Chapter 5
Evolutionary Computing

Evolutionary computing is the most important group of methods within the *biology-inspired approach*, because of their well-developed theoretical foundations as well as the variety of their practical applications. As has been mentioned in Sect. 3.2, the main idea of these methods consists of simulating natural evolutionary processes. Firstly, four types of such methods are presented, namely *genetic algorithms*, *evolution strategies*, *evolutionary programming*, and *genetic programming*. In the last section other biology-inspired models, such as *swarm intelligence* and *Artificial Immune Systems*, are introduced.

5.1 Genetic Algorithms

The first papers of Alex Fraser concerning *genetic algorithms* were published in 1957 [102]. However, this approach only became popular after the publication of an excellent monograph by Holland [139] in 1975. As we have mentioned at the end of the previous chapter, genetic algorithms can be treated as a significant extension of the heuristic search approach that is used for finding the optimum solution to a problem. In order to avoid finding local extrema instead of the global one and to avoid getting stuck in a *plateau* area (cf. Fig. 4.11), a genetic algorithm goes through a *space of* (potential) *solutions* with many search points, not with one search point as in (standard) heuristic search methods. Such search points are called *individuals*.[1] Thus, each individual in a solution space can be treated as a candidate for a (better or worse) solution to the problem. A set of individuals "living" in a solution space at any phase of a computation process is called a *population*. So, a population is a

[1]The analogy is with individuals living in biological environments.

© Springer International Publishing Switzerland 2016

M. Flasiński, *Introduction to Artificial Intelligence*,
DOI 10.1007/978-3-319-40022-8_5

set of representations of potential solutions to the problem. Succeeding populations constructed in iterated phases of a computation are called *generations*. This means that a *state space* in this approach is defined with succeeding generations constructed by a genetic algorithm.[2]

For example, let us look at an exemplary solution space shown in Fig. 5.1. The position of each individual in this space is determined by its coordinates (X_1, X_2). These coordinates determine, in turn, the *genotype* of the individual. Usually, it is assumed that a genotype consists of one *chromosome*. Each coordinate is *binary-coded*, i.e., it is of the form of a string of *genes*: 0000, 0001, 0010, ..., 1000, etc. Thus, a genotype built of one chromosome consists of eight genes. The first four genes determine the coordinate X_1 and the second four genes determine the coordinate X_2. For example, the individual marked with a circle in the solution space in Fig. 5.1 has the genotype 01010111. As we have mentioned in the previous chapter, each search point of a space of (potential) solutions represents a set of values ascribed to parameters of the problem considered.[3] In the terminology of genetic algorithms such a set of values is called the *phenotype* of this point/individual. In order to evaluate the "quality" of an individual (i.e., a potential solution), we evaluate its phenotype with the help of a *fitness function*.[4]

Let us assume for further considerations that we look for the (global) maximum of our fitness function, which equals *11* and is marked with a dark grey color in Fig. 5.1. In the solution space there are two local maxima with a fitness function value equal to *8*, which are marked with a light grey color. Let us notice that if we searched this space with the hill-climbing method and we started with the top-leftmost point, i.e., the point $(X_1, X_2) = (0000, 1000)$ with a fitness function value equal to *3*, then we would climb a local "peak" having coordinates $(X_1, X_2) = (0001, 0111)$ with a fitness function value equal to *8*. Then, we would stay at the "peak", because the fitness function gives worse values in the neighborhood of this "peak". However, this would make it impossible to find the best solution, i.e., the global maximum. Using genetic algorithms we avoid such situations.

Now, let us introduce the scheme of a genetic algorithm, which is shown in Fig. 5.2. Firstly, the initial population is defined by random generation of a fixed number (a parameter of the method) of individuals. These individuals are our initial search points in the solution space. For each individual the value of the fitness function is computed. (This corresponds to the evaluation of a heuristic function for the search methods discussed in a previous chapter.) In the next phase the best-fitted individuals are *selected* for "breeding offsprings". Such individuals create a *parent population*.

[2]Thus, succeeding populations are equivalent to states of this space.

[3]Each search point corresponds to a certain *solution*. (Such a "*solution*" does not satisfy us in most cases.) If we deal with an abstract model of a problem (as in the previous chapter), not with a solution space, then such a point corresponds to a certain phase of problem solving (for our example of a labyrinth it is the path we have gone down) instead of representing values ascribed to parameters of the problem.

[4]The fitness function plays an analogous role to the heuristic function defined for the search methods presented in the previous chapter.

