Chapter 14
Evolutionary Algorithms

14.1 Introduction

In nature, evolution is mostly determined by natural selection of different individuals competing for resources in the environment. Those individuals that are better are more likely to survive and propagate their genetic material. The encoding for genetic information (genome) is done in a way that admits asexual reproduction, which results in offspring that are genetically identical to the parent. Sexual reproduction allows some exchange and re-ordering of chromosomes, producing offspring that contain a combination of information from each parent. This is the recombination operation, which is often referred to as crossover because of the way strands of chromosomes cross over during the exchange. The diversity in the population is achieved by mutation operation.

Usually grouped under the term *evolutionary computation* or *evolutionary algorithms* [1][4], we find the domains of genetic algorithms [9], evolution strategies [26][27], evolutionary programming [28] and genetic programming [29]. They all share a common conceptual base of simulating the evolution of individual structures via processes of selection, recombination and mutation reproduction and thereby producing better solutions. The processes depend on the perceived performance of the individual structures as defined by the problem. The procedure is then iterated as illustrated in Figure 14.1.

Darwinian evolutionary theory principles of reproduction and natural selection (survival of the fittest) are the base of the evolutionary theory:

- individuals who survive are the ones best adapted to exist in their environment due to the possession of variations;
- individuals that survive will reproduce and transmit these variations to their offspring;
- as time and generations continue, many adaptations are perpetuated in individuals until new species evolve in forms different from the common ancestor;
- traits which are beneficial to the survival of an organism in a particular environment tend to be retained and passed on, increasing in frequency within the population;

C. Grosan and A. Abraham: Intelligent Systems, ISRL 17, pp. 345–386.
springerlink.com

- trait which have low survival tend to decrease in frequency;
- when environmental conditions change, traits that were formally associated with low survival may have greater survival.

The correspondence between natural evolution and problem solving inspired by it is given in Figure 14.2.

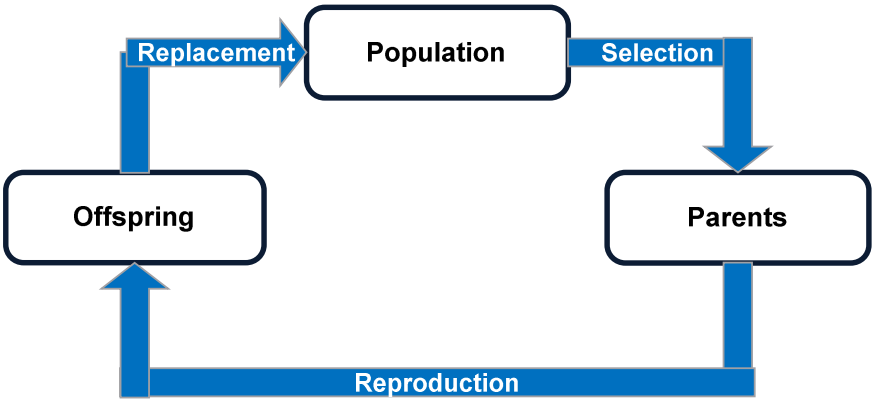

Fig. 14.1 Evolutionary scheme.

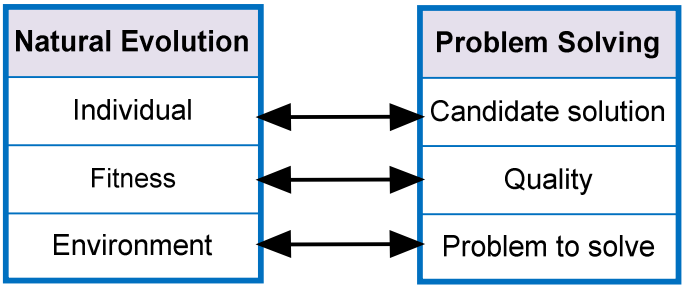

Fig. 14.2 Correspondence between natural evolution principles and problem solving.

The basic evolutionary algorithm is described below.

Evolutionary Algorithm
Step 1. Set t= 0;

Step 2. Randomly initialize the population $P(t)$.
Repeat
Step 3.1. Evaluate individuals from $P(t)$;
Step 3.2. Selection on $P(t)$
Let P'(t) be the set of selected individuals.

Step 3.3. Crossover on $P'(t)$;
Step 3.4. Mutation on $P'(t)$.
Step 3.5. Survival on P'(t).
Step 3.6. $t=t+1$.
Until *t=number of iterations*

14.2 How to Build an Evolutionary Algorithm?

In order to build an evolutionary algorithm there are a number of steps that we have to perform:

- design a representation;
- find a way to initialize the population;
- design a way of mapping a genotype to a phenotype;
- design a way of evaluating an individual;
- decide how to select individuals to be parents (design a selection mechanism);
- design suitable mutation operators;
- design suitable recombination operators;
- decide how to select individuals to be replaced;
- decide when to stop the algorithm.

There are many different variants of evolutionary algorithms, but they all share the same evolutionary structure: given a population of individuals (candidate solutions), the natural selection will influence the survival of the fittest. Based on the fitness (or an evaluation function), the best individuals (candidate solutions) will be selected to seed the population of the next generation. Recombination and/or mutation operators are applied to them. The new obtained candidate solutions (offspring) will compete with the old ones (based on their fitness) to be part of the next generation.

The process will be iterated until a candidate solution with sufficient quality is found or until the available resources are finished.

The variants of evolutionary algorithms differ only in technical details [2][4][5]:

- in the case of Genetic Algorithms (GAs) the solutions are represented as strings over a finite alphabet;
- solutions are represented as real-valued vectors in Evolutionary (Evolution) Strategies (ES);
- in the case of Evolutionary Programming (EP), solutions are finite state machines;
- Genetic Programming (GP) uses trees for solution representation.

Genetic Algorithms, Evolution Strategies, Evolutionary programming and Genetic Programming are the four main classes of Evolutionary Algorithms and will be independently presented in the following sections.

14.2.1 Designing a Representation

Representation here refers to a way of representing an individual as a genotype. An individual is also known as *chromosome* and it is composed of a set of *genes*.

Objects forming the original problem context are referred to as *phenotypes* and their encoding (which are the individuals in the evolutionary algorithm) are called *genotypes*.

Let us consider the problem of minimizing the function $(x\text{-}2)^2$, $x \in \{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5\}$.

The set of integers represents the set of phenotypes. If we use a binary code to represent them, then 1 can be seen as a phenotype and 000001 as the genotype representing it (in this case we used a string of size 6 to represent it). The phenotype space can be different from the genotype space but the whole search process takes place in the genotype space.

There are many ways to design a representation and the way we choose must be relevant to the problem that we are solving. When choosing a representation, we have to bear in mind how the genotypes will be evaluated and what the genetic operators might be. For one problem, there can be multiple representations which can fit. We should also be careful to select the most efficient one (in terms of resources consuming).

There are a few standard representations:

- binary representation;
- real representation;
- order based representation (permutations);
- tree representation.

14.2.2 Initializing the Population

A *population* is a set of genotypes (some of the population's members can have multiple copies). Initializing the population means specifying the number of individuals in it (which is given by population size).

Each member of the population represents a possible solution for the problem to solve. The *diversity* of a population is measured as the number of different solutions of that population. During the search process, the algorithm should be able to preserve diversity in the population. If the problem has multiple solutions (for example, the n-queens problem) then the final population should contain as many different solutions as possible.

If the problem has just a single solution, then the final population's individuals should be as similar among them as possible.

The population should be initialized (if possible) uniformly on the search space.

Each individual of the population is (unless otherwise mentioned) randomly initialized (defined) over the given domain (or search space).

14.2.3 Evaluating an Individual

In order to evaluate an individual, a *fitness function* – also called *evaluation function* or *quality function* – is employed.

The role of the evaluation function represents the requirements to adapt to [2]. Evaluation function is required for the selection process.

From the problem solving perspective it represents the task to solve in evolutionary context. This function actually assigns a quality measure to the genotypes.

For our example – minimization of $(x\text{-}2)^2$ – the fitness of the genotype 000001 is $(1\text{-}2)^2 = 1$.

Example

Let us consider the 8-Queens problem. The problem requires arranging 8 queens on an 8×8 chess board such as there will be no attacks among the queens.

The representation used is order-based representation, which is a vector of size 8 containing a permutation of the numbers 1 to 8 (whose meaning is: the value of position i in the vector represents the column of queen in line i).

If our representation is (graphically depicted in Figure 14.3):

(2 4 1 6 7 8 3 5)

then the fitness function's value can be calculated as:0+3+0+3+3+3+1+0=13 because first queen does not attack any other queen, second queen attacks three other queens on the diagonal, third queen attacks no queens and so on.

