

6 Artificial Evolution

There are only two ways we know of to make extremely complicated things. One is by engineering, and the other is evolution. And of the two, evolution will make the more complex.

Daniel Hillis

Artificial evolution is a classic topic of artificial life. Researchers in artificial evolution typically pursue one of the following goals: They are interested in understanding the principles of biological evolution; in the spirit of the synthetic methodology, the understanding of biological phenomena can be greatly enhanced by trying to mimic or model aspects of the biological system under examination. The other main goal is to use methods from artificial evolution as optimization tools, or more generally as design methods. In many areas, evolutionary algorithms have proved highly useful, especially for “hard” problems. In quite a number of areas they have produced solutions that humans could not easily have derived. Humans have their inescapable biases because they have grown up in a particular environments and societies. Evolutionary methods can be seen as ways to overcome these biases because computers only have those biases that have been explicitly programmed into them. A beautiful illustration of this point is the “tubing problem” that was described by Rechenberg in the 1960s. It is illustrated in Figure 6.1.

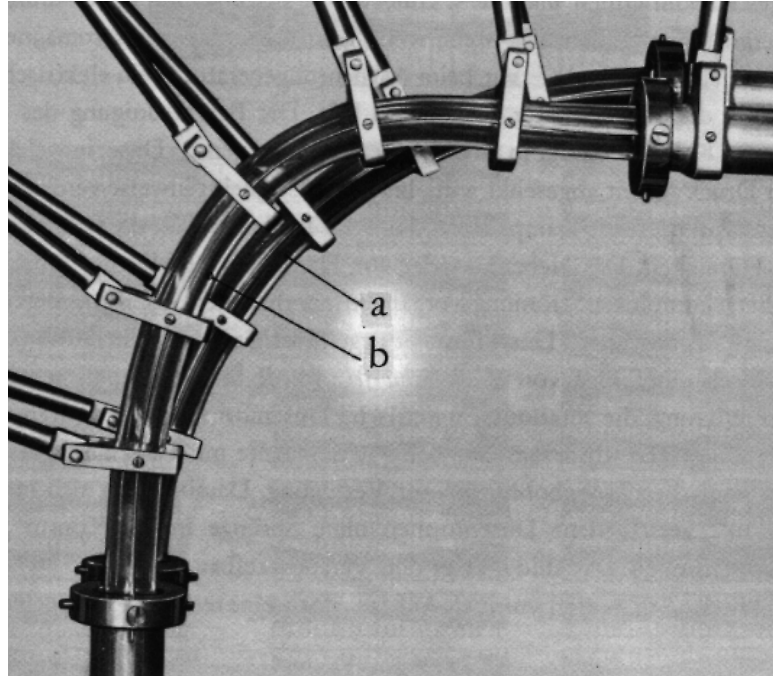


Figure 6.1: Rechenberg’s “tubing problem”. (a) the standard solution, and (b) the optimal solution.

The fuel flows into the tube from the left. The fuel should leave the tube (top right) such that the resistance of the fluid is minimized. The question is how the two tubes should be connected, i.e. what is the shape of

the connecting tube? The standard answer of most people is a quarter of a circle. Using an evolutionary method called “evolution strategy” or ES – a technical term for a particular class of algorithms (see below) – Rechenberg (1994) could show that the optimal solution for this problem has a kind of “hunch” on the top, a solution that one would not easily think of. Apparently this “hunch” helps minimizing turbulence.

6.1 Introduction: Basic Principles

In chapters 1 through 5 we have seen many examples of emergence of behavior and other patterns. The essential aspects have always been (i) there are many components in the system (cells, actors, grid-points), and (ii) there is interaction via local rules. In this chapter on artificial evolution we also work with systems containing many elements – these elements are called populations. Evolution only makes sense in the context of entire populations; it is not defined for individuals.

The field of artificial evolution started in the 1960s with developments of Ingo Rechenberg in Germany, and of John Holland and L.J. Fogel in the United States. Holland’s breed of evolutionary algorithms is called *genetic algorithms* or GAs (DeJong, 1975; Goldberg, 1989; Holland, 1975; Mitchell, 1997), Fogel’s *evolutionary programming* or EP (Fogel, 1962; Fogel, 1995), and Rechenberg’s *evolution strategies* or ESs (Rechenberg, 1973; Schwefel, 1975). Holland was interested in adaptation in natural systems, Fogel and Rechenberg more in exploiting evolutionary algorithms for optimization. They all share a strong belief in the power of evolution. While from an algorithmic perspective there are important differences between these three types of procedures, for our purposes these differences are not essential. For a comparison, the interested reader is referred to (Bäck and Schwefel, 1993).

There is a vast literature on evolutionary algorithms of various sorts. But for the better part, the principles upon which they are based are relatively similar and can be easily categorized. We start with an example, which helps introducing the basic concepts and then provide an overall scheme.

Example: Biomorphs

This description is based on “The Blind Watchmaker” by Richard Dawkins (Dawkins, 1986). Biomorphs are tree like structures used as a graphical representation of a number of simple genes. They are a fun way to demonstrate the power of evolution: random mutation followed by non-random selection. Remember the turtle graphics that we encountered in the context of Lindenmeyer systems. We will now use the same idea to illustrate artificial evolution.

Artificial evolution always starts from a genome, thus the problem to be solved must somehow be encoded in the genome. The “problem” in our case is simply to draw interesting and appealing figures. In the turtle graphics we could specify, for example, the length of the lines to be drawn, the angle the two lines make at a joint, and the depth of recursion. That would make a genome with 3 genes, the first and the second are real numbers, whereas the third (the depth of recursion) is an integer. In fact, Dawkins used 9 genes, 4 for the length of the lines and 4 for the angles and also a recursion depth. In this way, more complicated shapes can be produced because not all lines have to be of the same length. Because the idea is to mimic (in some very abstract sense) cell division, and because one cell always divides into two cells, the number of lines coming off a joint is always two. The drawing program can be seen as the process of ontogenetic

development that transforms the genotype (the genetic material, i.e. the numbers contained in the genome) into the phenotype, i.e. the figure on paper. The drawings themselves are called Biomorphs by Dawkins, a term which was originally coined by zoologist and painter Desmond Morris for the vaguely animal-like shapes in his surrealist paintings. (Desmond Morris claims that his Biomorphs “evolve” in his mind and that their evolution can be traced through successive paintings.)

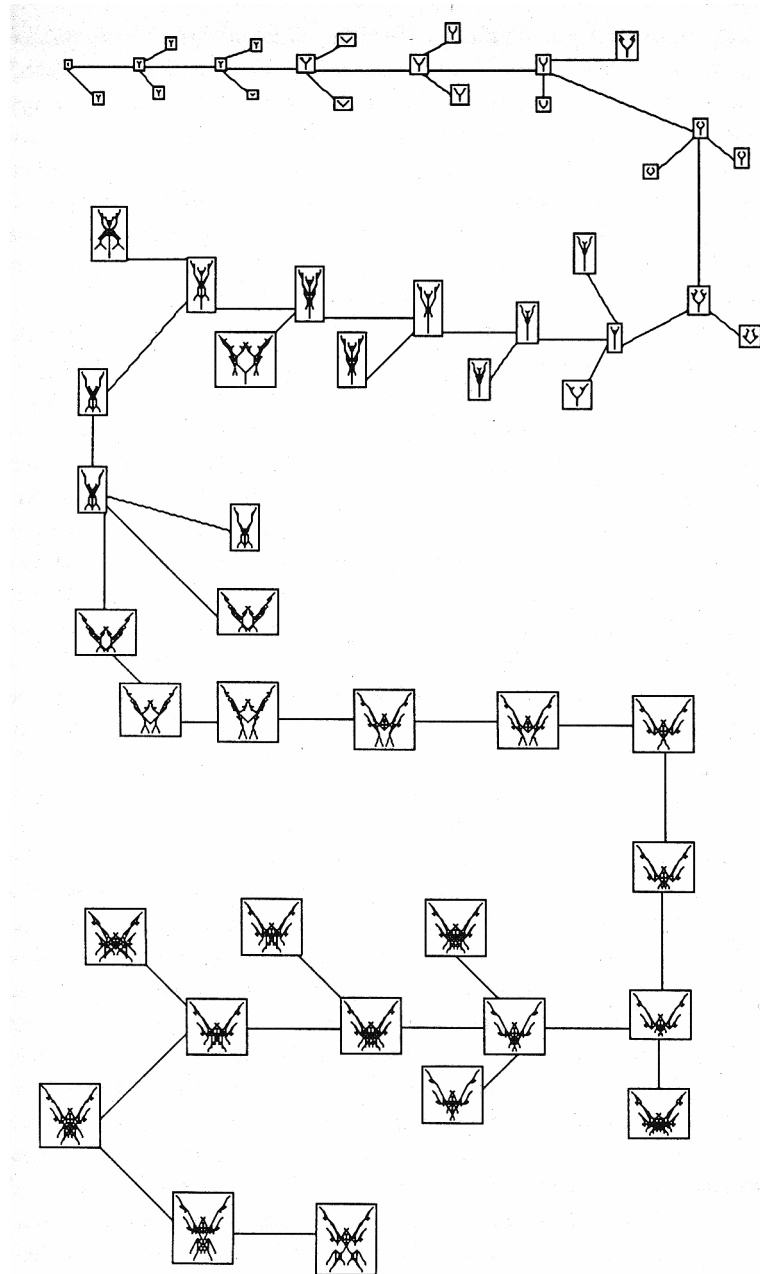


Figure 6.2: Starting from one Biomorph (represented by a dot) Dawkins evolves an insect-like individual after only 29 generations.