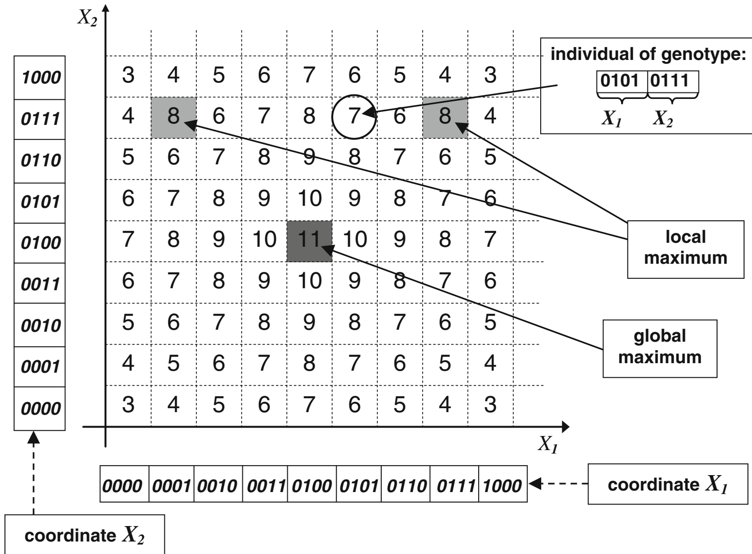

Fig. 5.1 Formulation of a search problem for a genetic algorithm

The simplest method for such a selection is called *roulette wheel selection*. In this method we assume that the roulette wheel area assigned to an individual is directly proportional to its fitness function value. For example, let the current population consist of four individuals $P = \{ind_1, ind_2, ind_3, ind_4\}$ for which the fitness function h gives the following values: $h(ind_1) = 50$, $h(ind_2) = 30$, $h(ind_3) = 20$, $h(ind_4) = 0$. Then, taking into account that the sum of the values equals 100, we assign the following roulette wheel areas to individuals: $ind_1 = 50/100\,\% = 50\,\%$, $ind_2 = 30\,\%$, $ind_3 = 20\,\%$, $ind_4 = 0\,\%$. Then, we randomly choose individuals with the help of the roulette wheel. Since no wheel area is assigned to the individual ind_4 (0 %), at least one of the remaining individuals must be chosen twice. (The individual ind_1 has the best chance, because its area comprises half of the wheel, i.e., the same area as ind_2 and ind_3 combined.)

In order to avoid a situation in which some individuals with a very small fitness function value (or even the zero value, as in the case of the individual ind_4) have no chance of being selected for "breeding offsprings", one can use *ranking selection*. In this method a ranking list which contains all the individuals, starting from the best-fitted one and ending with the worst-fitted one, is defined. Then, for each individual a *rank* is assigned. The rank is used for a random selection. For example, the rank can be defined in the following way. Let us fix a parameter for computing a rank, $p = 0.67$. Then, we choose the individual ind_1 from our previous example with probability $p_1 = p = 0.67$. The individual ind_2 is selected with probability $p_2 = (1 - p_1) \cdot p = 0.33 \cdot 0.67 = 0.22$. The succeeding individual ind_3 is chosen with probability $p_3 = (1-(p_1+p_2)) \cdot p = 0.11 \cdot 0.67 = 0.07$. Generally, the nth individual

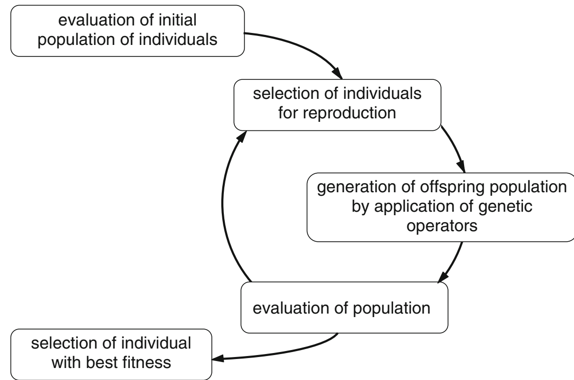

Fig. 5.2 The scheme of a genetic algorithm

from the ranking list is selected with probability $p_n = (1-(p_1+p_2+\cdots+p_{n-1}))\cdot p$. Let us notice that according to such a scheme we assign a non-zero value to the last individual ind_4, i.e., $p_4 = 1-(p_1+p_2+p_3) = 1-0.96 = 0.04$.