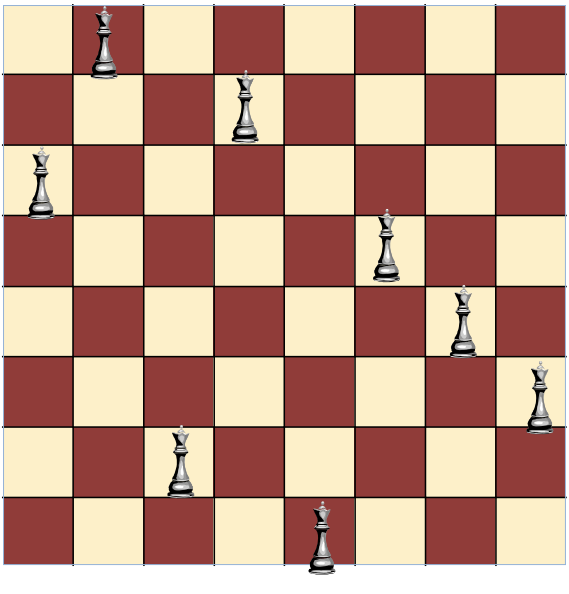

Fig. 14.3 Graphical representation of the solution (2 4 1 6 7 8 3 5) for the 8-queens problem.

14.2.4 Selection Mechanism

The role of selection mechanism is to distinguish among individuals based on their fitness in order to allow better individuals to become parents of the next generation. In evolutionary computation, parent selection is probabilistic: high quality individuals get higher chance to become parents. Low quality individual are also given a low change to become parents in order to avoid getting stuck in a local optima. There are several selection mechanisms, which will be discussed in detail in the following sections.

Remark
If, in the beginning of the evolutionary algorithms, the highest chances were given to best individuals, nowadays approaches also give chances to individuals, which do not have a high quality.

14.2.5 Designing Suitable Variation Operators

The role of variation operators is to create new individuals from the existing ones. The new created individuals should represent candidate solutions in the corresponding phenotype space. There are several known variation operators, which can be applied to individuals of an evolutionary algorithm population, but two of them are the most important and widely used:

- mutation operator and
- recombination or crossover operator.

14.2.5.1 Mutation Operator

Mutation operator is a unary operator (applied to one individual only) and it usually affects (or slightly modify) one genotype. The child obtain by applying mutation operator to his parent is only slightly different from it. Mutation can affect one or multiple alleles depending on the size of the candidate solution. The affected alleles will be chosen in a random manner.It has been proved that, given sufficient time, an evolutionary algorithm can reach the global optimum relying on the propriety that each genotype representing a possible solution can be reached by variation operators [2][3].

14.2.5.2 Crossover (Recombination) Operator

Crossover or recombination operator is a binary operator (there are some rare situations where more than two individuals are combined; this is sound mathematically but it has no biologically correspondence). Crossover merges information from two parent genotypes into one or two offspring genotypes. The operator chooses what parts of each parent are combined and the way in which these are combined in a random manner. Recombination operator is never used in evolutionary programming algorithms. The reason behind recombination is that by

mating two individuals with different features the offspring will combine both of these features (which is very successful in nature, especially for plants and livestock).

Variation operators are representation dependant: for various representations different variation operators have to be designed [2].

14.2.6 Designing a Replacement Scheme

Survivor selection mechanism or replacement has the role to distinguish among individuals, which will be kept for the next generation based on their quality. Since the size of the population is usually kept constant during the evolution process, there should exist a way to select among the existing and new obtained (by applying variation operators) candidate solutions.

The survivor selection process takes place after having created the offspring of the selected parents. This selection is based on the candidate solutions' quality (or fitness).

14.2.7 Designing a Way to Stop the Algorithm

There are two main cases of a suitable termination condition or stopping criterion for an evolutionary algorithm [2]:

(i) If the problem has a known optimal fitness level (for example, the optimum of an optimization problem is known, or the value of the fitness function for the expected solution is known) then reaching this level (with a given sufficiently small positive precision $\varepsilon > 0$) should be used as termination condition.

(ii) If the problem to solve doest not has a known optimum then the previous stopping condition cannot be used. In this case the termination condition may be one of the following:

a. a given number of generations is reached;
b. a given number of fitness evaluations is reached;
c. the available resources are overloaded (the maximally allowed CPU time elapses);
d. there is no improvement for the fitness function for a given number of consecutive generations or fitness evaluations;
e. the population diversity drops under a given threshold.

In these situations the termination condition may consists of two criteria: either the optimum was reach or one of the conditions above was satisfied.

14.3 Genetic Algorithms

The most widely known type of evolutionary algorithms and probable the most used are genetic algorithms. Simple to implement and use, genetic algorithms are

one of the most suitable techniques for any kind of optimization problem. The chapter presents the most important ways to design a representation, selection mechanisms, variation operators – recombination and mutation, survival schemes and population models.

14.3.1 Representing the Individuals

Representation of individuals is one of the most important steps in designing a genetic algorithm. Representation should be adequate for the problem to solve and should consume as less resources as possible. For one problem there can be multiple possible representations; thus, we have to be able to decide which one is more adequate to satisfy our needs and requirements.

There are some standard types of representing an individual, which have the same form for different problems but might have a different interpretation. Some of the most important ones are presented in what follows:

- binary representation;
- integer representation;
- real valued or floating- point representation;
- order based or permutation representation.

14.3.1.1 Binary Representation

Probable the most common type of representations used by the evolutionary algorithms is the binary representation. An example of a binary individual (chromosome) of size 8 is depicted in Figure 14.4.

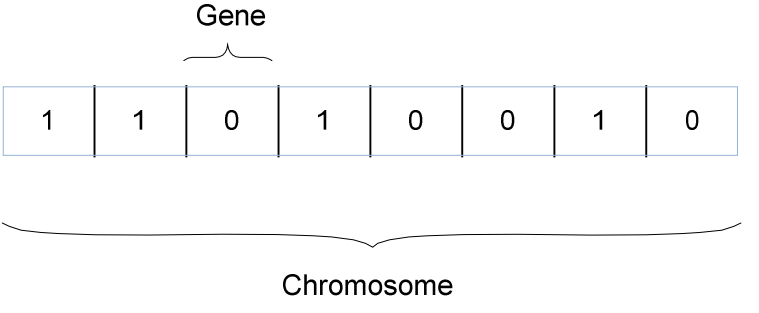

Fig. 14.4 Binary representation.

A binary individual can have different meanings depending on the problem it is used for.

Some of the possible meanings for certain problems are as follows (but the list is not limited to this):

- A real (or integer) number represented over the alphabet 2 (or using the base 2). In our example we have the number:

$1*2^7+1*2^6+0*2^5+1*2^4+0*2^3+0*2^2+1*2^1+0 = 128+64+0+16+0+0+2+0=210$

This integer number can be scaled to any interval or can be transformed into a real number in any given interval.

For instance, if we wish to have the real representation within [-1, 1] of the binary number in Figure 14.3, this is given by (we take into account that our binary number is a real number between 0 and $2^8 = 256$):

$$-1+\frac{210}{256}(1-(-1))=0.64$$

Note. We use here the formula of transforming a number x from the interval [0, *Max*] to the interval [min, max] which is (*Max* and max are different):

$$\min+\frac{x}{Max}(\max-\min)$$

- A solution for the 0-1 knapsack problem: the values of 1 represent the selected items and the values of 0 the items which are not selected. For instance, the chromosome in Figure 14.2 will select the items 1, 2, 4 and 7.
- A solution for the graph partitioning problem (partition of the nodes into two sets with various proprieties): the values of zero represent the nodes belonging to the first set and the values of one represent the nodes belonging to the second set. In our example in Figure 14.3, nodes 3, 5, 6 and 8 belong to the first set while the nodes 1, 2, 4 and 7 belong to the second set.

14.3.1.2 Real Representation

Real representation is mostly used for real function optimization. The size of the chromosome will be equal to the number of dimensions (variables).

An example of real representation for n variables is given in Figure 14.5.

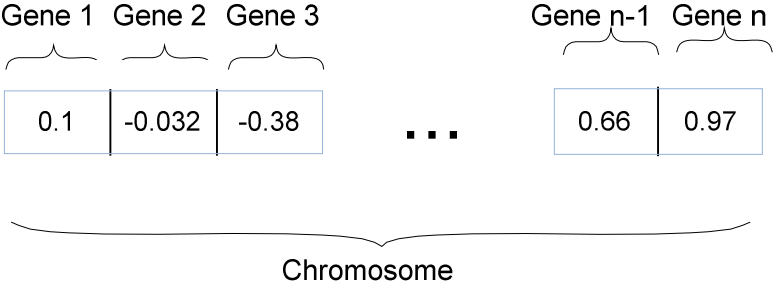

Fig. 14.5 Real representation.

