INTRODUCTION TO EVOLUTIONARY COMPUTATION

MATTIAS WAHDE

Department of Applied Mechanics
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2007

Introduction to Evolutionary Computation

MATTIAS WAHDE

Department of Applied Mechanics
Chalmers University of Technology
SE–412 96 Göteborg
Sweden
Telephone: +46 (0)31–772 1000

Contents

Chapter 1

Biological basis of evolutionary algorithms

1.1 Biology as a source of inspiration

Many adaptive algorithms in use today are inspired by the properties of biological systems in one way or another. Since nature is all about adaptation, this should not come as a surprise.

Evolution, the process of gradual hereditary change which has given rise to immensely complex biological structures, can be considered as a kind of ”meta–inspiration”, since it is evolutionary processes that have produced the assemblies of brain cells on which computational structures such as artificial neural networks are based. The search and optimization algorithms known as evolutionary algorithms are directly inspired by Darwinian evolution. Knowing that there are many other optimization algorithms available, one might ask oneself why algorithms based on evolution should be introduced. There are several reasons. First of all, as we shall see later, evolution – whether artificial or natural – has, under the right conditions, the ability to avoid getting stuck at *local* optima in a search space. Thus, given enough time, an evolutionary algorithm usually finds a solution close to the global optimum.

Furthermore, due partly to its stochastic nature, evolution can find several different (and equally viable) solutions to a given problem. The great diversity of species in nature, for instance, tells us that there are many different solutions to the problem of survival. Another classical example is the evolution of the eye. Richard Dawkins notes in his book ”Climbing Mount Improbable” [5] that the eye has evolved in forty (!) different ways, completely independently of each other. Thus, when nature was given the task of designing light–gathering devices to improve the chances of survival of previously blind species, a large number of different solutions were discovered. Two examples are the com-

pound eyes of insects and the lens eyes of mammals. There is a whole range of complexity from simple eyes which barely distinguish light from darkness, to strikingly complex eyes which provide their owner with very acute vision.

In artificial evolution, the ability of the algorithm to come up with solutions that were not anticipated is very important. Particularly in complex problems, it often happens that an evolutionary algorithm finds a solution which is remarkably simple, yet very difficult to arrive at by other means.

The ability of evolution to find different successful solutions applies not only to the construction of eyes, but also to any complex biological system such as the echo–location circuits of the bat or, of course, the human brain as a whole. Thus, evolution *works*, and the idea of inventing an algorithm based on it is hardly far–fetched.

Biologically inspired computation methods have begun to play a strong role in science and engineering. However, one should not exaggerate the analogy between the processes going on in biological systems and the adaptive algorithms that are used for solving engineering problems. At best, these algorithms are only a caricature of their biological counterparts. However, the intention is not to *reproduce* nature. Biology should serve as an inspiration for adaptive algorithms, not as a dictator.

In retrospect, one may wonder why biologically inspired computation only appeared in the mid–20th century, and gained widespread acceptance (to the extent that it has) only toward the very end of the 20th century. This lapse is rapidly being repaired though, and the computational methods described here are now becoming standard tools in many scientific disciplines.

1.2 Biological background and terminology

The algorithms considered in this course, namely evolutionary algorithms (EAs) (of which genetic algorithms (GAs) are a special case), are strongly inspired by biology, and most of the terminology regarding EAs therefore has its origin in biology. We will thus begin our study of EAs with a short introduction to the relevant biological terms and concepts.

As the name implies, EAs are based on processes similar to those that occur during biological evolution. A central concept in the theory of evolutionary change is the notion of a **population**, by which we mean a group of individuals of the same **species** (i.e. that can mate and have fertile offspring), normally confined to some particular area in which the members of the population live, reproduce, and die. The members of the population are referred to as **individuals**. In cases where members of the same species are separated by, for instance, a body of water or a mountain range, they form separate populations. Given enough time, **speciation** (i.e. the formation of new species) may occur.

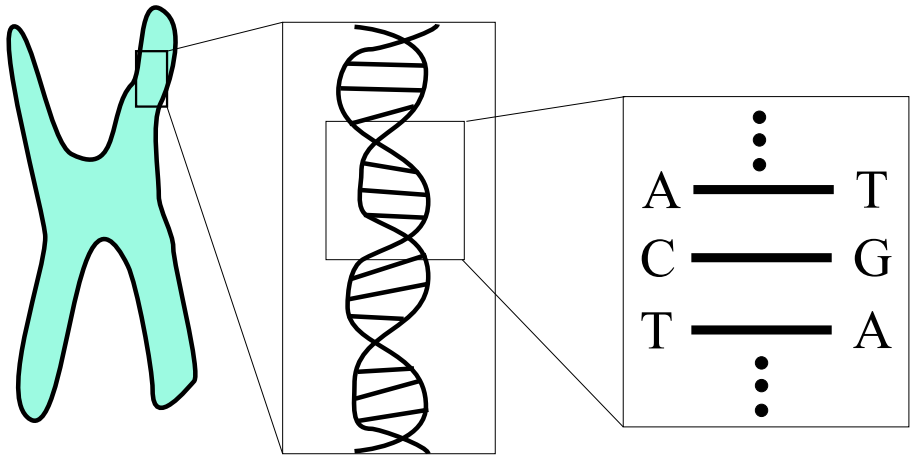

Figure 1.1: *A chromosome is shown in the left side of the figure. The two blow-ups on the right show the individual base pairs. Note that A is always paired with T, and C is always paired with G.*

One of the central ideas behind **Darwin's theory of evolution** is the idea of gradual, hereditary change: new features in a species, such as protective cover or fangs, evolve gradually in response to a challenge provided by the environment. For instance, in a species of predators, longer and sharper fangs may evolve as a result of the evolution of thicker skin in their prey. This gradual arms race between two species is known as **co–evolution**.

The central concept here is **heredity**, i.e the idea that the properties of an individual can be encoded in such a way that they can be transmitted to the next generation when, and if, an individual reproduces. Before (and even a significant time after) Darwin, this concept was not universally accepted, to put it mildly. There were competing theories of evolutionary change, as well as those who claimed that evolutionary change did not occur at all.

One such theory, known as the **Lamarckian theory** suggested that traits that were acquired by the individual during its lifetime could be transferred to the next generation. Thus, according to this theory, a person that acquired a particular skill in, say, painting, could transfer this skill to his or her children. However, this theory has been refuted, and it is known that the information encoded in the sex cells of an individual does not change during the lifetime of the individual, at least not as a result of learning. (Radiation, chemicals etc. may cause changes).

However, it should also be remembered that, at least for higher animals, the success or failure of an individual is of course not only determined by the genes: the environment in which the individual lives is also of great importance.

Anyway, let us return to the concept of heredity. How is the transferable information stored in an individual? The answer is that each individual of

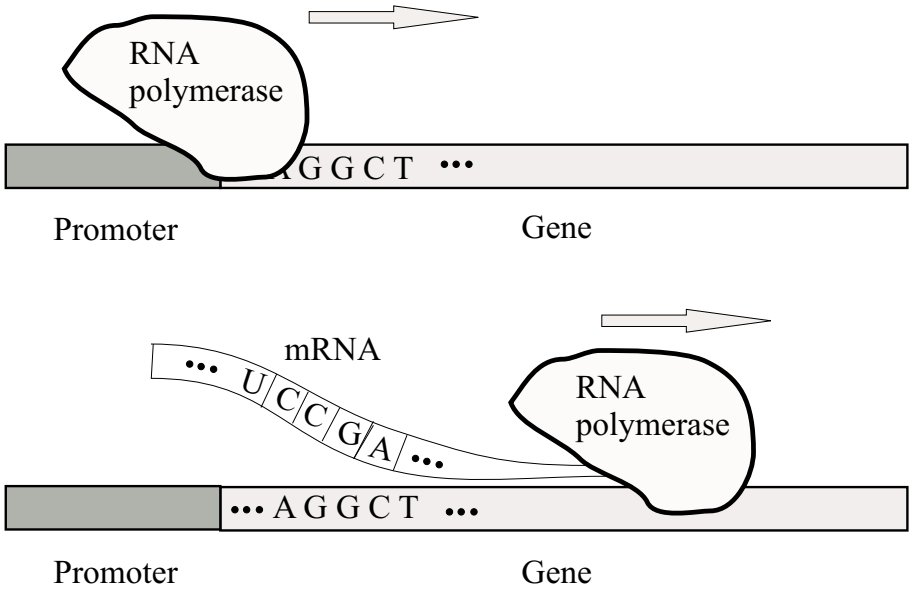

Figure 1.2: *The transcription process. The RNA polymerase binds to a promoter region, and then moves along the DNA molecule, generating an mRNA molecule.*

a species carries (in each cell of its body) a **genome** that, in higher animals, consists of *several* **chromosomes** in the form of DNA molecules. Each chromosome, in turn, contains a large number of **genes**, which are the units of heredity and which encode the information needed to build and maintain an individual. Each gene is composed, essentially, of a sequence of **bases**. There are four bases in chromosomes (or DNA molecules), denoted A,C,G, and T. Thus, the information is stored in a digital fashion, using an alphabet with four symbols, as illustrated in Fig. 1.1.

During development, as well as during the life of an individual, the DNA is read by an enzyme called **RNA polymerase**, and this process, known as **transcription**, produces another type of molecule called **messenger RNA** (mRNA). In this process, the DNA is (temporarily) split, so that the RNA polymerase can access one sequence of bases. Then, during the formation of the mRNA molecule, the RNA polymerase moves along the DNA molecule, using it as a template to form a chain of bases, as shown in Fig. 1.2. Note that the T base is replaced by another base (U) in mRNA molecules. Next, the **proteins**, which are chains of **amino acids**, are generated in a process called **translation** using mRNA as a template. A greatly simplified version of the translation process is given in Fig. 1.3. In translation, which takes place in the **ribosomes**, each sequence of three bases (referred to as a **codon**) codes for an amino acid, or for a *start* or *stop* command. Thus, for example, the sequence AUG is (normally) used as the start codon. Since there are only 20 amino acids, and $4^3 = 64$ possible three-letter combinations using the alphabet A, C, G, U, there is some

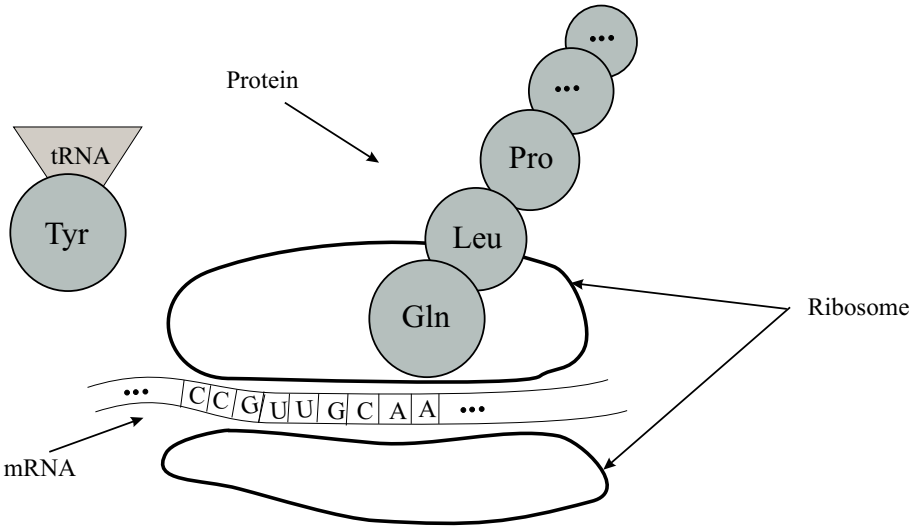

Figure 1.3: *The translation process. The tRNA molecules transport amino acids to the ribosomes. In the case shown in the figure, a tRNA molecule is just arriving, carrying a* Tyrosine *amino acid. The growing chain of amino acids generated at the ribosomes will, when completed, form a protein. In this example, the amino acid* Glutamine *(Gln), encoded by the sequence CAA, has just been added to the protein. Note that the figure is greatly simplifed; in reality, the process is quite complex.*

redundancy in the code. For example, the codons GCU, GCC, GCA, and GCG all code for the amino acid *Alanine*. Other amino acids, such as e.g. *Tryptophan* are encoded only by one single sequence (UGG in the case of *Tryptophan*). As shown in Fig. 1.3, the amino acids are transported to the ribosomes by **transfer-RNA** (tRNA) molecules, and are then used in the growing amino acid chain forming the protein. Proteins are the building blocks of life, and are involved in one way or another in almost every activity that takes place inside the living cell. However, not all genes are **expressed** (i.e. active) at all times. For example, some genes are primarily active during the development of the individual, whereas other genes are active during adult life. Also, different cells in a body may show different patterns of activity, even though the genetic material in each cell of a given individual is the same. Furthermore, as will be discussed in Subsect. 1.3.1 below, the function of many genes is simply to regulate (via their protein products) the activity of other genes.

Returning the structure of the DNA molecules, it should be noted that each gene can have several settings. As a simple example, consider a gene that encodes eye color in humans. There are several options available: eyes may be green, brown, blue etc. The settings of a gene are known as **alleles**. Of course, not all genes encode something that is as easy to visualize as eye color, but we can still accept the idea that each gene has several settings available. Now, the complete genome of an individual, with all its settings (encoding e.g. hair

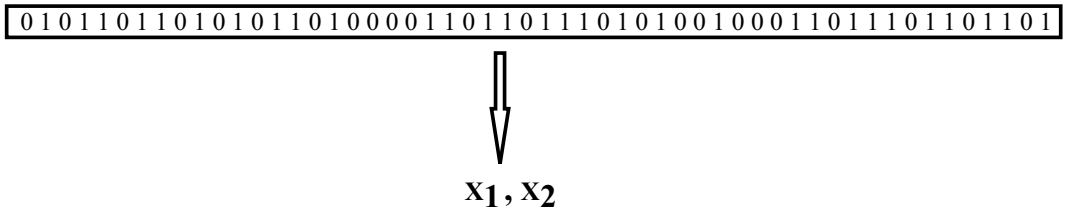

Figure 1.4: *Typical usage of a chromosome in a genetic algorithm. The 0s and 1s are the genes, which are used as binary numbers to form, in this case, two variables x_1 and x_2, using e.g. the first half of the chromosome to represent x_1 and the second half to represent x_2.*

color, eye color, etc.) is known as the **genotype**.

During development, the stored information is decoded, resulting in an individual carrying the traits encoded in the genome. The individual, with all its traits, is known as the **phenotype**, corresponding to the genotype.

Two central concepts in evolution are **fitness** and **selection** (for reproduction), and these concepts are often intertwined: individuals that are well adapted to their environment (which includes not only the climate and geography of the region where the individual lives, but also other members of the same species, as well as members of other species), i.e. those that are stronger or smarter than the others, have a larger chance to reproduce, and thus to spread their genetic material, resulting in more individuals having these properties etc.

Reproduction is the central moment for evolutionary change. Simplifying somewhat, we may say that during this process, the chromosomes of two (in the case of **sexual reproduction**) individuals are combined, some genes being taken from one parent and others from the other parent. The copying of genetic information takes place with remarkable accuracy, but nevertheless there occurs some errors. These errors are known as **mutations**, and constitute the providers of new information for evolution to act upon. In some simple species (e.g. bacteria) sexual reproduction does not occur. Instead, these species use **asexual reproduction**, in which only one parent is involved.

1.3 Biology vs. evolutionary algorithms

We should pause here to note that this description of reproduction (and indeed of evolution altogether) is greatly simplified: for example, in higher animals, the chromosomes are paired, allowing such concepts as recessive traits etc. Furthermore, not all parts of DNA are actually used in the production of an individual: a large part of the genetic information is dormant (but may come to be used in later generations).

Another simplification comes from the way chromosomes are used in evo-

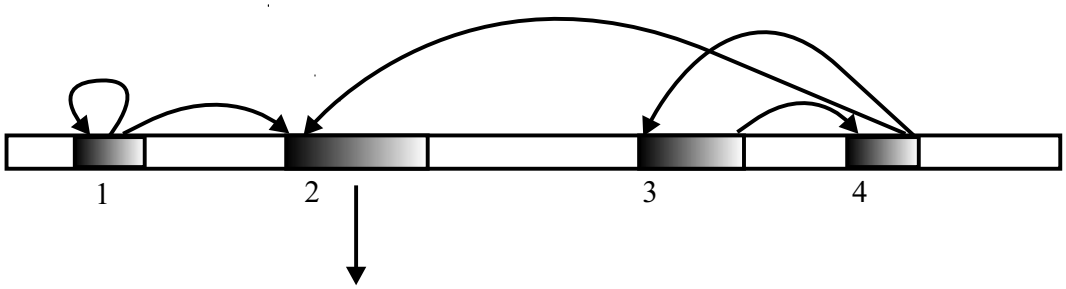

Figure 1.5: *A schematic illustration of the function of the genome in a biological system. Four genes are shown. Genes 1,3, and 4 are transcription factors, which regulates each other's levels of expressions, as well as that of gene 2, which is a structural gene. The arrow below gene 2 indicates that its product is used in the cell for some other purpose than gene regulation. Note that the figure is greatly simplified in that the intermediate step of translation is not shown.*

lutionary algorithms. The most common usage is illustrated in Fig. 1.4. The figure shows the typical procedure used when generating an individual from a chromosome in a genetic algorithm. As can be seen from the figure, the chromosome is used as a lookup table from which the traits of the corresponding individual are obtained. In the simple case shown in the figure, the individual obtained consists of two variables x_1 and x_2 which can be used e.g. in a function optimization problem.

1.3.1 Embryological development

The simple procedure shown in Fig. 1.4 is, in fact, just a caricature of the process taking place when an individual is generated in a biological system. The biological process is illustrated schematically in Fig. 1.5. First of all, in biological systems, the chromosome is *not* used as a simple lookup table. Instead, genes interact with each other to form complex **genetic regulatory networks**, in which the **activity** (i.e. the level of production of mRNA) of a gene often is regulated by the activity of several other genes [15]. In such cases, the product of a gene (i.e. a protein) may attach itself to an **operator** close (on the DNA molecule) to another gene, and thereby affect, i.e. increase or decrease, the ability of RNA polymerase to bind to the DNA molecule at the starting position of the gene (the **promoter region**).

Genes that regulate other genes are called **regulatory genes** or **transcription factors**. Some regulatory genes regulate their own expression, forming a direct feedback loop, which can act to keep the activity of the gene within specific bounds, see gene 1 in Fig. 1.5.

Gene regulation can occur in other ways as well. For example, a regulatory gene may activate a protein (i.e. the product of another gene) which then, in turn, may affect the transcription of other genes. Genes that have other

tasks than regulating genes are called **structural genes**. Such genes produce the many proteins needed for a body to function, i.e. proteins that appear in muscle tissues. In addition, many structural genes code for **enzymes**, which are proteins that catalyze various chemical reactions, such as e.g. breakdown of sugars.

During embryological development of an individual, the genome thus executes a complex program, resulting in a complete individual.

1.3.2 Multicellularity

An additional simplification in most evolutionary algorithms is the absence of **multicellularity**. By contrast, in biological systems, the development of an individual results in a system of many cells (except, of course, in the case of unicellular organisms), and the level of gene expression in each cell is determined by its interaction with other cells. Thus, signalling between cells is an important factor in biological systems. Note, however, that the genome is the *same* in all cells in the body. It is only the *expression* of genes that varies between cells, determining e.g. if a cell should become part of the brain (a neuron) or part of the muscle tissue.

1.3.3 Gene regulation and learning

In evolutionary algorithms, the individual resulting from the decoding procedure shown in Fig. 1.4 is usually fixed during its evaluation. However, in biological systems, the genome remains active throughout the life time of the individuals, and continues to produce the proteins needed in the body. In a computer analogy, the embryological development described above can be considered as a subroutine which comes to a halt when the individual is born. At that time, another subroutine, responsible for the growth of the newborn individual is activated, and is then followed by a subroutine active during adult life.

Among other things, it has been found that gene regulation affects learning. For example, in the nematode worm C. Elegans, it is known that the **knockout** (deactivation) of only two genes, aptly named *lrn-1* and *lrn-2*, destroys the ability of the animal to perform associative learning, i.e. to associate a substance with a reward or a punishment [3].

Much of what is known about the relation between gene regulation and learning comes from studies of the giant sea slug *Aplysia*, carried out by Kandel and co-workers [12]. This particular animal was chosen since it has a rather small number of neurons (around 20,000). Moreover, the neurons of *Aplysia* are quite large, and are therefore relatively easy to manipulate and study. Kandel *et al.* studied **sensitization**, i.e. a heigthened state of alertness that may occur

in an animal if its body is touched, much like a person will become sensitive to *any* sound immediately after being startled by a loud noise (e.g. a gunshot). In the case of *Aplysia*, a light touch on the tail sensitizes the animal for several minutes, so that, upon receiving another touch, on a different part of the body, its reaction will be stronger than it would have been, had the tail not been touched. A single touch leads to sensitization that lasts for a few minutes. However, if the animal is touched several times during a short time span, its memory of the event lasts longer (several days). Kandel *et al.* were able to show that, unlike the short-term process, long-term sensitization requires the synthesis of new proteins. In other words, long-term memory storage requires gene expression, and thus a link is formed between genes and memory formation (learning).

1.3.4 Bioinformatics and artificial life

Even after the mapping of the human genome, we do not have complete knowledge of how a human develops. Instead, work can now begin to try to uncover the complex interactions between genes.

Genetic regulatory networks are one of the topics of a new scientific discipline called **bioinformatics**, which combines computer science, biology, biochemistry, and mathematics in an attempt to understand how genes interact with each other.

Thus, we have allowed ourselves to make a few simplifications in the description of evolution, and yet more simplifications will be made when we begin to study evolutionary algorithms. Again it is important to note that we use biology as an inspiration, and are free to use whichever part of biological reality we find convenient. Note also that many of the details of evolution could have turned out differently, if the climate and chemical conditions on Earth had been different. For example, biological systems could have stored information using a binary alphabet (as in computers) instead of a tertiary alphabet (A,C,G,T). While the study of ”life as it is” is the subject of biology, the study of ”life as it could be” is the subject of a rather new discipline of science known as **artificial life** or ALIFE, which has strong connections with evolutionary biology.

1.4 Summary

To summarize our description of evolution, we note that it is a process that acts on populations of individuals. Information is stored in individuals in the form of chromosomes, consisting of many genes. Individuals that are well adapted to their environment are able to initiate the formation of new individuals through the process of reproduction, which combines the genetic information

Figure 1.6: *Two flying machines that were not assembled by evolution: their parts do not directly encode the information from previous generations. By contrast, a bird is a flying machine assembled by evolution, and its cells contain the information needed to make another flying machine of the same type. Photo: NASA.*

of two separate individuals. Mutations provide further material for the evolutionary process.

Finally, we should point out that evolution is *not* a random search through the space of possible biological entities: mutations are random, but selection is *not*. This may seem obvious, but has fooled many people, including well–known researchers who ought to have known better. One example is that of a researcher who claimed that belief in evolution was equivalent to believing that a storm passing through a junk yard can assemble an aircraft. This comparison is absolutely false, for two reasons; First, evolution is a theory of *gradual* change. No one claims that an aircraft, or the sonar system of a bat, or the brain of a human could be assembled in one step, corresponding to one giant mutation. Instead, evolution gradually builds more complex structures, by selecting those individuals that are most fit. Second, evolution involves *hereditary* change: it does not start from scratch with each new individual. Instead, in a way, every individual has in its genome the entire history of its species, and the evolutionary process has access to all this information as it tries to improve on an already present design.

Chapter 2

Basics of evolutionary algorithms

There are many computational methods inspired by biological evolution, and these methods are collected under the umbrella term **evolutionary algorithms (EAs)**. The most common type of EA is the **genetic algorithm (GA)**, which was developed by John Holland [10] and others, and which we will introduce in this chapter. Other algorithms, such as e.g. **genetic programming (GP)** will be introduced in Handout 6.

2.1 A simple genetic algorithm

When a GA is to be used for solving an optimization problem, the variables of the problem are encoded in strings known as chromosomes. Each chromosome consists of a number of elements known as genes, in accordance with the biological terminology introduced previously. The genes **encode** the information stored in the chromosome, and there are several different encoding schemes. In the first GAs that were developed, the encoding scheme was **binary**, meaning that the genes could only take the values (alleles) 0 or 1. Later on, other encoding schemes have been introduced as well.

When the algorithm is initialized, a population (i.e. a set) of N chromosomes are generated, by assigning random values to the genes in the chromosomes. When a chromosome is decoded, the corresponding individual is obtained, and the N individuals obtained from the N chromosomes form the first **generation**.