Initially, a set of numbers that constitute the genes are randomly generated (within certain limits). From these genotypes, the phenotypes are produced by applying the drawing program (through a process called *development*). The user selects from this set of Biomorphs the phenotype that he or she likes most (this is

called *selection*). This Biomorph is then taken as the starting point for further evolution, and it is allowed to reproduce. In other words, its genotype is mutated and another set of Biomorphs is drawn, etc., until there is a Biomorph that the user likes. This type of evolution where the user of the program decides which Biomorph is allowed to reproduce, is also called “animal breeder selection” (or “aesthetic selection”), since it resembles precisely what animal breeders do. Note the historical nature of this process: it contains, and in some sense accumulates, the user’s decisions. This is why the term “cumulative selection” is sometimes used. The genome is changed through mutation, i.e. random changes in the numbers that represent the various features in the genome (like angles and length of segments). A simulation starting from scratch, where the user can play “God” is shown here: <http://suhep.phy.syr.edu/courses/mirror/biomorph/>

In addition to Dawkins’s 9 genes, this demonstration includes 6 extra genes, yielding a total of 15 genes. Very briefly, the first 8 control the overall shape of the Biomorph: four are for encoding the angles on four subsequent steps and four are for the length of the segment to be drawn (similar to Dawkins); gene 9 encodes the depth of recursion (like Dawkins); genes 10, 11 and 12 encode the red, green and blue color components of the color in which the Biomorph is drawn. Gene 13 determines the number of segments in the Biomorph; gene 14 controls the size of the separation of the segments; and gene 15 the primitive used to draw the Biomorph (line, oval, rectangle, filled oval or filled rectangle). The simulation takes off with dots, but after a few generations fascinating forms start to appear.

In the following simulation Biomorphs are generated at random by clicking the “zap” button. Through “select” the current Biomorph is selected and small random mutations of this Biomorph are displayed in the other windows. The principle is the same as in the first example. This is the URL:

<http://www.math.ruu.nl/people/beukers/dawkins/dawkins.html>

The Basic Evolutionary Cycle

(adapted from Pfeifer and Scheier, 1999, Chapter 8, p. 232, The Evolutionary Process)

Figure 6.3a gives an overview of the evolutionary process. Evolution always works with populations of individuals. In nature these are creatures, in artificial evolution they are often solutions to problems. Each individual agent carries a description of some of its features (color of its hair, eyes, skin, body size, limb size, shape of nose, head, etc.). This description is called its genome. The term genotype refers to the set of genes contained in the genome. It is used to express the difference between the genetic setup and the final organism, the phenotype. The genome consists of a number of genes, where in the simplest case one gene describes one feature. Genes are identified by their position within the genome. If the individual members of the population to be investigated are of the same species, they all have the same numbers of genes and the genes are at the same location in the genome. But the values of the genes can differ. The values of all genes of an individual are determined before it starts to live and never changes during its life.

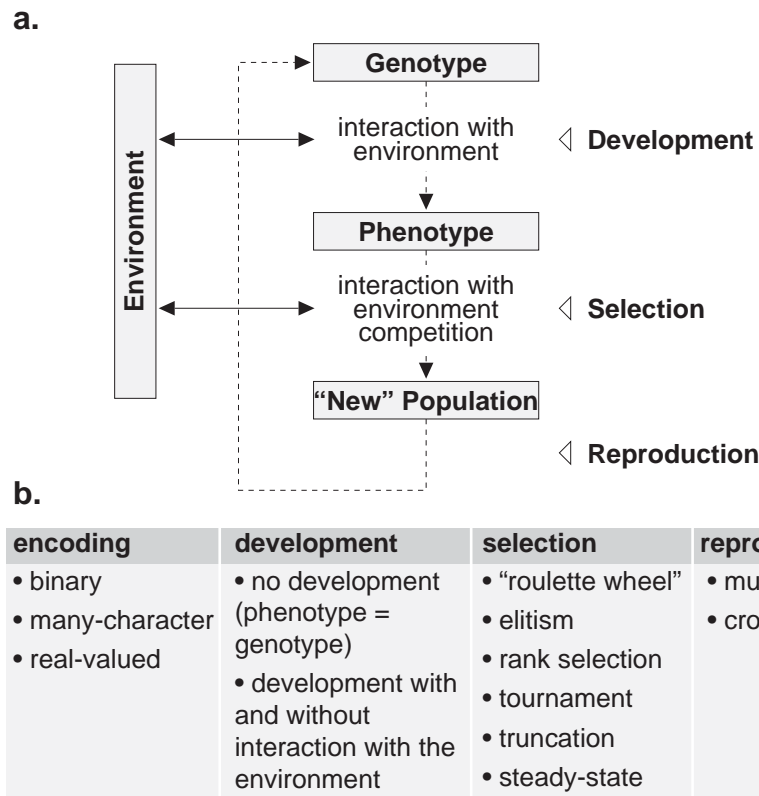


Figure 6.3: Overview of the process of evolution. (a) The main components. The genotype is translated into a phenotype through a process of development. The phenotypes compete with one another in their ecological niche, and the winners are selected (selection) to reproduce (reproduction), leading to new genotypes. (b) Evolutionary algorithms can be classified according to a number of dimensions: encoding scheme, nature of developmental process, selection method, and reproduction (genetic operators). Mitchell (1997, pp. 166-175) discusses the pros and cons of these various methods in detail.

Through a process of *development*, the genotype is translated into a phenotype. In this process, genes are expressed, i.e. they exert their influence on the phenotype, in various ways. The precise way in which the genes are expressed is determined by the growing organism’s interaction with the environment. In artificial evolution, the phenotype and the genotype are indeed often the same. If we are aware of the fact that we are dealing with algorithms, rather than with a model of natural evolution, there is absolutely nothing wrong with this. Then, the phenotype competes in its ecological niche for resources with other individuals of the same or other species. The competition in the Biomorph example consists in somehow trying to please the experimenter’s sense of aesthetics. The winners of this competition are selected by the experimenter, which leads to a new population. The members of this new population have higher average fitness than in the previous one. The individuals in this new population can now reproduce. It is characteristic of evolutionary approaches that they work with populations of individuals rather than individuals only. There are many variations of how selection can be done; a list is provided in Figure 6.3b.

So far we have discussed only asexual reproduction where an individual duplicates only its own genotype, possibly with some small random mutation. In the Biomorph example, there was only one individual involved in reproduction, i.e. the reproduction was asexual. There is also sexual reproduction where two individuals exchange parts of their genotype to produce new genotypes for their offspring. The most

common type of sexual reproduction is called crossover (see below). It is often used in combination with mutation. Finally, there is a reproduction process. Reproduction, like selection, comes in many variations.

Development, selection, and reproduction are closed in themselves: they receive a certain input and deliver some output. For example, the *development* process receives a genotype as input and eventually produces a phenotype as output. But the phenotype cannot influence the genotype. Since we are dealing with algorithms, it would be no problem to have the phenotype influence the genotype. But that has not been systematically explored, presumably because that would correspond to a so-called Lamarckian position. According to Lamarck, learned properties of an organism can be genetically passed on to the offspring.

The scheme of Figure 6.3 can be used to classify different evolutionary approaches: some comprise all these components in non-trivial ways, some do not have development – in fact, most evolutionary algorithms do not – and some have development, but without interaction with the environment. Additional classification can be made according to the way the features are encoded in the genome, the type of selection, and the kind of reproduction performed. We will only provide a short review and give examples of the most common kinds. There are many excellent textbooks that give systematic reviews of the various types of algorithms (e.g. Goldberg, 1989; Mitchell, 1997). Before we look at the different approaches in more detail let us consider some theoretical points.

Variations on Evolutionary Methods: Theoretical Issues

(from Pfeifer and Scheier, 1999, Chapter 8, page 236 ff)

As we have already mentioned, there are many variations on evolutionary methods. Like neural networks, evolutionary algorithms are fascinating and seem to exert an inescapable attraction urging the user to play around and tinker with them. Here we present a few variations; for systematic reviews, see, for example, Mitchell 1997.

Encoding Scheme: The most widely used encoding scheme is what we have seen in our earlier example, binary encoding in terms of bit strings. A few others appear elsewhere in this script, such as many-character encoding, as in Eggenberger's Artificial Evolutionary System (Eggenberger, 1997), or the graph structures used by Karl Sims (Sims, 1994a; see Section 6.3). The rule here is to use whatever is best suited. The choice of encoding scheme is not a matter of religious devotion to one scheme over all others.

Development: As we mentioned, development is often entirely neglected – it may not be necessary at all. In its absence, selection is performed directly on the genotype. There are also trivial forms of development in which the representation in the genome is directly mapped onto the organism's features without interaction with the environment. We have seen this in the example above. More complex, but still lacking interaction with the environment, is Sims' approach. One model that capitalizes on ontogenetic development is Eggenberger's Artificial Evolutionary System.

Selection: One gets the impression that researchers in the field have tried virtually any method of selection that even remotely promised to improve their algorithms' performance. All methods have their pros and cons – discussing them in any depth is well beyond the scope of this chapter. One might be tempted simply to take the best individuals and ignore the others. However, that would lead to a quick loss of diversity.

Those individuals currently not doing as well as the others may have properties that will prove superior in the long run. Thus, a great deal of attention has been devoted to getting just the right mix of individuals. The problem is sometimes termed the *exploration-exploitation trade-off*. The goal is to search the space in the region of the good individuals but still to explore other regions, because the currently best may turn out to be only locally optimal.

Holland (1975) proposed using a method in which an individual's probability of being selected is proportional to its fitness. This is also called *roulette wheel selection*: spin the roulette wheel and select the individual where it stops. The size of the segment of the roulette wheel for an individual is proportional to its fitness. *Elitism* is often added to various schemes, meaning that the best individuals are copied automatically to the next generation. In *rank selection*, individuals are chosen with a probability corresponding to their rank (in terms of fitness), rather than their actual fitness value. *Tournament selection* is based on a series of comparisons of two individuals: through some random procedure that takes their fitness into account, one individual “wins” and is selected for reproduction. Finally, there is a distinction between *generational* and *steady state selection*. As mentioned above, rather than producing an entirely new population at the same time, in steady-state selection, only a small part of the population changes at any particular time, while the rest is preserved.

