Evolutionary Robotics

The Use of Artificial Evolution in Robotics

A tutorial presented at Ro-Man 2007

Mattias Wahde

Technical Report TR-BBR-2007-001
Department of Applied Mechanics
Chalmers University of Technology
Göteborg, Sweden
http://www.me.chalmers.se/~mwahde/AdaptiveSystems/TechnicalReports.html

Contents

Abbreviations

The abbreviations used in the tutorial are listed below, in alphabetical order. In addition, the abbreviations are also defined the first time they occur in the text. If an abbreviation is shown in plural form, an s is added to the abbrevation. Thus, for example, the term *artificial neural networks* is abbreviated as ANNs.

AI	Artificial intelligence
ALIFE	Artificial Life
ANN	Artificial neural network
BBR	Behavior-based robotics
DOF	Degrees of freedom
EA	Evolutionary algorithm
EP	Evolutionary programming
ER	Evolutionary robotics
ES	Evolution strategy
FFNN	Feedforward neural network
FSM	Finite-state machine
GA	Genetic algorithm
GP	Genetic programming
IR	Infrared
LGP	Linear genetic programming
RNN	Recurrent neural network
ZMP	Zero-moment point

Chapter 1

Introduction to evolutionary robotics

The topic of this tutorial is **evolutionary robotics** (ER), the sub-field of robotics in which **evolutionary algorithms** (EAs) are used for generating and optimizing the (artificial) brains (and sometimes bodies) of robots.

In this chapter a brief introduction to the topics of evolution and autonomous robots will be given, followed by a simple example involving the evolution of a cleaning behavior.

1.1 Biological and artificial evolution

EAs are methods for search and optimization based on **darwinian evolution**, which will be described further in the next chapter.

Why should one use these algorithms in robotics? There are many good reasons for doing so; First of all, if properly designed, an EA allows structural as well as parametric optimization of the system under study. This is particularly important in ER, where it is rarely possible (or even desirable) to specify, in advance, the structure of e.g. the artificial brain of a robot. Thus, mere parametric optimization would not be sufficiently versatile for the problem at hand.

Second, 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.

Finally, 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, shows that there are many different solutions to the problem of survival. Another classical example is the evolution of the eye. Richard Dawkins notes in one of his books [23] that the eye has evolved

in forty (!) different and independent ways. Thus, when nature approached 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 compound 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 ER, the ability of the EA to come up with solutions that were not anticipated is very important. Particularly in complex problems, it often happens that an EA finds a solution which is remarkably simple, yet very difficult to arrive at by other means.

1.2 Autonomous robots

EAs can, in principle, be applied in most robotics problems. However, most applications of EAs in robotics concern **autonomous robots**, i.e. robots that move freely and without direct human supervision. While the number of autonomous robots is rapidly increasing, the most common types of robots in industries are still stationary robotic arms operating in very structured and controlled environments. Such robots are normally equipped with very limited cognitive abilities, and stationarity has therefore been a requirement rather than an option.

Autonomous robots, on the other hand, are expected to operate in **unstructured environments**, i.e. environments that change rapidly and in an unpredictable way, so that it is impossible to rely on pre-defined maps. Thus, autonomous robots have much more in common with biological organisms than stationary robotic arms, and it is therefore not surprising that computational methods based on biological phenomena have come to be used in connection with autonomous robots.

The influence from biology is evident already in the structure of the artificial brains used in autonomous robots; It is common to use **behavior-based robotics** (BBR), in which robots are equipped with a repertoire of simple behaviors, which are generally running concurrently and which are organized to form a complete robotic brain. By contrast, classical **artificial intelligence** (AI), with its sense-plan-act structure (see Chapter 3), has much less in common with biological systems.

Clearly, there exists many different approaches to the problem of generating brains for artificial robots, and this tutorial does not make an attempt to cover all of these methods. Instead, the discussion will be limited to methods involving EAs in one way or another. Readers interested in other methods are referred to [3] and references therein.

Within the field of autonomous robots, there exists many different robot types, the most common being **wheeled robots** and **walking robots**. Within

the sub-field of walking robots, which will be discussed further in Subsect. 3.4.1, there exists **bipedal robots** and **quadrupedal** robots, as well as robots with more than four legs. Bipedal robots with an approximately human shape are also called **humanoid robots**.

Note that, in this tutorial, a rather generous definition of the phrase *autonomous robot* will be used, including not only real, physical robots, but simulated robots as well. This is necessary, since, by their very nature, EAs must often be used in simulated environments. Evolution directly in hardware will be covered briefly as well, however, in Subsect. 3.4.2.

However, applications in **artificial life** (ALIFE) will not be considered. In ALIFE, the aim is often to study biological phenomena in their own right, whereas in ER, the aim is to generate robots with a specific purpose, and the discussion will henceforth omit results from ALIFE. The reader is referred to [69], and references therein, for further information on the topic of ALIFE.

Finally, before proceeding with two simple examples from the field of ER, the concept of a robotic brain should be defined: some researchers use the term *control system*. However, in the author's opinion, this term is misleading, as it leads the reader to think of classical control theory. Clearly, concepts from classical control theory *are* relevant in autonomous robots; For example, the low-level control of the motors of autonomous robots is often taken care of by PI- or PID-regulators. However, autonomous robots are expected to make their own decisions in complex, unstructured environments, in which systems based *only* on classical control theory simply are insufficient. Thus, hereafter, the term **robotic brain** (or, simply, *brain*) will be used when referring to the system that provides an autonomous robot, however simple, with the ability to process information and decide upon which actions to take.

1.3 A simple example

In this section, a brief introduction to ER will be given through a simple example, namely the evolution of a cleaning behavior. Consider an arena of the kind shown in the left panel of Fig. 1.1. The large cylinder represents a simple, differentially steered, simulated two-wheeled robot, whereas the smaller (stationary) objects are considered to be garbage, and are to be removed by the robot. The aim is to use an evolutionary algorithm to generate a brain capable of making the robot clean the arena.

1.3.1 Cleaning behavior in simulation

The first choice that must be made is whether to evolve robotic brains in simulation, or directly in hardware (an issue that will be discussed further in Subsect. 3.4.2). For the simple problem described here, which was part of an

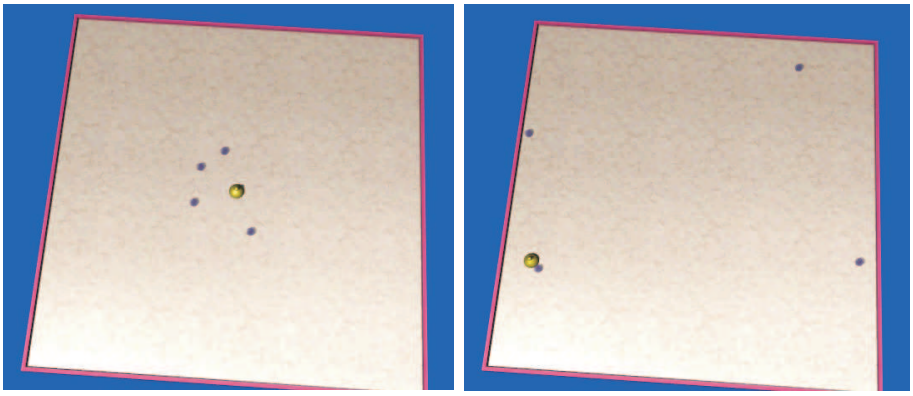


Figure 1.1: *A simple, simulated cleaning robot (the large, circular object) in action [137]. The initial state is shown in the left panel, and the final state in the right panel.*

investigation concerning elementary behavioral selection [137], the choice was made to use simulated robots during evolution, and then attempt to transfer the best robotic brain found in simulation to a physical robot.

Next, a representation must be chosen for the robotic brains. Ideally, the EA should be given as much flexibility as possible but, in practice, some limitations must generally be introduced. In this problem, the robotic brains were represented as a generalized finite-state machines (GFSMs, see Appendix B), and the EA acted directly on the GFSMs rather than on a chromosomal encoding of them (see Sect. 2.7.2). In the beginning of the simulation, the GFSMs were small (i.e. contained very few states). However, the EA was allowed to change the sizes of the GFSMs during the simulation.

The simulated robot was equipped with very simple sensors that could distinguish garbage objects from walls, but not much more.

The next step in the application of an EA is to choose a suitable fitness measure, i.e. a performance measure for the evolving robotic brains. In the particular case considered here, the aim of the robot was to place the garbage objects as far from the center of the arena as possible. Thus, the fitness measure was simply chosen as the mean square distance, counted from the center of the arena, of all the garbage objects at the end of the evaluation.

Furthermore, each robot was evaluated against several (usually five) different starting configurations, with garbage objects placed in different positions, to avoid a situation where the evolved robot would learn only to cope with a given configuration.

Next, an EA was set up, in which a population of robotic brains (in the form of GFSMs) was generated. As the EA progressed, evaluating robotic brains, and generating new ones using selection, crossover, and mutations (see Sect. 2.5), better and better results were obtained. In early generations, the simulated robots did little more than run around in circles near their starting po-

sition. However, some robots were lucky enough to hit one or a few garbage objects, thereby moving them slightly towards the walls of the arena. Before long, there appeared robots that would hit *all* the garbage objects.

The next evolutionary leap led to purposeful movement of objects. Here, an interesting method appeared: Since both the body of the robot and the garbage objects were round, objects could not easily be moved forward; Instead, they would slide away from the desired direction of motion. Thus, a method involving several states of the GFSM was found, in which the robot moved in a zig-zag fashion, thus managing to keep the garbage object in front.

Next, robots appeared that were able to deliver a garbage object at a wall, and then return towards the center of the arena in a curved, sweeping motion, in order to detect objects remaining near the center of the arena.

Towards the end of the run, the best evolved robots were able to place all, or almost all, garbage objects near a wall, regardless of the starting configuration.

An example of the motion of an evolved robot is provided on the CD associated with the tutorial, in the file `Cleaning_Simulation.avi`.

1.3.2 Cleaning behavior in a Khepera robot

Needless to say, the aim of ER is to generate real, physical robots capable of performing useful tasks. Simulation is a useful tool in ER, but the final results should be tested in physical robots.

The simulations for the cleaning robot discussed above were, in fact, strongly simplified, and no attempt was made to simulate a physical robot exactly. Nevertheless, the best robotic brain obtained in the simulations was adapted for the Khepera robot (see Appendix C), the adaptation consisting mainly of rescaling the parameters of the robotic brain, such as e.g. wheel speeds and sensor readings, to appropriate ranges. Quite amazingly, the evolved robotic brain worked almost as well in the Khepera robot as in the simulated robots, as can be seen in the film `Cleaning_Khepera.avi`, which is also available on the tutorial CD. Thus, in this particular case, the transition from simulation to real robots was quite simple, probably due to the simplicity of the problem. However, despite its simplicity, the problem just described still illustrates the power of EAs as a method for generating robotic behaviors. Now, before proceeding with a survey of results obtained in the field of ER, an introduction to EAs will be given.

Chapter 2

Fundamentals of evolutionary algorithms

2.1 Introduction

EAs are methods for search and optimization, (loosely) based on the properties of darwinian evolution. Typically, EAs are applied in optimization problems where classical optimization methods, such as e.g. gradient descent, Newton's method etc. [1] are difficult or impossible to apply. Such cases occur when the objective function (i.e. the function whose maximum or minimum one wants to find) is either not available in analytical form or is very complex (combining, say, thousands of real-valued and integer-valued variables).

Since the introduction of EAs in the 1970s, the number of applications of such algorithms has grown steadily, and EAs are today used in fields as diverse as engineering, computational biology, finance, astrophysics, and, of course, robotics.

Below, a brief introduction to EAs will be given. Clearly, it is impossible to review completely the vast topic of EAs on a few pages. Thus, readers interested in learning more about EAs should consult other sources as well, e.g. [4], [51], [84], and [88].

Before introducing EAs, however, it is necessary to give a brief introduction to the terms and concepts that appear in the study of biological evolution. Needless to say, this, too, is a vast topic, and only the most basic aspects will be given below. For more information on biological evolution, see e.g. [22] and [23]. As their name implies, EAs are based on processes similar to those that occur during biological evolution. A central concept in the theory of evolution 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.

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 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. Thus, each individual of a species carries 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. 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 **messenger RNA** (mRNA). Next, **proteins** are generated in a process called **translation**, using mRNA as a template.

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.

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. The complete genome of an individual, with all its settings (encoding e.g. hair color, eye color, height 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 more intelligent 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

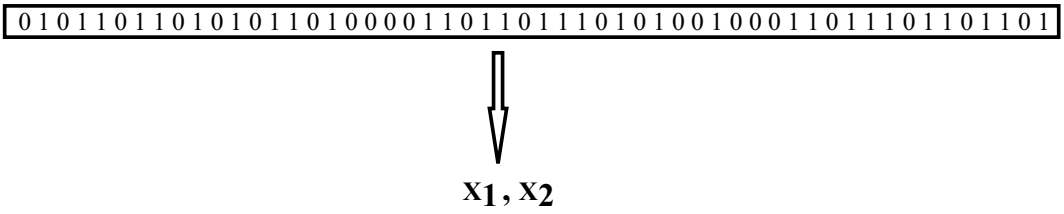


Figure 2.1: *Typical usage of a chromosome in a genetic algorithm. The 0s and 1s are the genes, which are used as binary numbers to form, in this case, two variables x_1 and x_2, using e.g. the first half of the chromosome to represent x_1 and the second half to represent x_2.*

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.

2.2 Biology vs. evolutionary algorithms

It should be noted that the description of reproduction above (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, relevant to the topic of EAs, comes from the way chromosomes are used. The most common usage in EAs is illustrated in Fig. 2.1. 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.

2.2.1 Embryological development

The simple procedure shown in Fig. 2.1 is, in fact, just a caricature of the process taking place when an individual is generated in a biological systems. The biological process is illustrated schematically in Fig. 2.2. First of all, in biolog-

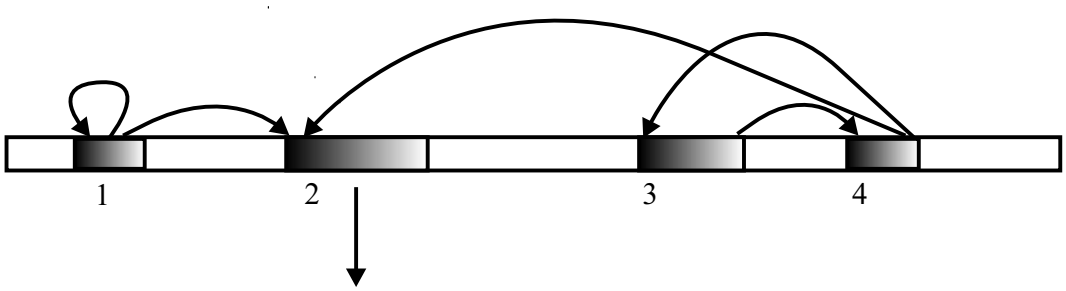


Figure 2.2: *A schematic illustration of the function of the genome in a biological system. Four genes are shown. Genes 1, 3, and 4 are transcription factors, which regulate each other's levels of expression, as well as that of gene 2, which is a structural gene. The arrow below gene 2 indicates that 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.*

ical 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 [109]. 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. 2.2).

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

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

2.2.2 Multicellularity

An additional simplification in most EAs 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 set of chromosomes, and therefore the set of genes, is the *same* in all **somatic** (non-germ) cells in a biological organism. 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.

2.2.3 Gene regulation in adult animals

In EAs, the individual resulting from the decoding procedure shown in Fig. 2.1 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 finally followed by a subroutine active during adult life. The amount needed of any given protein usually varies with time, and continuous gene regulation is thus essential in biological organisms.

2.2.4 Summary

To summarize the description of evolution, it can be noted that it is a process that acts on populations of individuals. Information is stored in the 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 of two separate individuals. Mutations provide further material for the evolutionary process.

Finally, it should be noted that, while EAs are inspired by biological evolution, they often represent a strong simplification of the very complex processes occurring in biological evolution. Clearly, there is no need to reproduce biology exactly: In ER, the aim is to generate e.g. artificial brains for autonomous robots by means of any method that allows one to do so, and there is nothing preventing deviations from biology.

2.3 Taxonomy of evolutionary algorithms

The taxonomy of EAs is sometimes confusing, particularly since different authors sometimes use slightly different definitions of the particular version of EAs they are using. Here, EAs will be considered the umbrella term, covering all types of algorithms based on darwinian evolution. A common special case

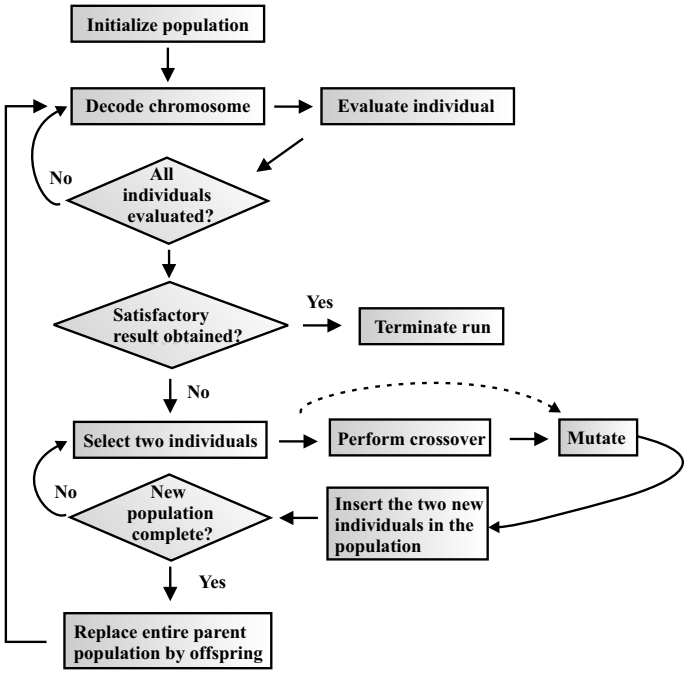


Figure 2.3: *The basic flow of a standard GA, using generational replacement (see Subsect. 2.5.4 below). The dashed arrow indicates that crossover is only carried out with a probability p_c.*

are **genetic algorithms** (GAs) [51], in which the variables of the problem are encoded in strings of digits which, when decoded, generate the system that is to be optimized (e.g. the brain of a robot represented, for example, by a neural network). Another version of EAs are **evolution strategies** (ESs) [5]. In a traditional ES, the variables of the problem were encoded as numbers taking any value in a given range (e.g. $[0, 1]$), whereas in traditional GAs, the digits were usually binary, i.e. either 0 or 1. There were other differences as well, such as the use of a variable mutation rate in ESs, whereas GAs typically used a constant mutation rate. However, GAs and ESs have gradually approached each other, and many practitioners of EAs today use real-valued encoding schemes and variable mutation rates in algorithms which they refer to as GAs.

Other versions exist as well, and they differ from the traditional GA mainly in the representations used. Thus, in **genetic programming** (GP) [68], the representation is usually a tree-like structure (even though a more recent version, linear GP, uses a representation closer to that used in GAs, further blurring the distinction between different algorithms). In **evolutionary programming** (EP) [38], a representation in the form of finite-state machines (see Appendix B) was originally used.

While there are many different versions of EAs, all such algorithms share certain important features - they operate on a population of candidate solu-

tions (individuals), they select individuals in proportion to their performance, and then apply various operators (the exact nature of which depends on the representation used) to form new individuals, providing a path of gradual, hereditary change towards the desired structure.

In evolutionary robotics, it is common to skip the step of encoding the information in strings of digits, and instead allow the EA act directly on the structure being optimized. For example, in the author's research group, EAs are often used that operate directly on recurrent neural networks (see Appendix A) or even on a behavioral selection system. The optimization of such systems will be discussed below. In general, a system being optimized by an EA will be called a **structure** whether or not it is obtained by decoding a chromosome. Thus, in a GA, one may optimize, say, a structure such as a neural network of fixed size, obtained from a chromosome. Alternatively, one may use a more generic type of EA, in which the encoding step is skipped, and the EA instead acts directly on the neural network.

2.4 Basic operation of evolutionary algorithms

Before discussing the various components of EAs, a brief description of the basic functionality of such algorithms will be given, centered around the example of function optimization using a standard GA. However, with only small modifications, the discussion is applicable to almost any EA. The basic flow of a GA is shown in Fig. 2.3. The first step of any EA (not shown in the figure), is to select a representation for the structures on which the algorithm will act. In the standard GA, the structures are strings of (binary) digits, known as **chromosomes**. An example of a chromosome is shown in Fig. 2.1. When decoded, the chromosome will generate the corresponding **individual**, i.e. the **phenotype** corresponding to the **genotype** given by the chromosome. In the case shown in Fig. 2.1, the individual simply consists of two real numbers x_1 and x_2, but more complex structures, such as e.g. ANNs, can also be encoded in chromosomes. In the case of ANNs, the real numbers obtained from the chromosome can be used for representing network weights.

As mentioned above, the encoding-decoding step is sometimes skipped in ER. However, for the remainder of this section, only encoding schemes making used of chromosomes, represented as linear strings, will be considered. Advanced topics, such as EAs operating directly on a complex structure (e.g. an ANN), or using structures of variable size, will be considered in Sect. 2.7.2.

Once a representation has been chosen, the next step is to initialize the chromosomes, which is normally done in a random fashion. Next, each chromosome is decoded to form the corresponding individual, which is then evaluated and assigned a fitness value, specified by the user. Consider the specific

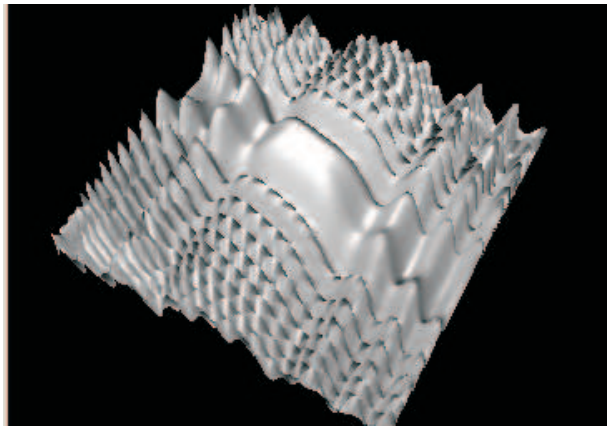


Figure 2.4: *The function ψ_{10}. The surface shows the function values obtained while keeping $x_3, x_4, \ldots, x_{10}$ fixed at zero, and thus only varying x_1 and x_2.*

case of function maximization, applied to the benchmark function

$$\psi_n(x_1, x_2, \ldots, x_n) = \frac{1}{2} + \frac{1}{2n} \exp\left(-\alpha \sum_{i=1}^{n} x_i^2\right) \sum_{i=1}^{n} \cos\left(\beta \sqrt{i} x_i \sum_{j=1}^{i} j x_j\right), \quad (2.1)$$

where α and β are constants. This function has a global maximum of 1 at $x_1 = x_2 = \ldots = x_n = 0$, and is illustrated in Fig. 2.4 for the case $n = 10$, and with $\alpha = 0.05, \beta = 0.25$. The figure was generated using the program `GA Function Maximizer v1.1`, which is provided on the CD associated with this tutorial. The fitness measure should be such that individuals that come close to the desired result receive higher fitness than those that do not. Thus, in the case of function maximization, a possible fitness measure is simply the function value itself[1]. Thus, for each individual, the variables $x_1, x_2, \ldots x_n$ are obtained from the chromosome, and the fitness value $f = \psi_n(x_1, x_2, \ldots, x_n)$ is computed.

Following the flow chart in Fig. 2.3, when all individuals have been evaluated, the production of new individuals can begin. Here, the first step is to select two (in case of sexual reproduction) individuals from the population, which will be used as parents for two new individuals. In general, the selection of individuals is performed in a fitness-proportional manner, thus favoring individuals with high fitness while at the same time allowing for the selection of less fit individuals. The details of various selection procedures will be given below in Subsect. 2.5.3. When two parents have been selected, two new individuals are formed, normally using **crossover** with a certain probability p_c (the **crossover probability**, in which the genetic material of the parents is mixed, and **mutation**, in which a small, random change is made to each of the two

[1] In case of minimization, the inverse of the function value can be used.

new chromosomes. The crossover and mutation operators are described in Subsects. 2.5.5 and 2.5.6 below. The procedure is repeated $N/2$ times, where N is the population size. The N new individuals thus generated replace the N individuals from which parents were selected, forming the second **generation**, which is evaluated in the same way as the first generation (see the flow chart). The whole procedure - evaluation, selection, crossover, and mutation - is repeated until a satisfactory solution to the problem has been found.

2.5 Basic components of evolutionary algorithms

Following the schematic example above, the main components of EAs will now be described in some detail. For a more detailed description, see e.g. [4], [84], and [88]. Note that the implementation of some operators, notably crossover and mutation, depends on the particular representation used.

2.5.1 Encoding schemes

The representation of the chromosomes can be chosen in several different ways. In the standard GA, **binary encoding** is used. Another alternative is **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 EAs, 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 directly available for binary encoding schemes. 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 ($[-d, d]$), according to

$$x = -d + 2dg, \tag{2.2}$$

assuming that the standard value $R = 1$ has been chosen for the range of allowed gene values. For standard binary encoding, the procedure is

$$x = -d + 2d\left(g_1 \times 2^{-1} + g_2 \times 2^{-2} + g_3 \times 2^{-3} + \ldots\right), \tag{2.3}$$

where g_i denotes the i^{th} gene used in the representation of the variable x. There are also other procedures for encoding numbers using binary chromosomes. An example is Gray coding, which will be discussed briefly in Sect. 2.7.1.

2.5.2 Fitness assignment

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 trigonometric function such as $f(x) = 1 + \sin(x)\cos(2x)$, 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.

It is therefore generally 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, which 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

$$f(i) = (N + 1 - R(i)). \tag{2.4}$$

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 less extreme fitness ranking, assigning fitness values according to

$$f(i) = f_{\max} - (f_{\max} - f_{\min}) \left(\frac{R(i) - 1}{N - 1} \right). \tag{2.5}$$

This ranking yields equidistant fitness values in the range $[f_{\min}, f_{\max}]$. The simple ranking in Eq. (2.4) 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, and the topic will be discussed further in Sect. 2.7.4.

2.5.3 Selection

Selection can be carried out in many different ways, two of the most common being **roulette-wheel selection** and **tournament selection**. All selection schemes preferentially select individuals with high fitness, while allowing selection of less fit individuals as well.

In roulette-wheel selection, two individuals are selected from the whole population using a procedure in which each individual is assigned a slice of a roulette wheel, with an opening angle proportional to its fitness. As the wheel is turned, the probability of an individual being selected is directly proportional to its fitness. An algorithm for roulette-wheel selection is to draw a random number r between 0 and 1, and the select the individual corresponding to the smallest value j which satisfies the inequality

$$\frac{\sum_{i=1}^{j} f_i}{\sum_{i=1}^{N} f_i} > r, \tag{2.6}$$

where f_i denotes the fitness of individual i.

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 takes 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.

2.5.4 Replacement

In the example described in the previous section, **generational replacement** was used, 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 par-

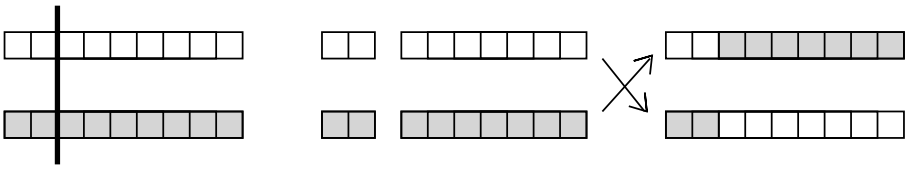


Figure 2.5: *Single-point crossover as implemented in a standard GA using fixed-length chromosomes. The crossover point is selected randomly.*

ents are considered for deletion, whereas in others, both parents and offspring are considered.

Deletion of individuals can be done 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.

2.5.5 Crossover

Crossover allows partial solutions from different regions of the search space to be assembled into a complete solution to the problem at hand.

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. 2.5. The procedure can be generalized to n**–point crossover**, where n crossover points are selected randomly, and the chromosome parts are chosen with equal probability from either parent. In **uniform crossover**, the number of crossover points is equal to the number of genes in the chromosome, minus one.

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**, in which only mutations are involved. p_{c} is commonly

chosen in the range $[0.7, 0.9]$.

2.5.6 Mutation

In natural evolution, mutation plays the role of 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 the computer run, and is thereafter left unchanged throughout the simulation. A typical value for the mutation probability is around $1/n$, 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, see e.g. [5].