14.3.1.3 Integer Representation

Sometimes real or binary representation is not suitable for a certain problem. For instance, in the case of map coloring problem[1], none of these representations might be used. For this problem the representation is a string of integers, having the size equal to the number of countries and taking values form {1, 2, …, *k*}.

An example of integer representation for the map-coloring problem with 10 countries and 6 available colors (denoted {1, 2, 3, 4, 5, 6}) is given in Figure 14.6 (it can be observed that only 5 of the 6 available colors are used for encoding the individual). In the case of integer representation, the individual is a string of integers whose values can be restricted to a given domain or set of possible values or unrestricted.

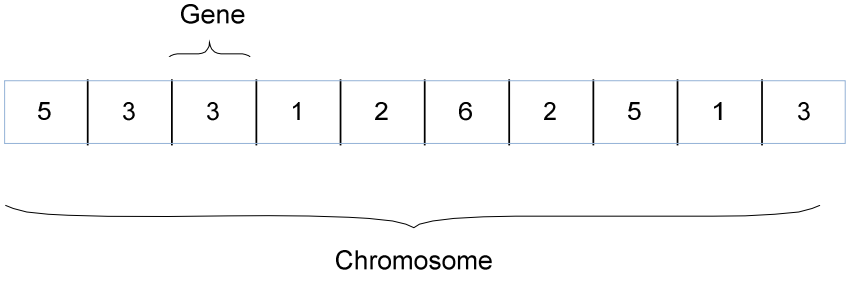

Fig. 14.6 Integer representation.

14.3.1.4 Order-Based Representation

In this case, the individuals are represented as permutations. This kind of representation is mostly used for ordering/sequencing problems. An example of chromosome using order-based representation is given in Figure 14.7.

Some famous example of problems which use permutation representation are:

- Traveling Salesman Problem (TSP): in this problem, every city gets assigned a unique number from 1 to *n.* A solution is a permutation of the numbers 1, 2, …, *n* representing the order in which the salesman visits the cities.
- *n*-Queens problem: a solution is a permutation of size n. The indices represent the queen on each row and the values represent the column each queen belongs to (we know that we cannot have two queens on the same row or column and by using this representation this is avoided; the only remaining attacks should be checked on the diagonals)
- Quadratic Assignment Problem;

[1] The map coloring problem we refer to states as follows: given a set of *n* countries and a set of *k* available colors, color each country with one of the *k* colors such as no neighboring countries will have the same color and the number of colors used is minimal (we consider the general case with *k* colors but it has been proven that 4 colors are enough to color any map).

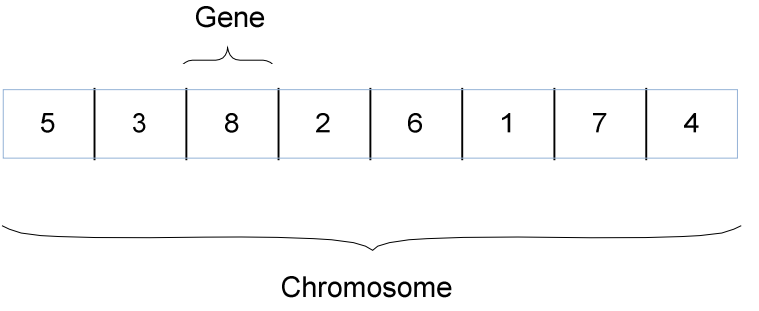

Fig. 14.7 Order-based representation.

14.3.2 Initializing the Population

Population initialization is the next step in developing a genetic algorithm once the representation has been decided. Initialization means (randomly) seeding a set with a given number of individuals having the chosen encoding/representation. While seeding the population, there are chances that some of the individuals might have multiple copies (the population is a multiset):

- For binary representation: an individual is a string of {0, 1}. The genes may be initialized with the values 0 or 1 with the probability 0.5.
- For real representation: an individual is a vector of real numbers. The genes can be initialized with random real values within the given domain (if the domain is finite), or they can be initialized using a distribution: Gaussian, Cauchy, etc.
- For order based representation: the individual is a permutation. Suppose the permutation has the size n. Each gene i will be initialized with a value between $\{1, \ldots, n\}$ which does not occur on the previous i-1 already initialized positions.
- For tree based representation: the individual is a tree. We have a set of terminals and a set of functions. Each node of the tree is randomly initialized:
    - if the node is a terminal then a random value from the terminals set will be taken (certain terminals can be used multiple times while others might not be used at all);
    - if the node is not a terminal, then a random function from the set of functions will be selected.

Examples

Binary representation: vector of size 10

0 1 1 0 0 0 1 1 0 1

Real representation: vector of size 10 on the domain [0, 1]

{0.004, 0.92, 0.34, 0.11, 0.26. 0.63, 0.0001, 0.019, 0.82, 0.0056}

Order based representation: vector of size 10

4 9 3 1 6 2 8 10 5 7

14.3.3 Selection Mechanisms

Selection process in a genetic algorithm occurs when parents who will be further used for crossover are to be selected. In a standard way, high priority and chances are given to fittest individuals.

Some of the most used selection mechanisms are:

- tournament selection;
- fitness proportional selection;
- roulette wheel selection;
- rank based selection.

14.3.3.1 Tournament Selection

Tournament selection is one of the simplest selection schemes. It is suitable when the population size is very large and it is not practical to compare or rank all the individuals at a time. Thus, this selection does not require any global knowledge of the population.

It relies on an ordering relation that can rank any two individuals.

There are two known versions of the tournament selection:

(i) binary tournament
(ii) *k*-tournir (or *k*-tournament).

In the case of *binary tournament*, two individuals are randomly selected from the population and the best among them (in terms of fitness value) is kept in a separate set. The procedure is repeated until the number of selected individuals equals the required number of individuals, which are to be selected.

The *k - tournament* is a generalization of the binary tournament in the sense that k individuals are randomly selected from the population and the best individual among all of them is kept in a separate set. The process is then again repeated until the required number of parents is selected from the whole population.

The probability that an individual will be selected as a result of a tournament depends on four factors [2][6][7][8]:

(i) its *rank in the population*: this is estimated without the need for sorting the whole population;
(ii) *the tournament size k*: the larger the tournament the more chance that it will contain members whose fitness is above average and the less that it will consist entirely of low-fitness members;

(iii) the *probability* that the most fit member of the tournament is selected; usually this probability is 1 (deterministic tournament) but there are stochastic versions which use a probability less than 1;

(iv) whether the individuals are chosen *with or without replacement*. In the second case, with deterministic tournaments, the k-1 least fit members of the population can never be selected. If the tournament candidates are picked with replacement, it is always possible for even the least-fit member of the population to be selected.

14.3.3.2 Fitness Proportional Selection

This selection mechanism was introduced in [9] (see also [10]). Let us denote by f_i the fitness of the i-th individual in the population and by N the population size (number of individuals in the population). Thus, the probability p_i that individual i is selected for mating (for recombination) is given by:

$$p_i = \frac{f_i}{\sum_{i=1}^{N} f_i}$$

This means that the selection probability depends on the absolute fitness value of the individual compared to the absolute fitness values of the rest of the population.

There are some problems with this selection mechanism [2]:

- individuals that are much better than the rest take over the entire population very quickly and this leads to *premature convergence*;
- when fitness values are very close there is almost no selection pressure;
- the mechanism behaves differently on transposed versions of the same fitness function.

14.3.3.3 Roulette Wheel Selection

The roulette-wheel selection is also called *stochastic sampling with replacement* and has been introduced in [11]. This is a stochastic algorithm and involves the following technique: the individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness. A random number is generated and the individual whose segment spans the random number is selected. The process is repeated until the desired number of individuals is obtained. This technique is analogous to a roulette wheel with each slice proportional in size to the fitness. Figure 14.8 shows an example containing 9 individuals for which the fitness value and the selection probability is displayed. Individual 1 is the fittest individual and occupies the largest interval, whereas individual 9 as the least fit individual has the smallest interval on the line.

Number of individual	1	2	3	4	5	6	7	8	9
Fitness value	5.0	4.5	4.0	3.5	3.0	2.5	2.0	1.5	1.0
Selection probability	0.19	0.17	0.15	0.13	0.11	0.09	0.07	0.05	0.04

Fig. 14.8 Example of individuals, their fitness values and the corresponding selection probability.

The line segment and the roulette wheel corresponding to this example are shown in Figures 14.9 and 14.10.