The exact procedure by which an individual is obtained from a chromosome varies from problem to problem. Let us now turn to an example.

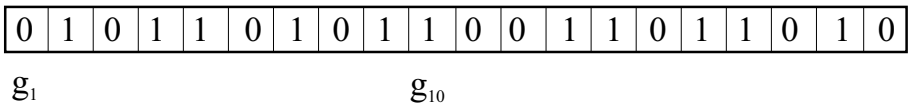

Figure 2.1: *A chromosome encoding two variables with 10–digit binary accuracy.*

Example 2.1 Consider the problem of finding the maximum of a simple function, namely

$$h(x_1, x_2) = \mathrm{e}^{-(x_1^2+x_2^2)}, \tag{2.1}$$

in the interval $x_1, x_2 \in [-2, 2]$. In the given interval, the function g obviously has its extremum (=1) at the point $x_1 = x_2 = 0$. Using binary encoding, a typical chromosome for this problem can take the form shown in Fig. 2.1. The procedure for decoding this chromosome operates as follows: the value of the first gene in the chromosome, denoted g_1, is multiplied by $2^{-1} = 0.5$, the second by $2^{-2} = 0.25$ etc. down to the tenth gene, whose value is multiplied by 2^{-10}. The ten numbers thus obtained are then added to form a temporary variable

$$x_{1,\mathrm{tmp}} = \sum_{k=1}^{10} 2^{-k} g_k \tag{2.2}$$

Alternatively, $x_{1,\mathrm{tmp}}$ can be obtained in a slightly more elegant way

$$x_{1,\mathrm{tmp}} = \frac{1}{2}(g_1 + \frac{1}{2}(g_2 + \frac{1}{2}(g_3 + \ldots) \tag{2.3}$$

This value is transformed to the requested interval $[-2, 2]$ by the transformation

$$x_1 = -2 + 4x_{1,\mathrm{tmp}}. \tag{2.4}$$

With the 10–digit encoding scheme used above, the range of the temporary variable will be from 0 to 0.999023, since

$$\sum_{k=1}^{10} 2^{-k} = \frac{2^{10} - 1}{2^{10}} \approx 0.999023 \tag{2.5}$$

Thus, the interval $[-2, 2]$ is not fully covered; the largest possible value of $x_1 = -2 + 4 \times 0.999023 = 1.99609$. In this example, where the maximum is not located on the boundary of the interval, the limited accuracy does not pose any problem. However, in many cases the maximum *is* located on the boundary. In such cases it is important that the possible values of x_1 should be able to reach the boundary of the interval. In the case above this could have been achieved by multiplying $x_{1,\mathrm{tmp}}$ by $4/0.999023$ rather than 4.

A similar transformation is applied to obtain the value of x_2

$$x_2 = -2 + 4x_{2,\mathrm{tmp}}, \tag{2.6}$$

where

$$x_{2,\mathrm{tmp}} = \sum_{k=1}^{10} 2^{-k} g_{k+10} \tag{2.7}$$

Thus, for each chromosome the corresponding individual, consisting of a value of x_1 and a value of x_2, can be obtained.

Now, when the N chromosomes have been initialized, they are decoded and the individuals are evaluated one at a time. The evaluation procedure is usually the most time–consuming part of any GA. It commonly uses more than 99% of the CPU time, the remaining 1% or less being spent on the decoding of chromosomes and the formation of new individuals (see below).

In the case of function maximization, the evaluation is simple: The two variables x_1 and x_2 obtained from the chromosome are used to form $h(x_1, x_2)$. Now, since the aim is to find the maximum of the function, the higher the value of h, the better the individual. Using biological terminology, we say that an individual with a high value of h has higher fitness, denoted f, than an individual with a low value of h. In this case, therefore, we have for individual i

$$f(i) = h(x_1, x_2), \tag{2.8}$$

where the values of x_1 and x_2 have been obtained from the chromosome corresponding to individual i. The procedure of decoding the chromosome, evaluating the individual, and making the fitness assignment, is repeated until all N individuals forming the first generation have been evaluated. The next step is to form the second generation. In analogy with the biological introduction given above, the concepts of selection of fit individuals, followed by crossover and mutation, are used in this phase.

Selection can be carried out in several different ways. In **roulette wheel selection**, two individuals are selected from the whole population using a fitness–proportionate selection procedure, in which each individual is assigned a slice of a roulette wheel, with an opening angle proportional to its fitness. Now, as the wheel is turned, an individual with high fitness has a greater chance of being selected than an individual with low fitness, and the probability is directly proportional to the fitness. In practice, one does not turn a roulette wheel. Instead, a random number r between 0 and 1 is drawn, and the selected individual is taken to be the one with the population index (see Example 2.2) corresponding to the smallest value j which satisfies the inequality

$$\frac{\sum_{i=1}^{j} f(i)}{\sum_{i=1}^{N} f(i)} > r. \tag{2.9}$$

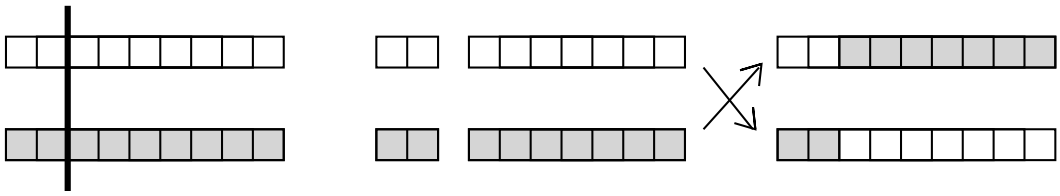

Figure 2.2: *The crossover procedure. The crossover point is chosen at random.*

Example 2.2 Consider a case in which the population consists of only 4 individuals, having fitnesses $f(1) = 0.2$, $f(2) = 0.5$, $f(3) = 0.3$, and $f(4) = 0.6$. The **population index** is used to enumerate the individuals. Thus, the first individual, with fitness 0.2 has population index 1, the second individual population index 2 etc.

Now, one can imagine a roulette wheel divided into four sectors, one for each individual, with sizes proportional to the fitness values of the individuals. Thus the individual with fitness 0.2 would occupy $0.2/(0.2 + 0.5 + 0.3 + 0.6) = 0.125 = 12.5\%$ of the circle, and therefore have a 12.5% chance of being selected etc.

By generating two random numbers, two individuals can be selected according to Eq. (2.9). Note that an individual can be selected several times, i.e. the selection procedure operates *with* replacement.

When two individuals have been selected, two new individuals are formed by crossover and mutation. In **crossover**, a **crossover point** is selected randomly. The two chromosomes are then cut at the crossover point, and the first part of the first chromosome is joined with the second part of the second chromosome, and vice versa. The crossover procedure is illustrated in Fig. 2.2. In the figure, the crossover point was (randomly) chosen between the second and third genes.

The final step in the formation of new individuals is mutation . Each of the two new chromosomes formed by crossover is subjected to mutation, which operates by drawing a random number for each gene along the chromosome. If this random number is smaller than the **mutation probability**, the corresponding gene is assigned a new random value in the allowed range. The procedure is illustrated in Fig. 2.3.

The two old individuals are discarded and replaced by the two new individuals. Often, **generational replacement** is used. In this case, the entire old generation is replaced by new individuals, obtained by repeating the proce-

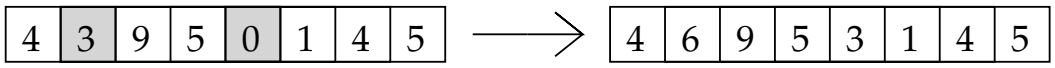

Figure 2.3: *Mutation, shown for the case of discrete decimal encoding. For each gene, a random number is drawn and compared with the mutation probability.*

dure of selection, crossover, and mutation $N/2$ times.

When the new (second) generation has been formed, it is evaluated in exactly the same way as the first generation, and is then replaced by the third generation etc. The procedure is repeated until a satisfactory solution to the problem has been found.

Example 2.1, continued Returning to the example involving the search for the maximum of $h(x_1, x_2)$, the progress of the GA can be illustrated by showing the location in the xy–plane of the individuals at successive generations. This is done in Fig. 2.4. The population size was constant and equal to 10. However, some individuals in later generations had almost equal chromosomes, so that the number of individuals in some frames appears to be less than 10.

2.2 Components of genetic algorithms

In the example above, one version of a very simple GA was presented. Here, some of the different operators involved in artificial evolution will be described.

It is important to realize that some parts of the implementation of a GA are strongly problem dependent, and so cannot be completely described in general terms. For instance, the construction of the decoding procedure, which turns a chromosome into an individual that can be evaluated, is different in each case, and the same applies to the construction of the fitness function.

As mentioned in the beginning of the chapter, the discussion is centered around genetic algorithms. However, many of the operators used in GAs are the same (or only slightly different) when used in connection with other versions of EAs, as we shall see in Handout 6.

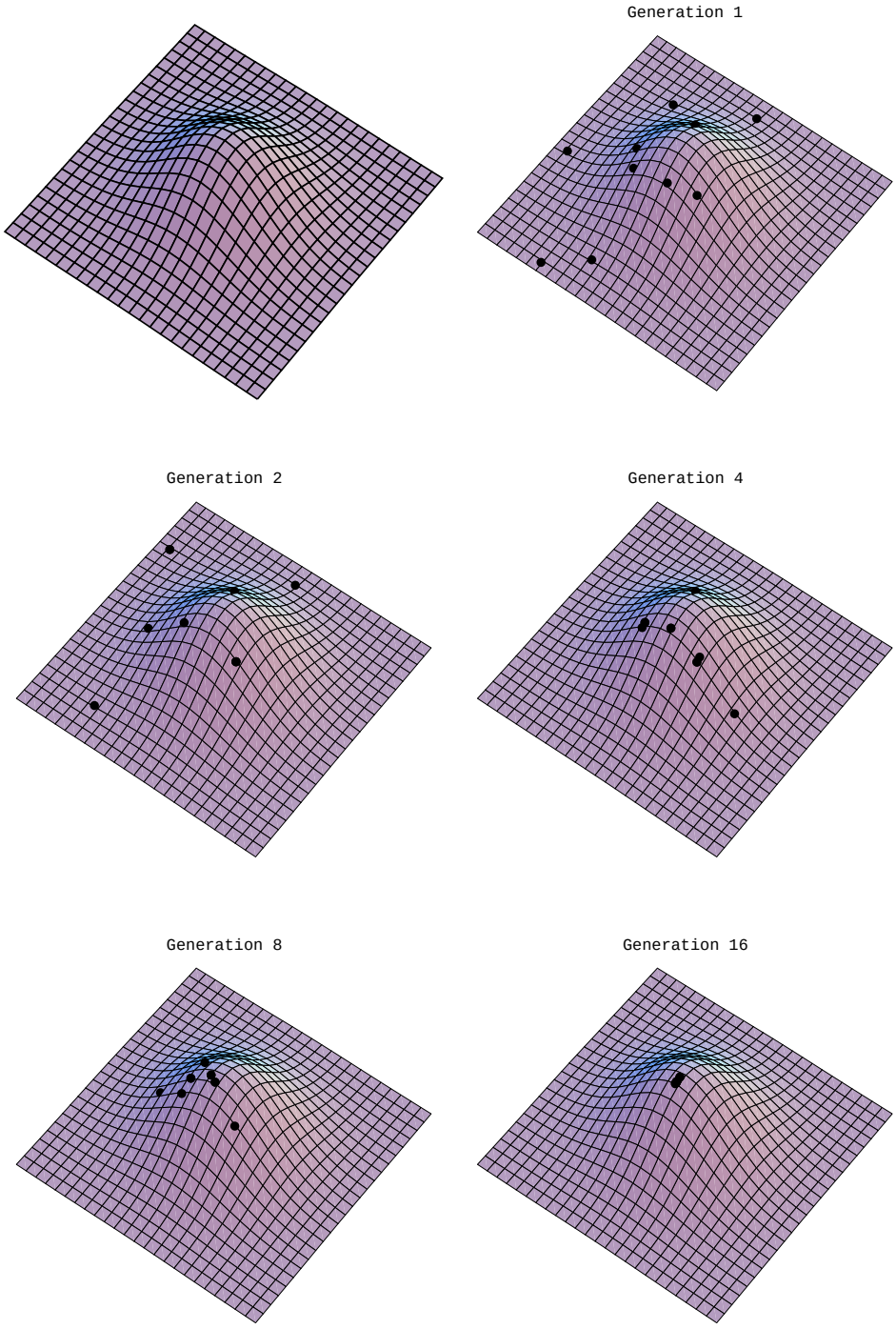

Figure 2.4: *The progress of a GA searching for the maximum of the function* $h(x_1, x_2) = \mathrm{e}^{-(x_1^2+x_2^2)}$*. The population size was equal to 10. Note how the population gradually converges to the maximum of the function.*

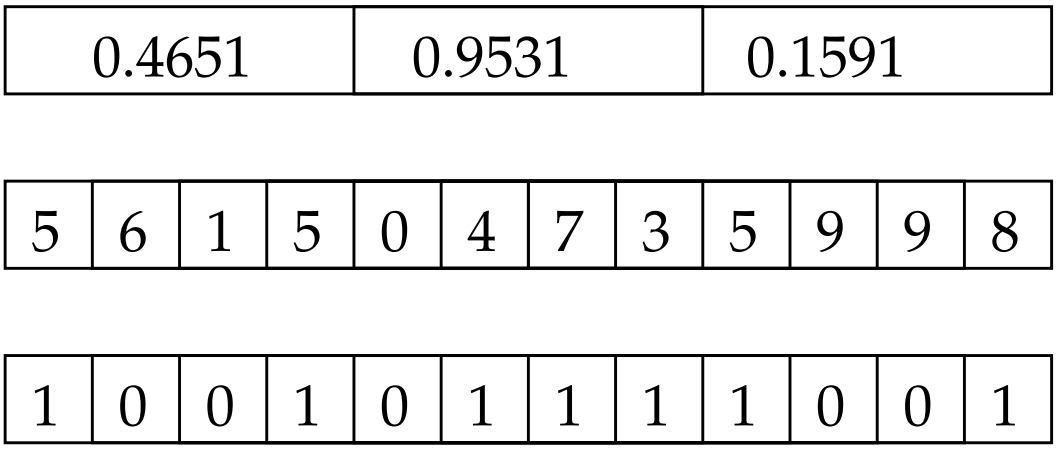

Figure 2.5: *Real–number encoding (top), discrete decimal encoding (middle), and binary encoding (bottom).*

2.2.1 Encoding scheme

The representation for the individual genes in a chromosome can be chosen in several different ways. The three most common are **binary encoding**, where genes take the values 0 or 1, **discrete decimal encoding**, where genes take integer values in the range $[0, 9]$, and **real–number encoding**, where genes take any value in the range $[0, R]$, where R is a non-negative real number (usually 1). As is often the case with GAs, it is not possible to say that one encoding scheme is always superior to the others, and it is also difficult to make a fair comparison between the various encoding schemes. Real–number encoding schemes often use slightly different mutation methods (see below), which improve their performance in a way that is not available for binary encodings. The three encoding schemes are shown in Fig. 2.5.

In real–number encoding, a single gene g is used to represent a number between 0 and 1. This number is then rescaled to the appropriate range, according to

$$x = -d + 2dg, \tag{2.10}$$

assuming that the standard value $R = 1$ has been chosen for the range of allowed gene values. The decoding procedure for discrete decimal encoding is given by

$$x = -d + 2d\left(g_1 \times 10^{-1} + g_2 \times 10^{-2} + g_3 \times 10^{-3} + \ldots\right), \tag{2.11}$$

giving a number x in the range $[-d, d[$. Similarly, for binary encoding, the procedure is

$$x = -d + 2d\left(g_1 \times 2^{-1} + g_2 \times 2^{-2} + g_3 \times 2^{-3} + \ldots\right). \tag{2.12}$$

In addition to the encoding schemes shown here, other encoding schemes exist as well. For example, in the traveling salesperson problem (TSP), see Handout 5, the chromosome may represent a permutation of the nodes involved in the problem. Each such permutation will generate a valid path for TSP, i.e. one in which each node is visited, and in which no node is visited twice. In addition, chromosomes may represent complicated structures such as neural networks, finite state machines etc. However, in these cases, the corresponding algorithm is usually not called a *genetic* algorithm; Instead, the umbrella term *evolutionary* algorithm is used, the term genetic algorithm being reserved for cases involving simple, linear chromosomes of the kind shown in Fig. 2.5.

2.2.2 Fitness assignment

The evaluation of an individual leads to a fitness assignment, which conveys information on the performance of the individual. The simplest possible fitness assignment consists of simply assigning the value obtained from the evaluation (assuming a maximization task) without any transformations. This value is known as the **raw fitness** value. For example, if the task is to find the maximum of a simple bounded trigonometric function such as $f(x) = \sin(x)\cos(2x) - \cos(x)\sin(\sqrt{2}x)$ in the interval $[0, 1]$, then the raw fitness value would be useful. However, if instead the task is to find the maximum of the function $g(x) = 1000 + f(x)$, the raw fitness values would be of little use, since they would all be of order 1000. The selection process would find it difficult to single out the best individuals.

Often it is therefore a good idea to *rescale* the fitness values before they are used. **Linear fitness ranking** is a commonly used rescaling procedure, where the best of the N individuals in the population is given fitness N, the second best fitness $N-1$ etc. down to the worst individual who is given fitness 1. Letting $R(i)$ denote the ranking of individual i, defined such that the best individual i_{best} has ranking $R(i_{\text{best}}) = 1$ etc. down to the worst individual with ranking $R(i_{\text{worst}}) = N$, the fitness assignment is given by

$$f(i) = (N + 1 - R(i)). \tag{2.13}$$

Fitness ranking must be used with caution, however, since individuals that are only slightly better than the others may receive very high fitness values and soon come to dominate the population, trapping it in a local optimum.

The tendency to converge on a local optimum can be decreased by using a slightly less extreme fitness ranking, assigning fitness values according to

$$f(i) = f_{\max} - (f_{\max} - f_{\min}) \left(\frac{R(i) - 1}{N - 1} \right). \tag{2.14}$$

This ranking yields equidistant fitness values in the range $[f_{\min}, f_{\max}]$. The simple ranking in Eq. (2.13) is obtained if $f_{\max} = N$, $f_{\min} = 1$.

The choice of the fitness measure often has a strong impact on the results obtained by the GA. In some cases it is quite simple to select a good fitness measure, whereas in other cases it can be very difficult. This topic will be discussed briefly in Handout 3.

2.2.3 Selection

In Example 2.2 above, roulette-wheel selection was used. Roulette-wheel selection is one example of a **fitness-proportionate selection scheme**, in which the expected number of copies of a given individual after selection (neglecting crossover and mutation) is directly proportional to its fitness. Roulette-wheel selection is easy to implement, but has some disadvantages. In addition, it is evident that roulette-wheel selection is a far cry from what happens in nature, where small groups of individuals, usually males, fight each other until there remains a single winner which is allowed to mate. **Tournament selection** tries to incorporate the main features of this process. In its simplest form, tournament selection consists of picking two individuals randomly from the population, and then selecting the best one as a parent. When two parents have been selected this way, crossover and mutation take place as usual. A more sophisticated tournament selection scheme involves selecting m individuals randomly from the population. Next, with probability r, the best of the m individuals is selected, and with probability $1 - r$ a random individual among the other $m - 1$ is selected. m is referred to as the **tournament size**, and r is the **tournament selection parameter**. A typical numerical value for r is around 0.75. The tournament size is commonly set as a (small) fraction of the population size. Thus, unlike the case with roulette-wheel selection, in tournament selection no individual is discarded without at least participating in a close combat involving a small number of individuals. Note also that, in tournament selection, negative fitness values are allowed, whereas all fitness values in roulette-wheel selection must be non-negative.

Both roulette-wheel selection and tournament selection operate with replacement, i.e. selected individuals are returned to the population and can thus be selected again.

In addition to roulette-wheel selection and tournament selection, there are other selection schemes. Some of these will be discussed briefly in Handout 5.

2.2.4 Replacement

In Example 2.1 (Fig. 2.4), we used generational replacement meaning that all individuals in the evaluated generation were replaced by an equal number

of offspring. Generational replacement is not very realistic from a biological point of view. In nature, different generations co–exist and individuals appear (and disappear) constantly, not only at specific intervals of time. By contrast, in generational replacement, there is no competition between individuals from different generations.

In general, replacement schemes can be characterized by their **generational gap** G, which simply measures the fraction of the population that is replaced in each selection cycle, i.e. in each generation. Thus, for generational replacement, $G = 1$.

At the opposite extreme are replacement schemes that only replace *one* individual in each step. In this case $G = 1/N$, where N is the population size. In **steady-state reproduction**, G is usually equal to $1/N$ or $2/N$, i.e. one or two individuals are replaced in each generation. In order to keep the population size constant, NG individuals must be deleted. The deletion procedure varies between different steady-state replacement schemes; in some, only the N parents are considered for deletion, whereas in others, both parents and offspring are considered.

Deletion of individuals can be performed in various ways, e.g. by removing the least fit individuals or by removing the oldest individuals.

As mentioned above, for $G = 1$ (i.e. generational replacement), individuals from different generations do not compete with each other. Note however that there may still be some *de facto* overlap between generations if the crossover and mutation rates are sufficiently low since, in that case, many of the offspring will be identical to their parents.

2.2.5 Crossover

Crossover is one of the most important operators in GAs. It allows partial solutions from different regions of the search space to be assembled into a complete solution to the problem at hand.

While crossover plays a very important role, its effects may be negative if the population size is small, which is almost always the case in artificial evolution where the population size N typically is of order 30–1,000, as compared to populations of several thousands or even millions of individuals in nature. The problem is that, through crossover, a successful (partial) solution will very quickly spread through the population causing it to become rather uniform or even completely uniform, in the absence of mutation. Thus, the population will experience **inbreeding** towards a possibly suboptimal solution.

A possible remedy is to allow crossover or sexual reproduction only with a certain probability p_c. In this case, some new individuals are formed using crossover followed by mutation, and some individuals are formed using **asexual reproduction**, i.e. only mutations. Example 2.1 above had $p_c = 1$. A more

common choice is $p_c = 0.8$.

The crossover procedure can be implemented in various ways. The simplest version is **one–point crossover**, in which a single crossover point is randomly chosen, and the first part of the first chromosome is joined with the second part of the second chromosome, as shown in Fig. 2.2. The procedure can be generalized to n**–point crossover**, where n crossover points are selected randomly, and the chromosome parts are chosen with 50% probability from either parent. In **uniform crossover**, the number of crossover points is equal to the number of genes in the chromosome, minus one.

2.2.6 Mutation

In natural evolution, mutation plays a subordinate, but still important, role providing the two other main operators, selection and crossover, with new material to work with. Most often, mutations are deleterious when they occur but may bring advantages in the long run, for instance when the environment suddenly undergoes changes such that individuals without the mutation in question have difficulties surviving.

In GAs, the value of the mutation probability p_{mut} is usually set by the user at the start of a GA run, and is thereafter left unchanged throughout the run. A typical value for the mutation probability is $1/n$, where n is the number of genes in the chromosome.

There are, however, some versions of EAs, notably evolution strategies, in which the mutation probabilities are allowed to vary. A simple prescription for a varying mutation rate is to increase the mutation probability if the **diversity** in the population falls below a certain value, and to decrease it if the diversity becomes too large.

The diversity measure can be defined as

$$D = \sqrt{\sum_{i=1}^{N} \sum_{j=i+1}^{N} d_{ij}^2}, \tag{2.15}$$

where

$$d_{ij} = \sum_{k=1}^{n} |g_k^i - g_k^j|, \tag{2.16}$$

where g_k^m denotes gene k in chromosome m.

In the case of discrete encodings (either binary or decimal) mutation normally consists of selecting a new random value for the gene in question. In real–number encoding, the modifications obtained by randomly selecting new values often become too large to be useful and therefore an alternative approach, known as **real–number creep**, is frequently used instead. In real–number creep, the mutated value is not completely unrelated to the value

before the mutation as in the discrete encodings. Instead, the mutation is centered on the previous value and the **creep rate** determines how far the mutation may take the new value. In **arithmetic creep**, the old value g of the gene is changed to a new value g' according to

$$g \rightarrow g' = g - c + 2cr, \tag{2.17}$$

where $c \in [0, 1]$ is the creep rate and r is a random number in $[0, 1]$. In **geometric greep**, the old value of the gene changes as

$$g \rightarrow g' = g(1 - c + 2cr), \tag{2.18}$$

Note that, in geometric creep, the variation in the value of the gene is proportional to the previous value. Thus, if g is small, the change in g will be small as well. In addition, it should be noted that geometric creep cannot change the *sign* of g. Thus, when using geometric creep, the encoding scheme should be the standard one for real-numbered encoding, in which genes take non-negative values (e.g. in $[0, 1]$).