Reproduction: The most-often-used genetic operators are mutation and crossover. We have seen both in the example above. Although evolutionary methods are easy to program and play around with, their behavior is difficult to understand. It is still subject to debate how they work and what the best strategies for reproduction and selection are. Let us turn to natural evolution for a moment. Once good partial solutions have been found for certain problems, they are kept around and are combined with other good solutions. Examples are eyes and visual systems: Once they had been “invented”, they were kept around and perhaps slightly improved. In evolutionary algorithms, there is a similar idea of good “building blocks” that are combined to increasingly better solutions.

Crossover is designed to combine partial solutions into complete ones with high fitness. There are a number of conjectures about why crossover leads to fast convergence while maintaining a high chance of reaching the global optimum. One is the schema theorem and, related to it, the building block hypothesis (e.g., Goldberg 1989). Schemas are particular patterns of genes that, depending on the algorithm chosen, proliferate in the population. The details need not concern us here; there is an ongoing debate as to the relevance of this theorem to evolutionary methods. The related topic of how useful crossover really is and how it contributes to resolving this trade-off is also still subject to debate (e.g., Srinivas and Patnaik 1994).

The preceding discussion can best be summarized as follows: There is no one best encoding scheme, selection strategy, or genetic operator. It is all a question of having the right balance suited for the particular issues one intends to investigate.

6.2 Different Approaches (GAs, ES, GP)

The most frequent approaches in the field of artificial evolution are the Genetic Algorithms (GAs), the Evolution Strategy (ES), and Genetic Programming (GP) (also called Evolutionary Programming, EP) as

briefly mentioned above. We start our discussion with the most popular version, the GAs. John Holland in the United States who was interested mostly in models of biological evolution invented them.

Genetic Algorithms

Genetic algorithms follow a particular scheme, which is common to all. It is illustrated in Figure 6.4.

Basic algorithm:

- Initialize population P
- Repeat for some length of time
 - Create an empty population, P'
 - Repeat until P' is full:
 - Select two individuals from P based on some fitness criterion
 - Optionally “mate”, and replace with offspring
 - Optionally mutate the individuals
 - Add the two individuals to P'
 - Let P now be equal to P'

Figure 6.4: Basic scheme of genetic algorithm (from Flake, 1998).

Typically, the initial population P is generated randomly. Two individuals are selected on the basis of their fitness using a selection strategy, e.g., roulette wheel selection. Then a genetic operator, e.g., crossover, is applied, creating two offspring, which replace the parents. Almost always, there is mutation. *Mutation* is performed with a certain probability. Figure 6.5 below illustrates selection and reproduction. If this procedure is applied, the subsequent population has a higher average fitness than the previous one, which is, of course, the purpose of the optimization algorithm. Crossover, mutation, and various selection strategies will be discussed below. Note that normally in GAs there is no process of development: the genotype is identical to the phenotype and the selection is performed directly on the genotype. This means that fitness evaluations have to take place on the basis of the genotype, which is biologically speaking, unrealistic.

a. selection

1. take the individual with the highest fitness
2. choose another individual from the population at random, irrespective of fitness, for sexual reproduction
3. add the fittest individual to the new population

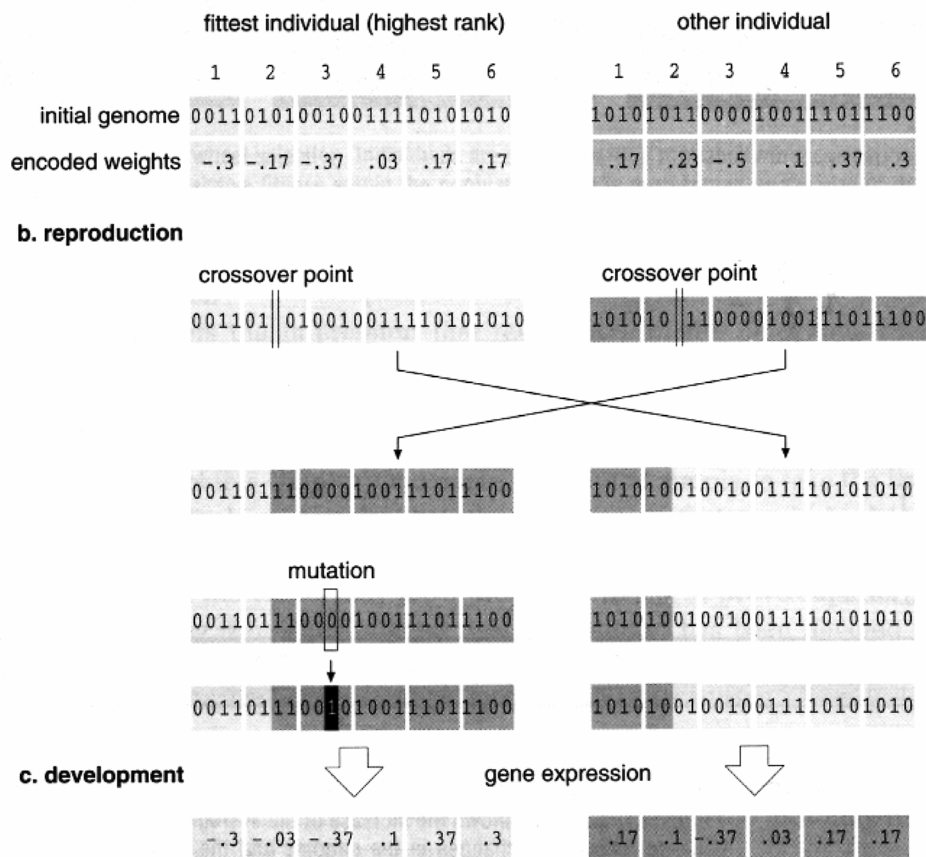


Figure 6.5: Selection and reproduction. a) Selection: After their final fitness values have been determined, individuals are selected for reproduction. b) Reproduction: The crossover point is chosen at random. The entire population is subject to a small mutation: c) Development: After reproduction, the new genome is expressed to become the new individual.

Example: The iterated prisoner's dilemma

The following description of the iterated prisoner's dilemma is based on (Mitchell, 1997).

The prisoner's dilemma has been studied extensively in game theory, economics, and political science, because it can be seen as an idealized model for real-world phenomena such as arms races. It can be formulated as follows: two individuals, Alice and Bob, are arrested for having committed a crime together and are being held in two separate cells, with no possibility of communication. Alice is offered the following deal: If she confesses and agrees to testify against Bob, she will receive a suspended sentence with probation (German: bedingt mit Bewährung), and Bob will be put in jail for 5 years. However, if at the same time, Bob confesses and agrees to testify against Alice, her testimony will be discredited, and each will receive 4 years of prison for pleading guilty. Alice is told that Bob is being offered precisely the same

deal. Both Alice and Bob know that if neither testifies against the other, they can be convicted only on a lesser charge for which they will have to spend only 2 years in jail.

Now, what should Alice do? Should she “defect” against Bob (i.e. should she confess and testify against Bob) and hope for the suspended sentence, risking a 4-year sentence if Bob also defects? Or should she “cooperate” with Bob (i.e. should she deny and not testify against Bob), hoping that Bob will also cooperate, in which case both will get only 2 years in prison. However, if Bob does not cooperate but defects, Alice will get 5 years and Bob nothing (i.e. only a suspended sentence with probation).

The game can be described more abstractly. Each player independently decides which move to make, i.e. whether to cooperate or defect. A “game” consists of each player making a decision (a “move”). The possible results of a single game are summarized in a payoff matrix, as shown in Figure 6.6.

	Player B		
		Cooperate	Defect
	player A		
	cooperate	3, 3	0, 5
	defect	5, 0	1, 1

Figure 6.6: Payoff matrix for the prisoner’s dilemma game (points representing reduction in years to be spent in jail).

Here, the goal is to get as many points as possible (few years in prison would correspond to too many points). For example, in the figure, the payoff in each case can be interpreted as 5 minus the number of years in prison. If both players cooperate, each gets 3 points (corresponding to the 2 years in prison). If player A defects and player B cooperates, then player A gets 5 points (suspended sentence with probation) and player B gets 0 points (5 years in prison), and vice versa if the situation is reversed. If both players defect, each gets 1 point. What is the best strategy to use in order to maximize one’s own payoff? If you suspect that your opponent is going to cooperate, then you should surely defect. If you suspect that your opponent is going to defect, then you should defect too. No matter what the other player does, it is always better to defect. The dilemma is that if both players defect each gets a worse score than if they cooperate. If the game is iterated, i.e. if the two players play several games in a row, both players’ always defecting will lead to a much lower total payoff than the players would get if they cooperated. How can reciprocal cooperation be induced? This question takes on special significance when the notions of cooperating and defecting correspond to actions in, say, a real-world arms race (reducing or increasing one’s arsenal).

Note that if the two players only play on single game, they have no information about the other’s strategy. However, if they play several games and they have the information of the opponent’s strategy in prior games, they have information on which they can base their decisions.

Encoding in genome

Whenever we want to solve a problem using a genetic algorithm we first have to define the encoding in the genotype. The encoding used here is based on (Axelrod, 1987). Suppose that the memory of each player contains one previous game. There are four possibilities for the previous game:

- CC (case 1),
- CD (case 2),
- DC (case 3),
- DD (case 4)

where C denotes “cooperate” and D denotes “defect.” Case 1 is where both players cooperated in the previous game, case 2 when player A cooperated and player B defected, and so on. A strategy is simply a rule that specifies an action in each of these cases. For example, TIT FOR TAT as played by player A is as follows:

- If CC (case 1), then C.
- If CD (case 2), then D.
- If DC (case 3), then C.
- If DD (case 4), then D.