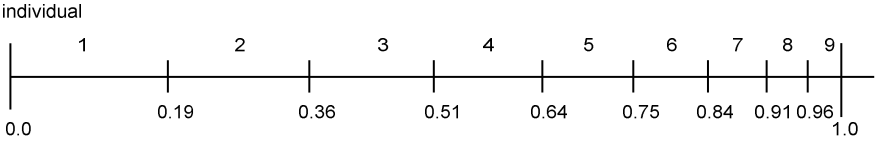

Fig. 14.9 Line segment corresponding to the example.

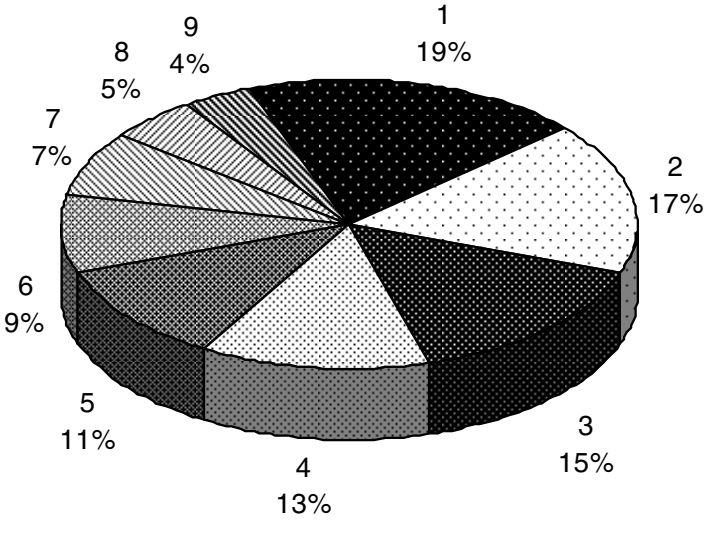

Fig. 14.10 Roulette wheel corresponding to the example.

For selecting the mating population the appropriate number of uniformly distributed random numbers (uniform distributed between 0.0 and 1.0) is independently generated.

For example, if the following 5 random numbers are generated:

```
0.61, 0.33, 0.95, 0.11, 0.45
```

the resulting selected individuals at each trial can be observed in Figure 14.11 which shows the selection process of the individuals for the example in table together with the above sample trials.

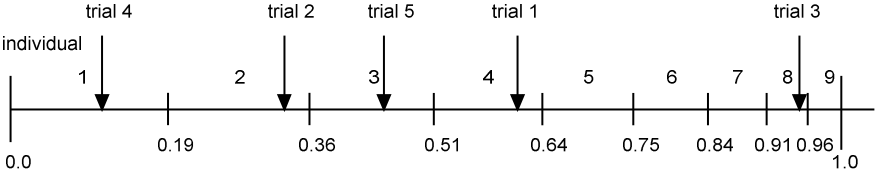

Fig. 14.11 Selection process of individuals for the example.

After selection the chosen individuals are:

```
4, 2, 8, 1, 3.
```

The roulette-wheel selection algorithm provides a zero bias but does not guarantee minimum spread.

14.3.3.4 Stochastic Universal Sampling

Stochastic universal sampling [11] provides zero bias and minimum spread. The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness exactly as in roulette-wheel selection.

Equally spaced pointers are placed over the line as many as there are individuals to be selected. Consider *NP* the number of individuals to be selected, then the distance between the pointers is 1/*NP* and the position of the first pointer is given by a randomly generated number in the range [0, 1/*NP*].

For 5 individuals to be selected, the distance between the pointers is 1/5=0.2. Figure 14.12 shows the selection for the above example with the random starting point 0.1.

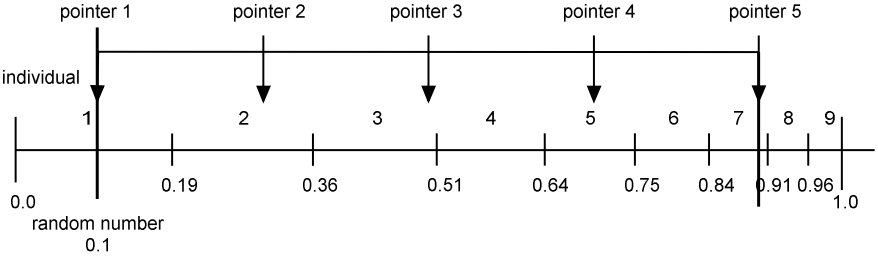

Fig. 14.12 Stochastic universal sampling.

After selection the mating population consists of the individuals:

```
1, 2, 3, 5, 7.
```

Stochastic universal sampling ensures a selection of offspring which is closer to what is deserved then roulette wheel selection.

14.3.3.5 Rank Based Selection

In rank-based fitness assignment, the population is sorted according to the fitness (quality) values. The fitness assigned to each individual depends only on its position in the individual's rank and not on the actual fitness value.

Rank-based fitness assignment overcomes the scaling problems of the proportional fitness assignment. (Stagnation in the case where the selective pressure is too small or premature convergence where selection has caused the search to narrow down too quickly.) The reproductive range is limited, so that no individuals generate an excessive number of offspring. Ranking introduces a uniform scaling across the population and provides a simple and effective way of controlling selective pressure [12]. Rank-based fitness assignment behaves in a more robust manner than proportional fitness assignment and, thus, is the method of choice [12][13][14].

Linear Ranking

Consider N the number of individuals in the population, i the position of an individual in this population (least fit individual has i =1, the fittest individual i =N) and SP the selective pressure. The fitness value for an individual is calculated as:

$$fitness(i) = 2 - SP + 2 \cdot (SP - 1) \cdot \frac{(i-1)}{(N-1)}$$

Linear ranking allows values of selective pressure in [1.0, 2.0].

Non-linear ranking

A new method for ranking using a non-linear distribution was introduced in [15]. The use of non-linear ranking permits higher selective pressures than the linear ranking method.

$$fitness(i) = \frac{N \cdot X^{i-1}}{\sum_{i=1}^{N} X^{i-1}}$$

X is computed as the root of the polynomial:

$$0 = (SP - N) \cdot X^{N-1} + SP \cdot X^{N-2} + \ldots + SP \cdot X + SP\,.$$

Non-linear ranking allows values of selective pressure in the interval [1, N - 2].

14.3.3.6 Local Selection

In local selection every individual resides inside a constrained environment called the local neighborhood. In the other selection methods the whole population or subpopulation is the selection pool or neighborhood. Individuals interact only with individuals inside this region. The neighborhood is defined by the structure in which the population is distributed. The neighborhood can be seen as the group of potential mating partners. The first step is the selection of the first half of the mating population uniform at random (or using one of the other mentioned selection algorithms, for example, stochastic universal sampling or truncation selection). Now a local neighborhood is defined for every selected individual. Inside this neighborhood the mating partner is selected (best, fitness proportional, or uniform at random) [12].

The structure of the neighborhood can be:

- linear:
  - full ring
  - half ring (see Figure 14.13)
- two-dimensional
  - full cross
  - half cross (see Figure 14.14, top)
  - full star
  - half star (see Figure 14.14, bottom)
- three-dimensional and more complex with any combination of the above structures [12].

The distance between possible neighbors together with the structure determines the size of the neighborhood.

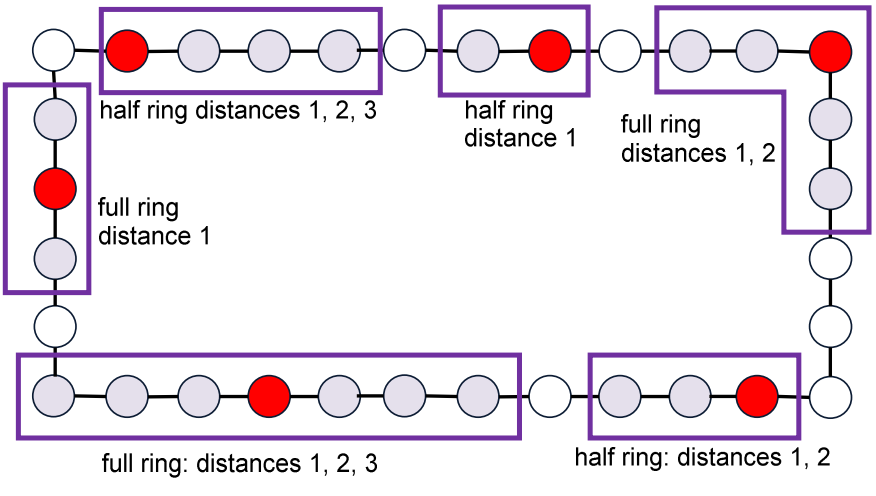

Fig. 14.13 Linear neighborhood: full and half rings.