Furthermore, in both arithmetic and geometric creep, it is possible to obtain new values g' outside the allowed range. In that case, g' is instead set to the limiting value.

Finally, it should be noted that creep mutations can be defined for binary representations as well. One procedure for doing so is to decode the genes representing a given variable, change the variable by a small amount, e.g. according to an equation similar to Eq. (2.17) or Eq. (2.18), and then to encode the new number back into the chromosome.

2.2.7 Elitism

Even though a fit individual has a large probability of being selected for reproduction, there is no guarantee that it will be selected. Furthermore, even if it is selected, it is probable that it will be destroyed during crossover. In order to make sure that the best individual is not lost, it is common to make one or a few exact copies of this individual and place them directly in the next generation, a procedure known as **elitism**. All the other new individuals are formed via the usual sequence of selection, crossover, and mutation.

2.2.8 A standard algorithm

As is obvious from the discussions above, there are many different ways of implementing GAs, and it is therefore convenient to define a standard algorithm against which other versions can be compared. The **standard algorithm** is here defined as the GA having the following properties:

- One population with N individuals (i.e. no subpopulations, see Handout 5).
- Binary encoding of the strings.
- Linear fitness ranking.
- Roulette wheel selection.
- One–point crossover.
- Fixed values of the crossover probability p_{c} and the mutation probability p_{mut}.
- Elitism: A copy of the best individual is transferred (unchanged) to the next generation.
- Generational replacement.

Chapter 3

Using evolutionary algorithms

In this chapter it will be shown how to implement an EA, namely a simple genetic algorithm (GA) for function optimization using Matlab.

The program code below is written in Matlab. While Matlab does have many advantages, foremost of which is its simplicity (from the user's point-of-view), it is, unfortunately, quite slow compared to implementations using e.g. C or Delphi. However, the main aim of this section is to illustrate some programming principles concerning GAs, rather than providing the fastest possible program. For that reason, the program below has also been written in such a way that it should be as easy as possible to read the code, and also as easy as possible to transfer the code to another high-level programming language such as C, Java, Pascal, or Ada. Thus, Matlab's vector handling capacities have not been used; instead loops have been written in the standard way, used by most high-level programming languages.

In addition, the text below is also intended as an introduction to Matlab M–file programming. Thus, it is *not* assumed that the reader is accustomed to using Matlab.

3.1 Implementation of a GA in Matlab

The Matlab code described in this section implements a simple genetic algorithm for function optimization, and will consist of a main program, contained in a Matlab M–file named `funcopt.m`, as well as several other functions contained in the files
`initpop.m`,
`decode_chromosome.m`,
`evaluate_individual.m`,
`tournament_select.m`,
`crossover.m`,

and
`mutate.m`.

3.1.1 Objective function

In function optimization, the aim is to find the maximum (or minimum) of a given **objective function**. When writing the program below, we will consider the function

$$f(x_1, x_2) = \frac{\mathrm{e}^{-x_1^2 - x_2^2} + \sqrt{5}\sin^2(x_2 x_1^2) + 2\cos^2(2x_1 + 3x_2)}{1 + x_1^2 + x_2^2}. \tag{3.1}$$

The goal will be to find the maximum of this function of two variables, in the range $x_1, x_2 \in [-3, 3]$. Later in the chapter, other functions will be considered as well.

3.1.2 Initialization

Let us first write a skeleton for the main file, containing the parameter definitions. First, create an empty directory named `SimpleEA`., either using the command `mkdir SimpleEA` (in Unix or Linux) or by making a new folder (in Windows). Start Matlab, by typing Matlab (in Unix or Linux) or by double-clicking the Matlab icon (in Windows). Start a text editor. Then, write an M-file containing the following code:

```
npop = 30;
ngenes = 20;
pcross = 0.8;
pmut = 0.05;
ptour = 0.75;
range = 5.0;

%population = initpop(npop,ngenes);
```

We have not yet written the function `initpop`, and can therefore not call it. The % sign indicates that a row is a comment. So far, the program only defines four parameters, namely the population size (`npop`), the number of genes in the chromosomes (`ngenes`), the crossover probability (`pcross`), the mutation probability (`pmut`), the range for the variables (`range`), and the tournament selection parameter (`ptour`). Now save the M-file in the new directory as `funcopt.m`. (for FUNCtion OPTimization). Test this simple program by typing

```
funcopt
```

If you now type e.g. `ngenes`, Matlab will respond with the value (20) of `ngenes`. Now, open the text editor again. Let us write the function `initpop`, which assigns random values to the genes. Type the following, and then save the file (as `initpop.m`)

```
function population = initpop(npop,ngenes);

for i = 1:npop
  for j = 1:ngenes
    s = rand;
    if (s < 0.5)
      population(i,j)=0;
    else
      population(i,j)=1;
    end
  end
end
```

Note, again, that Matlab prefers to work with vectors and matrices, and that its `for` loops are rather slow. Therefore, the `initpop` function would be much faster if it were defined as

```
function population = initpop(npop,ngenes);

population = fix(2.0*rand(npop,ngenes));
```

where the `fix` function rounds (downwards) to the nearest integer. However, as mentioned above, for clarity, we will most often use `for` loops in this introduction.

Now, remove the comment (%) in the `funcopt.m` file, save all files, and run the program by again typing

```
funcopt
```

There should be no output. If you type

```
population
```

the `npop` chromosomes will be listed. They should contain only 0's and 1's.

The next step is to evaluate the individuals of the population. Modify the `funcopt.m`. file to read

```
npop = 30;
ngenes = 20;
pcross = 0.8;
pmut = 0.05;
ptour = 0.75;
range = 3.0;

population = initpop(npop,ngenes);

for i = 1:npop
  x = decode_chromosome(population,i,range,ngenes);
  fitness(i) = evaluate_individual(x);
end
```

3.1.3 Decoding the chromosomes

The decoding procedure will read the chromosome, and produce the corresponding individual. Naturally, the exact nature of this procedure will vary from problem to problem. In the problem considered here, i.e. the maximization of the function given in Eq. (3.1), two variables are needed. In this case, the first half of the chromosome can be used for generating the first variable (x_1) and the second half for the second variable (x_2). Thus, the function `decode_chromosome` can be implemented as follows:

```
function x = decode_chromosome(population,i,range,ngenes);

nhalf = fix(ngenes/2);

x(1) = 0.0;
for j = 1:nhalf
  x(1) = x(1) + population(i,j)*2^(-j);
end
x(1) = -range + 2*range*x(1);
```

```
x(2) = 0.0;
for j = 1:nhalf
  x(2) = x(2) + population(i,j+nhalf)*2^(-j);
end
x(2) = -range + 2*range*x(2);
```

This function uses the genes in the first half of the chromosomes to obtain a value of `x(1)` in the range [0,1], and the remaining genes to obtain a value of `x(2)` in the same range. `x(1)` and `x(2)` are then rescaled to the interval [`-range`,`range`]. Save the file as `decode_chromosome.m`. Note that it is not absolutely necessary to pass the value of `ngenes` as input to the `decode_chromosome` function since `ngenes` also can be obtained using the `size` command (try typing `size(population)`).

3.1.4 Evaluation

The evaluation procedure should evaluate the individual and return a fitness value. In the case of function maximization this is very simple: the function value itself can be chosen as the fitness value. Thus, write the function `evaluate_individual.m` as follows:

```
function f = evaluate_individual(x);

f = (exp(-x(1)^2 - x(2)^2)+sqrt(5)*(sin(x(2)*x(1)*x(1))^2)+ ...
    2*(cos(2*x(1) + 3*x(2))^2))/(1 + x(1)^2 + x(2)^2);
```

The `...` characters indicate that the formula continues on the next row. Note: the most common error in the implementation of the genetic algorithm shown here is to make a mistake in typing in the function above. Check your writing carefully!

Now, save all open files, and test the program by typing `funcopt` in the Matlab command window. In its present state, the program will first randomly generate a population and then loop through the entire population once, generating and evaluating each individual. If you now type `fitness`, a list of 30 (=`npop`) fitness values should appear.

Selection and reproduction

The next step is to write the part of the code that selects new individuals. In order to achieve a monotonous increase in the fitness values, we should use elitism. Here we will make two copies of the best individual. Modify the file

`funcopt.m` to read:

```
npop = 30;
ngenes = 20;
pcross = 0.8;
pmut = 0.05;
ptour = 0.75;
range = 3.0;

population = initpop(npop,ngenes);

maxfitness = 0.0;
for i = 1:npop
  x = decode_chromosome(population,i,range,ngenes);
  fitness(i) = evaluate_individual(x);
  if (fitness(i) > maxfitness)
    maxfitness = fitness(i);
    best_individual = i;
  end
end

temp_pop = population;

temp_pop(1,:) = population(best_individual,:);
temp_pop(2,:) = population(best_individual,:);
```

The new variable `maxfitness` will contain the fitness of the best individual, whose population index value is contained in the variable `best_individual`. Note that Matlab contains a standard function `max` that returns the largest component of a vector. Thus, the lines

```
if (fitness(i) > maxfitness)
    maxfitness = fitness(i);
    best_individual = i;
  end
```

can be removed, if instead the line

```
[maxfitness,best_individual] = max(fitness);
```

is added *outside* the loop over `i`. The variable `temp_pop` is needed to store temporarily the new individuals. The two first new individuals are taken as exact copies of the best individuals in the first generation. The notation `temp_pop(1,:)` refers to *all* the elements of the first row in the matrix `temp_pop` etc.

For the selection, we will use tournament selection, with tournament size equal to 2. Again, open a new file and write the following:

```
function i = tournament_select(fitness,npop,ptour);

itmp1 = 1 + fix(rand*npop);
itmp2 = 1 + fix(rand*npop);

r = rand;

if (r < ptour)
  if (fitness(itmp1) > fitness(itmp2))
     i = itmp1;
  else
     i = itmp2;
  end
else
  if (fitness(itmp1) > fitness(itmp2))
     i = itmp2;
  else
     i = itmp1;
  end
end
```

Save the function as `tournament_select.m`. This function chooses the better of two randomly selected individuals with probability `ptour`. With probability 1-`ptour`, the worse individual is selected. Now, in a new file, write the function `crossover` as follows:

```
function new_individuals = crossover(population,i1,i2,ngenes);

cp = 1 + fix(rand*(ngenes-1));

for j = 1:ngenes
  if (j < cp)
     new_individuals(1,j) = population(i1,j);
     new_individuals(2,j) = population(i2,j);
  else
     new_individuals(1,j) = population(i2,j);
     new_individuals(2,j) = population(i1,j);
  end
end
```

This function defines a random crossover point, and makes two new temporary chromosomes using one–point crossover. Save this function as `crossover.m`. The next step is to define the mutation function, which is done as follows

```
function mutated_individual = mutate(individual,pmut,ngenes);

mutated_individual = individual;
for i = 1:ngenes
  r = rand;
  if (r < pmut)
    mutated_individual(i) = fix(2.0*rand);
  end
end
```

and save it in the file `mutate.m`. The expression `fix(2.0*rand)` produces an integer random number equal to 0 or 1, since `rand` produces random numbers in the *open* interval $[0, 1[$. Note also that the new value of the mutated gene is selected as either 0 or 1, with equal probability. Thus, a 0 may be mutated to a 0, and a 1 may be mutated to a 1, making the effective mutation rate only half of the assigned value (`pmut`). The function can easily be modified, however, to produce mutations with the rate `pmut`.

Now, we can add the functions for selection, crossover, and mutation to the main program `funcopt.m`, which then takes the form

```
npop = 30;
ngenes = 20;
pcross = 0.8;
pmut = 0.05;
ptour = 0.75;
range = 3.0;

population = initpop(npop,ngenes);

maxfitness = 0.0;
for i = 1:npop
  x = decode_chromosome(population,i,range,ngenes);
  fitness(i) = evaluate_individual(x);
  if (fitness(i) > maxfitness)
    maxfitness = fitness(i);
    best_individual = i;
```

```
  end
end

temp_pop = population;

temp_pop(1,:) = population(best_individual,:);
temp_pop(2,:) = population(best_individual,:);

for i = 3:2:npop
  i1 = tournament_select(fitness,npop,ptour);
  i2 = tournament_select(fitness,npop,ptour);
  r = rand;
  if (r < pcross)
    new_individuals = crossover(population,i1,i2,ngenes);
    temp_pop(i,:) = new_individuals(1,:);
    temp_pop(i+1,:) = new_individuals(2,:);
  else
    temp_pop(i,:) = population(i1,:);
    temp_pop(i+1,:) = population(i2,:);
  end
end

for i = 3:npop
  temp_individual = mutate(temp_pop(i,:),pmut,ngenes);
  temp_pop(i,:) = temp_individual;
end

population = temp_pop;
```

Note that the selection procedure is repeated (`npop`-1)/2 times. Crossover occurs with probability `pcross`. The new individuals are mutated one at a time. The first two individuals are supposed to be exact copies of the best individual in the previous generation, and are therefore not mutated at all. In the final step, the first generation is erased and replaced by the second generation.

3.1.5 The complete EA

So far, the code evaluates the first generation and produces its offspring making up the second generation. Normally, it is necessary to run many generations in order to achieve a good result. Therefore, we now add a loop over generations. First, add the parameter `maxgenerations` after the `range` parameter in `funcopt.m`, and set it to, say, 100. Then, immediately after the line `population = initpop(npop,ngenes);`, add the code

```
for gen = 1:maxgenerations
```

At the end of `funcopt.m`, add another `end` command, to close the loop. Also, indent the code between the two new lines to make it more aesthetically appealing.

In the part that determines the value of `maxfitness`, add also code to store the values of $x = x(1)$ and $y = x(2)$ for the best individual:

```
 if (fitness(i) > maxfitness)
    maxfitness = fitness(i);
    best_individual = i;
    xbest = x;
 end
```

Finally, immediately before the final `end` statement, add a line that prints the values of `maxfitness` and `xbest` for each generation:

```
maxfitness, xbest
```

Thus, the final `funcopt.m` will have the following structure:

```
npop = 30;
ngenes = 20;
pcross = 0.8;
pmut = 0.05;
ptour = 0.75;
range = 3.0;
maxgenerations=100;

population = initpop(npop,ngenes);

for gen = 1:maxgenerations
  maxfitness = 0.0;

  for i = 1:npop
    x = decode_chromosome(population,i,range,ngenes);
    fitness(i) = evaluate_individual(x);
    if (fitness(i) > maxfitness)
      maxfitness = fitness(i);
      best_individual = i;
      xbest = x;
    end
  end
```

```
  temp_pop = population;

  temp_pop(1,:) = population(best_individual,:);
  temp_pop(2,:) = population(best_individual,:);

  for i = 3:2:npop
    i1 = tournament_select(fitness,npop,ptour);
    i2 = tournament_select(fitness,npop,ptour);
    r = rand;
    if (r < pcross)
      new_individuals = crossover(population,i1,i2,ngenes);
      temp_pop(i,:) = new_individuals(1,:);
      temp_pop(i+1,:) = new_individuals(2,:);
    else
      temp_pop(i,:) = population(i1,:);
      temp_pop(i+1,:) = population(i2,:);
    end
  end

  for i = 3:npop
    temp_individual = mutate(temp_pop(i,:),pmut,ngenes);
    temp_pop(i,:) = temp_individual;
  end

  population = temp_pop;

  maxfitness,xbest

end
```

3.1.6 Running the program

The program is now complete. Run it by typing `funcopt` in Matlab. The maximum value is equal to 3, and is obtained for $x_1 = x_2 = 0$. Note that the program does not always reach the global maximum of the function. Try modifying the parameters (e.g. population size and number of generations), and investigate the impact on the results obtained from the program.

3.1.7 Refinements

Very often, it helps to get a graphical display of the progress of the GA. Matlab graphics is a large topic, and it will not be discussed in detail here. However, with only a few lines of code, it is possible to obtain a graph showing the progress of the GA: Directly after the line `maxgenerations = 100;`, before the

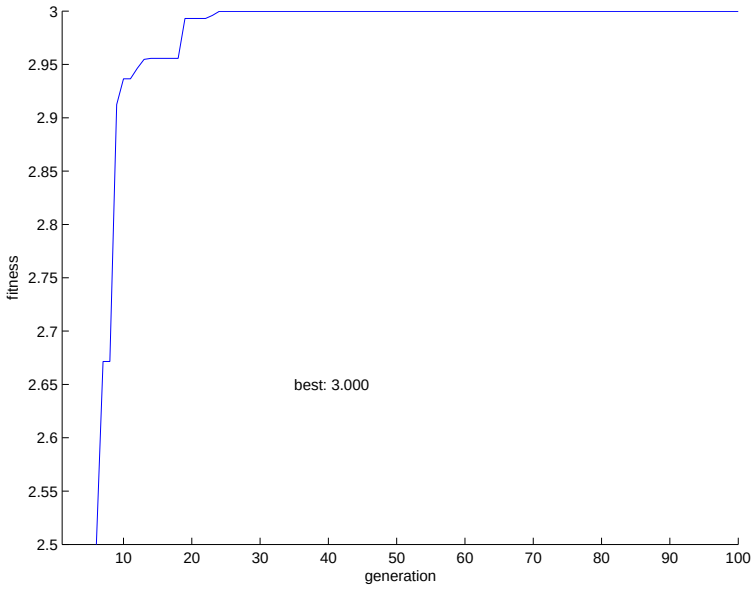

Figure 3.1: *Typical output from a run of the program. Note that the vertical axis ranges from 2.5 to 3.*

loop over generations (`gen`), add the following lines

```
hfig = figure;
hold on
set(hfig, 'Position',[50,50,500,300]);
set(hfig, 'DoubleBuffer','on');
axis([1 maxgenerations 2.5 3]);
hbestplot = plot(1:maxgenerations,zeros(1,maxgenerations));
htext = text(35,2.65,sprintf('best: %4.3f',0.0));
xlabel('generation');
ylabel('fitness');
hold off
drawnow;
```

Next, immediately after the loop over individuals (i.e. before the line `temp_pop = population;`), add the lines

```
plotvector = get(hbestplot,'YData');
plotvector(gen) = maxfitness;
set(hbestplot,'YData',plotvector);
set(htext,'String',sprintf('best: %4.3f',maxfitness));
drawnow;
```

Then save `funcopt.m`, and run the program again. A typical output from the program, using default value of the parameters (i.e. the values entered when

writing the program), is shown in Fig. 3.1. As can be seen, in this run, the GA solved the problem quite easily, reaching the maximum fitness after less than 30 generations. In other runs, the GA does not quite reach the global maximum of 3, but, in almost every run, it quickly finds a value within 0.1% or less of this value. The ability of a GA to find quickly a sufficiently good solution is one of the main reasons for its applicability and popularity in engineering problems, where time and budget constraints often make it a priority to find a good solution fast, even if that solution is not the best *possible* solution.

3.2 Function optimization

Here, the specific problem of optimization of mathematical functions will be considered. This is one of the most important applications of GAs, since almost any problem can be reduced to finding the minimum or maximum of some mathematical function. Often, the objective function is very complex, involving e.g. a combination of discrete and continuous variables. Here, the discussion will be limited to continuous, differentiable functions, even though there is nothing preventing the GA implemented here from being applied to functions with discontinuities.

In any application of a GA, a number of parameters must be set, and choices must be made concerning the encoding scheme, the fitness function, the selection procedure etc. It is easy to realize that an attempt to find the best possible GA for a given problem will be very time-consuming. Furthermore, such studies have shown that there is no unique, best GA for all problems. On the contrary, it is generally found that some experimentation is needed, for each new problem, in order to find good settings. However, luckily, it is also generally the case that the performance is not *very* sensitive to the exact choices made.

The problem of selecting the properties of a GA for a given problem is made even more complicated by the fact that, in some cases, the best results are obtained if some parameters (such as the mutation rate) are allowed to vary during a run.

Here, only a brief study will be made of the impact of variations in some GA properties.

3.2.1 A benchmark function

There are many functions that could be used for investigating the performance of a GA. Examples are the De Jong functions used in [11], and listed in Table 3.1. As is shown in the Table, De Jong considered not only deterministic functions, but stochastic functions (f_4) as well. Here, however, we will limit the study to deterministic functions. Clearly, the functions given in Table 3.1

Function	Range
$f_1(\mathbf{x}) = \sum_{i=1}^{3} x_i^2$	$[-5.12, 5.12]$
$f_2(\mathbf{x}) = 100\left(x_1^2 - x_2\right)^2 + (1 - x_1)^2$	$[-2.048, 2.048]$
$f_3(\mathbf{x}) = \sum_{i=1}^{5} \mathrm{int}(x_i)$	$[-5.12, 5.12]$
$f_4(\mathbf{x}) = \sum_{i=1}^{30} i x_i^4 + \mathrm{N}(0,1)$	$[-1.28, 1.28]$
$f_5(\mathbf{x}) = 0.002 + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2}(x_i - a_{ij})^6}$	$[-65.536, 65.536]$

Table 3.1: *The 5 functions used by De Jong [11]. For each function, it is the* minimum *that was sought, in the range listed in the second column. N(0,1) denotes Gaussian random numbers with mean zero and standard deviation one. The a_{ij} are constants.*

represent only a few choices from an infinitude of possible functions that could be used for performance testing. However, the choice of benchmark function is not arbitrary: For some functions it is, of course, easier to find the optimum than in others, and they will thus provide a less stringent test of the GA. As a general rule, a good benchmark function should contain many *local* optima, making it more difficult to find the *global* optimum. Furthermore, the local optima should preferably not be too evenly spaced, as some GA operators may exploit such even spacing. In addition, the benchmark function should *not* be separable, i.e it should not be possible to write $f(\mathbf{x})$ as $\sum f_i(x_i)$. Finally, the approach to the global optimum should not be smooth in any of the coordinate directions. In other words, following a coordinate axis towards the global optimum, one should encounter many local optima.

Keeping these guidelines in mind, we will here introduce a benchmark function ψ_n of n variables, defined as

$$\psi_n(x_1, x_2, ..., x_n) = \frac{1}{2} + \frac{1}{2n} \exp\left(-\alpha \sum_{i=1}^{n} x_i^2\right) \sum_{i=1}^{n} \cos\left(\beta \sqrt{i} x_i \sum_{j=1}^{i} j x_j\right), \qquad (3.2)$$

where α and β are two positive parameters. This function has a global maximum of 1 at $x_1 = x_2 = \ldots x_n = 0$. A plot of $\psi_2(x_1, x_2)$, with $\alpha = 0.05$ and $\beta = 25$ is shown in Fig. 3.2. The contour plot in the right panel of the figure clearly shows the uneven spacing of the local optima.

3.2.2 Experiments with the benchmark function

Using the benchmark function $\psi_n(\mathbf{x})$, the performance of a GA as a function of its parameter settings can be investigated. In view of the many parameters that can be varied, e.g. replacement scheme, crossover scheme, mutation rate, population size etc., it would be very time-consuming to cover all options, especially since GAs are stochastic so that several runs must be performed

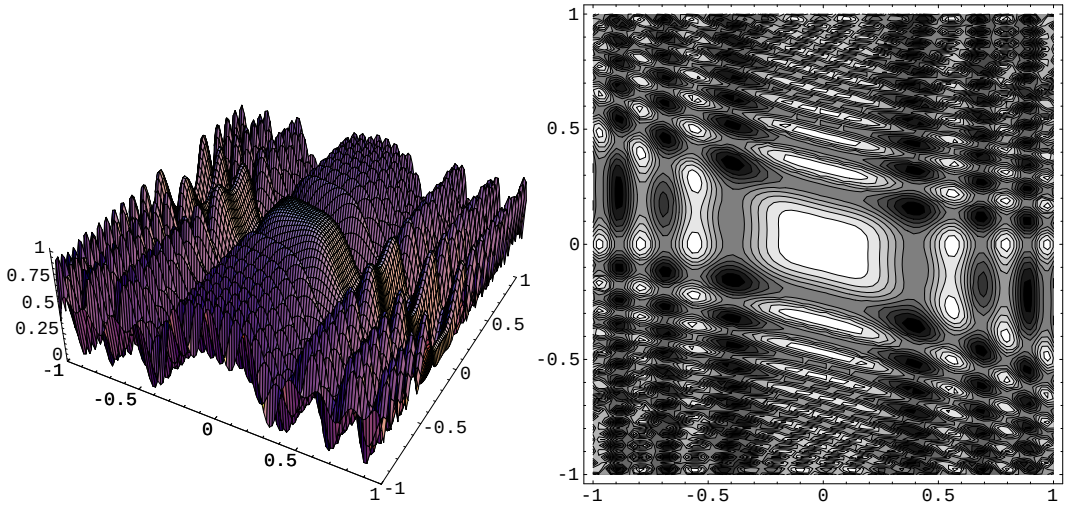

Figure 3.2: *The function $\psi_2(x_1, x_2)$.*

with each parameter setting in order to obtain statistically valid results. Thus, here only two representative cases will be studied.