In the case of chromosomes using binary encoding, mutation normally consists of changing the value of a gene to the complementary value, i.e. changing from 0 to 1 or from 1 to 0, depending on the value before mutation.

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

$$g \to g' = g - c + 2cr, \tag{2.7}$$

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

$$g \to g' = g(1 - c + 2cr), \tag{2.8}$$

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. (2.7) or Eq. (2.8), and then to encode the new number back into the chromosome.

2.5.7 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 or mutation. 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.

2.6 Genetic programming

As mentioned in Sect. 2.3, there exists many different types of EAs. In ER, one of the simplest methods of generating a robotic brain is to evolve the weights of an ANN (see Appendix A) of fixed size, using a GA with **direct encoding**, illustrated in Fig. 2.6. In such an approach the network cannot change size, and its success or failure depends entirely on the experimenter's ability to select an appropriate structure for the network.

More flexible approaches have also been developed, in which the size of the structure being optimized (be it an ANN or something else) is allowed to vary. Genetic programming (GP) is one such approach, and because of its frequent use in ER, a brief introduction will be given here. For a more thorough introduction, see e.g. [6] and [68].

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.

2.6.1 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,

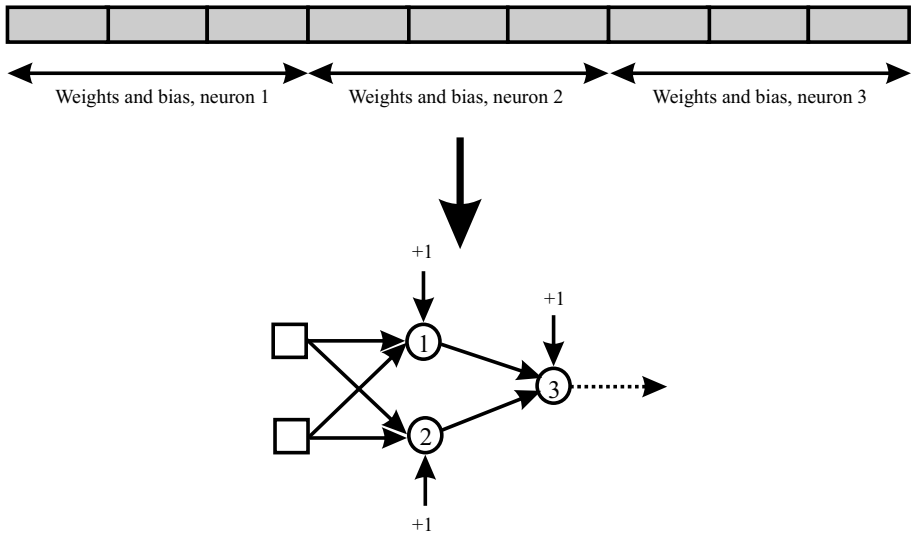


Figure 2.6: *Direct encoding of a simple ANN with three neurons and two input elements. The weights and biases (the latter shown as vertical arrows) are obtained directly from the chromosome, which is used simply as a lookup table. In this case, an ordinary GA would be an appropriate technique for optimizing the ANN.*

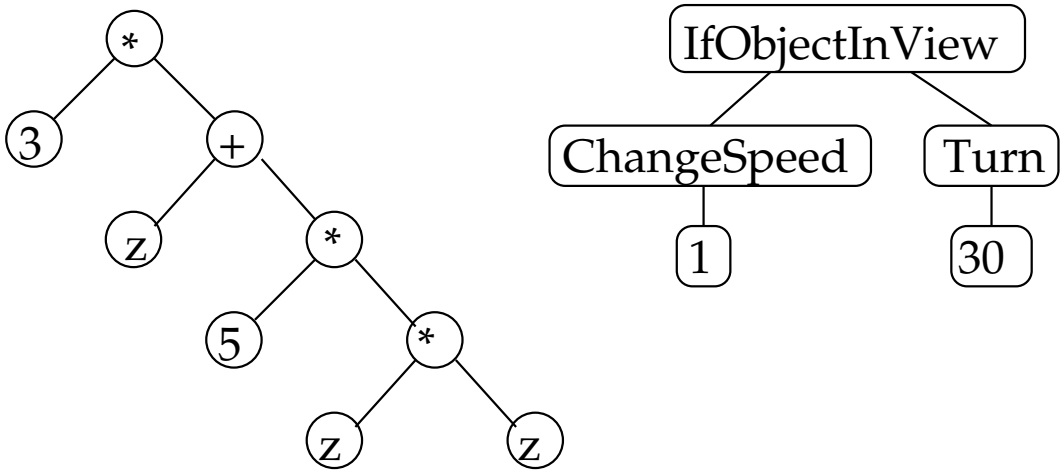


Figure 2.7: *Two GP trees. The tree to the left can be decoded to the function* $f(z) = 3(z + 5z^2)$*. The tree to the right tells a robot to increase its speed by one unit if it sees an object, and to turn by 30 degrees if it does not see any object.*

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 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. 2.7. 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. 2.8.

The mutation operator can change both terminals and elementary operators, but must be implemented in such a way as to maintain the syntactic correctness of the trees.

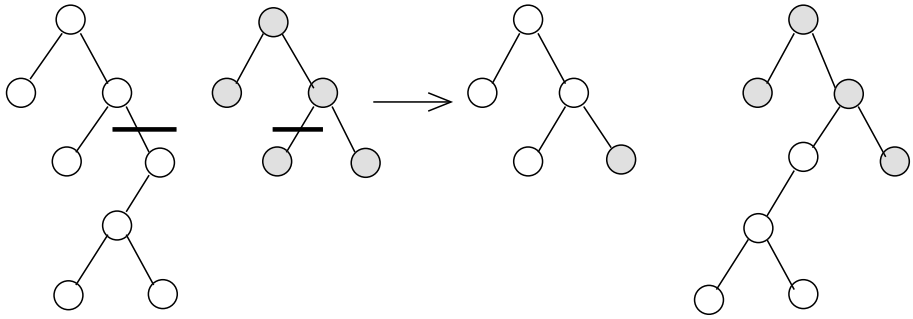


Figure 2.8: *The crossover procedure in tree-based GP.*

2.6.2 Linear genetic programming

Unlike tree-like GP, **linear GP** (LGP) [13] is used for evolving linear sequences of basic instructions defined in the framework of an **imperative programming language**.[2] 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;
```
(2.9)

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;
```
(2.10)

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

[2] 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 2.1: *Examples of typical LGP operators. Note that the operands can be either variable registers or constant registers.*

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

$$r_i := \begin{cases} r_j / r_k & \text{if } r_k \neq 0, \\ r_j + c_{\max} & \text{otherwise,} \end{cases} \tag{2.11}$$