If the cases are ordered in this way, the strategy can be expressed compactly by simply listing the right-hand sides, i.e. CDCD: Position 1 represents the answer to case 1 (i.e. C), position 2 the answer to case 2 (i.e. D), etc. It is a general principle in genetic algorithms to interpret the genes by position in the genome.

In Axelrod’s tournaments the strategies were based on three remembered previous games. For example, CC DC DD means that player A played C in the first game, player B played C in the first game. In the second game, player A played D (i.e. he defected), while player B cooperated, and in the third game, both defected. In total there are 64 possibilities:

- CC CC CC (case 1),
- CC CC CD (case 2),
- CC CC DC (case 3),
- CC CC DD (case 4),
- ...
- DD DD DC (case 63),
- DD DD DD (case 64).

A strategy in this case tells the player how to play, given the three games that the player remembers. In other words, for each case the player has to specify what will be the next move, i.e. C or D. Thus, a strategy can be encoded compactly as a string of 64 C’s or D’s, one for each case. Since for each case there are two possibilities, there are 2^{64} different strategies, i.e. 2^{64} strings of C’s and D’s (just think of 0 and 1, then this is like a string of binary digits). Actually, Axelrod used a 70-letter string: he also encoded what the players had played in the last three games (yielding 2^{70} different strategies, which is roughly 10^{21} - a very large number indeed!). Which one is the best one? Exhaustive search is clearly no longer possible. So, we can try a genetic algorithm.

For every GA an initial population has to be specified. Axelrod used a population of 20 such randomly generated strategies. The fitness of each strategy in the population was determined as follows: Axelrod organized Prisoner’s Dilemma tournaments. He solicited strategies from researchers in a number of

disciplines. Each participant had to submit a computer program that implemented a particular strategy. These programs played the iterated Prisoner's dilemma with three remembered games. The tournament also contained a strategy that simply made random moves. Some of the strategies submitted were rather complicated and used sophisticated modeling techniques in order to determine the best move. However, it turned out that the winner (the strategy with the highest average score) was the simplest of the submitted strategies: TIT FOR TAT, a strategy that we have encountered before in the context of agent-based simulation. This strategy cooperates in the first game and then, in subsequent games, does whatever the other player did in its move in the previous game. In other words, it offers cooperation and reciprocates it. But if the other player defects, he will be punished until he begins to cooperate again.

Axelrod found that 8 of these strategies were sufficient to predict the performance of a strategy on all 63 entries. This set of 8 strategies served as the "environment" for the evolving strategies in the population. Each individual in the population played iterated games with each of the 8 fixed strategies, and the individual's fitness was taken to be its average score (of these eight games).

The results were as follows: Often the TIT FOR TAT strategy was evolved. The problem with this result is that it is based on the assumption of a stable environment. But individuals would not maintain the same strategy if they see what strategies the others use. In other words, the environment, which consists of the strategies of other individuals, will necessarily change. If we make this much more realistic assumption, the conclusions are less straightforward.

Evolution Strategy

The concept of Evolution Strategy (ES) was developed back in the '60s by Bienert, Rechenberg and Schwefel. It merely concentrates on the usage of the fundamental mechanisms of biological evolution for technical optimization problems. In the beginning ES was not a specific algorithm to be used with computers but a method to find optimal parameter settings in laboratory experiments. A simple algorithmic method based on random changes of experimental setups was used to decide how the parameter settings should be changed. Adjustment took place in discrete steps only and was not population based. Thus, the first and simplest evolution process is a $(1+1)$ -ES, where one parent and one offspring are evaluated and compared. Subsequently the one that is fitter becomes the parent of the next generation. By contrast with genetic algorithms, ES imitates the effects of genetic procedures on the phenotype, i.e. on the object to be optimized (e.g. the tube shown in Fig. 6.1).

Fitness always depends on the solution an individual offers to a certain problem in the respective problem-specific environment. In order to measure fitness, characteristic data of an individual have to be collected and evaluated. The respective parameters are then optimized in an evolution-based process. In the 1960s these parameters were represented by integer variables. Nowadays, with computer implementations, the parameters are arranged in vectors of real numbers on which operators for recombination (cross-over) and mutation are defined. The data-structure of a single individual is realized in the form of two vectors of real numbers. One is called object-parameter, the other one strategy-parameter. While object-parameters contain the variables to be optimized, strategy parameters control mutation of the object-parameters. The fact that

the optimization takes place on both, the object-parameters as well as the strategy-parameters is seen as one of the major qualities of ES (often called self-adaptation).

There are three main criteria used to characterize an ES. They are: size of population, number of offspring in each generation and whether the new population is selected from both, parents and offspring, or just from offspring only.

Based on these criteria there are basically four different types of evolution processes. They differ in how the parents for the next generation are selected and whether to or not recombination is used. The four types are (μ, λ) -ES, $(\mu + \lambda)$ -ES, $(\mu/\rho, \lambda)$ -ES, $(\mu/\rho + \lambda)$ -ES. They characterize ES with increasing level of imitation of biological evolution. The letter μ stands for the total number of parents, ρ marks the number of parents, which will be recombined, and λ stands for the number of offspring. Only in the two latter cases offspring are “produced” using mutation and recombination.

The two notations $(\dots, +\lambda)$ and (\dots, λ) describe the selection mechanism, in $(\mu + \lambda)$ -ES and $(\mu/\rho + \lambda)$ -ES the fittest μ individuals out of the union of parents and offspring are selected (so-called plus notation), in (μ, λ) -ES and $(\mu/\rho, \lambda)$ -ES only the fittest μ offspring form the next generation of parents (comma notation). Thus in the latter case λ has to be greater than μ .

The first and simplest evolution process is a $(1+1)$ -ES used in laboratory experiments. The parent individual “produces” one offspring by mutation (random change). The offspring is assigned a fitness value. If the offspring proves to be better than its parent it becomes the next generation’s parent otherwise the parent stays the same. This ES does not incorporate a population typical for evolution but only one parent and one offspring.

As mentioned above, ES is especially suited for real-world optimization problems because typically they involve real numbers. ES works with fewer individuals, and all dimensions are mutated simultaneously. Furthermore the range of the mutation itself is subjected to the evolutionary process.

An example of how the evolution strategy can be applied to a real-world problem is how to find the optimal shape of a lens. The shape of the lens has to be appropriately parameterized (encoded in the genome) for the evolution strategy to work. An animation can be found at the following website:

<http://lautaro.fb10.tu-berlin.de/user/michael/english/lens/lens.html>

Genetic Programming

In genetic programming (GP) the goal is to evolve entire computer programs. The first and most straightforward approach would be – as in the standard GA approach – to directly encode a program into a fixed-length bit-string. This would be easy to do because programs are represented as bit-strings anyhow. However, this would not be suitable mainly for the following two reasons:

First, most bit-strings do not represent valid computer programs. So, assuming, for example, that before mutation, there is in fact a bit-string that does represent a valid program, random mutation and crossover would almost certainly lead to non-functioning programs. If a program is non-functioning, it has zero fitness (or no fitness) and it does not deliver any feedback on the progress of evolution. Thus, it is important that the genetic operators always yield running programs. These programs may often not do much useful work, but since they are running programs, their fitness can be calculated and they influence the evolutionary process. One could object by saying that this is not really a problem, since any non-valid individual could simply be rejected by testing for its validity before accepting the mutation. The GP algorithm would then merely apply its genetic operators (i.e. mutation and recombination/crossover) until it has generated a sufficient number of valid offspring. But for sure such an approach would be extremely inefficient.

Second, genomes as we have discussed so far are limited to a fixed length. Therefore, in a fixed bit-string representation that length would limit the maximum potential complexity of the program. In order to alleviate this problem, the designer might simply specify a very long genome. However, this would significantly increase the computational complexity of the GP algorithm and therefore slow down the rate of progress. A similar problem is encountered when evolving neural networks. Typically, the weights of the network are encoded in the genome and thus the length of the genome determines the maximum size of the neural network (see also below).

In order to cope with these two problems, GP represents a program in a tree structure consisting of a root node, links to further nodes, which can in turn have links to yet further nodes, and so forth, recursively. If the nodes have no further links to other nodes, they are called leaves. In such a tree structure nodes represent processor operations, the instructions such as add, multiply, and divide, whereas the links represent their parameters, i.e. program data. In this scenario, leaves represent parameters and because they are terminals, they encode constants, variables, registers, or external inputs.

Trees can be conveniently represented as list structures, which are linear strings of symbols. This is how programs are represented in the programming language LISP. The tree structure shown in upper-left part of Figure 6.4 corresponds to `add(mul(r1,3),sin(r2,v1))` in linear representation [or `(add (mul r1 3) (sin r2 v1))` in LISP notation]. This is a convenient representation on top of which operations can be defined which lead to other tree structures representing valid programs.

In the first design step, the designer decides on the set of operators, e.g. mul, add, div, pow, etc., the set of possible constants, e.g. +1, -1, +2, -2, and so on, and the set of possible variables, registers as well as inputs. Then, the mutation operator has to be defined in order to ensure that the outcome is again a tree structure representing a valid program. The easiest option is to replace any constant, register, variable, or input by

another, and similarly to replace operators by valid operators. When exchanging operators by other operators, care must be taken if the operators require a different number of parameters. Possible options are to simply discard parameters (in the case of fewer parameters) or to add neutral constants (in the case of more parameters).

When applied to tree structures, the recombination operators work as follows. The algorithm first selects two individuals. Then, the recombination operator selects a node or leaf of each individual's genome. Finally, the operator exchanges the entire sub-trees that are associated with each selected node or leaf respectively. We note that the recombination operator does not have any problem with the number of parameters of an operator because each selected node or leaf originates from one ancestor, and thus will automatically have the correct number of parameters.