After the selection phase, an *offspring population* is generated with the help of *genetic operators* (cf. Fig. 5.2). Reproduction is performed with the *crossover (recombination) operator* in the following way. Firstly, we randomly choose[5] pairs of individuals from the parent population as candidates for mating. These pairs of parents "breed" pairs of offspring individuals by a recombination of sequences of their chromosomes. For each pair of parents we randomly choose the *crossover point*, which determines the place at which the chromosome is "cut". For example, a crossover operation for two parent individuals having chromosomes 01001000 and 01110011, with a fitness function value of *7* (for both of them) is depicted in Fig. 5.3. The randomly chosen crossover point is *4*, which means that both chromosomes are cut after the fourth gene. Then, we recombine the first part of the first chromosome, i.e., 0100, with the second part of the second chromosome, i.e., 0011, which gives a new individual (offspring) with the chromosome 01000011. In the same way we obtain a second new individual having the chromosome 01111000 (by a recombination of the first part of the second chromosome and the second part of the first chromosome). Let us notice that one "child" (the one having the chromosome 01111000) is "worse fitted" (to the environment) than the "parents", because its fitness function value equals *4*. This individual corresponds to a worse solution of the problem. On the other hand, the fitness function value of the second "child" (01000011) equals *10* and this individual reaches a satisfying solution (the global maximum) in one step. Sometimes we use more than one crossover point; this technique is called *multiple-point crossover*.

[5]This random choice is made with high probability, usually of a value from the interval [0.6, 1] in order to allow many parents to take part in a reproduction process. This probability is a parameter of the algorithm.

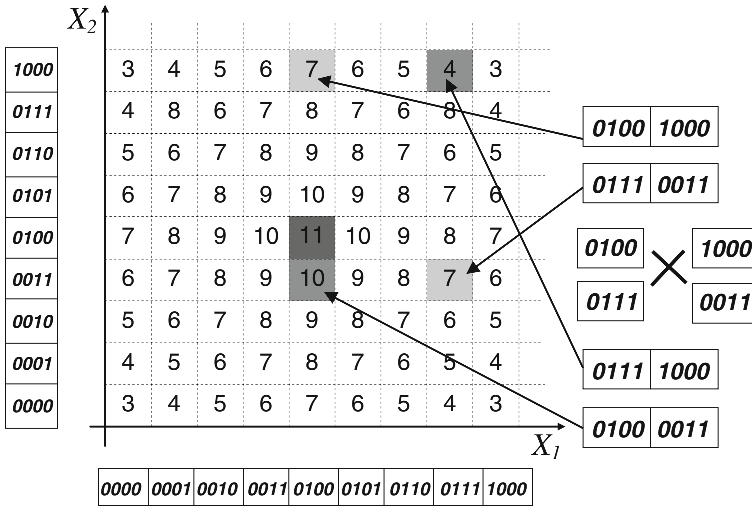

Fig. 5.3 Applying the crossover operator in a genetic algorithm

The *mutation operator* changes the value of a single gene from 0 to 1 or from 1 to 0. Contrary to a crossover, a mutation is performed very rarely.[6] For example, in Fig. 5.4 we can see an individual represented by the chromosome 01110111. Although its fitness function value (*8*) is quite big, this individual is a local maximum. If we used this individual in the hill-climbing method, then we would get stuck in this place (cf. Fig. 4.11). However, if we mutated the third gene of its chromosome from 1 to 0, then we would obtain a new individual, which has the chromosome 01010111 (cf. Fig. 5.4). This search point has a better chance to reach the global maximum.[7]

In the third phase, called *evaluation of a population*, values of the fitness function are computed for individuals of the new offspring population (cf. Fig. 5.2). After the evaluation, the termination condition of the algorithm is checked. If the condition is fulfilled, then we select the individual with the best fitness as our solution of the problem. The termination condition can be defined based on a fixed number of generations determined, the computation time, reaching a satisfying value of the fitness function for some individual, etc. If the condition is not fulfilled, then the work of the algorithm continues (cf. Fig. 5.2).

In this section we have introduced fundamental notions and ideas for genetic algorithms. As we can see, this approach is based on biological intuitions. Of course, for the purpose of constructing an AI system, we should formalize it with mathematical notions and models. The *Markov chain* model is one of the most elegant formalizations used in this case. This model is introduced in Appendix B.2.

---

[6] Since mutations occur very rarely in nature, we assume a small probability in this case, e.g., a value from the interval [0, 0.01]. This probability is a parameter of the algorithm.

[7] In genetic algorithms mutation plays a secondary role. However, as we will see in subsequent sections, mutation is a very important operator in other methods of evolutionary computing.