The performance of different GAs can be compared in different ways, two of the most common being to run each GA until (1) a given number of individuals have been evaluated or (2) a given fitness value has been reached. The latter approach is often much more time-consuming since, for some parameter settings, a very large number of individuals may be needed to reach the given fitness cutoff. Thus, in the investigations presented below, the analyses will be based on the results of each GA after a fixed number of evaluated individuals.

Varying the population size

The population size is a very important parameter in any GA. If it is set too small, the population risks undergoing rapid inbreeding, after which only mutations can provide variation, a process that usually is very slow. On the other hand, if the population size is set too large, the evaluation of a generation will take a very long time, making the GA very slow.

When comparing runs with different population sizes, it is important to measure the performance as a function of the number of evaluated *individuals* rather than the number of evaluated *generations*. This is so, since a GA with large population size will sample more points in the search space (in a given generation), than a GA with small population size, thus biasing a comparison based on the number of evaluated generations in favor of the GA with large population size.

In Table 3.2, the performance of GAs with different population sizes is compared. In these runs, the objective function was $\psi_{10}(\mathbf{x})$ with $\alpha = 0.10$ and

Population size	Avg. best fitness @ 10^5 ind.	Avg. best fitness @ 10^6 ind.
30	0.95312 ± 0.02832	0.97224 ± 0.028194
100	0.97226 ± 0.02142	0.99706 ± 0.007562
300	0.99719 ± 0.00718	1.00000 ± 0.000000
1000	0.99997 ± 0.00001	1.00000 ± 0.000000

Table 3.2: *The average best fitness after 100,000 (second column) and 1,000,000 (third column) evaluated individuals for $\psi_{10}(\mathbf{x})$ is shown for various population sizes. In all cases, averages were taken over 10 runs.*

Mutation rate	Avg. best fitness @ 10^5 ind.	Avg. best fitness @ 10^6 ind.
0.003	0.92958 ± 0.02677	0.93310 ± 0.03389
0.010	0.95542 ± 0.02018	0.97666 ± 0.01693
0.030	1.00000 ± 0.00000	1.00000 ± 0.00000
0.100	0.89221 ± 0.01961	0.95635 ± 0.02226

Table 3.3: *The average best fitness after 100,000 (second column) and 1,000,000 (third column) evaluated individuals for $\psi_{10}(\mathbf{x})$ is shown for various mutation rates. In all cases, averages were taken over 10 runs.*

$\beta = 20$, and runs were terminated after a given number of evaluated individuals. A binary encoding scheme was used, with 25 genes per variable (i.e. a total of 250 genes per chromosome). Selection was performed using tournament selection with a tournament size of 5, and a probability of 0.75 of selecting the best individual. Generational replacement, with elitism, was used. The mutation rate was set to 0.02. Creep mutations were *not* used. The crossover probability was set to 0.80, and single-point crossover was used. The variable range was set to $[-1.0, 1.0]$.

Varying the mutation rate

In a GA, mutations provide new material for evolution to work with. As is the case with the population size, if the mutation rate is set too low, the result will be rapid inbreeding, usually resulting in failure. On the other, the mutation rate cannot be set *too* high, either, since mutations are random and therefore normally have a negative immediate effect (even though the long-term effect of a mutation can of course be positive). For example, if the mutation rate is set to 1, the result will be a completely random search.

In Table 3.3, the performance of GAs with different mutation rates is compared. Only full-range mutations were considered, i.e. no creep mutations were used. The population size was equal to 100, and all other parameters were set as in the investigation of population sizes above. Again, averages

were taken over 10 runs. Clearly, for this particular problem, a mutation rate of 0.03 was by far the best.

Investigations of the type above give important insight into the problem of parameter settings in GAs. However, one should be careful before drawing far-reaching conclusions: the best parameter settings for a GA are problem-dependent, and it therefore a good idea to perform an analysis of the kind shown above before starting a set of production runs.

Chapter 4

Properties of evolutionary algorithms

The operation of an EAs is very different from that of a random search: Mutations, which provide new material that the evolutionary process can work with, are random, but *selection* is not. But how do EAs really work? We will now try to give at least a partial answer to this question by considering a fundamental theorem known as the **schema theorem**. The importance of the schema theorem is sometimes exaggerated – all it really does is to give an *indication* of how the algorithms work. This, however, will be sufficient for our purposes. Next, the problem of premature convergence will be discussed, and, in the final section of this Handout, some analytical properties of a simple GA will be discussed. As in Handout 2, the discussion will be centered around GAs.

4.1 The schema theorem

Consider a GA in which binary encoding is used. A **schema** is defined as a pattern consisting of fixed positions, corresponding to genes taking the values 0 or 1, and wild–card positions, for which the genes take the value x, where x is an arbitrary value (i.e. either 0 or 1). Examples of schemata are 100xx1, xx0xx0x1, and 11x1. The first example, 100xx1, can be used to represent all of the strings 100001, 100011, 100101, and 100111.

With the introduction of the wild–card symbol x, each position in a schema can take either of three values; 0,1, or x. Thus, a chromosome of length n has a total of 3^n schemata. It is also easy to show that the number of schemata represented in a population with N individuals is between 2^n and $N2^n$.

Now, different schemata have different survival value. For instance, consider an example where the chromosome encodes the two numbers x_1 and x_2,

using three–digit encoding, and the task is to maximize the function $f(x_1, x_2) = \mathrm{e}^{x_1 x_2}$. With this encoding scheme the string 101011, say, will result in $x_1 = 2^1 + 2^{-3} = 0.625$ and $x_2 = 2^{-2} + 2^{-3} = 0.375$. In this example, the schema 11xxxx clearly is associated with higher fitness values than e.g. the schema 0000xx. The main idea behind the schema theorem is the realization that a GA will process schemata in such a way as to increase the number of schemata associated with high fitness values.

Let $\overline{f}(S)$ denote the average fitness of a schema S in a population, defined as the average fitness of those individuals whose chromosomes contain the schema in question. Assuming that the selection of individuals is done in proportion to their fitness, the probability of an individual with fitness f_i being selected, in a single selection step, is equal to f_i/f, where $f = \sum_{i=1}^{N} f_i$ is the total (summed) fitness in the population. Furthermore, let $\overline{f}$ denote the average fitness in the population, i.e. $\overline{f} = f/N$. The number of copies of S expected in generation $g+1$ is related to the number of copies of S present in generation g according to

$$\Gamma(S, g+1) = N\frac{\overline{f}(S)\Gamma(S, g)}{f}, \tag{4.1}$$

where $\Gamma(S, g)$ denotes the number of copies of S in generation g. Using the fact that $N/f = 1/\overline{f}$, we arrive at the equation

$$\Gamma(S, g+1) = \frac{\overline{f}(S)}{\overline{f}}\Gamma(S, g). \tag{4.2}$$

Clearly, if a schema is consistently associated with an above average fitness, i.e. if $\overline{f}(S)/\overline{f} = 1 + \alpha > 1$, where α is a constant, the number of copies of that schema present in the population will grow exponentially in time:

$$\Gamma(S, g+k) = \Gamma(S, g)(1+\alpha)^k. \tag{4.3}$$

In addition to selection, however, there are also the processes of crossover and mutation, which tend to destroy long schemata. In order to quantify this statement, let us introduce the **defining length** $D(S)$ of a schema S as the distance between the first and the last non–wild–card position in the schema. Thus, the schema $S_1 =$ 1x10x00xxx has defining length $D(S_1) = 7 - 1 = 6$, since the first non–wild–card gene in the string is at position 1, and the last such gene is at position 7. Let us also introduce the **order** $O(S)$ of a schema, defined as the number of fixed positions in the schema. With this definition, the schema $S_2 =$ 00x0110x is of order 6.

Consider now the crossover procedure. We will assume that single–point crossover is used, with random selection of the crossover point. The probability of destruction, during crossover, of a schema with defining length $D(S)$ is then given by

$$p_{\mathrm{d}} = \frac{D(S)}{n-1}. \tag{4.4}$$

Thus, a schema of defining length $n-1$, i.e. one that lacks wild–cards altogether, will (obviously) be destroyed with probability 1. Clearly, it is only the defining length that matters. Wild–cards in the very beginning or end of a string can be replaced with any symbol (0,1 or x) without destroying the schema in question. Since crossover occurs with probability p_{c}, the survival probability of schema S will be equal to $1 - p_{\mathrm{c}} D(S)/(n-1)$. In fact, the probability of survival is slightly larger than this estimate; even if a schema is destroyed, it may be reassembled if the other individual has exactly the same sequence up to and including the point where the cut is made. For example, the schema xx011x is destroyed if the cut occurs between the third and the fourth position. However, if the partial string xx0 (resulting from the cut) is joined with a string of the form 111, 110, or 11x, the resulting string will be xx0111, xx0110, or xx011x, respectively, and the schema will survive.

Finally, during mutation, a schema will be destroyed if any of its non–wild–card genes are changed. For each gene, the probability of mutation is equal to p_{mut}. Thus, the probability that the schema S will *not* mutate in any of the fixed positions equals $(1 - p_{\mathrm{mut}})^{O(S)}$.

Including the effects of crossover and mutation in Eq. (4.2), the expected number of copies of the schema S in generation $g+1$ will be

$$\begin{aligned}
\Gamma(S, g+1) &\geq \frac{\overline{f}(S)}{\overline{f}} \Gamma(S,g) \left(1 - p_{\mathrm{c}} \frac{D(S)}{n-1}\right) (1 - p_{\mathrm{mut}})^{O(S)} \approx \\
&\approx \frac{\overline{f}(S)}{\overline{f}} \Gamma(S,g) \left(1 - p_{\mathrm{c}} \frac{D(S)}{n-1}\right) (1 - O(S) p_{\mathrm{mut}}) \approx \\
&\approx \frac{\overline{f}(S)}{\overline{f}} \Gamma(S,g) \left(1 - p_{\mathrm{c}} \frac{D(S)}{n-1} - O(S) p_{\mathrm{mut}}\right), \qquad (4.5)
\end{aligned}$$

where, in the second step, p_{mut} has been assumed to be much smaller than 1 and, in the final step, quadratic terms have been neglected. Eq. (4.5) can be summarized by noting that the number of copies of a schema with low defining length, low order, and above average fitness will increase exponentially with time. Such schemata are referred to as **building blocks**, and the result obtained in Eq. (4.5) is known as the schema theorem.

Thus, we have shown that building blocks will become increasingly common in a population as more and more generations are evaluated. While this does not, in itself, explain fully how GAs work, it is a highly relevant result. However, a more complete theory should also prove that building blocks will tend to join other building blocks at a rapid rate, to form chromosomes of high fitness.

The hypothesis that building blocks *are* important and that a GA operates by manipulating such elements is known as the **building block hypothesis**. There is no general proof of this hypothesis, but it has nevertheless stood up to many difficult *empirical* tests. There are also some theoretical indications that

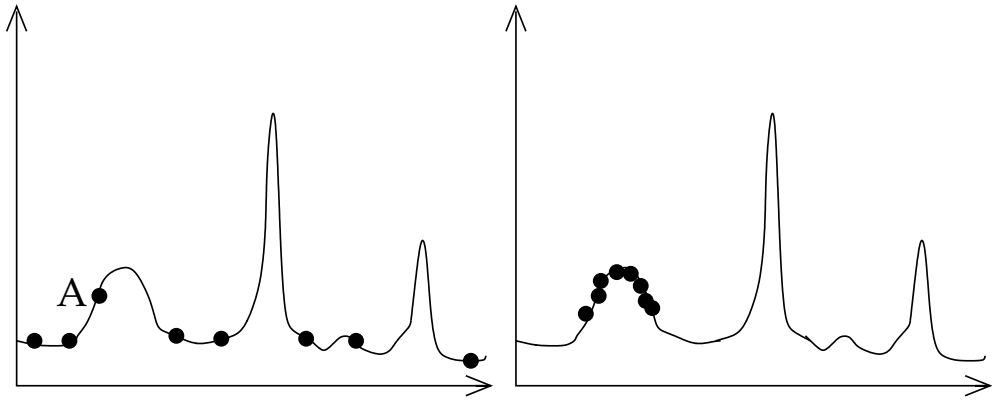

Figure 4.1: *Premature convergence. The vertical axis measures the fitness, and the individuals are shown as black dots along the horizontal direction. In the left panel, one individual (A) is slightly better than the others. In the right panel, the population has converged on a suboptimal solution, close to the original location of individual A.*

schema processing is efficient. For example, it can be shown that the number of schemata processed in a population of N individuals is proportional to N^3. Thus, even though each generation contain only N chromosomes, a much larger number of schemata are considered. This result was originally obtained by Holland [10], who referred to it as **implicit parallelism**.

4.2 Premature convergence

The most common problem encountered when using EAs is **premature convergence**, in which the population converges toward a suboptimal result. This phenomenon can be compared with inbreeding in nature, which produces less fit individuals, at least in the long run.

Problems for which there exists a single, global optimum can be solved using a deterministic gradient-descent method. EAs, by contrast, are normally applied to problems for which the **fitness landscape** has a very complex structure with many local optima. The occurrence of premature convergence can be understood by considering a rugged fitness landscape as shown in Fig. 4.1. Initially, the individuals are all very far from any optimum. However, some individuals are less bad than others, and the individual marked with the letter A in the left panel of the figure happens to be situated fairly close to a local optimum. Since its fitness exceeds that of the other individuals, it will produce many offspring. Some time will pass before it dominates the population, and during that time a lucky mutation may place some individual close to another (and better) local optimum, or even the global optimum. However, if the better optima are rather narrow, as in the figure, this is unlikely to occur. Instead, the entire population will gather around the local optimum, and the result is

premature convergence. Any subsequent change of the population will be due solely to mutations – crossover will only produce individuals close to the local optimum. In fact, in order to escape from the local optimum, an individual would have to undergo a so called **macro mutation**, consisting of a number of fortuitous changes occurring simultaneously. Clearly, such a mutation is very unlikely, indicating that the search has more or less come to an end after convergence has taken place.

How can premature convergence be avoided? To some extent, the problem can be mitigated by the inclusion of fitness ranking, which tends to reduce the difference between, say, the best individual and the second best individual. In Fig. 4.1 the ratio between the fitness of the best individual and the fitness of the second best individual (in the left panel) is, roughly, a factor 2. If there are N individuals, (linear) fitness ranking would reduce this ratio to $N/(N-1) \to 1$ for large N. On the other hand, fitness ranking is not always sufficient.

Another method for reducing the probability of premature convergence is the introduction of a number of **subpopulations**. In this case, the N individuals of the population are divided into N_{g} groups of individuals, each with $N_{\mathrm{s}} = N/N_{\mathrm{g}}$ individuals. Here, crossover and mutation occur only *within* the subpopulations. However, with probability p_{t} (the **tunneling probability**), a newly formed individual is transferred to another subpopulation. In order to maintain a constant number of individuals in the subpopulations, a randomly selected individual is also transferred in the other direction. If the tunneling probability is equal to 0, the subpopulation–based EA corresponds simply to N_{g} independent EAs with N_{s} individuals each. If $p_{\mathrm{t}} > 0$, however, individuals are occasionally moved between the groups, and the injection of new genetic material acts to prevent premature convergence.

Finally, the risk of premature convergence can also be decreased by introducing a varying mutation rate. After convergence, the difference between the chromosomes will be rather small. In other words, the diversity measure will be low. Thus, if the diversity measure falls below a certain value, the mutation rate can be increased. Similarly, the mutation rate can be decreased if the diversity measure exceeds a given threshold.

4.3 Analytical properties of evolutionary algorithms

In most realistic cases, it is very difficult to study the properties of EAs from an analytical point of view. This is so since, for example, the result of e.g. selection involves a weighted sum over the entire population (in the case of roulette-wheel selection). This sum must be available in closed form for an analytical treatment to be viable, and this is rarely the case. Crossover and mutations are also, in general, difficult to treat analytically for an arbitrary fitness function.

Vose [17] and others have studied the analytical properties of simplified GAs. In these studies, crossover and mutation were modelled by a single operator M. The introduction of this operator simplifies the analysis, but, at the same time, makes the simplified GA somewhat different from a standard GA. In order to obtain exact results, the population size was assumed to be infinite, which will also be the case for the analysis below, where a different simplified GA, without crossover and with slightly simplified mutations, will be considered. As the purpose of the analysis below is to give an introduction to the analytical properties of EAs, the analysis will mainly be based on the simplest possible problem, namely the **counting-ones problem**, . In the counting-ones problem, a binary encoding is used, i.e. the chromosomes are strings of zeros and ones. Hereafter, a gene will be called a **zero-gene** or a **one-gene**, depending on its value. In the counting-ones problem the fitness of an individual is simply the number of one-genes in the corresponding chromosome. Because of the simplicity of the fitness function, the decoding step is trivial and therefore no distinction will here be made between an individual and its chromosome.

Thus, in a case with 10 genes, the chromosome 1100100001 would receive a fitness score of 4. Note that, through the introduction of the fitness function used in this example, the search problem has been reduced to a one-dimensional problem, since the location of the genes in the chromosome is of no consequence in this case.

Furthermore, a case with an infinite population will be considered, for which the relevant quantity is not the number of individuals with a certain fitness value, but instead the *probability* of finding such an individual.

Now, let n denote the length of the chromosomes and assume that, in the initial population, the chromosomes are random with equal probabilities for the two alleles 0 and 1. Furthermore, let $p_1(k)$ denote the probability that a chromosome in the initial generation contains exactly k one-genes. Since the two alleles are chosen with equal probability, all possible chromosomes are equally probable. However, it is only the number of one-genes in a chromosome that matters, and the number of ways in which a chromosome with k one-genes can be generated varies with k. Consider first a chromosome with 0 one-genes. Clearly, there is only one way to generate such a chromosome. Next, consider a chromosome with a single one-gene. The position of the 1 can be chosen in n different ways, and the number of such chromosomes is thus equal to n. It is easy to show that, in general, a chromosome with k one-genes can be generated in π_k ways, where

$$\pi_k = \binom{n}{k}. \tag{4.6}$$

The total number of possible chromosomes equals 2^n, and therefore the prob-

Generation (s)	Average fitness analytical	numerical	$p_s(n)$ $(n = 100)$
1	50.0000	49.9892	7.89×10^{-31}
2	50.5000	50.5011	1.58×10^{-30}
3	50.9901	51.0035	3.12×10^{-30}
4	51.4708	51.5013	6.13×10^{-30}

Table 4.1: *Average fitness values for the counting-ones problem, with $n = 100$. For the numerical calculations, a population size of 100,000 was used. The rightmost column shows the (analytically computed) probability of finding, in generation s, an individual with maximum fitness, i.e. $p_s(n)$.*

ability distribution (in the initial population) $p_1(k)$ will be

$$p_1(k) = 2^{-n}\binom{n}{k}. \tag{4.7}$$

Using Eq. (A6) in appendix A, the average fitness of the population (which obviously equals $n/2$) can be computed as

$$\overline{f}_1 = \sum_{k=0}^{n} k p_1(k) = 2^{-n} \sum_{k=0}^{n} k\binom{n}{k} = \frac{n}{2}. \tag{4.8}$$

Selection First, the selection step will be considered by itself, i.e. neglecting crossover and mutation. The aim is to show how selection acts on a population to increase both the average fitness and the probability of reaching the optimum.

In roulette-wheel selection, the probability of selection of an individual is proportional to its fitness, i.e. to k (in this case). Thus, the probability that an individual in the second generation contains k ones is

$$\begin{aligned} p_2(k) &= \frac{k p_1(k)}{\sum_{k=0}^{n} k p_1(k)} = \frac{k\binom{n}{k}}{\sum_{k=0}^{n} k\binom{n}{k}} = \\ &= 2^{1-n}\frac{k}{n}\binom{n}{k}. \end{aligned} \tag{4.9}$$

Using Eq. (A7) in appendix A, the average fitness in the second generation is obtained as

$$\overline{f}_2 = \sum_{k=0}^{n} k p_2(k) = \frac{2^{1-n}}{n} \sum_{k=0}^{n} k^2\binom{n}{k} = \frac{n+1}{2}. \tag{4.10}$$

Thus, the average fitness value increases by $1/2$ after the first selection step.

Continuing the procedure in an iterative fashion, the distributions $p_3(k)$, $p_4(k)$ etc. can be obtained, as well as their respective averages. Unfortunately, while $p_s(k)$ easily can be computed from $p_{s-1}(k)$, using the general equation

$$p_s(k) = \frac{f(k)p_{s-1}(k)}{\sum_{k=0}^{n} f(k)p_{s-1}(k)}, \tag{4.11}$$

there is no simple, closed-form expression for $p_s(k)$ in general (even though approximations can be found), even for this simple problem. However, the distributions and their averages can of course be obtained numerically. Fig. 4.2 shows the distribution for the counting-ones problem in the first, second, third, fourth, fifth, 10^{th}, 15^{th}, and 20^{th} generation of a genetic algorithm with selection only. The number of genes (n) was equal to 100, and the population size was 100,000. Note that, in a numerical simulation, it is of course impossible to use an infinite population as in the analytical calculations above. Thus, at some point, the effects of the finite population size will become noticeable (usually causing the population to converge on a suboptimal solution, see below). This effect is evident in the right panels of Fig. 4.2. However, in the first generations, the effects of inbreeding are negligible, as indicated in Table 4.1, which shows average fitness values obtained from analytical calculations and from the numerical simulation.

As the genetic algorithm progresses from one generation to the next, not only the average fitness increases, but also the probability of finding individuals with maximum fitness (n). In the first generation, this probability equals $p_1(n) = 2^{-n}$, and in the second generation it reaches $p_2(n) = 2^{-n+1}$, i.e. it increases by a factor 2. In Table 4.1, $p_s(n)$ is shown for the first four generations.

Mutations In addition to fitness-proportional selection, genetic algorithms also use operators that modify the chromosomes in the population. Usually, two such operators are used, namely crossover and mutation. Crossover is complicated to treat analytically, and will therefore not be considered here. However, it should be noted that the complexity of an analytical treatment of crossover is the *only* reason that it is left out from this analysis. In general, crossover is an essential part of efficient genetic algorithms, and it should normally not be omitted in *applications* of such algorithms.

Using only selection and mutation, the steps involved in going from generation $s-1$ to generation s will be

$$p_{s-1}(k) \rightarrow_{\text{selection}} \hat{p}_s(k) \rightarrow_{\text{mutation}} p_s(k), \tag{4.12}$$

where $\hat{p}_s(k)$ denotes the distribution after selection.

