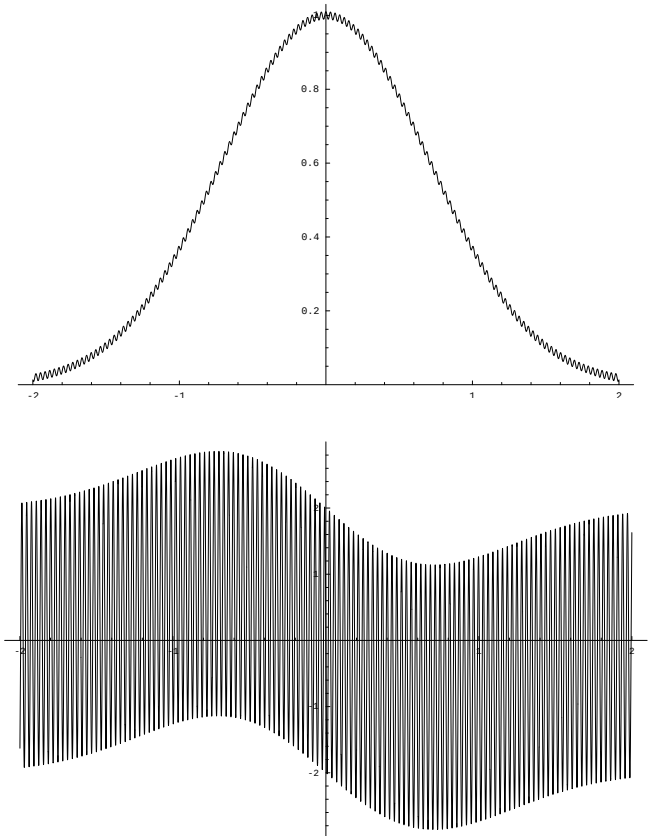


Figure 2.5: The function f_4 (top) and its derivative (bottom).

5 milliseconds computation time, respectively (on the same computer as above).

In order to repeat the above comparison, BFGS with line search and random selection of the initial value was applied to f_4 as well. The global optimum was found after 30 trials (initial values) with perfect accuracy, but 9 milliseconds of computation time.

We see that, depending on the structure of the objective function, a GA can even outperform an acknowledged conventional method which makes use of derivatives.

2.2.5 Discussion

Finally, let us summarize some conclusions about the four examples above:

Algorithm 2.5 is very universal. More or less, the same algorithm has been applied to four fundamentally different optimization tasks.

As seen in 2.2.4, GAs can even be faster in finding global maxima than conventional methods, in particular when derivatives provide misleading information. We should not forget, however, that, in most cases where conventional methods can be applied, GAs are much slower because they do not take auxiliary information like derivatives into account. In these optimization problems, there is no need to apply a GA which gives less accurate solutions after much longer computation time. The enormous potential of GAs lies elsewhere—in optimization of non-differentiable or even discontinuous functions, discrete optimization, and program induction.

Chapter 3

Analysis

> *Although the belief that an organ so perfect as the eye could have been formed by natural selection, is enough to stagger any one; yet in the case of any organ, if we know of a long series of gradations in complexity, each good for its possessor, then, under changing conditions of life, there is no logical impossibility in the acquirement of any conceivable degree of perfection through natural selection.*
>
> Charles R. Darwin

In this remark, Darwin, in some sense, tries to turn around the burden of proof for his theory simply by saying that there is no evidence against it. This chapter is intended to give an answer to the question why genetic algorithms work—in a way which is philosophically more correct than Darwin's. However, we will see that, as in Darwin's theory of evolution, the complexity of the mechanisms makes mathematical analysis difficult and complicated.

For conventional deterministic optimization methods, such as gradient methods, Newton- or Quasi-Newton methods, etc., it is rather usual to have results which guarantee that the sequence of iterations converges to a local optimum with a certain speed or order. For any probabilistic optimization method, theorems of this kind cannot be formulated, because the behavior of the algorithm is not determinable in general. Statements about the convergence of probabilistic optimization methods can only give information about the expected or average behavior. In the case of genetic algorithms, there are a few circumstances which make it even more difficult to investigate their convergence behavior:

- Since a single transition from one generation to the next is a combination of usually three probabilistic operators (selection, crossover, and mutation), the inner structure of a genetic algorithm is rather complicated.
- For each of the involved probabilistic operators, many different variants have been proposed, thus it is not possible to give general convergence results due to the fact that the choice of the operators influences the convergence fundamentally.

In the following, we will not be able to give "hard" convergence theorems, but only a summary of results giving a clue why genetic algorithms work for many problems but not necessarily for all problems. For simplicity, we will restrict to algorithms of type 2.1, i.e. GAs with a fixed number m of binary strings of fixed length n. Unless stated otherwise, no specific assumptions about selection, crossover, or mutation will be made.

Let us briefly reconsider the example in 2.2.1. We saw that the transition from the first to the second generation is given as follows:

Gen. #1	$f(x)$		Gen. #2	$f(x)$
0 1 1 0 1	169		0 1 1 0 0	144
1 1 0 0 0	576	$\Longrightarrow$	1 1 0 0 1	625
0 1 0 0 0	64		1 1 0 1 1	729
1 0 0 1 1	361		1 0 0 0 0	256

It is easy to see that it is advantageous to have a 1 in the first position. In fact, the number of strings having this property increased from 2 in the first to 3 in the second generation. The question arises whether this is a coincidence or simply a clue to the basic principle why GAs work. The answer will be that the latter is the case. In order to investigate these aspects formally, let us make the following definition.

3.1 Definition. A string $H = (h_1, \ldots, h_n)$ over the alphabet $\{0, 1, *\}$ is called a (binary) *schema* of length n. An $h_i \neq *$ is called a *specification* of H, an $h_i = *$ is called *wildcard*.

It is not difficult to see that schemata can be considered as specific subsets of $\{0,1\}^n$ if we consider the following function which maps a schema to its associated subset.

$$\begin{array}{rccl} i: & \{0,1,*\}^n & \longrightarrow & \mathcal{P}(\{0,1\}^n) \\ & H & \longmapsto & \{S \mid \forall 1 \leq i \leq n: \ (h_i \neq *) \Rightarrow (h_i = s_i)\} \end{array}$$

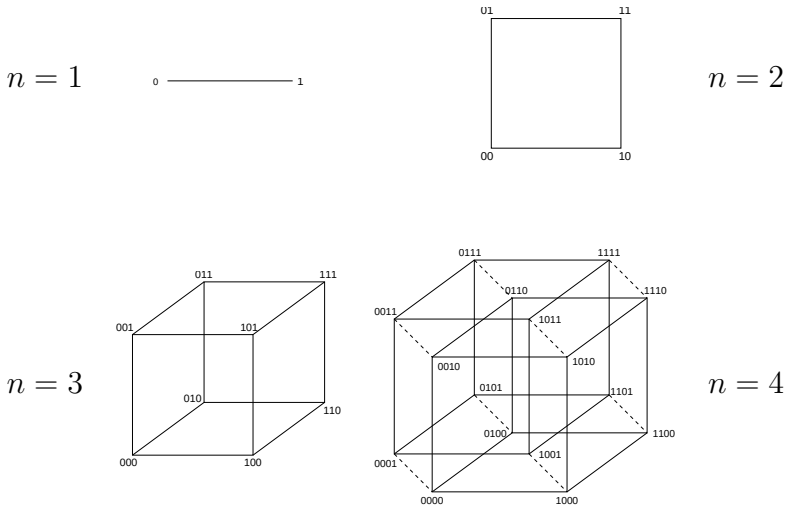


Figure 3.1: Hypercubes of dimensions 1–4.

If we interpret binary strings of length n as hypercubes of dimension n (cf. Figure 3.1), schemata can be interpreted as hyperplanes in these hypercubes (see Figure 3.2 for an example with $n = 3$).

Before turning to the first important result, let us make some fundamental definitions concerning schemata.

3.2 Definition.

1. A string $S = (s_1, \ldots, s_n)$ over the alphabet $\{0, 1\}$ *fulfills* the schema $H = (h_1, \ldots, h_n)$ if and only if it matches H is all non-wildcard positions:
$$\forall i \in \{j \mid h_j \neq *\} : \; s_i = h_i$$
According to the discussion above, we write $S \in H$.

2. The *number of specifications* of a schema H is called *order* and denoted as
$$\mathcal{O}(H) = |\{i \in \{1, \ldots, n\} | h_i \neq *\}|.$$

3. The distance between the first and the last specification
$$\delta(H) = \max\{i | h_i \neq *\} - \min\{i | h_i \neq *\}$$
is called the *defining length* of a schema H.

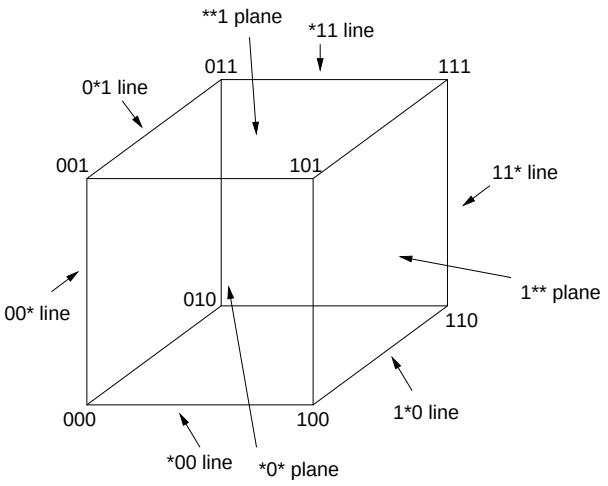


Figure 3.2: A hyperplane interpretation of schemata for $n = 3$.

3.1 The Schema Theorem

In this section, we will formulate and prove *the* fundamental result on the behavior of genetic algorithms—the so-called Schema Theorem. Although being completely incomparable with convergence results for conventional optimization methods, it still provides valuable insight into the intrinsic principles of GAs.

Assume in the following, that we have a genetic algorithm of type 2.1 with proportional selection and an arbitrary but fixed fitness function f. Let us make the following notations:

1. The number of individuals which fulfill H at time step t are denoted as
$$r_{H,t} = |\mathcal{B}_t \cap H| .$$
2. The expression $\bar{f}(t)$ refers to the observed average fitness at time t:
$$\bar{f}(t) = \frac{1}{m} \sum_{i=1}^{m} f(b_{i,t})$$
3. The term $\bar{f}(H,t)$ stands for the observed average fitness of schema H in time step t:
$$\bar{f}(H,t) = \frac{1}{r_{H,t}} \sum_{i \in \{j | b_{j,t} \in H\}} f(b_{i,t})$$

3.3 Theorem (Schema Theorem—Holland 1975). *Assuming we consider a genetic algorithm of type 2.5, the following inequality holds for every schema H:*

$$\mathsf{E}[r_{H,t+1}] \geq r_{H,t} \cdot \frac{\bar{f}(H,t)}{\bar{f}(t)} \cdot \left(1 - p_{\mathbf{C}} \cdot \frac{\delta(H)}{n-1}\right) \cdot (1-p_{\mathbf{M}})^{\mathcal{O}(H)} \tag{3.1}$$

Proof. The probability that we select an individual fulfilling H is (compare with Eq. (2.1))

$$\frac{\sum\limits_{i \in \{j \mid b_{j,t} \in H\}} f(b_{i,t})}{\sum\limits_{i=1}^{m} f(b_{i,t})}. \tag{3.2}$$

This probability does not change throughout the execution of the selection loop. Moreover, every of the m individuals is selected completely independently from the others. Hence, the number of selected individuals, which fulfill H, is binomially distributed with sample amount m and the probability in (3.2). We obtain, therefore, that the expected number of selected individuals fulfilling H is

$$\begin{aligned} m \cdot \frac{\sum\limits_{i \in \{j \mid b_{j,t} \in H\}} f(b_{i,t})}{\sum\limits_{i=1}^{m} f(b_{i,t})} &= m \cdot \frac{r_{H,t}}{r_{H,t}} \cdot \frac{\sum\limits_{i \in \{j \mid b_{j,t} \in H\}} f(b_{i,t})}{\sum\limits_{i=1}^{m} f(b_{i,t})} \\ &= r_{H,t} \cdot \frac{\frac{\sum\limits_{i \in \{j \mid b_{j,t} \in H\}} f(b_{i,t})}{r_{H,t}}}{\frac{\sum\limits_{i=1}^{m} f(b_{i,t})}{m}} \qquad = r_{H,t} \cdot \frac{\bar{f}(H,t)}{\bar{f}(t)} \end{aligned}$$

If two individuals are crossed, which both fulfill H, the two offsprings again fulfill H. The number of strings fulfilling H can only decrease if one string, which fulfills H, is crossed with a string which does not fulfill H, but, obviously, only in the case that the cross site is chosen somewhere in between the specifications of H. The probability that the cross site is chosen within the defining length of H is

$$\frac{\delta(H)}{n-1}.$$

Hence the survival probability $p_{\mathbf{S}}$ of H, i.e. the probability that a string fulfilling H produces an offspring also fulfilling H, can be estimated as follows (crossover is only done with probability $p_{\mathbf{C}}$):

$$p_{\mathbf{S}} \geq 1 - p_{\mathbf{C}} \cdot \frac{\delta(H)}{n-1}$$

Selection and crossover are carried out independently, so we may compute the expected number of strings fulfilling H after crossover simply as

$$\frac{\bar{f}(H,t)}{\bar{f}(t)} \cdot r_{H,t} \cdot p_{\mathbf{S}} \geq \frac{\bar{f}(H,t)}{\bar{f}(t)} \cdot r_{H,t} \cdot \left(1 - p_{\mathbf{C}} \cdot \frac{\delta(H)}{n-1}\right).$$

After crossover, the number of strings fulfilling H can only decrease if a string fulfilling H is altered by mutation at a specification of H. The probability that all specifications of H remain untouched by mutation is obviously

$$(1 - p_{\mathbf{M}})^{\mathcal{O}(H)}.$$

Applying the same argument like above, Equation (3.1) follows. □

The arguments in the proof of the Schema Theorem can be applied analogously to many other crossover and mutation operations.

3.4 Corollary. *For a genetic algorithm of type 2.1 with roulette wheel selection, the inequality holds*

$$\mathsf{E}[r_{H,t+1}] \geq \frac{\bar{f}(H,t)}{\bar{f}(t)} \cdot r_{H,t} \cdot P_{\mathbf{C}}(H) \cdot P_{\mathbf{M}}(H) \tag{3.3}$$

for any schema H, where $P_{\mathbf{C}}(H)$ is a constant only depending on the schema H and the crossover method and $P_{\mathbf{M}}(H)$ is a constant which solely depends on H and the involved mutation operator. For the variants discussed in 2.1.2 and 2.1.3, we can give the following estimates:

$P_{\mathbf{C}}(H) = 1 - p_{\mathbf{C}} \cdot \frac{\delta(H)}{n-1}$	*one-point crossing over*
$P_{\mathbf{C}}(H) = 1 - p_{\mathbf{C}} \cdot \left(1 - \left(\frac{1}{2}\right)^{\mathcal{O}(H)}\right)$	*uniform crossing over*
$P_{\mathbf{C}}(H) = 1 - p_{\mathbf{C}}$	*any other crossing over method*
$P_{\mathbf{M}}(H) = (1 - p_{\mathbf{M}})^{\mathcal{O}(H)}$	*bitwise mutation*
$P_{\mathbf{M}}(H) = 1 - p_{\mathbf{M}} \cdot \frac{\mathcal{O}(H)}{n}$	*inversion of a single bit*
$P_{\mathbf{M}}(H) = 1 - p_{\mathbf{M}}$	*bitwise inversion*
$P_{\mathbf{M}}(H) = 1 - p_{\mathbf{M}} \cdot \frac{\lvert H\rvert}{2^n}$	*random selection*

Even the inattentive reader must have observed that the Schema Theorem is somehow different from convergence results for conventional optimization methods. It seems that this result raises more questions than it is ever able to answer. At least one insight is more or less obvious: Schemata with above-average fitness and short defining length—let us put aside the generalizations

made in Corollary 3.4 for our following studies—tend to produce more offsprings than others. For brevity, let us call such schemata *building blocks*. It will become clear in a moment why this term is appropriate. If we assume that the quotient

$$\frac{\bar{f}(H,t)}{\bar{f}(t)}$$

is approximately stationary, i.e. independent of time and the actual generations, we immediately see that the number of strings, which belong to above-average schemata with short defining lengths, grows exponentially (like a geometric sequence).

This discovery poses the question whether it is a wise strategy to let above-average schemata receive an exponentially increasing number of trials and, if the answer is yes, why this is the case. In 3.1.1, we will try to shed more light on this problem.

There is one other fundamental question we have yet not touched at all: Undoubtedly, GAs operate on binary strings, but not on schemata. The Schema Theorem, more or less, provides an observation of all schemata, which all grow and decay according to their observed average fitness values in parallel. What is actually the interpretation of this behavior and why is this a good thing to do? Subsection 3.1.2 is devoted to this topic.

Finally, one might ask where the crucial role of schemata with above-average fitness and short defining length comes from and what the influence of the fitness function and the coding scheme is. We will attack these problems in 3.2.

3.1.1 The Optimal Allocation of Trials

The Schema Theorem has provided the insight that building blocks receive exponentially increasing trials in future generations. The question remains, however, why this could be a good strategy. This leads to an important and well-analyzed problem from statistical decision theory—the *two-armed bandit problem* and its generalization, the *k-armed bandit problem*. Although this seems like a detour from our main concern, we shall soon understand the connection to genetic algorithms.

Suppose we have a gambling machine with two slots for coins and two arms. The gambler can deposit the coin either into the left or the right slot. After pulling the corresponding arm, either a reward is payed or the coin is lost. For mathematical simplicity, we just work with outcomes, i.e. the difference between the reward (which can be zero) and the value of the coin.

Let us assume that the left arm produces an outcome with mean value μ_1 and a variance σ_1^2 while the right arm produces an outcome with mean value μ_2 and variance σ_2^2. Without loss of generality, although the gambler does not know this, assume that $\mu_1 \geq \mu_2$.

The question arises which arm should be played. Since we do not know beforehand which arm is associated with the higher outcome, we are faced with an interesting dilemma. Not only must me make a sequence of decisions which arm to play, we have to collect, at the same time, information about which is the better arm. This trade-off between exploration of knowledge and its exploitation is the key issue in this problem and, as turns out later, in genetic algorithms, too.