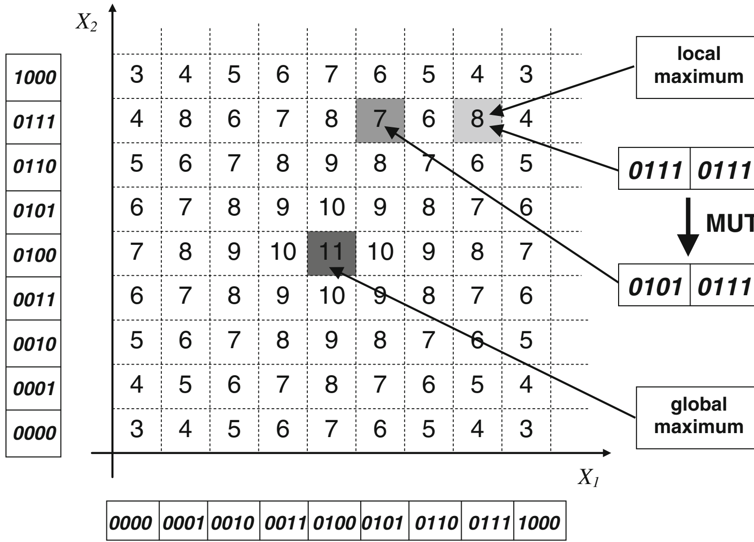

Fig. 5.4 Applying the mutation operator in a genetic algorithm

5.2 Evolution Strategies

In the genetic algorithm discussed in the previous section, an individual (a potential solution) has been coded in binary. Such a representation is convenient if we are searching a *discrete* solution space,[8] for example in the case of a discrete optimization problem. *Evolution strategies* were developed in the 1960s by Rechenberg [236] and Schwefel [267] at the Technical University of Berlin in order to support their research into *numerical optimization* problems.[9] In this approach, an individual is represented by a pair of vectors $(\mathbf{X}, \boldsymbol{\sigma})$, where $\mathbf{X} = (X_1, X_2, \ldots, X_n)$ determines the location of the individual in the n-dimensional (continuous) solution space,[10] $\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ is a string of parameters of the method.[11]

Let us discuss the general scheme of an evolution strategy shown in Fig. 5.5a. Similarly to the case of genetic algorithms, we begin with the initialization and evaluation of a μ-element parent population R. Then, we start the basic cycle of the method, which consists of three phases.

During the first phase λ-element offspring population O is generated. Each descendant is created in the following way. Firstly, ρ individuals, which will be used

---

[8] An example of a discrete solution space has been defined in the previous section in Fig. 5.1.

[9] The research into fluid dynamics was carried out at the Hermann Föttinger Institute for Hydrodynamics at TUB.

[10] A vector $\mathbf{X}$ represents here the chromosome of the individual, and its components $X_1, X_2, \ldots, X_n$, being real numbers, correspond to its genes.

[11] A parameter σ_i is used for mutating a gene X_i.

for the production of a given descendant, are chosen.[12] These "parents" are drawn with replacement according to the uniform distribution.[13] Then, these ρ parents produce a "preliminary version" of the descendant with the crossover operator. After recombination the element $\boldsymbol{\sigma}$ of the child chromosome, which contains parameters of the method, is mutated. Finally, a mutation of the individual, i.e., a mutation of the element $\mathbf{X}$ of its chromosome, is performed with the help of the mutated parameters of $\boldsymbol{\sigma}$.

In the second phase an evaluation of the offspring population O is made. This is performed in an analogous way to genetic algorithms, that is with the help of the fitness function.

The third phase consists of the selection of μ individuals to form a new parent population P according to the values of the fitness function. There are two main approaches to selection. In the selection of the $(\mu + \lambda)$ type a choice is made from among individuals which belong to both the (old) parent population and the offspring population. This means that the best parents and the best children create the next generation.[14] In selection of the (μ, λ) type we choose individuals to form the next generation from the offspring population.

Similarly to genetic algorithms, a termination condition is checked at the end of the cycle. If it is fulfilled, the best individual is chosen as our solution to the problem. Otherwise, a new cycle is started (cf. Fig. 5.5a).

After describing the general scheme let us introduce a way of denoting evolution strategies [23]. We assume an interpretation of parameters μ, λ, ρ as in the description above. If we use selection of the $(\mu + \lambda)$ type, then the evolution strategy is denoted by $(\mu/\rho + \lambda)$. If we use selection of the (μ, λ) type, then the evolution strategy is denoted by $(\mu/\rho, \lambda)$.

Now, we present a way of defining genetic operators for evolution strategies.

Let us assume that both parents *Father* and *Mother* are placed in a solution space according to vectors $\mathbf{X^F} = (X_1^F, X_2^F)$ and $\mathbf{X^M} = (X_1^M, X_2^M)$, respectively, as shown in Fig. 5.5b.[15] Since an individual is represented by a vector of real numbers, calculating the average of the values of corresponding genes which belong to the parents is the most natural way of defining the *crossover operator*. Therefore, the position of *Child* is determined by the vector $\mathbf{X^C} = (X_1^C, X_2^C)$, where $X_1^C = (X_1^F + X_1^M)/2$ and $X_2^C = (X_2^F + X_2^M)/2$ (cf. Fig. 5.5b).

In case of crossover by averaging, we also compute $\boldsymbol{\sigma^C}$ by taking the average values of $\boldsymbol{\sigma^F}$ and $\boldsymbol{\sigma^M}$, which represent the parameters of the method. An exchange of single genes of parents can be an alternative way of creating an offspring.

[12]This means that a "child" can have more than two "parents".