The mutation procedure must also be somewhat simplified in order for it to be analytically tractable. Normally, the mutation procedure operates such

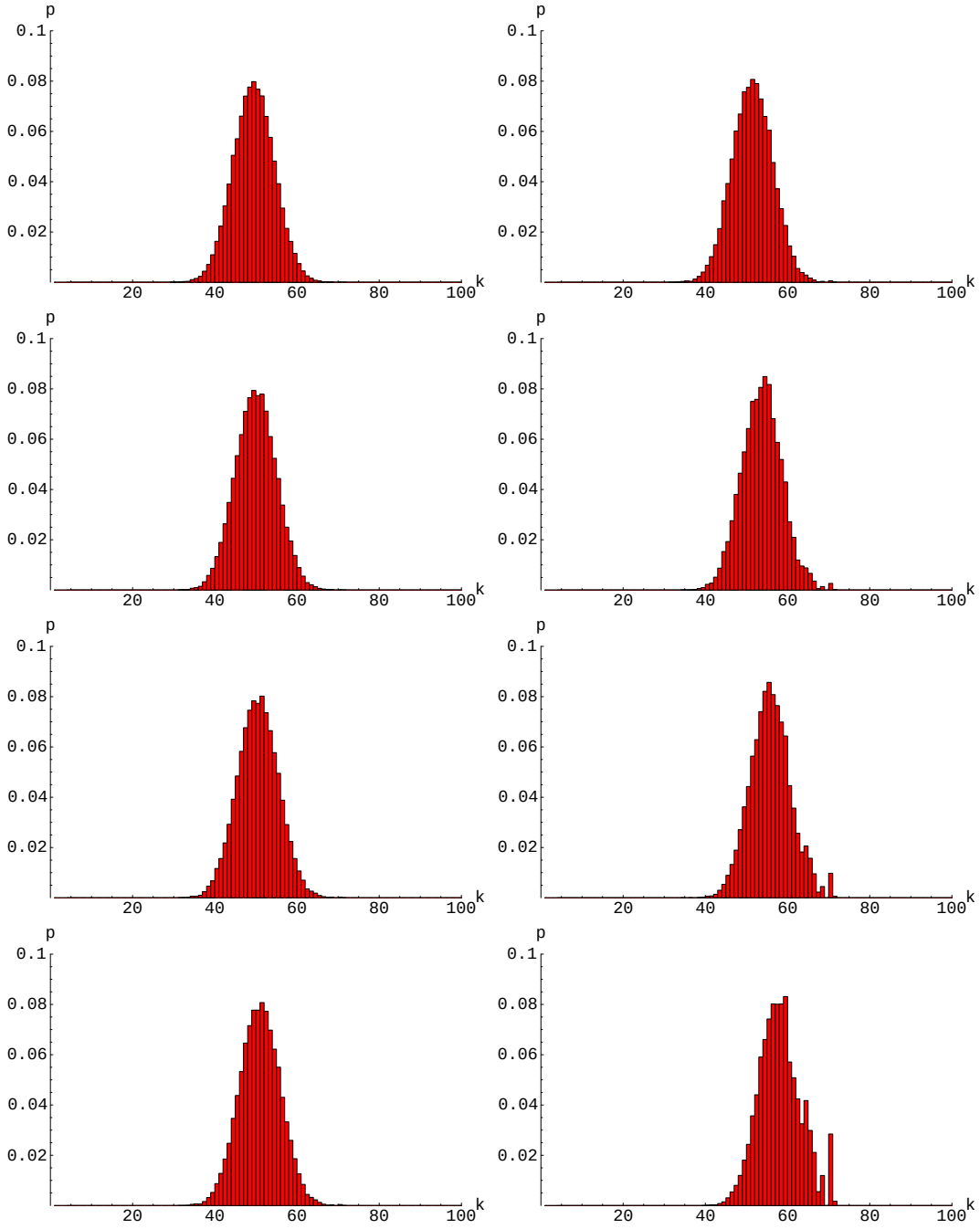

Figure 4.2: *Histograms of the distribution of fitness values for the counting-ones problem, shown at different generations. From top to bottom, the left panels show the distribution of the number of one-genes, k, in generations 1, 2, 3, and 4, whereas the right panels show the distributions in generations 5, 10, 15, and 20. The histograms were obtained from a genetic algorithm using (roulette-wheel) selection only, and a population size of 100,000. The number of genes in the chromosome was equal to* 100*. The effects of the finite population size begin to be noticed at around 10 generations.*

that it considers each gene separately, and performs a mutation with probability p_{mut}. Thus, in a chromosome with n genes, the number of mutated genes can range from 0 to n, with an average of np_{mut}. It turns out, however, that this mutation scheme is also difficult to treat analytically. Therefore, a simpler mutation scheme, in which at most one gene per chromosome is mutated, will be considered here. While this scheme rules out multiple simultaneous mutations, it is not too far off the mark, since the mutation rate is often chosen as (approximately) $1/p_{\mathrm{mut}}$ so that, on average, one mutation occurs per chromosome.

Thus, consider a mutation procedure in which a single gene is mutated with probability p_{m}. If p_{m} is set to 1, exactly one mutation will occur. If the mutated gene is a zero-gene, it will be changed to a one-gene, and vice versa.

Now, for any given k, the contribution to $p_s(k)$ will come from three sources: chromosomes with k one-genes that do not mutate, chromosomes with $k-1$ one-genes in which a zero-gene is mutated, and, finally, chromosomes with $k+1$ one-genes in which a one-gene is mutated. Hence

$$p_s(k) = (1-p_{\mathrm{m}})\hat{p}_s(k) + p_{\mathrm{m}}\left(\frac{n-k+1}{n}\hat{p}_s(k-1) + \frac{k+1}{n}\hat{p}_s(k+1)\right). \quad (4.13)$$

Returning to the first generation of the counting-ones problem, with the distribution after selection (now denoted $\hat{p}_2(k)$ rather than $p_2(k)$) given by Eq. (4.9), the distribution after mutation becomes

$$\begin{aligned} p_2(k) &= (1-p_{\mathrm{m}})2^{1-n}\frac{k}{n}\binom{n}{k} + \\ &+ p_{\mathrm{m}}2^{1-n}\left[\frac{(n-k+1)(k-1)}{n^2}\binom{n}{k-1} + \frac{(k+1)^2}{n^2}\binom{n}{k+1}\right] = \\ &= (1-p_{\mathrm{m}})2^{1-n}\frac{k}{n}\binom{n}{k} + p_{\mathrm{m}}2^{1-n}\frac{nk+n-2k}{n^2}\binom{n}{k} = \\ &= 2^{1-n}\left(\frac{k}{n} + p_{\mathrm{m}}\frac{n-2k}{n^2}\right)\binom{n}{k}, \end{aligned} \quad (4.14)$$

where, in the second step, the two binomial identities (see Appendix A)

$$\binom{n}{k-1} = \frac{k}{n-k+1}\binom{n}{k}, \quad (4.15)$$

and

$$\binom{n}{k+1} = \frac{n-k}{k+1}\binom{n}{k} \quad (4.16)$$

have been used.

From Eq. (4.14) it is clear that, at least in the situation considered here, mutations have a negative effect on individuals with above-average fitness, for

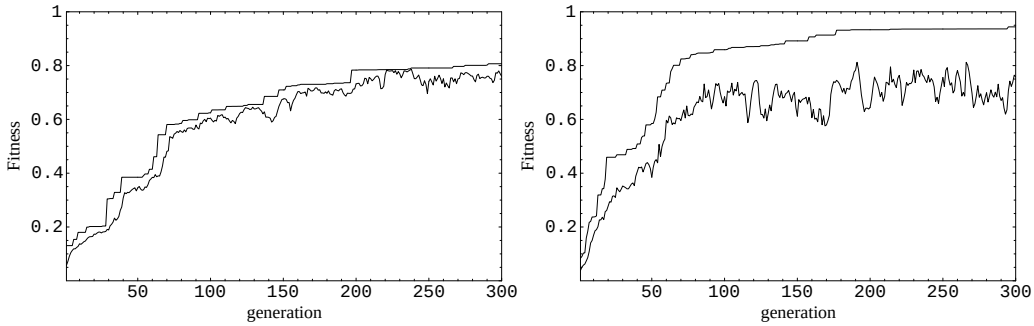

Figure 4.3: *Typical results from a GA applied to the benchmark function ϕ_{10} defined in Eq. (4.17). The population size was equal to 30, and the chromosome contained 100 genes (i.e. 10 genes per variable x_i). Crossover was not used. The maximum attainable fitness is equal to 1. For the curves in the left panel, the mutation rate was 0.001, and for the right panel it was 0.01. The upper curve in each panel shows the maximum fitness, and the lower curve shows the average. Note the convergence to suboptimal fitness values in the left panel.*

which the term $n - 2k$ is negative. In fact, it is commonly so that the immediate effect of a mutation is negative, particularly for fit individuals. This is not surprising, as a random change to a fine-tuned system (whether it is a chromosome or something else) rarely produces a good result. However, mutations are important since they provide new material for selection (and crossover) to work with.

In the analytical case considered above, where the population size is infinite, all possible chromosomes will be represented in the initial population, and selection would therefore be sufficient to increase the probability $p(n)$ of finding the best individual. In realistic applications, however, population sizes are usually far from infinite. The most common situation is one in which the evaluation of an individual is very time-consuming, implying that rather small population sizes must be used. If mutations are *not* used in a genetic algorithm with a small population, the result is rapid inbreeding and premature convergence. Fig. 4.3 shows the result of using two different mutation rates, $p_{\text{low}} = 0.001$ and $p_{\text{high}} = 0.01$, in a genetic algorithm with a population size of 30 applied to the benchmark function

$$\phi_N(x_1, x_2, ..., x_N) = \exp\,(-\sum_{i=1}^{N} x_i^2)\frac{1 + \frac{1}{N}\sum_{i=1}^{N}\cos(20\sqrt{i}x_i)}{2}, \tag{4.17}$$

with $N = 10$. This function has a global maximum of 1 at $x_1 = x_2 = \ldots x_N = 0$. As can be seen from Fig. 4.3, the run with the low mutation rate (left panel) rapidly converges to a suboptimal fitness value, whereas the run with the higher mutation rate maintains diversity in the population throughout the run, as evidenced by the large difference between the average and maximum fitness values. Elitism was used, i.e. an unchanged copy of the best individual

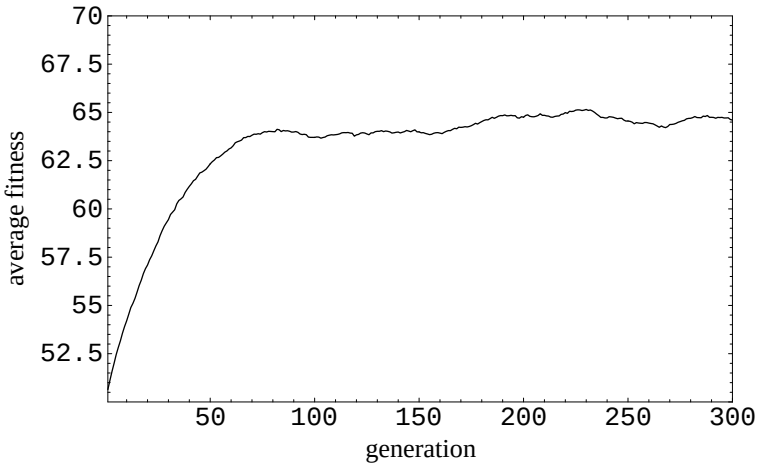

Figure 4.4: *The average fitness as a function of generation for the counting-ones problem. The curve was obtained for a run with population size 10,000 and chromosome length $n = 100$. The parameter p_{m} was set to 1, i.e. exactly one mutation was performed per chromosome.*

was transferred to each new generation.

Equilibrium point for the average fitness In Fig. 4.4, the average fitness is shown for a genetic algorithm applied to the counting-ones problem. As can be seen from the figure, after around 100 generations, the average fitness reaches a plateau where it remains for the rest of the run, albeit with small oscillations. The fact that the average fitness stabilizes at some value far from the optimum is a common occurrence regardless of the fitness function used (see also the right panel of Fig. 4.3). In the early generations, the (immediate) negative effects of mutations are much smaller than the positive effects of selection, but in later generations, the two effects are approximately of the same order of magnitude, causing the population average to reach its plateau.

In fact, for the counting-ones problem, it can be shown that the approximate level of the plateau is given by

$$\overline{f} - \overline{f}^2 - \frac{2}{n}\overline{f^2} + \overline{f^2} = 0, \tag{4.18}$$

where $\overline{f}$ denotes the average fitness in the population, and $\overline{f^2}$ the average value of the fitness squared. Assuming that the fitness distribution keeps its original binomial shape, i.e. takes the form

$$p(k) = 2^{-n}\binom{n}{k - \overline{f} + \frac{n}{2}}, \tag{4.19}$$

the equilibrium point for $n = 100$ is roughly 68, rather close to the observed values, in Fig. 4.4, of around 63-65.

4.3.1 Multi-dimensional fitness functions

Even though the number of genes can take any positive integer value in the counting-ones problem considered above, the problem is one-dimensional, since the fitness function only depends on the sum of the values of all genes.

Clearly, the general case, in which the fitness function cannot be reduced from its most general form, i.e.

$$f = f(g_1, g_2, \ldots, g_n), \quad g_i \in \{0, 1\}, \ i = 1, \ldots, n \tag{4.20}$$

is more complicated but also more interesting. In this case, the effects of e.g. mutations are often quite complex to deal with analytically since, even if only single mutations are considered, the fitness function may change in n different ways rather than just 2 ways as was the case for the counting-ones problem (see Eq. 4.13).

However, the computations can, of course, be performed numerically, and thus provide useful insights into the operation of the GA for various parameter settings. Assuming that binary encoding is used, and that there are n genes in the chromosome, there will be a total of 2^n chromosomes of the form

$$G_k = g_1 g_2 \ldots g_n, \tag{4.21}$$

where k enumerates the chromosomes (e.g. such that $k = 1$ for $00\ldots00$, $k = 2$ for $00\ldots01$ etc. up to $k = 2^n$ for $11\ldots11$).

In order to obtain exact solutions, in the form of probability distributions for the chromosomes G_k, it must be assumed that the population size is infinite but other than that, no approximations need be made. The computations are quite time-consuming though.

For example, consider mutations occuring with a probability of p_{mut}. In the general case, anything from 0 to n mutations *may* occur, even though the expected number of mutations is around np_{mut}. During mutation, the probability distributions for any given chromosome G will consist of contributions from G itself (to account for cases in which no mutations occur, an event that takes place with probability $(1-p)^n$), contributions for chromosomes G' that differ from G in only gene *and* for which a single mutation occurs in the gene that differs, contributions from chromosomes G'' that differ from G in two genes, and for which both those genes mutate etc.

Thus, the computation time will rise rapidly with n and becomes impractical for n larger than around 10. However, for small n, an exact computation of the kind just outlined is feasible, even though it will not be considered further here.

Chapter 5

Advanced topics

In the previous chapters, the basic properties of GAs have been introduced. In this chapter, a few advanced topics will be covered, starting with a description of encoding schemes, selection operators, and fitness measures, and contiuning with a discussion of some different types of advanced GAs. It should be noted that there is a very large number of variations on the theme of EAs. Thus, the description below is intended as an illustration of a few advanced topics related to EAs, and is by no means exhaustive.

5.1 Representations

5.1.1 Gray coding of binary-valued chromosomes

In the traditional, standard GA, a binary representation scheme is used in the chromosomes. While simple to implement, such a scheme may have some disadvantages, one of them being that a small change in the decoded value obtained from a chromosome may require flipping many bits which, in turn, is an unlikely event. Thus, the algorithm may get stuck simply as a result of the encoding scheme. Consider, as an example, a ten-bit binary encoding scheme, and assume that the best possible chromosome is 1000000000. Now, consider a case in which the population has converged to 0111111111. In order to reach the best chromosome from this starting position, the algorithm would need to mutate *all* ten genes in the chromosome!

An alternative representation scheme, which avoids this problem is the **Gray code**, which was patented in 1953 by Frank Gray at Bell Laboratories, but which had been used already in telegraphs in the 1870s. a Gray code is simply a binary representation of all the integers k, in the range $[0, 2^n]$, such that, in going from k to $k + 1$, only *one* bit changes in the representation. Thus, a Gray code representation of the numbers $0, 1, 2, 3$ is given by $00, 01, 11, 10$.

Other 2-bit Gray code representations exist as well, e.g. 10, 11, 01, 00 or 00, 10, 11, 01. However, these representations differ from the original code only in that the binary numbers have been permuted or inverted. An interesting question is whether the Gray code is unique, if permutations and inversions are disregarded. The answer turns out to be negative for $n > 3$. Gray codes can be generated in various ways.

5.1.2 Messy encoding schemes

In the schema theorem (see Handout 4), the concept of building blocks were introduced, as schemata with low defining length, low order, and above average fitness, and it was noted that schemata in which the non-wild-card positions are far from each other on the chromosome suffer a larger probability of being destroyed than schemata in which this is not the case. Thus, for example, the schema S_1 =1xxxxx01 is more likely to be destroyed (during crossover), than the schema S_2 =101xxxxx. Now, if S_1 is associated with high fitness, it is unfortunate that it is so easily destroyed. Note, however, that the likely destruction of S_1 is a simply result of the encoding scheme: if the genes were instead placed on the chromosome in such a way that the first gene in S_1 was placed *last* in the chromosome, the resulting schema S_1' = xxxxx011 would be equivalent to S_1 but would be much less likely to be destroyed.

Messy encoding schemes generate a less position-sensitive representation of a chromosome, by associating each gene with a number determining its position in the chromosome. Thus, the string

$$c_1 = ((1,0),(5,1),(3,0),(4,1),(2,1)), \tag{5.1}$$

represents a messy encoding of 01011, if the first number in each pair is interpreted as the position of the corresponding gene, whose allele is given by the second number in the pair. Thus, if the schema S =0xxx1 is associated with high fitness, it has a much greater chance of surviving in the messy representation (in which genes 1 and 5 are adjacent) than in the ordinary representation.

There are also some problems associated with messy encoding schemes, however. For example, it may happen that a string contains several copies of a given gene position, e.g.

$$c = ((1,0),(3,1),(3,0),(2,1),(4,0),\ldots). \tag{5.2}$$

One way of resolving such conflicts is simply to use the first occurence of a given gene position, and discard all other occurences. For the chromosome shown in Eq.(5.2) this procedure would give the chromosome $1110\ldots$, in ordinary binary encoding.

The opposite problem may also occur, in which some gene positions are not represented on the chromosome at all.

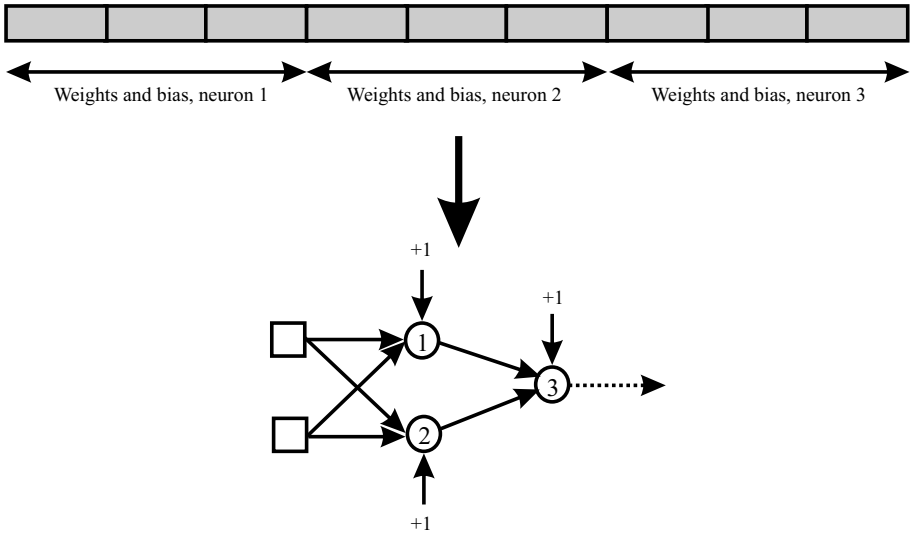

Figure 5.1: *Encoding of a simple three-neuron ANN, using a chromosome with nine real-valued genes. Each weight and bias, shown as solid arrows in the ANN, is obtained from a gene in the chromosome.*

However, further details of messy encoding schemes are beyond the scope of this text. The interested reader is referred to [6].

5.1.3 Variable length encoding schemes

In the standard GA, all chromosomes are of the same, fixed size, which is a suitable state of affairs for many problems. For example, in the optimization of a function, with a known number of variables, it is easy to specify a chromosome length. It is not entirely trivial, though: the number of genes per variable must be set sufficiently high to give a representation with adequate accuracy for the variables. However, if the desired accuracy is difficult to determine, a safe approach is simply to set the number of genes per variable to a high value (50, say), and then run the GA with chromosomes of length 50 times the number of variables. Thus, there is no need to introduce chromosomes with varying length. However, in many other situations, it *is* desirable, or even essential, to use chromosomes with varying length. Indeed, during biological evolution, many different genome sizes have resulted (in different species, both current and extinct ones). Variations in genome length may result from accidents during the formation of new chromosomes, such as duplication of a gene or parts thereof (see Sect. 5.4.1 below). Clearly, in nature, there can be no given, optimal and non-changing genome size. The same applies to artificial evolution of complex structures, such as e.g. **artificial neural networks** (ANNs, see Appendix B), **finite-state machines** (FSMs) or **artificial brains** for autonomous robots (which may take many different forms). In general, vary-

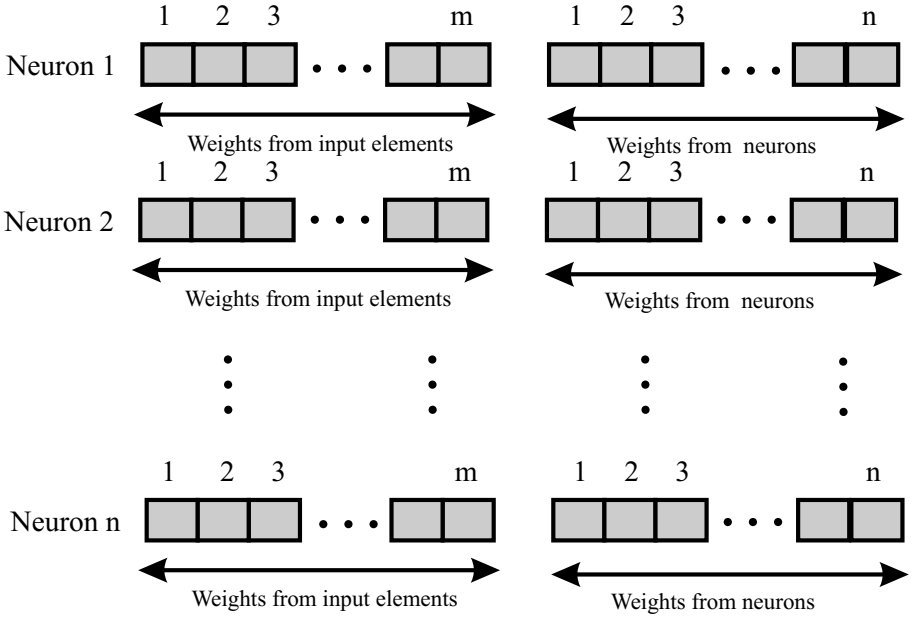

Figure 5.2: *An example of a representation for an ANN. The network has m input elements, and n neurons. Each neuron can, in principle, be connected to all other neurons, provided that all weights are non-zero.*

ing genome size should be introduced in cases where there is insufficient *a priori* knowledge of the optimal size and structure of the systems being optimized. Here, one such case will be studied in detail, namely ANNs.

Encoding schemes for artificial neural networks

In the standard GA, the structure to be optimized is encoded in a chromosome of fixed length.

In the case of FFNNs (see Appendix B for the definition of network types) with given size, the procedure is straightforward: If real-number encoding is used, each gene can represent a network weight, and the decoding procedure can easily be written so that it associates the weights with the correct neuron. An example is shown in Fig. 5.1. Here, a simple FFNN with three neurons, two in the hidden layer and one in the output layer, is encoded in a chromosome containing 9 genes, shown as elongated boxes. If instead binary encoding were to be used, each box would represent several genes which, when decoded, would yield the weight value.

In more complex applications, however, the encoding of information in a linear chromosome is often an unnecessary complication, and the EA can instead be made to operate directly on the structures that are to be optimized. For such implementations, object-oriented programming is very useful. Here, a type (i.e. a data structure) representing a neural network can be defined as a

list of neurons, each of which is equipped with a list of incoming connections from input elements, as well as a list of incoming connections from neurons. Both the latter two lists, and the list of neurons, can then be allowed to vary in size. The encoding scheme is illustrated in Fig. 5.2.

Using EAs in artificial neural networks

When generating an ANN (see Appendix B), one of the first decisions that must be made concerns the training algorithm. One possible choice is to use an EA as the training algorithm. The generation of ANNs using evolutionary algorithms has been considered extensively in the literature, see e.g. [14] and [19] for reviews.

However, there also exists many specialized algorithms for training networks, such as the backpropagation algorithm, which is applicable to FFNNs. As a general rule, whenever there exists a specialized algorithm, it is often more efficient to use that algorithm rather than using an EA. In fact, some researchers use such statements to argue against EAs (”Algorithm A is faster than an EA on problem B..” etc.). However, EAs derive their strength from their generality. For example, there exists specialized algorithms for e.g. function optimization using FFNNs and for solving the travelling salesperson problem (TSP, see below). However, these algorithms can only solve the problem they were designed to solve. EAs, by contrast, can, with little modification, solve both the two problems mentioned, and a vast array of other problems as well. Furthermore, even in cases where there exists a specialized algorithm, an EA may be useful in parts of the optimization process. For example, a common problem when using backpropagation concerns the selection of the number of neurons in hidden layers (see Appendix B). Simply put, there exists no clear method for selecting an optimal number of hidden neurons. Usually, rules-of-thumb or trial-and-error is used instead. A common rule-of-thumb for setting the number N^{H} of neurons in the hidden layer of an FFNN is

$$N^{\mathrm{H}} = \sqrt{N^{\mathrm{I}} N^{\mathrm{O}}}, \tag{5.3}$$

where N^{I} and N^{O} denote the number of input elements and the number of output neurons, respectively. However, an alternative procedure is to use an EA to set the number of hidden neurons (and possibly also the parameters used in backpropagation). Thus, in this case, the (very simple) chromosome would contain genes representing the number of hidden neurons and the training parameters, and the decoding procedure would generate an FFNN as prescibed by the chromosome, and with (initially) random connections. Next, the evaluation of the individual would consist of training the FFNN using backpropatation with the parameters obtained from the chromosome. The fitness measure can be taken e.g. as the number of backpropagation training steps need to

achieve a given result, e.g. an FFNN representing some given function to a certain accuracy.