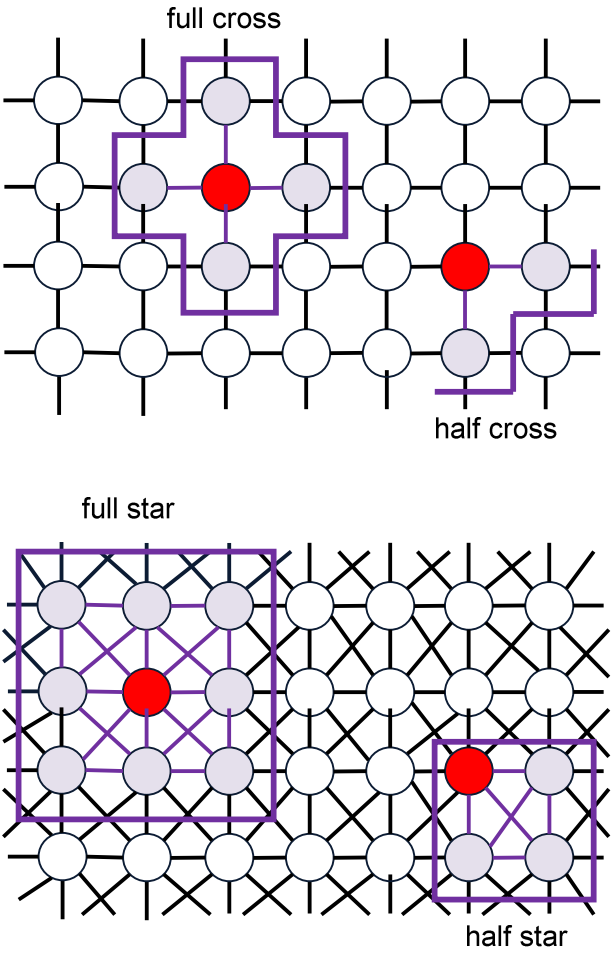

Fig. 14.14 Two-dimensional neighborhood: full and half cross (top); full and half star (bottom).

Between individuals of a population isolation by distance exists. The smaller the neighborhood, the bigger the isolation distances. However, because of overlapping neighborhoods, propagation of new variants takes place. This assures the exchange of information between all individuals.

The size of the neighborhood determines the speed of propagation of information between the individuals of a population, thus deciding between rapid propagation and maintenance of a high diversity/variability in the population. A higher variability is often desired, thus preventing problems such as premature convergence to a local minimum. Local selection in a small neighborhood performed

better than local selection in a bigger neighborhood. Nevertheless, the interconnection of the whole population must still be provided. Two-dimensional neighborhood with structure half star using a distance of 1 is recommended for local selection. However, if the population is bigger (>100 individuals) a greater distance and/or another two-dimensional neighborhood should be used [12].

14.3.4 Variation Operators

The main variation operators are recombination or crossover and mutation. Each of them has specific forms for different individual representations and will be presented in what follows.

14.3.4.1 Crossover or Recombination

The role of recombination operator is to produces new individuals by combining the information contained in two or more parents. This is done by combining the variable values of the parents. Depending on the representation of the variables different methods must be used.

14.3.4.1.1 Recombination for Binary Representation

This section describes recombination methods for individuals with binary variables. During the recombination of binary variables only parts of the individuals are exchanged between the individuals. Depending on the number of parts, the individuals are divided before the exchange of variables (the number of cross points).

Single point crossover

Let us denote by *nrvar* the length of the binary string used to encode an individual (the number of variables). In single-point crossover [9] one crossover position (cutting point) $k \in [1, 2, ..., nrvar\text{-}1]$, is selected uniformly at random. Two new offspring are produced by exchanging variables between the individuals about this point.

Consider the following two individuals of size (length) 10:

```
parent 1     1  1  1  0  0  0  1  1  0  1
parent 2     1  0  0  0  1  1  0  1  1  0
```

and the chosen crossover position 4.

After crossover the two new individuals created are (see Figure 14.15):

```
offspring 1      1  1  1  0  1  1  0  1  1  0
offspring 2      1  0  0  0  0  0  1  1  0  1
```

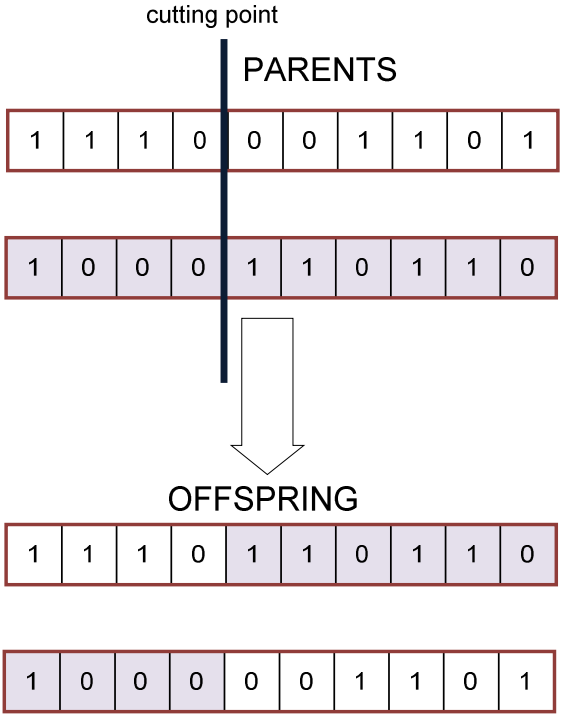

Fig. 14.15 Single point crossover

Double point / multi point crossover

In the case of double-point crossover two crossover positions are selected uniformly at random and the variables are exchanged between the individuals between these points. Two new offspring are produced. An example of double point crossover is shown in Figure 14.16.

Single-point and double-point crossover are special cases of the general method multi-point crossover.

For multi-point crossover, m crossover positions $k_i \in [1, 2, ..., nrvar-1]$, $i=1,\ldots,m$, are chosen at random with no duplicates and sorted into ascending order. Then, the variables between successive crossover points are exchanged between the two parents to produce two new offspring.

Consider the following two individuals with 15 binary variables each:

```
parent 1    1 1 | 1 1 1 | 1 1 1 | 1 1 1 | 1 1 | 1 1 1
parent 2    0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 | 0 0 0
```

We choose 5 crossover points; the chosen crossover positions are 2, 5, 8, 11, 13.

After crossover the two new individuals created are (see Figure 14.17):

```
offspring 1     1  1 | 0  0  0 | 1  1  1 | 0  0  0 | 1  1 | 0  0
offspring 2     0  0 | 1  1  1 | 0  0  0 | 1  1  1 | 0  0 | 1  1
```

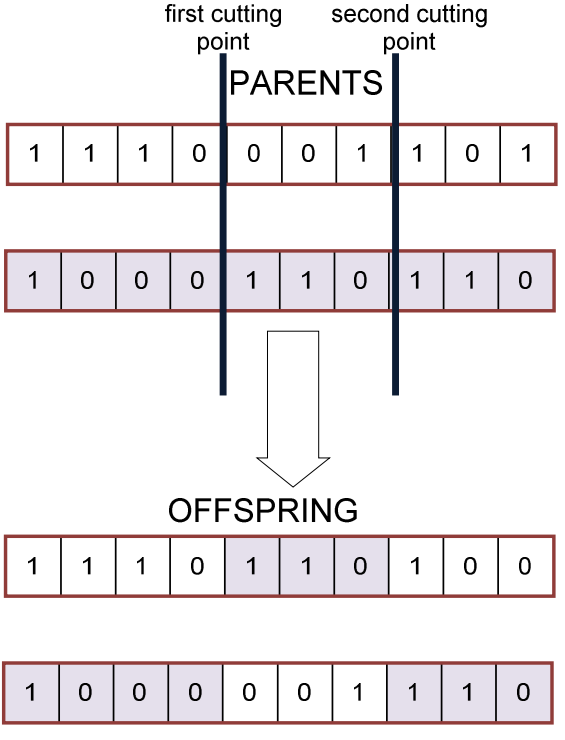

Fig. 14.16 Double point crossover.

Uniform crossover

Uniform crossover [16] generalizes the multi point crossover to make every locus a potential crossover point. For each variable the parent who contributes its variable to the offspring is chosen randomly with equal probability. Uniform crossover works by treating each gene independently and making a random choice as to which parent it should be inherited.

This is implemented by generating a string of random variables from a uniform distribution over [0, 1] whose size is equal to the size of an individual from the population. In each position, if the value is below a parameter p (usually 0.5), the gene is inherited from the first parent; otherwise from the second. The second offspring is created using the inverse mapping.