A simple approach to this problem is to separate exploration from exploitation. More specifically, we could perform a single experiment at the beginning and thereafter make an irreversible decision that depends on the results of the experiment. Suppose we have N coins. If we first allocate an equal number n (where $2n \leq N$) of trials to both arms, we could allocate the remaining $N - 2n$ trials to the observed better arm. Assuming we know all involved parameters [13], the expected loss is given as

$$L(N, n) = (\mu_1 - \mu_2) \cdot \big((N - n)q(n) + n(1 - q(n))\big)$$

where $q(n)$ is the probability that the worst arm is the observed best arm after the $2n$ experimental trials. The underlying idea is obvious: In case that we observe that the worse arm is the best, which happens with probability $q(n)$, the total number of trials allocated to the right arm is $N - n$. The loss is, therefore, $(\mu_1 - \mu_2) \cdot (N - n)$. In the reverse case that we actually observe that the best arm is the best, which happens with probability $1 - q(n)$, the loss is only what we get less because we played the worse arm n times, i.e. $(\mu_1 - \mu_2) \cdot n$. Taking the central limit theorem into account, we can approximate $q(n)$ with the tail of a normal distribution:

$$q(n) \approx \frac{1}{\sqrt{2\pi}} \cdot \frac{e^{-c^2/2}}{c}, \qquad \text{where } c = \frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2}} \cdot \sqrt{n}$$

Now we have to specify a reasonable experiment size n. Obviously, if we choose $n = 1$, the obtained information is potentially unreliable. If we choose, however, $n = \frac{N}{2}$ there are no trials left to make use of the information gained through the experimental phase. What we see is again the trade-off between exploitation with almost no exploration ($n = 1$) and exploration without exploitation ($n = \frac{N}{2}$). It does not take a Nobel price winner to see that the optimal way is somewhere in the middle. Holland [16] has studied this

problem is very detail. He came to the conclusion that the optimal strategy is given by the following equation:

$$n^* \approx b^2 \ln\left(\frac{N^2}{8\pi b^4 \ln N^2}\right), \qquad \text{where } b = \frac{\sigma_1}{\mu_1 - \mu_2}.$$

Making a few transformations, we obtain that

$$N - n^* \approx \sqrt{8\pi b^4 \ln N^2} \cdot e^{n^*/2b^2},$$

i.e. the optimal strategy is to allocate slightly more than an exponentially increasing number of trials to the observed best arm. Although no gambler is able to apply this strategy in practice, because it requires knowledge of the mean values μ_1 and μ_2, we still have found an important bound of performance a decision strategy should try to approach.

A genetic algorithm, although the direct connection is not yet fully clear, actually comes close to this ideal, giving at least an exponentially increasing number trials to the observed best building blocks. However, one may still wonder how the two-armed bandit problem and GAs are related. Let us consider an arbitrary string position. Then there are two schemata of order one which have their only specification in this position. According to the Schema Theorem, the GA implicitly decides between these two schemata, where only incomplete data are available (observed average fitness values). In this sense, a GA solves a lot of two-armed problems in parallel.

The Schema Theorem, however, is not restricted to schemata with an order of 1. Looking at competing schemata (different schemata which are specified in the same positions), we observe that a GA is solving an enormous number of k-armed bandit problems in parallel. The k-armed bandit problem, although much more complicated, is solved in an analogous way [13, 16]—the observed better alternatives should receive an exponentially increasing number of trials. *This is exactly what a genetic algorithm does!*

3.1.2 Implicit Parallelism

So far we have discovered two distinct, seemingly conflicting views of genetic algorithms:

1. The algorithmic view that GAs operate on strings.

2. The schema-based interpretation.

So, we may ask what a GA really processes, strings or schemata? The answer is surprising: *Both*. Nowadays, the common interpretation is that a GA processes an enormous amount of schemata implicitly. This is accomplished by exploiting the currently available, incomplete information about these schemata continuously, while trying to explore more information about them and other, possibly better schemata.

This remarkable property is commonly called the *implicit parallelism* of genetic algorithms.

A simple GA as presented in Chapter 2 processes only m structures in one time step, without any memory or bookkeeping about the previous generations. We will now try to get a feeling how many schemata a GA actually processes.

Obviously, there are 3^n schemata of length n. A single binary string fulfills n schemata of order 1, $\binom{n}{2}$ schemata of order 2, in general, $\binom{n}{k}$ schemata of order k. Hence, a string fulfills

$$\sum_{k=1}^{n} \binom{n}{k} = 2^n$$

schemata. Thus, for any generation, we obtain that there are between 2^n and $m \cdot 2^n$ schemata which have at least one representative. But how many schemata are actually processed? Holland [16] has given an estimation of the quantity of schemata that are taken over to the next generation. Although the result seems somewhat clumsy, it still provides important information about the large quantity of schemata which are inherently processed in parallel while, in fact, considering a relatively small quantity of strings.

3.5 Theorem. *Consider a randomly generated start population of a simple GA of type 2.5 and let $\varepsilon \in (0,1)$ be a fixed error bound. Then schemata of length*

$$l_s < \varepsilon \cdot (n-1) + 1$$

have a probability of at least $1 - \varepsilon$ to survive one-point crossover (compare with the proof of the Schema Theorem). If the population size is chosen as $m = 2^{l_s/2}$, the number of schemata, which survive for the next generation, is of order $\mathcal{O}(m^3)$.

3.2 Building Blocks and the Coding Problem

We have already introduced the term "building block" for a schema with high average fitness and short defining length (implying small order). Now

it is time to explain why this notation is appropriate. We have seen in the Schema Theorem and 3.1.1 that building blocks receive an exponentially increasing number of trials. The considerations in 3.1.2 have demonstrated that a lot of schemata (including building blocks) are evaluated implicitly and in parallel. What we still miss is the link to performance, i.e. convergence. Unfortunately, there is no complete theory which gives a clear answer, just a hypothesis.

3.6 Building Block Hypothesis.
A genetic algorithm creates stepwise better solutions by recombining, crossing, and mutating short, low-order, high-fitness schemata.

Goldberg [13] has found a good comparison for pointing out the main assertion of this hypothesis:

> *Just as a child creates magnificent fortresses through the arrangement of simple blocks of wood, so does a genetic algorithm seek near optimal performance through the juxtaposition of short, low-order, high-performance schemata, or building blocks.*

This seems a reasonable assumption and fits well to the Schema Theorem. The question is now if and when it holds. We first consider an affine linear fitness function

$$f(s) = a + \sum_{i=1}^{n} c_i \cdot s[i],$$

i.e. the fitness is computed as a linear combination of all genes. It is easy to see that the optimal value can be determined for every gene independently (only depending on the sign of the scaling factors c_i).

Conversely, let us consider a needle-in-haystack problem as the other extreme:

$$f(x) = \begin{cases} 1 & \text{if } x = x_0 \\ 0 & \text{otherwise} \end{cases}$$

Obviously, there is a single string x_0 which is the optimum, but all other strings have equal fitness values. In this case, certain values on single positions (schemata) do not provide any information for guiding an optimization algorithm to the global optimum.

In the linear case, the building block hypothesis seems justified. For the second function, however, it cannot be true, since there is absolutely no information available which could guide a GA to the global solution through partial, sub-optimal solutions. In other words, the more the positions can be

judged independently, the easier it is for a GA. On the other hand, the more positions are coupled, the more difficult it is for a GA (and for any other optimization method).

Biologists have come up with a special term for this kind of nonlinearity—*epistasis*. Empirical studies have shown that GAs are appropriate for problems with medium epistasis. While almost linear problems (i.e. with low epistasis) can be solved much more efficiently with conventional methods, highly epistatic problems cannot be solved efficiently at all [15].

We will now come to a very important question which is strongly related to epistasis: Do good parents always produce children of comparable or even better fitness (the building block hypothesis implicitly relies on this)? In natural evolution, this is almost always true. For genetic algorithms, this is not so easy to guarantee. The disillusioning fact is that the user has to take care of an appropriate coding in order to make this fundamental property hold.

In order to get a feeling for optimization tasks which could foul a GA, we will now try to construct a very simple misleading example. Apparently, for $n = 1$, no problems can occur, the two-bit problem $n = 2$ is the first. Without loss of generality, assume that 11 is the global maximum. Next we introduce the element of deception necessary to make this a tough problem for a simple GA. To do this, we want a problem where one or both of the suboptimal order-1 schemata are better than the optimal order-1 schemata. Mathematically, we want one or both of the following conditions to be fulfilled:

$$f(0*) > f(1*), \tag{3.4}$$
$$f(*0) > f(*1), \tag{3.5}$$

i.e.

$$\frac{f(00) + f(01)}{2} > \frac{f(10) + f(11)}{2}, \tag{3.6}$$
$$\frac{f(00) + f(10)}{2} > \frac{f(01) + f(11)}{2}. \tag{3.7}$$

Both expressions cannot hold simultaneously, since this would contradict to the maximality of 11. Without any loss of generality, we choose the first condition for our further considerations.

In order to put the problem into closer perspective, we normalize all fitness values with respect to the complement of the global optimum:

$$r = \frac{f(11)}{f(00)} \qquad c = \frac{f(01)}{f(00)} \qquad c' = \frac{f(10)}{f(00)}$$

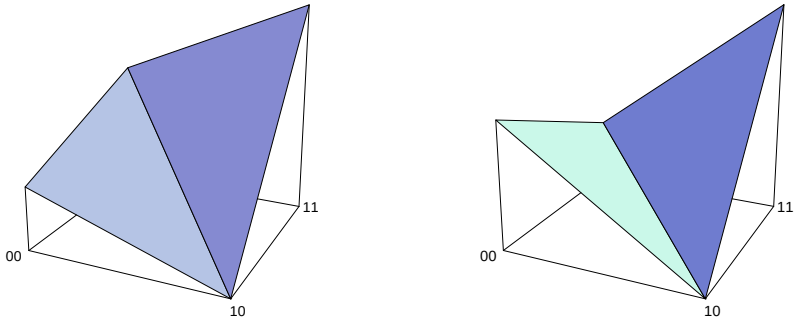


Figure 3.3: Minimal deceptive problems of type I (left) and type II (right).

The maximality condition implies:

$$r > c \qquad r > 1 \qquad r > c'$$

The deception conditions (3.4) and (3.6), respectively, read as follows:

$$r < 1 + c - c'$$

From these conditions, we can conclude the following facts:

$$c' < 1 \qquad c' < c$$

We see that there are two possible types of minimal deceptive two-bit problems based on (3.4):

$$\begin{array}{lll} \text{Type I:} & f(\mathtt{01}) > f(\mathtt{00}) & (c > 1) \\ \text{Type II:} & f(\mathtt{01}) \leq f(\mathtt{00}) & (c \leq 1) \end{array}$$

Figure 3.3 shows sketches of these two fundamental types of deceptive problems. It is easy to see that both fitness functions are nonlinear. In this sense, epistasis is again the bad property behind the deception in these problems.

3.2.1 Example: The Traveling Salesman Problem

We have already mentioned that it is essential for a genetic algorithm that good individuals produce comparably good or even better offsprings. We will now study a non-trivial example which is well-known in logistics—the *traveling salesman problem* (TSP). Assume we are given a finite set of vertices/cities $\{v_1, \ldots, v_N\}$. For every pair of cities (v_i, v_j), the distance $D_{i,j}$ is

known (i.e. we have a symmetric $K \times K$ distance matrix). What we want to find is a permutation $(p_1, \ldots, p_N)$ such that the total way—the sum of distances—is minimal:

$$f(p) = \sum_{i=1}^{N-1} D_{p_i,p_{i+1}} + D_{p_N,p_1}$$

This problem appears in route planning, VLSI design, etc.

For solving the TSP with a genetic algorithm, we need a coding, a crossover method, and a mutation method. All these three components should work together such the building block hypothesis is satisfiable.

First of all, it seems promising to encode a permutation as a string of integer numbers where entry no. i refers to the i-th city which is visited. Since every number between 1 and K may only occur exactly once—otherwise we do not have a complete tour—the conventional one-point crossover method is not inappropriate like all other methods we have considered. If we put aside mutation for a moment, the key problem remains how to define an appropriate crossover operation for the TSP.

Partially Mapped Crossover

Partially mapped crossover (PMX) aims at keeping as many positions from the parents as possible. To achieve this goal, a substring is swapped like in two-point crossover and the values are kept in all other non-conflicting positions. The conflicting positions are replaced by the values which were swapped to the other offspring. An example:

$$p_1 = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9)$$
$$p_2 = (4\ 5\ 2\ 1\ 8\ 7\ 6\ 9\ 3)$$

Assume that positions 4–7 are selected for swapping. Then the two offsprings are given as follows if we omit the conflicting positions:

$$o_1 = (*\ 2\ 3|1\ 8\ 7\ 6|*\ 9)$$
$$o_2 = (*\ *\ 2|4\ 5\ 6\ 7|9\ 3)$$

Now we take the conflicting positions and fill in what was swapped to the other offspring. For instance, 1 and 4 were swapped. Therefore, we have to replace the 1 in the first position of o_1 by 4, and so on:

$$o_1 = (4\ 2\ 3\ 1\ 8\ 7\ 6\ 5\ 9)$$
$$o_2 = (1\ 8\ 2\ 4\ 5\ 6\ 7\ 9\ 3)$$

Order Crossover

Order crossover (OX) relies on the idea that the order of cities is more important than their absolute positions in the strings. Like PMX, it swaps two aligned substrings. The computation of the remaining substrings of the offsprings, however, is done in a different way. In order to illustrate this rather simple idea, let us consider the same example (p_1, p_2) as above. Simply swapping the two substrings and omitting all other positions, we obtain the following:

$$o_1 = (* \ * \ *|1 \ 8 \ 7 \ 6|* \ *)$$
$$o_2 = (* \ * \ *|4 \ 5 \ 6 \ 7|* \ *)$$

For computing the open positions of o_2, let us write down the positions in p_1, but starting from the position after the second crossover site:

$$9 \ 3 \ 4 \ 5 \ 2 \ 1 \ 8 \ 7 \ 6$$

If we omit all those values which are already in the offspring after the swapping operation (4, 5, 6, and 7), we end up in the following shortened list:

$$9 \ 3 \ 2 \ 1 \ 8$$

Now we insert this list into o_2 starting after the second crossover site and we obtain

$$o_2 = (2 \ 1 \ 8 \ 4 \ 5 \ 6 \ 7 \ 9 \ 3).$$

Applying the same technique to o_1 produces the following result:

$$o_1 = (3 \ 4 \ 5 \ 1 \ 8 \ 7 \ 6 \ 9 \ 2).$$

Cycle Crossover

PMX and OX have in common that they usually introduce alleles outside the crossover sites which have not been present in either parent. For instance, the 3 in the first position of o_1 in the OX example above neither appears in p_1 nor in p_2. Cycle crossover (CX) tries to overcome this problem—the goal is to guarantee that every string position in any offspring comes from one of the two parents. We consider the following example:

$$p_1 = (1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9)$$
$$p_2 = (4 \ 1 \ 2 \ 8 \ 7 \ 6 \ 9 \ 3 \ 5)$$

We start from the first position of o_1:

$$o_1 = (1\ *\ *\ *\ *\ *\ *\ *\ *)$$
$$o_2 = (*\ *\ *\ *\ *\ *\ *\ *\ *)$$

Then o_2 may only have a 4 in the first position, because we do not want new values to be introduced there:

$$o_1 = (1\ *\ *\ *\ *\ *\ *\ *\ *)$$
$$o_2 = (4\ *\ *\ *\ *\ *\ *\ *\ *)$$

Since the 4 is already fixed for o_2 now, we have to keep it in the same position for o_1 in order to guarantee that no new positions for the 4 are introduced. We have to keep the 8 in the fourth position of o_2 automatically for the same reason:

$$o_1 = (1\ *\ *\ 4\ *\ *\ *\ *\ *)$$
$$o_2 = (4\ *\ *\ 8\ *\ *\ *\ *\ *)$$

This process must be repeated until we end up in a value which have previously be considered, i.e. we have completed a cycle:

$$o_1 = (1\ 2\ 3\ 4\ *\ *\ *\ 8\ *)$$
$$o_2 = (4\ 1\ 2\ 8\ *\ *\ *\ 3\ *)$$

For the second cycle, we can start with a value from p_2 and insert it into o_1:

$$o_1 = (1\ 2\ 3\ 4\ 7\ *\ *\ 8\ *)$$
$$o_2 = (4\ 1\ 2\ 8\ 5\ *\ *\ 3\ *)$$

After the same tedious computations, we end up with the following:

$$o_1 = (1\ 2\ 3\ 4\ 7\ *\ 9\ 8\ 5)$$
$$o_2 = (4\ 1\ 2\ 8\ 5\ *\ 7\ 3\ 9)$$

The last cycle is a trivial one (6–6) and the final offsprings are given as follows:

$$o_1 = (1\ 2\ 3\ 4\ 7\ 6\ 9\ 8\ 5)$$
$$o_2 = (4\ 1\ 2\ 8\ 5\ 6\ 7\ 3\ 9)$$

In case that the two parents form one single cycle, no crossover can take place.

It is worth to mention that empirical studies have shown that OX gives 11% better results and PMX and 15 % better results than CX. In general, the performance of all three methods is rather poor.

A Coding with Reference List

Now we discuss an approach which modifies the coding scheme such that all conventional crossover methods are applicable. It works as follows: A reference list is initialized with $\{1, \ldots, N\}$. Starting from the first position, we take the index of the actual element in the list which is then removed from the list. An example:

$$p = (1\ 2\ 4\ 3\ 8\ 5\ 9\ 6\ 7)$$

The first element is 1 and its position in the reference list $\{1, \ldots, 9\}$ is 1. Hence,

$$\tilde{p} = (1\ *\ *\ *\ *\ *\ *\ *\ *).$$

The next entry is 2 and its position in the remaining reference list $\{2, \ldots, 9\}$ is 1 and we can go further:

$$\tilde{p} = (1\ 1\ *\ *\ *\ *\ *\ *\ *).$$

The third allele is 4 and its position in the remaining reference list $\{3, \ldots, 9\}$ is 2 and we obtain:

$$\tilde{p} = (1\ 1\ 2\ *\ *\ *\ *\ *\ *).$$

It is left to the reader as an exercise to continue with this example. He/she will come to the conclusion that

$$\tilde{p} = (1\ 1\ 2\ 1\ 4\ 1\ 3\ 1\ 1).$$

The attentive reader might have guessed that a string in this coding is a valid permutation if and only if the following holds for all $1 \leq i \leq N$:

$$1 \leq \tilde{p}_i \leq N - i + 1$$

Since this criterion applies only to single string positions, completely independently from other positions, it can never be violated by any crossover method which we have discussed for binary strings. This is, without any doubt, a good property. The next example, however, drastically shows that one-point crossover produces more or less random values behind the crossover site:

$$\begin{aligned}
\tilde{p}_1 &= (1\ 1\ 2\ 1|4\ 1\ 3\ 1\ 1) & p_1 &= (1\ 2\ 4\ 3\ 8\ 5\ 9\ 6\ 7)\\
\tilde{p}_2 &= (5\ 1\ 5\ 5|5\ 3\ 3\ 2\ 1) & p_2 &= (5\ 1\ 7\ 8\ 9\ 6\ 4\ 3\ 2)\\
\tilde{o}_1 &= (1\ 1\ 2\ 1|5\ 3\ 3\ 2\ 1) & o_1 &= (1\ 2\ 4\ 3\ 9\ 8\ 7\ 6\ 5)\\
\tilde{o}_2 &= (5\ 1\ 5\ 5|4\ 1\ 3\ 1\ 1) & o_2 &= (5\ 1\ 7\ 8\ 6\ 2\ 9\ 4\ 3)
\end{aligned}$$

Edge Recombination

Absolute string positions do not have any meaning at all—we may start a given round-trip at a different city and still observe the same total length. The order, as in OX, already has a greater importance. However, it is not order itself that makes a trip efficient, it is the set of edges between cities, where it is obviously not important in which direction we pass such an edge. In this sense, the real building blocks in the TS problem are hidden in the connections between cities. A method called Edge Recombination (ER) rests upon this discovery. The basic idea is to cache information about all edges and to compute an offspring from this edge list.