Let us illustrate the GP algorithm with the example presented in Figure 6.7. Here, we have two genomes with the list representations $x1 = \text{add}(\text{mul}(r1, 3), \text{sin}(r2, v1))$ and $x2 = \text{add}(\text{add}(v1, \text{sin}(4, v2)), -2)$. First, the selected parents are copied. Then, the algorithm applies the recombination operator at the two randomly chosen positions (as indicated in Fig. 6.5) leading to two new offspring with the linear representations $\text{add}(\text{mul}(r1, 3), \text{sin}(r2, \text{add}(v1, \text{sin}(4, v1))))$ and $\text{add}(v1, -2)$. After applying mutation, the first genome changes to $\text{add}(\text{mul}(r1, 3), \text{sin}(r2, \text{add}(v1, \text{sin}(4, v3))))$.

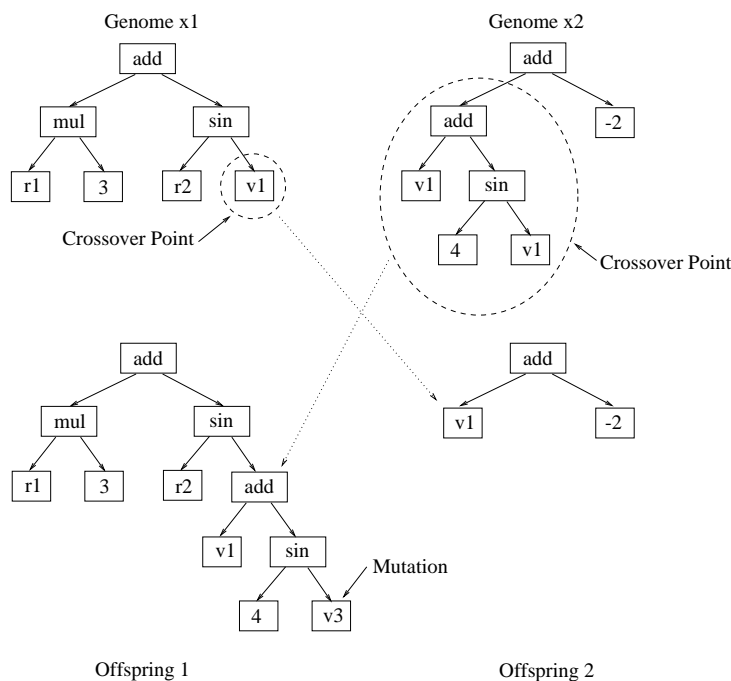


Figure 6.7: Illustration of crossover and mutation on trees.

By exchanging complete sub-trees, the involved individuals are allowed to grow and shrink. The result (the grown or the shrunk individual) is passed on to the selection process that does its work based on fitness evaluation. Growing of trees (genomes) is a fundamental problem in GP, since the search space may grow dramatically. In order to deal with this problem, the designer normally adds an additional term to the fitness function. Let us assume that we have a fitness function $f(x)$. Then, the augmented fitness function $f'(x)$ is defined as $f'(x) = f(x) + c \cdot l(x)$, with $l(x)$ representing the genome's "length" and c is a constant < 0 . This

length might be, for example, the number of nodes including the leaves. The main strategy is that a shorter genome 'x1' has a better fitness $f(x1)$ than a longer genome 'x2', even though both genomes have identical fitness values in terms of the simple fitness function $f(x)$, i.e. $f(x1) = f(x2)$. The constant c plays a critical role in the augmented fitness function $f'(x)$. The designer has to find a good compromise between the genome length and the actual fitness values $f(x)$.

This short example can only provide the basic idea of how the genetic operators can be defined to yield valid programs. The story of GP is more complex than described here – we cannot do full justice to it in such a short summary. We simply wanted to introduce the basic ideas: Applying evolution algorithms to the automatic generation of computer programs, where the individuals have variable length genomes, which can grow and shrink. This feature endows GPs with enormous power, the power to evolve any thinkable machine learning system (e.g. Banzhaf et al., 1998).

It should be noted that the applicability of GPs is by no means restricted to the evolution of computer programs. GP can be applied to any problem that can be reasonably represented by tree structures, a very large class of problems indeed. Thus, applications abound not only in the field of computer programming (acyclic graph evaluation, caching algorithms, sorting algorithms, computer animation, computer security, data compression, etc.), but also in biochemistry, in data mining in general, control of autonomous agents, pattern recognition, signal processing, electronic circuit design etc. For a more comprehensive list of applications, see Banzhaf et al. pp. 342-345.

6.3 Morphogenesis

Often in the field of genetic algorithms, as mentioned earlier, there is no process of development: the phenotype is taken to be identical to the genotype as in the iterated prisoner's dilemma. In this section we extend this view by discussing mainly two approaches: Karl Sims's virtual creatures and Peter Eggenberger's "Artificial Evolutionary System."

Karl Sims's Virtual Creatures

(from Pfeifer and Scheier, 1999, chapter 8, page 245 ff)

Let us now look at a complex example, Karl Sims' virtual creatures (1994a, 1994b). These creatures hold an inescapable fascination---and they are a lot of fun. A number of factors underlie this fascination: First, Sims evolves morphology and neural control. This relieves the human designer of having to come up with a fixed design for the entire low-level specification: The designer commitments are pushed one step further back. We examine these below. And second, Sims was one of the first to use a 3-D world of simulated physics in the context of virtual reality applications. Simulating physics includes considerations of gravity, friction, collision detection, collision response, and viscous fluid effects (e.g. in simulated water).

The details of such simulations are not essential for our purposes; what is essential is that the creatures must perform and compete against each other in this virtual world. As we will see, evolution generates some fascinating morphologies for agents that occupy these virtual worlds, and because of the simulated physics, these agents interact in many unexpected ways with the environment.

In developing his agents, Sims needed to specify the representation of the genotype, the process of development, the selection process, and the reproduction strategy.

Representation of the Genotype

Again, we apply the scheme of Fig. 6.3 to describe Sims' approach. The genotype specifies a directed graph, which is considerably more complex than the bit strings used in the standard evolutionary algorithm. The genotype specifies how the phenotype has to be generated. This requires an interpreter for the genotype that knows what the specifications mean and how the various parts are to be expressed in the phenotype. Figure 6.8 gives an idea of what this might look like.

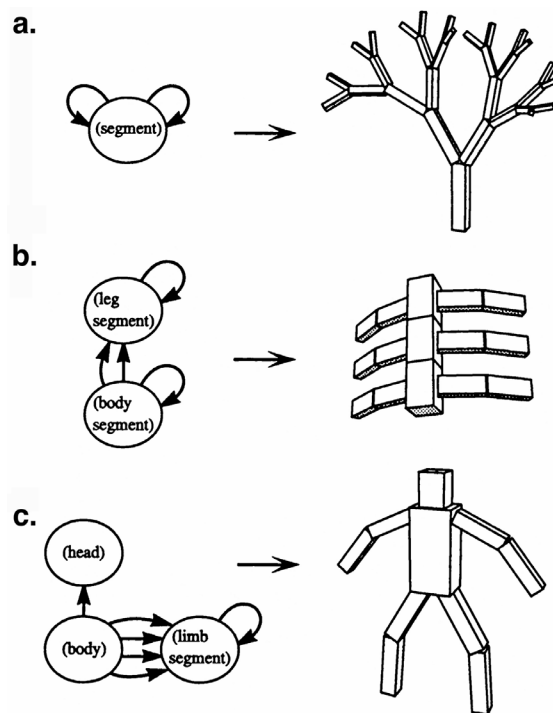


Figure 6.8: Generating a phenotype from a genotype in Sims's approach. (a) A tree-structure: The graph indicates that each segment spawns two other segments. (b) A body and a six-legged creature. (c) A human-like body.

Development

The Sims system uses procedural development. The phenotype consists of a structure of three-dimensional rigid parts described by a directed graph (see Fig. 6.8). The developmental process requires an interpreter for these graph structures, depicted on the left side of each panel in the figure. The graph in Fig. 6.8a, for example, states that two segments are to be simultaneously attached to the existing segment. The shape, length, joint types, angles at which the joints are attached, and various other parameters are all subject to variation by the genetic algorithm; that is, they are subject to mutation and crossover. One of these parameters is the number of times a particular schema is to be applied. In this case, as can be seen on the right of Fig. 6.8a, this number is four, which leads to the standard, treelike structures shown. The examples in Figs. 6.8b and 6.8c are somewhat more complicated, but the principle is the same. The developmental procedure always maps the same genotype onto the same phenotype: There is no interaction with the

environment during development. The genotype also encodes information about sensors, effectors, and neurons that connect the sensors and effectors. The tactile sensors can be put onto all the structures' faces. Light sensors can also be defined in the genotype. Each different morphology (body, limbs, position of sensors) requires a different neural controller to match the morphology's requirements. Figure 6.9 shows how the sensors, effectors and neural connections are encoded.

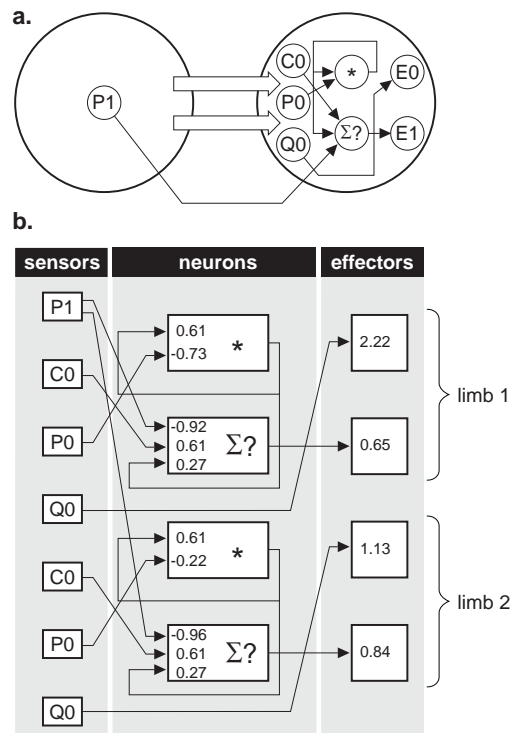


Figure 6.9: Encoding of the sensors, effectors, and neural connections.