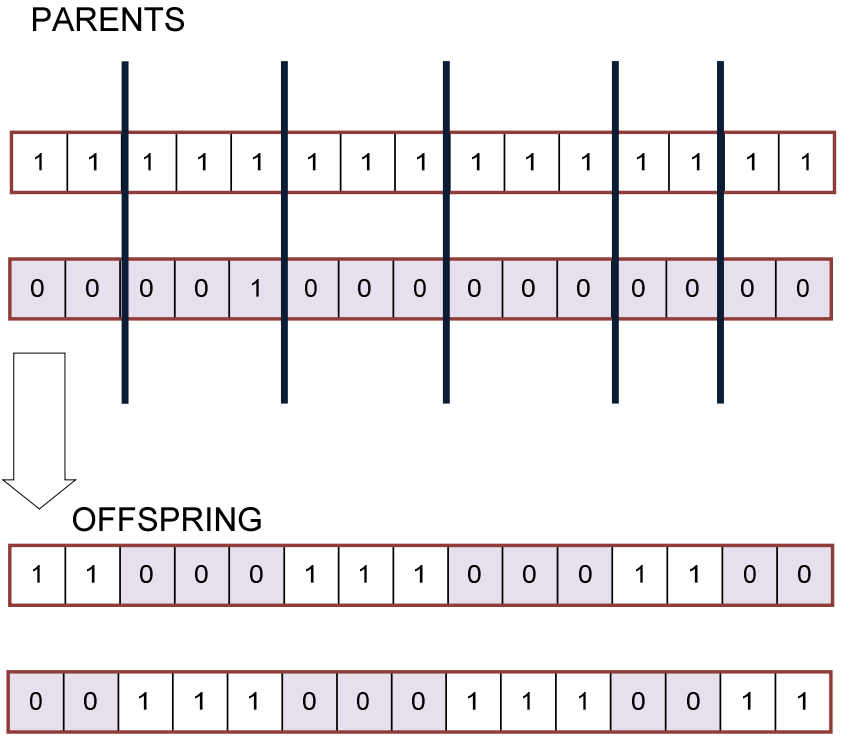

Fig. 14.17 Multi point crossover (5 cutting points on the positions 2, 5, 8, 11 and 13).

If we have the parents of size 10:

```
parent 1      1  1  1  1  1  1  1  1  1  1
parent 2      0  0  0  0  0  0  0  0  0  0
```

and the string of random variables:

```
[0.17, 0.80, 0.33, 0.45, 0.51, 0.97, 0.12, 0.66, 0.73, 0.23]
```

then the offspring obtained after crossover are:

```
offspring 1      1  0  1  1  0  0  1  0  0  1
offspring 2      0  1  0  0  1  1  0  1  1  0
```

Uniform crossover, like multi-point crossover, has been claimed to reduce the bias associated with the length of the binary representation used and the particular coding for a given parameter set. This helps to overcome the bias in single-point crossover towards short substrings without requiring precise understanding of the

significance of the individual bits in the individual's representation. In [17] it is demonstrated how uniform crossover may be parameterized by applying a probability to the swapping of bits. This extra parameter can be used to control the amount of disruption during recombination without introducing a bias towards the length of the representation used [12].

14.3.4.1.2 Recombination for Real Representation

There are two ways to perform recombination for real values representation [2]:

(i) Using an operator similar to the one used for binary representation. This has the disadvantage that only mutation can insert new values in the population since the recombination only gives new combinations of the existing floats. Recombination operators of this type are known as discrete recombination.

(ii) Using an operator that, in each gene position, creates a new value in the offspring that lies between those of the parents. If we have the parents x and y and the offspring z, then we have $z_i = \alpha x_i + (1-\alpha)y_i$, $\alpha \in [0, 1]$. Operators for this type are known as arithmetic or intermediate recombination.

Arithmetic recombination

There are three types of arithmetic recombination [2][18]: simple, single arithmetic and whole arithmetic. The choice of parameter α is made at random between [0, 1] but in practice it is common to use the value 0.5 for it (in this case we have *uniform arithmetic recombination*).

Simple recombination

In this case a crossover position k is randomly selected between {1, 2, …, *nrvar*-1}. For the first child, the first k floats of the first parent are taken. The rest is the arithmetic average of parent 1 and 2. Child 2 is analogue with the parents reversed.

$$\text{parent 1: } x_1,\ x_2,\ \ldots,\ x_k,\ x_{k+1}\ \ldots,\ x_{nrvar}$$
$$\text{parent 2: } y_1,\ y_2,\ \ldots,\ y_k,\ y_{k+1}\ \ldots,\ y_{nrvar}$$

$$\text{offspring 1: } x_1,\ x_2,\ \ldots,\ x_k,\ \alpha \cdot x_{k+1} + (1-\alpha)\cdot y_{k+1}\ \ldots,\ \alpha \cdot x_{nrvar} + (1-\alpha)\cdot y_{nrvar}$$
$$\text{offspring 2: } y_1,\ y_2,\ \ldots,\ y_k,\ \alpha \cdot y_{k+1} + (1-\alpha)\cdot x_{k+1}\ \ldots,\ \alpha \cdot y_{nrvar} + (1-\alpha)\cdot x_{nrvar}$$

An example of simple recombination is shown in Figure 14.18. The value of k is 5 and the value of α is 0.5. The size of the individual (chromosomes) is 8.

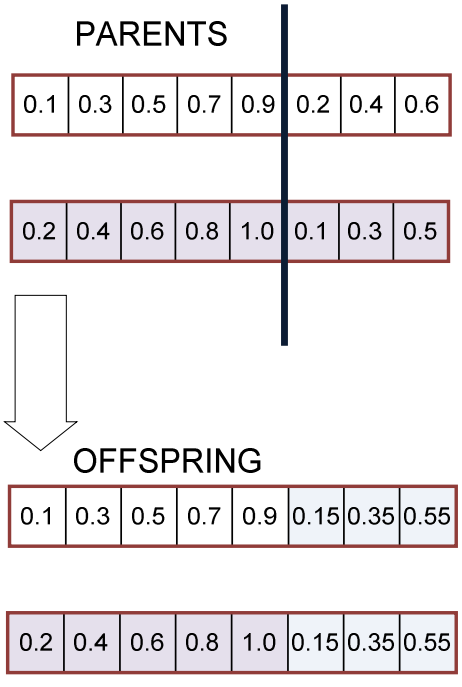

Fig. 14.18 Simple arithmetic recombination.

Single arithmetic recombination

In this case one gene k of the chromosome is picked at random. At that position, the arithmetic average of the two parents is taken. Rest of the chromosome remains the same. The second child is created in the same way with the parents reversed.

$$\begin{aligned}&\text{parent 1: } x_1,\ x_2,\ \ldots,\ x_k,\ \ldots,\ x_{nrvar}\\&\text{parent 2: } y_1,\ y_2,\ \ldots,\ y_k,\ \ldots,\ y_{nrvar}\end{aligned}$$

$$\begin{aligned}&\text{offspring 1: } x_1,\ x_2,\ \ldots,\ \alpha\cdot x_k+(1-\alpha)\cdot y_k\ \ldots,\ x_{nrvar}\\&\text{offspring 2: } y_1,\ y_2,\ \ldots,\ \alpha\cdot y_k+(1-\alpha)\cdot x_k\ \ldots,\ y_{nrvar}\end{aligned}$$

An example for $k = 5$ and $\alpha = 0.5$ is shown in Figure 14.19.

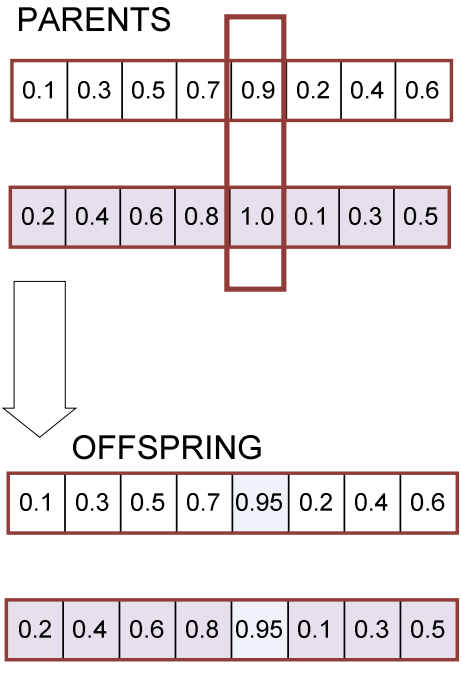

Fig. 14.19 Single arithmetic crossover.