[13]Firstly, this means that each individual has the same chance to breed an offspring (the uniform distribution). Secondly, any individual can be used for breeding several times (drawing with replacement). This is the main difference in comparison to genetic algorithms, in which the best-fitted individuals have better chances to breed an offspring (roulette wheel selection, ranking selection).

[14]As one can see, we try to improve on the law of Nature. A "second life" is given to outstanding parents.

[15]The reader is advised to compare Fig. 5.5b with Fig. 4.11 in the previous chapter. For clarity there is no axis $h(X_1, X_2)$ corresponding to the fitness function in Fig. 5.5b.

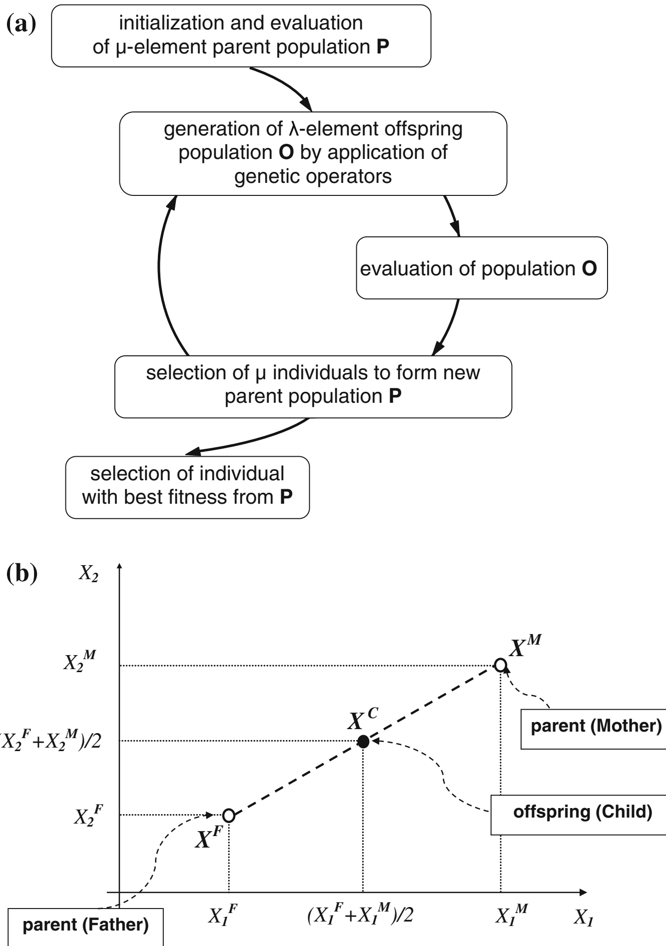

Fig. 5.5 **a** The scheme of an evolution strategy, **b** an example of crossover by averaging in an evolution strategy

As we have mentioned above, firstly, *mutation* is performed for an element $\boldsymbol{\sigma}$, secondly for an element **X**. A mutation of the element $\boldsymbol{\sigma}$ consists of multiplying its every gene by a certain coefficient determined by a random number,[16] which is generated according to the normal distribution.[17] After modifying the vector $\boldsymbol{\sigma}$, we use it for a replacement of the individual in the solution space, as shown in Fig. 5.6. As we can see in the figure, the position of the individual represented by **X** is determined by its genes (coordinates) X_1 and X_2. Now, we can e.g., mutate its gene X_1 by adding

[16]Sometimes, the coefficient is determined by several random numbers.

[17]Formal definitions of notions of probability theory which are used in this chapter are contained in Appendix B.1.

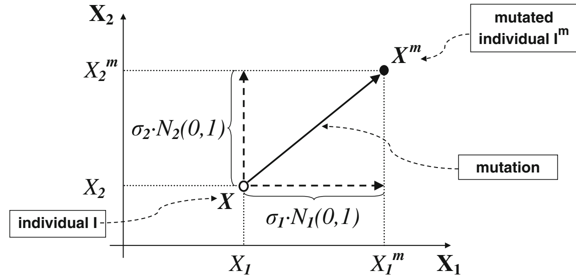

Fig. 5.6 Mutation of an individual in an evolution strategy

a number $\sigma_1 \cdot N_1(0, 1)$, where σ_1 is its corresponding gene—and the parameter of the method—and $N_1(0, 1)$ is a random number generated according to the normal distribution with an *expected value (average)* equal to 0 and a *standard deviation* equal to 1.

Let us notice that the element $\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ contains parameters which determine how big a mutation is.[18] As we have seen, these parameters are modified,[19] which means that evolution strategies are self-adapting.