We will study the basic principle with the help of a simple example:

$$p_1 = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9)$$
$$p_2 = (4\ 1\ 2\ 8\ 7\ 6\ 9\ 3\ 5)$$

The first thing is to compute all vertices which occur in the two parents. What we obtain is a list of 2–4 cities with which every city is connected:

$$\begin{array}{lcl} 1 & \rightarrow & 2, 4, 9 \\ 2 & \rightarrow & 1, 3, 8 \\ 3 & \rightarrow & 2, 4, 5, 9 \\ 4 & \rightarrow & 1, 3, 5 \\ 5 & \rightarrow & 3, 4, 6 \\ 6 & \rightarrow & 5, 7, 9 \\ 7 & \rightarrow & 6, 8 \\ 8 & \rightarrow & 2, 7, 9 \\ 9 & \rightarrow & 1, 3, 6, 8 \end{array}$$

We start from the city with the lowest number of neighbors (7 in this example), put it into the offspring, and erase it from all adjacency lists. From 7, we have two possibilities to move next—6 and 8. We always take the one with the smaller number of neighbors. If these numbers are equal, random selection takes place. This procedure must be repeated until the permutation is ready or a conflict occurs (no edges left, but permutation not yet complete). Empirical studies have shown that the probability not to run into a conflict is about 98%. This probability is high enough to have a good chance when trying it a second time. Continuing the example, the following offspring could be obtained:

$$o = (7\ 6\ 5\ 4\ 1\ 9\ 8\ 2\ 3)$$

There are a few variants for improving the convergence of a GA with ER. First of all, it seems reasonable to mark all edges which occur in both parents and to favor them in the selection of the next neighbor. Moreover, it could be helpful to incorporate information about the lengths of single edges.

3.3 Concluding Remarks

In this chapter, we have collected several important results which provide valuable insight into the intrinsic principles of genetic algorithms. These insights were not given as hard mathematical results, but only as a loose collection of interpretations. In order to bring a structure into this mess, let us summarize our achievements:

1. Short, low-order schemata with above-average fitness (building blocks) receive an exponentially increasing number of trials. By the help of a detour to the two-armed bandit problem, we have seen that this is a near-optimal strategy.

2. Although a genetic algorithm only processes m structures at a time, it implicitly accumulates and exploits information about an enormous number of schemata in parallel.

3. We were tempted to believe that a genetic algorithm produces solutions by the juxtaposition of small efficient parts—the building blocks. Our detailed considerations have shown, however, that this good property can only hold if the coding is chosen properly. One sophisticated example, the TSP, has shown how difficult this can be for non-trivial problems.

Chapter 4

Variants

> *Ich möchte aber behaupten, daß die Experimentiermethode der Evolution gleichfalls einer Evolutions unterliegt. Es ist nämlich nicht nur die momentane Lebensleistung eines Individuums für das Überleben der Art wichtig; nach mehreren Generationen wird auch die bessere Vererbungs-Strategie, die eine schnellere Umweltanpassung zustandebringt, ausgelesen und weiterentwickelt.*
>
> Ingo Rechenberg

As Rechenberg pointed out correctly [25], the mechanisms behind evolution themselves are subject to evolution. The diversity and the stage of development of nature as we see it today would have never been achieved only with asexual reproduction. It is exactly the sophistication of genetic mechanisms which allowed faster and faster adaptation of genetic material. So far, we have only considered a very simple class of GAs. This chapter is intended to provide an overview of more sophisticated variants.

4.1 Messy Genetic Algorithms

In a "classical" genetic algorithm, the genes are encoded in a fixed order. The meaning of a single gene is determined by its position inside the string. We have seen in the previous chapter that a genetic algorithm is likely to converge well if the optimization task can be divided into several short building blocks. What, however, happens if the coding is chosen such that couplings occur between distant genes? Of course, one-point crossover tends to disadvantage long schemata (even if they have low order) over short ones.

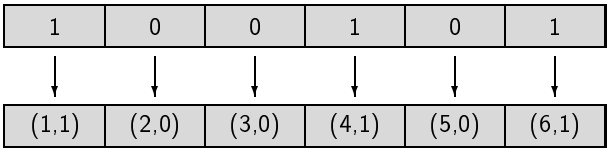


Figure 4.1: A messy coding.

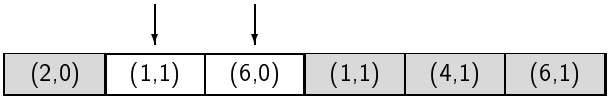


Figure 4.2: Positional preference: Genes with index 1 and 6 occur twice, the first occurrences are used.

Messy genetic algorithms try to overcome this difficulty by using a variable-length, position-independent coding. The key idea is to append an index to each gene which allows to identify its position [14, 15]. A gene, therefore, is no longer represented as a single allele value and a fixed position, but as a pair of an index and an allele. Figure 4.1 shows how this "messy" coding works for a string of length 6.

Since the genes can be identified uniquely by the help of the index, genes may swapped arbitrarily without changing the meaning of the string. With appropriate genetic operations, which also change the order of the pairs, the GA could possibly group coupled genes together automatically.

Due to the free arrangement of genes and the variable length of the encoding, we can, however, run into problems which do not occur in a simple GA. First of all, it can happen that there are two entries in a string which correspond to the same index, but have conflicting alleles. The most obvious way to overcome this "over-specification" is positional preference—the first entry which refers to a gene is taken. Figure 4.2 shows an example.

The reader may have observed that the genes with indices 3 and 5 do not occur at all in the example in Figure 4.2. This problem of "under-specification" is more complicated and its solution is not as obvious as for over-specification. Of course, a lot of variants are reasonable. One approach could be to check all possible combinations and to take the best one (for k missing genes, there are 2^k combinations). With the objective to reduce this effort, Goldberg et al. [14] have suggested to use so-called competitive

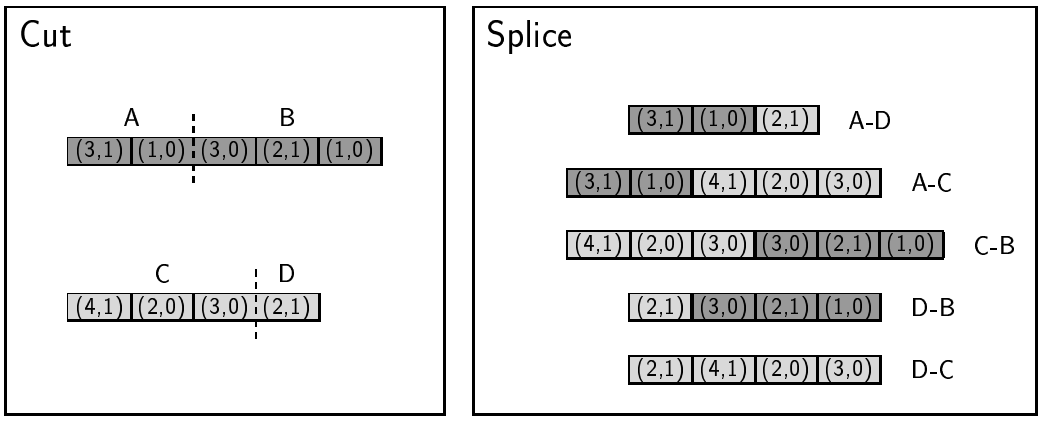


Figure 4.3: The cut and splice operation. There are 12 possible ways to splice the four parts. This example shows five of them.

templates for finding specifications for k missing genes. It is nothing else than applying a local hill climbing method with random initial value to the k missing genes.

While messy GAs usually work with the same mutation operator as simple GAs (every allele is altered with a low probability $p_{\mathbf{M}}$), the crossover operator is replaced by a more general cut and splice operator which also allows to mate parents with different lengths. The basic idea is to choose cut sites for both parents independently and to splice the four fragments. Figure 4.3 shows an example.

4.2 Alternative Selection Schemes

Depending on the actual problem, other selection schemes than the roulette wheel can be useful:

Linear rank selection: In the beginning, the potentially good individuals sometimes fill the population too fast which can lead to premature convergence into local maxima. On the other hand, refinement in the end phase can be slow since the individuals have similar fitness values. These problems can be overcome by taking the rank of the fitness values as the basis for selection instead of the values themselves.

Tournament selection: Closely related to problems above, it can be better

not to use the fitness values themselves. In this scheme, a small group of individuals is sampled from the population and the individual with best fitness is chosen for reproduction. This selection scheme is also applicable when the fitness function is given in implicit form, i.e. when we only have a comparison relation which determines which of two given individuals is better.

Moreover, there is one "plug-in" which is frequently used in conjunction with any of the three selection schemes we know so far—*elitism*. The idea is to avoid that the observed best-fitted individual dies out just by selecting it for the next generation without any random experiment. Elitism is widely used for speeding up the convergence of a GA. It should, however, be used with caution, because it can lead to premature convergence.

4.3 Adaptive Genetic Algorithms

Adaptive genetic algorithms are GAs whose parameters, such as the population size, the crossing over probability, or the mutation probability are varied while the GA is running (e.g. see [8]). A simple variant could be the following: The mutation rate is changed according to changes in the population; the longer the population does not improve, the higher the mutation rate is chosen. Vice versa, it is decreased again as soon as an improvement of the population occurs.

4.4 Hybrid Genetic Algorithms

As they use the fitness function only in the selection step, genetic algorithms are blind optimizers which do not use any auxiliary information such as derivatives or other specific knowledge about the special structure of the objective function. If there is such knowledge, however, it is unwise and inefficient not to make use of it. Several investigations have shown that a lot of synergism lies in the combination of genetic algorithms and conventional methods.

The basic idea is to divide the optimization task into two complementary parts. The coarse, global optimization is done by the GA while local refinement is done by the conventional method (e.g. gradient-based, hill climbing, greedy algorithm, simulated annealing, etc.). A number of variants is reasonable:

1. The GA performs coarse search first. After the GA is completed, local refinement is done.

2. The local method is integrated in the GA. For instance, every K generations, the population is doped with a locally optimal individual.

3. Both methods run in parallel: All individuals are continuously used as initial values for the local method. The locally optimized individuals are re-implanted into the current generation.

4.5 Self-Organizing Genetic Algorithms

As already mentioned, the reproduction methods and the representations of the genetic material were adapted through the billions of years of evolution [25]. Many of these adaptations were able to increase the speed of adaptation of the individuals. We have seen several times that the choice of the coding method and the genetic operators is crucial for the convergence of a GA. Therefore, it is promising not to encode only the raw genetic information, but also some additional information, for example, parameters of the coding function or the genetic operators. If this is done properly, the GA could find its own optimal way for representing and manipulating data automatically.

Chapter 5

Tuning of Fuzzy Systems Using Genetic Algorithms

There are two concepts within fuzzy logic which play a central role in its applications. The first is that of a linguistic variable, that is, a variable whose values are words or sentences in a natural or synthetic language. The other is that of a fuzzy if-then rule in which the antecedent and consequent are propositions containing linguistic variables. The essential function served by linguistic variables is that of granulation of variables and their dependencies. In effect, the use of linguistic variables and fuzzy if-then rules results—through granulation—in soft data compression which exploits the tolerance for imprecision and uncertainty. In this respect, fuzzy logic mimics the crucial ability of the human mind to summarize data and focus on decision-relevant information.

Lotfi A. Zadeh

Since it is not the main topic of this lecture, a detailed introduction to fuzzy systems is omitted here. We restrict ourselves to a few basic facts which are sufficient for understanding this chapter (the reader is referred to the literature for more information [20, 21, 27, 31]).

The quotation above brilliantly expresses what the core of fuzzy systems is: Linguistic if-then rules involving vague propositions (e.g. “large”, “small”, “old”, “around zero”, etc.). By this way, fuzzy systems allow reproducible automation of tasks for which no analytic model is known, but for which linguistic expert knowledge is available. Examples range from complicated chemical processes over power plant control, quality control, etc.

This sounds fine at first glance, but poses a few questions: How can such vague propositions be represented mathematically and how can we process them? The idea is simple but effective: Such vague assertions are modeled by means of so-called fuzzy sets, i.e. sets which can have intermediate degrees of membership (the unit interval $[0, 1]$ is usually taken as the domain of membership degrees). By this way, it is possible to model concepts like "tall men" which can never be represented in classical set theory without drawing ambiguous, counter-intuitive boundaries.

In order to summarize, there are three essential components of fuzzy systems:

1. The rules, i.e. a verbal description of the relationships.
2. The fuzzy sets (membership functions), i.e. the semantics of the vague expressions used in the rules.
3. An inference machine, i.e. a mathematical methodology for processing a given input through the rule base.

Since this is not a major concern in this lecture, let us assume that a reasonable inference scheme is given. There are still two important components left which have to be specified in order to make a fuzzy system work—the rules and the fuzzy sets. In many cases, they can both be found simply by using common sense (some consider fuzzy systems as nothing else than a mathematical model of common sense knowledge). In most problems, however, there is only an incomplete or inexact description of the automation task. Therefore, researchers have begun soon to investigate methods for finding or optimizing the parameters of fuzzy systems. So far, we can distinguish between the following three fundamental learning tasks:

1. The rules are given, but the fuzzy sets are unknown at all and must be found or, what happens more often, they can only be estimated und need to be optimized. A typical example would be the following: The rules for driving a car are taught in every driving school, e.g. "for starting a car, let in the clutch gently and, simultaneously, step on the gas carefully", but the beginner must learn from practical experience what "letting in the clutch gently" actually means.
2. The semantical interpretation of the rules is known sufficiently well, but the relationships between input and output, i.e. the rules, are not known. A typical example is extracting certain risk factors from patient data. In this case, it is sufficiently known which blood pressures are

high and which are low, but the factors, which really influence the risk of getting a certain disease, are unknown.

3. Nothing is known, both fuzzy sets and rules must be acquired, for instance, from sample data.

5.1 Tuning of Fuzzy Sets

Let us start with the first learning task—how to find optimal configurations of fuzzy sets. In Chapter 2, we have presented a universal algorithm for solving a very general class of optimization problems. We will now study how such a simple GA can be applied to the optimization of fuzzy sets. All we need is an appropriate coding, genetic operators (in case that the standard variants are not sufficient), and a fitness measure.

5.1.1 Coding Fuzzy Subsets of an Interval

Since this is by far the most important case in applications of fuzzy systems, let us restrict to fuzzy subsets of a given real interval $[a, b]$. Of course, we will never be able to find a coding which accommodates any possible fuzzy set. It is usual in applications to fix a certain subclass which can be represented by means of a finite set of parameters. Descriptions of such fuzzy sets can then be encoded by coding these parameters.

The first class we mention here are piecewise linear membership functions with a fixed set of grid points ($a = x_0, x_1, \ldots, x_{n-1}, x_n = b$), an equally spaced grid in the simplest case. Popular fuzzy control software tools like *fuzzy*TECH or TILShell use this technique for their internal representations of fuzzy sets. It is easy to see that the shape of the membership function is uniquely determined by the membership degrees in the grid points (see Figure 5.1 for an example). Therefore, we can simply encode such a fuzzy set by putting codings of all these membership values in one large string:

$$\boxed{c_{n,[0,1]}(\mu(x_0))}\,\boxed{c_{n,[0,1]}(\mu(x_1))}\cdots\boxed{c_{n,[0,1]}(\mu(x_n))}$$

A reasonable resolution for encoding the membership degrees is $n = 8$. Such an 8-bit coding is used in several software systems, too.

For most problems, however, simpler representations of fuzzy sets are sufficient. Many real-world applications use triangular and trapezoidal membership functions (cf. Figure 5.2). Not really surprising, a triangular fuzzy set can be encoded as

$c_{n,[a,b]}(r)$	$c_{n,[0,\delta]}(u)$	$c_{n,[0,\delta]}(v)$

,

where δ is an upper boundary for the size of the offsets, for example $\delta = (b-a)/2$. The same can be done analogously for trapezoidal fuzzy sets:

$c_{n,[a,b]}(r)$	$c_{n,[0,\delta]}(q)$	$c_{n,[0,\delta]}(u)$	$c_{n,[0,\delta]}(v)$

.

In specific control applications, where the smoothness of the control surface plays an important role, fuzzy sets of higher differentiability must be used. The most prominent representative is the bell-shaped fuzzy set whose membership function is given by a Gaussian bell function:

$$\mu(x) = e^{-\frac{(x-r)^2}{2u^2}}$$

The "bell-shaped analogue" to trapezoidal fuzzy sets are so-called radial basis functions:

$$\mu(x) = \begin{cases} e^{-\frac{(|x-r|-q)^2}{2u^2}} & \text{if } |x-r| > q \\ 1 & \text{if } |x-r| \leq q \end{cases}$$

Figure 5.3 shows a typical bell-shaped membership function. Again the coding method is straightforward, i.e.

$c_{n,[a,b]}(r)$	$c_{n,[\varepsilon,\delta]}(u)$

where ε is a lower limit for the spread u. Analogously for radial basis functions:

$c_{n,[a,b]}(r)$	$c_{n,[0,\delta]}(q)$	$c_{n,[\varepsilon,\delta]}(u)$

The final step is simple and obvious: In order to define a coding of the whole configuration, i.e. of all fuzzy sets involved, it is sufficient to put codings of all relevant fuzzy sets into one large string.