Perhaps more importantly, there exists many problems for which few algorithms other than EAs are suitable. Examples of such problems are the construction of RNNs for time series prediction and the generation of RNNs to be used as artificial brains in autonomous robots. In both cases, it is very difficult to specify the optimal number of neurons in advance. The case of artificial brains for autonomous robots is particularly difficult since the reward (if any) for a given action generally occurs long after the action was taken. By contrast, for backpropagation training of FFNNs, it is required that the reward is immediate: for any input signal, it must be possible to judge directly the quality of the output by means of an error function. For this reason, backpropagation belongs to a family of training algorithms known as **supervised training algorithms**. Incidentally, it can be mentioned that the error function must be differentiable for backpropagation to function properly. No such restrictions are required in the case of ANN optimization by means of EAs.

Returning to the case of the autonomous robot, there exists a **credit assignment problem**: the robot must, somehow, be able to make connections between actions and rewards that occur much later. In the most extreme case, there is only a single scalar feedback signal at the end of the evalution. In such cases, supervised training algorithms are not applicable, and EAs are often used instead[1]. When using an EA to optimize a structure (such as an RNN) that may vary in size during evolution, clearly a varying chromosome size must also be allowed, which poses some problems (not least programming problems). In particular, there must be defined several mutation operators, which can modify not only the weights, but also the structure of the networks. A set of seven mutation operators (M1-M7) for RNNs is shown in Fig. 5.3. M1 and M2 modify the strength of connections between units (see Appendix B) in the network (i.e. between neurons or from input elements to neurons) that are already present, whereas M3-M7 modify the structure of the network: M3 adds a connection between two randomly chosen units, and M4 removes an already present connection. M5 removes an entire neuron, and all its incoming and outgoing weights. M6 and M7 add neurons. In the case of M6, the neuron is added without any incoming or outgoing weights. Thus, two mutations of type M3 are needed in order for the neuron to have an effect on the computation performed by the network. M7, by contrast, adds a neuron with a direct connection from an input element to one of the output neurons (shown as filled circles in the figure). Note that many other neuron addition operators can be defined.

In addition, crossover operators can be defined in order to combine chro-

[1]There are other algorithms, such as **reinforcement learning** [1] which are able to assign credit to past events. However, such algorithms will not be considered in this course.

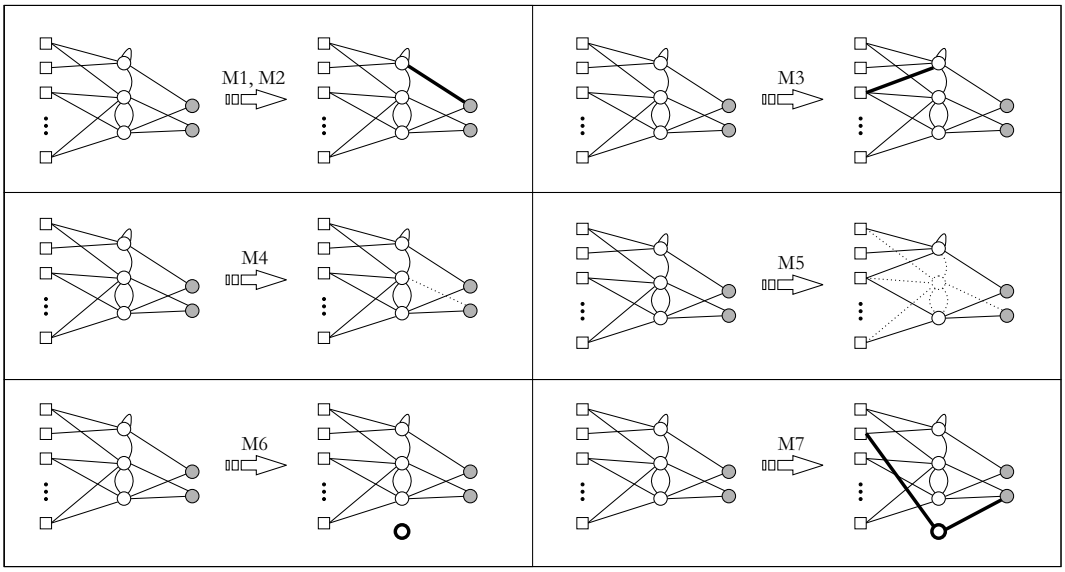

Figure 5.3: *Mutation operators (M1-M7). Modifications and additions are shown as bold lines and removed items are shown as dotted lines. The mutations are (M1-M2):* weight mutations*, either by a random value or a value centered on the previous value; (M3-M4):* connectivity mutations*, addition of an incoming weight with random origin (M3), or removal of an incoming weight; (M5-M7):* neuron mutations*, removal of a neuron and all of its associated connections (M5), insertion of an unconnected neuron (*zero-weight addition*) (M6), and addition of a neuron with a single incoming and a single outgoing connection (*single connection addition*) (M7). Figure courtesy of J. Pettersson.*

mosomes of varying size, unless the equivalent of species is introduced (see below), in which case only chromosomes of equal size are allowed in the crossover procedure. In general, due to the distributed nature of computation in neural networks, it is difficult to define a good crossover operator, even in cases where the networks are of equal size. This is so, since half an ANN (say) does not perform half of the computation of the complete ANN. More likely, any part of an ANN will not perform any useful computation at all. Thus, cutting two networks in pieces and joining the first piece of the first network with the second piece of the second network (and vice versa) often amounts to a huge **macro-mutation**, decreasing the fitness of the network, and thus generally eliminating it from the population. However, putting this difficulty aside for the moment, how should crossover be defined for neural network? One possibility is to encapsulate neurons, with their incoming connections into units, and only swap these units (using, e.g. uniform crossover) during crossover, rather than using a single crossover point at any location. This crossover procedure is illustrated in Fig. 5.4, for a case where the two networks are of equal size.

Clearly, with this procedure, crossover can be performed with any two net-

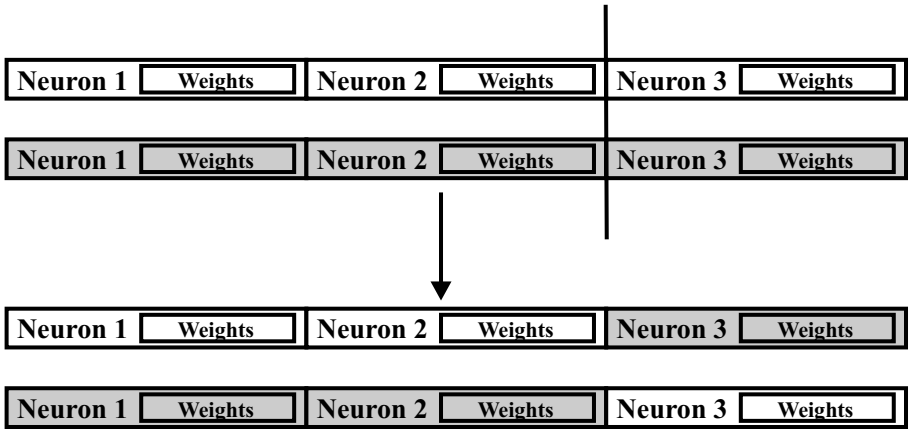

Figure 5.4: *A crossover procedure for neural networks. In this case, cuts are only made between neurons.*

works. However, there is a more subtle problem concerning the identity of the weights. If the list of incoming weights to the neurons represents neuron indices, crossover may completely disrupt the network. For example, consider a case where neuron 3 takes input from neurons 1, 4, and 5. If, during crossover, a single additional neuron is inserted between neurons 3 and 4, say, then the inserted neuron will be the new neuron 4, and the old neuron 4 will become neuron 5 etc., thus completely changing the numerical values of the weights, and also limiting the usefulness of crossover.

The are biologically inspired methods for mitigating this problem, however: Consider another type of network, namely the genetic regulatory networks introduced in Handout 1. Here, some genes (transcription factors) can regulate the expression of other genes, by producing (via mRNA) protein products that bind to a binding site close to the regulated genes. The procedure of binding is an ingenious one: instead of say, stating that e.g. ”the product of gene 45 binds to gene 32” (which would create problems like those discussed above, in the case of gene insertion or deletion), the binding procedure may say something like ”the product of gene g binds to any gene with a binding site containing the nucleotide sequence AATCGATAG”. In that case, if another gene, x say, is preceded (on the chromosome) by a binding site with the sequence AATCGATAG, the product of gene g will bind to gene x regardless of their relative position on the chromosome. Likewise, the connection can be broken if the sequence on the binding site preceding gene x is mutated to, say, ATTCGATCG. Encoding schemes using neuron labels instead of neuron indices *can* be implemented for the evolution of ANNs. However, such topics are beyond the scope of this text.

As mentioned above, crossover between networks often leads to lower fitness. However, there are crossover operators that modify networks more gently. One such operator is **averaging crossover**, which can be applied to net-

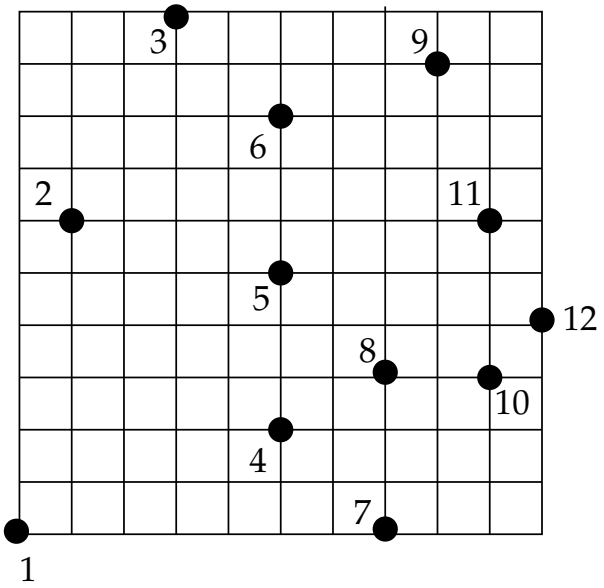

Figure 5.5: *Map for the travelling salesperson problem. The cities are numbered from 1 to 12, and their locations are given by two integer coordinates* (x, y).

works of equal size. Consider a network of a given size, using an encoding scheme as that illustrated in Fig. 5.1. In averaging crossover, the value of gene x in the two offspring, denoted x_1' and x_2' is given by

$$x_1' = \alpha x_1 + (1 - \alpha)x_2, \tag{5.4}$$

and

$$x_2' = (1 - \alpha)x_1 + \alpha x_2, \tag{5.5}$$

where x_1 and x_2 denote the values of x in the parents. α is a number in the range $[0, 1]$. In case α is equal to 0 or 1, no crossover occurs, but for all other values of α there will be a mixing of genetic material from both individuals in the offspring. If α is close to 0 (or 1), the mixing is very gentle.

This concludes our study of encoding schemes for ANNs. However, in Handout 7, examples of the use of such encoding schemes will be given, in the form of a discussion of a few different applications.

Encoding scheme for the travelling salesperson problem

The **travelling salesperson problem** (TSP), can be formulated as follows: given a set of N nodes, specified by their positions $\{x_i, y_i\},\ i = 1, \ldots, N$ (in the two-dimensional case), which should be visited (e.g. by a salesperson, hence the name) once *and only once*, the final step of the path being a return to the starting point, what is the shortest path? The problem is illustrated in Fig. 5.5, for the case $N = 12$.

The TSP has several important applications, such as e.g. the placement of components in electronic circuits, and, just as for FFNNs, there exists specialized algorithms for solving this problem. The TSP can also be solved using EAs [6], In this case, the chromosome generally contains a simple list enumerating the cities. In order for such a chromosome to generate a valid path, however, each city can only occur once in the chromosome. Thus, so-called **permutation encoding** is commonly used, in which the chromosome is simply a permutation (i.e. an ordering) of the numbers $1, \ldots, N$. Thus, for the problem illustrated in Fig. 5.5 a typical chromosome would be e.g $c_1 = \{4, 6, 9, 1, 3, 7, 12, 10, 8, 5, 2, 11\}$. Note that, in this type of encoding, the absolute position of the genes make no difference. Thus, for example, $c_2 = \{6, 9, 1, 3, 7, 12, 10, 8, 5, 2, 11, 4\}$ encodes the same path as c_1.

In TSP, operators for both mutation and crossover must be implemented with care, to avoid generating invalid paths. Mutation operators for TSP are usually based on the swapping of genes between different positions in the chromosome. Thus, for example, a TSP chromosome $\{3, 4, 2, 1, 5\}$ can mutate to $\{3, 1, 2, 4, 5\}$ by swapping the values for the second and fourth genes. Clearly, many different mutation operators can be defined.

Crossover is more tricky, since it must combine information from two individuals while, at the same time, generating two new, valid paths. Not surprisingly, several crossover operators have been defined for TSP. One of them, **order crossover** is defined in the problems (see the corresponding handout).

5.1.4 Grammatical encoding

The introduction of variable-length chromosomes, as discussed above, adds considerable flexibility to an EA, and is crucial in the solution of certain problems. One motivation for the introduction of variable-length chromosomes was the fact that such chromosomes have more similarity with chromosomes found in natural evolution. However, as was discussed in Handout 1, even the variable-length chromosomes differ considerably from biological chromosomes. A particularly important difference is the fact that biological chromosomes do not, in general, encode the parameters of a biological organism directly, whereas the chromosomes used in the EAs *do* use such **direct encoding**, i.e. an encoding scheme such that all parameters are obtained directly (possibly with re-scaling) from the chromosome.

An example should be given to illustrate the state of affairs in biological systems. Consider the human brain. This very complex computer contain on the order of 10^{11} computational elements (neurons), and around $10^{14} - 10^{15}$ connections (weights) between neurons, i.e. around 1,000 - 10,000 connections per neuron. Now, if every connection were to be encoded in the chromosome the information content of the chromosome would have to be around 10^5 Gb,

Figure 5.6: *Grammatical encoding, as implemented by Kitano [13].*

even if the strength (and sign) of each connection weight were to be encoded using a single byte. However, the actual size of the human genome is around 3 Gb. Furthermore, the chromosome does many other things than just specifying the structure of the brain. Thus, it is evident that rather than encoding the brain down to the smallest detail, the chromosome encodes the *procedure* by which the brain is formed.

In EAs, encoding schemes that encode a procedure for generating e.g. a neural network, rather than the network itself, are known as *grammatical encoding* schemes [13], [7]. In such methods, the chromosome can be seen as a sentence expressed using a grammar. When the sentence is read, i.e. when the chromosome is decoded, the individual is generated, using the grammar. An early example of grammatical encoding is the method developed by Kitano [13] for encoding FFNNs.

In Kitano's method, each chromosome is encoded in a string of the form shown in Fig. 5.6. The method was applied to the specific case of FFNNs containing, at most, 8 neurons. The S in the chromosome is a start symbol, which, when read, generates a matrix of the following 4 symbols: (ABCD, in the case shown in the figure)

$$S \rightarrow \begin{pmatrix} \mathrm{A} & \mathrm{B} \\ \mathrm{C} & \mathrm{D} \end{pmatrix}. \tag{5.6}$$

The rule generating the matrix is, of course, quite arbitrary. For example, the elements could have been placed in a different order in the matrix. S and the symbols A, B, C, and D are **non-terminals**, i.e. symbols that will themselves be read and will then generate some other structure, which may contain both non-terminals or **terminals**, i.e. symbols which are not processed further. Each capital-letter symbol (Kitano used all letters A-Z) encodes a specific matrix of lower-case letters, taken from an alphabet (a-p) of 16 symbols that encode all 16 possible 2×2 matrices. Thus, rules for decoding, say, the matrices A and B are taken from the chromosome. For the chromosome shown in Fig. 5.6 above, the results would be

$$A \rightarrow \begin{pmatrix} \mathrm{a} & \mathrm{f} \\ \mathrm{b} & \mathrm{a} \end{pmatrix}, \tag{5.7}$$

and

$$B \rightarrow \begin{pmatrix} \mathrm{c} & \mathrm{a} \\ \mathrm{d} & \mathrm{a} \end{pmatrix}, \tag{5.8}$$

etc. Note that the total number of matrices of the kind shown in the right-hand sides of Eqs. (5.7) and (5.8) equals $16^4 = 2^{16} = 65,536$. The result of decoding

the matrices in Eqs. (5.7) and (5.8) are given by

$$a \rightarrow \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \; b \rightarrow \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, \; \ldots \; p \rightarrow \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}. \tag{5.9}$$

Thus, the first step generates a 2×2 matrix of capital letters, the second step a 4×4 matrix of lowercase letters, and the final step an 8×8 matrix of 0s and 1s. This method is then used for generating a simple FFNN. The diagonal elements are used for determining the presence (1) or absence (0) of a neuron, and the off-diagonal elements (w_{ij}) in the 8×8 matrix are used for determining the connection weights from neuron i to neuron j. Since Kitano's aim was to generate FFNNs, recurrent connections were simply ignored, as were connections to non-existent neurons (i.e. those neurons encoded by rows with a 0 as diagonal element).

In the encoding scheme showed in Fig. 5.6, it is possible that the same capital-letter symbol may appear in several rules. In this case, only the *first* (e.g. leftmost on the chromosome) rule is used in the decoding scheme implemented by Kitano.

Several other grammatical encoding schemes have been introduced in the literature (see e.g. [7]), but those schemes are beyond the scope of this text.

5.2 Selection

The selection of individuals in an EA can be performed in many different ways. So far, in this course, tournament selection and roulette-wheel selection have been considered. Here, two additional selection methods will be introduced briefly, namely Boltzmann selection and competitive selection.

5.2.1 Boltzmann selection

The **Boltzmann selection** scheme introduces concepts from physics into the mechanisms of EAs. In this selection scheme, the notion of a temperature T is introduced in the EA, and the basic idea behind the selection scheme is to use T as a tunable parameter that determines the extent to which good individuals are preferred over bad individuals during selection. The mechanism derives its name from the fact that the equations (see below) for Boltzmann selection are similar to the **Boltzmann distribution** which, among other things, can be used for determining the distribution of particle speeds in a gas. In addition, due to the presence of the temperature parameter T, Boltzmann selection is related to heating and cooling (annealing) processes in physics. Boltzmann selection can be implemented in various different ways [2]. In one version, the selection of an individual from a randomly selected pair of individuals is

based on the function b given by

$$b(f_{j_1}, f_{j_2}) = \frac{1}{1 + \mathrm{e}^{\frac{1}{T}(\frac{1}{f_{j_1}} - \frac{1}{f_{j_2}})}}, \tag{5.10}$$

where f_{j_1} and f_{j_2} are the fitness values of the two individuals in the pair. During selection, a random number r is generated and the selected individual j is determined according to

$$j = \begin{cases} j_1 & \text{if } b(f_{j_1}, f_{j_2}) > r \\ j_2 & \text{otherwise} \end{cases} \tag{5.11}$$

If T is low, and $f_{j_1} > f_{j_2}$, individual j_1 will be selected with a probability approaching 1 as T tends to zero. On the other hand, if T is large, the selection procedure tends to select j_1 and j_2 with almost equal probability, regardless of the difference in fitness. Normally, in runs with EAs using Boltzmann selection, T is initially set to a high value, allowing the EA to sample the search space as much as possible. T is then gradually reduced, making the EA home in on the better solutions found in the early stages of the run.

An alternative Boltzmann selection scheme selects individual j with probability p_j given by

$$p_j(f_j) = \frac{\mathrm{e}^{\frac{f_j}{T}}}{\sum_{k=1}^{N} \mathrm{e}^{\frac{f_k}{T}}}, \tag{5.12}$$

where f_k denotes the fitness of individual k and N is the number of individuals in the population. As in the other Boltzmann selection scheme presented above, individuals are selected with approximately equal probability if T is large, whereas for small T, individuals with high fitness are more likely to be selected.

5.2.2 Competitive selection and co-evolution

In all the selection schemes presented so far, the basis for the selection has been a user-defined fitness function, which has always been specified before the start of an EA run. However, this way of specifying a fitness function is very different from the notion of fitness in biological systems, where there is no such thing as an absolute fitness measure. Instead, whether an individual is fit or not depends not only on itself, but also on other individuals, both those of the same species and those of other species. These ideas have been exploited in EAs as well. In **competitive selection** schemes, the fitness of an individual is measured relative to that of other individuals. Such selection schemes are often used in connection with **co-evolution**, i.e. the simultaneous evolution of two (or more) species. In nature, co-evolution is a frequently occurring phenomenon. For example, predators may grow sharper fangs as a result of

thicker skin in their prey (and the prey, in turn, will then grow even thicker skin etc.). In EAs, co-evolution is often implemented by considering two populations, where the fitness of the members of the first population is obtained from interactions with the members of the second population, and vice versa. A specific example are the sorting networks evolved by Hillis [9]. In this application, the goal was to find **sorting networks** of order n, i.e. networks that can sort any permutation of the number $1, 2, \ldots, n$. In his experiments, Hillis used one population of sorting networks, which was evolved against a population of sequences to be sorted. The fitness of the sorting networks was measured by the ability to sort test sequences, and the fitness of the test sequences was measured by their ability to fool the sorting networks, i.e. to make them sort incorrectly.

A problem with co-evolution is the issue of measuring *absolute* improvements, given that the fitness measure is a relative one. This problem can be attacked in various ways. A simple procedure (used e.g. by Wahde and Nordahl [18] in their work on pursuit and evasion in artificial creatures) is to measure the performance of the best individual (a pursuer, say, in the application considered in [18]) against a given, fixed individual. However, such an individual cannot easily be defined in all applications.

Co-evolution is an interesting (and biologically motivated) idea, but it is not applicable to all problems. In addition, the problems involved in measuring **co-evolutionary progress** (as discussed above) have made the use of co-evolution in EAs quite rare.

5.3 Fitness measures

In many problems, e.g. function optimization, it is easy to specify such a function. However, in other problems it may be more difficult. An example of such a case is one in which the fitness function should take into account several, possibly conflicting, criteria (**multiobjective optimization**). Another example is **constrained optimization** where not all possible solutions are valid.

The selection of the fitness measure has a great impact on the performance of an EA. For example, consider the problem of finding the maximum of the simple function

$$f(x) = 1000 + \sin x. \tag{5.13}$$

It would be possible, clearly, to select $f(x)$ directly as the fitness measure. However, it would be very unwise to do so, as most individuals would obtain a fitness value of around 1000, thus giving the EA very little to work with. On the other hand, taking the fitness as, say, $f(x) - 999$ would give a non-negative fitness measure which would strongly favor good solutions.

Another example can be taken from the field of evolutionary robotics. Consider the problem of evolving a **gait** (i.e. a means of walking) for a simple

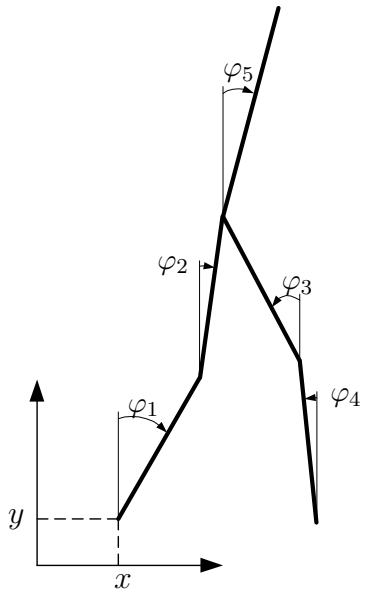

Figure 5.7: *Configuration of a five-link bipedal walking robot.*

model of a bipedal robot, namely the five-link robot shown in Fig. 5.7, using, say, a neural network as the brain of the robot. An obvious choice of fitness measure for this problem is simply the distance walked in a given amount of time, i.e. the position of the center-of-mass (COM) of the robot, at the end of an evaluation. However, with this fitness measure it is usually found that the robot will throw itself forward, thus terminating the evalution, but at least reaching further than robots that simply collapse. Here, the problem stems from the fact that the step from a random neural network to one that can generate the cyclical pattern needed for legged locomotion is a very large one. Indeed, evolving a gait (for a pre-defined body shape) from random initial conditions is a very complex task, and the EA therefore takes the easier route of just throwing the body of the robot forward. Of course, if the EA *did* find an actual gait, however bad, that could keep the robot upright while walking slowly forward, the corresponding individual would obtain a higher fitness value. However, such an individual is unlikely to be present in the early generations of the EA. In this particular case, the problem can be solved by combining several criteria. For example, the fitness measure can be taken as a combination of the position of the *feet* and the posture (e.g. the vertical position of the COM) at the end of the evaluation.