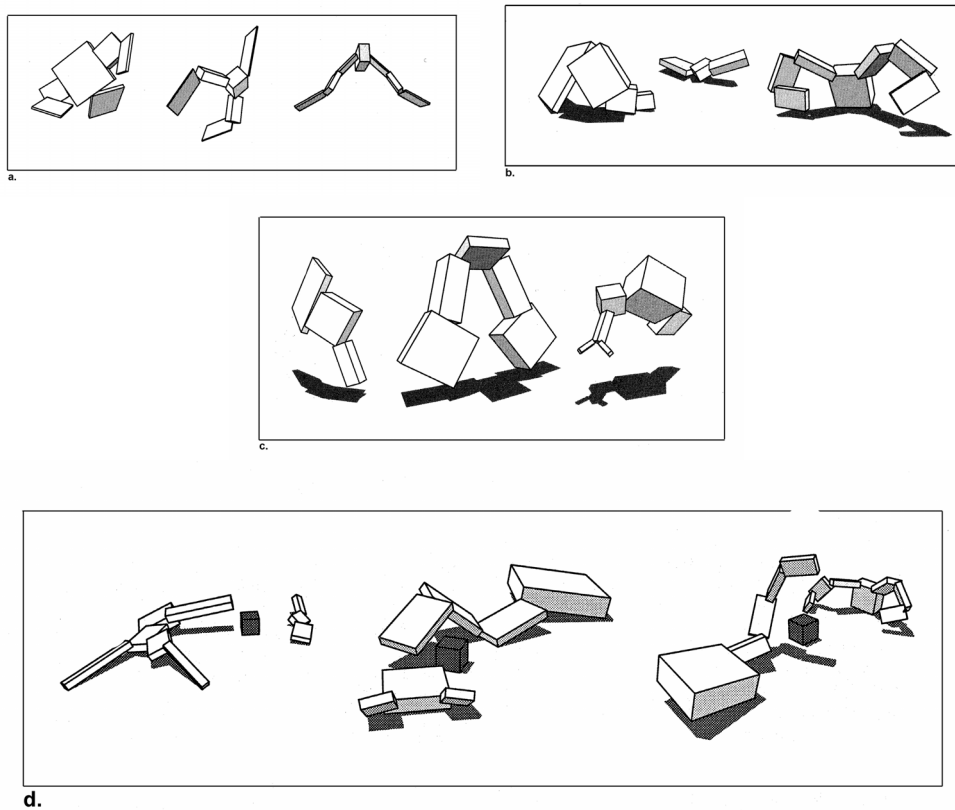


Figure 6.10: Selection of evolved creatures (a) for swimming, (b) for walking, (c) for jumping, and (d) for getting control over a cube.

Fitness and Selection

Once the phenotypes have been generated, they have to perform in the simulated world of physics. Swimming, walking, jumping, following, and getting control over a cube have been used to evaluate the creatures' fitness. The creature that can swim the longest distance within a given period of time is selected, and similarly for walking and jumping. Figure 6.10a shows creatures that have been evolved for swimming. Figures 6.10b and 6.10c show creatures evolved for walking and jumping, respectively. In the case of the following behavior, the average speed at which a creature moves toward a light source is taken as the fitness criterion. In another set of experiments, inspired by nature, the creatures must compete directly for a particular resource. In nature, creatures always have to compete in their ecological niche. In Sims' experiments, they have to try to get control over a cube. The creatures' final distances from the cube were used to calculate their fitness scores (the closer to the cube, the higher the score). Again, the details are not essential for our purposes. Figure 6.10d shows examples of creatures evolved by direct competition. The simulations typically use a population size of 300. Selection is by truncation, meaning that the population is "truncated", so that only the agents in the upper, say, 20 percent survive for reproduction. Furthermore, each surviving individual generates a number of offspring proportional to its fitness.

Reproduction

The creatures are then subjected to a reproduction process that includes mutation and crossover. Both operations are more complicated than in the case of simple GAs in which the genotype is simply a bit string. Here, graph structures have to be manipulated appropriately to yield structures that the developmental process can interpret. (For details on the reproduction process, the reader is referred to Sims' original papers 1994a, 1994b).

What can we learn from this example? First, the example shows that it is indeed possible to evolve creatures even if the morphology is not given a priori. Second, the creatures that evolved were surprising and funny. Especially if we look at their ways of locomotion, we find that they can be truly innovative. For example, one creature in Fig. 6.8b moves by continuously flipping over. Such unexpected things can happen because the search space, that is, the space of possible creatures, is enormous, and the more possibilities there are, the higher are the odds that among them are creatures that can adapt to the demands of the environment. The third lesson follows directly from this point: The larger the search space, the more computation is required. Computation required to evolve these creatures is immense. Not only must we consider the space of connection weights in the neural network we must also consider the space required by possible morphologies. We know that by introducing constraints, we can cut down computation by orders of magnitude. However, and this is the fourth lesson to be gleaned from the example, the more constraints, the fewer the degrees of freedom, and the less surprise. This is similar to the exploration-exploitation trade-off. If everything is entirely unconstrained, we are certain not to get any convergence, that is, there are no creatures with good fitness values. Note that in spite of the fact that morphology is not given, those that result are still very constrained. The possible morphologies are composed of particular types of segments joined in a limited number of ways. This certainly helps evolution to converge, but it also forces it in a particular direction. The final message that we would like to take home from the example above, is that certain kinds of locomotion that can be evolved are not found in natural systems. One example is a creature that sort of rolls over. We see that evolution, at least artificial evolution, is by no means confined to the organisms we find in nature: It potentially exploits anything that is possible. And this is, among many other things, what makes it so fascinating.

Cell Growth from Genome-Based Cell-to-Cell Communication

(from Pfeifer and Scheier, 1999, chapter 8, page 250 ff)

Although vaguely inspired by nature, Sims is not trying to imitate specific natural systems. Natural systems always include a process of development. Although Sims has to translate genotype into phenotype, this process is entirely deterministic: If the genotype is given, the phenotype is determined. Peter Eggenberger, a medical doctor and theoretical physicist, is interested in modeling development from a biological perspective. He wants to create a computer simulation starting from what is currently known about the mechanisms of cell growth. His ultimate goal is to evolve an entire organism: its morphology, its neural substrate, its sensors, and its motor system. In particular, he wants to study the interaction of genetic and environmental factors in development. Of course, it will be a while before this can be achieved. As a first test, Eggenberger's Artificial Evolutionary System was used to grow simple shapes and controllers for

existing robots (Eggenberger 1996, 1997). Because many authors have tackled the latter task, we focus on the first, the evolution of shapes.

The Artificial Evolutionary System is based on the notion of genome-based cell-to-cell communication. What is encoded in the genome in this system is not the organism's structure, but rather the growth processes. Here is how it works: All the cells are placed on the points of a 3-D grid, which is then immersed in a solution of transcription factors: proteins produced by different cells. The concentrations of these transcription factors determine what a cell is going to do next. So let us briefly look at how individual cells "work".

Every cell contains a genome consisting of so-called regulatory genes and structural genes. The regulatory genes determine whether a particular structural gene is turned on. If turned on, the structural genes each perform their predefined functions, namely

- producing a transcription factor (dumping a transcription factor into the environment)
- forming a receptor (forming a receptor on the surface of the cell)
- forming a so-called cell adhesion molecule (CAM) on the surface of a cell
- cell division
- cell death
- searching for partner (searching for matching CAM in the cell environment)

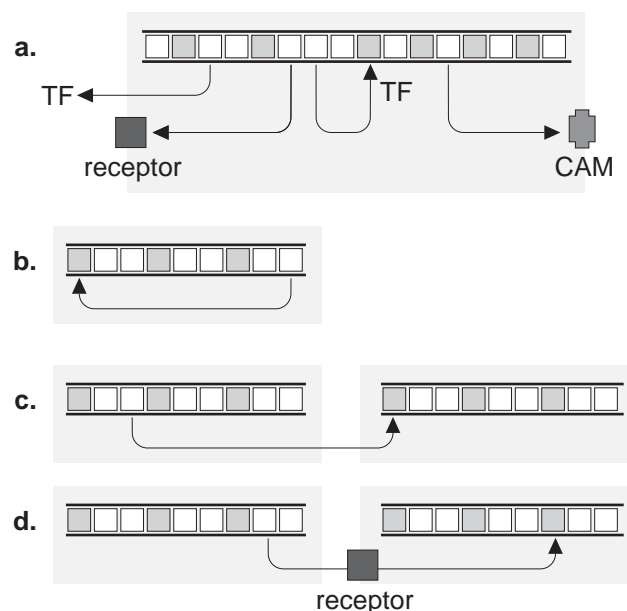


Figure 6.11: Basic mechanisms of the Artificial Evolutionary System. (a) Some basic functions of a gene: production of transcription factor, formation of receptor cell, creation of a CAM, used by other cells to make connections. This is required to build neural networks. (b) A transcription factor influences a regulatory gene within the same cell. (c) A transcription factor diffuses to another cell. (d) A transcription factor binds to an affine receptor.