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 2.1. 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;
```
(2.12)

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.

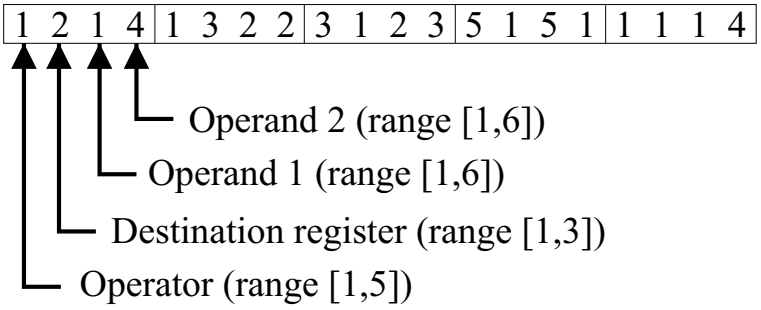


Figure 2.9: *An example of an LGP chromosome.*

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 2.2: *Evaluation of the chromosome shown in Fig. 2.9, 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.*

In LGP, chromosomes are used, similar to the those employed in a GA. An example of an LGP chromosome is shown in Fig. 2.9. 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. 2.9, there are three variable registers available (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

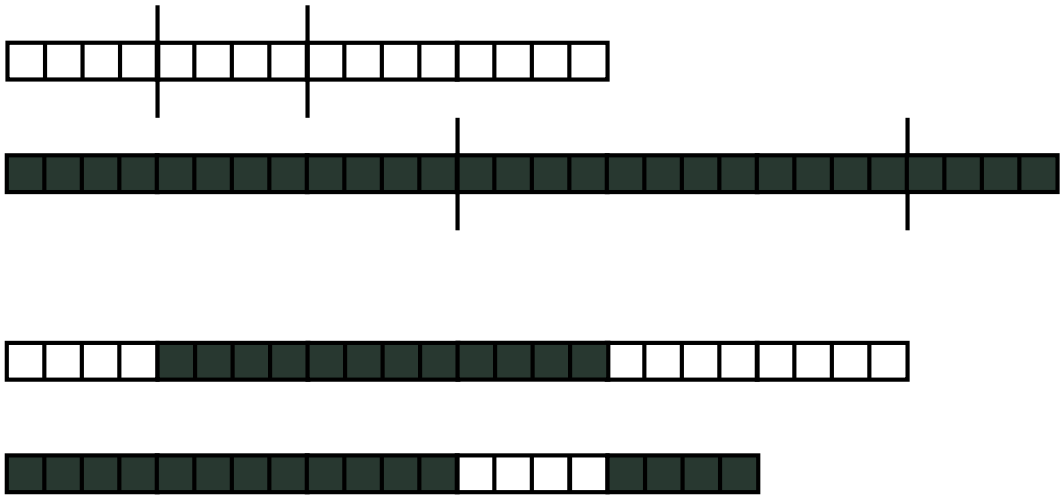


Figure 2.10: *An illustration of crossover in LGP. The two parent chromosomes are shown in the upper part of the figure, and the two offspring chromosomes are shown below. Note that two crossover points are selected in each chromosome.*

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. 2.9, 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 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. 2.9, in a case where r_1 (the input) was set to 1, is given in Table 2.2.

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. 2.10.

2.7 Advanced topics

In this section, a few advanced topics will be covered, with particular emphasis on topics relevant to practitioners of ER. In 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. The interested reader can find more information concerning advanced EA topics in e.g. [4].

2.7.1 Gray coding of binary-valued chromosomes

In the 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, even though the numerical difference between the decoded values obtained from the two chromosomes may be very small.

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 $00, 01, 11, 10$.

Of course, other 2-bit Gray code representations exist as well, for example $10, 11, 01, 00$ 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.

2.7.2 Variable-size structures

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 optimize structures of varying size. 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 Subsect. 2.7.5 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 artificial brains for autonomous robots, whether the evolving structure is a behavioral selection system (see Sect. 3.3), a neural network (see Appendix A) or a finite-state machine (see Appendix B) representing a single behavior, or some other structure. Evolutionary size variation can be implemented either by allowing chromosomes to take variable length or by letting the EA act directly on the structure (e.g. a neural network) being optimized.

In general, varying size should be introduced in cases where there is insufficient *a priori* knowledge of the optimal size of the structures being optimized.

In this section, two examples of encoding schemes will be given, starting with a description of encoding schemes for ANNs. In view of their frequent use in ER, ANNs occupy a central position in research concerning autonomous robots, and therefore the description of ANNs will be quite detailed. However, the reader should keep in mind that there are also many cases in which architectures other than ANNs are used in ER. For example, while ANNs are very useful for representing many individual behaviors in robots, other architectures are sometimes used when evolving behavioral selection (see Sect. 3.3).

Encoding schemes for neural networks

In the case of FFNNs (see Appendix A for the definition of network types) of given size, the encoding 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. 2.6. 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 in-

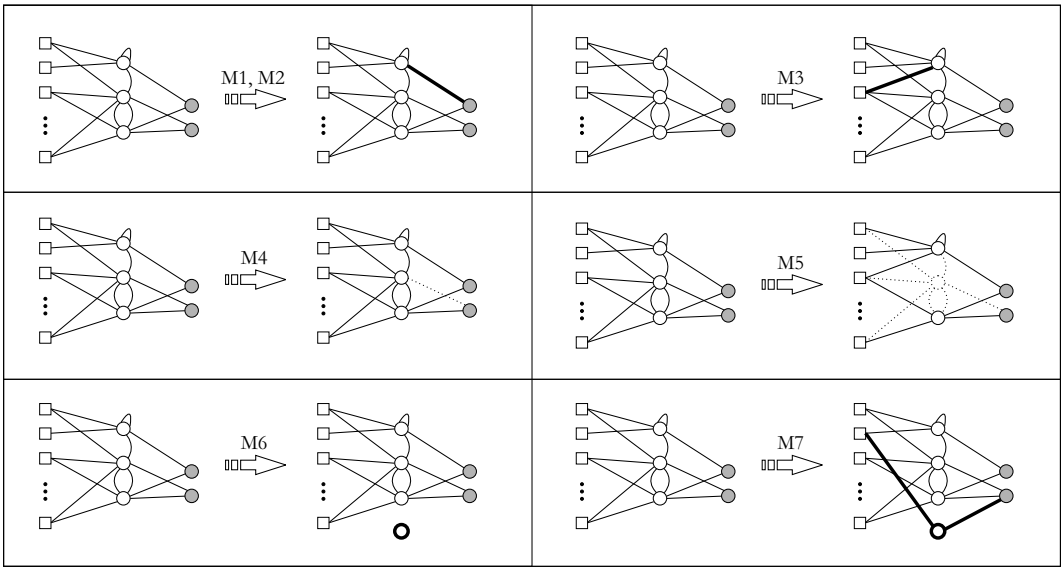


Figure 2.11: *Mutation operators (M1-M7). Modifications and additions are shown as bold lines and removed items are shown as dotted lines. The mutations are (M1-M2):* weight mutations*, either by a random value or a value centered on the previous value; (M3-M4):* connectivity mutations*, addition of an incoming weight with random origin (M3), or removal of an incoming weight (M4); (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). Reproduced with kind permission of Mr. J. Pettersson.*

stead 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 generation of ANNs using EAs has been considered extensively in the literature, see e.g. [146] for a review.

When generating brains for autonomous robots using EAs, there is rarely any reason to limit oneself to FFNNs. Instead, the EA is commonly used for optimizing completely general RNNs. Note, however, that the possibility of generating an FFNN is not excluded, since such networks are special cases of RNNs, and may appear as a result of the optimization procedure. When using an EA to optimize RNNs of varying size, some problems must be tackled. In particular, several mutation operators must be defined, which can modify not only the weights, but also the architecture (i.e. the structure) of the networks. A set of seven mutation operators (M1-M7) for RNNs is shown in Fig. 2.11. M1 and M2 modify the strengths of already present connections between units

(see Appendix A) in the network, whereas M3-M7 modify the architecture 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 an output neuron (shown as filled circles in the figure). Note that many other neuron addition operators can be defined.

In addition, crossover operators can be defined that can combine chromosomes 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 networks? 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.

Clearly, with this procedure, crossover can be performed with any two networks. 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, say, takes input from neurons 1, 4, and 5. If, during crossover, a single additional neuron is inserted between neurons 3 and 4, say, the inserted neuron will be the new neuron 4, and the old neuron 4 will become neuron 5 etc., thus completely changing the identities (and, therefore, the numerical values) of the weights, and also limiting the usefulness of crossover.

The problem of modified neuron identities can, of course, be mitigated by simply rewiring the network (after e.g. neuron insertion) to make sure that the identities of the neurons and their weights remain unchanged. However, there are also biologically inspired methods for mitigating this problem: Consider another type of network, namely a genetic regulatory network. 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, but they will not be considered further here.

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 networks of equal size. Consider a network of a given size, using an encoding scheme as that illustrated in Fig. 2.6. In averaging crossover, the value of gene x in the two offspring, denoted x'_1 and x'_2 is given by

$$x'_1 = \alpha x_1 + (1 - \alpha)x_2, \tag{2.13}$$

and

$$x'_2 = (1 - \alpha)x_1 + \alpha x_2, \tag{2.14}$$

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.

Grammatical encoding

The introduction of variable-size structures, as discussed above, adds considerable flexibility to an EA, and is crucial in the solution of certain problems.

Another motivation for the introduction of variable-size structures is the fact that such structures have more similarity with chromosomes found in natural evolution. However, as was discussed in Sect. 2.2, even the variable-size structures used in EAs 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 structures (e.g. linear chromosomes) used in EAs most often *do* use such direct encoding.

An example should be given to illustrate the state of affairs in biological systems. Consider the human brain. This very complex computer contains 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

Figure 2.12: *Grammatical encoding, as implemented by Kitano [66].*

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, even if the strength (and sign) of each connection weight could be encoded using a single byte. However, the actual size of the human genome is around 3 Gb. Furthermore, the genome 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 genome 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 [45], [66]. 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 [66] for encoding FFNNs. In that method, each chromosome is encoded in a string of the form shown in Fig. 2.12. 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 4 symbols: (ABCD, in the case shown in the figure)

$$S \rightarrow \begin{pmatrix} \mathrm{A} & \mathrm{B} \\ \mathrm{C} & \mathrm{D} \end{pmatrix}. \tag{2.15}$$

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 (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. 2.12 above, the results would be

$$A \rightarrow \begin{pmatrix} \mathrm{a} & \mathrm{f} \\ \mathrm{b} & \mathrm{a} \end{pmatrix}, \tag{2.16}$$

and

$$B \rightarrow \begin{pmatrix} \mathrm{c} & \mathrm{a} \\ \mathrm{d} & \mathrm{a} \end{pmatrix}, \tag{2.17}$$

Note that the total number of matrices of the kind shown in the right-hand sides of Eqs. (2.16) and (2.17) equal $16^4 = 2^{16} = 65,536$. The results of decoding the matrices in Eqs. (2.16) and (2.17) are given by

$$a \rightarrow \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \; b \rightarrow \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, \; \ldots \; p \rightarrow \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}. \tag{2.18}$$

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 1. 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. 2.12, 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.

Grammatical encoding schemes have been used in ER as well and some examples will be given in Subsect. 3.4.3 below.

2.7.3 Selection

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

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 [4]. In one version,

the selection of an individual from a randomly selected pair of individuals is based on the function b given by

$$b(f_{j_1}, f_{j_2}) = \frac{1}{1 + \mathrm{e}^{\frac{1}{T}(\frac{1}{f_{j_1}} - \frac{1}{f_{j_2}})}}, \tag{2.19}$$

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

$$j = \begin{cases} j_1 & \text{if } b(f_{j_1}, f_{j_2}) > r \\ j_2 & \text{otherwise} \end{cases} \tag{2.20}$$

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

$$p_j(f_j) = \frac{\mathrm{e}^{\frac{f_j}{T}}}{\sum_{k=1}^{N} \mathrm{e}^{\frac{f_k}{T}}}, \tag{2.21}$$

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.

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 [49]. 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 their 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 [132] 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 [132]) 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. However, there are some applications involving ER, and some references to those are given in Sect. 3.4.

2.7.4 Fitness measures

In many problems, e.g. function optimization, it is relatively easy to specify a fitness measure. However, in other problems, it is much more difficult. An example can be taken from the ER. Consider the problem of evolving a **gait** (i.e. a means of walking) for a simple model of a bipedal robot, namely the five-link robot shown in Fig. 3.24 in Subsect. 3.4.1 below, 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.

Furthermore, in ER, 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 should 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. Procedures for doing so will be given in Sect. 3.2.

Other complicated situations are those in which the fitness measure should take into account several, possibly conflicting, criteria (**multiobjective optimization**). Another example is **constrained optimization** where not all possible solutions are valid. Constrained optimization will not be considered further in this tutorial. The interested reader is referred to [84]. Multi-objective optimization will, at least implicitly, be considered in some of the examples in Chapter 3.

2.7.5 Alternative schemes for reproduction and replacement

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. For a more detailed discussion, see e.g. [4].

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 mate[3] and have offspring, or more correctly *fertile* offspring[4]. Speciation, i.e. the generation of

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

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

new species, occurs frequently 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 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 [22] 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*[5]. The *hox* gene complexes are a set of transcription factors 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 on 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.

In general, one of the main ideas behind the introduction of species is to avoid matings of very different individuals, since such matings often result in rather bad individuals, especially when the EA has been running for a while so that the parents have already started approaching (local) optima.

2.7.6 Subpopulation-based EAs

In **subpopulation-based EAs**, the population of N individuals is divided into N_s groups with $\nu = N/N_s$ individuals each. Mating is only allowed to take

[5]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 in the, a result of mutations in homeobox genes

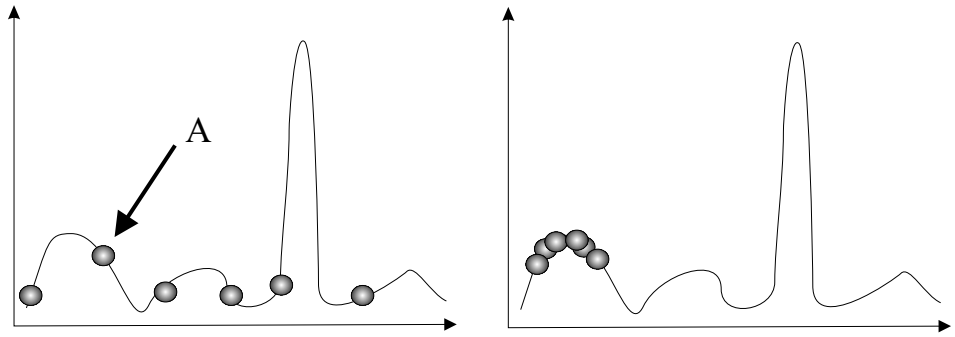


Figure 2.13: *Premature convergence in an EA. In the original population, shown in the left panel, one individual (marked A) has significantly higher fitness than the others. Thus, the genetic material of individual A will spread rapidly in the population, normally resulting in the situation shown in the right panel: Convergence to a local optimum.*

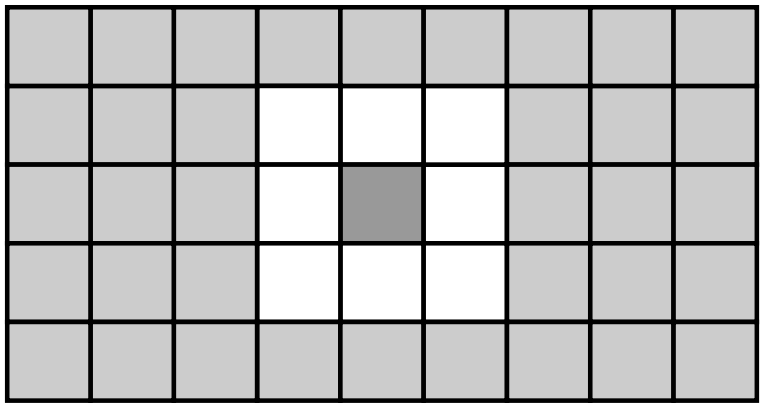


Figure 2.14: *Mating restriction in a grid-based EA. The selected individual, represented by the dark, central square in the figure, is only allowed to mate with one of its immediate neighbors, shown as white squares.*

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. 2.13. 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. Thus, in later generations, when a certain amount of convergence inevitably occurs, it is more likely to find an individual reaching the global maximum rather than a local one.

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, **tunneling** is allowed with some small probabil-

ity p_t. Tunneling can be achieved simply by swapping two indiviuals between subpopulations.

2.7.7 Grid-based EAs

In **grid-based EAs**, individuals are placed in a regular pattern as shown in Fig. 2.14, 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 rectangular grid are joined to form a toroidal space. Grid-based EAs are also called **diffusion models**.

2.8 Related algorithms

In view of their simplicity and general applicability, EAs have been applied successfully in many different problems involving optimization. However, during the last decade or so, several other stochastic optimization algorithms, with roughly the same applicability as EAs, have entered the stage. Two such methods are **ant colony optimization** (ACO) [26] and **particle swarm optimization** (PSO) [62]. Like EAs, both algorithms are population-based, i.e. a large number of candidate solutions to the problem at hand are maintained. In addition, both algorithms are biologically inspired: ACO is based on the foraging behavior of ants, whereas PSO is based on the flocking behavior of birds (or, equivalently, the schooling behavior of fish).

During foraging, ants communicate by means of **pheromones**, a chemical substance that indicates the recent presence of an ant. When an ant discovers a food source and then returns to the nest, other ants will be able to follow its pheromone trail to the same food source, thus further increasing the amount of pheromone along the trail. ACO operates essentially in the same way, using artificial pheromones. A typical application is the traveling salesman problem, where the objective is to find the shortest path visiting N cities (or nodes in a network) once and only once. In order to solve the problem, a group of artificial ants is released and given the task of moving between the N cities, depositing artificial pheromone as they go, and also selecting their moves depending on the level of artificial pheromone that they encounter. Once all ants have completed their task, a new group of ants is released etc. The level of pheromone deposited by an ant is then taken essentially inversely proportional to the length of the corresponding path, making this path a more likely

choice than a longer path for the next set of ants.

ACO is commonly more efficient than EAs on the traveling salesman problem. However, not all problems can easily be formulated in a way suitable for ACO. By contrast, PSO can be applied to any problem that EAs can solve. In PSO, a population of candidate solutions (referred to as *particles*) is generated, just as in an EA. In addition to keeping track of the position $(x_1, x_2, \ldots, x_K)$ of a given particle in the $K-$dimensional search space (just as one would do in an EA, where the position would be defined by the variables decoded from the chromosome of the individual), each particle is also given a velocity. Thus, rather than generating new individuals by crossover and (random) mutations, as in an EA, in PSO particles are moved (rather than replaced) by integrating their equations of motion. The acceleration of a given particle is determined by (1) the position of each particle in relation to the overall best position, i.e. the one associated with the highest value of the fitness function $f(x_1, x_2, \ldots, x_K)$ (assuming maximization) found so far, and (2) the position of each particle in relation to the previous best position of the particle itself. Thus, particles are drawn towards regions of higher fitness, while still exploring other regions on the way.

Even though both PSO and ACO are becoming increasingly popular in robotics, it is so far EAs that have found most applications in the field of autonomous robots, and the remainder of this tutorial will be focused on the application of EAs in robotics.

Chapter 3

Evolutionary robotics

3.1 Introduction

In this chapter the main topic of this tutorial, namely the generation of robotic bodies and brains by means of EAs will be considered. This is a vast and rapidly growing area of research, and therefore, by necessity, only a few aspects will be covered in detail in this tutorial. However, it is the author's hope that the topics covered with be representative of the field as a whole, and that the reference list at the end of the tutorial will provide useful directions for further reading.

ER has been applied in a large variety of different situations. Most applications, however, can be considered part of the subfield of robotics known as **behavior-based robotics**, and it is therefore appropriate to give a very brief introduction to this topic. The reader is referred to [3], and references therein, for a more thorough introduction.

3.1.1 Behavior-based robotics

The concept of behavior-based robotics (BBR) was introduced in the mid 1980s, and was championed by Rodney Brooks [14] and others. BBR approaches intelligence in a way that is very different from the classical AI approach, and a schematic illustration of the difference in given in Fig. 3.1. In classical **artificial intelligence** (AI), the flow of information is as shown in the left panel of the figure. First, the sensors of the robot sense the environment. Next, a (usually very complex) **world model** is built, and the robot reasons about the effects of various actions within the framework of this world model, before finally deciding upon an action, which is executed in the real world.

Now, this procedure, shown in the left panel of Fig. 3.1, is very different from the distributed form of computation found in the brains of biological organisms, and, above all, it is generally very *slow*, strongly reducing its survival

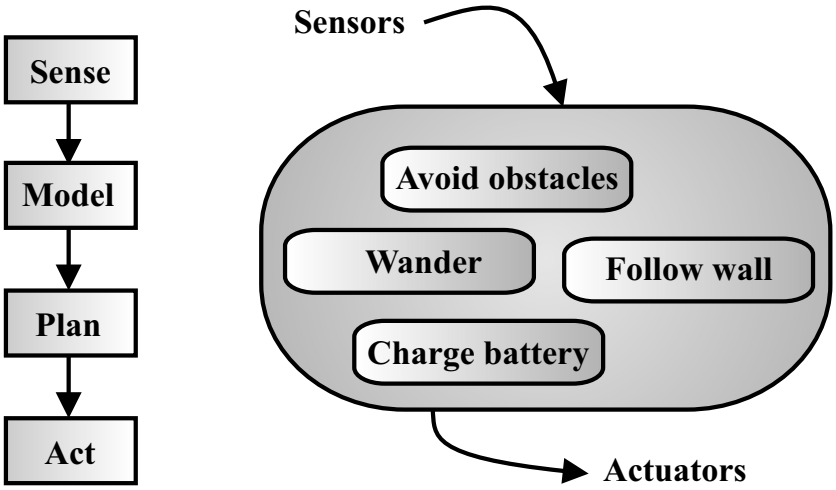


Figure 3.1: *A comparison of the information flow in classical AI (left panel) and in BBR (right panel). For BBR, any number of behaviors may be involved, and the figure only shows an example involving four behaviors.*

value. Clearly, this is not the way most biological organisms function. As a good counterexample, consider the evasive maneuvers displayed by noctuid moths, as they attempt to escape from a pursuing predator (e.g. a bat) [31]. A possible way of achieving evasive behavior would be to build a model of the world, considering many different bat trajectories, and calculating the appropriate response. However, even if the brain of the moth were capable of such a feat (it is not), it would most likely find itself eaten before deciding what to do. Instead, as will be discussed below, moths use a much simpler procedure.

As is evident from the left panel of Fig. 3.1, classical AI is strongly focused on high-level **reasoning**, i.e. an advanced cognitive procedure displayed in humans and, perhaps, some other mammals. Attempting to emulate such complex biological systems has proven simply to be too complex as a starting-point for research in robotics: Classical AI has had great success in many of the subfields it has spawned (e.g. pattern recognition, path planning etc.), but has made little progress toward the goal of generating truly intelligent machines, capable of autonomous operation. For a good introduction to AI, see e.g. [116].

BBR, illustrated in the right panel of Fig. 3.1 is an alternative to classical AI, in which intelligent behavior is built in a bottom-up fashion, starting from simple **behaviors** , many of which may be active simultaneously in a given robotic brain. Examples of simple behaviors are *avoid obstacles*, *pursue moving target*, *follow wall*, *find power supply* etc. An important task in behavior–based robotics is therefore to generate basic behaviors, and the construction of such behaviors will be one of the main topics in the remainder of this tutorial. First, however, an attempt should be made to define the concepts of **behaviors** and **actions**, since they are used somewhat differently by different authors. Here,

a behavior will be defined simply as a sequence of actions performed in order to achieve some goal. Thus, for example, an obstacle avoidance behavior may consist of the actions of stopping, turning, and starting to move again. Note, however, that the definition is not very strict, and that the terms *behavior* and *action* remain somewhat blurred.

Returning to the example of the moth, its evasive behavior is very simple indeed, and is in fact based on only a few neurons and an ingenious positioning of the ears on its body. This simple system enables the moth to fly away from an approaching bat and, if it is unable to shake off the pursuer, start to fly erratically to confuse the predator [31].

The example with the moth shows that **intelligent behavior** does *not* require reasoning[1], and in BBR one generally uses a more generous definition of intelligent behavior than that implicity used in AI. Thus, in BBR, one may define intelligent behavior as *the ability to survive, and to strive to reach other goals, in an unstructured environment*. This definition is more in tune with the fact that most biological organisms are capable of highly intelligent behavior in the environment for which they have been designed, even though they may fail quite badly in novel environments (as illustrated by the failure of e.g. a fly caught in front of a window). As mentioned in Chapter 1, an **unstructured environment** is an environment that changes rapidly and unexpectedly, so that it is impossible to rely on pre–defined maps, something that holds true for most real-world environment. For example, if it is the task of a robot to transport equipment in a hospital, it must first be able to avoid (moving) obstacles. Putting a complete plan of the hospital into the robot will not be sufficient, since the environment changes on a continuous basis. The robot may pass an empty room, only to find the room full of people or equipment when it returns.

Different behavior-based robots generally share certain features, even though not all features are present in all such robots. To start with, behavior-based robots are first provided with the most basic behaviors such as obstacle avoidance or battery charging, needed to function in an unstructured environment. More complex behaviors are added at a later stage. Secondly, several behaviors are generally in operation simultaneously, and the final action taken by the robot represents either a choice between the suggestions given by the various behaviors, or an average action formed from the actions suggested by several behaviors. Thirdly, BBR is mostly concerned with autonomous robots, i.e. robots that are able to move freely and without direct human supervision. Finally, the concept of **situatedness** is a central tenet of BBR: Behavior-based robots do not build complex, abstract world models. Instead, they read the information available through their sensors, and take actions based on that information. Thus, behavior-based robots are generally situated (i.e. operate in the real world), and many behaviors in BBR are **reactive** [95], i.e. have a di-

[1]The concept of rational behavior will be considered in Subsect. 3.3.3.

rect coupling between sensors and actuators. However, **internal states** are of course allowed and are often very useful, but, to the extent that such states are used, they are not in the form of abstract world models.

Note that, in ER, it is often necessary to use simulations rather than actual robots. While the use of simulations represents a step away from the requirement of situatedness, the simulations used in BBR differ strongly from the construction of abstract world models: Just as in a real, behavior-based robot, a simulated robot in BBR will rely only on the information it can gather through its simulated sensors, and the reading of a sensor, in itself, never provides explicit information about e.g. the distance to a detected object.

Clearly, no simulation can capture all the facets of reality. Thus, it is important to test, in real robots, any results obtained through simulations. The issue of evolution in simulations versus evolution in hardware will be considered briefly in Subsect. 3.4.2.

In general, the construction of a complete brain for a behavior-based robot can be considered a two-stage process: First the individual behaviors must be defined (or evolved, in the case or ER). Next, a system for selecting which action to take, based on the suggestions available from the individual behaviors, must be constructed as well. Clearly, in any robot intended for complex applications, the behavioral selection system is just as important as the individual behaviors themselves. For example, returning to the example of the hospital robot, it is clear that if it starts to run out of power, it must reprioritize its goals and quickly try to find a power supply, even if, by doing so, it comes no closer to achieving its task of delivering objects. Methods for **Behavioral organization**, (also called **action selection**, **behavioral coordination**, and **behavioral selection**) have been the subject of much research in BBR. While a complete survey of such methods is beyond the scope of this text, a few methods for behavioral selection based on evolutionary algorithms will be considered below in Sect. 3.3. For a review of methods for behavioral organization, see e.g. [108]. Other relevant references on this topic are e.g. [9], [78], and [131].

3.1.2 Evolving robots

The aim of ER is to use EAs to evolve robotic brains[2] (or bodies, or both), rather than designing them by other means, e.g. by hand. There are many reasons for using artificial evolution to generate robotic brains [46], [47], one of the most important being that it is very difficult to design such systems by hand in any but the simplest cases. In a non-controlled setting, i.e. any realistic environment, there will always be many sources of noise, as well as both

[2] As mentioned in Chapter 1, in this tutorial, the term **robotic brain** will be used instead of the term **control system**, since the latter term is indicative of the more limited types of systems considered in classical control theory.

stationary and moving obstacles and, perhaps, other robots as well. Trying to predict which situations may occur in such an environment is a daunting task, and hand-coded robotic brains are therefore generally non-robust and prone to failure. On the other hand, properly evolved robotic brains, i.e. those generated using either physical robots or simulations involving realistic noise at all levels, are often able to cope quite well with their environment, even though, in fairness, it should be mentioned that the robotic brains evolved so far are quite simple compared to the ultimate goal of truly intelligent machines. However, the complexity of evolved robotic brains is steadily increasing, and EAs are now used not only for constructing simple behaviors, but also for generating complex systems for behavioral organization [107], [131].

Furthermore, evolution (whether natural or artificial) is often able to find solutions that are remarkably simple, yet difficult to achieve by other means.

Another advantage with EAs in connection with robots, is that their ability to function even with very limited feedback. This is important in robotics, where it is often known *what* a robot should do, but perhaps not *how* it should do it, e.g. in what order different behaviors should be activated, at what point a task should be suspended in order to engage in self-preserving activities such as obstacle avoidance etc. In other words, it is difficult to specify directly the adequacy of any single action: Feedback is often obtained long after an action is performed, resulting in a **credit assignment problem**, i.e. the problem of assigning credit (or blame) to previous actions. EAs are very well suited to this kind of problems, where only rather vague guidance can be given. An example is the evolution of the simple cleaning behavior presented in Chapter 1: The simulated evolving robots were not told first to approach objects, then start pushing them, correcting the heading when needed etc. Instead, the only feedback available was the final state, and the intermediate steps were the result of evolution.