5.1.2 Coding Whole Fuzzy Partitions

There is often a-priori knowledge about the approximate configuration, for instance, something like an ordering of the fuzzy sets. A general method, which encodes all fuzzy sets belonging to one linguistic variable independently like above, yields an unnecessarily large search space. A typical situation, not only in control applications, is that we have a certain number of fuzzy sets

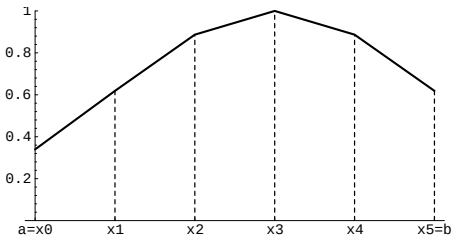


Figure 5.1: Piecewise linear membership function with fixed grid points.

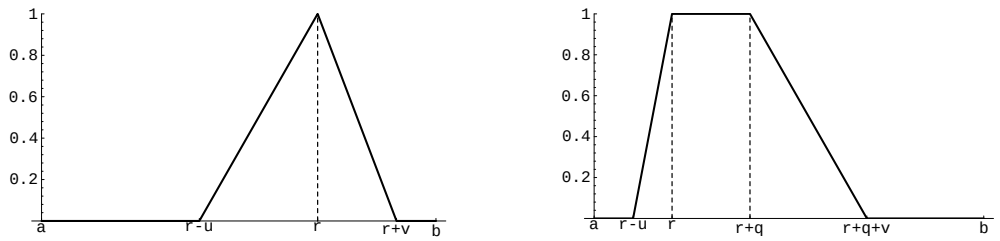


Figure 5.2: Simple fuzzy sets with piecewise linear membership functions (triangular left, trapezoidal right).

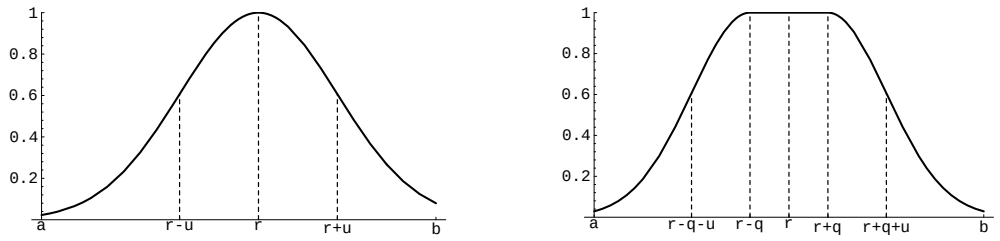


Figure 5.3: Simple fuzzy sets with smooth membership functions (bell-shaped left, radial basis right).

with labels, like "small", "medium", and "large" or "negative big", "negative medium", "negative small", "approximately zero", "positive small", "positive medium", and "positive big". In such a case, we have a natural ordering of the fuzzy sets. By including appropriate constraints, the ordering of the fuzzy sets can be preserved while reducing the number of degrees of freedom.

We will now study a simple example—an increasing sequence of trapezoidal fuzzy sets. Such a "fuzzy partition" is uniquely determined by an increasing sequence of $2N$ points, where N is the number of linguistic values we consider. The mathematical formulation is the following:

$$\mu_1(x) = \begin{cases} 1 & \text{if } x \in [x_0, x_1] \\ \frac{x_2 - x}{x_2 - x_1} & \text{if } x \in (x_1, x_2) \\ 0 & \text{otherwise} \end{cases}$$

$$\mu_i(x) = \begin{cases} \frac{x - x_{2i-3}}{x_{2i-2} - x_{2i-3}} & \text{if } x \in (x_{2i-3}, x_{2i-2}) \\ 1 & \text{if } x \in [x_{2i-2}, x_{2i-1}] \\ \frac{x_{2i} - x}{x_{2i} - x_{2i-1}} & \text{if } x \in (x_{2i}, x_{2i-1}) \\ 0 & \text{otherwise} \end{cases} \qquad \text{for } 2 \le i \le N-1$$

$$\mu_N(x) = \begin{cases} \frac{x - x_{2N-3}}{x_{2N-2} - x_{2N-3}} & \text{if } x \in (x_{2N-3}, x_{2N-2}) \\ 1 & \text{if } x \ge x_{2N-2} \\ 0 & \text{otherwise} \end{cases}$$

Figure 5.4 shows a typical example with $N = 4$. It is not wise to encode the values x_i as they are, since this requires constraints for ensuring that x_i are non-decreasing. A good alternative is to encode the offsets:

$c_{n,[0,\delta]}(x_1)$	$c_{n,[0,\delta]}(x_2 - x_1)$	$\cdots$	$c_{n,[0,\delta]}(x_{2N-2} - x_{2N-3})$

5.1.3 Standard Fitness Functions

Although it is impossible to formulate a general recipe which fits for all kinds of applications, there is one important standard situation—the case where a set of representative input-output examples is given. Assume that $F(\vec{v}, x)$ is the function which computes the output for a given input x with respect to the parameter vector $\vec{v}$. Example data is given as a list of couples (x_i, y_i) with $1 \le i \le K$ (K is the number of data samples). Obviously, the goal is to find a parameter configuration $\vec{v}$ such that the corresponding outputs $F(\vec{v}, x_i)$ match the sample outputs y_i as well as possible. This can be achieved by

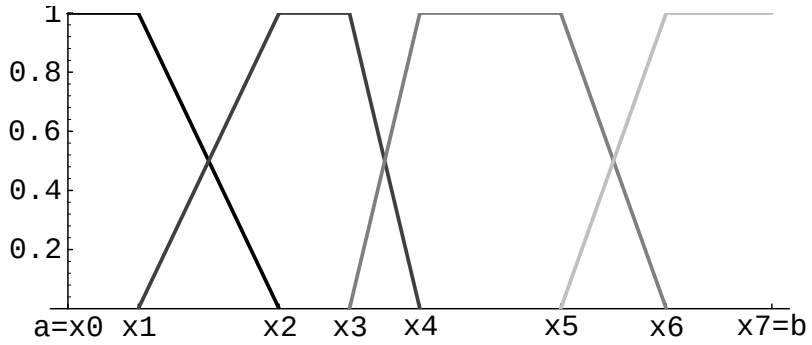


Figure 5.4: A fuzzy partition with $N = 4$ trapezoidal parts.

minimizing the error function

$$f(\vec{v}) = \sum_{i=1}^{K} d\big(F(\vec{v}, x_i), y_i\big),$$

where $d(.,.)$ is some distance measure defined on the output space. In case that the output consists of real numbers, one prominent example is the well-known sum of quadratic errors:

$$f(\vec{v}) = \sum_{i=1}^{K} \big(F(\vec{v}, x_i) - y_i\big)^2$$

5.1.4 Genetic Operators

Since we have only dealt with binary representations of fuzzy sets and partitions, all the operators from Chapter 2 are also applicable here. We should be aware, however, that the offset encoding of fuzzy partitions is highly epistatic. More specifically, if the first bit encoding x_1 is changed, the whole partition is shifted. If this results in bad convergence, the crossover operator should be modified. A suggestion can be found, for instance, in [3]. Figure 5.5 shows an example what happens if two fuzzy partitions are crossed with normal one-point crossover. Figure 5.6 shows the same for bitwise mutation.

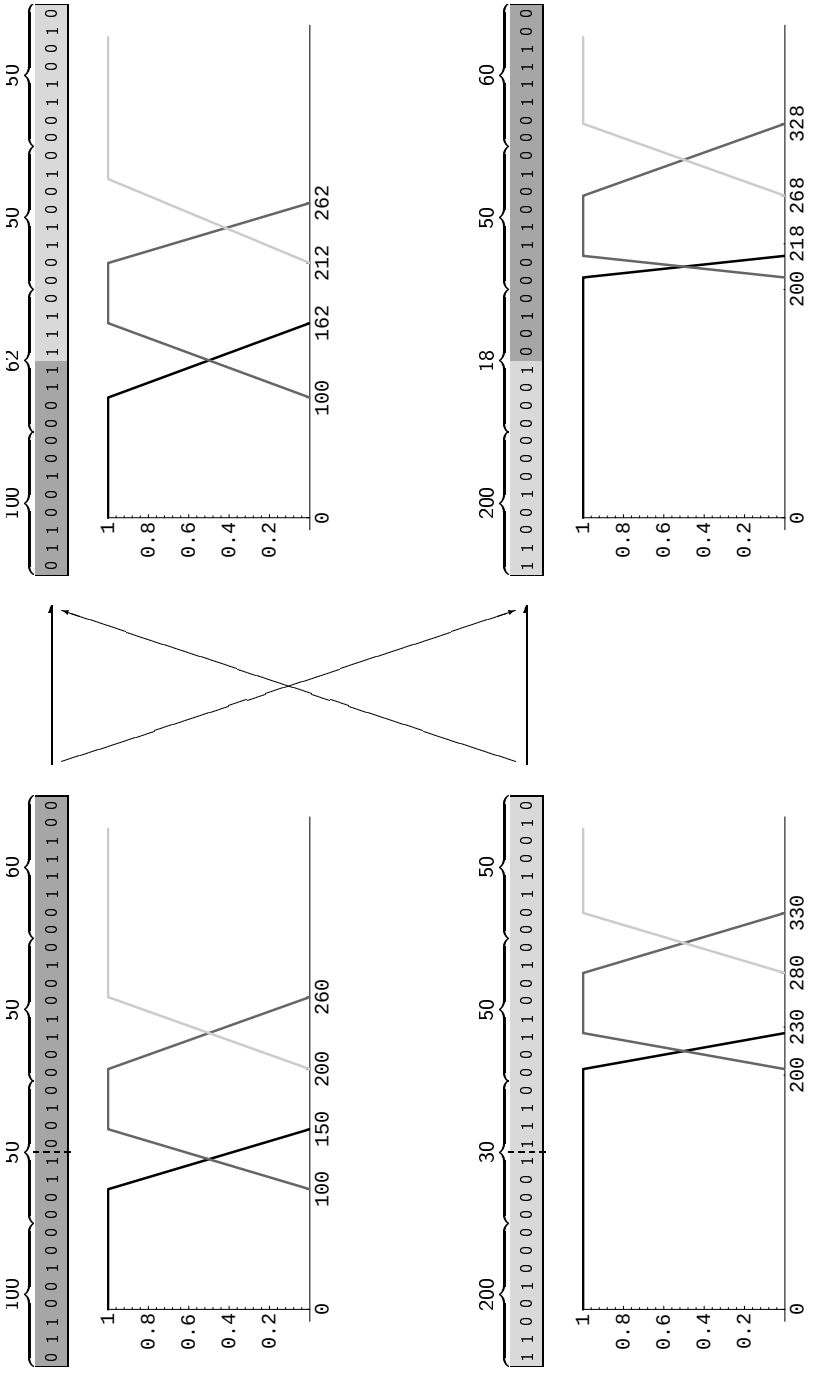


Figure 5.5: Example for one-point crossover of fuzzy partitions.

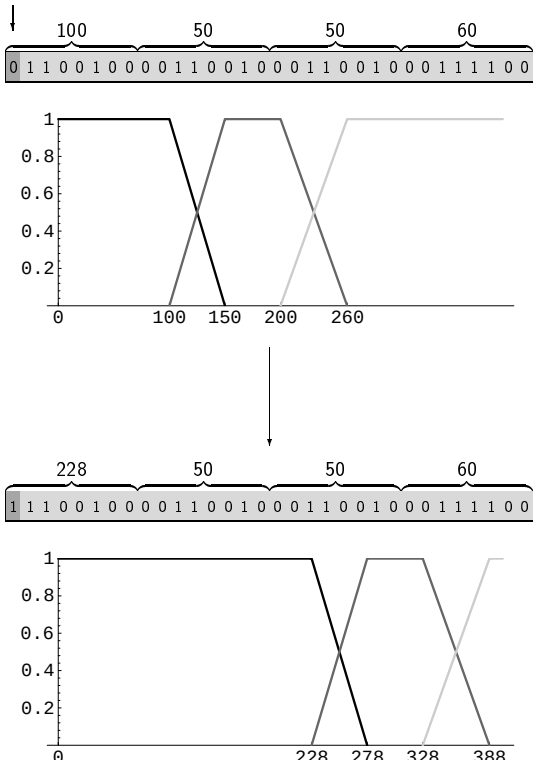


Figure 5.6: Mutating a fuzzy partition.

5.2 A Practical Example

Pixel classification is an important preprocessing task in many image processing applications. In this project, where the FLLL developed an inspection system for a silk-screen printing process, it was necessary to extract regions from the print image which had to be checked by applying different criteria:

Homogeneous area: Uniformly colored area;

Edge area: Pixels within or close to visually significant edges;

Halftone: Area which looks rather homogeneous from a certain distance, although it is actually obtained by printing small raster dots of two or more colors;

Picture: Rastered area with high, chaotic deviations, in particular small high-contrasted details.

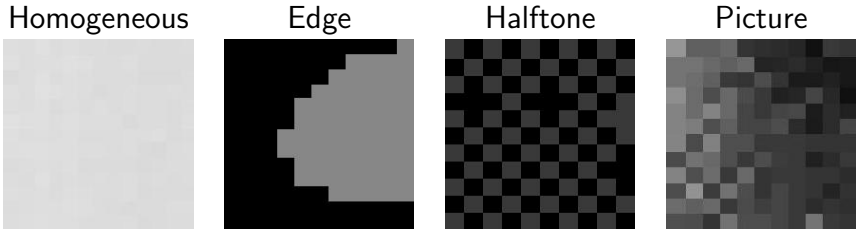
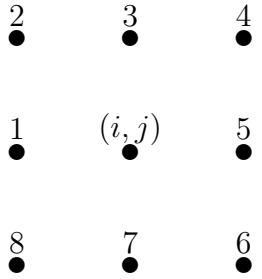


Figure 5.7: Magnifications of typical representatives of the four types of pixels.



k	$l(k)$
1	$(\quad i \quad, j-1)$
2	$(i-1, j-1)$
3	$(i-1, \quad j \quad)$
4	$(i-1, j+1)$
5	$(\quad i \quad, j+1)$
6	$(i+1, j+1)$
7	$(i+1, \quad j \quad)$
8	$(i+1, j+1)$

Figure 5.8: Clockwise enumeration of neighbor pixels.

The magnifications in Figure 5.7 show how these areas typically look like at the pixel level. Of course, transitions between two or more of these areas are possible, hence a fuzzy model is recommendable.

If we plot the gray values of the eight neighbor pixels according to a clockwise enumeration (cf. Figure 5.8), we typically get curves like those shown in Figure 5.9. Seemingly, the size of the deviations, e.g. by computing the variance, can be used to distinguish between homogeneous areas, halftones and the other two types. On the other hand, a method which judges the width and connectedness of the peaks should be used in order to separate edge areas from pictures. A simple but effective method for this purpose is the so-called discrepancy norm, for which there are already other applications in pattern recognition (cf. [22]):

$$\|\vec{x}\|_D = \max_{1 \le \alpha \le \beta \le n} \left| \sum_{i=\alpha}^{\beta} x_i \right|$$

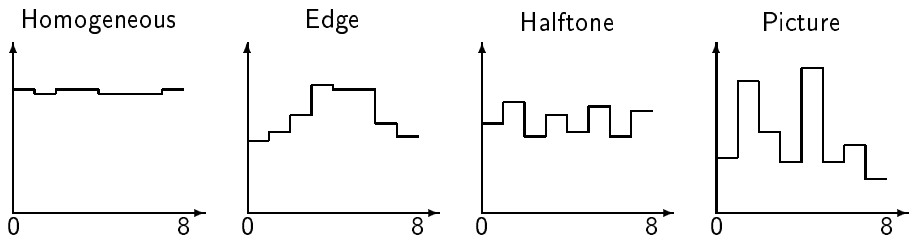


Figure 5.9: Typical gray value curves corresponding to the four types.

A more detailed analysis of the discrepancy norm, especially how it can be computed in linear time, can be found in [2].

5.2.1 The Fuzzy System

For each pixel (i, j), we consider its nearest eight neighbors enumerated as described above, which yields a vector of eight gray values. As already mentioned, the variance of the gray value vector can be taken as a measure for the size of the deviations in the neighborhood of the pixel. Let us denote this value with $v(i, j)$. On the other hand, the discrepancy norm of the vector, where we subtract each entry by the mean value of all entries, can be used as a criterion whether the pixel is within or close to a visually significant edge (let us call this value $e(i, j)$ in the following).

The fuzzy decision is then carried out for each pixel (i, j) independently: First of all, the characteristic values $v(i, j)$ and $e(i, j)$ are computed. These values are taken as the input of a small fuzzy system with two inputs and one output. Let us denote the linguistic variables on the input side with v and e. Since the position of the pixel is of no relevance for the decision in this concrete application, indices can be omitted here. The input space of the variable v is represented by three fuzzy sets which are labeled "low", "med", and "high". Analogously, the input space of the variable e is represented by two fuzzy sets, which are labeled "low" and "high". Experiments have shown that $[0, 600]$ and $[0, 200]$ are appropriate domains for v and e, respectively. For the decomposition of the input domains, simple fuzzy partitions consisting of trapezoidal fuzzy subsets were chosen. Figure 5.10 shows how these partitions basically look like.

The output space is a set of linguistic labels, namely "Ho", "Ed", "Ha", and "Pi", which are, of course, just abbreviations of the names of the four types. Let us denote the output variable itself with t. Finally, the output of

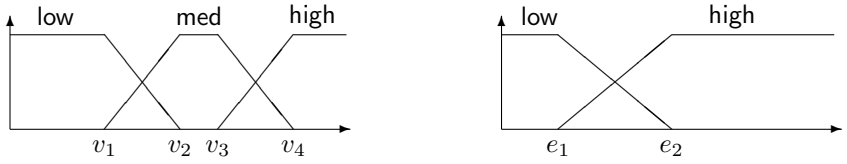


Figure 5.10: The linguistic variables v and e.

the system for each pixel (i, j) is a fuzzy subset of {"Ho", "Ed", "Ha", "Pi"}. This output set is computed by processing the values $v(i, j)$ and $e(i, j)$ through a rule base with five rules, which cover all the possible combinations:

IF	v is low			THEN	$t =$ Ho
IF	v is med	AND	e is high	THEN	$t =$ Ed
IF	v is high	AND	e is high	THEN	$t =$ Ed
IF	v is med	AND	e is low	THEN	$t =$ Ha
IF	v is high	AND	e is low	THEN	$t =$ Pi

In this application, ordinary Mamdani min/max-inference is used. Finally, the degree to which "Ho", "Ed", "Ha", or "Pi" belong to the output set can be regarded as the degree to which the particular pixel belongs to area Homogeneous, Edge, Halftone, or Picture, respectively.

5.2.2 The Optimization of the Classification System

The behavior of the fuzzy system depends on six parameters, $v_1, \ldots, v_4$, e_1, and e_2, which determine the shape of the two fuzzy partitions. In the first step, these parameters were tuned manually. Of course, we have also taken into consideration to use (semi)automatic methods for finding the optimal parameters.