5.3 Evolutionary Programming

In 1966 Lawrence J. Fogel introduced an approach to evolutionary computing which is called *evolutionary programming* [99]. The main idea of this approach differs from the methods discussed above. One difference concerns the level of abstraction at which evolution processes are simulated. In genetic algorithms and evolution strategies search points in a solution space correspond to individuals in a population. However, in the case of evolutionary programming instead of individuals we deal with a species-level abstraction.[20] This influences, of course, how the method is constructed. First of all, there is no crossover operation, since there is no crossover among species. Secondly, a mutation is defined in such a way that radical changes occur with low probability and small changes are preferred.

The second important difference with respect to the methods introduced in previous sections is the fact that in evolutionary programming we do not assume any specific form of representation of an individual.[21] A representation of an indi-

[18] Bigger values of these parameters cause a bigger change to an individual in a solution space. Strictly speaking, the bigger the value of a parameter σ_k, the more the gene X_k is mutated, which corresponds to moving the individual along the $\mathbf{X_k}$ axis.

[19] The probabilities of both a crossover and a mutation are constant parameters in genetic algorithms.

[20] Let us remember that a *population* is a set of individuals of the same species which live in the same area. Thus, in the case of evolutionary programming we should rather use the term *biocoenosis* instead of *population*, which is more correct from the point of view of biology.

[21] We have assumed a binary representation of individuals in genetic algorithms and real number vectors in evolution strategies.

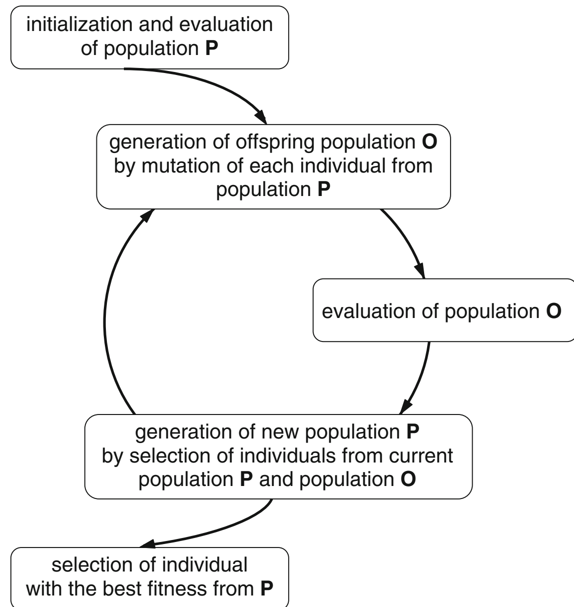

Fig. 5.7 The scheme of evolutionary programming

vidual should, simply, be adequate for a given problem. A variety of representations (variable-length vectors, matrices, etc.) are used in evolutionary programming projects for defining abstract models of problems.

Now, we present a scheme of evolutionary programming, which is shown in Fig. 5.7. The initialization and evaluation of a (parent) population P is a preliminary phase. Then, we begin the basic cycle of the method. The generation of the offspring population O by a mutation of each individual from the population P is performed in the first phase. Mutation is made randomly according to the normal distribution. The second phase consists of the evaluation of the population O. The generation of a new population P by a selection of individuals from the current population P and the population O is performed in the third phase. In a standard version of evolutionary programming a ranking selection is applied for this purpose. Then, as in previous methods, a termination condition is tested. If it is fulfilled, we choose the best individual as the solution to the problem. If not, a new cycle is begun.

At the end of the twentieth century David Fogel[22] introduced two improvements in evolutionary programming. Firstly, instead of ranking selection, a certain variant of *tournament selection* is applied. Tournament selection consists of dividing a population into groups which usually contain several individuals and selecting the best

[22]David Fogel—a researcher in the area of evolutionary computing. In his famous research project *Blondie24*, an evolutionary-computing-based AI system became an expert in English draughts (checkers). Fogel has been the President of the IEEE Computational Intelligence Society and the Editor-in-Chief of IEEE Transactions on Evolutionary Computation. He is the son of Lawrence J. Fogel.

individuals from each group separately. This method is especially useful for multi-criteria optimization problems, when we optimize more than one function. Secondly, D. Fogel has introduced self-adapting mechanisms similar to those used in evolution strategies.

Since no specific form of representation of an individual is assumed in evolutionary programming, the approach may be applied to a variety of problems, e.g., control systems, pharmaceutical design, power engineering, cancer diagnosis, and signal processing. In Artificial Intelligence the approach is used not only for solving problems, mainly optimization and combinatorial problems, but also for constructing self-learning systems.