Figure 6.11 illustrates some of these functions. When a transcription factor is produced, its concentration is highest at the location of the cell where it was produced. Further away on the grid, the concentration of this particular factor is lower because of diffusion. Regulatory genes are activated whenever the concentration of a transcription factor at a particular cell's location is high enough. Whether activation occurs depends on concentration and affinity. Affinity is calculated on the basis of geometric properties: The geometric properties of the transcription factors are compared to the geometric properties of the regulatory gene or the receptor protein on the surface of the cell. These geometric properties are represented in the genotype as sequences of four digits – 1, 2, 3, and 4 – meant to model the four bases of DNA: adenine, thymine, guanine, and cytosine. Figure 6.12 shows the encoding scheme. The typical genome used in the simulations consisted of 8 units. Each unit consisted of two regulatory and two structural genes, for a total of 32 genes (or one regulatory unit and two structural genes, as shown in Figure 6.12).

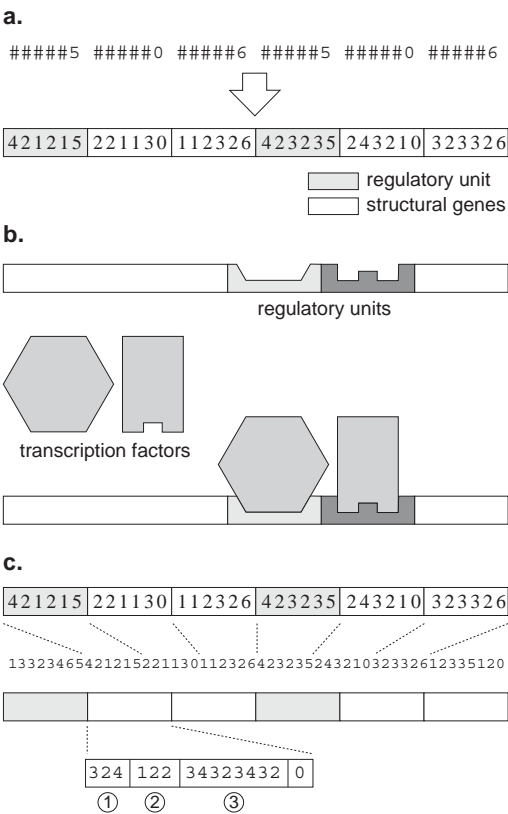


Figure 6.12: Implementation of the genotype. (a) Generation of initial genotype. Panel (b) depicts the matching process. (c) Details of encoding.

Encoding of the Genotype

At the beginning of a simulation run, a sequence of the 4 bases is generated at random (Figure 6.12a). In each cycle, the cells “read” the concentrations of the transcription factors on the 3-D grid where they are located. Depending on the affinity of these transcription factors with the regulatory genes of the cell and their concentration, the regulatory genes are activated and the structural genes turned on. Figure 6.12b shows how a structural gene is activated. Activation in turn causes the structural genes to perform their function. We have discussed the production of a transcription factor and the formation of a receptor cell.

CAMs are used to form connections between cells, connections needed to grow neural networks. CAMs are used together with the function “searching for partner”. If a cell “searches for a partner”, it looks in the cell environment for a matching CAM. The search radius is encoded in the genome (see Figure 6.12c). If a match is found, a connection is established. If the gene for cell division is turned on, the neighboring grid points are searched for an empty space. If all are occupied, the cell does not divide. In other words, cells inside the organism can no longer divide. If the gene for cell death is on, the cell is removed from the grid.

Development, Fitness, and Selection

In this setup, development results from highly complex dynamics. The organism's structure is not predefined in the genome. Figure 6.13 illustrates some of the shapes that have been grown. The goal in these examples was to grow organisms of a fixed size with a T-shape. The fitness function in these examples therefore has two components: (1) the number of cells in the final organism, and (2) a measure of “T-shapeness”. The fitness function was set to 0 if the number of cells in the organism was more than 4,000. The “T-shapeness” measure was implemented as follows: A 3-D model of a T-shape was defined in Cartesian coordinates, and whenever a cell happened to be placed within this shape, the organism's fitness value was increased, otherwise it was decreased. Each generation consisted of 40 individuals. The final organism emerged after 72 generations.

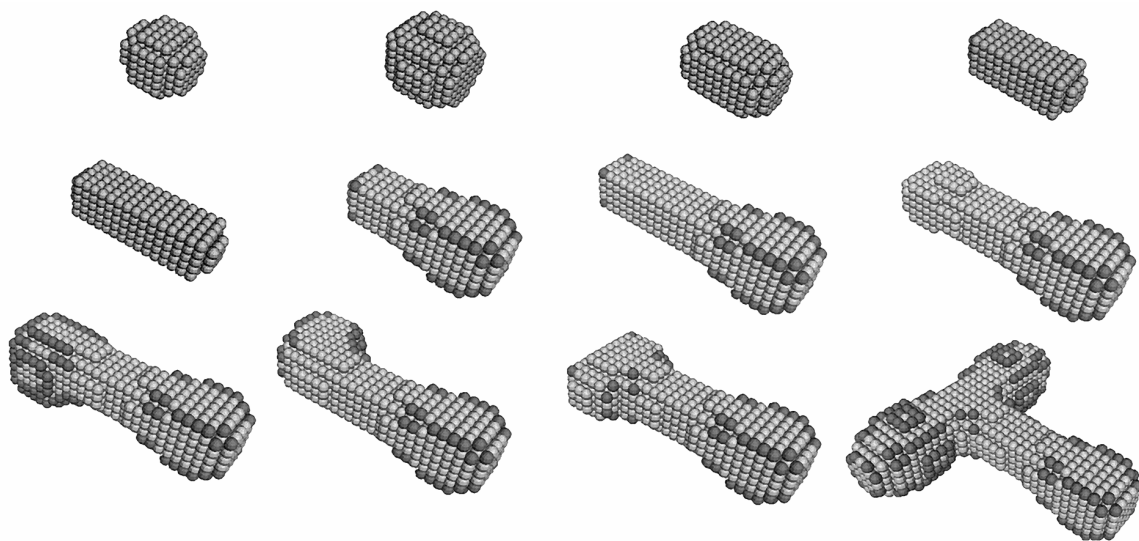


Figure 6.13: *Phylogenetic development of an organism using the Artificial Evolutionary System. The goal was to evolve T-shapes. The shapes of the best individuals are shown after every six generations. The final shape had emerged after 72 generations.*

The procedure, in general, works as follows. Start with a population of 3-D grids; these will eventually host a population of organisms. In each of these grids, put one cell on a grid point. The genome for this cell is initialized to a random sequence of letters, as shown in Figure 6.12a. Depending on the initial concentrations of transcription factors, certain genes will be activated. Let the cells do their work, that is, divide, produce transcription factors, a receptor, a CAM, and so forth. Calculate the new concentrations of the transcription factor for each grid point according to the laws of diffusion. This leads to a modified organism and altered levels of transcription factors. Repeat this cycle for all organisms a preset number of

times. (We have to be careful not to confuse an organism, that is, a collection of cells, with the population of organisms.)

What can we learn from this example? First, it demonstrates a fascinating way of growing entire organisms without predefining their final structure in the genome. This makes the length of the genome independent of the organism's size. Second, the example shows a way biological insights can be translated into a simulation model in a natural way. For example, having the same genome for all the organism's cells and having cell differentiation, as the result of which genes are active, is a biologically motivated assumption. Any resulting organism has emerged because of a complex dynamic process. If this process is influenced, for example, by introducing additional transcription factors, the organism's shape changes. Third, the process of development in this example is more realistic than in the other models discussed so far. The designer does not pre-code the organism's shape. Fourth, as always, computation is expensive. The search space is very large. It is a real challenge to find appropriate constraints. And fifth, at the moment, the organisms have only shape. It would be more realistic if they also displayed interesting behavior. After all, behavior is the business we are interested in.

In summary, although this model is only a beginning, it opens up the possibility of experimenting with shapes.

6.4 Evolution of Hardware

Another interesting question is how the ideas of artificial evolution can be used to evolve hardware. Currently there are technological limitations mainly regarding time and costs, which prevent the application of such ideas in many areas. Nevertheless there are interesting developments in this field such as the configuration of Field Programmable Gate Arrays (FPGA) using Evolutionary Algorithms. An FPGA is a Very Large Scale Integration (VLSI) Silicon Chip containing a large array of components and wires. Switches distributed throughout the chip determine how each component behaves and how the components are connected to the wires. Configuring these switches determines the behavior of an FPGA. The specific arrangement is stored in its configuration memory. Since the FPGA can be interfaced to a host computer its configuration memory can be written to by an Evolutionary Algorithm (EA) (a Genetic Algorithm, Evolution Strategies or Evolutionary Programming; see Section 6.2) running on the host computer. By setting the switches the EA creates a physical (real) electronic circuit, which can be tested and evaluated according to its real-world performance.