Whole arithmetic recombination

This is the most common used recombination operator for real representation and works by taking the weighted average sum of the two parents for each gene:

parent 1: $x_1, x_2, \ldots, x_k, \ldots, x_{nrvar}$
parent 2: $y_1, y_2, \ldots, y_k, \ldots, y_{nrvar}$

offspring 1: $\alpha \cdot x_1 + (1-\alpha) \cdot y_1, \alpha \cdot x_2 + (1-\alpha) \cdot y_2, \ldots, \alpha \cdot x_{nrvar} + (1-\alpha) \cdot y_{nrvar}$ offspring 2: $\alpha \cdot y_1 + (1-\alpha) \cdot x_1, \alpha \cdot y_2 + (1-\alpha) \cdot x_2, \ldots, \alpha \cdot y_{nrvar} + (1-\alpha) \cdot x_{nrvar}$

An example for $\alpha = 0.5$ is shown in Figure 14.20.

14.3.4.1.3 Recombination for Order-Based Representation

For order based representation it is difficult to apply any of the operators discussed above suitable for binary and real value encodings due to the fact that the new individuals obtained will not remain a permutation. There are some specific recombination operators for permutations, which will be discussed in what follows.

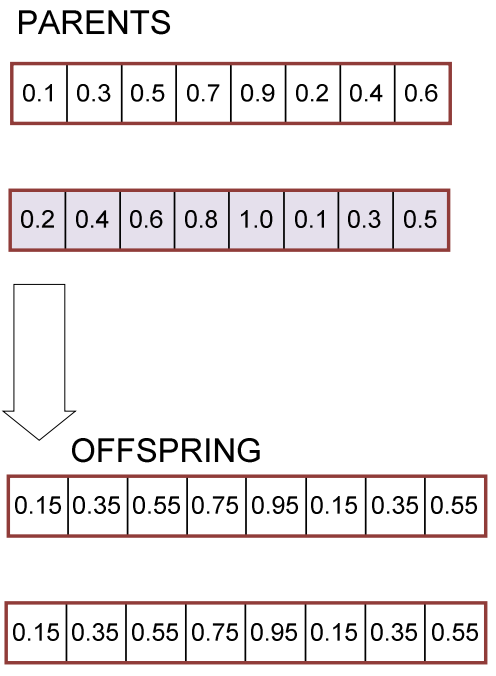

Fig. 14.20 Whole arithmetic recombination.

Partially mapped crossover

Partially mapped crossover (PMX) has been proposed in [19] as a recombination operator for the TSP. There exist now several variants of it. The steps of PMX are as follows [2][20]:

1) Choose two crossover points at random and copy the segment between them from the first parent into the first offspring.
2) Starting from the first crossover point, look for elements in that segment of the second parents that have not been copied.
3) For each i of these, look in the offspring to see what element (j) has been copied in its place from the first parent.
4) Place i into the position occupied by j in the second parent.
5) If the place occupied by j in the second parent has already been filled in the offspring by another element k, put i in the position occupied by k in the second parent.
6) Once the elements from the crossover segment have been dealt with, the rest of the offspring can be filled from the second parent.
7) The second offspring is created in a similar manner with the parents reversed.

A graphical illustration of the PMX operator is presented in Figure 14.21.

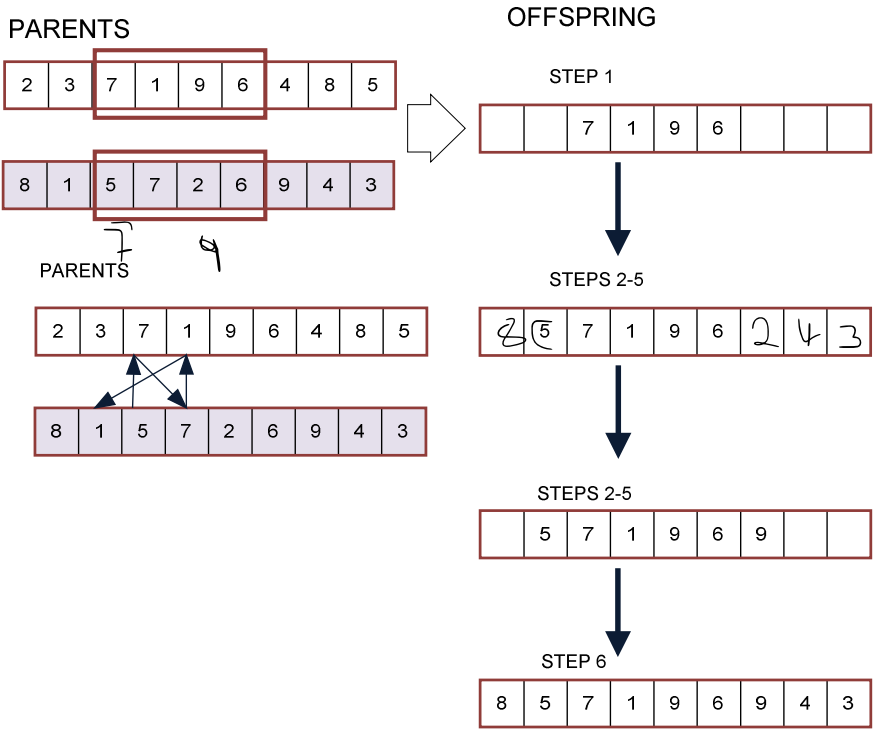

Fig. 14.21 PMX operator.

Order crossover

The order crossover operator [21] is in a way similar to PMX and has the following steps [2]:

1) Choose two crossover points at random and copy the segment between them from the first parent into the first offspring.
2) Starting from the second crossover point in the second parent, copy the remaining unused elements into the first offspring in the order that they appear in the second parent, wrapping around at the end of the list (treating string as toroidal).
3) Create the second offspring in an analogous manner, reversing the parents.

An example of the order crossover operator is shown in Figure 14.22.

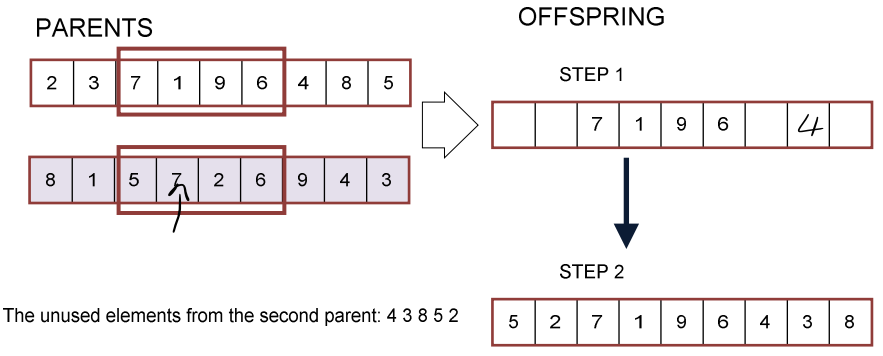

Fig. 14.22 Order crossover.

Cycle crossover

Cycle crossover [22] is concerned with preserving as much information as possible about the absolute position in which elements occur. The operator works by dividing the elements into cycles. A cycle is a subset of elements that has the propriety that each element always occurs paired with another element of the same cycle when the two parents are aligned. Once the permutations are divided in cycles, the offspring are created by selecting alternate cycles from each parent. The steps of the procedure are [2]:

1) Start with the first unused position of the first parent.
2) Look at the allele in the same position in the second parent.
3) Go to the position with the same allele in the first parent.
4) Add this allele to the cycle.
5) Repeat the steps 2-4 until you arrive at the first allele of the first parent.

An example of Cycle crossover is presented in:

- Figure 14.23 – identifying the cycles;
- Figure 14.24 – building the offspring.

```
Cycle 1: 2, 9, 8, 4 in the first parent (8, 2, 9, 4
respectively in the second parent)

Cycle 2: 3, 7, 1, 5 in the first parent (1, 5, 7, 3
respectively in the second parent).

Cycle 3: 6.
```