In fact, the history of this approach began in 1966 in the area of self-learning systems. L.J. Fogel in his pioneering paper [99] discussed the problem of formal grammar induction,[23] strictly speaking the problem of synthesizing a formal automaton[24] on the basis of a sample of sentences belonging to some language. A formal automaton is a system used to analyse a formal language. The synthesis of an automaton A by an AI system consists of an automatic construction of A on the basis of a sample of sentences, which belong to a formal language. L.J. Fogel showed that such a synthesis can be made via evolutionary programming. In his model a formal automaton evolves by the simulation of processes of crossover and mutation in order to be able to analyze a formal language. Let us notice a difference between problem solving by genetic algorithms/evolution strategies discussed previously and solving the problem of synthesis of an automaton by the AI system constructed by L.J. Fogel. In the first case, generation of *a problem solution* is the goal of the method, whereas in the second case we want to generate *a system* (automaton) that solves a certain class of problems (a formal language analysis). Such an idea would appear twenty years later in the work of M.L. Cramer, which concern genetic programming. This approach is introduced in the next section.

5.4 Genetic Programming

Although *genetic programming* was popularized in the 1990s by John Koza due to his well-known monograph [172], the main idea of this approach was introduced in 1985 by Cramer [61]. In genetic programming instead of searching a solution space with the help of a program, which is implemented on the basis of principles of evolution theory, a population of programs is created. Then, a space of programs is searched in order to find the one which can solve a class of problems in a satisfactory way. Of course, we have to define a function, to assess the quality (adequacy) of such programs. Thus, automatic synthesis of a computer program to solve a given problem is the objective of genetic programming. This objective has been extended to other

[23] This problem is discussed in Sect. 8.4.

[24] Formal automata are introduced in Sect. 8.2.

systems in the technical sciences, such as digital circuits (electronics), controllers (automatic control), etc.[25]

In order to achieve such an ambitious objective, a human designer has to deliver certain knowledge to an AI system [173]. Firstly, the system has to know which components are to be used for generating a solution. In the case of a software system synthesis, arithmetic operations, mathematical functions, and various instructions of a programming language are such components. In the case of a digital circuit synthesis AND, OR, NOT, NAND, NOR logic gates, various flip-flops, etc. are such components. Secondly, a human designer has to define the fitness function. It seems that this is the most difficult problem. In order to define the fitness function, one has to formalize the task of a synthesized system precisely. Solving problems which belong to a certain class is, of course, the main objective of a synthesized system. Thus, systems-individuals should evolve in order to solve problems in a satisfactory way. In other words, the fitness function should define how well the system solves these problems. Thirdly, a human designer should deliver control parameters such as the size of the population, the probability of applying genetic operators, the termination condition, etc.

In genetic programming, programs are usually represented by tree or graph structures. In Fig. 5.8a the expression $-b/2a$ is represented with the help of a tree structure. All expressions of a programming language can be represented with such a tree representation. The specific form of the "chromosomes" of individuals results in the specific form of the genetic operators. A mutation is shown in Figs. 5.8a and 5.8c. A part of the first tree, which is mutated—the subtree $-b$ (it is encircled by a dashed line in Fig. 5.8a) is removed. A subtree representing the expression $-b - \sqrt{\Delta}$ (encircled by a dashed line in Fig. 5.8c) is joined to the tree in place of the removed part. A tree which represents the expression $\dfrac{-b - \sqrt{\Delta}}{2a}$ is obtained as a result of this mutation. A crossover operator consists of exchanging subtrees of trees (individuals). Let us cross a tree which has been obtained as a result of the mutation (Fig. 5.8c) and a tree which represents the expression $\dfrac{a + b}{-b + \sqrt{\Delta}}$, which is shown in Fig. 5.8b. The parts of the "chromosomes" which are exchanged are encircled by a dashed line. As a result of the crossover we obtain the tree shown in Fig. 5.8d, which represents the expression $\dfrac{a + b}{-b - \sqrt{\Delta}}$ (it is not interesting), and the tree shown in Fig. 5.8e, which represents the expression $\dfrac{-b + \sqrt{\Delta}}{2a}$. This second expression has a well-known mathematical interpretation, as does the initial expression $-b/2a$ shown in Fig. 5.8a.

Let us notice that we have to formulate the *well-defined* fitness function in order to generate (with genetic operators) a system which solves a certain class of problems. The fitness function directs the actions of the genetic operators. In genetic program-

[25] If we analyze the applications of genetic programming, it seems that the synthesis of systems of electronics or automatic control is easier than the synthesis of software systems.