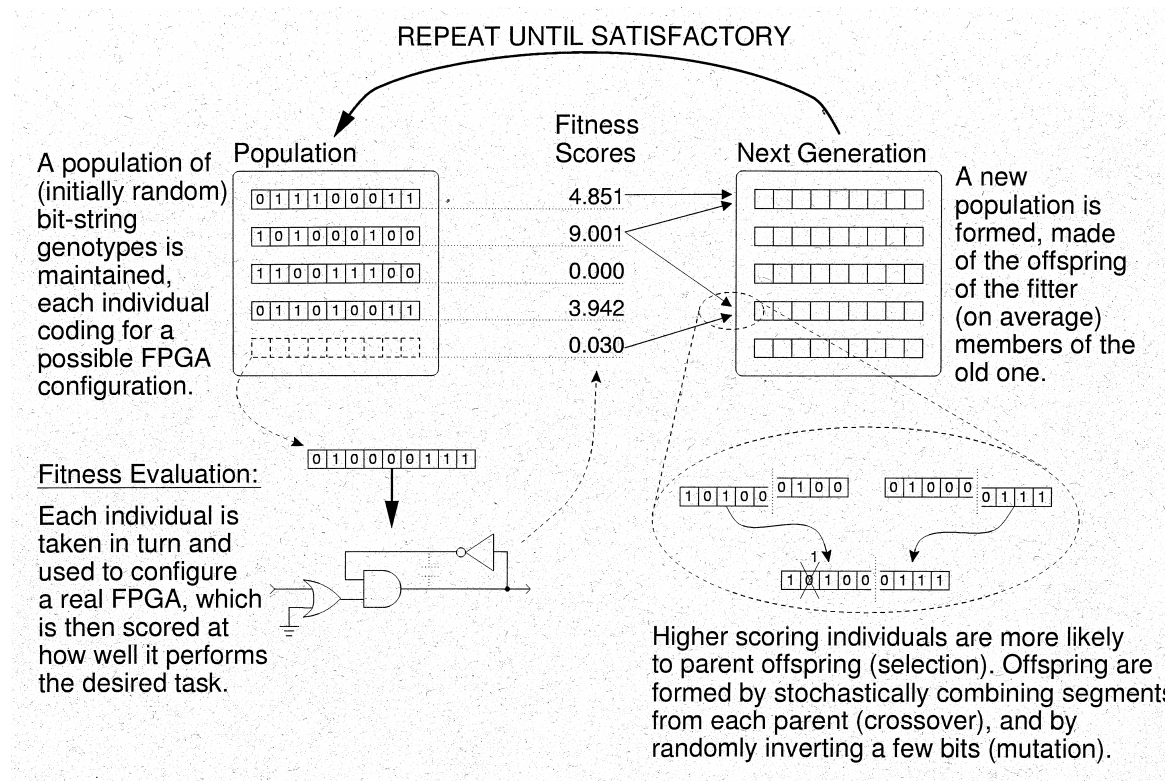


Figure 6.14: Evolving an FPGA configuration using a simple genetic algorithm.

Adrian Thompson of the University of Sussex used the above setup to design electronic circuits. The circuit has a single input and a single output. Its task is to distinguish between 1kHz and 10kHz audio tones (a possible first step to pattern recognition and signal processing). Its fitness function consists in maximizing the difference in average output voltage caused by the two different input signals. In the experimental setting no synchronizing clock is used (consequently the input period cannot be timed) therefore evolution can exploit the rich natural unconstrained dynamics of the silicon to achieve the task. By automatically designing electronic circuits using an evolutionary algorithm major differences between the circuits designed according to conventional goal-oriented methods and those configured using evolutionary methods could be discovered. Perfect behavior was achieved using only a 10x10 array out of the whole 64x64 array to implement an electronic circuit (see Figure 6.15) whereas conventional design would need a much larger area to achieve the same performance. Within the 10x10 array not all cells contributed to the circuits behavior and among them are cells which are not even connected to the main part of the circuit but do still influence the behavior of the system (see cells shaded in gray in Figure 6.15). These components must be interacting with the rest of the system by other means (e.g. electromagnetic coupling or power-supply loading). The detailed physical properties of the silicon the spatial relationships of the components and their interactions have been exploited by evolution in a fascinating way. A circuit evolved with an EA can use the electronic resources more effectively than an equivalent circuit designed by conventional methods.

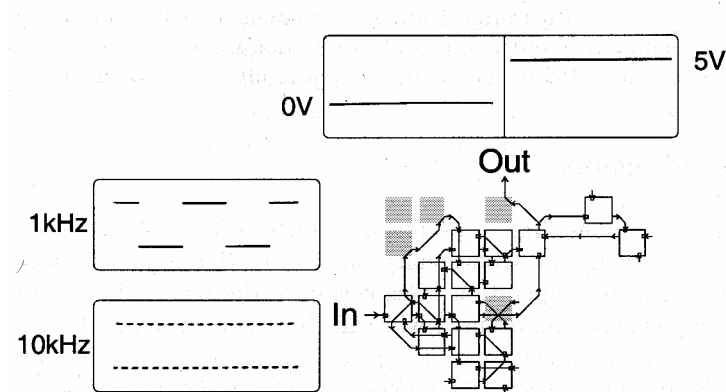


Figure 6.15: The tone-discriminator evolved on a single FPGA to distinguishing between two input signals. Note the cells shaded in gray which are not connected to the main part of the circuit but do influence the circuit's behavior.

Since “unconstrained” circuits do not fulfill the present norms and expectation of the industry Thompson is concentrating on the evolution of robust circuits. Robustness is especially important for the use of so evolved circuits in real-world applications. Robustness means in this context that the circuit is able to operate in a satisfactory way even when variations in its environment or implementation occur.

6.5 Conclusion

The field of artificial evolution has grown into a large research field of its own. By copying certain aspects of biological systems artificial evolution helps to understand biological evolution and to solve optimization problems using new and unusual approaches. In this chapter we could not cover all parts of artificial evolution, topics we have not mentioned include more biologically inspired approaches such as Tom Ray's Tierra system. Besides we only touched the broad field of co-evolution when discussing the prisoner's dilemma in Section 6.2.

Our knowledge of biological evolution and tied to it our knowledge of artificial evolution is not yet complete but it is rapidly evolving. We hope to have at least conveyed the basic ideas of this fascinating research field.

6.6 Bibliography

- Axelrod, R. (1987). The evolution of strategies in the iterated prisoner's dilemma. In: L.D. Davis (ed.). *Genetic Algorithms and Simulated Annealing*. Pitman: London, 32-41.
- Bäck, T.B. and Schwefel, H.-P. (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1:1-23.
- Bäck, T.B. (1996). *Evolutionary Algorithms in Theory and Practice*, New York, Oxford University Press.
- Banzhaf, W., Nordin, P., Keller, R.E. and Francone, F.D. (1998). *Genetic Programming — An Introduction*. Morgan Kaufmann: San Francisco, CA.
- Bongard, J.C. (2002). Evolving modular genetic regulatory networks. In: *Proc. of the Congress on Evolutionary Computation*, 305-311.
- Dawkins, R. (1986). *The Blind Watchmaker*. Longman.

- De Jong, K.A. (1975). *An analysis of the behavior of a class of genetic adaptive systems*. Ph.D. dissertation, University of Michigan, Ann Arbor.
- Eggenberger, P. (1997). Creation of neural networks based on developmental and evolutionary principles. *Proc. Int. Conf. on Artificial Neural Networks*.
- Eggenberger, P. (1997). Evolving morphologies of simulated 3D organisms based on differential gene expression. *Proc. Fourth European Conference on Artificial Life*.
- Eggenberger, P. and Dravid, R. (1999). An evolutionary approach to pattern formation mechanisms on lepidopteran wings.
- Eggenberger, P. (1999). *Evolution of three-dimensional artificial organisms: simulations of developmental processes*. Unpublished PhD Dissertation, Medical Faculty, University of Zurich.
- Flake, G.W. (1998). *The Computational Beauty of Nature*. MIT Press: Cambridge, MA.
- Fogel, D.B. (1995). Computer simulation of natural evolution. In *Evolutionary computation: Toward a new philosophy of machine intelligence*. In D.B. Fogel (Ed.) (pp. 67-119). Piscataway, NJ: IEEE Press.
- Fogel, L.J. (1962). Autonomous automata *Industrial Research* 4, 14-19.
- Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley: Reading, MA.
- Holland, J.H. (1975). *Adaption in Natural and Artificial Systems*. University of Michigan Press: Ann Arbor.
- Koza, J.R. (1992). *Genetic Programming: On the Programming of Computers by Natural Selection*. Cambridge, MA: MIT Press.
- Koza, J.R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press: Cambridge, MA.
- Lipson, H. and Pollack, J.B. (2000). Automatic design and manufacture of artificial life-forms. *Nature*, 406:974-978.
- Mitchell, M. (1997). *An Introduction to Genetic Algorithms*. MIT Press: Cambridge, MA.
- Pfeifer, R. and Scheier, C. (1999). *Understanding Intelligence*. MIT Press: Cambridge, MA.
- Pfeifer, R. and Bongard, J.C. (2006). *How the Body Shapes the Way we Think*. MIT Press: Cambridge, MA.
- Rechenberg, I. (1973). *Evolutionsstrategie Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzbog: Stuttgart.
- Rechenberg, I. (1994). *Evolutionsstrategie '94*. Frommann-Holzbog: Stuttgart.
- Schwefel, H.-P. (1977). *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*. Birkenhäuser: Basel.
- Sims, K. (1994a). Evolving virtual creatures. *Computer Graphics*, 28:15-34.
- Sims, K. (1994b). Evolving 3D morphology and behavior by competition. In R. Brooks and P. Maes (eds.), *Artificial Life IV Proceedings*. MIT Press: Cambridge, MA, 28-39.
- Srinivas, M., and Patnaik, L.M. (1994). Genetic algorithms: a survey. *IEEE Computer*, 27:17-26.
- Thompson, A. (1998). On the automatic design of robust electronics through artificial evolution. In M. Sipper, D. Mange, A. Perez-Urbe (eds.), *From Biology to Hardware: Proc. of Second Int. Conf. on Evolvable Systems*. Lecture Notes in Computer Science 1478, Springer-Verlag: Berlin, New York.
- Thompson, A. (1997). Artificial evolution in the physical world. In: T. Gomi (ed.), *Evolutionary Robotics: From Intelligent Robots to Artificial Life (ER'97)*, AAAI Books.
- Thompson, A. and Layzell, P. (2000). Evolution of robustness in an electronic design. *From Biology to Hardware: Proc. third International Conference on Evolvable Systems*, Lecture Notes in Computer Science 1801, Springer-Verlag: Berlin, New York, pp. 218-228.