Of course, as with all methods, there are also some drawbacks with ER, one of the most serious being that, in view of the many candidate robotic brains that must be examined before a satisfactory one is found, it is usually required to resort to the use of simulations, rather than evolving in actual, physical robots. Making realistic simulations is indeed difficult, and there exists a **reality gap** [59] between simulated and physical robots. However, as will be discussed in Subsect. 3.4.2 below, it is possible to overcome the reality gap, and the need for using simulations does not, in fact, introduce fundamental limits on the results obtained from ER.

In the remainder of this chapter, several examples of ER will be given, along with a discussion of several general issues that appear in most ER applications. The flow of an ER investigation is shown in Fig. 3.2, and it is quite similar to the flow of a general EA. The figure is simplified: The procedure of forming individuals can involve anything from setting a few parameters to running a complex developmental process (see Subsect. 3.4.3 below), generating both

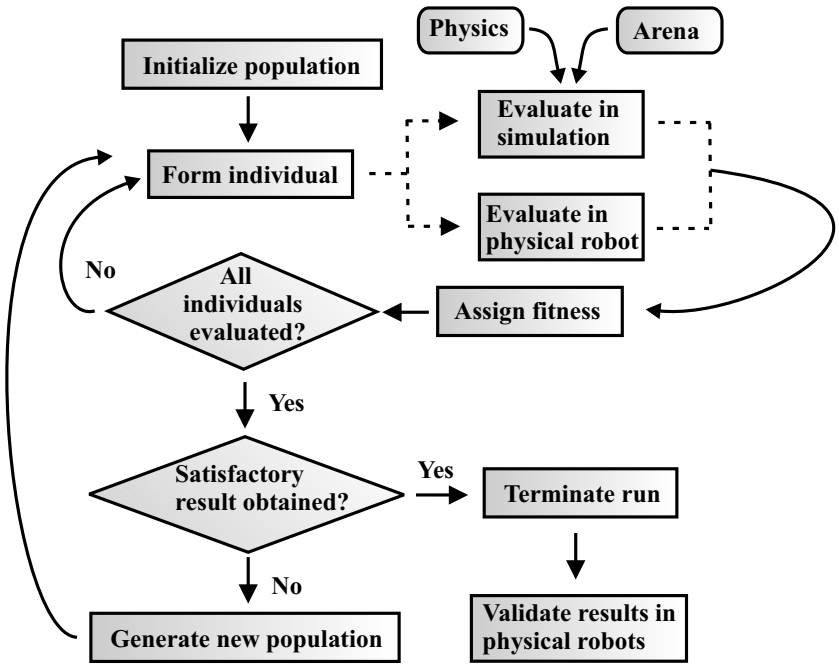


Figure 3.2: *The flow of an ER investigation. Note that some steps, e.g. formation of individuals and new populations, are shown in a strongly simplified way.*

the brain and the body of a robot. The dashed arrows in the figure indicate a choice: As mentioned above, in many ER investigations, simulations are used and, as indicated in the figure, the evaluation of an individual requires simulating the physics of the arena in which the evaluation takes place. The other option is to use actual, physical robots, and, in this case, the most common approach is to generate individuals in a computer, upload them one by one on a single robot, evaluate the robot, and return its fitness. There are other approaches as well (not shown in the figure), such as **embodied evolution** [33], [140], where the entire evolutionary process takes place on a population of physical robots. Incidentally, if evolution is carried out in physical robots according to the scheme shown in Fig. 3.2, some restrictions are placed on the fitness measure, since it must be possible for the robot to assess its fitness and return the result to the computer, at least in cases where the EA is to operate without continuous human supervision. Thus, the fitness measure must be based on quantities that are readily available to the robot, such as e.g. sensors readings, motor speeds etc. [34].

The final step in Fig. 3.2, i.e. the validation of the results in physical robots, is perhaps the most important step. A robotic brain that only functions in simulation is of little use. Note that validation should be carried out even if the evolution has been performed using a single physical robot. This is so, since all physical robots have individual characteristics. For example, there is

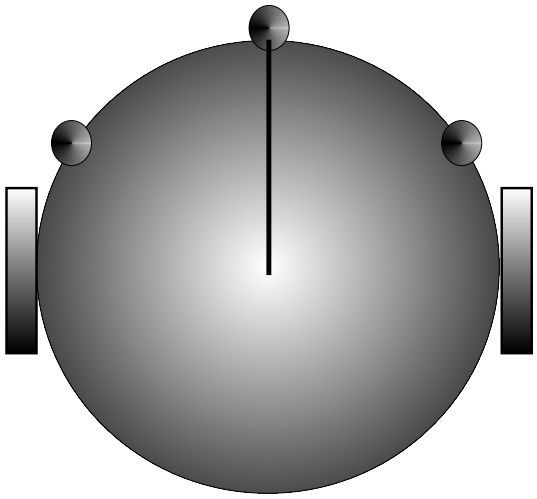


Figure 3.3: *A schematic illustration of a differentially steered robot, equipped with three sensors. The wheels, shown as rectangles, are independently controlled. The vertical line indicates the direction of motion (for positive wheel speeds).*

no such thing as two completely identical IR sensors, even if they come from the same manufacturer and have the same product number.

If the validation does not give satisfactory results, there are two different options, either to extend the EA run, or to attempt to adjust the robotic brain (or body) by hand. A useful approach in extended runs may be to switch from simulations to physical robots. Even a short extended run may then lead to improved results [86].

The architectures used for robotic brains can be chosen in a variety of ways. It is common to use ANNs (see Appendix A), but other architectures are used as well, as will be discussed below.

Robots used in ER investigations are of many kinds, including **walking robots** (see Subsect. 3.4.1) and **wheeled robots**. A common type of wheeled robot for ER is the Khepera robot (see Appendix C), for which there also exists several simulators (see Subsect. 3.4.4). In general, differentially steered robots, such as Khepera, are also simple to model and thus to use in simulations. The basic equations of motion for such robots can be taken as

$$M\dot{v} + \alpha v = A\left(\tau_{\mathrm{L}} + \tau_{\mathrm{R}}\right), \tag{3.1}$$

and

$$\overline{I}\ddot{\varphi} + \beta\dot{\varphi} = B\left(-\tau_{\mathrm{L}} + \tau_{\mathrm{R}}\right), \tag{3.2}$$

where M is the mass of the robot, $\overline{I}$ its moment of inertia (with respect to the vertial axis), v its speed, and φ its direction of motion. τ_L and τ_R are the torques acting on the left and right wheels, respectively. A and B are constants depending on geometrical and physical factors (such as the radius and mass

of a wheel). A schematic view of a differentially steered robot, equipped with 3 sensors, is shown in Fig. 3.3.

3.2 Evolving single behaviors

In this section, some examples of the evolution of single behaviors will be given. However, already at this stage, it can be noted that the definition of behaviors is somewhat fuzzy. For example, in the first example below, i.e. the evolution of navigation, it is normally required that the robots be able not only to move, but also to avoid collisions. Thus, the question of whether to consider motion without collisions as a single behavior or as a combination of two behaviors arises. However, in this particular example, the two aspects of the robot's behavior - motion and obstacle avoidance - are so closely integrated with each other, and can rather easily be achieved through a clever choice of fitness function, that they can perhaps be considered as part of a single behavior.

While the examples are hopefully interesting in their own right, the reader should pay attention to a few particularly important aspects of any application of ER, namely (1) the representation used for the evolving systems, (2) the level of complexity of the simulator (if applicable), and (3) the fitness measure used in the simulations. In general, the choice of representation (i.e. whether to use, say, direct encoding or grammatical encoding, in the case of neural networks) and the choice of fitness measure have a very strong impact on the results of an ER investigation: If the representation is chosen badly, the EA may be hampered in its search for a good solution, and if the fitness measure is chosen badly, the results of the evolution may be different from the desired results. In simulations, the level of complexity of the simulator influences the possibility of transferring the results of a simulation onto a real robot.

3.2.1 Navigation

Navigation, i.e. the problem of moving towards a goal without colliding with obstacles, is clearly a basic competence of any robot, and it has been studied extensively in the ER literature [34], [87], [117]. For a review, see [97].

Using a broad definition of navigation, the concept may also include behaviors such as wandering, i.e. motion without a specific goal. The first two subsections below will consider this type of navigation. More advanced forms of navigation require that the robot should know its position. Such applications are considered in the subsection on path planning below.

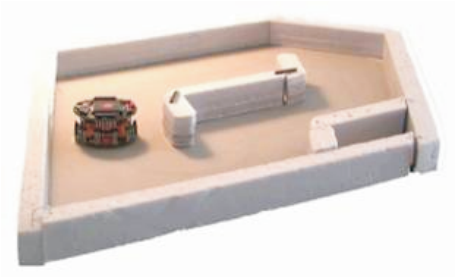


Figure 3.4: *The environment used in the evolution of navigation in [34]. Reproduced with kind permission of Dr. D. Floreano.*

Basic navigation

In [34], Floreano and Mondada evolved basic navigation. The authors specifically pointed out the difficulties in constructing accurate simulations, and chose instead to evolve navigation behavior in a real Khepera robot (see Appendix C). The aim was to evolve collision-free navigation in an environment with stationary obstacles, shown in Fig. 3.4. In the experiments, the authors used a simple, fixed-length chromosome to encode the weights of neural networks of fixed structure. The networks consisted of a single layer of synaptic weights, connecting the 8 IR sensors of the Khepera robot to two output neurons, connected to the motors. In addition, recurrent connections were introduced within the output layer.

The fitness contribution for each time step during the motion of the robot was chosen as

$$\Phi = V\left(1 - \sqrt{|\Delta v|}\right)(1 - i), \tag{3.3}$$

where V denotes the average rotation speed of the wheels, Δv is the difference between the (signed) speeds of the wheels, and i is the value of the IR sensor with the highest reading. V, $|\Delta v|$, and i were all normalized to the range $[0, 1]$, so that Φ also was constrained to this range. The complete fitness function f was obtained by summing the values of Φ obtained for each time step, and then dividing the results by the number of time steps. The same fitness measure was used also by Nolfi et al. [99].

The first factor (V) in Eq. (3.3) promotes high speed, whereas the second factor, $1 - \sqrt{|\Delta V|}$, promotes straight-line motion, and the third factor $(1 - i)$ rewards obstacle avoidance. Note that the different factors in the fitness measure counteract each other, to some extent. For example, the environment was such that the robot was required to turn quite often in order to avoid collisions, even though the second factor in the fitness measure would discourage it to do so. This is an example of what could be called an **explicit fitness measure**, since explicit punishments (in the form of a reduced increase in fitness) are given if the speed is low, if the robot does not move in a straight line, or if

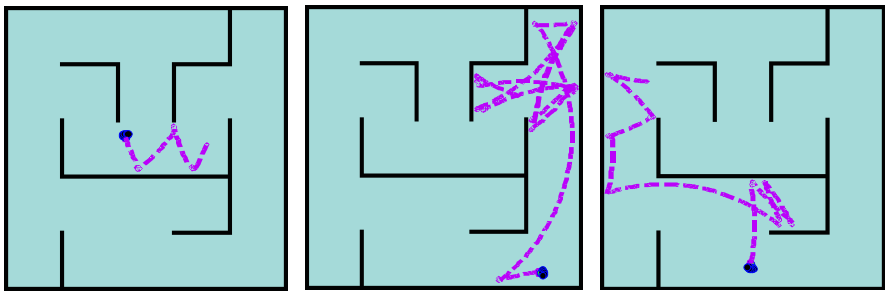


Figure 3.5: *The maze used by Nelson* et al. *[92] for the evolution of navigation, reproduced with kind permission of the authors. The figure shows some results obtained in simulation. Qualitatively similar results were obtained using a physical robot, EvBot, shown in Fig. 3.6.*

the robot approaches an obstacle. The alternative is to use an **implicit fitness measure** for *some* of the aspects of the robot's motion. For example, in [59], Jakobi et al. used a fitness measure similar to the one given in Eq. (3.3), but without the factor $(1 - i)$, since it was found that this factor is unnecessary in a sufficiently cluttered environment, where obstacle avoidance is an implicit requirement for any form of motion.

Floreano and Mondada report that the best evolved robots successfully navigated through their environment, and that some robots also showed intelligent behavior beyond what was explicitly encoded in the fitness function. For example, some of the best robots learned to modulate the speed, keeping it at around three quarters of the maximum allowed speed in the vicinity of obstacles.

While evolving robotic behaviors in hardware has obvious advantages, a possible disadvantage is that it is often quite time-consuming. Indeed, in the experiments reported in [34], each generation lasted around 40 minutes. In order to evolve successful robots, around 50-100 generations were needed.

In addition to basic navigation, Floreano and Mondada also evolved *homing navigation*, in which the robot was required periodically to recharge its (simulated) batteries. However, this was a more complex task, and it will be considered briefly in the discussion on complex behaviors and behavioral organization in Sect. 3.3 below.

While some recurrent connections were included in the output layer of the neural networks used in [34], Nelson *et al.* [92] made more explicit use of temporal processing of information, motivated by the fact that robots commonly show suboptimal performance using reactive behaviors based on very poor sensory inputs.

Thus, Nelson *et al.* considered neural networks with several layers containing recurrent couplings as well as time-delayed couplings. The neural networks were evolved in simulations involving robots equipped with 5 simple

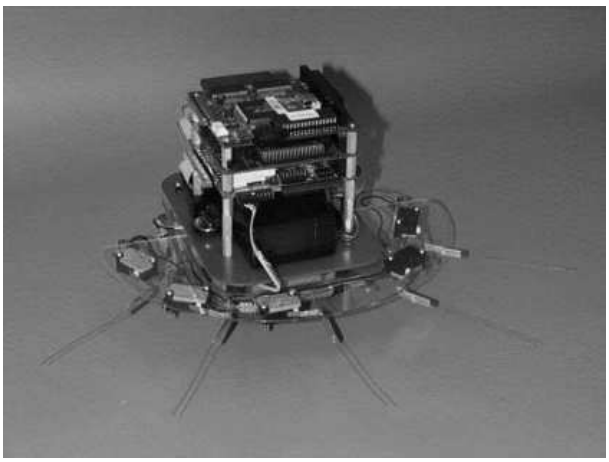


Figure 3.6: *The EvBot robot used in [92]. The robot is equipped with an array of tactile sensors, and can also be fitted with other sensors, such as video cameras. Reproduced with kind permission of Nelson* et al.

tactile sensors. The task of the simulated robots was to navigate in a maze, shown in Fig. 3.5. The fitness measure was taken essentially as the distance travelled, with a penalty for situations in which the robot became stuck.

It was indeed found that robots equipped with memory, in the form of recurrent couplings and time-delay elements, performed the navigation task better than purely reactive robots. The best results were obtained for networks of moderate complexity, containing 1-3 hidden layers with 5-10 neurons per layer. The best networks obtained during evolution were transferred, with retained functionality, to an actual robot (EvBot, shown in Fig. 3.6), equipped with five tactile sensors.

Wandering behavior

Miglino et al. [87] evolved an exploratory behavior (also called *wandering*) in simulations. The resulting behaviors were then tested in actual (Lego) robots. In this case, the brains of the robots were represented by very simple neural networks, with two inputs from a front sensor and a rear sensor, two hidden units, and two outputs controlling the two motors. In addition, a single memory neuron was used, connecting the hidden layer to itself through a recurrent coupling. All signals were binary (a Heaviside step function was used as the threshold function in the neurons), and the simple neural networks used in this study could, in fact, be represented as finite-state machines (see Appendix B).

In order to achieve wandering behavior, the authors used a fitness measure which rewarded movements that would take the robot to locations where it had not been before. The robot was placed in an arena of size 2.6×2.6 meters,

consisting of squares with 0.10 m side length. The central 20×20 square had white color, and the remaining squares were black, so that the robot could determine whether or not it was close to a boundary.

The fitness of the evolving robotic brains was incremented by one every time the robot visited a square it had not previously visited, and the final fitness measure was taken as the fraction of squares visited.

An important issue in ER is the fact that an EA will try to exploit the particularities of the encountered situations as much as possible. Thus, in the evolution of wandering behavior, if a single starting position was used in all evaluations, the EA would quickly find a way to optimize the motion of the robot based on the given starting position. In order to avoid such problems, Miglino et al. evaluated each robotic brain 10 times, starting from a random location in the grid.

The authors report that the best evolved robotic brains managed to make the corresponding robot visit an average of slightly more than half of the available squares. The evolutionary process exhibited three clear stages, during which the maximum fitness was essentially constant. The stages corresponded to different levels of complexity. For example, in the first stage, the evolved neural networks made no use of the memory unit, whereas in the second stage they did.

The neural networks obtained in the simulations where then implemented in actual Lego robots. While the behaviors of the actual robots were similar to those of the simulated ones, there were some clear differences, some of which were attributed to noise (absent in the simulations). Thus, some simulations were made with noise added, and it was found that a closer correspondence between the behavior of real robots and simulated robots could be found if the latter were evolved at intermediate levels of noise.

Path planning

Path planning, the problem of determining the optimal route from a starting position to a goal position, has been studied extensively in robotics in general, and also, to some extent, within the field of ER. A basic requirement for path planning is **localization**, i.e. knowledge of one's position. In many ER-related path planning studies, it is simply assumed that the localization problem has been solved, and the investigation instead centers on the problem of finding suitable paths that avoid collisions with obstacles. However, some authors have considered the problem of localization. For example, Duckett [27] describes an EA-based method for simultaneous localization and mapping (SLAM) and Pettersson et al. [105] recently developed a localization algorithm based on odometry, where the duration of execution of the behavior for odometry calibration (based on laser range finder scan matching) is essentially determined by an EA. With the assumption of sufficiently accurate

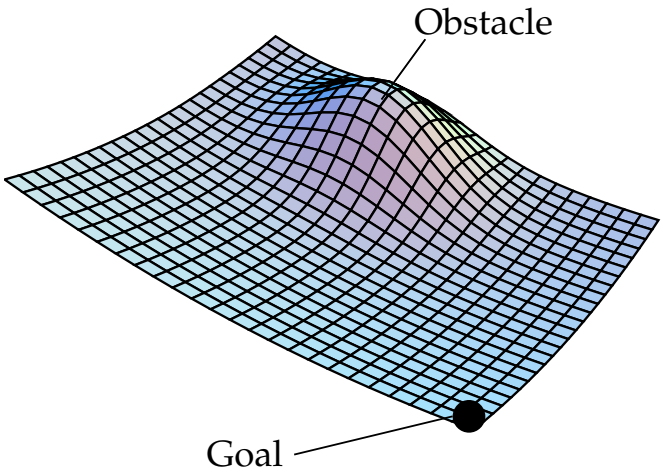


Figure 3.7: *A potential field containing a single obstacle and a navigation goal.*

localization, Sugihara and Smith [123] used an EA to evolve path planning both for two-dimensional and three-dimensional movements, dividing the environment into grid cells. Geisler and Manikas [43] describe an EA-based method for local obstacle avoidance, whereas Erinc and Carpin [29] considered the problem of path planning for non-holonomic robots (i.e. robots with non-controllable degrees of freedom), such as car-like robots. Another approach to path planning, which will now be described in some detail, is to use a potential field.

Potential-field based navigation In [117], an EA was used in connection with **potential field navigation** [64] to evolve (in simulation) robots capable of navigating toward a goal in an environment containing stationary obstacles.

In the potential field navigation method, the robot moves in the direction suggested by the (negative) gradient of an artificial potential field, generated by the objects (such as obstacles) in the vicinity of the robot and by the navigation goal. As shown in Fig. 3.7, the potential field can be interpreted as a landscape with hills and valleys, and the motion of the robot can be compared to that of a ball rolling through this landscape. The navigation goal is usually assigned a potential field corresponding to a gentle down-hill slope, e.g.

$$\Phi^{\mathrm{g}} = k_{\mathrm{g}} \left(\mathbf{x} - \mathbf{x}_{\mathrm{g}} \right)^2 , \tag{3.4}$$

where k_{g} is a positive constant, $\mathbf{x}$ the position of the robot, and $\mathbf{x}_{\mathrm{g}}$ the position of the goal. Continuing with the landscape analogy, obstacles should generate potentials corresponding to steep hills. Thus, the potential of an obstacle can

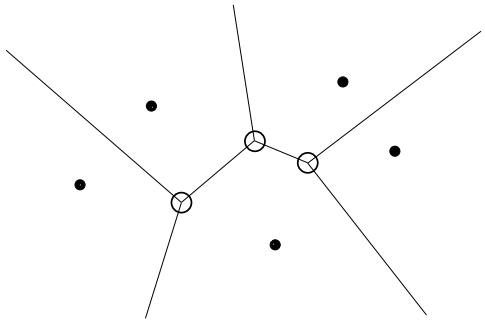


Figure 3.8: *A Voronoi diagram. The black dots represent the centers of obstacles, and the rings at the vertices of the resulting Voronoi diagram represent the locations of waypoints.*

be defined as

$$\Phi^{\mathrm{o}} = k_{\mathrm{o}} \mathrm{e}^{-\frac{(\mathbf{x}-\mathbf{x}_{\mathrm{o}})^2}{w_{\mathrm{o}}^2}}, \tag{3.5}$$

where k_{o} and w_{o} are constants, determining the height and width of the obstacle, respectively, and $\mathbf{x}_{\mathrm{o}}$ is the position of the obstacle. The total potential is given by

$$\Phi = \Phi^{\mathrm{g}} + \sum_{i=1}^{n_{\mathrm{o}}} \Phi_i^{\mathrm{o}}, \tag{3.6}$$

where n_{o} is the number of obstacles. Once the potential field has been defined, the desired direction of motion $\hat{\mathbf{r}}$ of the robot can be computed as the negative of the normalized gradient of the field

$$\hat{\mathbf{r}} = -\frac{\nabla\Phi}{|\nabla\Phi|}. \tag{3.7}$$

Thus, the potential field provides the desired direction of motion of the robot. In addition, the speed of the robot must be set. In [117], this was achieved using a pre-specified set speed and a simple proportional control law.

In a complex environment, a robot using simple potential field navigation will often get stuck, due to the presence of local minima in the potential field: Since the method is gradient-based, the robot will be unable to escape from such minima. The problem can be solved by the introduction of waypoints, i.e. local goals (attractive potentials) along the path of the robot. It is not trivial, however, to select the location of waypoints and the exact shape of waypoint potentials, in order to arrive at efficient, yet collision-free, navigation.

In [117], the placement of waypoint was based on **Voronoi diagrams**. Briefly summarized, in the generation of a Voronoi diagram the obstacles are considered as point-like objects, and are taken as central points (Voronoi generators) for the spatial tessellation. Next, polygons are shaped by drawing lines perpendicular to the lines connecting Voronoi generators, and the corners of the

resulting polygons are taken as the waypoint locations. The procedure is illustrated in Fig. 3.8.

The potentials of the waypoints can be modelled e.g. as

$$\Phi^{\mathrm{p}} = k_{\mathrm{p}} \mathrm{e}^{-\frac{(\mathbf{x}-\mathbf{x}_{\mathrm{p}})^2}{w_{\mathrm{p}}^2}}, \tag{3.8}$$

i.e. the same form as the obstacle potentials, but with *negative* constants k_{p}. When the waypoints have been placed, and their potentials have been determined, navigation proceeds as in standard potential field navigation, with the direction of motion provided by the potential field, the only difference being that waypoints are successively removed as the robot passes in their vicinity to prevent it from being attracted back toward waypoints that have already been passed.

Once the location of obstacles in a given environment is known, the location of waypoints is generated deterministically through the procedure just described. What remains to be determined is the depth (or height, in the case of obstacles) of the potentials for the goal, obstacles, and waypoints, as well as the widths of the potentials (i.e. w_{o} for obstacles etc.). In [117], the authors used an EA to determine suitable values for these parameters. In addition, the reference speed used in the proportional control law was determined by the EA. In fact, two reference speeds were used, v_{ref}^1 (for general navigation) and v_{ref}^0 (near obstacles). Furthermore, the EA determined the distance d_{o} from the closest obstacle at which a simulated robot would lower its set speed from v_{ref}^1 to v_{ref}^0, as well as the distance d_{w} (between a waypoint and the robot) at which the waypoint was removed.

Thus, in all, a total of 10 parameters had to be determined, and, for this purpose, the authors used a standard genetic algorithm (GA) with tournament selection, single-point crossover, and mutation.

As in the navigation of wandering behavior discussed above, the authors evaluated each set of parameters several times, in order to avoid overfitting to any particular environment.

For each evaluation, the fitness measure was calculated as

$$f_i = \frac{T_{\max}}{T} \mathrm{e}^{-\frac{d}{D}}, \tag{3.9}$$

where T denotes the time at which the robot reached the navigation goal, at which point the evaluation would be terminated. In addition, all evaluations had a *maximum* duration of $T_{\max}$, whether or not the robot reached the goal. Furthermore, if the robot physically hit an obstacle, the evaluation was terminated immediately and T was set to $T_{\max}$. d denotes the distance between the robot and the goal at termination, and D is the initial distance between the robot and the goal. With this fitness measure, the robot was rewarded for moving quickly, and without collisions, toward the goal.

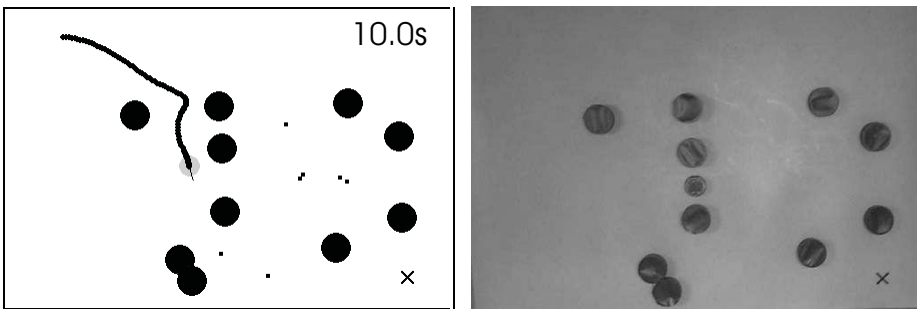


Figure 3.9: *The navigation of a simulated robot from [117] is shown in the left panel. The snapshot was taken after 10.0s in a simulation in which the goal was reached after 29.8s. In the right panel, a photograph of the actual Khepera II robot, navigating in the same environment, is shown. The robot, seen from above, is the smaller circle near the center of the photograph.*

In all, 20 different environments, with different locations of obstacles, were used in the evaluation of each individual (i.e. parameter set), and a final fitness measure was formed by weighing the results of the $N_{\mathrm{e}} = 20$ evaluations. Now, a common approach for doing so is to consider the average fitness, i.e.

$$f^{(1)} = \frac{1}{N_{\mathrm{e}}} \sum_{i=1}^{N_{\mathrm{e}}} f_i, \tag{3.10}$$

The authors report that the runs with fitness measure $f^{(1)}$ achieved only mediocre results. This is understandable, as the fitness measure $f^{(1)}$ only measures average performance, therefore making it possible for the simulated robot to fail completely in some environments and still achieve a rather high average. By contrast, fitness measure $f^{(2)}$

$$f^{(2)} = \min_i f_i, \tag{3.11}$$

which was also used, focuses completely on the worst performance of the robot, and tends to generate results that are more robust. In fact, in *all* runs with this fitness measure, the simulated robot with the motion determined by the best individual of the EA managed to reach its goal position in all environments.

Next, the results from the simulations were transferred to an actual Khepera II robot (see Appendix C). Clearly, the potential field navigation method requires accurate positioning, and it turned out that the drift in the deviation between actual and estimated position (based on dead reckoning) was too large if the dynamics of the robot was integrated as in the simulations. Thus, a simplified scheme was implemented, in which the robot navigated in discrete steps. In each step the robot started from a standstill, determined (via

the potential field) its desired direction of heading, moved a distance δ in this direction, and stopped again. Provided that δ was chosen sufficiently small, this navigation procedure represents a slow-motion version of that used in the simulations. Using this procedure, the Khepera robot rather successfully (but slowly) navigated through one of the most difficult environments used in the simulations. Snapshots of a simulated robot and a physical robot navigating in this environment are shown in Fig. 3.9.

3.2.2 Box-pushing

Simple *Box-pushing* can be used as a metaphor for behaviors needed, for instance, in transportation robots, and it has been investigated by several authors, e.g. [71], [120], [121], [144]. The results from two such investigations will now be discussed briefly.

Single-robot box pushing

Lee et al. [71], evolved box-pushing in simulations, and transferred their results onto a Khepera robot. In order to reduce the differences between simulations and real-world implementations of box-pushing, the authors used the interesting approach of sampling the *actual* sensors on the Khepera, storing the results in lookup tables that were then used in the simulations. In the simulations, the task of the robot was to push a small box towards a light source.

In fact, the simulations reported in [71] involved the evolution of organization of two behaviors, namely *box-pushing* and *box-side-circling*. The latter behavior was used in order to place the robot in the correct direction, so that it could use the *box-pushing* to move the box toward the light source. Here, however, only the box-pushing behavior will be discussed, behavioral organization being deferred to Sect. 3.3.

Lee et al. used GP to evolve tree-like structures achieving box-pushing, using the readings of the 8 IR sensors on the Khepera as input. The authors used the measure

$$e = \sum_{t=1}^{T} \alpha\,(1 - s(t)) + \beta\,(1 - v(t)) + \gamma w(t), \tag{3.12}$$

where $s(t)$ denotes the average of the normalized activations of the two front sensors (sensors 2 and 3, see Appendix C) on the (simulated) Khepera robot, $v(t)$ the normalized forward speed of the wheels, and $w(t)$ the normalized speed difference between the two wheels. Note that the measure e should be minimized, and is thus an error measure rather than a fitness measure in the traditional sense. However, an EA can of course easily be modified to strive to

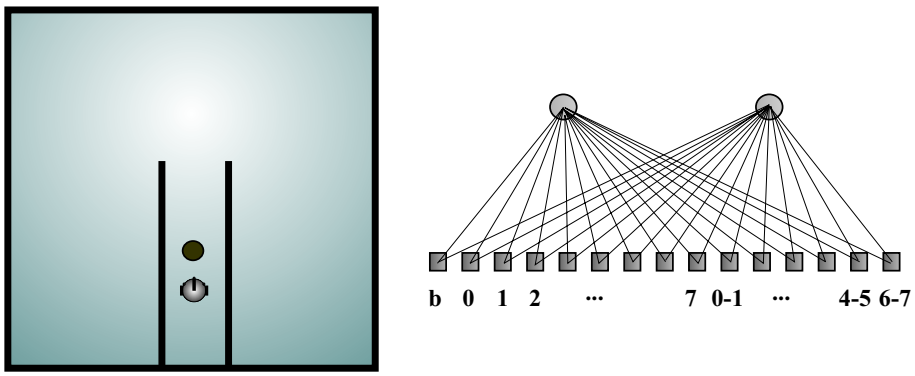


Figure 3.10: *The left panel shows a schematic view of the environment used by Sprinkhuizen et al. [121] in the evolution of box-pushing behavior. Note the light source placed at the end of the corridor. The right panel shows the structure of the FFNNs used.*

minimize an objective function. The measure e penalizes low activation of the front sensors (i.e. loosing the object), low speed, and curved motion.

The EA used by Lee et al. made use of subpopulations (see Subsect. 2.7.6) in order to maintain the diversity of the population, and thus to avoid costly computations of almost identical individuals. Successful box-pushing behavior was achieved in 50 generations, using a total population size of 100.

Sprinkhuizen-Kuyper et al. [121] evolved box-pushing in simulations, using a Khepera simulator involving a slightly more complex environment (see the left panel of Fig. 3.10) containing walls. In this case a simple, one-layer, FFNN was used to represent the brain of the simulated robots. However, in addition to the 8 input signals provided by the IR sensors and a bias signal, the authors added six simple **edge detectors**, consisting simply of the differences between the readings of adjacent sensors, as shown in the right panel of Fig. 3.10. All 15 input signals were connected directly to the two outputs, which, in turn, provided the motor signals. Sprinkhuizen-Kuyper et al. noted that the fitness of an individual can be defined in several different ways, and the concepts of global, local, internal, and external fitness measures were introduced. A **global fitness measure** was defined as one only taking into account the difference between the final state and the starting state of the robot, in a given evaluation, whereas a **local fitness measure** assesses a robotic brain based on its performance at each time step. An **internal fitness measure** was defined as one only based on the information available to the robot, through its sensors, whereas an **external fitness measure** is based on information that is not directly available to the robot, such as e.g. its position. The authors defined four fitness measures, using all possible combinations, i.e. global external, global internal, local external, and local internal. For example, the global