Our optimization procedure consists of a painting program which offers tools, such as a pencil, a rubber, a filling algorithm, and many more. This painting tool can be used to make a reference classification of a given representative image by hand. Then an optimization algorithm can be used to find that configuration of parameters which yields the maximal degree of matching between the desired result and the output actually obtained by the classification system.

Assume that we have a set of N sample pixels for which the input values $(\tilde{v}_k, \tilde{e}_k)_{k\in\{1,\ldots,N\}}$ are computed and that we already have a reference classifi-

cation of these pixels

$$\tilde{t}(k) = (\tilde{t}_{\mathrm{Ho}}(k), \tilde{t}_{\mathrm{Ed}}(k), \tilde{t}_{\mathrm{Ha}}(k), \tilde{t}_{\mathrm{Pi}}(k)),$$

where $k \in \{1, \ldots, N\}$. Since, as soon as the values $\tilde{v}$ and $\tilde{e}$ are computed, the geometry of the image plays no role anymore, we can switch to one-dimensional indices here. One possible way to define the performance (fitness) of the fuzzy system would be

$$\frac{1}{N} \sum_{k=1}^{N} d(t(k), \tilde{t}(k)), \tag{5.1}$$

where $t(k) = (t_{\mathrm{Ho}}(k), t_{\mathrm{Ed}}(k), t_{\mathrm{Ha}}(k), t_{\mathrm{Pi}}(k))$ are the classifications actually obtained by the fuzzy system for the input pairs $(\tilde{v}_k, \tilde{e}_k)$ with respect to the parameters v_1, v_2, v_3, v_4, e_1, and e_2; $d(.,.)$ is an arbitrary (pseudo-)metric on $[0,1]^4$. The problem of this brute force approach is that the output of the fuzzy system has to be evaluated for each pair (v_k, e_k), even if many of these values are similar or even equal. In order to keep the amount of computation low, we "simplified" the procedure by a "clustering process" as follows:

We choose a partition $(P_1, \ldots, P_K)$ of the input space, where $(n_1, \ldots, n_K)$ are the numbers of sample points $\{p_1^i, \ldots, p_{n_i}^i\}$ each part contains. Then the desired classification of a certain part (cluster) can be defined as

$$\tilde{t}_X(P_i) = \frac{1}{n_i} \sum_{j=1}^{n_i} \tilde{t}_X(p_j^i),$$

where $X \in \{\mathrm{Ho}, \mathrm{Ed}, \mathrm{Ha}, \mathrm{Pi}\}$.

If ϕ is a function which maps each cluster to a representative value (e.g., its center of gravity), we can define the fitness (objective) function as

$$\frac{100}{N} \sum_{i=1}^{K} n_i \cdot \left(1 - \frac{1}{2} \cdot \sum_{X \in \{\mathrm{Ho,Ed,Ha,Pi}\}} \left(\tilde{t}_X(P_i) - t_X(\phi(P_i)) \right)^2 \right), \tag{5.2}$$

If the number of parts is chosen moderately (e.g. a rectangular 64×32 net which yields $K = 2048$) the evaluation of the fitness function takes considerably less time than a direct application of formula (5.1).

Note that in (5.2) the fitness is already transformed such that it can be regarded as a degree of matching between the desired and the actually obtained classification measured in percent. This value has to be maximized.

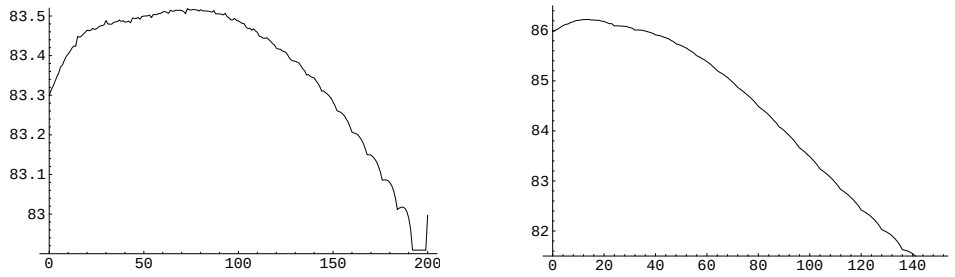


Figure 5.11: Cross sections of a function of type (5.2).

In fact, fitness functions of this type are, in almost all cases, continuous but not differentiable and have a lot of local maxima. Figure 5.11 shows cross sections of such functions. Therefore, it is more reasonable rather to use probabilistic optimization algorithms than to apply continuous optimization methods, which make excessive use of derivatives. This, first of all, requires a (binary) coding of the parameters. We decided to use a coding which maps the parameters v_1, v_2, v_3, v_4, e_1, and e_2 to a string of six 8-bit integers $s_1, \ldots, s_6$ which range from 0 to 255. The following table shows how the encoding and decoding is done:

$$\begin{array}{lcllcl} s_1 & = & v_1 & v_1 & = & s_1 \\ s_2 & = & v_2 - v_1 & v_2 & = & s_1 + s_2 \\ s_3 & = & v_3 - v_2 & v_3 & = & s_1 + s_2 + s_3 \\ s_4 & = & v_4 - v_3 & v_4 & = & s_1 + s_2 + s_3 + s_4 \\ s_5 & = & e_1 & e_1 & = & s_5 \\ s_6 & = & e_2 - e_1 & e_2 & = & s_5 + s_6 \end{array}$$

We first tried a simple GA with standard roulette wheel selection, one-point crossover with uniform selection of the crossing point, and bitwise mutation. The length of the strings was, as shown above, 48.

In order to compare the performance of the GAs with other well-known probabilistic optimization methods, we additionally considered the following methods:

Hill climbing: always moves to the best-fitted neighbor of the current string until a local maximum is reached; the initial string is generated randomly.

Simulated annealing: powerful, often-used probabilistic method which is based on the imitation of the solidification of a crystal under slowly decreasing temperature

	$f_{\max}$	$f_{\min}$	$\bar{f}$	σ_f	It
Hill Climbing	94.3659	89.6629	93.5536	1.106	862
Simulated Annealing	94.3648	89.6625	93.5639	1.390	1510
Improved Simulated Annealing	94.3773	93.7056	94.2697	0.229	21968
GA	94.3760	93.5927	94.2485	0.218	9910
Hybrid GA (elite)	94.3760	93.6299	94.2775	0.207	7460
Hybrid GA (random)	94.3776	94.3362	94.3693	0.009	18631

Figure 5.12: A comparison of results obtained by several different optimization methods.

Each one of these methods requires only a few binary operations in each step. Most of the time is consumed by the evaluation of the fitness function. So, it is near at hand to take the number of evaluations as a measure for the speed of the algorithms.

Results

All these algorithms are probabilistic methods; therefore, their results are not well-determined, they can differ randomly within certain boundaries. In order to get more information about their average behavior, we tried out each one of them 20 times for one certain problem. For the given problem, we found out that the maximal degree of matching between the reference classification and the classification actually obtained by the fuzzy system was 94.3776%. In the table in Figure 5.12, $f_{\max}$ is the fitness of the best and $f_{\min}$ is the fitness of the worst solution; $\bar{f}$ denotes the average fitness of the 20 solutions, σ_f denotes the standard deviation of the fitness values of the 20 solutions, and # stands for the average number of evaluations of the fitness function which was necessary to reach the solution.

The hill climbing method with random selection of the initial string converged rather quickly. Unfortunately, it was always trapped in a local maximum, but never reached the global solution (at least in these 20 trials).

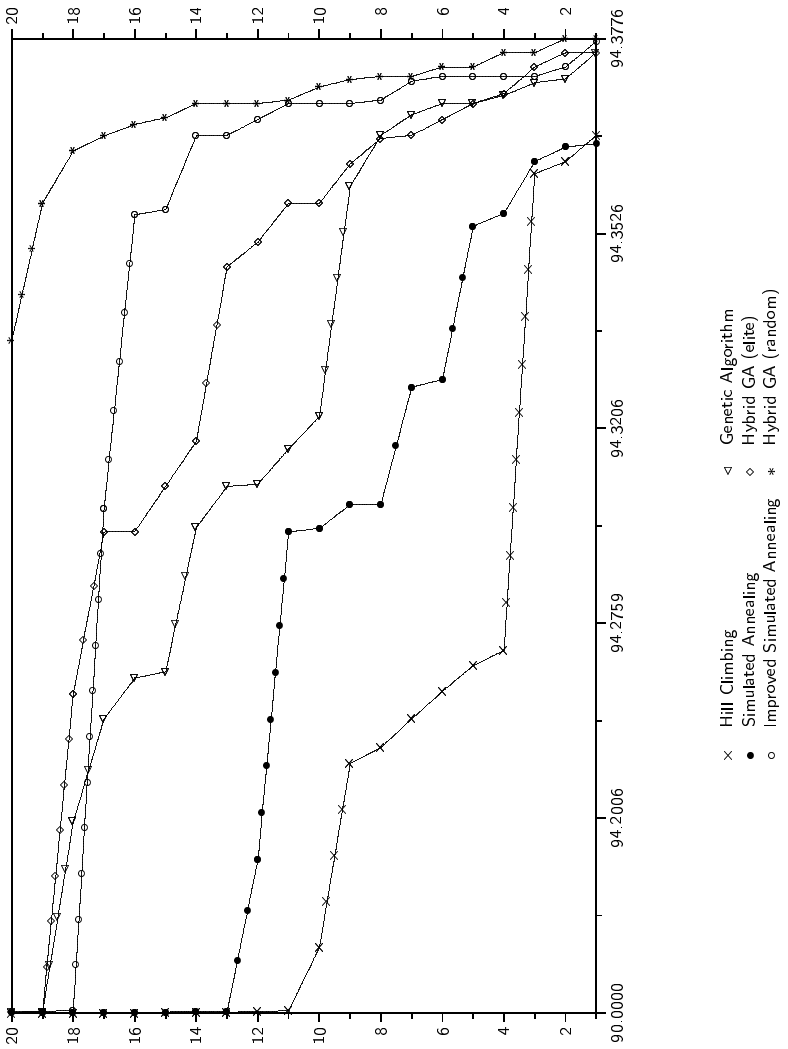


Figure 5.13: A graphical representation of the results.

The simulated annealing algorithm showed similar behavior at the very beginning. After tuning the parameters involved, the performance improved remarkably.

The raw genetic algorithm was implemented with a population size of 20; the crossover probability was set to 0.85, the mutation probability was 0.005 for each byte. It behaved pretty well from the beginning, but it seemed inferior to the improved simulated annealing.

Next, we tried a hybrid GA, where we kept the genetic operations and parameters of the raw GA, but every 50th generation the best-fitted individual was taken as initial string for a hill climbing method. Although the performance increased slightly, the hybrid method still seemed to be worse than the improved simulated annealing algorithm. The reason that the effects of this modification were not so dramatic might be that the probability is rather high that the best individual is already a local maximum. So we modified the procedure again. This time a *randomly chosen individual* of every 25th generation was used as initial solution of the hill climbing method. The result exceeded the expectations by far. The algorithm was, in all cases, nearer to the global solution than the improved simulated annealing (compare with table in Figure 5.12), but, surprisingly, sufficed with less invocations of the fitness function. The graph in Figure 5.13 shows the results graphically. Each line in this graph corresponds to one algorithm. The curve shows, for a given fitness value x, how many of the 20 different solutions had a fitness higher or equal to x. It can be seen easily from this graph that the hybrid GA with random selection led to the best results. Note that the x-axis is not a linear scale in this figure. It was transformed in order to make small differences visible.

5.2.3 Concluding Remarks

In this example, we have investigated the suitability of genetic algorithms for finding the optimal parameters of a fuzzy system, especially if the analytical properties of the objective function are bad. Moreover, hybridization has been discovered as an enormous potential for improvements of genetic algorithms.

5.3 Finding Rule Bases with GAs

Now let us briefly turn to the second learning problem from Page 58. If we find a method for encoding a rule base into a string of a fixed length, all

the genetic methods we have dealt with so far, are applicable with only little modifications. Of course, we have to assume in this case that the numbers of linguistic values of all linguistic variables are finite.

The simplest case is that of coding a complete rule base which covers all the possible cases. Such a rule base is represented as a list for one input variable, as a matrix for two variables, and as a tensor in the case of more than two input variables. For example, consider a rule base of the following form (the generalization to more than two input variable is straightforward):

$$\mathsf{IF} \quad x_1 \text{ is } A_i \text{ AND } x_2 \text{ is } B_j \quad \mathsf{THEN} \quad y \text{ is } \tilde{C}_{i,j}$$

A_i and B_j are verbal values of the variables x_1 and x_2, respectively. All the values A_i are pairwise different, analogously for the values B_j; i ranges from 1 to N_1, the total number of linguistic values of variable x_1; j ranges from 1 to N_2, the total number of linguistic values of variable x_2. The values $\tilde{C}_{i,j}$ are arbitrary elements of the set of pairwise different linguistic values $\{C_1, \ldots, C_{N_y}\}$ associated with the output variable y. Obviously, such a rule base is uniquely represented by a matrix, a so-called decision table:

	B_1	$\cdots$	B_{N_2}
A_1	$\tilde{C}_{1,1}$	$\cdots$	$\tilde{C}_{1,N_2}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
A_{N_1}	$\tilde{C}_{N_1,1}$	$\cdots$	$\tilde{C}_{N_1,N_2}$

Of course, the representation is still unique if we replace the values $\tilde{C}_{i,j}$ by their unique indices within the set $\{C_1, \ldots, C_{N_y}\}$ and we have found a proper coding scheme for table-based rule bases.

5.1 Example. Consider a fuzzy system with two inputs (x_1 and x_2) and one output y. The domains of all three variables are divided into four fuzzy sets labeled "small", "medium", "large", and "very large" (for short, abbreviated "S", "M", "L", and "V"). We will now study how the following decision table can be encoded into a string:

	S	M	L	V
S	S	S	S	M
M	S	S	M	L
L	S	M	L	V
V	M	L	V	V

For example, the third entry "M" in the second row reads as follows:

$$\mathsf{IF} \quad x_1 \text{ is medium } \mathsf{AND} \; x_2 \text{ is large} \quad \mathsf{THEN} \quad y \text{ is medium}$$

If we assign indices ranging from 0–3 to the four linguistic values associated with the output variable y, we can write the decision table as an integer string with length 16:

$$(0, 0, 0, 1, 0, 0, 1, 2, 0, 1, 2, 3, 1, 2, 3, 3)$$

Replacing the integer values with two-bit binary strings, we obtain a 32-bit binary string which uniquely describes the above decision table:

$$(\texttt{00000001000001100001101101101111})$$

For the method above, genetic algorithms of type 2.5 are perfectly suitable. Of course, the fitness functions, which we have introduced in 5.1.3, can also be used without any modifications.

It is easy to see that the approach above works consequent-oriented, meaning that the premises are fixed—only the consequent values must be acquired. Such an idea can only be applied to optimization of complete rule bases which are, in more complex applications, not so easy to handle. Moreover, complete rule bases are often an overkill and require a lot of storage resources. In many applications, especially in control, it is enough to have an incomplete rule base consisting of a certain number of rules which cover the input space sufficiently well.

The acquisition of incomplete rule bases is a task, which is not so easy to solve with representations of fixed length. We will come to this point a little later.

Chapter 6

Genetic Programming

> *How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what is needed to be done, without being told explicitly how to do it?*
>
> John R. Koza

Mathematicians and computer scientists, in their everyday practice, do nothing else than searching for programs which solve given problems properly. They usually try to design such programs based on their knowledge of the problem, its underlying principles, mathematical models, their intuition, etc. Koza's questions seem somehow provocative and utopian. His answers, however, are remarkable and worth to be discussed here in more detail. The basic idea is simple but appealing—to apply genetic algorithms to the problem of automatic program induction. All we need in order to do so are appropriate modifications of all genetic operations we have discussed so far. This includes random initialization, crossover, and mutation. For selection and sampling, we do not necessarily need anything new, because these methods are independent of the underlying data representation.

Of course, this sounds great. The question arises, however, whether this kind of *Genetic Programming (GP)* can work at all. Koza, in his remarkable monograph [19], starts with a rather vague hypothesis.

6.1 The Genetic Programming Paradigm. *Provided that we are given a solvable problem, a definition of an appropriate programming language, and a sufficiently large set of representative test examples (correct input-output pairs), a genetic algorithm is able to find a program which (approximately) solves the problem.*

This seems to be a matter of believe. Nobody has been able to prove this hypothesis so far and it is doubtable whether this will ever be possible. Instead of giving a proof, Koza has elaborated a large set of well-chosen examples which underline his hypothesis empirically. The problems he has solved successfully with GP include the following:

- Process control (bang bang control of inverted pendulum)
- Logistics (simple robot control, stacking problems)
- Automatic programming (pseudo-random number generators, prime number program, ANN design)
- Game strategies (Poker, Tic Tac Toe)
- Inverse kinematics
- Classification
- Symbolic computation:
  - Sequence induction (Fibonacci sequence, etc.)
  - Symbolic regression
  - Solving equations (power series-based solutions of functional, differential, and integral equations)
  - Symbolic differentiation and integration
  - Automatic discovery of trigonometric identities

This chapter is devoted to a brief introduction to genetic programming. We will restrict ourselves to the basic methodological issues and omit to elaborate examples in detail. For a nice example, the reader is referred to a [12].

6.1 Data Representation

Without any doubt, programs can be considered as strings. There are, however, two important limitations which make it impossible to use the representations and operations from our simple GA:

1. It is mostly inappropriate to assume a fixed length of programs.

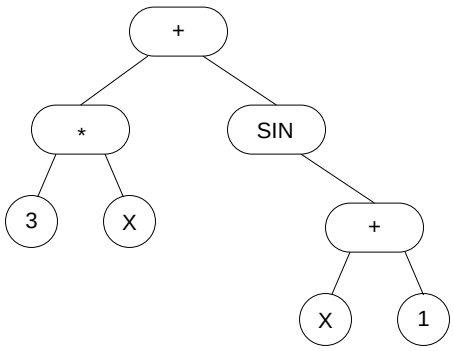


Figure 6.1: The tree representation of `(+ (* 3 X) (SIN (+ X 1)))`.

2. The probability to obtain syntactically correct programs when applying our simple initialization, crossover, and mutation procedures is hopelessly low.

It is, therefore, indispensable to modify the data representation and the operations such that syntactical correctness is easier to guarantee. The common approach to represent programs in genetic programming is to consider programs as trees. By doing so, initialization can be done recursively, crossover can be done by exchanging subtrees, and random replacement of subtrees can serve as mutation operation.