There exists a vast literature concerning multi-objective optimization and constrained optimization, and here only a brief introduction will be given. The interested reader is referred to [2] and [20].

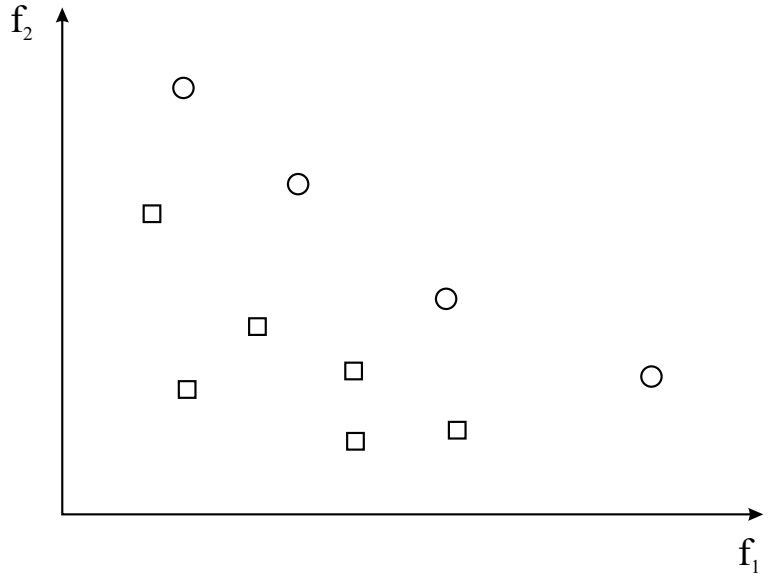

Figure 5.8: *Pareto optimality. The circles represent the set of non-dominated points, i.e. the pareto-optimal set.*

5.3.1 Multi-objective optimization

It is a common occurrence that there are several criteria involved in the evaluation of an individual. For example, if an EA is used for the optimization of, say, the engine of a car, three relevant criteria are performance, cost, and weight. These criteria are conflicting: the performance can perhaps be increased by selecting better components. However, doing so will increase the cost of the engine etc.

Multi-objective optimization can be approached in a variety of ways. One way is to use the notion of **pareto-optimality**.

Pareto-optimality

Consider a potential solution to a an optimization problem, given by the vector $\mathbf{x}$. For this vector, the fitness criteria f_i thus takes the values $f_i(\mathbf{x})\ \ i = 1, \ldots, K$, where K is the number of optimization criteria. $\mathbf{x}$ is said to be **pareto-optimal** if there exists *no* individual $\mathbf{y}$ such that $f_i(\mathbf{y}) \geq f_i(\mathbf{x})$ for all i, and $f_j(\mathbf{y}) > f_j(\mathbf{x})$ for at least one $j,\ \ j = 1, \ldots, K$. The set of such solutions form the **pareto-optimal set**, and the vectors belonging to this set are also referred to a **non-dominated**. In general, the pareto-optimal set consists of more than one element.

The notion of **pareto-optimality** is illustrated in Fig. 5.8. The points shown as circles form the pareto-optimal set, among the available points.

Multiple evaluations

In many problems in **evolutionary robotics**, it is common that each individual must be evaluated in a variety of situations, to prevent the EA from finding solutions that only perform well in certain specialized situations. Thus, in such cases, the robot will be evaluated in a variety of different situations, and each evaluation will result in a partial fitness value. Thus, when a robot has been completely evaluated, there will be a vector of partial fitness values, from which a scalar fitness value must be obtained. One way of doing so is to take the average of the partial fitness values as the fitness of the individual, i.e. to define the fitness f as

$$f = \sum_{i=1}^{N_\mathrm{e}} f_i, \tag{5.14}$$

where N_e is the number of evaluations, and f_i is the partial fitness obtained from evaluation i. However, taking the average does not always lead to good results. For example, in a navigation task involving autonomous robots, it was found in [16] that this fitness measure was quite slow in eliminating bad individuals. This was so, since an individual could compensate for bad performance in one evaluation by performing better (perhaps as a result of a fortuitous starting configuration) on other evaluations.

An alternative procedure is to use the *minimum* partial fitness value as the final fitness value of the individual, i.e.

$$f = \min_i f_i. \tag{5.15}$$

In the robotic navigation problem considered in [16], this fitness measure gave much better overall performance, since individuals that performed very badly on even a single evaluation would be given low fitness, and would thus be eliminated from the population.

Regardless of the weighting procedure used, however, it is important to give the EA some positive feedback even for bad solutions (which usually dominate the population in early generations). Thus, for example, if a robot is given the task of navigating from point A to point B in an environment with moving obstacles, the fitness for a given evaluation can be taken as e.g. the distance to B at the time of the first collision, or at the end of the evaluation time, whichever event occurs first. With this fitness measure, there will be a smooth road for the EA, from very bad solutions towards better solutions.

5.3.2 Constrained optimization

In **unconstrained optimization**, the variables of the problem are allowed to take any value among the possible values. By contrast, in **constrained optimization**, this is not the case. For example, the problem of finding the max-

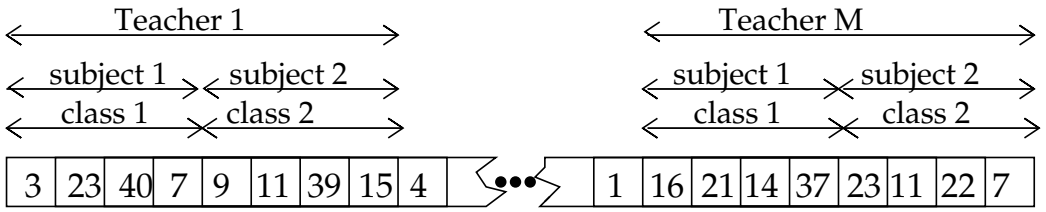

Figure 5.9: *A chromosome for a simple scheduling problem.*

imum of a function $f(x_1, x_2)$ over the set of real numbers is a case of unconstrained optimization. Adding a set of conditions $h_i(x_1, x_2) = 0,\ i = 1, \ldots, N_c$, where N_c is the number of conditions, turns the problem into a case of constrained optimization. Constraints can be of two kinds. **Soft constraints** are allowed to be broken, but there is a cost (i.e. in terms of lower fitness, if an EA is used), involved in breaking them, whereas **hard constraints** may not be broken: if a hard constraint is broken, the corresponding solution is considered not to be valid, and is either given a low fitness value or is never evaluated. In the latter case, there must be a screening of newly formed individuals, in order to remove those solutions that are not valid, much as in biological systems, where a developing embryo dies e.g. if its genome contains a mutation which would lead to disastrous results in the phenotype. In practice, the screening of individuals is implemented as an extra loop in the stage where new individuals are formed: if, after selection, crossover, and mutation, a newly formed individual is deemed not valid, it is simply discarded, and the selection step is repeated.

As an illustration of fitness assignment for constrained optimization, consider the problem of generating, by means of an EA, a schedule for the teachers at a school. The hard constraints in this case are that students and teachers can only be at one location at a time, and that a lecture room can only be used for one class at a time. Soft constraints may concern e.g. the interval between successive lectures in the same subject, to allow time for homework. Both teachers and students may also have preferences regarding suitable lecture times. There may be several conflicting constraints, and the fitness function must be chosen in such a way as to produce the best possible solution, taking into account as many as possible of the soft constraints (and all of the hard constraints).

A simple encoding scheme for school scheduling is shown in Fig. 5.9. The scheme simply encodes the lecture times for each teacher, numbered from 1 to 40, where 1 corresponds to the first lesson on Monday morning and 40 to the last lesson on Friday afternoon. The genes thus take integer values in the range $[1, 40]$.Obviously, the subjects taught by each teacher are known, and so need not be encoded in the chromosome. The students are assumed to belong, at all times, to a given fixed class.

Fig. 5.9 shows the (unrealistic) case in which each teacher has only two

subjects and two classes, four times a week. The length of the corresponding chromosome equals $8M$, where M is the number of teachers.

When the chromosomes are decoded, the schedule for each teacher is obtained immediately, and the schedule for each class can also easily be found. Schedules for the occupancy of the various lecture rooms are not included in the encoding scheme considered here.

The fitness measure can now be defined as

$$f = \alpha^{n_c}, \tag{5.16}$$

where n_c is the number of collisions or violations of hard constraints, and α is a constant in the range $]0, 1[$.

Since the chromosomes are of fixed length, crossover can easily be implemented. Mutations are also easy to implement, and can be made more useful by only mutating those genes that lead to constraint violations.

5.4 EAs with mating restrictions

In nature, there are restrictions on mating, the most obvious one being that individuals of one species generally do not mate with individuals of another species. However, mating restrictions exist even within species, as a result of, for example, geographical separation between individuals. In addition, to prevent inbreeding, biological organisms are designed to avoid mating with close relatives. The concept of mating restriction has been exploited in EAs as well, and some brief examples will now be given.

5.4.1 Species-based EAs

In biology, one introduces the notion of **species**, to classify animals (and plants). Members of the same animal species are simply those that can breed[2] and have offspring, or more correctly *fertile* offspring[3]. Speciation, i.e. the generation of new species commonly occurs in nature, often as a result of physical separation (**allopatry**) of groups of individuals belonging to the same species. After many generations, the descendants of those individuals may have evolved to become so different (genetically) that members from one group can no longer breed with members of the other group, should they meet. This is so, since evolution will fine-tune animals (i.e. their genomes) to prevailing conditions. In addition, random accidents such as mutations, occur during recombination of chromosomes and, if beneficial, such accidental changes may spread rapidly

[2]For species that reproduce asexually, this definition can obviously not be used.

[3]For example, a female tiger and a male lion (or vice versa) can have offspring (called a liger), but the offspring is sterile.

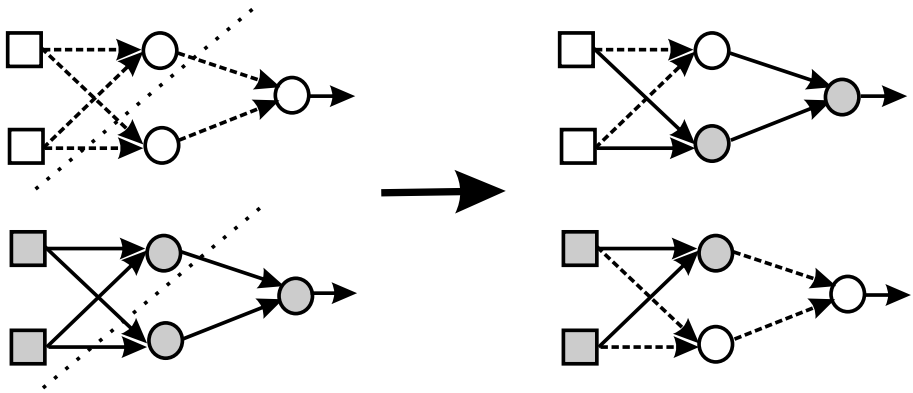

Figure 5.10: *One version of mating restriction in ANNs: only networks with equal structure are used in the crossover procedure*

in a population, thus making it genetically different from another population in which the same random change has not taken place. In addition to mutations, gene duplication may also take place: it is common to find several copies of the same gene in a genome. In fact, it is believed [4] that at least two complete *genome* duplications have occurred in early vertebrates. Evidence for this quadrupling of the genome can be obtained from studying a complex of genes known as *hox*[4]. The *hox* gene complexes are a set of transcription factors (see Handout 1) which are active during development of an animal, and which are responsible for determining the identity of different regions of a body, i.e. whether a part is to become a limb or something else. In many vertebrates (e.g. mammals), there are four *hox* gene complexes, suggesting that two genome duplications have occurred.

Some EAs also use the concept of species. In such EAs, individuals are only allowed to mate with individuals that share some properties, either on the genotype level or the phenotype level. One of the simplest speciation schemes is to allow crossover only between individuals for which the **Hamming distance** of the chromosomes, defined as the number of genes for which the two chromosomes have different alleles (assuming a binary representation), does not exceed a pre-specified maximum value D.

In other EAs, the properties of the objects being optimized may provide a natural basis for mating restriction. For example, in the optimization of ANNs, a simple mating restriction procedure is to allow crossover only between individuals with identical structure, as illustrated in Fig. 5.10.

In general, one of the main ideas behind the introduction of species is to

[4]The *hox* genes belong to a larger family of genes called **homeobox genes**. Mutations to homeobox genes can cause visible phenotypic changes. An example is the fruit fly Drosophila Melanogaster, in which an extra set of legs may be placed in the position normally occupied by antennae, a result of mutations in homeobox genes

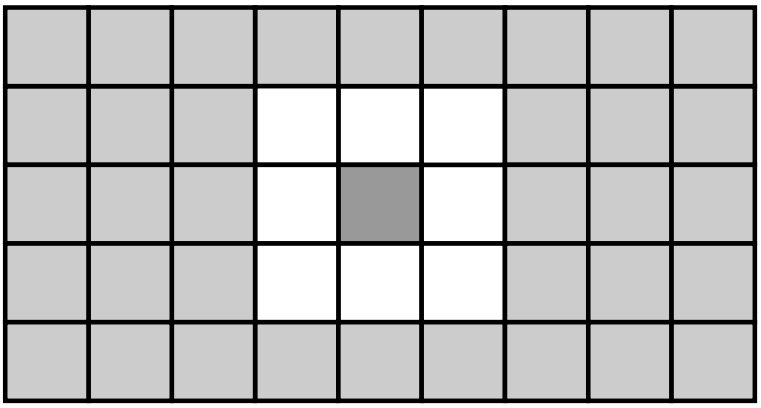

Figure 5.11: *Mating restriction in a grid-based EA. The selected individual, represented by the dark, central square in the figure, is only allowed to mate with one of its immediate neighbors, shown as white squares.*

maintain diversity in the population.

5.4.2 Subpopulation-based EAs

In **subpopulation-based EA**, the population of N individuals is divided into N_s groups with $\nu = N/N_s$ individuals each. Mating is only allowed to take place between individuals in the same subpopulation. Such EAs are also called **island models** , for obvious reasons.

The idea behind subpopulation-based EAs is to prevent situations in which an EA rapidly converges towards a local optimum (a situation known as **premature convergence**, making it difficult to reach the global optimum, as illustrated in Fig. 4.1. In a subpopulation-based EA, it is less likely that, in the initial generation, all the best individuals in the subpopulations are located close to the same local optimum.

However, if no interaction is allowed between individuals of different subpopulations, a subpopulation-based EA simply amount to running N_s EAs with ν individuals each. Thus, in some cases **tunneling** is allowed with some small probability p_t. Tunneling can be achieved simply by swapping two indiviuals between subpopulations.

5.4.3 Grid-based EAs

In **grid-based EAs**, also known as **diffusion models**, individuals are placed in a regular pattern as shown in Fig. 5.11, and mating restrictions are introduced based on the placement of individuals on the grid. When an individual i on the grid is to be replaced by a new individual, parents are selected only in

the neighborhood of the individual i. In the figure, the neighborhood contains eight individuals, but other neighborhood sizes are, of course, possible as well.

The topology of the grid can be chosen in many different ways. A common procedure is to use **periodic boundary conditions**, in which the two edges of a square grid are joined to form a toroidal space.

5.5 Experiment design

In many applications, the task is to find a system (e.g. a neural network) that can represent a given data set. For example, a neural network can be evolved to represent the mapping $(x_1, x_2, \ldots, x_n) \rightarrow f(x_1, x_2, \ldots, x_n)$. Another common example is data classification, in which a data set is to be divided into two (or more) classes, based on some criteria available in the data. The task of the EA is then to generate a classifier capable of doing so. For example, in binary classification (i.e. classification into two classes), linear classifiers of the form

$$\alpha_1 c_1 + \alpha_2 c2 + \ldots \alpha_n c_n > \beta, \tag{5.17}$$

can be evolved, where α_i and β are constants and c_i are some attributes available in the data set. With this type of classifier, an object (in the data set), with attributes $(c_1, c_2, \ldots, c_n)$ belongs to class I (say) if the inequality in Eq. (5.17) is satisfied, and to class II if it is not.

In all applications of the kinds described above, it is important to define both a **training data set** and **validation data set**. The training data is used when generating the representation, i.e. while running the EA. Now, if all the data is placed in the training set, it is often so that a very good representation of the data can be found. However, in that case, there is no way to investigate whether the representation indeed focuses on important aspects of the data, or whether it just manages to represent the noise in the data (a phenomenon known as **overfitting**). In order to investigate the quality of a representation, the validation data set is used. This data set should *not* be made available to the EA during training. If, during validation, it is found that the performance of the evolved system is similar to its performance on the training data set, one may conclude that a useful representation of the data set (i.e. one with predictive value) has been found.

Note that a validation data set can also be used for determining when to terminate the training process. It is commonly so that the errors (i.e. the inverse of the fitness) over both the training data set and the validation data set fall rapidly in the beginning of the training procedure. However, at some point, the error over the validation data set (which, again, is *not* made available to the training algorithm) usually starts rising. If the rising trend in the validation error persists, the training should be terminated, and the evolved

system for which the validation error was minimal can be taken as the best result obtained.

There are no strict rules for dividing a data set into training and validation parts. A common rule of thumb, however, is to use around 80% of the data for training, and the remaining 20% for validation [8].

Chapter 6

Versions of evolutionary algorithms

6.1 Evolutionary algorithms: different versions

During its development, the study of EAs has spawned a number of versions of the basic algorithm, which are all based on the same general principles but still differ, to some extent, from each other. In this chapter, a brief discussion of the various different versions of EAs will be given. The most important are

- Genetic algorithms,
- Genetic programming,
- Evolution strategies,
- Evolutionary programming.

Within each category, there exists also a number of different versions, putting a complete survey beyond the scope of this text.

6.2 Genetic algorithms

In **genetic algorithms** (GA), the variables of the problem are encoded in strings of genes, normally consisting of binary numbers or decimal numbers in the range $[0, 1]$. A population of such strings is generated, the chromosomes are decoded and the corresponding individuals are evaluated. Thereafter, new individuals are formed through selection, crossover, and mutation. It is possible to use a variable mutation probability, but its value is set externally, and is thus not encoded into the bit strings. In a standard GA, all strings are of the same length, and are therefore easy to manipulate, for instance when performing crossover between two chromosomes.

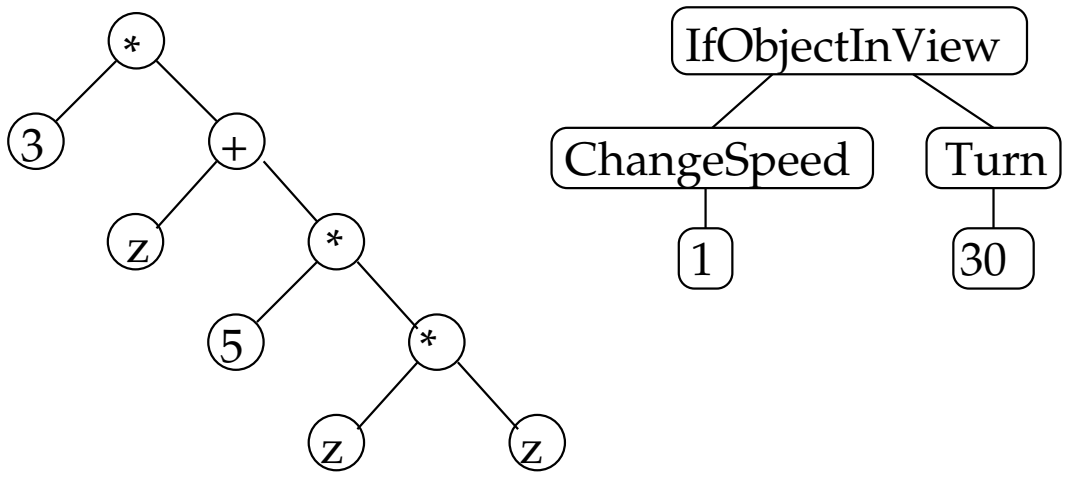

Figure 6.1: *Two GP trees. The tree to the left can be decoded to the function* $f(z) = 3(z + 5z^2)$*. The tree to the right tells a robot to increase its speed by one unit if it sees an object, and to turn by 30 degrees if it does not see any object.*

6.3 Genetic programming

After GAs, genetic programming (GP) is probably the most widely used type of evolutionary algorithms, and several different versions have been developed. In the original formulation of GP, which will be described first, tree-based representations were used for the individuals. In later versions, such as linear GP, simple, linear chromosomes have been used instead.

6.3.1 Tree-based genetic programming

As the name implies, tree-based GP is used for evolving combinations (trees) of elementary instructions, i.e. computer programs rather than strings of digits. Very often, GP is implemented using the LISP programming language, whose structure fits well with the tree–like structure of individuals in standard GP. However, GP can also be implemented in other programming languages. In tree-based GP, trees consisting of **elementary operators** and **terminals** are generated. The elementary operators require a number of input arguments, whereas the terminals take no inputs. For example, the operator $+$ requires two arguments, whereas the operator $\sin()$ requires one. The standard GP begins with the selection (by the user) of a suitable set of elementary operators and terminals. In a problem involving function approximation, a possible set of operators is $\{+, -, \times, /, \exp(), \sin(), \cos(), \ln()\}$, and the terminals could be chosen as the set of real numbers and the variable x. If the problem instead is to evolve a search strategy for an autonomous robot, the operators could consist of the set { IfObjectInView(,), Turn(), ChangeSpeed()}, where the operator

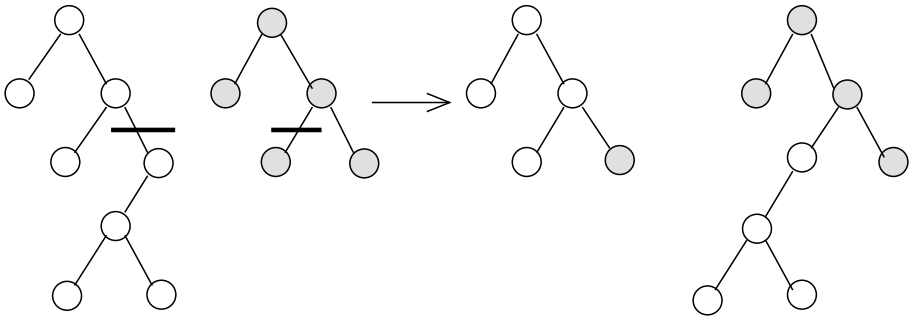

Figure 6.2: *The crossover procedure in GP.*

IfObjectInView takes two arguments, one saying what to do if an object is visible, and one saying what to do if no object is visible. The terminals would be the real numbers encoding the magnitudes of the changes in direction and speed, as well as the Stop action.

When a GP run is started, a population of random trees (individuals) is generated. Two examples are shown in Fig. 6.1. The tree in the left panel of the figure can be decoded to yield $f(z) = 3(z + 5z^2)$ which, using LISP–like notation, also can be written as $(*, 3, (+, z, (*, 5, (*, z, z))))$. The tree in the right panel gives the following control system for an object–seeking robot IfObjectInView(ChangeSpeed(1),Turn(30)), which simply means that the robot will increase its speed by one unit if it sees an object, and change its direction of motion by 30 degrees (clockwise, say) if it does not see an object. The IfObjectInView operator is the **root** of the tree.

Normally, some limits are set on the size of the trees in the first generation, by setting a limit on the number of elementary operators or the depth of the tree (i.e. the number of branchings from the root to the terminal furthest from the root). When GP trees are generated, it is vital to ensure that they are syntactically correct so that each elementary operator has the correct number of inputs.

After the initial trees have been generated, they are evaluated one by one. In the case of a function approximation problem, the difference (e.g. mean–square) between the correct function and the function provided by the GP tree would be measured, and the fitness would essentially be the inverse of the difference.