external fitness measure was defined as

$$f_{\mathrm{GE}} = d(B_T, B_0) - \frac{1}{2} d(B_T, R_T), \tag{3.13}$$

where $d(B_T, B_0)$ denotes the difference in position of the box between the final state (at time T) and the initial state, and $d(B_T, R_T)$ is the difference in position between the box and the robot at the final state. The second term was introduced as a penalty for robots that did not keep pushing the box until the end of the simulation. The local external fitness measure instead used a sum of terms similar to those given in Eq. (3.13). The global internal fitness measure required the introduction of lights, so that the robot could distinguish good locations from bad ones. Finally, the local internal fitness measure was taken as that used by Lee et al. [71].

Robotic brains were evolved using each of the four fitness measures. Next, each robotic brain was tested for its performance based on the other three fitness measure. It was found that the best performance was obtained with the global external fitness measure, indicating that the best performance is obtained when the EA is allowed to explore its search space quite freely, without risking a penalty for occasional bad moves. The local internal fitness measure did not do very well, however, and its failure was attributed to the fact that this fitness measure would reward robots pushing against a wall (with sliding wheels). The final results were succesfully implemented in a Khepera robot.

3.2.3 Visual discrimination of objects

In many ways, vision is a much more complex sensory modality than mere proximity detection, as it generally involves an array or a matrix of data that somehow must be integrated in order for a robot to make sense of the environment it is looking at. However, using vision, it it possible for a robot to discriminate accurately between different objects.

In [100], the evolution of visual discrimination of objects was studied. In the initial experiments, Nolfi and Marocco, used an arena with two dark stripes of different widths on an otherwise white wall (see Fig. 3.11). The goal was to evolve a neural network brain that would allow the robot to approach the wide stripe, and to avoid approaching the narrow stripe. The task is made more difficult by the fact that a given stripe viewed by the robot can be either the wide stripe seen from a large distance, or the narrow stripe seen from a smaller distance. Thus, the robot must somehow learn to distinguish between the two stripes in order to achieve its goals.

The experiments were performed in simulations, in which a simulated Khepera robot, equipped with the standard IR sensors as well as a linear vision array (see Appendix C), was used. The robotic brain was a fully connected single-layer FFNN, with 14 inputs corresponding to the (normalized) readings of the

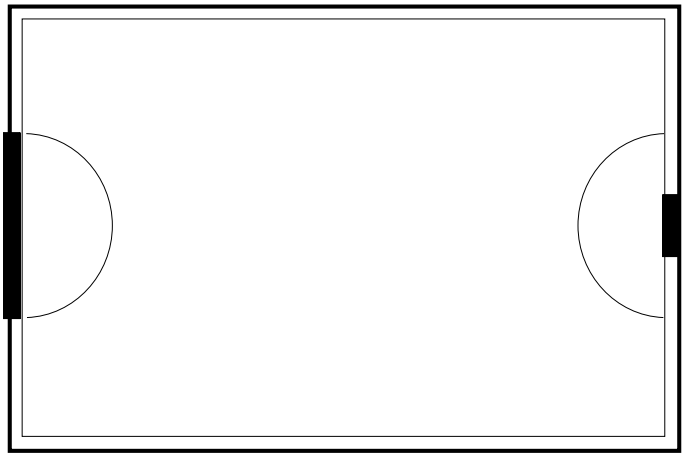


Figure 3.11: *A schematic view of the arena used in the visual discrimination experiments carried out by Nolfi and Marocco [100]. The stripes are shown as black lines, and the semi-circular regions indicate the distance at which evaluations were terminated.*

6 frontal IR sensors, and the readings from 8 equally spaced pixels of the 64 available pixels in the linear vision array. The network had two output signals, that were scaled up to the range $[-10, 10]$ and then used for setting the desired speed of the two motors. A direct encoding scheme, as described in Sect. 2.6 was used for the neural networks.

The authors note that the simulations, which took about 1 hour each on a standard PC, would have taken around 52 days if performed in the real robot.

In the simulations, each robot was evaluated in 30 trials (or **epochs**), each lasting a maximum of 30 seconds. Evaluations were terminated either if the robot ran out of time, or if it managed to reach the vicinity of either stripe. In cases where the robot reached the wide stripe, its fitness was increase by one unit. Similarly, the fitness was decreased by one unit if it reached the narrow stripe.

The best evolved robots generally managed to approach the wide stripe, by cleverly exploiting the fact that (1) the change in angular size of an object, as it is approached, varies with the distance to the object and (2) the duration of a sweep across a stripe is proportional to its perceived width. Thus, the successful robots turned until a stripe was detected, then moved toward the stripe, while turning slightly. Once visual contact with the stripe was lost, the robot turned to fact the other stripe, and approached while turning slightly, until visual contact was lost. Using this procedure, the robot tended to move more in the direction of the wide stripe, thus successfully approaching it.

In order to verify the results, the best neural network was implemented in an actual Khepera robot. The robot was started in nine different locations, using four different initial directions, leading to a total of 36 evaluations. Overall, the performance of the Khepera robot was only slightly worse than that of the

simulated robots.

Additional experiments were also performed, using instead an environment containing a small and a large black cylinders, whose sizes varied between different evaluations of the same individual. For more details concerning the behavior of the evolved robots in these experiments, see [100].

3.2.4 Corridor-following behavior

Corridor-following could be considered as an instance of the more general behavior *navigation* considered above. However, following corridors is an important special case, particularly in indoor applications. Thus, this topic will be given a brief separate consideration.

In [114], corridor-following behavior was evolved in simulations, using GP. Reynolds' main point was that previous investigations of corridor-following had been fully deterministic and noise-free, leading to brittle results, because of the ability of the EA to exploit the particular details of the environment in question. Reynolds, by contrast, introduced both sensory noise and motor noise in the simulations.

The simulations were performed using a very simple robot model, with a lower limit on its turning radius, essentially forcing it to move along the corridor in which is was placed. As is common in GP, the structure of the robotic brain was allowed to vary during evolution. The simulated robots were equipped with simple sensors, measuring wall proximity.

The author chose a fitness measure that encouraged even very slight improvements. Thus, each individual was evaluated for a maximum of 16 trials. In each trial a graded fitness measure was used, based on the percentage of the corridor negotiated (without collisions) by the simulated robot, starting from a random direction of heading, as illustrated in Fig. 3.12. If a robot collided with a wall, it was not allowed to perform further evaluations. Thus, there was a selection pressure in favor of individuals capable of general navigation in the corridor.

Simulations were performed using either a fixed set of (nine) proximity sensors, or a variable sensor setup, determined by the EA. Thus, in the latter case, both the brain and (part of) the body of the simulated robots were evolved. As expected, it was found that sensors placed in forward-pointing directions, were more useful in corridor following than sensors pointing to the side or backwards. A certain degree of lateral symmetry was also found, even though the symmetry was not exact. The issue of simultaneous evolution of body and brain will be considered in more detail below (see Subsect. 3.4.3).

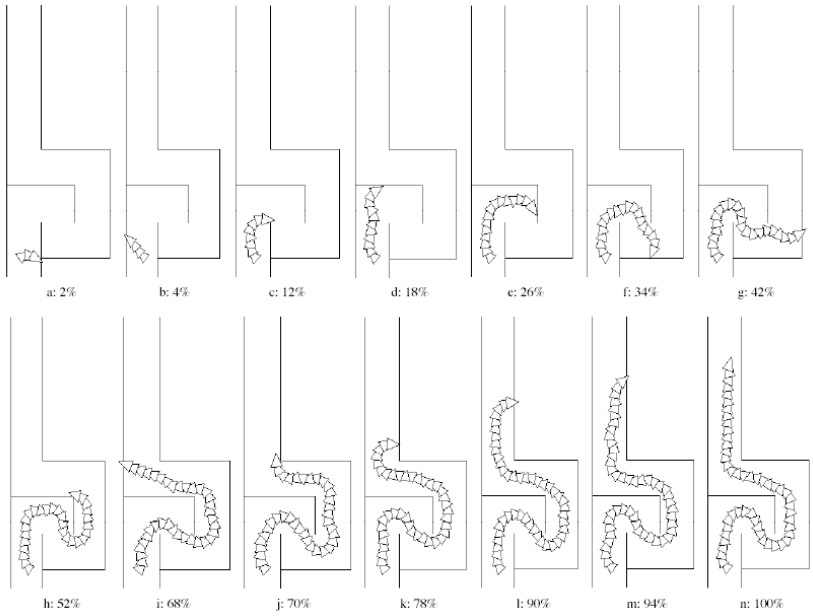


Figure 3.12: *Gradually improved results obtained during the evolution of* corridor-following *behavior. Reproduced with kind permission of Dr. C. Reynolds.*

3.2.5 Behaviors for robot soccer

In recent years, there has been a great interest in robot soccer, manifested in events such as the Robocup tournament (see www.robocup.org). Robot soccer, or indeed any two-person (or two-team) game for robots, leads to interesting questions concerning not only evolution of behaviors in general, but also multi-robot coordination, co-evolution etc. Research on this topic has resulted in a very large number of publications, and only a few brief examples will be given here. The interested reader is referred to www.robocup.org for further information.

The ultimate aim of Robocup is to generate a team of (humanoid) robotic football players, capable of beating the best human team, a seemingly distant goal. In addition to humanoid robot soccer, wheeled robots and simulated robots have been used as well in the framework of Robocup. In [70], Lazarus and Hu evolved goalkeeper behavior in simulations using GP. In the simulations, the function set (see Sect. 2.6) included the standard operators of addition (`add`), multiplication (`mult`) etc., as well as problem-specific operators such as e.g. `kick` and `catch`. The terminal set included e.g. the direction to the ball and the distance to the ball. Each individual goalkeeper robot was tested against attacks from 10 different directions, as shown schematically in Fig. 3.13. An attack consisted of the attacker kicking the ball in a straight line toward the goal. The authors attempted to evolve active goalkeepers, that would not only catch an incoming ball, but would also attempt actively to

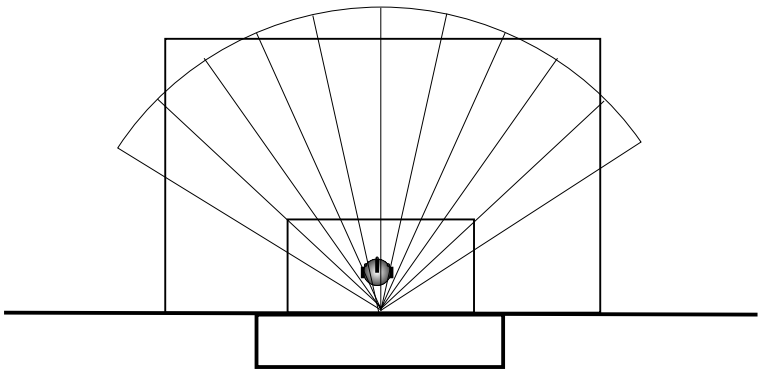


Figure 3.13: *A schematic illustration of the 10 different attack directions used by Lazarus and Hu in the evolution of a soccer goalkeeper [70].*

move toward the ball and intercept it. To this end, a fitness measure involving five parts, namely ball saving, self-localization, ball localization, movements, and positioning was used. When using a multimodal fitness measure, the problem of weighing different fitness cases against each other always appears. In [70], it turned out that the best results were found using equal weights.

In [138], simple ball-following behavior was evolved, also using GP and employing several (in this case, 8) different trials in order to evolve robust solutions. Luke et al. [75] used co-evolution to evolve a simulated robot soccer team.

3.2.6 Motion of a robotic arm

As mentioned in the introduction to this chapter, BBR (and therefore ER) is mostly concerned with autonomous robots. However, EAs have also been used in connection with stationary robots such as robotic arms. An example of such an application is reported in [91], where obstacle avoidance was evolved for an OSCAR-6 robotic arm. The authors noted that, while basic manipulator-eye coordination had been obtained using neural networks trained in a supervised manner (using, for example, backpropagation [48]), such approaches are mainly useful in structured environments without obstacles. In supervised training, the neural network must generally be given feedback for each input-output mapping, something which is difficult to provide e.g. in obstacle-filled environments. Thus, Moriarty and Miikkulainen [91] used an EA to evolve neural networks for controlling the robotic arm, guiding the search by means of a single fitness measure at the end of each evaluation, rather than providing the neural network with a detailed performance measure for every single motion carried out.

The task of approaching a target was divided into two phases, an initial approach phase phase, which brought the end effector to the vicinity of the tar-

get, and a final approach phase which used smaller movements to reach within grasping distance of the target. Based on this division, the authors devised a control system consisting of two neural networks, a primary network for the initial approach phase, and a secondary network for the final approach phase. Each network was represented as a fully connected FFNN, with 9 input units, a single hidden layer with 16 neurons, and 7 output neurons. The 9 input units consisted of 6 proximity sensors, as well as the x, y, and z components of the distance between the end effector and the target. Six of the output units were used for determining the direction of rotation and magnitude of rotation for three joints in the robotic arm. The output units determining the magnitude of rotation were scaled differently in the primary and secondary networks, in such a way that the motion of the secondary networks used smaller steps. The seventh output unit was used to override the output of the other six outputs, and thus to stop the arm, in emergency situations.

The training of the arms was made by starting the arm with random joint positions, and a random target position, selected from a set of 400 pre-generated positions. An additional 50 target positions were generated for use during testing. In addition, obstacles were placed in one of 12 imaginary boxes located in the path of robotic arm. The motion was executed as a series of discrete moves, and simulations were terminated e.g. if a collision occurred. Fitness was assigned as the percentage of the initial distance (between the position of the end effector and the target) covered during the motion.

The results of the EA showed that the combination of primary and secondary networks could, in fact, control the arm to within industry standards, i.e. with a deviation of one cm or less between the actual and desired end effector positions. Furthermore, using the 50 test cases, it was shown that the evolved solutions were robust; Obstacles were hit only in around 2% of the test cases.

3.3 Evolution of complex behaviors and behavioral organization

A central issue in BBR is behavioral organization (also known as **behavioral coordination**, **behavior arbitration**, or **action selection**), i.e. the process of determining which behavior to activate at a given time. Several methods for behavioral organization have been suggested, e.g. subsumption [14], activation networks [78], potential fields [64], distributed architecture for mobile navigation (DAMN) [115]. For reviews of such methods see e.g. [3], [9], and [108].

However, in many methods suggested to date, it is not uncommon, as pointed out in [9], [79], and [131], that the user must specify the architecture of the behavioral organization system by hand. This is far from optimal, for

several reasons: First, as mentioned in Subsect. 3.1.1, behavior-based robots are generally expected to operate in unstructured environments, which cannot easily be predicted, making it difficult to tune parameters by hand. Second, even if the environment happens to be fairly well structured, it is difficult to assess (at least quantitatively) the relevance of some behaviors, especially those that are not directly related to the assigned task of the robot. For example, a battery-driven robot that is assigned the task of transporting goods between two locations, must occasionally recharge its batteries. Clearly, the behavior of charging batteries is important, but it is difficult to judge quantitatively its relevance against the relevance of the assigned task, and thus to assign an optimal reward for it. Third, hand-coding a behavioral organizer represents a step away from the biologically inspired behavior-based architecture.

The purpose of any method for behavioral organization is to determine when different behaviors should be activated, and it is therefore natural to categorize methods of behavioral organizations based on the procedure they use for selecting behaviors. There are two main categories, namely **arbitration methods** and **cooperative methods**[3]. In arbitration methods, exactly one behavior is active, and the selection of which behavior to activate is generally a function both of present sensor readings and the internal state of the robot. In cooperative methods, the action taken by the robot is a weighted combination of the actions suggested by several behaviors.

The problem of behavioral selection has, of course, also been heavily studied in the field of **ethology** (see e.g. [30] for a review). The brains of biological organisms have been optimized through millions of years of evolution resulting in behavioral selection systems, manifested in the beliefs and desires of the organism, that maximize the chances of survival. Thus, the concepts and ideas introduced in ethology should be carefully considered when proposing methods for behavioral selection in robotics.

From the point of view of ER, the aim, of course, is to reach beyond the simple (and often purely reactive) robotic brains described above, and thus to generate robotic brains capable of more complex behaviors. However, not all such studies have centered on behavioral organization in the sense described above, let alone *evolution* of behavioral organization. Basically, the published studies that focus on the evolution of complex behaviors can be divided into three categories, namely (1) those in which complex behaviors are evolved without explicit behavioral selection, i.e. without the presence of an independently generated behavioral organizer that selects which behavior to activate [34], [44], [67], [94], [137], (2) those in which behaviors are evolved independently and are then used in a complete robotic brain equipped with a behavioral organizer which is *not* evolved itself [65], and (3) those in which the central issue is the evolution of the behavioral organizer, and where the individual behav-

[3]Cooperative methods are sometimes also called **command fusion methods.**

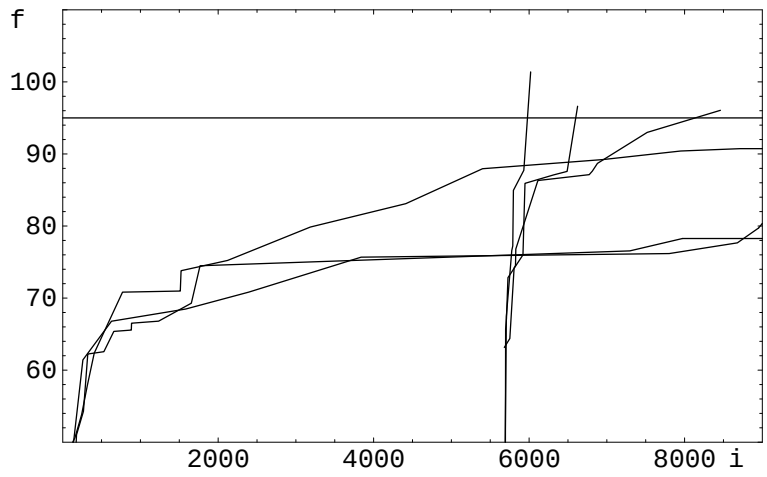


Figure 3.14: *Maximum fitness as a function of the number of evaluated individuals during the evolution of a complex behavior, divided into two constituent behaviors [137]. The three short and almost vertical curves show the results from runs in which the constituent behaviors were first evolved separately, and then combined to form the complete robotic brain.*

iors are generated either through an EA or by other means [71], [131].Here, a short description of each category will be given, with the strongest emphasis placed on the third category (see Subsect. 3.3.3 below), for which the utility function method [131] will be described in some detail.

3.3.1 Complex behaviors without explicit arbitration

As pointed out in [44] and [137], a common problem when evolving complex behaviors is that the difference between the initial, randomly generated, robotic brains and those capable of carrying out the desired complex behavior, is so large that the EA without exception finds itself stuck in a local optimum. One obvious solution is to break down the problem into simpler tasks, for which the EA *will* be able to find an adequate solution, and then either to fuse several evolved robotic brains into one complete brain [137], or to make the problem progressively more difficult until the desired level of complexity is reached, a procedure called **incremental evolution** [44].

An example, taken from [137], is shown in Fig. 3.14. Here, the task was to evolve a complex robotic brain, capable both of cleaning an arena *and* avoiding moving obstacles. Two different approaches were used, namely (1) attempting to evolve the complex behavior directly, starting from random robotic brains (which, in this investigation, were represented by generalized FSMs, see Appendix C), and (2) first evolving *cleaning* and *evasion* separately, then fusing the two generalized FSMs by adding random connections between them and, finally, continuing the evolution until the complex behavior emerged. In Fig.

3.14, the results of the investigation are shown. The three lower curves, extending over the whole interval, show the maximum fitness as a function of the number of evaluated individuals, whereas the three short, almost vertical curves show the results obtained when first evolving the **constituent behaviors** separately. In the latter case, the two curves have been shifted to right by an amount that corresponds to the number of evaluated individuals needed to reach acceptable performance of the two constituent behaviors. As is evident from the figure, the approach of first evolving constituent behaviors separately is, by far, the most successful one, quite rapidly reaching the empirically determined fitness threshold for acceptable performance of the complex behavior (indicated by the horizontal line). By contrast, the attempts to evolve the complex behavior directly were not successful: In no case was the fitness threshold reached.

A similar conclusion was reached by Gomez and Miikkulainen [44], who studied the evolution of complex *prey capture* behavior in contests of pursuit and evasion. In their study, a simulated robot was required to capture a prey, moving with speed s, which was given a head start of n moves, a task denoted E_n^s. It was found that the complex behavior $E_4^{1.0}$ could not be generated by direct evolution. However, using several intermediate steps, first evolving $E_0^{0.0}$, then $E_2^{0.0}$ etc., the $E_4^{1.0}$ behavior could be achieved easily.

Thus, the conclusion reached in [44] and [137] is that a complex behavior can be evolved incrementally. However, this conclusion is perhaps not a general one. Floreano and Mondada [34], by contrast, point out that complex behavior can, at least in certain situations, be achieved simply by increasing the complexity of the environment, and *decreasing* the complexity of the fitness function, in order to avoid canalizing the evolving robotic brains toward suboptimal solutions. In [34], the task of navigation described in Subsect. 3.2.1, was extended also to include the limited energy storage capacity of the battery of the robot. Thus, the behaviors of *homing navigation* (toward a light source close to the charging station) and *battery charging* had to be included in the brain of the robot. Rather than using incremental evolution, Floreano and Mondada simplified their fitness measure (see Eq. 3.3), removing the requirement of straight-line motion, and allowed the evolving individuals to extend their life by periodically returning to the charging station, in which case their simulated batteries were instantaneously recharged. Indeed, during evolution, individuals were found that could return to the charging station, thereby extending their life, even though no fitness was directly associated with doing so. In fact, the ideas presented in [34] bear some resemblance to the dichotomy of task behaviors and auxiliary behaviors, discussed in connection with the utility function method described below.

It should be noted, however, that the task used in [34] was quite simplified (e.g. by allowing instantaneous recharging), and that the authors did not attempt to compare with incremental evolution through which, perhaps, the

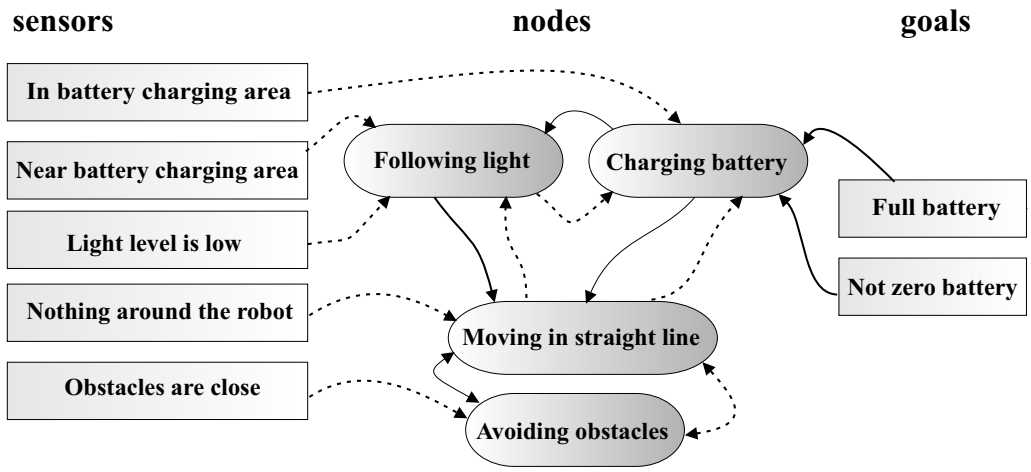


Figure 3.15: *A schematic representation of the behavioral organizer used by Kim and Cho [65]. The dashed lines are* **successor links** *and the solid ones are* **predecessor** *links, as defined in the method of activation networks [79].*

complex behavior could have evolved even faster. Nevertheless, the results reported above show that there are several ways of achieving complex behaviors.

In addition, Floreano and Mondada carried out a painstaking analysis of their evolved neural network, in order to determine how it generated the complex behavior. While their analysis was impressive, it does, perhaps, point to a weakness with respect to the use of neural networks. The advantages of neural networks, i.e. noise tolerance, absence of bias introduced by the user etc., have been eloquently presented by several authors (e.g. [19], [34], and [94]). However, neural networks are also notoriously difficult to interpret. Thus, a complex behavior based on neural networks will, most often, have to be used as a black box. An alternative approach is to consider the robotic brain as a two-part system containing (1) a set of basic behaviors, which *may* be implemented as neural networks, but also in other ways, using e.g. finite-state machines or `if-then-else`-rules, and (2) a behavioral organizer (or **arbitrator**) that dynamically selects which behavior to activate, at all times. Such systems will be introduced in the two following subsections.

3.3.2 Evolving behavioral repertoires

Kim and Cho [65] evolved a set of behaviors (also called a **behavioral repertoire** [131]) for a task similar to the homing navigation considered by Floreano and Mondada [34]. Their repertoire consisted of four behaviors, namely *charg-*

ing battery, *following light*, *avoiding obstacles*, and *moving in a straight line*. For each behavior, a neural network known as a **CAM-brain** (based on cellular automata) was evolved, using incremental evolution [44]. However, the behavioral organizer was *not* evolved. Instead the method of activation networks [78], [79] was used. Kim and Cho found that the behavioral organization system was able to select behaviors in the appropriate order, so as to achieve the main task of navigating in the arena, without collisions and without running out of battery energy.

As pointed out above, an advantage of their behavioral organizer, compared to complex behaviors implemented in a neural network, is its relative simplicity, as indicated in Fig. 3.15. However, the method of activation networks, as indeed most methods of behavioral organization, requires that the user be able to set a large number of properties manually, something which becomes more and more difficult as the size and complexity of the behavioral repertoire grows.

Thus, as an alternative to such methods, one may *evolve* the behavioral organizer itself. To conclude this section, one such method will now be described in some detail.

3.3.3 Evolution of behavioral organization by means of the utility function method

The utility function method developed by Wahde [131] is a biologically inspired arbitration method, in which the selection of behaviors is based on a quantity called **utility**. Thus, before the method itself is described, a short introduction will be given to the concept of utility and its uses in rational decision-making.

Utility and rational decision-making

The concept of utility is used both for studying decision-making in economic theory and for studying behavioral selection in ethology. [4]

Consider a case in which a robot must decide which of n actions A_1, A_2, ..., A_n to perform. For simplicity, consider the case $n = 3$, and assume that the actions have been ordered according to preference, such that the robot prefers A_1 to A_2, and A_2 to A_3. The situation can be denoted $A_1 > A_2$, $A_2 > A_3$, where $>$ in this case means "is preferred to". If this is the case, then A_1 must also be preferred to A_3, so that $A_1 > A_2 > A_3$. This property, known as **transitivity of choice**, underlies all rational decision-making. An agent exhibiting transitivity

[4] In ethology, the terms **benefit** or **negative cost** are often used instead of utility. Here, however, the term utility with be used throughout.

of choice is called a **rational agent**. Note that the use of *reason* is not a necessary prerequisite for rationality. Rational choices can be made unwittingly as a result of design, as is the case in reactive robots and most likely in insects and, probably, in many higher animals as well.

In order to rank possible actions in order of preference, a **common currency** must be available. Furthermore, it can be shown (see e.g. [30], [31], and [32]) that an agent with transitive preferences uses a maximimization procedure, i.e. will select the action whose value, measured in the common currency, is maximal. The common currency is usually called **utility**, and a rational agent thus selects the action to perform ($A_{i_{\mathrm{sel}}}$) according to

$$A_{i_{\mathrm{sel}}} = \mathrm{argmax}\ U(A_i), \tag{3.14}$$

where $U(A_i)$ denotes the utility of action A_i in the current situation.

As an example, consider a floor-sweeping robot, which is given a fitness increment for each square meter of floor it sweeps. Clearly, the robot should try to sweep as much floor as possible, in order to maximize its fitness. However, if the robot runs out of battery energy, it will no longer be able to move. Thus, the utility of a behavior that forces the robot to suspend, temporarily, its floor-sweeping activities to charge its batteries will rise as the energy in the battery decreases, even though the battery charging behavior does not lead to any *direct* increase in fitness. Hence, in order to receive as much fitness as possible over a long period of time, the robot must, in fact, maximize *utility*.

Utility also provides a means of allocating limited resources in an optimal way. The life of any animal (or robot) inevitably involves many trade-offs, where less relevant behaviors must be sacrificed or at least postponed in order to perform the most relevant behaviors, i.e. those associated with largest utility value.

Organizing behaviors using the utility function method

In the utility function method [131], the robot is equipped with a behavioral repertoire, in which each behavior B_i is assigned a utility function U_i, which depends on the values of the **state variables** of the robot. Three kinds of state variables are defined, namely **external variables**, denoted s, representing e.g. sensor readings, **internal physical variables**, denoted p, representing e.g. the energy in the batteries of the robot, and **internal abstract variables**, denoted x, which are dimensionless variables used in the behavioral selection procedure, and which roughly correspond to signalling substances (e.g. hormones) in biological systems. Thus, the most general form of a utility function is