Since their only construct are nested lists, programs in LISP-like languages already have a kind of tree-like structure. Figure 6.1 shows an example how the function $3x + \sin(x + 1)$ can be implemented in a LISP-like language and how such a LISP-like function can be split up into a tree. Obviously, the tree representation directly corresponds to the nested lists the program consists of; atomic expressions, like variables and constants, are leaves while functions correspond to non-leave nodes.

There is one important disadvantage of the LISP approach—it is difficult to introduce type checking. In case of a purely numeric function like in the above example, there is no problem at all. However, it can be desirable to process numeric data, strings, and logical expressions simultaneously. This is difficult to handle if we use a tree representation like in Figure 6.1.

A very general approach, which overcomes this problem allowing maximum flexibility, has been proposed by A. Geyer-Schulz. He suggested to represent programs by their syntactical derivation trees with respect to a recursive definition of underlying language in Backus-Naur Form (BNF) [10].

This works for any context-free language. It is far beyond the scope of this lecture to go into much detail about formal languages. We will explain the basics with the help of a simple example. Consider the following language which is suitable for implementing binary logical expressions:

$$
\begin{array}{lcl}
\text{S} & := & \langle \text{exp} \rangle \ ; \\
\langle \text{exp} \rangle & := & \langle \text{var} \rangle \mid \text{“(”} \ \langle \text{neg} \rangle \ \langle \text{exp} \rangle \ \text{“)”} \mid \text{“(”} \ \langle \text{exp} \rangle \ \langle \text{bin} \rangle \ \langle \text{exp} \rangle \ \text{“)”} \ ; \\
\langle \text{var} \rangle & := & \text{“}x\text{”} \mid \text{“}y\text{”}; \\
\langle \text{neg} \rangle & := & \text{“NOT”} \ ; \\
\langle \text{bin} \rangle & := & \text{“AND”} \mid \text{“OR”} \ ;
\end{array}
$$

The BNF description consists of so-called syntactical rules. Symbols in angular brackets $\langle\rangle$ are called non-terminal symbols, i.e. symbols which have to be expanded. Symbols between quotation marks are called terminal symbols, i.e. they cannot be expanded any further. The first rule S:=$\langle$exp$\rangle$; defines the starting symbol. A BNF rule of the general shape

$$\langle \text{non-terminal} \rangle \quad := \quad \langle \text{deriv}_1 \rangle \mid \langle \text{deriv}_2 \rangle \mid \cdots \mid \langle \text{deriv}_n \rangle \ ;$$

defines how a non-terminal symbol may be expanded, where the different variants are separated by vertical bars.

In order to get a feeling how to work with the BNF grammar description, we will now show step by step how the expression (NOT (x OR y)) can be derivated from the above language. For simplicity, we omit quotation marks for the terminal symbols:

1. We have to begin with the start symbol: $\langle$exp$\rangle$
2. We replace $\langle$exp$\rangle$ with the second possible derivation:

   $$\langle \text{exp} \rangle \longrightarrow (\langle \text{neg} \rangle \ \langle \text{exp} \rangle)$$

3. The symbol $\langle$neg$\rangle$ may only be expanded with the terminal symbol NOT:

   $$(\langle \text{neg} \rangle \ \langle \text{exp} \rangle) \longrightarrow (\text{NOT} \ \langle \text{exp} \rangle)$$

4. Next, we replace $\langle$exp$\rangle$ with the third possible derivation:

   $$(\text{NOT} \ \langle \text{exp} \rangle) \longrightarrow (\text{NOT} \ (\langle \text{exp} \rangle \ \langle \text{bin} \rangle \ \langle \text{exp} \rangle))$$

5. We expand the second possible derivation for $\langle$bin$\rangle$:

   $$(\text{NOT} \ (\langle \text{exp} \rangle \ \langle \text{bin} \rangle \ \langle \text{exp} \rangle)) \longrightarrow (\text{NOT} \ (\langle \text{exp} \rangle \ \text{OR} \ \langle \text{exp} \rangle))$$

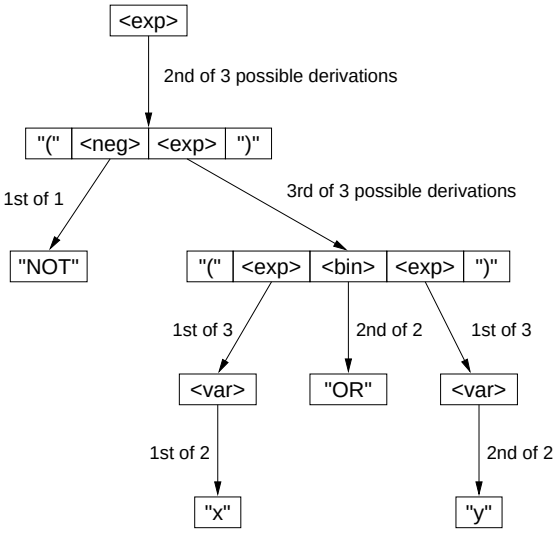


Figure 6.2: The derivation tree of (NOT (x OR y)).

6. The first occurrence of ⟨exp⟩ is expanded with the first derivation:

$$(\text{NOT } (\langle\text{exp}\rangle \text{ OR } \langle\text{exp}\rangle)) \longrightarrow (\text{NOT } (\langle\text{var}\rangle \text{ OR } \langle\text{exp}\rangle))$$

7. The second occurrence of ⟨exp⟩ is expanded with the first derivation, too:

$$(\text{NOT } (\langle\text{var}\rangle \text{ OR } \langle\text{exp}\rangle)) \longrightarrow (\text{NOT } (\langle\text{var}\rangle \text{ OR } \langle\text{var}\rangle))$$

8. Now we replace the first ⟨var⟩ with the corresponding first alternative:

$$(\text{NOT } (\langle\text{var}\rangle \text{ OR } \langle\text{var}\rangle)) \longrightarrow (\text{NOT } (x \text{ OR } \langle\text{var}\rangle))$$

9. Finally, the last non-terminal symbol is expanded with the second alternative:

$$(\text{NOT } (x \text{ OR } \langle\text{var}\rangle)) \longrightarrow (\text{NOT } (x \text{ OR } y))$$

Such a recursive derivation has an inherent tree structure. For the above example, this derivation tree has been visualized in Figure 6.2.

6.1.1 The Choice of the Programming Language

The syntax of modern programming languages can be specified in BNF. Hence, our data model would be applicable to all of them. The question

is whether this is useful. Koza's hypothesis includes that the programming language has to be chosen such that the given problem is solvable. This does not necessarily imply that we have to choose the language such that virtually any solvable problem can be solved. It is obvious that the size of the search space grows with the complexity of the language. We know that the size of the search space influences the performance of a genetic algorithm—the larger the slower.

It is, therefore, recommendable to restrict the language to necessary constructs and to avoid superfluous constructs. Assume, for example, that we want to do symbolic regression, but we are only interested in polynomials with integer coefficients. For such an application, it would be an overkill to introduce rational constants or to include exponential functions in the language. A good choice could be the following:

$$
\begin{array}{lcl}
\mathrm{S} & := & \langle \mathrm{func} \rangle \ ; \\
\langle \mathrm{func} \rangle & := & \langle \mathrm{var} \rangle \mid \langle \mathrm{const} \rangle \mid \text{“(” } \langle \mathrm{func} \rangle \ \langle \mathrm{bin} \rangle \ \langle \mathrm{func} \rangle \text{ “)”} \ ; \\
\langle \mathrm{var} \rangle & := & \text{“}x\text{”} \ ; \\
\langle \mathrm{const} \rangle & := & \langle \mathrm{int} \rangle \mid \langle \mathrm{const} \rangle \ \langle \mathrm{int} \rangle \ ; \\
\langle \mathrm{int} \rangle & := & \text{“0”} \mid \cdots \mid \text{“9”} \ ; \\
\langle \mathrm{bin} \rangle & := & \text{“+”} \mid \text{“-”} \mid \text{“}*\text{”} \ ;
\end{array}
$$

For representing rational functions with integer coefficients, it is sufficient to add the division symbol “/” to the possible derivations of the binary operator $\langle \mathrm{bin} \rangle$.

Another example: The following language could be appropriate for discovering trigonometric identities:

$$
\begin{array}{lcl}
\mathrm{S} & := & \langle \mathrm{func} \rangle \ ; \\
\langle \mathrm{func} \rangle & := & \langle \mathrm{var} \rangle \mid \langle \mathrm{const} \rangle \mid \langle \mathrm{trig} \rangle \text{ “(” } \langle \mathrm{func} \rangle \text{ “)”} \mid \\
 & & \text{“(” } \langle \mathrm{func} \rangle \ \langle \mathrm{bin} \rangle \ \langle \mathrm{func} \rangle \text{ “)”} \ ; \\
\langle \mathrm{var} \rangle & := & \text{“}x\text{”} \ ; \\
\langle \mathrm{const} \rangle & := & \text{“0”} \mid \text{“1”} \mid \text{“}\pi\text{”} \ ; \\
\langle \mathrm{trig} \rangle & := & \text{“sin”} \mid \text{“cos”} \ ; \\
\langle \mathrm{bin} \rangle & := & \text{“+”} \mid \text{“-”} \mid \text{“}*\text{”} \ ;
\end{array}
$$

6.2 Manipulating Programs

We have a generic coding of programs—the derivation trees. It remains to define the three operators random initialization, crossover, and mutation for derivations trees.

6.2.1 Random Initialization

Until now, we did not pay any attention to the creation of the initial population. We assumed implicitly that the individuals of the first generation can be generated randomly with respect to a certain probability distribution (mostly uniform). Undoubtedly, this is an absolutely trivial task if we deal with binary strings of fixed length. The random generation of derivation trees, however, is a much more subtle task.

There are basically two different variants how to generate random programs with respect to a given BNF grammar:

1. Beginning from the starting symbol, it is possible to expand non-terminal symbols recursively, where we have to choose randomly if we have more than one alternative derivations. This approach is simple and fast, but has some disadvantages: Firstly, it is almost impossible to realize a uniform distribution. Secondly, one has to implement some constraints with respect to the depth of the derivation trees in order to avoid excessive growth of the programs. Depending on the complexity of the underlying grammar, this can be a tedious task.

2. Geyer-Schulz [11] has suggested to prepare a list of all possible derivation trees up to a certain depth[1] and to select from this list randomly applying a uniform distribution. Obviously, in this approach, the problems in terms of depth and the resulting probability distribution are elegantly solved, but these advantages go along with considerably long computation times.

6.2.2 Crossing Programs

It is trivial to see that primitive string-based crossover of programs almost never yield syntactically correct programs. Instead, we should use the perfect syntax information a derivation tree provides. Already in the LISP times of genetic programming, some time before the BNF-based representation was known, crossover was usually implemented as the exchange of randomly selected subtrees. In case that the subtrees (subexpressions) may have different types of return values (e.g. logical and numerical), it is not guaranteed that crossover preserves syntactical correctness.

[1] The depth is defined as the number of all non-terminal symbols in the derivation tree. There is no one-to-one correspondence to the height of the tree.

The derivation tree-based representation overcomes this problem in a very elegant way. If we only exchange subtrees which start from the same non-terminal symbol, crossover can never violate syntactical correctness. In this sense, the derivation tree model provides implicit typechecking.

In order to demonstrate in more detail how this crossover operation works, let us reconsider the example of binary logical expressions (grammar defined on page 80). As parents, we take the following expressions:

(NOT (x OR y))
((NOT x) OR (x AND y))

Figure 6.3 shows graphically how the two children

(NOT (x OR (x AND y)))
((NOT x) OR y)

are obtained.

6.2.3 Mutating Programs

We have always considered mutation as the random deformation of a small part of a chromosome. It is, therefore, not surprising that the most common mutation in genetic programming is the random replacement of a randomly selected subtree. This can be accomplished with the method presented in 6.2.1. The only modification is that we do not necessarily start from the start symbol, but from the non-terminal symbol at the root of the subtree we consider. Figure 6.4 shows an example where, in the logical expression (NOT (x OR y)), the variable y is replaced by (NOT y).

6.2.4 The Fitness Function

There is no common recipe for specifying an appropriate fitness function which strongly depends on the given problem. It is, however, worth to emphasize that it is necessary to provide enough information to guide the GA to the solution. More specifically, it is not sufficient to define a fitness function which assigns 0 to a program which does not solve the problem and 1 to a program which solves the problem—such a fitness function would correspond to a needle-in-haystack problem. In this sense, a proper fitness measure should be a gradual concept for judging the correctness of programs.

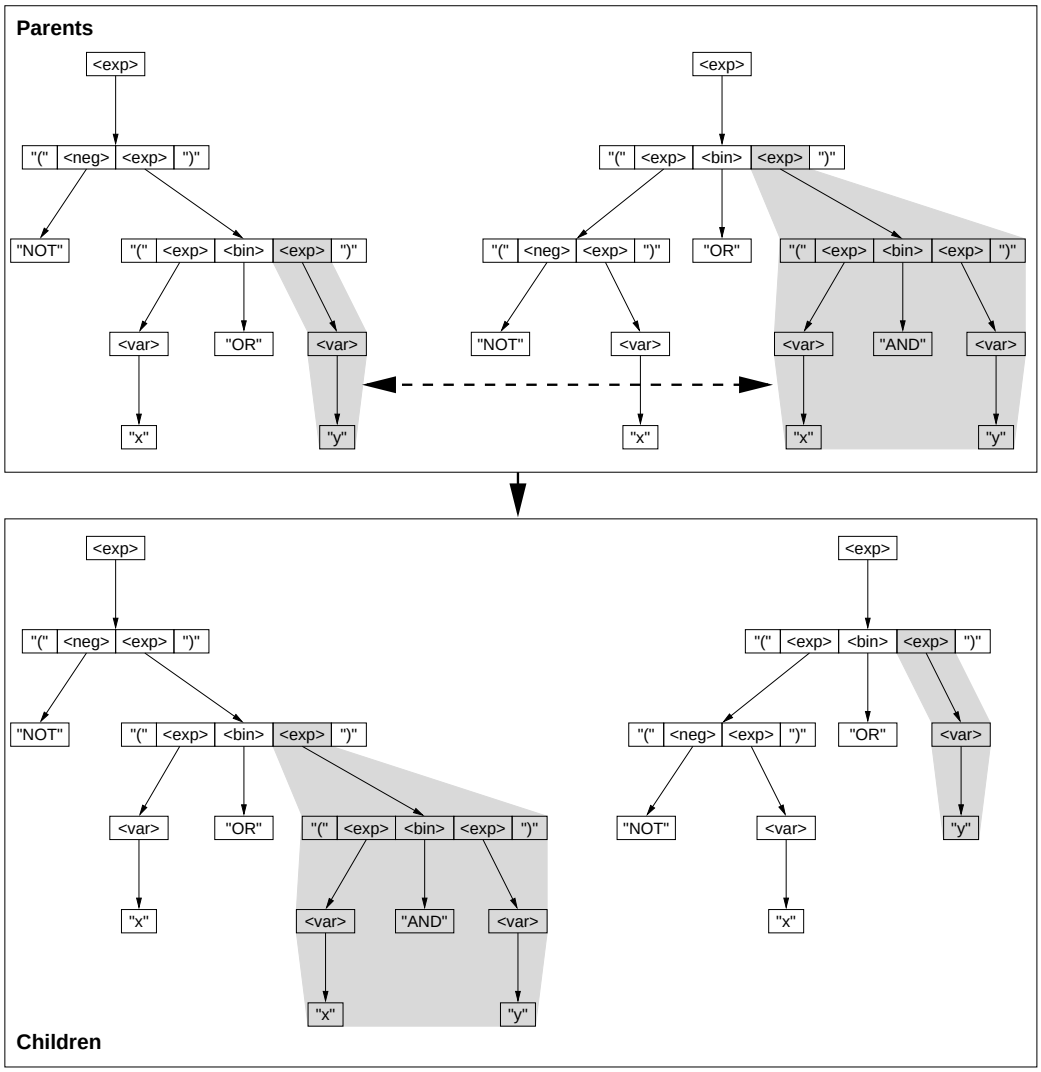


Figure 6.3: An example for crossing two binary logical expressions.

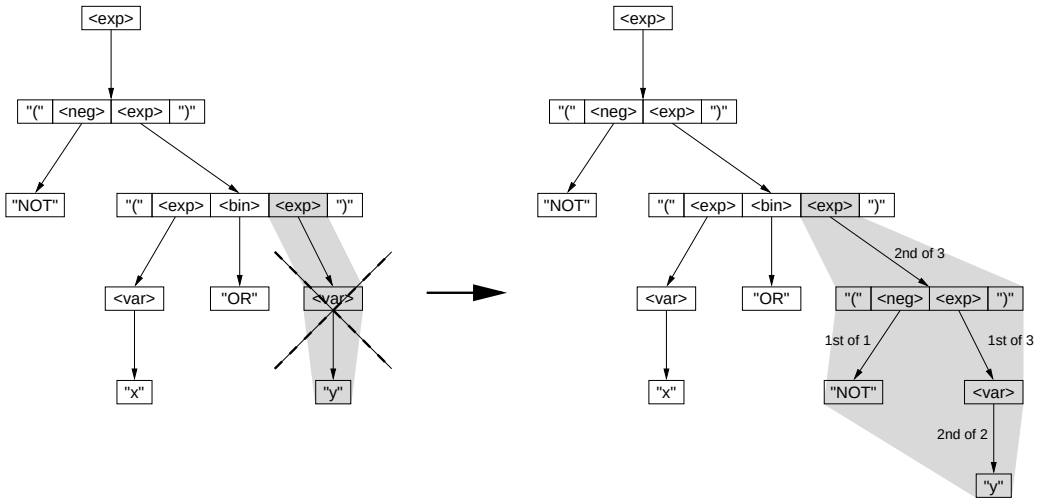


Figure 6.4: An example for mutating a derivation tree.

In many applications, the fitness function is based on a comparison of desired and actually obtained output (compare with 5.1.3, p. 62). Koza, for instance, uses the simple sum of quadratic errors for symbolic regression and the discovery of trigonometric identities:

$$f(F) = \sum_{i=1}^{N} (y_i - F(x_i))^2$$

In this definition, F is the mathematical function which corresponds to the program under evaluation. The list $(x_i, y_i)_{1 \le i \le N}$ consists of reference pairs—a desired output y_i is assigned to each input x_i. Clearly, the samples have to be chosen such that the considered input space is covered sufficiently well.

Numeric error-based fitness functions usually imply minimization problems. Some other applications may imply maximization tasks. There are basically two well-known transformations which allow to standardize fitness functions such that always minimization or maximization tasks are obtained.