When all trees have been evaluated, new trees are formed through selection, crossover, and mutation. The crossover operator differs from that used in GAs. In GP, the two trees that are to be crossed over are split at random locations, and the subtrees below these locations are swapped between the two. The procedure, which clearly leads to trees of varying size, is illustrated in Fig. 6.2.

The mutation operator can change both terminals and elementary oper-

ators, but must be implemented in such a way as to maintain the syntactic correctness of the trees.

6.3.2 Linear genetic programming

Unlike tree-like GP, **linear GP** (LGP) is used for evolving linear sequences of basic instructions defined in the framework of an **imperative programming language**.[1] Two central concepts in LGP are **registers** and **instructions**. Registers are of two basic kinds, **variable registers** and **constant registers**. The former can be used for providing input, manipulating data, and storing the output resulting from a calculation. As a specific example, consider the expression

```
r3 := r1 + r2;
```
(6.1)

In this example, the instruction consists of assigning a variable register (`r3`) the sum of the contents in two other variable registers (`r1` and `r2`). As a second example, consider the instruction

```
r1 := r1*c1;
```
(6.2)

Here, a variable register `r1`, is assigned its previous value multiplied by the contents of a constant register `c1`. In LGP, a sequence of instructions of the kind just described are specified in a linear chromosome. The encoding scheme must thus be such that it identifies the two **operands** (i.e. `r1` and `c1` in the second example above), the **operator** (multiplication, in the second example), as well as the **destination register** (`r1`). An LGP instruction can thus be represented as a sequence of integers that identify the operator, the index of the destination registers, and the indices of the registers used as operands. For example, if only the standard arithmetic operators (addition, subtraction, multiplication, and division) are used, they can be identified e.g. by the numbers 1,2,3, and 4. The set of all allowed instructions is called the **instruction set** and it may, of course, vary from case to case. However, no matter which instruction set is employed, the user must make sure that all operations generate valid results. For example, divisions by zero must be avoided. Therefore, in practice, **protected definitions** are used. An example of a protected definition of the division operator is

$$r_i := \begin{cases} r_j/r_k & \text{if } r_k \neq 0, \\ r_j + c_{\max} & \text{otherwise,} \end{cases} \tag{6.3}$$

[1] In imperative programming, computation is carried out in the form of an algorithm consisting of a sequence of instructions. Typical examples of imperative programming languages are C, Fortran, Java, and Pascal, as well as the machine languages used in common microprocessors and microcontrollers. By contrast, **declarative programming** involves a description of a structure but no explicit algorithm (the specification of an algorithm is left to the supporting software that interprets the declarative language). A web page written in HTML is thus declarative.

Instruction	Description	Instruction	Description
Addition	$r_i := r_j + r_k$	Sine	$r_i := \sin r_j$
Subtraction	$r_i := r_j - r_k$	Cosine	$r_j := \cos r_j$
Multiplication	$r_i := r_j \times r_k$	Square	$r_i := r_j^2$
Division	$r_i := r_j / r_k$	Square root	$r_i := \sqrt{r}_j$
Exponentiation	$r_i := \mathrm{e}^{r_j}$	Conditional branch	if $r_i > r_j$
Logarithm	$r_i := \ln r_j$	Conditional branch	if $r_i \leq r_j$

Table 6.1: *Examples of typical LGP operators. Note that the operands can be either variable registers or constant registers.*

where $c_{\max}$ is a large (pre-specified) constant. Note that the instruction set can, of course, contain operators other than the simple arithmetic ones. Some examples of common LGP instructions are shown in Table 6.1. The usage of the various instructions should be clear except, perhaps, in the case of the branching instructions. Commonly, these instructions are used in such a way that the *next* instruction is skipped *unless* the condition is satisfied. Thus, for example, if the following two instructions appear in sequence

```
if (r1 < r2)
 r1 := r2 + r3;
```
(6.4)

the value of `r1` will be set to `r2+r3` *only* if `r1 < r2` when the first instruction is executed. Note that it is possible to use a sequence of conditional branches in order to generate more complex conditions. Clearly, conditional branching can be augmented to allow e.g. jumps to a given location in the sequence of instructions. However, the use of jumping conditions can raise significantly the complexity of the representation. As soon as jumps are allowed, one faces the problems of (1) making sure that the program terminates, i.e. does not enter an infinite loop, and (2) avoiding jumps to non-existent locations. In particular, application of the crossover and mutation operators (see below) must then be followed by a screening of a newly generated program, to ascertain that the program can be executed correctly. Jumping instructions will not be considered further here.

In LGP, chromosomes are used, similar to the those employed in a GA. An example of an LGP chromosome is shown in Fig. 6.3. Note that some instructions (e.g. addition) need four numbers for their specification, whereas others (e.g. exponentiation) need only three. However, in order to simplify the representation, one may still represent each instruction by four numbers, simply ignoring the fourth number for those instructions where it is not needed. Note that the four numbers constituting an instruction may have different range. For example, the operands may involve both variable registers and constant registers, whereas the destination register must be a variable register. In the specific example shown in Fig. 6.3, there are three variable registers available

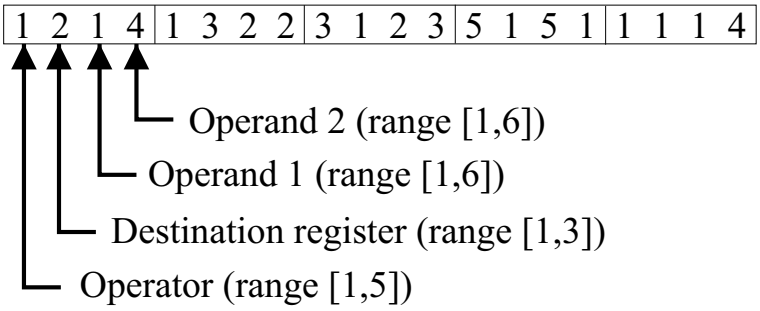

Figure 6.3: *An example of an LGP chromosome.*

Genes	Instruction	Result
1, 2, 1, 4	$r_2 := r_1 + c_1$	$r_1 = 1, r_2 = 2, r_3 = 0$
1, 3, 2, 2	$r_3 := r_2 + r_2$	$r_1 = 1, r_2 = 2, r_3 = 4$
3, 1, 2, 3	$r_1 := r_2 \times r_3$	$r_1 = 8, r_2 = 2, r_3 = 4$
5, 1, 5, 1	if $(r_1 > c_2)$	$r_1 = 8, r_2 = 2, r_3 = 4$
1, 1, 1, 4	$r_1 := r_1 + c_1$	$r_1 = 9, r_2 = 2, r_3 = 4$

Table 6.2: *Evaluation of the chromosome shown in Fig. 6.3, in a case where the input register r_1 was initially assigned the value 1. The variable registers r_2 and r_3 were both set to 0, and the constant registers were set as $c_1 = 1, c_2 = 3, c_3 = 10$. The first instruction (top line) is decoded from the first four genes in the chromosome etc. The resulting output was taken as the value contained in r_1 at the end of the calculation.*

(r_1, r_2, and r_3) and three constant registers (c_1, c_2, and c_3), as well as five operators, namely addition, subtraction, multiplication, division, and the conditional branch instruction 'if $(r_i > r_j)$'. Let $\mathcal{R}$ denote the set of variable registers, i.e. $\mathcal{R} = \{r_1, r_2, r_3\}$, and $\mathcal{C}$ the set of constant registers, i.e. $\mathcal{C} = \{c_1, c_2, c_3\}$. Let $\mathcal{A}$ denote the union of these two sets, so that $\mathcal{A} = \{r_1, r_2, r_3, c_1, c_2, c_3\}$. The set of operators, finally, is denoted $\mathcal{O}$. Thus, in the set $\mathcal{O} = \{o_1, o_2, o_3, o_4, o_5\}$, the first operator ($o_1$) represents + (addition) etc. An instruction is encoded using four numbers. The first number, in the range [1,5], determines the operator as obtained from the set $\mathcal{O}$ and the second number determines the destination register, i.e. an element from the set $\mathcal{R}$. The third and fourth numbers determine the two operands, taken from the set $\mathcal{A}$. For some operators, only one operand is needed. In such cases, the fourth number is simply ignored, as mentioned above.

Before the chromosome can be evaluated, the registers must be initialized. In the particular case considered in Fig. 6.3, the constant registers were set as $c_1 = 1$, $c_2 = 3$, and $c_3 = 10$. These values then remained constant throughout the entire run, i.e. for all individuals. The variable registers should be initialized just before the evaluation of each individual. The input was provided

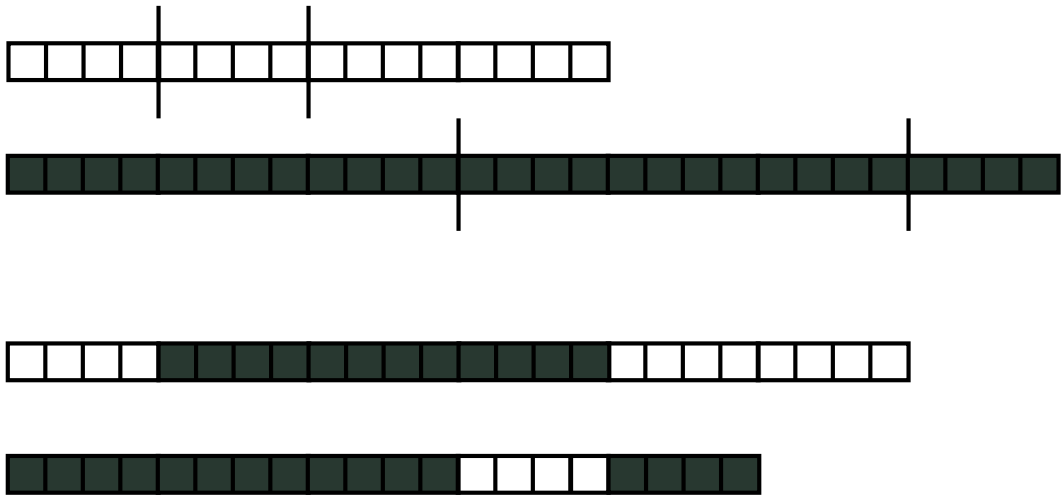

Figure 6.4: *An illustration of crossover in LGP. The two parent chromosomes are shown in the upper part of the figure, and the two offspring chromosomes are shown below. Note that two crossover points are selected in each chromosome.*

through register r_1, and the other two variable registers (r_2 and r_3) were initialized to zero. The output could in principle be taken from any register(s). In this case, r_1 was used. The computatation obtained from the chromosome shown in Fig. 6.3, in a case where r_1 (the input) was set to 1, is given in Table 6.2.

The evolutionary operators used in connection with LGP are quite similar to those used in an ordinary GA. However, two-point crossover is commonly used (instead of single-point crossover), since there is no reason to assume that the length of original, random chromosomes would be optimal. With two-point crossover, normally applied with crossover points between (rather than within) instructions, length variation is obtained. The crossover procedure is illustrated in Fig. 6.4.

6.4 Evolution strategies

Evolution strategies (ES) were invented in Germany, and were initially developed independently of GA and GP (which are mainly American inventions). Nowadays, the different versions of EAs have borrowed properties from each other, and the differences are therefore less marked than before. Here we will introduce a standard evolution strategy lacking most of the bells and whistles. Originally, ES used real number encodings, and new individuals were formed using only a mutation operator. For simplicity, let us consider a function maximization task for a function $g = g(x_1, \ldots, x_n)$. The evolution strategy operates as follows: The population consists of only one individual, which is

represented by a pair of vectors $\{(x_1, \ldots, x_n), (\sigma_1, \ldots, \sigma_n)\}$, where $(x_1, \ldots, x_n)$ are the variables of the problem and $(\sigma_1, \ldots, \sigma_n)$ is a **variance vector** which is used when forming new individuals.

An individual is evaluated by simply computing $g(x_1, \ldots, x_n)$. Then, the vector $(x_1, \ldots, x_n)$ is varied (mutated) according to

$$x_i \rightarrow x_i' = x_i + N(0, \sigma_i), \ i = 1, \ldots, n, \tag{6.5}$$

where $N(0, \sigma_i)$ is a Gaussian random number with expectation value 0 and standard deviation σ_i. When the vector $(x_1', \ldots, x_n')$ has been found, the new individual is evaluated by computing $g(x_1', \ldots, x_n')$. If the result is better than $g(x_1, \ldots, x_n)$, i.e. gives a higher value, the old individual is replaced by the new one. If not, the old individual is kept and a new mutation of $(x_1, \ldots, x_n)$ is made according to Eq. (6.5).

An important property of ES is **adaptation** of the variance vector. A rule of thumb is that the ratio Γ of the number of successful mutations (those that give $g(x_1', \ldots, x_n') > g(x_1, \ldots, x_n)$) to the total number of mutations should be $1/5$. If Γ exceeds $1/5$ over a period of five updates, i.e. if more than one of the five mutations are successful, the values of the elements of the variance vector (which we here assume are equal) are increased by a constant $\alpha > 1$, otherwise they are decreased by a constant $\beta < 1$.

In later versions of ES, **self–adaptation** was introduced for the elements of the variance vector. In this case, the values σ_i all undergo independent mutations. Other additions, which will not be considered here, have been the introduction of larger populations and crossover operators.

6.5 Evolutionary programming

Evolutionary programming (EP) was developed in the 1960s, with the specific aim of generating machine intelligence. Initially, EP was implemented using FSMs. In later years, generalizations and extensions have been made, and EP is today used also in e.g. function optimization problems.

In EP, a population of random FSMs is generated, usually with some limits on the sizes of the FSMs, just as for the initial generation of trees in GP. The task of the FSMs is to provide the best action (for a robot, say), given information of all previous events. The events consist of the presentation of input symbols from an input alphabet. As a response to the input, the FSM produces an output (e.g. a move). The performance of each FSM is evaluated, and new FSMs are formed through selection and mutation. The mutation operator can have different effects: it may change the starting state of the FSMs, the transition conditions, the transition targets, as well as the number of states. The EP is allowed to run until the robot must respond to some external event. At this point, the best FSM from the population is allowed to make the response for

the robot. Then, the input symbol which triggered the response is added to the list of symbols experienced by the robot, and the population continues to evolve (this time using an input sequence which is one step longer than before the response) until a new response is called for.

Appendix A: Binomial identities

In this appendix will be derived some relations involving binomial coefficients, which are needed in the discussion concerning analytical properties of GAs. Using the equation

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k} \tag{A1}$$

where the binomial coefficient $\binom{n}{k}$ is defined as

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{A2}$$

and setting $a = x, b = 1$, the equation

$$(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k \tag{A3}$$

is obtained. Setting $x = 1$, it is easy to see that

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n. \tag{A4}$$

Differentiating Eq. (A3) with respect to x, one obtains

$$n(1+x)^{n-1} = \sum_{k=0}^{n} k \binom{n}{k} x^{k-1}. \tag{A5}$$

Again setting $x = 1$, the result

$$n2^{n-1} = \sum_{k=0}^{n} k \binom{n}{k} \tag{A6}$$

is obtained. Multiplying both sides of Eq. (A5) by x, and differentiating again, the identity

$$\sum_{k=0}^{n} k^2 \binom{n}{k} = n(n+1)2^{n-2} \tag{A7}$$

is generated. Continuing the procedure one step further, the equation

$$\sum_{k=0}^{n} k^3 \binom{n}{k} = n^2(n+3)2^{n-3}. \tag{A8}$$

is obtained. Using this iterative procedure the sum

$$s_p = \sum_{k=0}^{n} k^p \binom{n}{k}, \tag{A9}$$

can be derived for any p, given s_{p-1}, even though a simple closed-form expression of s_p cannot be generated for arbitrary p.

From the definition of the binomical coefficients, the following equations can also be derived

$$\begin{aligned} \binom{n}{k-1} &= \frac{n!}{(k-1)!(n-k+1)!} = \frac{k}{n-k+1}\frac{n!}{k!(n-k)!} = \\ &= \frac{k}{n-k+1}\binom{n}{k}, \end{aligned} \tag{A10}$$

and

$$\binom{n}{k+1} = \frac{n!}{(k+1)!(n-k-1)!} = \frac{n-k}{k+1}\frac{n!}{k!(n-k)!} = \frac{n-k}{k+1}\binom{n}{k}. \tag{A11}$$

Appendix B: Artificial neural networks

The aim of this appendix is to give a very brief introduction to neural networks. For a more thorough introduction see [8]. Artificial neural networks (ANNs) are computational structures (loosely) based on the structure and function of biological neural networks that appear in the brains of animals. Basically, an ANN is a set of interconnected elements, called **neurons**.

Neurons

The main computational element in neural networks, the neuron, is illustrated in Fig. B1. In essence a neuron receives weighted inputs, either from other neurons (or from the neuron itself, in some networks, see below), or from input elements, sums these inputs, adds a **bias** to form an internal signal, and produces an output by passing the internal signal through a **squashing function**. ANNs may operate either in discrete time or continouous time. In the former case, the output at time t of neuron i in a network, denoted $x_i(t)$ is computed as

$$x_i(t) = \sigma\left(\sum_j w_{ij} z_j + b_i\right), \tag{B1}$$

where w_{ij} are the **connection weights**, z_j the input signals to the neuron and b_i is the bias. In the continuous case, the output at time $x(t)$ is typically given by the differential equation

$$\tau_i \dot{x}_i + x_i = \sigma\left(\sum_j w_{ij} z_j + b_i\right), \tag{B2}$$

where τ_i are time constants. Typically, the z_j are inputs from other neurons (or even the neuron i itself), thus joining the equations for different neurons into a

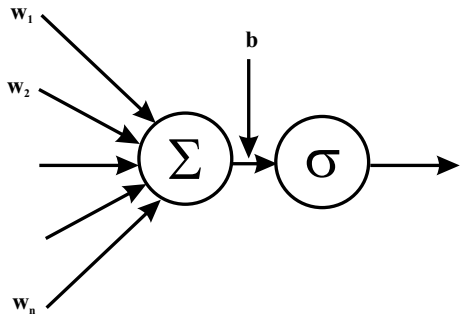

Figure B1: *A schematic illustration of a neuron from an ANN.*

set of coupled non-linear differential equations. Other equations are also possible, of course. In any case, the neuron equations presented above represent very strong simplifications of the very intricate functionality of actual biological neurons. The squashing function can be chosen in various different ways. Two common choices are the **logistic function**

$$\sigma(z) = \frac{1}{1 + \mathrm{e}^{-cz}}, \tag{B3}$$

which restricts the output to the range $[0, 1]$, and the hyperbolic tangent

$$\sigma(z) = \tanh cz, \tag{B4}$$

which restricts the output to $[-1, 1]$. In both functions, c is a positive constant, which determines the steepness of the squashing function. If c is set to a very large value, $\sigma(z)$ approaches a step function.

Network types

The two main types of neural networks are **feedforward neural networks** (FFNNs) and **recurrent neural networks** (RNNs). In the former, neurons are arranged in layers, and signals flow from the input to the first layer, from the first layer to the second layer etc., until the output layer is reached. By contrast, in RNNs, any neuron may be connected to any other neuron. RNNs often operate in continouous time, using Eq. (B2) to represent neurons. The difference between FFNNs and RNNs is illustrated in Fig. B2. For the FFNN, shown in the left panel of the figure, the input signals are denoted I_j, the neurons in the middle layer (the **hidden layer**) x^{H}, and the neurons in the output layer x^{O}. Assuming the network operates in discrete time, which is common in FFNNs, the corresponding network equations will be

$$x_i^{\mathrm{H}} = \sigma\left(\sum_j w_{ij}^{\mathrm{IH}} I_j + b_i\right), \tag{B5}$$

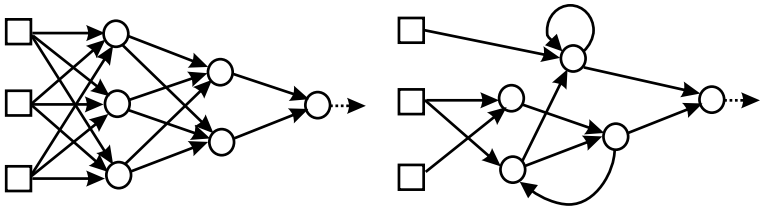

Figure B2: *Different types of ANNs: the left panel shows a feedforward neural network (FFNN), and the right panel a recurrent neural network (RNN).*

where w_{ij}^{IH} are the weights connecting **input element** j to neuron i in the hidden layer, and

$$x_i^{\mathrm{O}} = \sigma\left(\sum_j w_{ij}^{\mathrm{HO}} x_j^{\mathrm{H}} + b_i\right), \tag{B6}$$

where w_{ij}^{HO} are the weights connecting the hidden layer to the output layer. Note that the first (leftmost) layer (the **input layer** in the FFNN) does *not* consist of neurons: the elements in this layer, indicated by squares to distinguish them from neurons that are indicated by circles, simply serve to distribute the input signals to the neurons in the hidden layer. No squashing function is associated with the elements in the input layer. In general, the term **units** will be used to refer to either input elements or neurons.

An RNN is shown in the right panel of Fig. B2. Here, neurons are not arranged in layers, and the output of a typical neurons is given by

$$\tau_i \dot{x}_i + x_i = \sigma\left(\sum_j w_{ij} x_j(t) + \sum_j w_{ij}^{\mathrm{I}} I_j(t) + b_i\right), \tag{B7}$$

where w_{ij} are the weights connecting neurons to each other, and w_{ij}^{I} are weights connecting input j to neuron i. Again, the inputs are shown as squares in the figure, and the neurons are shown as circles. Since an RNN is not arranged in layers, there are no natural output neurons. Instead, some neurons are simply selected to be the output neurons. The number of such neurons varies, of course, from problem to problem.

Training an ANN

In a neural network, the computation is intertwined with the structure of the network (i.e. the number of neurons, and their connections to each other). If the structure is changed, the computation changes as well. Thus, in order for a network to perform a given type of computation, both the number of neurons and the network weights must be set to appropriate values, a process known as **training**. There are various algorithms for training neural networks. The most

common algorithm, applicable to FFNNs, is the **backpropagation** algorithm [8].

Bibliography

[1] Sutton, R.S. and Barto, A.G. *Reinforcement Learning: An Introduction*, MIT Press, 1998

[2] Bäck, T., Fogel, D.B., and Michalewicz, Z. (Eds.), *Handbook of Evolutionary Computation*, Institute of Physics Publishing and Oxford University Press, 1997

[3] Clark, W.R. and Grunstein, M. *Are we hardwired? The role of genes in human behavior*, Oxford University Press, 2000

[4] Davidson, E.H. *Genomic regulatory systems - development and evolution*, Academic Press, 2001

[5] Dawkins, R. *Climbing Mount Improbable*, Penguin Books, 1997

[6] Goldberg, D. *Genetic Algorithms in Search, Optimization, and Learning*, Addison-Wesley, 1989

[7] Gruau, F. *Automatic definition of modular neural networks*, Adaptive Behavior **3**, pp. 151-183, 1995

[8] Haykin, S. *Neural networks - A comprehensive foundation*, Prentice-Hall, 1994

[9] Hillis, W.D., *Co-evolving parasites improve simulated evolution as an optimization procedure*, Physica D, **42**, pp. 228-234, 1990

[10] Holland, J. H. *Adaptation in Natural and Artificial systems*, University of Michigan Press, 1975 (2nd Ed.: MIT Press, 1992)

[11] de Jong, K. A. *An analysis of the behavior of a class of genetic adaptive systems*, Doctoral dissertation, University of Michigan, 1975

[12] Kandel, E. R. *The molecular biology of memory storage: A dialog between genes and synapses*, Bioscience Reports **24**, pp. 477-533, 2004

[13] Kitano, H. *Designing neural networks using genetic algorithms with graph generation system*, Complex Systems **4**, pp. 461-476, 1990

[14] Mitchell, M. *An introduction to genetic algorithms*, MIT Press, 1996

[15] Ptashne, M. *A genetic switch: Phage λ and higher organisms*, 2nd ed., Cell press and Blackwell scientific publications, 1992

[16] Savage, J., Marquez, E., Pettersson, J., Trygg, N., Petersson, A., and Wahde, M. *Optimization of Waypoint-Guided Potential Field Navigation Using Evolutionary Algorithms* To appear in: Proc. of IROS2004, 2004

[17] Vose, M. D. *The simple genetic algorithm*, MIT press, 1999

[18] Wahde, M. and Nordahl, M. *Co-evolving pursuit-evasion strategies in open and confined regions*, In: Proc. of Artificial Life VI, pp. 472-476, 1998

[19] Yao, X. *Evolving artificial neural networks*, Proceedings of the IEEE, **87**, pp. 1423-1447, 1999

[20] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*, 3^{rd} Ed., Springer, 1996