$$U_i = U_i\left(s_1, \ldots, s_{n_{\mathrm{e}}}, p_1, \ldots, p_{n_{\mathrm{p}}}, x_1, \ldots, x_{\mathrm{n_i}}\right), \tag{3.15}$$

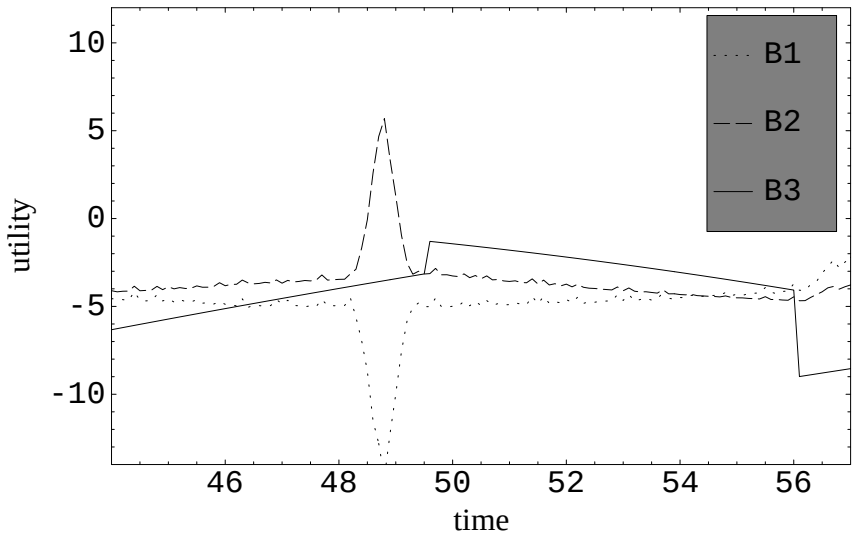


Figure 3.16: *The utility functions for a behavioral organization problem, shown as functions of time during part of the evaluation of a simulated robot.*

where n_{e}, n_{p}, and n_{i} denote the number of external, internal physical, and internal abstract variables, respectively. However, in most cases, each utility function will only depend on a subset of the set of available variables.

In the utility function method, behaviors are divided into two categories. **Task behaviors** are directly related to the task of the robot and increase its fitness, if successfully executed. An example of a task behavior is the floor-sweeping behavior described in the example above. **Auxiliary behaviors**, on the other hand, do *not* change the fitness of the robot, but are nevertheless necessary for the robot to be able to perform its duties. The battery charging behavior (for the floor-sweeping robot described above) is an example of an auxiliary behavior.

Furthermore, a time variable t_i, referred to as the **behavior time** of behavior B_i, is defined for all behaviors. t_i is set to zero every time B_i is activated, and then follows the same rate of increase as the global time variable t, which measures the time since the robot was initialized. As soon as behavior B_i becomes inactive, t_i is again set to zero, where it remains until the behavior is activated again.

Behavioral selection is straightforward in the utility function method; At regular intervals, the values of the utility functions are computed, using the latest available readings of the state variables as input, and the behavior with the highest utility value is executed. This is illustrated in Fig. 3.16, in which is shown the variation in time of three utility functions obtained for a behavioral organization problem involving three behaviors. In the part of the simulation shown in the figure, behavior B_2 is initially active, since its utility value ex-

ceeds that of the other behaviors, and remains active from $t = 44$ (arbitrary units) until around $t = 49.5$, when B_3, which in this case happened to be an auxiliary battery charging behavior, was activated. Around $t = 56$, B_1 was activated.

The problem, of course, is to determine the shape of the utility functions. In the utility function (UF) method, the optimization of utility functions is normally carried out using EAs. In general, the utility functions depend on several state variables, and should provide appropriate utility values for any combination of the relevant inputs. Thus, determining the exact shape of the utility functions is a formidable task, and one for which EAs are very well suited. In principle, GP can be used, in which case any function of the state variables can be evolved. However, it is often sufficient to make an ansatz for the functional form of each utility function, and then implement the EA as a standard GA for the optimization of the parameters in the utility function.

For example, for a utility function U_i (associated with a behavior B_i) that depends on the two variables s and p, a common ansatz is

$$U_i(s,p) = a_{i,00} + a_{i,10}s + a_{i,01}p + a_{i,20}s^2 + a_{i,11}sp + a_{i,02}p^2, \tag{3.16}$$

where the $a_{i,jk}$ are constants. When optimizing the utility functions the $a_{i,jk}$ (and the corresponding constants in all other utility functions), should be encoded in the chromosomes used by the GA. The fitness gained while executing the task behavior(s) is used as the optimization criterion.

A problem that may occur when selecting behaviors based on utility functions is **rapid behavior switching** in which the robot keeps swapping back and forth between two (or more) behaviors, and thus failing to achieve its goals. One of the purposes of the internal abstract variables is to prevent such rapid switching. Thus, an internal abstract variable x_i, which takes non-zero values only in a specific behavior B_i, can be introduced. When the robot switches from another behavior to B_i, x_i is immediately set to a non-zero value (e.g. 1), and if the utility functions are properly chosen, the utility of B_i will then be raised sufficiently to avoid immediate switching to another behavior. As an example consider, in Fig. 3.16, the jump in utility for B_3 at the moment when it becomes active. The internal abstract variables may depend both on other state variables and on the behavior time t_i, and the exact variation can be optimized by the GA as well. For example, a possible ansatz for an internal abstract variable x_i is

$$x_i = \begin{cases} b_{i,1} + b_{i,2}\mathrm{e}^{-|b_{i,3}|t_i} & \text{If } B_i \text{ is active} \\ 0 & \text{Otherwise} \end{cases} \tag{3.17}$$

where the $b_{i,j}$ are constants. Thus, with this ansatz for the internal abstract variables, the chromosomes used by the GA will encode not only the constants $a_{i,jk}$, but the constants $b_{i,j}$ as well.

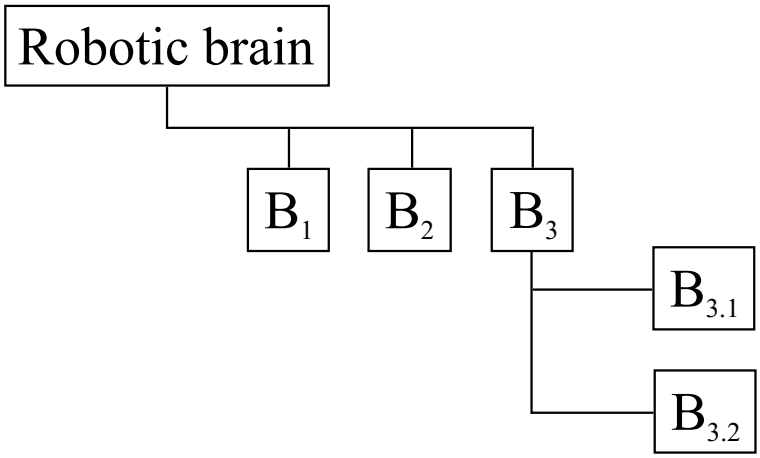


Figure 3.17: *An example of a robotic brain involving five behaviors on two hierarchical levels, taken from [136].*

Behavioral hierarchies

The presentation above is somewhat simplified, in that it does not consider the fact that, in order to keep the level of complexity of the constituent behaviors in a robotic brain at a reasonable level, the behaviors should often be divided into sub-behaviors. Battery energy maintenance is a case in point: maintaining the energy in the batteries of a robot requires that (at least) two separate procedures be executed: one for finding an energy source, and one for connecting to the energy source and carrying out the actual charging. Evidently, the entire energy maintenance sequence *could* be considered as one single behavior. However, generating such a behavior, and making it reliable, for example in the case of sudden interruptions due to obstacles in the path of the robot, would indeed be a daunting task. An alternative procedure is to introduce one behavior for locating an energy source, and one for battery charging, and to consider these behaviors as sub-behaviors to an overall energy maintenance behavior. Thus, in such a case, the robotic brain would have a hierarchical structure, an example of which is shown in Fig. 3.17.

This figure shows the structure of the robotic brain for a simple, two-wheeled guard robot considered in [136]. The behavioral repertoire consisted of a total of five behaviors, namely *straight-line navigation* (B_1), *obstacle avoidance* (B_2), *energy maintenance* (B_3), *corner seeking* $(B_{3.1})$, and *battery charging* $(B_{3.2})$. In the arena where the robot operated, the battery charging stations were placed in the corners. Thus, the *corner seeking* behavior corresponded to charging station localization.

As shown in the figure, the five behaviors were placed on two hierarchical levels. The utility function is able to cope with multiple hierarchical levels of behaviors, by comparing utility values on a level-by-level basis. Thus, in the

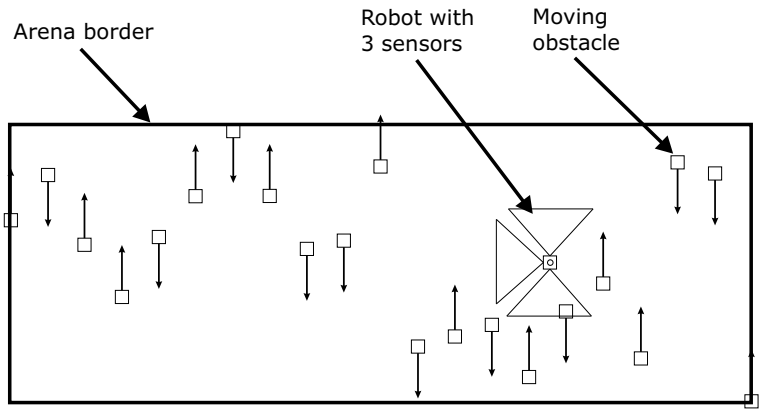


Figure 3.18: *The arena used in [107]. The aim of the simulated robot is to move without collisions, from right to left, in the arena.*

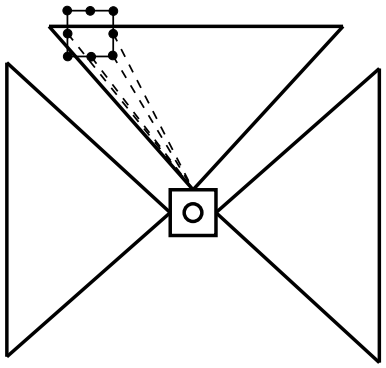


Figure 3.19: *The robot used in [107] as seen from above, equipped with three simple proximity sensors, each with a triangular-shaped detection range. The filled black circles illustrate points on an obstacle that are detectable by the range sensors. Dashed lines indicate detected points in this particular configuration.*

case shown in Fig. 3.17, it was determined which of the three utility functions $U_1, U_2,$ and U_3 took the highest value. If it happened to be U_3, the comparison of $U_{3.1}$ and $U_{3.2}$ would determine which of these two behaviors was active.

The procedure is actually slightly more complex than this since the switch from, say, B_2 to B_3 (and then to one of the sub-behaviors $B_{3.1}$ or $B_{3.2}$) may require modification of the internal abstract variables (if any) in B_3, in order to avoid rapid behavior switching as discussed above. In practice, this is accomplished by executing an *enter procedure* on each hierarchical level. Thus, when U_3 exceeded U_2, the enter procedure for U_3 was executed (thus possibly further raising U_3 to prevent rapid switching back to B_2), Next, the enter procedure of either $B_{3.1}$ or $B_{3.2}$ was called, depending on their relative utility values, and the robot then proceeded by executing the active behavior, i.e. either $B_{3.1}$ or $B_{3.2}$.

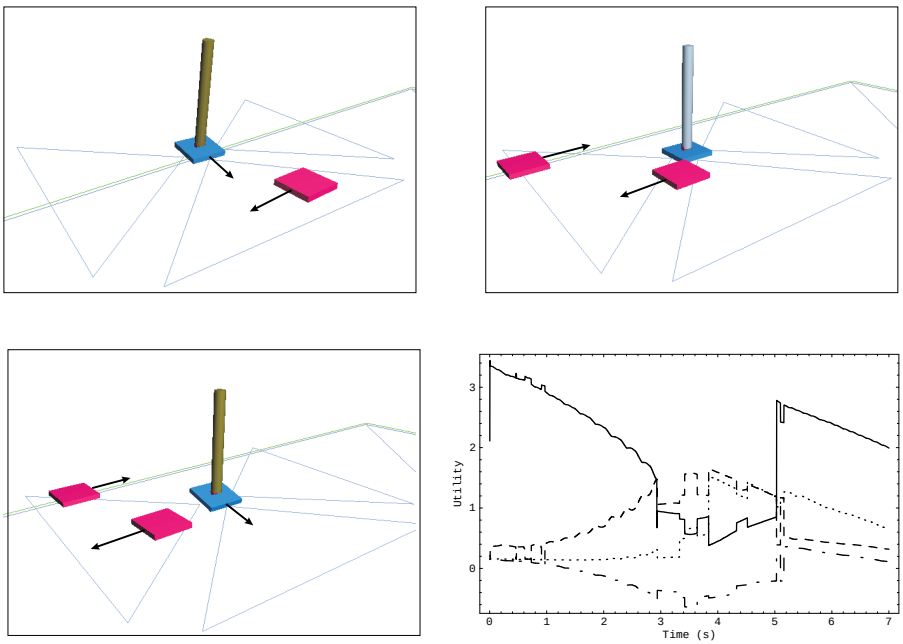


Figure 3.20: *The early stages of a typical run. The bottom right panel shows the variation of the four utility functions in one of the best performing individuals during evolution with the dynamic model. The curves represent the utility values for the behaviors* move forward *(solid curve),* move backward *(dash-dotted curve,),* stop *(dotted curved), and* charge *(dashed curve). Note that the utility of the* move backward *behavior is very low throughout the simulation.*

Example 1: Locomotion in an arena with moving obstacles

In [107], the utility function method was used for generating a behavioral organization system for a hopping robot traversing an arena with moving obstacles, illustrated in Fig. 3.18. The simulated robot, consisting of a foot plate and a leg with two **degrees of freedom** (DOF), implemented as two revolute joints, was equipped with four behaviors: *move forward* (B1), *move backward* (B2), *stop* (B3), and *charge batteries* (B4). The robot was simulated using full newtonian dynamics, implemented in the EvoDyn package developed by Pettersson [104]. In B4, the robot was required to remain at a standstill, charging its batteries using simulated solar cells.

While the UF method itself uses an EA to generate behavioral organization, the constituent behaviors can be generated by any method. In the particular case considered in [107], the behaviors were implemented as continuous-time, recurrent neural networks (see Appendix A), and were optimized using an EA. Note, however, that the EA optimizing the individual behaviors should not be confused with the EA used for generating the behavioral organizer through the UF method.

Once the four constituent behaviors had been generated, the behavioral

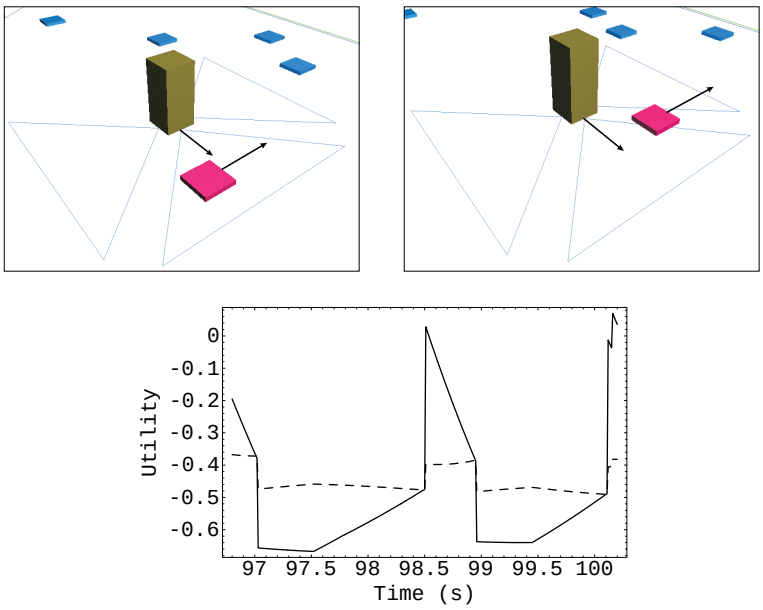


Figure 3.21: *A behavior switching sequence where, at first, behavior B1 (*move forward*) is active. Shortly thereafter behavior B4 (*charge*) is activated due to the detection of an approaching obstacle. As the obstacle passes, and the sensor signal decreases, B1 is activated again. For clarity, only the utility values for B1 (solid line) and B4 (dashed line) are shown in the bottom panel.*

organizer, assigned the task of selecting between behaviors B1-B4, was generated using the utility function method. The simulated robot was equipped with three proximity sensors, as illustrated in Fig. 3.19. In addition, the robot was able to measure the amount of available energy in its batteries. Thus, the ansatz for each utility function was taken as a second-degree polynomial $U_i = U_i(s_1, s_2, s_3, E, x_i)$, where s_1, s_2, and s_3 are the readings of the three sensors, E is the battery energy, and x_i is a behavior-specific internal abstract variable, whose variation was modelled as in Eq. (3.17). The total number of parameters to be optimized by the GA was 96.

The settings of the simulated robot's battery were such that frequent recharging was necessary. Thus, the task of moving through the arena was strongly non-trivial, requiring constant vigilance in order to avoid collisions or an empty battery (in which cases the evaluation was terminated). The fitness measure was simply taken as the distance moved in the forward direction.

Despite the many parameters determining the utility functions and the abstract internal variables, a successful behavioral organizer was found quite quickly. An example is shown in Fig. 3.20, and a movie of the robot is available on the tutorial CD in the file `HoppingRobot_BehavioralOrganization.mpg`.

However, the dynamically simulated robot was unable to traverse the whole arena. The failure could be attributed to the limited quality of the constituent

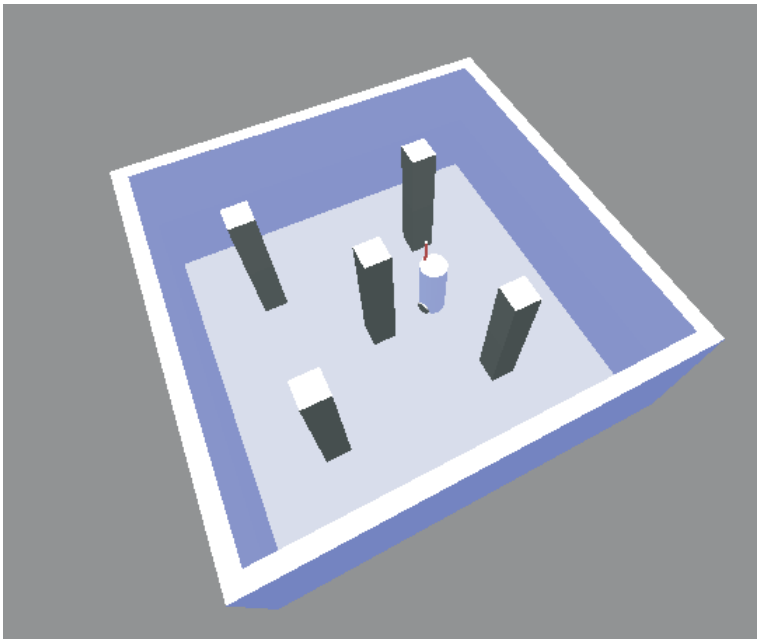


Figure 3.22: *A two-wheeled exploration robot in a simple arena with 5 stationary obstacles.*

behaviors B1-B4, rather than a failure of the behavioral organization system. Thus, an obvious conclusion was that the constituent behaviors must at least reach some minimum level of quality, in order for a behavioral organizer to perform well.

In order to study the evolution of behavioral organizer in its own right, Pettersson and Wahde also carried out some simulations using a much simpler dynamical model (essentially a point-particle model) for the simulated robots. In this case, the EA found utility functions that would allow the robot to traverse the entire arena without collisions. An example of the motion of the simplified robot is shown in Fig. 3.21. In addition, two movies are available on the tutorial CD, in the files `SimpleRobot_UM1.mpg` and `SimpleRobot_UM2.mpg`.

Example 2: A simple exploration robot

As a second example of an application of the utility function method, a simple exploration robot will be considered. The two-wheeled robot, shown in Fig. 3.22, is differentially steered, and equipped with a two-dimensional laser range finder for measuring the distance to the nearest objects in a certain sector. In this example, the sector of measurement will be assumed to cover a narrow range centered around the current direction of motion of the robot. The task of this robot will be to explore a given arena, also shown in Fig. 3.22, while avoiding collisions with stationary obstacles.

Obviously, in a more realistic application, the robot would have to be able

to find a charging station and charge its batteries when necessary, avoid moving obstacles (in addition to stationary ones), and carry out purposeful motion rather than the more or less random wandering that will be considered here. In such cases, a rather complex robotic brain involving many behaviors, probably distributed on various hierarchical levels, e.g. as illustrated in Fig. 3.17, would be needed. The utility function method would easily handle such a situation, but, for simplicity and clarity, a greatly simplified situation will be studied in this example, where the capacity of the robot's battery is assumed to be infinite. In fact, only two behaviors will be included in the robotic brain, namely *straight-line navigation* (B_1) and *obstacle avoidance* (B_2). In B_1, the brain of the robot sends equal signals to the two motors of the robot, making it move in a straight line (after an initial transient, in case the robot was turning at the time of activation of B_1). In B_2, equal signals, but of opposite sign, will be sent to the two motors, until the minimum distance (as measured by the laser range finder) to the nearest obstacle exceeds a certain minimum value.

The fitness measure is basically taken as the amount of time spent executing B_1. However, the robot receives a fitness increase *only* if it executes B_1 continuously for at least T_0 seconds. The value of T_0 was chosen, somewhat arbitrarily, as 0.2 s. With this fitness measure, the robot has a strong incentive for executing B_1 as much as possible, without dithering between the two behaviors. On the other hand, it is also necessary for it sometimes to activate B_2, in order to avoid collisions, since the evaluation is terminated if the robot collides with a wall or with some other obstacle. Thus, the robot is faced with a behavioral selection problem, involving a trade-off between carrying out the assigned task (by executing B_1) and surviving (by executing B_2). Clearly, in this simple case, it would not have been very difficult to write down a procedure for behavioral selection by hand. However, as indicated above, this simple example of a behavioral selection problem was chosen primarily for its (intended) pedagogical value, not for its (rather limited) usefulness.

In order to apply the utility function method, an ansatz is needed for each utility function. Here, U_1 will be taken as a p^{th} degree polynomial in the variables s_1, s_2, and s_3 which represent the (inverted) readings along three rays of the laser range finder (in the directions 0, $\pm$ 30 degrees, relative to the forward direction). The inverted reading y along a ray equals $R - z$, where R is the maximum range (4.0 m, in this case) of the laser range finder, and z is the measured distance to the nearest object along the ray. Thus, y will be in the range $[0, R]$. For B_2, the utility function (U_2) depends on two variables: s_{avg}, which equals the average reading of all rays of the laser range finder and x, which is an internal abstract variable. x can be interpreted as a hormone whose level is raised if the robot senses fear, e.g. as a result of an imminent collision. The variation of the variable x is slightly simplified in this example, however: it takes the value 0 if B_1 is active, and the value 1 if B_2 is active.

Thus, the utility functions U_1 and U_2 will be polynomials of three and two

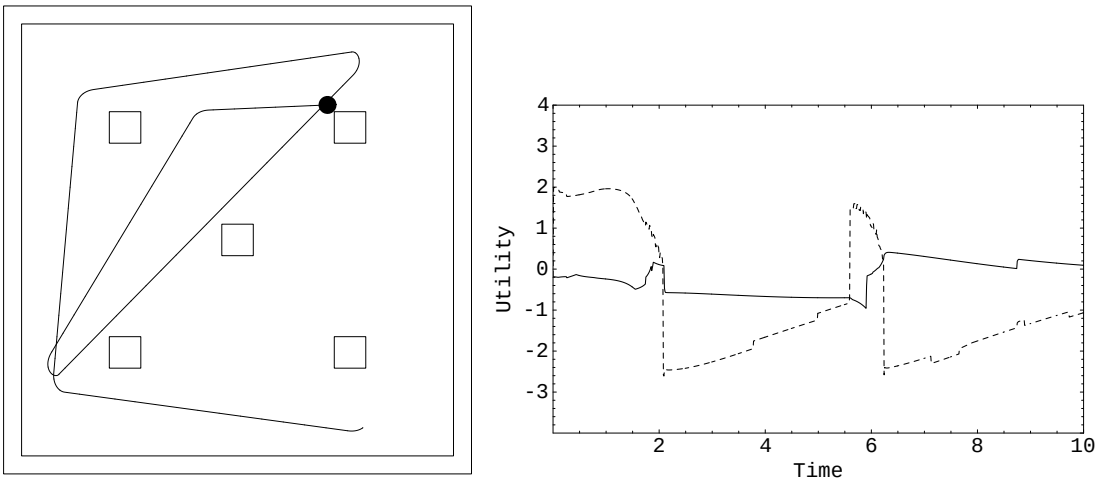


Figure 3.23: *Left panel: The motion of the robot, whose initial position is marked by a black disk. Right panel: The variation of the utility functions for the first 10 seconds of the motion. The solid line shows U_1, and the dashed line U_2.*

variables, respectively, i.e.

$$U_1(s_1, s_2, s_3) = a_{000} + a_{100}s_1 + a_{010}s_2 + a_{001}s_3 + a_{110}s_1s_2 + \ldots, \tag{3.18}$$

and

$$U_2(s_{\text{avg}}, x) = a_{00} + a_{10}s_{\text{avg}} + a_{01}x + a_{20}s_{\text{avg}}^2 + a_{11}s_{\text{avg}}x + a_{02}x^2 + \ldots \tag{3.19}$$

Thus, the task of the evolutionary algorithm will be to determine the coefficents a_{ijk} and a_{ij} so as to maximize the fitness of the robot in evaluations of a given maximum duration T (here set to 50 s).

The two behaviors B_1 and B_2 were implemented in the format defined by the `UF Library` simulation package described in Sect. 3.4.4. Next, an executable application was built, and the simulations for this exploration robot were carried out, using utility function polynomials of degree $p = 2$. Due to the simplicity of the problem, the evolutionary algorithm rather quickly found robotic brains that were able to keep the robot from colliding with obstacles for the full duration of an evaluation, while spending as much time as possible executing B_1. In the left panel of Fig. 3.23, the motion of an evolved individual is shown, seen from above. The right panel of the same figure shows the variation of the utility functions for the first 10 seconds of the evaluation. At the initial moment of the evaluation, the robot was placed right in front of an obstacle, forcing it first to activate B_2 in order to rotate to a different direction (this part of the motion is not visible in the left panel of Fig. 3.23, which only displays translational motion). Once a clear path was found, at $t \approx 2$ seconds, the robot began executing B_1. At $t \approx 5.5$ seconds, another obstacle was en-

countered, and the robot again activated B_2 for a fraction of a second, before resuming the execution of B_1.

3.3.4 Closing comments on behavioral organization

In the description of methods for behavioral organization above, the considered applications were focused on motor behaviors, i.e. behaviors involving the actuators (e.g. the wheels) of the robot. Indeed, most methods for behavioral organization have, in practice, been applied to navigation-related problems. Thus, while behavioral organization make possible the development of slightly more complex robotic brains than can easily be implemented using single behaviors, there is still a long way to go in order to reach the goal of truly intelligent behavior-based robots. An obvious step in that direction would be to allow purely mental processes, i.e. processes that do not involve the use of the robot's actuators, to run in parallel with the motion-related behaviors (which can then be organized either through cooperative methods or arbitration methods). Indeed, this is the goal of the development of the successor to the utiliy function method, currently in progress in the author's research group.

3.4 Other issues in evolutionary robotics

ER is a vast field, and this tutorial does not, of course, completely cover all aspects of the field. In the rest of this section, four different topics will be considered briefly, namely (1) ER applied to balancing, walking, and hopping robots, which constitute an increasingly important special case, (2) the relation between simulations and evolution in hardware (i.e. on actual robots) in ER, (3) simultaneous evolution of body and brain of robots, and (4) simulation software for ER.

In addition to these topics, there are many others which will not be covered in detail, such as (1) the relation between evolution and learning in ER, i.e. modification of the brain of a robot during its operation [8], [35], [96], [122], [128], [129], [145], (2) **interactive evolution**, in which the user provides a **subjective fitness measure** [60], [77], (3) **evolvable hardware** [63], [81], [126], (4) **swarm robotics** (see e.g. `www.swarm-robotics.org`) and the evolution of multi-robot (collective) behaviors [7], [93], [111], [112], [127], (5) **co-evolution**, i.e. simultaneous evolution of two (or more) populations in competition with each other, see Subsect. 2.7.3 above and e.g. [21], [36], [75], [101], [132], and (6) **behavior acquisition**, i.e. the generation of behaviors from primitive actions [143].

3.4.1 Balancing, walking, and hopping robots

In this subsection, the use of evolutionary techniques in connection with the important special case of robots equipped with legs rather than wheels will be considered, starting with a brief, general survey of walking robots.

Walking robots

Attempts at building **walking robots** can be traced back at least to the 1960s. In addition to research concerning **bipedal robots** efforts were also made to develop monopedal [113] and quadrupedal robots [41]. One of the first functioning bipedal robots was developed in the 1970s by Kato [61].

As the name implies, the term bipedal robot refers to a robot that walks on two legs, whereas the definition of the term **humanoid robot** is more loose. In general, a humanoid robot is defined as a robot with some human-like features (not necessarily the ability to walk on two legs). For example, some humanoid robots consist only of an upper body or a head (see e.g. [89]).

Some impressive examples of humanoid robots include the Honda ASIMO robots (world.honda.com/ASIMO) and the Sony QRIO robots (www.sony.net/SonyInfo/QRIO). Advanced quadrupedal robots have also been developed. A good example is the Sony AIBO pet robot (www.sony.net/Products/aibo).

There are many motivations for using bipedal robots, despite the fact that it is much more difficult to implement algorithms for reliable locomotion in such robots than in wheeled robots. First, bipedal robots are able to move in areas that are normally inaccessible to wheeled robots, such as stairs and areas littered with obstacles that make wheeled locomotion impossible. Second, walking robots cause less damage on the ground than wheeled robots. Third, it may be easier for people to interact with walking robots with a humanoid shape rather than robots with a nonhuman shape [16]. It is also easier for a (full-scale) humanoid robot to function in areas designed for people (e.g. houses, factories), since its humanlike shape allows it to reach shelves etc.

Bipedal locomotion

One of the simplest models of a walking robot is the 5-link biped introduced by Furusho and Masubuchi [42] and subsequently used by several authors, e.g. [18], [106]. This simulated robot, which is shown in Fig. 3.24, is constrained to move in the **sagittal plane** (i.e. the plane of the paper), and has five DOF. There exists many different formulations of the equations of motion for a bipedal robot, e.g. the Lagrangian formulation and the Newton-Euler formulation. The Lagrangian equations of motion for the simple five-link robot shown in Fig. 3.24 are

$$\mathbf{M}(\mathbf{z})\ddot{\mathbf{z}} + \mathbf{C}(\mathbf{z}, \dot{\mathbf{z}})\dot{\mathbf{z}} + \mathbf{N}(\mathbf{z}) + \mathbf{A}^{\mathrm{T}}\boldsymbol{\lambda} = \boldsymbol{\Gamma}, \tag{3.20}$$

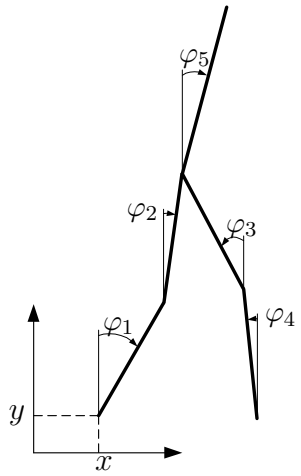


Figure 3.24: *A simple five-link bipedal walking robot, used in [18], [42], and [106].*

where $\mathbf{M}$ is the inertia matrix, $\mathbf{C}$ is the matrix of Coriolis and centrifugal forces, $\mathbf{N}$ contains gravity terms, $\mathbf{A}$ is the constraint matrix, $\boldsymbol{\lambda}$ contains the corresponding Lagrange multipliers, and $\boldsymbol{\Gamma}$ are the generalized forces. $\mathbf{z} = [\varphi_1, \ldots, \varphi_5, x, y]^{\mathrm{T}}$ are the generalized coordinates, where x and y indicate the coordinates for one foot of the robot.

The Lagrangian equations of motion have the advantage that the internal forces of constraint need not be explicitly represented in order to determine the motion of the robot. However, in general, the Newton-Euler formulation is computationally the most efficient, with the computation time growing linearly with the number of degrees of freedom. For a discussion of this formulation see e.g. [139]. In more advanced simulation models, the motion is not constrained to the sagittal plane, and the models most often include feet, arms, as well as additional DOF in the hip [2], [40], [50].

In general, there are two types of bipedal gaits: Static walking, where the projection of the centre-of-mass of the robot is kept within the area of the supporting foot, and dynamic walking where this is not the case. Static walking is easier to implement, but is usually unacceptably slow, with individual steps taking several seconds [18]. In dynamic walking, posture control based on dynamic generalizations of the concept of centre-of-mass, such as the zero-moment point (ZMP) [2] are used for generating stable bipedal gaits. The ZMP, originally introduced by Vukobratovic and Juricic [130] is the point on the ground where the torques around the (horizontal) x and y axes, generated by reaction forces and torques, are equal to zero. If the ZMP is contained within the convex hull of the support region defined by the feet (or foot, in the case of the single support phase), the gait is dynamically balanced, i.e. the robot does not fall.

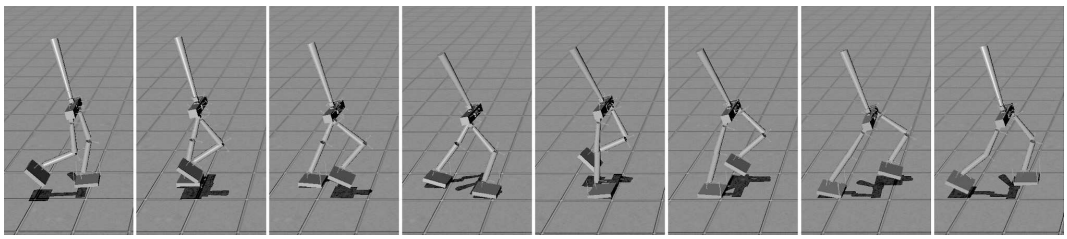


Figure 3.25: *The evolved bipedal gait obtained in [142].*

There are several different approaches to gait generation. Assuming that an adequate model of the environment can be generated, it is possible to generate bipedal gaits off-line, and then implement them in the robot [50], [54]. Another approach is to use an on-line controller, which generates the appropriate torques according to some control law [40]. For a review of bipedal and humanoid robots, see e.g. [135].

Applications of EAs in walking robots

Since walking robots are inspired by the properties of their biological counterparts, it is not far-fetched to consider biologically inspired computation methods, and in particular, EAs, when generating gaits and other behaviors for such robots. The use of EAs in walking robots is well motivated since these methods can be implemented even in cases where a complete dynamical model of the system under study is either unavailable or too complex to be useful [141].

Arakawa and Fukuda [2] used a GA to generate natural, human-like bipedal motion based on energy optimization, and Wolff and Nordin [141] implemented an evolutionary strategy to improve a hand-coded bipedal gait in a small experimental humanoid (ELVINA). Currently, researchers in the author's group are using LGP (see subsect. 2.6.2 above) for generating bipedal gaits. Pettersson et al. [106] used an EA to develop energy-optimized bipedal gaits as well as robust balancing in the presence of external perturbations. GAs have also been used for generating robust gaits both on the Aibo quadrupedal robot [52] and on robots with more than four legs [147]. Central pattern generators (CPGs), which are believed to control oscillatory processes in animals, were used in [118]. A recurrent neural network CPG was optimized using a GA, and was shown to generate a stable bipedal walking pattern. A similar approach was used in [142], where not only the network parameters but also the *structure* of the CPG networks was evolved. Thus, in [142], feedback circuits relaying information to the CPG networks regarding joint angles and ground contact (of the feet) were evolved, resulting in a gait capable of keeping a simulated, fully three-dimensional, 14-DOF robot walking indefinitely on a planar surface. The gait is illustrated in Fig. 3.25. The evolution of locomotion in one-legged hopping robots has also been considered, see Subsect. 3.3.3 above.

3.4.2 Simulations vs. evolution in actual robots

Evolving behaviors is often a time-consuming process, particularly in cases where the evaluation of a single individual takes a long time. In general, an EA needs at least several hundred evaluations (and sometimes thousands or even millions) in order to arrive at a useful result. Thus, at a first glance, evolving behaviors in simulation appear to be a more feasible approach than evolution in hardware and simulations are indeed often used in ER. However, it is important to keep in mind that a simulation is, at best, only a caricature of the real world and, at worst, an outright misrepresentation of it. This view has been championed, among other, by Brooks [15], who takes a particularly pessimistic view of the possibility of transferring results from simulations to reality. There are several reasons for believing that simulations will not transfer well to reality [86], a problem known as the **reality gap** [59]. First of all, it is difficult to generate an accurate physical model of the world, including the motions of real-world objects, variations in lighting etc. Second, real environments are invariably noisy, on all levels. Thus, noise is present in sensor readings, actuator signals, motion etc. Third, there is also a variation in supposedly identical hardware components. For example, Miglino et al. [86] mention a measurement on two sensors of the same type, in which it was found that the range of one sensor was nearly twice as large as the other, and the diffence in angular variation between the two sensors was also of the same magnitude.