6.2 Definition. Consider an arbitrary "raw" fitness function f. Assuming that the number of individuals in the population is not fixed (m_t at time t), the *standardized fitness* is computed as

$$f_{\mathbf{S}}(b_{i,t}) = f(b_{i,t}) - \max_{j=1}^{m_t} f(b_{j,t})$$

in case that f is to maximize and as

$$f_{\mathbf{S}}(b_{i,t}) = f(b_{i,t}) - \min_{j=1}^{m_t} f(b_{j,t})$$

if f has to be minimized. One possible variant is to consider the best individual of the last k generations instead of only considering the actual generation.

Obviously, standardized fitness transforms any optimization problem into a minimization task. Roulette wheel selection relies on the fact that the objective is maximization of the fitness function. Koza has suggested a simple transformation such that, in any case, a maximization problem is obtained.

6.3 Definition. With the assumptions of Definition 6.2, the *adjusted fitness* is computed as

$$f_{\mathbf{A}}(b_{i,t}) = \max_{j=1}^{m_t} f_{\mathbf{S}}(b_{j,t}) - f_{\mathbf{S}}(b_{j,t}).$$

Another variant of adjusted fitness is defined as

$$f'_{\mathbf{A}}(b_{i,t}) = \frac{1}{1 + f_{\mathbf{S}}(b_{j,t})}.$$

6.3 Fuzzy Genetic Programming

It was already mentioned that the acquisition of fuzzy rule bases from example data is an important problem (Points 2. and 3. according to the classification on pp. 58ff.). We have seen in 5.3, however, that the possibilities for finding rule bases automatically are strongly limited. A revolutionary idea was introduced by A. Geyer-Schulz: To specify a rule language in BNF and to apply genetic programming. For obvious reasons, we refer to this synergistic combination as *fuzzy genetic programming*. Fuzzy genetic programming elegantly overcomes limitations of all other approaches:

1. If a rule base is represented as a list of rules of arbitrary length, we are not restricted to complete decision tables.

2. We are not restricted to atomic expressions—it is easily possible to introduce additional connectives and linguistic modifiers, such as “very”, “at least”, “roughly”, etc.

The following example shows how a fuzzy rule language can be specified in Backus-Naur form. Obviously, this fuzzy system has two inputs x_1 and x_2.

The output variable is y. The domain of x_1 is divided into three fuzzy sets "neg", "approx. zero", and "pos". The domain of x_2 is divided into three fuzzy sets which are labeled "small", "medium", and "large". For the output variable y, five atomic fuzzy sets called "nb", "nm', "z", "pm', and "pb" are specified.

S	:=	⟨rb⟩ ;
⟨rb⟩	:=	⟨rule⟩ | ⟨rule⟩ "," ⟨rb⟩ ;
⟨rule⟩	:=	"IF" ⟨premise⟩ "THEN" ⟨conclusion⟩ ;
⟨premise⟩	:=	⟨atomic⟩ | "(" ⟨neg⟩ ⟨premise⟩ ")" |
		"(" ⟨premise⟩ ⟨bin⟩ ⟨premise⟩ ")" ;
⟨neg⟩	:=	"NOT" ;
⟨bin⟩	:=	"AND" | "OR" ;
⟨atomic⟩	:=	"x_1" "is" ⟨val_1⟩ | "x_2" "is" ⟨val_2⟩ ;
⟨conclusion⟩	:=	"y" "is" ⟨val_3⟩ ;
⟨val_1⟩	:=	⟨adjective_1⟩ | ⟨adverb⟩ ⟨adjective_1⟩ ;
⟨val_2⟩	:=	⟨adjective_2⟩ | ⟨adverb⟩ ⟨adjective_2⟩ ;
⟨val_3⟩	:=	⟨adjective_3⟩ | ⟨adverb⟩ ⟨adjective_3⟩ ;
⟨adverb⟩	:=	"at least" | "at most" | "roughly" ;
⟨adjective_1⟩	:=	"neg" | "approx. zero" | "pos" ;
⟨adjective_2⟩	:=	"small" | "medium" | "large" ;
⟨adjective_3⟩	:=	"nb" | "nm" | "z" | "pm' | "pb";

A very nice example on an application of genetic programming and fuzzy genetic programming to a stock management problem can be found in [12].

6.4 A Checklist for Applying Genetic Programming

We conclude this chapter with a checklist of things which are necessary to apply genetic programming to a given problem:

1. An appropriate fitness function which provides enough information to guide the GA to the solution (mostly based on examples).
2. A syntactical description of a programming language which contains as much elements as necessary for solving the problem.
3. An interpreter for the programming language.

Chapter 7

Classifier Systems

Ever since Socrates taught geometry to the slave boy in Plato's Meno, the nature of learning has been an active topic of investigation. For centuries, it was province of philosophers, who analytically studied inductive and deductive inference. A hundred years ago, psychology began to use experimental methods to investigate learning in humans and other organisms. Still more recently, the computer has provided a research tool, engendering the field of machine learning.

J. H. Holland, K. J. Holyoak, R. E. Nisbett, and P. R. Thagard

7.1 Introduction

Almost all GA-based approaches to machine learning problems have in common, firstly, that they operate on populations of models/descriptions/rule bases and, secondly, that the individuals are judged globally, i.e. there is one fitness value for each model indicating how good it describes the actual interrelations in the data. The main advantage of such approaches is simplicity: There are only two things one has to find—a data representation which is suitable for a genetic algorithm and a fitness function. In particular, if the representation is rule-based, no complicated examination which rules are responsible for success or failure has to be done.

The convergence of such methods, however, can be weak, because single obstructive parts can deteriorate the fitness of a whole description which could contain useful, well-performing rules. Moreover, genetic algorithms are

often perfect in finding suboptimal global solutions quickly; local refinement, on the other hand, can take a long time.

Another aspect is that it is sometimes difficult to define a global quality measure which provides enough information to guide a genetic algorithm to a proper solution. Consider, for instance, the game of chess: A global quality measure could be the percentage of successes in a large number of games or, using more specific knowledge, the number of moves it took to be successful in the case of success and the number of moves it had been possible to postpone the winning of the opponent in the case of failure. It is easy to see that such information provides only a scarce foundation for learning chess, even if more detailed information, such as the number of captured pieces, is involved. On the contrary, it is easier to learn the principles of chess, when the direct effect of the application of a certain rule can be observed immediately or at least after a few steps. The problem, not only in the case of chess, is that early actions can also contribute much to a final success.

In the following, we will deal with a paradigm which can provide solutions to some of the above problems—the so-called classifier systems of the Michigan type. Roughly speaking, they try to find rules for solving a task in an online process according to responses from the environment by employing a genetic algorithm. Figure 7.1 shows the basic architecture of such a system. The main components are:

1. A production system containing a rule base which processes incoming messages from the environment and sends output messages to the environment.

2. An apportionment of credit system which receives payoff from the environment and determines which rules had been responsible for that feedback; this component assigns strength values to the single rules in the rule base. These values represent the performance and usefulness of the rules.

3. A genetic algorithm which recombines well-performing rules to new, hopefully better ones, where the strengths of the rules are used as objective function values.

Obviously, the learning task is divided into two subtasks—the judgment of already existing and the discovery of new rules.

There are a few basic characteristics of such systems which are worth to be mentioned:

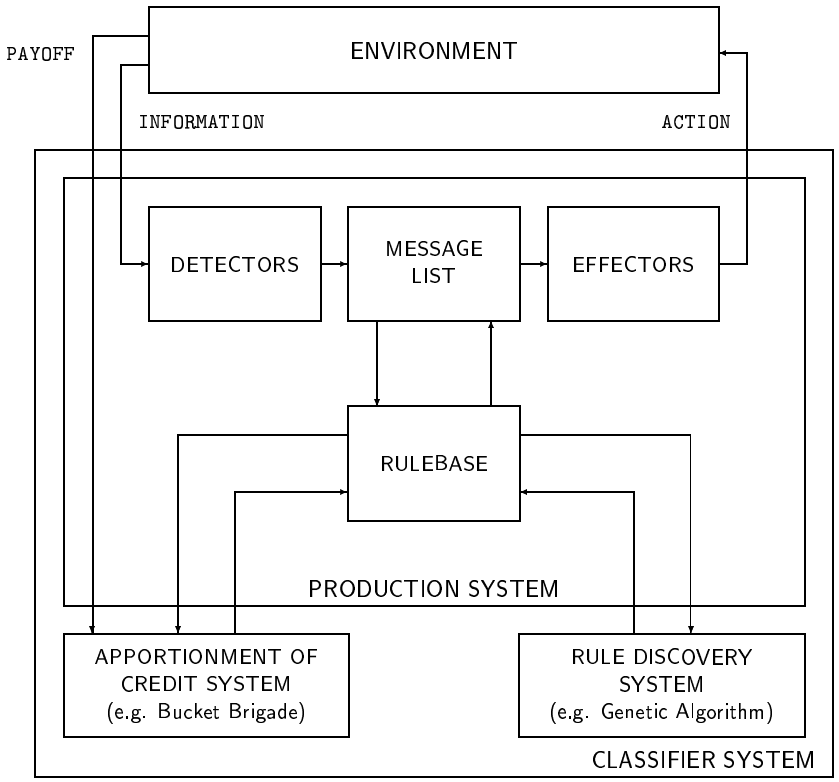


Figure 7.1: Basic architecture of a classifier system of the Michigan type.

1. The basis of the search does not consist of examples which describe the task as in many classical ML methods, but of feedback from the environment (payoff) which judges the correctness/usefulness of the last decisions/actions.

2. There is no strict distinction between learning and working like, for instance, in many ANN approaches.

3. Since learning is done in an online process, Michigan classifier systems can adapt to changing circumstances in the environment.

7.2 Holland Classifier Systems

For illustrating in more detail how such systems basically work, let us first consider a common variant—the so-called Holland classifier system.

A Holland classifier system is a classifier system of the Michigan type which processes binary messages of a fixed length through a rule base whose rules are adapted according to response of the environment [11, 16, 17].

7.2.1 The Production System

First of all, the communication of the production system with the environment is done via an arbitrarily long list of messages. The detectors translate responses from the environment into binary messages and place them on the message list which is then scanned and changed by the rule base. Finally, the effectors translate output messages into actions on the environment, such as forces or movements.

Messages are binary strings of the same length k. More formally, a message belongs to $\{0,1\}^k$. The rule base consists of a fixed number m of rules (classifiers) which consist of a fixed number r of conditions and an action, where both conditions and actions are strings of length k over the alphabet $\{0,1,*\}$. The asterisk plays the role of a wildcard, a "don't care" symbol.

A condition is matched, if and only if there is a message in the list which matches the condition in all non-wildcard positions. Moreover, conditions, except the first one, may be negated by adding a "–" prefix. Such a prefixed condition is satisfied, if and only if there is no message in the list which matches the string associated with the condition. Finally, a rule fires, if and only if all the conditions are satisfied, i.e. the conditions are connected with AND. Such "firing" rules compete to put their action messages on the message list. This competition will soon be discussed in connection with the apportionment of credit problem.

In the action parts, the wildcard symbols have a different meaning. They take the role of "pass through" element. The output message of a firing rule, whose action part contains a wildcard, is composed from the non-wildcard positions of the action and the message which satisfies the first condition of the classifier. This is actually the reason why negations of the first conditions are not allowed. More formally, the outgoing message $\tilde{m}$ is defined as

$$\tilde{m}[i] = \begin{cases} a[i] & \text{if } a[i] \neq * \\ m[i] & \text{if } a[i] = * \end{cases} \qquad i = 1, \ldots, k,$$

where a is the action part of the classifier and m is the message which matches the first condition. Formally, a classifier is a string of the form

$$\text{Cond}_1, [\text{“–”}]\text{Cond}_2, \ldots, [\text{“–”}]\text{Cond}_r/\text{Action},$$

where the brackets should express the optionality of the "–" prefixes.

Depending on the concrete needs of the task to be solved, it may be desirable to allow messages to be preserved for the next step. More specifically, if a message is not interpreted and removed by the effector interface, it can make another classifier fire in the next step. In practical applications, this is usually accomplished by reserving a few bits of the messages for identifying the origin of the messages (a kind of variable index called *tag*). Tagging offers new opportunities to transfer information about the current step into the next step simply by placing tagged messages on the list which are not interpreted by the output interface. These messages, which obviously contain information about the previous step, can support the decisions in the next step. Hence, appropriate use of tags permits rules to be coupled to act sequentially. In some sense, such messages are the memory of the system.

A single execution cycle of the production system consists of the following steps:

1. Messages from the environment are appended to the message list.
2. All the conditions of all classifiers are checked against the message list to obtain the set of firing rules.
3. The message list is erased.
4. The firing classifiers participate in a competition to place their messages on the list (see 7.2.2).
5. The winning classifiers place their actions on the list.
6. The messages directed to the effectors are executed.

This procedure is repeated iteratively.

How 6. is done, if these messages are deleted or not, and so on, depends on the concrete implementation. It is, on the one hand, possible to choose a representation such that each output message can be interpreted by the effectors. On the other hand, it is possible to direct messages explicitly to the effectors with a special tag. If no messages are directed to the effectors, the system is in a thinking phase.

A classifier R_1 is called consumer of a classifier R_2 if and only if there is a message m' which fulfills at least one of R_1's conditions and has been placed on the list by R_2. Conversely, R_2 is called a supplier of R_1.

7.2.2 The Bucket Brigade Algorithm

As already mentioned, in each time step t, we assign a strength value $u_{i,t}$ to each classifier R_i. This strength value represents the correctness and importance of a classifier. On the one hand, the strength value influences the chance of a classifier to place its action on the output list. On the other hand, the strength values are used by the rule discovery system which we will soon discuss.

In Holland classifier systems, the adaptation of the strength values depending on the feedback (payoff) from the environment is done by the so-called *bucket brigade algorithm*. It can be regarded as a simulated economic system in which various agents, here the classifiers, participate in an auction, where the chance to buy the right to post the action depends on the strength of the agents.

The bid of classifier R_i at time t is defined as

$$B_{i,t} = c_L \cdot u_{i,t} \cdot s_i,$$

where $c_L \in [0, 1]$ is a learning parameter, similar to learning rates in artificial neural nets, and s_i is the specifity, the number of non-wildcard symbols in the condition part of the classifier. If c_L is chosen small, the system adapts slowly. If it is chosen too high, the strengths tend to oscillate chaotically.

Then the rules have to compete for the right for placing their output messages on the list. In the simplest case, this can be done by a random experiment like the selection in a genetic algorithm. For each bidding classifier it is decided randomly if it wins or not, where the probability that it wins is proportional to its bid:

$$\mathsf{P}[R_i \text{ wins}] = \frac{B_{i,t}}{\sum\limits_{j \in \mathrm{Sat}_t} B_{j,t}}$$

In this equation, Sat_t is the set of indices all classifiers which are satisfied at time t. Classifiers which get the right to post their output messages are called winning classifiers.

Obviously, in this approach, more than one winning classifier is allowed. Of course, other selection schemes are reasonable, for instance, the highest bidding agent wins alone. This can be necessary to avoid that two winning classifiers direct conflicting actions to the effectors.

Now let us discuss how payoff from the environment is distributed and how the strengths are adapted. For this purpose, let us denote the set of

classifiers, which have supplied a winning agent R_i in step t, with $S_{i,t}$. Then the new strength of a winning agent is reduced by its bid and increased by its portion of the payoff P_t received from the environment:

$$u_{i,t+1} = u_{i,t} + \frac{P_t}{w_t} - B_{i,t},$$

where w_t is the number of winning agents in the actual time step. A winning agent pays its bid to its suppliers which share the bid among each other, equally in the simplest case:

$$u_{l,t+1} = u_{l,t} + \frac{B_{i,t}}{|S_{i,t}|} \qquad \text{for all } R_l \in S_{i,t}$$

If a winning agent has also been active in the previous step and supplies another winning agent, the value above is additionally increased by one portion of the bid the consumer offers. In the case that two winning agents have supplied each other mutually, the portions of the bids are exchanged in the above manner. The strengths of all other classifiers R_n, which are neither winning agents nor suppliers of winning agents, are reduced by a certain factor (they pay a tax):

$$u_{n,t+1} = u_{n,t} \cdot (1 - T),$$

T is a small values from $[0, 1]$. The intention of taxation is to punish classifiers which never contribute anything to the output of the system. With this concept, redundant classifiers, which never become active, can be filtered out.

The idea behind credit assignment in general and bucket brigade in particular is to increase the strengths of rules which have set the stage for later successful actions. The problem of determining such classifiers, which were responsible for conditions under which it was later on possible to receive a high payoff, can be very difficult. Consider, for instance, the game of chess again, in which very early moves can be significant for a late success or failure. In fact, the bucket brigade algorithm can solve this problem, although strength is only transferred to the suppliers which were active in the previous step. Each time the same sequence is activated, however, a little bit of the payoff is transferred one step back in the sequence. It is easy to see that repeated successful execution of a sequence increases the strengths of all involved classifiers.

Figure 7.2 shows a simple example how the bucket brigade algorithm works. For simplicity, we consider a sequence of five classifiers which always

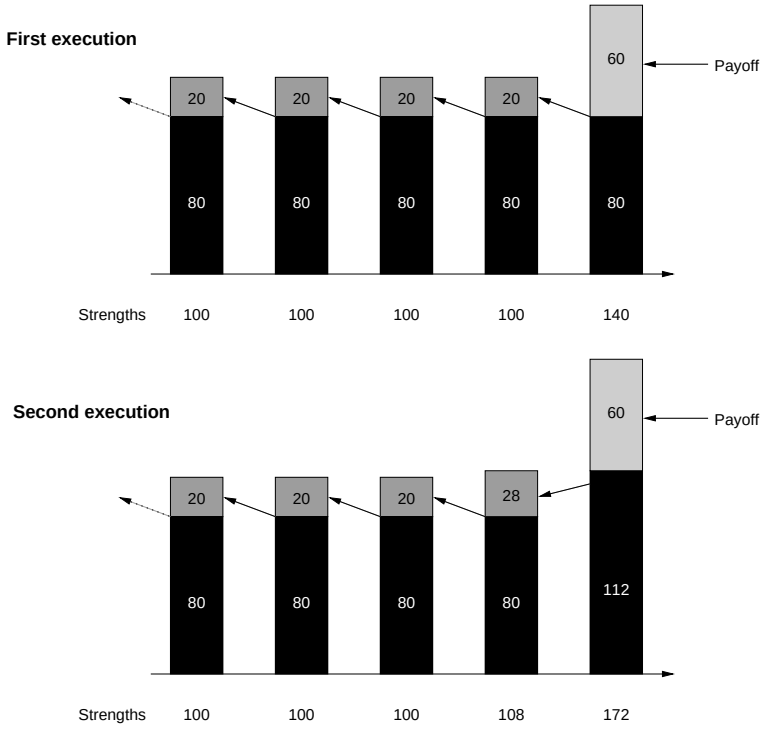


Figure 7.2: The bucket brigade principle.

bid 20 percent of their strength. Only after the fifth step, after the activation of the fifth classifier, a payoff of 60 is received. The further development of the strengths in this example is shown in the table in Figure 7.3. It is easy to see from this example that the reinforcement of the strengths is slow at the beginning, but it accelerates later. Exactly this property contributes much to the robustness of classifier systems—they tend to be cautious at the beginning, trying not to rush conclusions, but, after a certain number of similar situations, the system adopts the rules more and more. Figure 7.4 shows a graphical visualization of this fact interpreting the table shown in Figure 7.3 as a two-dimensional surface.