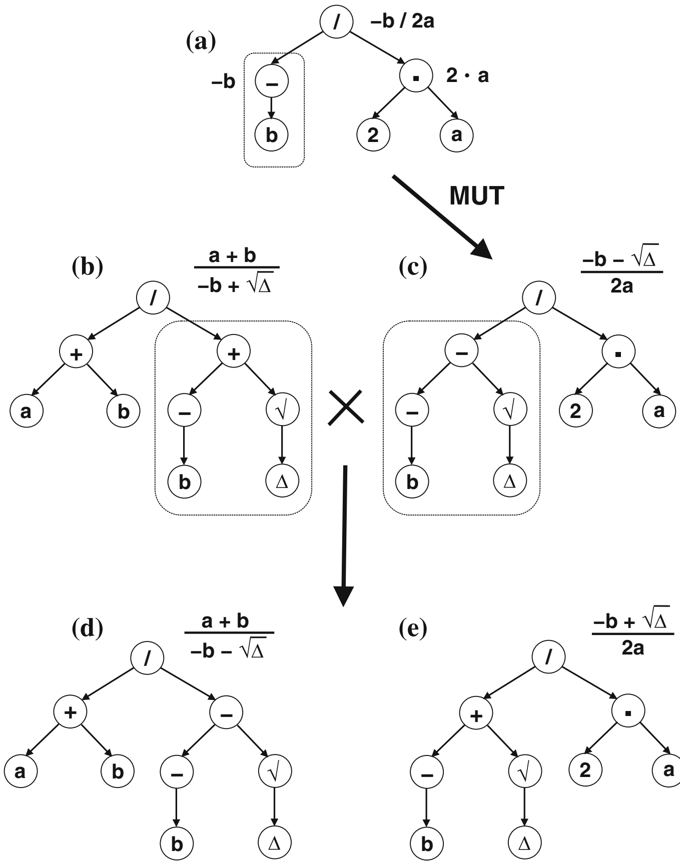

Fig. 5.8 **a** An exemplary expression of a programming language which is coded as a tree structure and its mutation, **b**, **c** the crossover of two structures, **d**, **e** descendant structures which are results of the crossover

ming crossing reasonable solutions is a basic operator. The mutation operator plays an auxiliary role.

On the basis of genetic programming a very interesting approach, called *meta-genetic programming*, was defined in 1987 by Jürgen Schmidhuber of the Technical University of Munich [265]. In this approach both chromosomes and genetic operators evolve themselves, instead of being determined by a human designer. Thus, a meta-genetic programming system evolves itself with the help of genetic programming. A similar approach was used for constructing the *Eurisko* system by Douglas Lenat[26] in 1976. The system, which is based on heuristic rules, also contains meta-rules allowing it to change these heuristic rules.

[26]Douglas Lenat—a professor of computer science at Stanford University and Carnegie-Mellon University. A president of Cycorp, Inc., which researches the construction of a common-sense knowledge base (ontology) *Cyc*.

5.5 Other Biology-Inspired Models

Biology-based models have been used for developing other interesting methods in Artificial Intelligence. The best known methods include *swarm intelligence*, introduced by Gerardo Beni and Jing Wang in 1989 [22], and *Artificial Immune Systems*, developed by Farmer et al. [86] in 1986.

Modeling an AI system as a self-organized population of autonomous individuals which interact with one another and with their environment is the main idea of *swarm intelligence*. An individual can take the form of an *agent*,[27] which transforms observations into actions in order to achieve a pre-specified target. It can also take the form of a *boid* introduced by Craig Reynolds in 1987 [238]. Boids cooperate in a flock according to three rules: a separation rule (keep a required distance from other boids to avoid crowding), a cohesion rule (move towards the center of mass of the flock to avoid fragmenting the flock) and an alignment rule (move in the direction of the average target of the flock). There are a lot of algorithms defined using this approach. *Ant Colony Optimization, ACO*, algorithms were proposed by Marco Dorigo in 1992 [72]. Agents are modeled as "artificial ants", which seek solutions in a solution space and lay down "pheromone trails". Pheromone values increase for promising places, whereas for places which are not visited frequently pheromones "evaporate". This results in more and more ants visiting promising areas of the solution space [298].

Particle Swarm Optimization, PSO, is a method of searching for the best solution in an n-dimensional solution space. It was introduced by Russell Eberhart and James Kennedy in 1995 [82]. A solution is searched for by a swarm of particles moving in the solution space. The swarm moves in a direction of leaders, i.e., particles having the best fitness function values. Each time a better solution is found the swarm changes its direction of motion and accelerates in this new direction. Experiments have shown that the method is resilient to problems related to local extrema.

Artificial Immune Systems, AISs [43], are mainly used for solving problems related to detecting anomalies. The idea of differentiating between normal/"own" cases and pathological/"alien" cases is based on the immune system of a biological organism. All the cases that are not "similar" to known cases are classified as anomalies. When an unknown case appears and its characteristics are similar to those recognized by one of the detectors of anomalies, it is assumed to be an "alien" and this detector is activated. The activated detector is "processed" with operators such as mutation and duplication. In such a way the system learns how to recognize pathological cases.

Bibliographical Note

A general introduction to the field can be found in [83, 65, 169, 201, 260]. Genetic algorithms are discussed in [111, 139], evolution strategies in [268], evolutionary programming in [100], and genetic programming in [172].

[27] Agent systems are discussed in Chap. 14. Therefore, we do not define them in this section.