Evolution in hardware

The obvious alternative to evolving behaviors in simulation is, of course, to evolve in physical robots, and this approach has been used by many authors (see e.g. [33], [34], [52], [53], [140], [141]). However, evolution in hardware is also associated with numerous difficulties. First of all, the evaluation of individuals generally takes longer time than in actual robots than in simulated robots (see e.g. [34] or [52] for timing estimates). In order to arrive at a result in a reasonable amount of time, it is therefore often necessary to resort to an EA with a small population size, and to run it only for a small number of generations [141]. Second, supplying the robots with continuous power is an omnipresent problem: In real robots, the amount of available energy is finite, and so recharging will be necessary at regular intervals. This problem can be overcome by the introduction of a powered floor [33], or by requiring the robots periodically to return to a charging station. Charging batteries is a rather slow process, but the transfer of energy to robots can be speeded up using capacitors instead of batteries [102]. Third, in many applications, e.g. the evolution of gaits for bipedal robots, the evolutionary process must be monitored continuously. An example of such an application is a project, currently in progress in the author's group, aimed at optimizing a simple gait in a

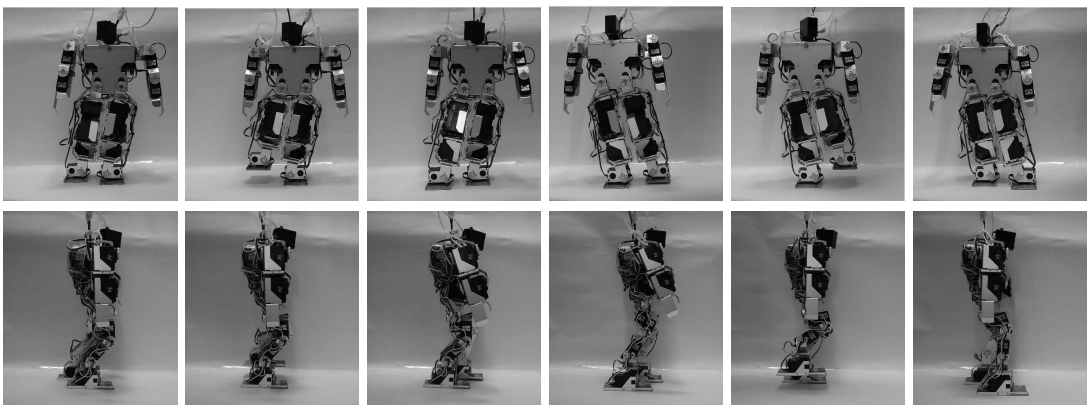


Figure 3.26: *A Kondo robot in action. This particular gait is statically stable and consists of six states, shown from the front and from the side.*

small, commercially available bipedal robot, manufactured by Kondo Kagaku Co., Ltd., (`www.kondo-robot.com`). A typical gait displayed by this robot is shown in Fig. 3.26. The most common approach to evolution in hardware is to evaluate individuals in a serial fashion, uploading them one after the other in a single robot [34], [94]. In this case, all evolutionary processes, such as e.g. reproduction and formation of new individuals take place on a computer, and the resulting individuals are then placed in the robot for evaluation.

An alternative method, called **embodied evolution** [33], [140] is to evolve a population of real robots. In embodied evolution, infrared communication can be used for exchanging genetic material between robots. Such exchanges take place only when two robots are within infrared communication range. The probability of a robot sending a gene, can be set proportional to its current performance, and the probability of a robot accepting a gene from another robot (and thus to overwrite its own gene), can be set proportional to one minus the probability of sending the gene.

Improving simulations

The comparison between simulations and evolution in hardware shows that there are advantages and disadvantages with both approaches. A substantial amount of work has been carried out in order to improve the quality of simulations (see e.g. [56], [57], [58], [59], [81], [94], [97], and [99]). Some of the most important aspects to keep in mind when setting up ER simulations [59] is to (1) base the simulation on empirical data, rather than e.g. artificial sensor models without a real-world counterpart, (2) add the correct level of noise (see below) in all parts of the simulation, (3) use a representation that is noise-tolerant, e.g. an ANN.

Sensors can be implemented in several different ways in ER simulations. One approach is simply to measure sensor readings on an *actual* robot, and store them (e.g. in a lookup table) for later use in a simulator [86], [99]. Noise can be added in different ways, either by slightly perturbing the measured sensor readings, or by using actual sensor readings taken from a slightly different position and angle than that currently held by the robot, a procedure called **conservative position noise** [86]. In this approach, sensor readings must be taken at many different positions, rendering the procedure quite time-consuming in all but the simplest environments. An alternative approach is to set up a physical model of the sensors, and to determine the values of the model parameters through system identification. For example, Jakobi et al. [59] used this approach to model the IR sensors on a Khepera robot (see Appendix C). They used a ray tracing technique which in which n ray tracings were carried out (in different directions), and the resulting sensor reading was then modelled as

$$s = \sum_{i=1}^{n} \cos \beta_i \left(\frac{a}{d_i^2} + b \right), \tag{3.21}$$

where β_i is the angle at which ray i emanates from the sensor, and d_i is the distance to an object along the ray. a and b were determined empirically [59].

Jakobi et al. [59] also studied the effects of noise, and found that the smallest performance difference between simulated robots and physical robots was obtained when the noise level in simulations was set approximately equal to the empirically determined noise level in the real world. Interestingly, it was also found that, in simulations with high noise levels, simulated robots could make use of noise to achieve good results in simulations that could not be reproduced in physical robots, showing that noise levels should not be set too high.

Jakobi [56], [58] has introduced a simulation procedure called **minimal simulations**, which recognizes the inevitable discrepancies between the simulated and real worlds. Thus, in a minimal simulation, the various aspects of the interaction between a robot and its environment are divided into **base set aspects** and **implementational aspects**, where the former have a physical counterpart in the real world, whereas the latter do not. Thus, in a minimal simulation, the implementational aspects should be varied randomly from evaluation to evaluation, thus rendering them useless to evolution, and forcing the EA to focus on the base set aspects. An evolved robotic brain that only relies on base set aspects is called **base set exclusive**. In addition, a certain degree of variation is introduced also in the base set aspects, in order to capture the fact that even in the base set aspects there will be discrepancies between the simulated and real worlds. A robot that can cope also with such discrepancies, is termed **base set robust**.

Thus, summarizing, it is evident that the choice between evolution in sim-

ulation and evolution in hardware is a non-trivial one. Simulations have the disadvantage of never being able to capture all aspects of the real world. However, carefully designed simulations will nevertheless lead to results that can be transferred to physical robots. Indeed, one may, perhaps, consider the results obtained in a simulation as a first iteration toward the desired results. Of course, a hybrid approach can be used, in which evolution is first carried out in simulations and then briefly continued in hardware. Miglino et al [86] report successful results from such an approach.

3.4.3 Simultaneous evolution of body and brain

The most common application of ER is to evolve the brain of a robot, adapting it for use in a body of fixed size and shape. However, there is nothing preventing the use of EAs for optimizing both the body and the brain of a robot.

Simultaneous optimization of body and brain has been considered in artificial life, where Sims [119] evolved swimming, running, and hopping creatures, consisting of limbs composed of chains of block-like structures and neural networks for moving the limbs in an oscillatory fashion. For a recent review, and also an extension of Sims' work, see [124]. The procedure of simultaneously optimizing body and brain has been used also by several authors in the field of ER [5] A brief review of the topic can be found in [98].

In many such applications, see e.g. [11], [21], [24], [25], [82], [132], and [133], an **artificial ontogeny** (or **morphogenesis**) is employed, i.e. the body and brain of the individual are generated as the result of a developmental process, modelled on the embryological development of biological organisms, involving gene regulation and differential gene expression both in space and time.[6] In these cases, the encoding scheme used in the EA is, in general, much more complex than the simple lookup-table coding used in many other applications. A simple example of an encoding scheme involving an element of ontogeny is the grammatical encoding scheme described in Subsect. 2.7.2 above.

The ability of EAs to optimize the morphology of robots opens up a plethora of new possibilities, and reduces the severity of the problem of selecting an appropriate body structure *a priori*. As a simple example, in the pursuit-evasion contests studied in [132] and [133], it was found that **pursuers** evolved forward-pointing sensors, whereas **evaders** evolved backward-pointing sensors. While these investigations were carried out in simulation, Lichtensteiger and Eggen-

[5] Some authors use the term *co-evolution* to indicate simultaneous evolution of body and brain. However, in order not to confuse this process with co-evolution involving different, competing species, see Subsect. 2.7.3 above, the term co-evolution should, perhaps, be avoided in this context.

[6] For an introduction to gene regulation in biological systems, see e.g. [22] and [109].

Figure 3.27: *A crawling robot evolved by Lipson and Pollack [74]. The left panel shows the simulated robot, and the right panel shows the corresponding physical robot. Reproduced with kind permission of Dr. H. Lipson.*

berger [73] successfully evolved the positioning of sensors forming a compound robotic eye on a physical robot.

Complex artificial morphological processes, involving artificial cell division and gene regulation, have been developed [11], [24], [25], [82]. However, so far, the use of artificial developmental processes has been limited to rather simple applications.

It is also possible to evolve body and brain using a direct encoding scheme (i.e. bypassing complex developmental processes) [10], [28], [76], [103]. For example, Bongard and Paul [10], [103] evolved the shape of a bipedal robot, i.e its mass distribution, moments of inertia and, in some cases, its total mass, together with the brain (implemented in the form of an RNN). It was found that the addition of a variable morphology improved the performance of the EA, at least in some cases, despite the larger search space compared to the fixed-morphology case. The improved performance was attributed to the modification of the fitness space resulting from the additional evolutionary degrees of freedom.

In an interesting series of experiments, the research group of Jordan Pollack has evolved simple moving creatures composed of rods, joints, and actuators, controlled by a neural network, and somewhat similar to the virtual creatures evolved by Sims [119]. However, in the experiments performed by Pollack's group [74], the evolved creatures were also given physical form using techniques for automatic three-dimensional printing. An example is shown in Fig. 3.27. The generated structures were manually fitted with motors, resulting in simple robots that were able to reproduce the behaviors found in simulation.

3.4.4 Simulation software

Since ER often involves the use of simulations, it is not surprising that many simulation packages (i.e. collections of source code that can be used for generating simulator for autonomous robots) and stand-alone robotic simulators have been developed. A few tools for ER simulations will now be introduced.

EvoRobot

The `EvoRobot` program is a simulator for Khepera robots, available on the world wide web at
`http://gral.ip.rm.cnr.it/evorobot/simulator.html`.
EvoRobot allows the user to perform various ER experiments, such as evolving neural networks for navigation and obstacle avoidance. A variety of different robot parameters, such as the number and types of sensors, can be set through the graphical user interface. Evolution can be carried out both in simulation and using an actual Khepera robot. If simulations are used, the results can also be uploaded and tested in an actual Khepera robot.

Evolutionary robotics toolbox

The `Evolutionary robotics toolbox` (ERT) is a software package for Matlab for simulation of Khepera robots. As in the `EvoRobot` package, the results from a simulation carried out in ERT can be tested on an actual Khepera robot. The package can be downloaded from
`http://diwww.epfl.ch/lami/team/urzelai/ertbx.html`.

Webots

`Webots`™ is a commercially available software package [85] manufactured by Cyberbotics (www.cyberbotics.com). `Webots` allows simulation of many different types of robots, including Khepera, Koala (manufactured by K-team, www.k-team.com), and Aibo (manufactured by Sony, www.sony.com/aibo). As with the other robotics packages mentioned above, the results obtained from a `Webots` simulation can be uploaded to an actual robot. In addition `Webots` supports, for instance, evolution of robot morphology and multi-robot simulations (such as robot soccer). `Webots` is equipped with an advanced graphical user interface, allowing 3D rendering of simulated robots (see Fig. 3.28). Furthermore, it is possible to import 3D models deleveloped in 3D modelling languages such as VRML.

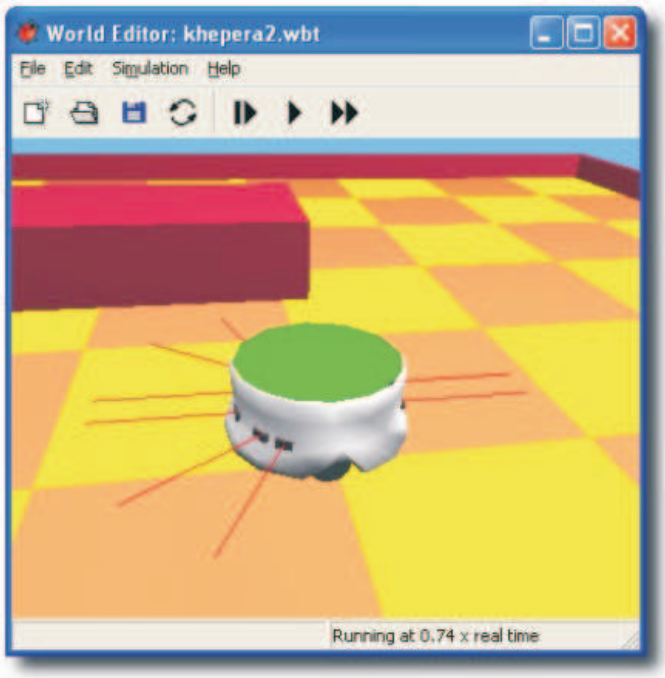


Figure 3.28: *A screenshot from a Webots™ simulation of a Khepera II robot. Reproduced with kind permission of Dr. O. Michel, Cyberbotics.*

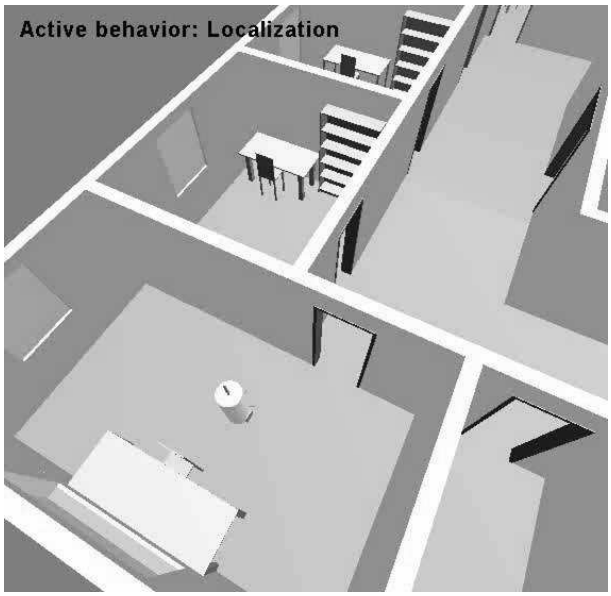


Figure 3.29: *A screenshot from a simulation based on the UFLibrary software package. This particular environment has been used in a project (in the author's research group) aimed at generating a behavior selection system for a delivery robot (shown as a cylindrical object).*

Darwin2K

`Darwin2K` [72] is an open source packages for robotic simulation, containing an evolutionary algorithm for ER simulations. Like `Webots`, `Darwin2K` includes advanced robot models and has, among other things, been used in connection with research on planetary exploration robots. The Darwin2K package is available at `darwin2k.sourceforge.net`.

Utility function library

The author's research group is currently developing a general simulation package, the utility function library or `UFLibrary` for ER applications involving behavioral organization using the utility function method (see Subsect. 3.3.3). The utility function library has been developed using Delphi object-oriented Pascal (www.borland.com), and the first released version (v1.0.1), together with a tutorial and reference manual, can now be downloaded from the internet [134]. At present, the `UFLibrary` consists of a total of more than 12,000 lines of code, divided into 51 source units, ranging from general units concerning e.g. the evolutionary algorithm used for the optimization of utility functions to more specific units such as those defining, for example, a DC motor or a laser range finder. In addition, a few basic behaviors, such as a navigation behavior, an obstacle avoidance behavior etc. have been included in the `UFLibrary`.

UFLibrary makes possible the construction of robotic brains of arbitrary complexity, provided that certain guidelines (as enforced by the utility function method) are followed. The functionality is provided in the form of a software package rather than as a stand-alone application. Although some programming is required to make full use of the `UFLibrary`, for example to define new behaviors or to define a new stand-alone application based on the package, it is possible, for basic usage, to control most aspects of the simulations through the use of definition files in the form of simple text files. When implementing an application, a minimum of two definition files are required, one for the world (arena) in which the robot is supposed to move and operate, and one for the robot itself. The definition files have an object-oriented structure, and provide an intuitive interface, even to users who have limited programming experience. More detailed information concerning the use of the `UFLibrary` can be found in [134]. Also, the `UFLibrary` and the related documents can be found on the CD accompanying this tutorial. A screenshot of a typical environment for a robotics simulation using this software is shown in Fig. 3.29.

It should be noted that a new version of the utility function method is being developed. Concurrently, a new version of the software is also being written, in which behaviors will be represented in a unified way, such that new behav-

iors can be added through simple text files (definition files). Thus, in the new version, the user need not carry out any programming in the traditional sense. In addition, the generation of the definition files will eventually be based on a graphical user interface, further improving the user friendliness of the method. For further information regardng the utility function method and the associated software library, please contact the author (mattias.wahde@chalmers.se).

Robotics studio

Recently, Microsoft has entered the field of robotics with its ambitious software package Robotics Studio. The aim of Robotics Studio is to allow users to create software for robotics applications, and to simulate robots in highly realistic 3D environments. In addition, Robotics Studio allows users to control physical robots through, for example, a web browser. For a full description of this software package, see `msdn.microsoft.com/robotics/`.

3.5 The future of evolutionary robotics

Predicting the future is a difficult business. Long-term investigations of such predictions often show a general pattern: Forecasters are too optimistic in their short-term (5-10 years, say) predictions, and too pessimistic in their long-term predictions (50-100 years, say). Furthermore, new inventions that cannot even be imagined today are certain to appear, at least in the long term, rendering the task of predicting the future even more difficult.

Some long-term predictions concerning robotics in general (i.e. not specifically ER), can be found e.g. in [17], [55], and [90]. An interesting observation made by Moravec [90] is that, since around 1990, the growth of computational power available to robotics (and AI in general) approximately follows Moore's law, i.e. a doubling of computational power every 18 months. This was not so before 1990, due to decreasing funding for AI and robotics, which almost precisely offset the computational gains resulting from Moore's law. However, during the 1990s and early 2000s computers and computer parts have become so cheap that experiments in robotics (and other related fields) can be carried out on a very limited budget.

This trend is likely to remain in place, and Moravec even extrapolates to 2040, when (if Moore's law can be sustained) robots and computers should be able to perform 100 million million instructions per second (100 million MIPS), which is roughly equivalent to the computational speed of the human brain.

Clearly, computing power alone is not sufficient for generating robots with intelligent behavior, but it certainly helps.

Furthermore, there are many autonomous robots that are soon to be released or are already available on the market, particularly in Japan, where pet

Figure 3.30: *The PaPeRo personal robot. © NEC. Reproduced with kind permission of the NEC corporation.*

robots such as AIBO, and humanoid robots such as ASIMO, have had great success. The fact that robots such as AIBO are mass produced is also important: The possibility of making profits from the construction of autonomous robots is, of course, an essential driving force in their development.

So far, the capabilities of commercially available robots are quite limited, even though they are important precursors for more advanced robots. In addition to entertainment robots such as AIBO, commercially available robots include lawn moving robots, vacuuming robots, and robots for autonomous transportation in hospitals. There are also prototypes for general-purpose household robots, such as the NEC PaPeRo robot shown in Fig. 3.30.

Returning now the ER, some general trends can be observed. Note that the following paragraphs reflect the author's view. For an alternative (but not necessarily conflicting) view of the short- to intermediate-term future of ER, see e.g. [37].

First of all, a result of the increasing complexity of robotic brains is to reduce further the possibility of generating such systems by hand. Thus, if anything, ER is likely to become *more* useful as complexity grows. This can be seen e.g. in recent EA-based methods for behavioral organization [131]. Indeed, behavioral organization is of paramount importance in for ER to be able to deliver robots with the capabilities needed for autonomous operation in realistic environments. Current research efforts in this field show that EA-based methods indeed are able to generate complex robotic brains, with a minimum

of hand-coding by the experimenter [107].

Furthermore, it is likely that the EAs used for ER (and also for other applications) will become increasingly sophisticated incorporating, among other things, artificial gene regulation and ontogeny, as described in Subsects. 2.7.2 and 3.4.3, to a greater degree than today. Indeed, the march towards more complex EAs is inspired, in part, by the rapid developments in molecular biology [22] and bioinformatics [12]. In particular, the advent of massively parallel gene expression measurements using **microarray** techniques [110] has led to a rapid increase in the amount of biological data available, and the cooperation of biology and computer science in bioinformatics continues to deliver new information concerning the organization and function of biological systems, information that is likely to be assimilated also in EAs.

In addition, the simultaneous evolution of body and brain is likely to become increasingly important in ER. In biological systems, brains and bodies have, of course, evolved together, and form an integrated system. Thus, despite the increasing complexity introduced by variable morphologies, it is possible that simultaneous evolution of body and brain may, in fact, speed up evolution [10], [103].

Finally, it is likely that there will appear many new multi-robot applications, in which the evolutionary process must also take into account the communication between robots [80], [144], as well as the communication between robots and people [39]. A particularly interesting topic is the emergence of complex, self-organized collective behavior from simple, local interactions, akin to **stigmergy** [125] in social insects such as ants and termites.

To summarize, it is evident that the influence from molecular biology, genetics, and ethology will continue to play an important role in ER and in the near future, increasingly advanced robots, based in whole or in part on ER, will in all probability appear on the market, ready and able to help people with a variety of tedious and dangerous tasks.

Appendix A: Artificial neural networks

The aim of this appendix is to give a very brief introduction to neural networks, introducing only those parts of this vast topic that will be used in this tutorial. For a more thorough introduction see [48]. Artificial neural networks (ANNs) are computational structures (loosely) based on the structure and function of biological neural networks that constitute 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. A1. 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 input, 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 continuous time. In the former case, the output of neuron i in a network, denoted x_i is computed as

$$x_i = \sigma\left(\sum_j w_{ij} z_j + b_i\right), \tag{A1}$$

where $w_i j$ 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

$$\tau_i \dot{x}_i(t) + x_i(t) = \sigma\left(\sum_j w_{ij} z_j + b_i\right), \tag{A2}$$

where τ_i are time constants. Typically, the z_j are inputs from other neurons (or even the neuron i itself), so that the equations for different neurons are joined into a set of coupled non-linear differential equations. Other equations are also possible, of course. In any case, the neuron equations presented above

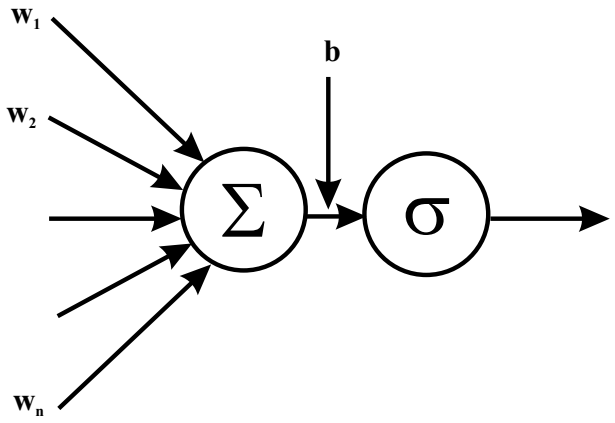


Figure A1: *An artificial neuron. First, the input signals, multiplied by the weights, are added. Next, the bias b is added, and the output is computed using the squashing function σ.*

represent very a strong simplification of the very intricate method of operation of actual biological neurons. The squashing function can be chosen in various different ways. Two common choices are the **logistic function**

$$\sigma(z) = \frac{1}{1 + \mathrm{e}^{-cz}}, \tag{A3}$$

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

$$\sigma(z) = \tanh cz, \tag{A4}$$

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 layer to the first **hidden layer**, from the first hidden layer to the second hidden 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 continuous time, using Eq. (A2) to represent neurons. The difference between FFNNs and RNNs is illustrated in Fig. A2. For the FFNN, shown in the left panel of the figure, the input signals are denoted I_j, the neurons in the middle 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

$$x_i^{\mathrm{H}} = \sigma\left(\sum_j w_{ij}^{\mathrm{IH}} I_j + b_i\right), \tag{A5}$$

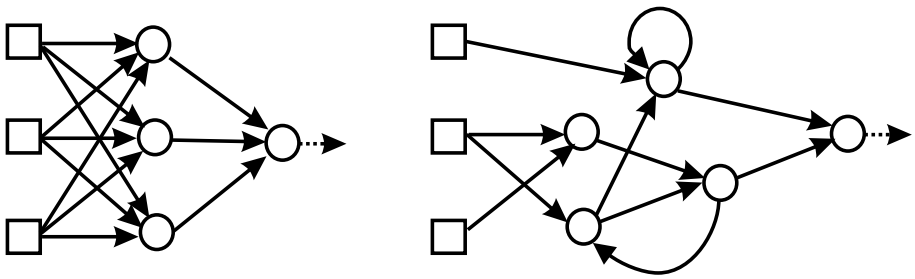


Figure A2: *The left panel shows a feedforward neural network (FFNN), whereas the right panel shows a recurrent neural network (RNN), in which feedback connections are present.*

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

$$x_i^{\mathrm{O}} = \sigma\left(\sum_j w_{ij}^{\mathrm{HO}} x_j^{\mathrm{H}} + b_i\right), \tag{A6}$$

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** can be used to refer either to input elements or to neurons.

An RNN is shown in the right panel of Fig. A2. Here, neurons are not (necessarily) arranged in layers, and the output of a typical neuron is given by

$$\tau_i \dot{x}_i(t) + x_i(t) = \sigma\left(\sum_j w_{ij} x_j(t) + \sum_j w_{ij}^{\mathrm{I}} I_j(t) + b_i\right), \tag{A7}$$

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 RNN are 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 [48]. This training algorithm requires 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.

However, supervised training methods are rarely applicable in ER, since immediate rewards do not normally occur. Instead, the reward (if any) for a given action generally occurs long after the action was taken. In addition backpropagation, as indeed many other methods developed in the neural network literature, require that the structure of the ANN be specified in advance. In ER it is generally impossible to specify the best structure of a ANN-based robotic brain beforehand. Thus, for these reasons, EAs are often the method of choice when generating ANNs in ER and, in this tutorial, only EA-based generation of ANNs will be considered.

Appendix B: Finite-state machines

In addition to neural networks (see Appendix A), **finite-state machines** are sometimes used for representing robotic brains in connection with ER.

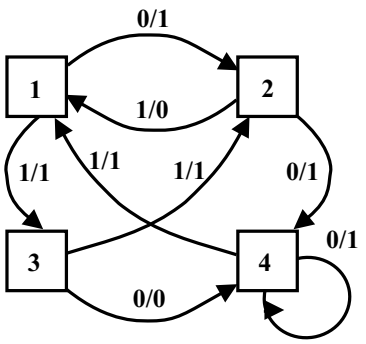
A **finite-state machine** [38] (hereafter FSM) consists, as the name implies, of a finite number of states and **conditional transitions** between those states. At every time step, an FSM reads an input symbol, and produces an output symbol as it jumps to the next **target state**.

In FSMs, the input and output symbols are taken from two discrete alphabets. Commonly, the **input alphabet**, i.e. the symbols received by the FSM, is the same as the **output alphabet**, i.e. the symbols generated by the FSM.
A simple FSM is shown in Fig. B1. In this case, the input and output alphabets both consist of the symbols 0 and 1, and the input-output mapping is shown in the right panel of the figure. Unlike FFNN (see Appendix A), FSMs can be equipped with a dynamic memory of previous events, and can thus be used e.g. to generate robotic brains that reach beyond the realm of purely reactive behaviors.

FSMs, as introduced above, operate with discrete variables, however, and are therefore not always suitable in connection with autonomous robots, which, of course, operate in continuous time. A useful extension to FSMs are **generalized finite-state machines** (GFSMs) [137]. Like ordinary FSMs, GFSMs consist of a finite number of states and transitional conditions between the states. However, in GFSMs, each state consists of a specific setting of (some of) the variables of the system under study. For example, in the case of a Khepera robot (see Appendix C), the variables specified in a state may be the speeds of the two motors.

From each state i, there are M_i conditional transitions to other states. In the case of autonomous robots, the conditions often involve the sensor readings, but other variables can also be used. The conditions can, in principle, be arbitrarily complex, and are thus not restricted to a particular alphabet, as in FSMs. If a condition is satisfied, the GFSM immediately jumps to the corresponding target state. No particular action is associated with the jump from one state to another. If none of the conditions are satisfied, the GFSM remains in the same state, as indicated for state 1 in Fig. B2.

Thus, in GFSMs, each state is associated with some specific variable setting,