It might be clear, that a Holland classifier system only works if successful sequences of classifier activations are observed sufficiently often. Otherwise the bucket brigade algorithm does not have a chance to reinforce the strengths of the successful sequence properly.

Strength after the					
3rd	100.00	100.00	101.60	120.80	172.00
4th	100.00	100.32	105.44	136.16	197.60
5th	100.06	101.34	111.58	152.54	234.46
6th	100.32	103.39	119.78	168.93	247.57
⋮					
10th	106.56	124.17	164.44	224.84	278.52
⋮					
25th	215.86	253.20	280.36	294.52	299.24
⋮					
execution of the sequence					

Figure 7.3: An example for repeated propagation of payoffs.

7.2.3 Rule Generation

While the apportionment of credit system just judges the rules, the purpose of the rule discovery system is to eliminate low-fitted rules and to replace them by hopefully better ones. The fitness of a rule is simply its strength. Since the classifiers of a Holland classifier system themselves are strings, the application of a genetic algorithm to the problem of rule induction is straightforward, though many variants are reasonable. Almost all variants have in common that the GA is not invoked in each time step, but only every n-th step, where n has to be set such that enough information about the performance of new classifiers can be obtained in the meantime.

A. Geyer-Schulz [11], for instance, suggests the following procedure, where the strength of new classifiers is initialized with the average strength of the current rule base:

1. Select a subpopulation of a certain size at random.
2. Compute a new set of rules by applying the genetic operations selection, crossing over, and mutation to this subpopulation.
3. Merge the new subpopulation with the rule base omitting duplicates and replacing the worst classifiers.

This process of acquiring new rules has an interesting side effect. It is more than just the exchange of parts of conditions and actions. Since we

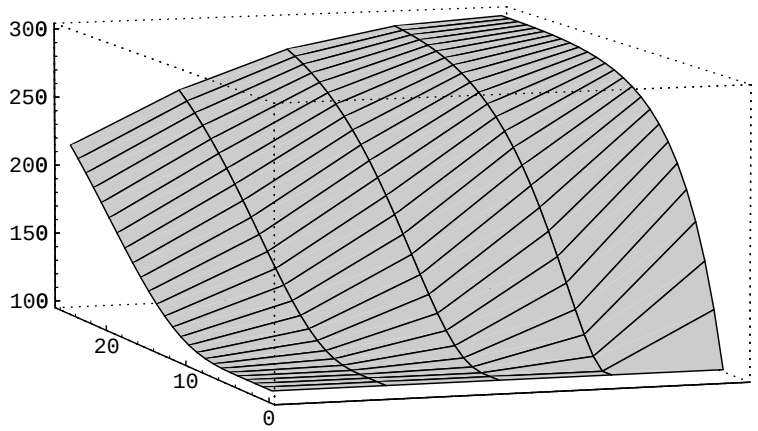


Figure 7.4: A graphical representation of the table shown in Figure 7.3.

have not stated restrictions for manipulating tags, the genetic algorithm can recombine parts of already existing tags to invent new tags. In the following, tags spawn related tags establishing new couplings. These new tags survive if they contribute to useful interactions. In this sense, the GA additionally creates experience-based internal structures autonomously.

7.3 Fuzzy Classifier Systems of the Michigan Type

While classifier systems of the Michigan type have been introduced by J. H. Holland already in the Seventies, their fuzzification awaited discovery many years. The first fuzzy classifier system of the Michigan type was introduced by M. Valenzuela-Rendón [28, 29]. It is, more or less, a straightforward fuzzification of a Holland classifier system. An alternative approach has been developed by A. Bonarini [5, 6], who introduced a different scheme of competition between classifiers. These two approaches have in common that they operate only on the fuzzy rules—the shape of the membership functions is fixed. A third method, which was introduced by P. Bonelli and A. Parodi [24], tries to optimize even the membership functions and the output weights in accordance to payoff from the environment.

7.3.1 Directly Fuzzifying Holland Classifier Systems

The Production System

We consider a fuzzy controller with real-valued inputs and outputs. The system has, unlike ordinary fuzzy controllers, three different types of variables—input, output, and internal variables. As we will see later, internal variables are for the purpose of storing information about the near past. They correspond to the internally tagged messages in Holland classifier systems. For the sake of generality and simplicity, all domains of all variables are intervals transformed to the unit interval $[0, 1]$. For each variable, the same number of membership functions n is assumed. These membership functions are fixed at the beginning. They are not changed throughout the learning process. M. Valenzuela-Rendón took bell-shaped function dividing the input domain equally.

A message is a binary string of length $l + n$, where n is the number of membership functions defined above and l is the length of the prefix (tag), which identifies the variable to which the message belongs. A perfect choice for l would be $\lceil \log_2 K \rceil$, where K is the total number of variables we want to consider. To each message, an *activity level*, which represents a truth value, is assigned. Consider, for instance, the following message ($l = 3$, $n = 5$):

$$\underbrace{010}_{=2} : 00010 \rightarrow 0.6$$

Its meaning is "Input value no. 2 belongs to fuzzy set no. 4 with a degree of 0.6". On the message list, only so-called minimal messages are used, i.e. messages with only one 1 in the right part which corresponds to the indices of the fuzzy sets.

Classifiers again consist of a fixed number r of conditions and an action part. Note that, in this approach, no wildcards and no "–" prefixes are used. Both condition and action part are also binary strings of length $l + n$, where the tag and the identifiers of the fuzzy sets are separated by a colon. The degree to which such a condition is matched is a truth value between 0 and 1. The degree of matching of a condition is computed as the maximal activity of messages on the list, which have the same tag and whose 1s are a subset of those of the condition. Figure 7.5 shows a simple example how this matching is done. The degree of satisfaction of the whole classifier is then computed as the minimum of matching degrees of the conditions. This value is then used as the activity level which is assigned to the output message (corresponds to Mamdani inference).

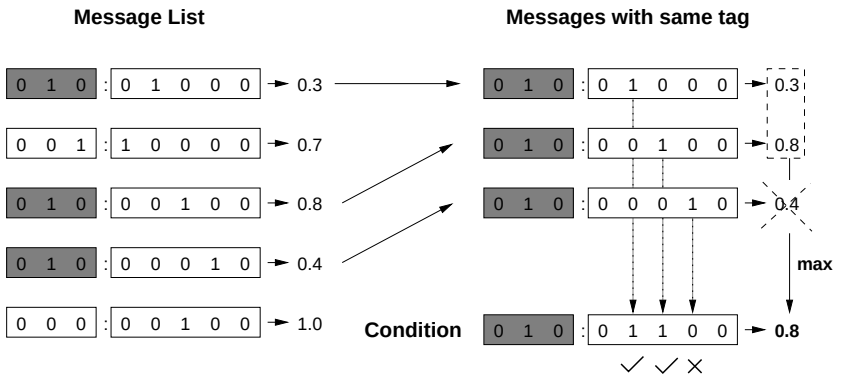


Figure 7.5: Matching a fuzzy condition.

The whole rule base consists of a fixed number m of such classifiers. Similarly to Holland classifier systems, one execution step of the production system is carried out as follows:

1. The detectors receive crisp input values from the environment and translate them into minimal messages which are then added to the message list.
2. The degrees of matching are computed for all classifiers.
3. The message list is erased.
4. The output messages of some matched classifiers (see below) are placed on the message list.
5. The output messages are translated into minimal messages. For instance, the message $010 : 00110 \rightarrow 0.9$ is split up into the two messages $010 : 00010 \rightarrow 0.9$ and $010 : 00100 \rightarrow 0.9$.
6. The effectors discard the output messages (referring to output variables) from the list and translate them into instructions to the environment.

Step 6 is done by a slightly modified Mamdani inference: The sum (instead of the maximum or another t-conorm) of activity levels of messages, which refer to the same fuzzy set of a variable, is computed. The membership functions are then scaled with these sums. Finally, the center of gravity of the "union" (i.e. maximum) of these functions, which belong to one variable, is computed (Sum-Prod inference).

Credit Assignment

Since fuzzy systems have been designed to model transitions, a probabilistic auction process as discussed in connection with Holland classifier systems, where only a small number of rules is allowed to fire, is not desirable. Of course, we assign strength values to the classifiers again.

If we are dealing with a one-stage system, in which payoff for a certain action is received immediately, where no long-term strategies must be evolved, we can suffice with allowing all matched rules to post their outputs and sharing the payoff among the rules, which were active in the last step, according to their activity levels in this step. For example, if S_t is the set of classifiers, which have been active at time t, and P_t is the payoff received after the t-th step, the modification of the strengths of firing rules can be defined as

$$u_{i,t+1} = u_{i,t} + P_t \cdot \frac{a_{i,t}}{\sum\limits_{R_j \in S_t} a_{j,t}} \qquad \forall R_i \in S_t, \tag{7.1}$$

where $a_{i,t}$ denotes the activity level of the classifier R_i at time t. It is again possible to reduce the strength of inactive classifiers by a certain tax.

In the case, that the problem is so complex that long-term strategies are indispensable, a fuzzification of the bucket brigade mechanism must be found. While Valenzuela-Rendón only provides a few vague ideas, we state one possible variant, where the firing rules pay a certain value to their suppliers which depends on the activity level. The strength of a classifier, which has recently been active in time step t, is then increased by a portion of the payoff as defined in (7.1), but it is additionally decreased by a value

$$B_{i,t} = c_L \cdot u_{i,t} \cdot a_{i,t},$$

where $c_L \in [0, 1]$ is the learning parameter. Of course, it is again possible to incorporate terms which depend on the specifity of the classifier.

This "fuzzy bid" is then shared among the suppliers of such a firing classifier according to the amount they have contributed to the matching of the consumer. If we consider an arbitrary but fixed classifier R_j, which has been active in step t and if we denote the set of classifiers supplying R_j, which have been active in step $t-1$, with $S_{j,t}$, the change of the strengths of these suppliers can be defined as

$$u_{k,t+1} = u_{k,t} + B_{j,t} \cdot \frac{a_{k,t-1}}{\sum\limits_{R_l \in S_{j,t}} a_{l,t-1}} \qquad \text{for all } R_k \in S_{j,t}.$$

Rule Discovery

The adaptation of a genetic algorithm to the problem of manipulating classifiers in our system is again straightforward. We only have to take special care that tags in conditional parts must not refer to output variables and that tags in the action parts of the classifiers must not refer to input variables of the system.

Analogously to our previous considerations, if we admit a certain number of internal variables, the system can build up internal chains automatically. By means of internal variables, a classifier system of this type does not only learn stupid input-output actions, it also tries to discover causal interrelations.

7.3.2 Bonarini's ELF Method

In [5], A. Bonarini presents his ELF (evolutionary learning of fuzzy rules) method and applies it to the problem of guiding an autonomous robot. The key issue of ELF is to find a small rule base which only contains important rules. While he takes over many of M. Valenzuela-Rendón's ideas, his way of modifying the rule base differs strongly from Valenzuela-Rendón's straightforward fuzzification of Holland classifier systems.

Bonarini calls the modification scheme "cover-detector algorithm". The number of rules can be varied in each time step depending on the number of rules which match the actual situation. This is done by two mutually exclusive operations:

1. If the rules, which match the actual situation, are too many, the worst of them is deleted.

2. If there are too few rules matching the current inputs, a new rule, the antecedents of which cover the current state, is added to the rule base with randomly chosen consequent value.

The genetic operations are only applied to the consequent values of the rules. Since the antecedents are generated on demand in the different time steps, no taxation is necessary.

Obviously, such a simple modification scheme can only be applied to so-called one-stage problems, where the effect of each rule can be observed in the next time step. For applications where this is not the case, e.g. backing up a truck, Bonarini introduced an additional concept to his ELF algorithm—the

notion of an *episode*, which is a given number of subsequent control actions, after which the reached state is evaluated (for details, see [6]).

7.3.3 Online Modification of the Whole Knowledge Base

While the last two methods only manipulate rules and work with fixed membership functions, there is at least one variant of fuzzy classifier systems where also the shapes of the membership functions are involved in the learning process. This variant was introduced by A. Parodi and P. Bonelli in [24]. Let us restrict to the very basic idea here: A rule is not encoded with indices pointing to membership functions of a given shape. Instead, each rule contains codings of fuzzy sets like the ones we discussed in 5.1.

Bibliography

[1] Bagley, J. D. *The Behavior of Adaptive Systems Which Employ Genetic and Correlative Algorithms.* PhD thesis, University of Michigan, Ann Arbor, 1967.

[2] Bauer, P., Bodenhofer, U., and Klement, E. P. A fuzzy algorithm for pixel classification based on the discrepancy norm. In *Proc. FUZZ-IEEE'96* (1996), vol. III, pp. 2007–2012.

[3] Bodenhofer, U. Tuning of fuzzy systems using genetic algorithms. Master's thesis, Johannes Kepler Universität Linz, March 1996.

[4] Bodenhofer, U., and Herrera, F. Ten lectures on genetic fuzzy systems. In *Preprints of the International Summer School: Advanced Control—Fuzzy, Neural, Genetic*, R. Mesiar, Ed. Slovak Technical University, Bratislava, 1997, pp. 1–69.

[5] Bonarini, A. ELF: Learning incomplete fuzzy rule sets for an autonomous robot. In *Proc. EUFIT'93* (1993), vol. I, pp. 69–75.

[6] Bonarini, A. Evolutionary learning of fuzzy rules: Competition and cooperation. In *Fuzzy Modeling: Paradigms and Practice*, W. Pedrycz, Ed. Kluwer Academic Publishers, Dordrecht, 1996, pp. 265–283.

[7] Bulirsch, R., and Stoer, J. *Introduction to Numerical Analysis.* Springer, Berlin, 1980.

[8] Chen, C. L., and Chang, M. H. An enhanced genetic algorithm. In *Proc. EUFIT'93* (1993), vol. II, pp. 1105–1109.

[9] Darwin, C. R. *On the Origin of Species by means of Natural Selection* and *The Descent of Man and Selection in Relation to Sex*, third ed., vol. 49 of *Great Books of the Western World,* Editor in chief: M. J. Adler. Robert P. Gwinn, Chicago, IL, 1991. First edition John Murray, London, 1859.

[10] ENGESSER, H., Ed. *Duden Informatik: Ein Sachlexikon für Studium und Praxis*, second ed. Brockhaus, Mannheim, 1993.

[11] GEYER-SCHULZ, A. *Fuzzy Rule-Based Expert Systems and Genetic Machine Learning*, vol. 3 of *Studies in Fuzziness*. Physica Verlag, Heidelberg, 1995.

[12] GEYER-SCHULZ, A. The MIT beer distribution game revisited: Genetic machine learning and managerial behavior in a dynamic decision making experiment. In *Genetic Algorithms and Soft Computing*, F. Herrera and J. L. Verdegay, Eds. Physica Verlag, Heidelberg, 1996, pp. 658–682.

[13] GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[14] GOLDBERG, D. E., KORB, B., AND DEB, K. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems 3* (1989), 493–530.

[15] HOFFMANN, F. *Entwurf von Fuzzy-Reglern mit Genetischen Algorithmen*. Deutscher Universitäts-Verlag, Wiesbaden, 1997.

[16] HOLLAND, J. H. *Adaptation in Natural and Artificial Systems*, first MIT Press ed. The MIT Press, Cambridge, MA, 1992. First edition: University of Michigan Press, 1975.

[17] HOLLAND, J. H., HOLYOAK, K. J., NISBETT, R. E., AND THAGARD, P. R. *Induction: Processes of Inference, Learning, and Discovery*. Computational Models of Cognition and Perception. The MIT Press, Cambridge, MA, 1986.

[18] HÖRNER, H. A C++ class library for genetic programming: The Vienna university of economics genetic programming kernel. Tech. rep., Vienna University of Economics, May 1996.

[19] KOZA, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, MA, 1992.

[20] KRUSE, R., GEBHARDT, J., AND KLAWONN, F. *Fuzzy-Systeme*. B. G. Teubner, Stuttgart, 1993.

[21] KRUSE, R., GEBHARDT, J., AND KLAWONN, F. *Foundations of Fuzzy Systems*. John Wiley & Sons, New York, 1994.

[22] NEUNZERT, H., AND WETTON, B. Pattern recognition using measure space metrics. Tech. Rep. 28, Universität Kaiserslautern, Fachbereich Mathematik, November 1987.

[23] OTTEN, R. H. J. M., AND VAN GINNEKEN, L. P. P. P. *The Annealing Algorithm*. Kluwer Academic Publishers, Boston, 1989.

[24] PARODI, A., AND BONELLI, P. A new approach to fuzzy classifier systems. In *Proc. ICGA'97* (Los Altos, CA, 1993), S. Forrest, Ed., Morgan Kaufmann, pp. 223–230.

[25] RECHENBERG, I. *Evolutionsstrategie*, vol. 15 of *Problemata*. Friedrich Frommann Verlag (Günther Holzboog KG), Stuttgart, 1973.

[26] RUMELHART, D. E., AND MCCLELLAND, J. L. *Parallel Distributed Processing—Exploration in the Microstructures of Cognition, Volume I: Foundations*. MIT Press, Cambridge, MA, 1986.

[27] TILLI, T. *Automatisierung mit Fuzzy-Logik*. Franzis-Verlag, München, 1992.

[28] VALENZUELA-RENDÓN, M. The fuzzy classifier system: A classifier system for continuously varying variables. In *Proc. ICGA'91* (San Mateo, CA, 1991), R. K. Belew and L. B. Booker, Eds., Morgan Kaufmann, pp. 346–353.

[29] VALENZUELA-RENDÓN, M. The fuzzy classifier system: Motivations and first results. In *Parallel Problem Solving from Nature*, H.-P. Schwefel and R. Männer, Eds. Springer, Berlin, 1991, pp. 330–334.

[30] VAN LAARHOVEN, P. J. M., AND AARTS, E. H. L. *Simulated Annealing: Theory and Applications*. Kluwer Academic Publishers, Dordrecht, 1987.

[31] ZIMMERMANN, H.-J. *Fuzzy Set Theory—and its Applications*, second ed. Kluwer Academic Publishers, Boston, 1991.

[32] ZURADA, J. M. *Introduction to Artificial Neural Networks*. West Publishing, St. Paul, 1992.