state (before)	input	output	state (after)
1	0	1	2
2	1	0	1
1	1	1	3
3	0	0	4
4	1	1	1
1	0	1	2
⋮	⋮	⋮	⋮

Figure B1: *A simple finite-state machine.*

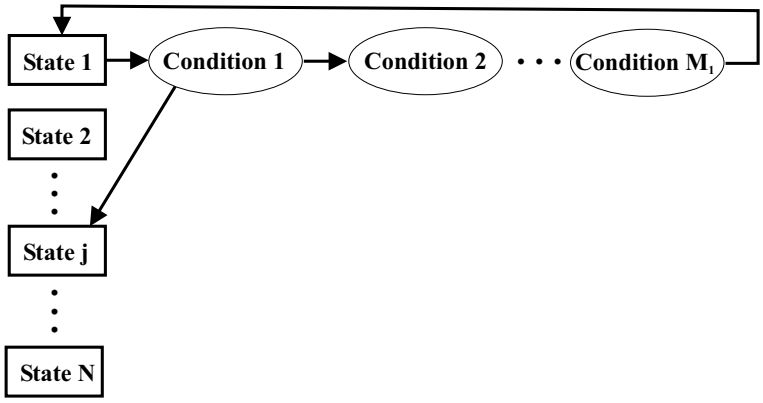


Figure B2: *A GFSM. There are N states and, for each state i, there are M_i conditional transitions (which differ from state to state). Only one conditional transition is shown in the figure, namely from state 1 to state j.*

whereas in ordinary FSMs it is the state transitions that are associated with the actions. This reduces the number of possible variable settings for GFSMs compared to FSMs for a given state machine size since, normally, there are more transitions than states. However, it also makes the architecture more transparent and easy to interpret.

Appendix C: The Khepera robot

During the last few decades, a large number of autonomous robots have been developed, both by robotics researchers and, more recently, by companies such as Sony, Honda etc. In this appendix, the properties of one commercially available robot, namely the **Khepera robot** developed by K-team (www.k-team.com) will be described briefly. The Khepera robot has, for many years, been a popular tool in ER, due to the facts that it is very compact and light (and thus easy to transport), easy to program, and equipped with basic sensory capabilities. In addition, the fact that the robot is so widely used, as evidenced by the many examples in Chapter 3, facilitates the comparison of the results obtained by different research groups. The also exists several simulators for Khepera (see Subsect. 3.4.4), some of which allow evolved robotic brains to be uploaded directly onto an actual Khepera robot.

Khepera is a small differentially steered robot, equipped with two wheels and eight infrared sensors in the standard configuration. Several additional modules, such as linear vision arrays and gripper arms, can be added to the standard Khepera configuration. Two versions of Khepera exist: the original version (called Khepera I in the remainder of this appendix) with a diameter of around 55mm, and an improved version (Khepera II), with a diameter of around 70mm. The weight of the robot is approximately 80g (Khepera II).

Pictures of Khepera I and Khepera II are shown in Fig. C1. The robot is equipped with a processor from the Motorola 68K family, and has a RAM memory of 256 Kb (Khepera I) or 512 Kb (Khepera II).

The robot can be programmed in C, and the communication between the computer and the robot is carried out via a serial cable. Once a program has been downloaded into a Khepera robot, the cable can be removed and the robot is then fully autonomous. Each Khepera robot carries four rechargeable NiCd batteries, which allow the robot to operate for around 45 minutes (Khepera I) to one hour (Khepera II). However, for longer operation, the robot can be powered via the serial cable.

The robot is equipped with two DC motors, one on each side. The low-level control of both motors is handled by PID controllers, which allow the user to set either a speed or a position, without worrying about the details of the motor control. The top speed of the robot is around 1 m/s. It is possible

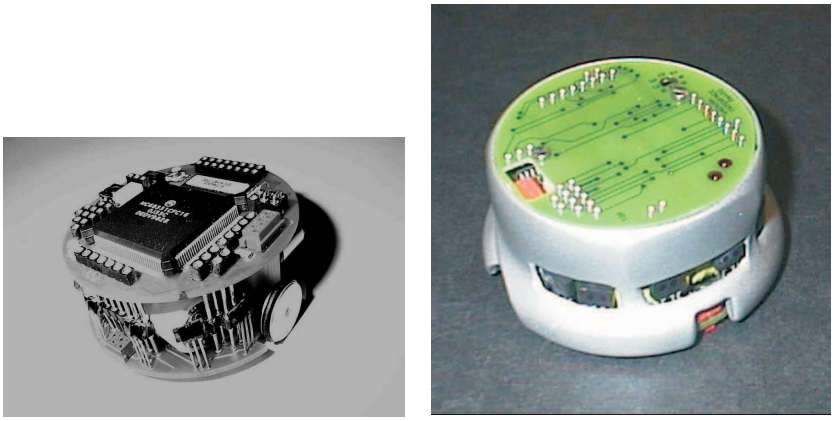


Figure C1: *The Khepera family: Khepera I (left panel) and Khepera II (right panel). Photos by the author.*

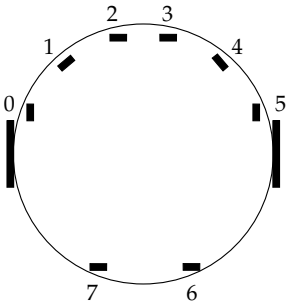


Figure C2: *A schematic view of the Khepera robot. The infrared sensors are numbered from 0 to 7.*

also to override the motor controllers and take more direct control over the motion of the robot.

As shown in the schematic view of the robot (Fig. C2), the robot has eight infrared proximity sensors numbered from 0 (just in front of the left wheel in Fig. C2) to 7 (behind the left wheel). The sensors can be used in two modes, either to measure the ambient light levels (using only the reception of infrared light) or to measure the light reflected by objects surrounding the robot (by emitting infrared light and measuring the reflection). The sensors update their readings every 20 ms, allowing high resolution (in time) of the measurements. The readings for each sensor are integer values between 0 and 1023.

The sensor range for measurement of reflected light varies with the reflectivity of the object in question, but is around 20–50 mm (i.e. of the same order of magnitude as the size of the robot itself) for Khepera I and up to 100 mm for Khepera II. It should be noted that the Khepera is rather sensitive to the amount of ambient light present, and can easily be blinded if a strong light source is suddenly turned on.

Bibliographical notes

These tutorial notes are, of course, intended as an introduction to the ER literature. However, the reader should also consult the original sources, given in the reference list below. Because of the length of the reference list, however, readers who are new to the field of ER may be helped by some guidance, especially since many of the references do not directly deal with ER but instead with closely related topics, such as BBR, ANNs etc.

Thus, in addition to these tutorial notes, the reader is recommended to begin by considering the book on evolutionary robotics by Nolfi and Floreano [97], as well as review papers such as [46], [47], [83], and [99]. It is also recommended to read introductory texts on behavior-based robotics [3], ethology [30], and molecular biology [22], since all these fields have a strong influence on research in ER.

Furthermore, despite its length, the reference list below is only a partial (and probably biased) sample of the many papers available in ER. Thus, the reader is recommended to consult journals in which ER-based research appears, e.g. Adaptive Behavior, Artificial Life, Evolutionary Computation, IEEE Transactions on Evolutionary Computation, IEEE Transactions on Systems, Man, and Cybernetics, and Robotics and Autonomous Systems, as well as the proceedings of the many conferences in which ER is often featured, e.g. Simulation of Adaptive Behavior (SAB), Artificial life (ALIFE), Evolutionary Robotics (ER), Genetic and Evolutionary Computation (GECCO), Intelligent Robots and Systems (IROS), Robotics and Automation (ICRA), and Systems, Man, and Cybernetics (SMC). The author maintains a regularly updated list of conferences in ER and related fields, which is available at
`http://www.me.chalmers.se/~mwahde/ConferenceList.php`

Bibliography

[1] Andreasson, N., Evgrafov, A., and Patriksson, M. *An introduction to continuous optimization: foundations and fundamental algorithms*, Studentlitteratur, 2005

[2] Arakawa, T and Fukuda, T. *Natural motion trajectory generation of biped locomotion using genetic algorithm through energy optimization*, In: Proc. of the 1996 IEEE Int. Conf. on Systems, Man, and Cybernetics, pp. 1495-1500, 1996

[3] Arkin, R.C. *Behavior-based robotics*, MIT Press, 1998

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

[5] Bäck, T., Hoffmeister, F., and Schwefel, H.-P. *A Survey of Evolution Strategies*, In: Proc. of the 4^{th} Int. Conf. on Genetic Algorithms, Morgan Kaufmann Publishers, pp. 2-9, 1991

[6] Banzhaf, W., Nordin, P., Keller, R.E., and Francone, F.D. *Genetic Programming - An Introduction*, Morgan Kaufmann Publishers, 1998

[7] Bassett, J.K. and De Jong, K.A. *Evolving Behaviors for Cooperating Agents*, In: Proc. of the 12^{th} Int. Symp. on Methodologies for Intelligent Systems, pp. 157-165, 2000

[8] Belew, R. K. and Mitchell, M. *Adaptive Individuals in Evolving Populations: Models and Algorithms*, Santa Fe Institute Studies in the Science of Complexity, **26**, pp. 1-22, 1996

[9] Blumberg, B.M. *Action-selection in Hamsterdam: lessons from ethology*. In: From Animals to Animats 3: Proc. of the 3^{rd} Int. Conf. on Simulation of Adaptive Behavior (SAB94), MIT Press, 1994

[10] Bongard, J.C. and Paul, C. *Making Evolution an Offer It Can't Refuse: Morphology and the Extradimensional Bypass*, Lecture Notes in Computer Science, **2159**, pp. 401-412, 2001

[11] Bongard, J.C. and Pfeifer, R. *Evolving Complete Agents using Artificial Ontogeny*, In: Morpho-functional Machines: The New Species, Springer-Verlag, 237-258, 2003

[12] Bower, J.M. and Bolouri, H. (Eds.) *Computational Modeling of Genetic and Biochemical Networks*, MIT Press, 2001

[13] Brameier, M. *On linear genetic programming*, PhD thesis, Dortmund University, Dortmund, Germany, 2003

[14] Brooks, R.A. *A Robust Layered Control System for a Mobile Robot*, IEEE Journal of Robotics and Automation, **RA-2**, No. 1, pp. 14-23, 1986

[15] Brooks, R.A. *Artificial life and real robots*, In: Proc. of the first European conference on artificial life (ECAL91), MIT Press, pp. 3-10, 1992

[16] Brooks, R.A. *Prospects for human level intelligence for humanoid robots*, In: Proc. of the First International Conference on Humanoid Robots (HURO-96)

[17] Brooks, R.A. *Robot: The Future of Flesh and Machines*, Penguin Press, Paperback Ed., 2003

[18] Cheng, M.-Y. and Lin, C.-S. *Genetic algorithm for control design of biped locomotion*, In: Proc. of the IEEE Int. Conf. on Robotics and Automation, pp. 1315-1320, 1995

[19] Cliff, D., Harvey, I., and Husbands, P. *Explorations in Evolutionary Robotics*, Adaptive Behavior **2**:1, pp. 73-110, 1993

[20] Cliff, D., Husbands, P., and Harvey, I. *Evolving visually guided robots* In: Proc. of the 2^{nd} Int. Conf. on Simulation of Adaptive Behavior (SAB92), MIT Press/Bradford books pp 374–383, 1993

[21] Cliff, D. and Miller, G.F. *Co-evolution of pursuit and evasion II: Simulation methods and results*, in Proc. of the 4^{th} Int. Conf. on the Simulation of Adaptive Behavior (SAB96), MIT Press, pp. 506-515, 1996

[22] Davidson, E.H. *Genomic Regulatory Systems - Development and Evolution*, Academic Press, 2001

[23] Dawkins, R. *Climbining Mount Improbable*, Penguin books, 1997

[24] Dellaert, F. and Beer, R.D. *A Developmental Model for the Evolution of Complete Autonomous Agents*, In: Proc. of the 4^{th} Int. Conf. on Simulation of Adaptive Behavior (SAB96), MIT Press, pp. 393-401, 1996

[25] Dellaert, F. and Beer, R.D. *Toward an evolvable model of development for autonomous agent synthesis*, In: Proc. of the 4^{th} Int. Conf. on Artificial Life, MIT Press, 1994

[26] Dorigo, M. and Stützle, T. *Ant Colony Optimization*, MIT Press, 2004

[27] Duckett, T. *A Genetic Algorithm for Simultaneous Localization and Mapping*, In: Proc. of the 2003 Int. Conf. on Robotics and Automation (ICRA2003), pp. 434-439, 2003

[28] Endo, K., Maeno, T., and Kitano, H. *Co-evolution of Morphology and Walking Pattern of Biped Humanoid Robot using Evolutionary Computation - Evolutionary Designing Method and its Evaluation*, In: Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, pp. 340-345, 2003

[29] Erinc, G. and Carpin, S. *A genetic algorithm for nonholonomic motion planning*, In: Proc. of the IEEE 2007 Int. Conf. on Robotics and Automation (ICRA2007), pp. 1843-1849, 2007

[30] McFarland, D. *Animal Behaviour*, Addison-Wesley, 1999

[31] McFarland, D., and Bösser, T. *Intelligent Behavior in Animals and Robots*, MIT Press, 1993

[32] McFarland, D. and Spier, E. *Basic cycles, utility, and opportunism in self-sufficient robots*, Robots and Autonomous Systems, **20**, pp. 179-190, 1997

[33] Ficici, S.G., Watson, R.A., and Pollack, J.B. *Embodied Evolution: A Response to Challenges in Evolutionary Robotics*, In: Proc. of the 8^{th} European Workshop on Learning Robots, pp. 14-22, 1999

[34] Floreano, D. and Mondada, F. *Evolution of homing navigation in a real mobile robot*, IEEE Transactions on Systems, Man, and Cybernetics, Part B, **26**, pp. 396–407, 1996

[35] Floreano, D. and Mondada, F. *Evolution of plastic neurocontrollers for situated agents*, In: Proc. of the 4^{th} Int. Conf. on Simulation of Adaptive Behavior (SAB96), pp. 402-410, 1996

[36] Floreano, D., Nolfi, S., and Mondada, F. *Competitive Co-Evolutionary Robotics: From Theory to Practice*, In: Proc. of the 5^{th} Int. Conf. on Simulation of Adaptive Behavior (SAB98), pp. 515-524, 1998

[37] Floreano, D. and Urzelai, J. *Evolutionary Robots: The Next Generation*, In: Proc. of the 7^{th} Int. Symp. on Evolutionary Robotics (ER2000), 2000

[38] Fogel, L.J. *Intelligence through Simulated Evolution*, Wiley, 1999

[39] Fong, T., Nourbakhsh, I., and Dautenhahn, K. *A survey of socially interactive robots*, Robotics and Autonomous Systems **42**, pp. 143-166, 2003

[40] Fujimoto, Y. and Kawamura, A. *Simulation of an autonomous biped walking robot including environmental force interaction*, IEEE Robotics and Automation Magazine **5**, No. 2, pp. 33-42, 1998

[41] Furusho, J., Akihito, S., Masamichi, S., and Eichi, K. *Realization of bounce gait in a quadruped robot with articular-joint-type legs* In: Proc. of the IEEE Int. Conf. on Robotics and Automation, pp. 697-702, 1995

[42] Furusho, J., Masubuchi, M. *Control of a dynamical biped locomotion system for steady walking*, Journal of Dynamical Systems, Measurements, and Control, **108**, pp. 111-118, 1986

[43] Geisler, T. and Manikas, T.W. *Autonomous robot navigation system using a novel value encoded genetic algorithm*, In: Proc. of the 45^{th} IEEE Midwest Symp. on Circuits and Systems, 2002

[44] Gomez, F. and Miikkulainen, R.. *Incremental evolution of complex general behavior*, Adaptive Behavior, **5**, pp. 317–342, 1997

[45] Gruau, F. *Automatic Definition of Modular Neural Networks*, Adaptive Behavior **3**:2, pp. 151-183, 1994

[46] Harvey, I., Husbands, P., and Cliff, D. *Issues in Evolutionary Robotics*, In: From Animals to Animats II: Proc. of the 2^{nd} Int. Conf. on Simulation of Adaptive Behavior (SAB92), pp. 364-373, 1992

[47] Harvey, I., Husbands, P., Cliff, D., Thompson, A., and Jakobi, N. *Evolutionary robotics: the Sussex approach*, Robotics and Autonomous Systems, **20**, No 2-4, pp. 205-224, 1997

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

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

[50] Hirai, K., Hirose, M., Haikawa, Y., and Takenaka, T. *The development of the Honda humanoid robot*, In: Proc. of the IEEE Int. Conf. on Robotics and Automation, **2**, pp. 1321-1326, 1998

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

[52] Hornby, G., Fujita, M., Takamura, S., Yamamoto, T., and Hanagata, O. *Autonomous evolution of gaits with the Sony quadruped robot*, In: Proc. of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann, vol. 2, pp. 1297-1304, 1999

[53] Hornby, G., Takamura, S., Yokono, J., Hanagata, O., Yamamoto, T., and Fujita, M. *Evolving robust gaits with aibo* In: Proc. of the IEEE Int. Conf. on Robotics and Automation, pp. 3040-3045, 2000

[54] Huang, Q., Nakamura, Y., Inamura, T. *Humanoids walk with feedforward dynamic pattern and feedback sensory reflection*, In: Proc. of the IEEE Int. Conf. on Robotics and Automation, pp. 4220-4225, 2001

[55] Inoue, H. and Hirukawa, H. *Explorations of Humanoid Robot Applications*, In: Proc. of the IEEE-RAS Int. Conf. on Humanoid Robots, pp. 497-499, 2001

[56] Jakobi, N. *Minimal Simulations for Evolutionary Robotics*, PhD thesis, School of Cognitive and Computing Sciences, University of Sussex, 1998

[57] Jakobi, N. *Running across the reality gap: Octopod locomotion evolved in a minimal simulation.* In: Proceedings of Evorobot, Springer Verlag, 1998

[58] Jakobi, N. *The Minimal Simulation Approach to Evolutionary Robotics*, In: Evolutionary Robotics - From Intelligent Robots to Artificial Life, proc. of ER98, AAI Books, 1998

[59] Jakobi, N., Husbands, P., and Harvey, I. *Noise and the reality gap: The use of simulation in evolutionary robotics*, Lecture Notes in Computer Science, **929**, pp. 704-720, 1995

[60] Katagami, D. and Yamada, S. *Teachers's Load and Timing of Teaching based on Interactive Evolutionary Robotics*, In: Proc. of the 2003 IEEE Int. Symp. on Computational Intelligence in Robotics and Automation, 2003

[61] Kato, I. and Tsuiki, H. *The hydraulically powered walking machine with a high carrying capacity* In: 4^{th} Int. Symp. on External Control of Human Extremities. Yugoslav Committee for Electronics and Automation, pp. 410-421, 1972

[62] Kennedy, J. and Eberhart, R. *Particle swarm optimization*, In: Proc. of the IEEE Int. Conf. on Neural Networks, Piscataway, NJ, pp. 1942-1948, 1995

[63] Keymeulen, D., Iwata, M., Kuniyoshi, Y., and Higuchi, T. *Comparison between an Off-line Model-free and an On-line Model-based Evolution applied to a Robotics Navigation System using Evolvable Hardware*, Proc. of the 6th Int. Conf. on Artificial Life, pp. 199-208, 1998

[64] Khatib, O. *Real-time obstacle avoidance for manipulators and mobile robots*, Int. J. of Robotics Research, **5**:1, pp. 90-98, 1986

[65] Kim, K.-J. and Cho, S.-B. *Robot action selection for higher behaviors with CAM-Brain modules*, In: Int. Symp. on Robotics, pp. 1623-1628, 2001

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

[67] Koza, J.R. *Evolution of Subsumption Using Genetic Programming*, In: Proc. of the First European Conf. on Artificial Life, MIT Press, pp. 110-119, 1992

[68] Koza, J.R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992

[69] Langton, C. (Ed.) *Artificial Life - An Overview*, MIT Press, 1997

[70] Lazarus, C. and Hu, H. *Evolving goalkeeper behaviours for simulated soccer competition*, In: Proceedings of the 3rd IASTED International Conference on Artificial Intelligence and Applications (AIA 2003), 2003

[71] Lee, W.-P., Hallam, J., Lund, H.H. *Applying Genetic Programming to Evolve Behavior Primitives and Arbitrators for Mobile Robots*, In: Proc. of the 4^{th} Int. Conf. on Evolutionary Computation, 1997

[72] Leger, C. *Darwin2K: An Evolutionary Approach to Automated Design for Robotics*, Kluwer Academic Publishers, 2000

[73] Lichtensteiger, L. and Eggenberger, P. *Evolving the Morphology of a Compound Eye on a Robot*, In: Proc. of the Third European Workshop on Advanced Mobile Robots (Eurobot99), pp. 127-134, 1999

[74] Lipson, H. and Pollack, J.B. *Automatic design and manufacture of robotic lifeforms*, Nature, **406**, pp. 974-978, 2000

[75] Luke, S., Hohn, C., Farris, J., Jackson, G., and Hendler, J. *Co-evolving Soccer Softbot Team Coordination with Genetic Programming*, In: RoboCup 97: Robot Soccer World Cup, Springer Verlag, pp. 398-411, 1997

[76] Lund, H.H., Hallam, J., and Lee, W. *Evolving robot morphology*, In: Proc. of the IEEE 4^{th} Int. Conf. on Evolutionary Computation, IEEE Press, pp. 197-202, 1997

[77] Lund, H.H., Miglino, O., Pagliarini, L., Billard, A., and Ijspeert, A. *Evolutionary Robotics - A Children's Game*, In: Proc. of the 5^{th} IEEE Int. Conf. on Evolutionary Computation, IEEE Press, pp. 154-158, 1998

[78] Maes, P. *How to do the right thing*, Connection Sci. J., **1**, No. 3, pp. 291-323, 1989

[79] Maes, P. *Modeling Adaptive Autonomous Agent*, Artificial Life, **1**:1-2, pp. 135-162, 1994

[80] Mataric, M. *Learning to Behave Socially*, In: Proc. of the 3^{rd} Int. Conf. on Simulation of Adaptive Behavior (SAB94), pp. 453-462, 1994

[81] Mataric, M. and Cliff, D. *Challenges in evolving controllers for physical robots*, Robotics and Autonomous Systems, **19**, No. 1, pp. 67-83, 1996

[82] Mautner, C. and Belew R. K. *Evolving Robot Morphology and Control*, In: Proc. of Artificial Life and Robotics 1999 (AROB99), 1999

[83] Meyer, J.-A. *Evolutionary Approaches to Neural Control in Mobile Robots*, In: Proc. of the IEEE Int. Conf. on Systems, Man, and Cybernetics, 1998

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

[85] Michel, O. *Webots*$^{\mathrm{TM}}$*: Professional Mobile Robot Simulation*, Int. J. of Advanced Robotic Systems, **1**:1, pp. 39-42, 2004

[86] Miglino, O., Lund, H.H., and Nolfi, S. *Evolving Mobile Robots in Simulated and Real Environments*, Artificial Life, **2**, pp. 417-434, 1996

[87] Miglino, O., Nafasi, C., and Taylor, C. *Selection for Wandering Behavior in a Small Robot*, Tech. Rep. UCLA-CRSP-94-01, Department of Cognitive Science, UCLA, 1994.

[88] Mitchell, M. *An Introduction to Genetic Algorithms*, MIT Press, 1996

[89] Miwa, H., Takanobu, H. and Takanishi, A. *Development of a human-like head robot we-3rv with various robot personalities*, In: Proc. of the IEEE-RAS International Conference on Humanoid Robots, pp. 117-124, 2001

[90] Moravec, H. *Robot: Mere Machine to Transcendent Mind*, Oxford University Press, 1999

[91] Moriarty, D. E. and Miikkulainen, R. *Evolving obstacle avoidance behavior in a robot arm*. In: From Animals to Animats: Proc. of the 4^{th} Int. Conf. on Simulation of Adaptive Behavior (SAB96), MIT Press, pp. 468-475, 1996

[92] Nelson, A.L., Grant, E., Galeotti, J.M., and Rhody, S. *Maze exploration behaviors using an integrated evolutionary robotics environment*, Robotics and Autonomous Systems **46**, pp. 159-173, 2004

[93] Nelson, A.L., Grant, E., Henderson, T.C. *Evolution of neural controllers for competitive game playing with teams of mobile robots*, Robotics and Autonomous Systems **46**, pp. 135-150, 2004

[94] Nolfi, S. *Evolving non-trivial behaviors on real robots: A garbage collecting robot.* Robotics and Autonomous Systems, **22**, pp. 187–198, 1997

[95] Nolfi, S. *Power and Limits of Reactive Agents*. Neurocomputing, **42**, pp. 119-145, 2002

[96] Nolfi, S. and Floreano, D. *Learning and evolution* Autonomous Robots, **7**:1, pp. 89-113, 1998

[97] Nolfi, S. and Floreano, D. *Evolutionary robotics*, MIT Press, 2000.

[98] Nolfi, S. and Floreano, D. *Synthesis of autonomous robots through evolution*, Trends in Cognitive Science, **6**:1, pp. 31-36, 2002

[99] Nolfi, S., Floreano, D., Miglino, O., and Mondada, F. *How to evolve autonomous robots: Different approaches in evolutionary robotics*, In: Artificial Life IV, pp. 190–197, 1994.

[100] Nolfi, S. and Marocco, D. *Evolving Robots Able to Visually Discriminate Between Objects with Different Size*, International Journal of Robotics and Automation (17) **4**, pp. 163-170, 2002

[101] Ostergaard, E.H. and Lund, H.H. *Co-Evolving Complex Robot Behavior*, In: Proc. of the 5^{th} Int. Conf. on Evolvable Systems (ICES2003), pp. 308-319, Springer Verlag, 2003

[102] Parker, G.B., Georgescu, R., and Northcutt, K. *Continuous Power Supply for a Robot Colony*, Proceedings of the World Automation Congress, 2004

[103] Paul, C. and Bongard, J.C. *The Road Less Travelled: Morphology in the Optimization of Biped Robot Locomotion*, In: Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS2001), IEEE Press, **1**, pp. 226-232, 2001

[104] Pettersson, J. *Generating Motor Behaviors for Bipedal Robots Using Biologically Inspired Computation Methods*, Licentiate Thesis, Chalmers University of Technology, 2003

[105] Pettersson, J., Hartono, P., and Wahde, M. *A behavior module for odometry calibration in autonomous robots*, Submitted to AMiRE2007.

[106] Pettersson, J., Sandholt, H., and Wahde, M. *A flexible evolutionary method for the generation and implementation of behaviors in humanoid robots* In: Proc. of the IEEE/RAS Int. Conf. on Humanoid Robots, pp. 279-286, 2001

[107] Pettersson, J. and Wahde, M. *Application of the utility manifold method for behavioral organization in a locomotion task*, submitted to IEEE Trans. on Evolutionary Computation, 2004

[108] Pirjanian, P. *Behavior coordination mechanisms – state-of-the-art*, Technical Report IRIS-99-375, Institute of Robotics and Intelligent Systems, University of Southern California, Los Angeles, 1999

[109] Ptashne, M. *A Genetic Switch: Phage λ and Higher Organisms*, 2^{nd} Ed., Cell Press and Blackwell Science, 1992

[110] Quackenbush, J. *Computational Analysis of Microarray Data*, Nature Genetics Reviews, **2**, pp. 418-427, 2001

[111] Quinn, M., Smith, L., Mayley, G., and Husbands, P. *Evolving Team Behaviour for Real robots*, In: EPSRC/BBSRC Int. Workshop on Biologically Inspired Robots: The Legacy of W. Grey Walter, pp. 217-224, 2002

[112] Quinn, M., Smith, L., Mayley, G., and Husbands, P. *Evolving teamwork and role allocation for real robots*, In: Proc. of the 8^{th} Int. Conf. on Artificial Life, pp. 302-311, 2002

[113] Raibert, M. *Legged robots that balance*, MIT Press, 1986

[114] Reynolds, C.W. *Evolution of Corridor Following Behavior in a Noisy World.*, In: Proc. of the 3^{rd} Int. Conf. on Simulation of Adaptive Behavior, pp. 402–410, 1994

[115] Rosenblatt, J.K. *DAMN: A distributed architecture for mobile navigation*, J. of Exp. and Theor. Artificial Intelligence, **9**:2-3, pp. 339-360, 1997

[116] Russell, S.J. and Norvig, P. *Artificial Intelligence: A Modern Approach*, Prentice-Hall, 2^{nd} Ed., 2002

[117] 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 the 2004 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS2004), 2004

[118] Shan, J., Junshi, C., and Jiapin, C. *Design of central pattern generator for humanoid robot walking based on multi-objective GA*, In: Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, pp. 1930-1935, 2000

[119] Sims, K. *Evolving 3D Morphology and Behavior by Competition*, Artificial Life **1**, pp. 353-372, 1994

[120] Sprinkhuizen-Kuyper, I.G. *Artificial Evolution of Box-pushing Behavior*, Technical Reports in Computer Science, CS 01-02, 2001

[121] Sprinkhuizen-Kuyper, I.G., Kortmann, R., and Postma, E.O. *Fitness functions for evolving box-pushing behaviour* In: Proceedings of the Twelfth Belgium-Netherlands Artificial Intelligence Conference, pp. 275-282, 2000

[122] Stanley, K.O. and Miikkulainen, R. *Evolving Adaptive Neural Networks with and without Adaptive Synapses*, Proc. of the Genetic and Evolutionary Computation Conference, late breaking papers (GECCO 2003), 2003

[123] Sugihara, K. and Smith, J. *A Genetic Algorithm for 3-D Path Planning of a Mobile Robot*, Tech. Rep. No. 96-09-01, Software Engineering Research Laboratory, Department of Information and Computer Sciences, University of Hawaii at Manoa, 1996

[124] Taylor, T. and Massey, C. *Recent Developments in the Evolution of Morphologies and Controllers for Physically Simulated Creatures*, Artificial Life **7**:1, pp. 77-87, 2001

[125] Theraulaz, G. and Bonabeau, E. *A Brief History of Stigmergy*, Artificial Life **5**, pp. 97-116, 1999

[126] Thompson, A. *Evolving Electronic Robot Controllers that Exploit Hardware Resources*, In: Proc. of the 3^{rd} Int. Conf. on Artificial Life, Springer Verlag, pp. 640-656, 1995

[127] Trianni, V., Tuci, E., Dorigo, M. *Evolving Functional Self-Assembling in a Swarm of Autonomous Robots*, In: Proc of the 8^{th} Int. Conf. on the Simulation of Adaptive Behavior (SAB04), pp. 405-414, 2004

[128] Tuci, E., Harvey, I., and Quinn, M. *Evolving integrated controllers for autonomous learning robots using dynamic neural networks*, Proc. of the 7^{th} Int. Conf. on the Simulation of Adaptive Behavior (SAB02), 2002

[129] Urzelai, J. and Floreano, D. *Evolutionary Robotics: Coping with Environmental Change*, In: Proc. of the Genetic and Evolutionary Computation Conference (GECCO 2000), pp. 941-948, 2000

[130] Vukobratovic, M. and Juricic, D. *Contribution to the synthesis of bipedal gait*, IEEE Trans. Bio-Med. Eng. **16**, No. 1, pp. 1-6, 1969

[131] Wahde, M. *A method for behavioural organization for autonomous robots based on evolutionary optimization of utility functions*, J. Systems and Control Engineering, **217**, pp. 249-258, 2003

[132] Wahde, M. and Nordahl, M. *Co-evolving Pursuit-Evasion Strategies in Open and Confined Regions*, In: Proc. of the 6^{th} Int. Conf. on Artificial Life, MIT Press, pp. 472-476, 1998

[133] Wahde, M. and Nordahl, M. *Evolution of protean behavior in pursuit-evasion contests*, In: Proc. of the 5^{th} Int. Conf. on the Simulation of Adaptive Behavior (SAB98), MIT Press, pp. 557-561, 1998

[134] Wahde, M. and Pettersson, P. *UFLibrary v1.0.1 Tutorial*, available at www.me.chalmers.se/~mwahde/robotics/UFMethod/UFLibrary/index.html

[135] Wahde, M. and Pettersson, J. *A brief review of bipedal robotics research*, In: Proc. of the 8^{th} UK Mechatronics Forum International Conference (Mechatronics 2002), pp. 480-488, 2002

[136] Wahde, M., Pettersson, J., Sandholt, H., and Wolff, K. *Behavioral Selection Using the Utility Function Method: A Case Study Involving a Simple Guard Robot*, to appear in Proc. of the 3^{rd} International Symposium in Autonomous Minirobots for Research and Edutainment (AMiRE2005), 2005

[137] Wahde, M. and Sandholt, H. *Evolving Complex Behaviors on Autonomous Robots*, In: Proc. of the 7^{th} UK Mechatronics Forum International Conference (Mechatronics 2000), Pergamon Press, 2000

[138] Walker, M. *Evolution of a Robotic Soccer Player* Res. Lett. Inf. Math. Sci, **3**, pp. 15-23, 2002

[139] Walker, M. and Orin, D. *Efficient dynamic computer simulation of robotic mechanisms*, Journal of Dynamic Systems, Measurements, and Control **104**, pp. 205-211, 1982

[140] Watson, R. A., Ficici, S. G., and Pollack, J. B. *Embodied Evolution: Embodying an Evolutionary Algorithm in a Population of Robots*, In: 1999 Congress on Evolutionary Computation, IEEE Press, pp. 335-342, 1999

[141] Wolff, K. and Nordin, P. *Evolution of efficient gait with humanoids using visual feedback*, In: Proc. of the IEEE-RAS Int. Conf. on Humanoid Robots, pp. 99-106, 2001

[142] Wolff, K., Pettersson, J., Heralic, A., and Wahde, M. *Structural Evolution of Central Pattern Generators for Bipedal Walking in 3D Simulation*, In: Proc. of the 2006 IEEE Conference on Systems, Man, and Cybernetics (SMC 2006), Taipei, Taiwan, pp. 227-234, 2006

[143] Yamada, S. *Evolutionary Design of Behaviors for Action-Based Environment Modeling by a Mobile Robot*, Proc. of the Genetic and Evolutionary Computation Conference (GECCO-2000), pp. 957-964, 2000

[144] Yamada, S., Saito, J. *Adaptive Action Selection Without Explicit Communication for Mutirobot Box-Pushing*, IEEE Trans. on Systems, Man, and Cybernetics, Part C, **31**, No.3, pp.398-404, 2001

[145] Yamauchi, B.M. and Beer, R.D. *Sequential Behavior and Learning in Evolved Dynamical Neural Networks*, Adaptive Behavior **2**:3, pp. 219-246, 1994

[146] Yao, X. *Evolving Artificial Neural Networks*, Proc. of the IEEE **87**:9, pp. 1423-1447, 1999

[147] Zykov, V., Bongard, J., and Lipson, H., *Evolving Dynamic Gaits on a Physical Robot*, Proc. of the Genetic and Evolutionary Computation Conference, Late Breaking Paper, GECCO04, 2004