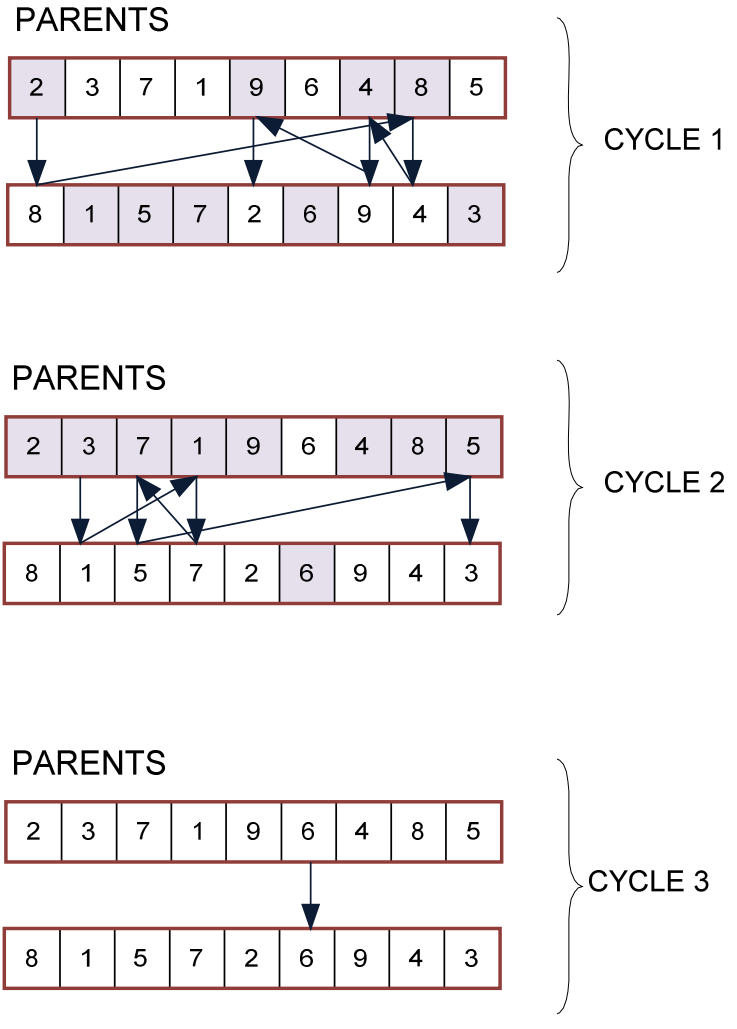

Fig. 14.23 Cycle crossover: identifying the cycles.

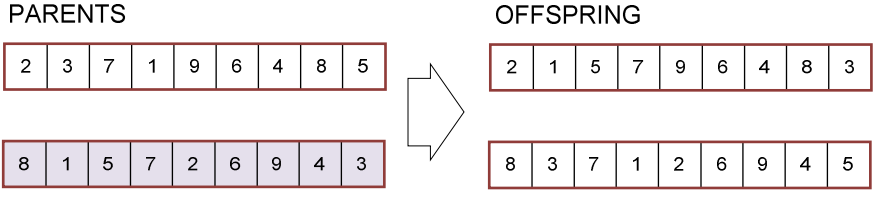

Fig. 14.24 Cycle crossover: building the offspring.

Edge crossover

Edge recombination uses the idea that an offspring should be created as far as possible using only edges that are present in one or both parents. For this, an edge table (adjacent list) is constructed which, for each element, lists the other elements that are linked to it in the two parents. A "+" in the table indicates that the edge is present in both parents. The steps of the procedure are as follows [2]:

1) Construct edge table.
2) Pick an initial element at random and put it in the offspring.
3) Set the variable current_elem = entry.
4) Remove all references to current_elem from the table.
5) Examine the list of current_elem:
    a. If there is a common edge, pick that to be the next element.
    b. Otherwise pick the entry in the list which itself has the shortest list.
    c. Ties are split at random.
6) In the case of reaching an empty list, the other end of the offspring is examined for extension. Otherwise a new element is chosen at random.

Figure 14.25 illustrates an example taken from [2] for which the parents are:

```
parent 1: 1 2 3 4 5 6 7 8 9
parent 2: 9 3 7 8 2 6 5 1 4.
```

14.3.4.1.4 Recombination for Integer Representation

For integer representation, one can apply the same operators as in the case of binary representation (the operators used for real representation might yield to non-integer values).

14.3.4.2 Mutation

By mutation individuals are randomly altered. These variations (mutation steps) are mostly small. Mutation is only applied to one individual and will produce one offspring. They will be applied to the variables of the individuals with a low probability (mutation probability or mutation rate). Normally, offspring are mutated after being created by recombination.

As in the case of recombination, mutation operator takes various forms depending on the individual's representation used.

14.3.4.2.1 Mutation for Binary Representation

For binary valued individuals mutation means the flipping of variable values, because every variable has only two states. Thus, the size of the mutation step is always 1. For every individual the variable value to change is chosen (mostly uniform at random). Figure 14.26 shows an example of a binary mutation for an individual with 10 variables, where variable 4 is mutated.

EDGE TABLE

Element	Edges
1	2, 5, 4, 9
2	1, 3, 6, 8
3	2, 4, 7, 9
4	1, 3, 5, 9
5	1, 4, 6+
6	2, 5+, 7
7	3, 6, 8+
8	2, 7+, 9
9	1, 3, 4, 8

PERMUTATION CONSTRUCTION

Choices	Element selected	Reason	Partial result
All	1	Random	1
2, 5, 4, 9	5	Shortest list	1 5
4, 6	6	Common edge	1 5 6
2, 7	2	Random	1 5 6 2
3, 8	8	Shortest list	1 5 6 2 8
7, 9	7	Common edge	1 5 6 2 8 7
3	3	Only element in list	1 5 6 2 8 7 3
4, 9	9	Random	1 5 6 2 8 7 3 9
4	4	Last element	1 5 6 2 8 7 3 9 4

Fig. 14.25 Edge crossover.

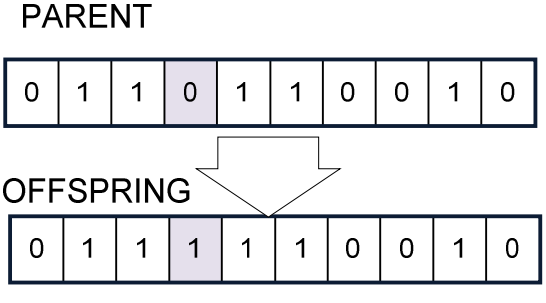

Fig. 23.26 One bit mutation example.

The most common mutation operator considers each gene separately and allows each bit to flip with a small probability. The actual number of values changed is thus not fixed but depends on the sequence of random numbers drawn. Let us consider the probability 0.5 for a bit to flip and the string of probabilities for the example above as being:

0.2, 0.35, 0.17, 0.76, 0.52, 0.27, 0.13, 0.88, 0.95, 0.12

Thus, the offspring obtained after mutation is depicted in Figure 14.27 (the genes 1, 2, 3, 6, 7 and 10 are flipped).

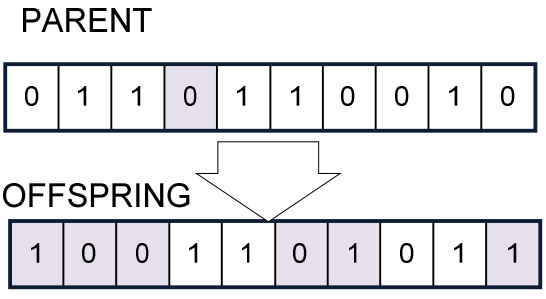

Fig. 14.27 Mutation for binary representation.

14.3.4.2.2 Mutation for Real Representation

Mutation of real variables means that randomly created values are added to the variables with a low probability. Thus, the probability of mutating a variable (mutation rate) and the size of the changes for each mutated variable (mutation step) must be defined.

The probability of mutating a variable is inversely proportional to the number of variables (dimensions). The more dimensions one individual has, the smaller is the mutation probability. Different papers reported results for the optimal mutation rate. In [23] it is mentioned that a mutation rate of 1/*nrvar* (*nrvar* represents the number of variables of an individual) produced good results for a wide variety of test functions. That means that only one variable per individual is mutated. Thus, the mutation rate is independent of the size of the population.

Two types of mutation can be distinguished according to the probability distribution from which the new gene values are drawn [2]:

- uniform mutation and
- non-uniform mutation.

Uniform Mutation

In this case the values of the genes in the offspring are drawn uniformly randomly from the definition domain. This option is analogue to bit-flipping for binary representation and the random resetting for integer representation. It is normally used with a position-wise mutation probability.

Nonuniform Mutation with a Fixed Distribution

This form of mutation is designed so that the amount of change introduced is small. This is achieved by adding to the current gene value an amount drawn randomly from a Gaussian (or normal) distribution with mean zero and user specified standard deviation. The obtained value is then scaled to the definition domain if necessarily. The Gaussian distribution has the propriety that approximately two thirds of the samples drawn lie within one standard deviation. This means that most of the changes made will be small but there is nonzero probability of generating very large changes since the tail of the distribution never reaches zero. This operator is usually applied with probability one per gene. An alternative to Gaussian distribution is to use the Cauchy distribution. The probability of generating larger values is slightly bigger than for Gaussian distribution with the same standard deviation [2][24].

14.3.4.2.3 Mutation for Order-Based Representation

For permutations it is not possible to use any of the forms of the mutation operators presented above. There are four common forms of the mutation operator for order-based representation [2][25]:

(i) swap mutation;
(ii) insert mutation;
(iii) scramble mutation;
(iv) inversion mutation.

Swap mutation

This form of mutation works by randomly picking two genes in the string and swapping their allele values. Figure 14.28 shows an example of this mutation.

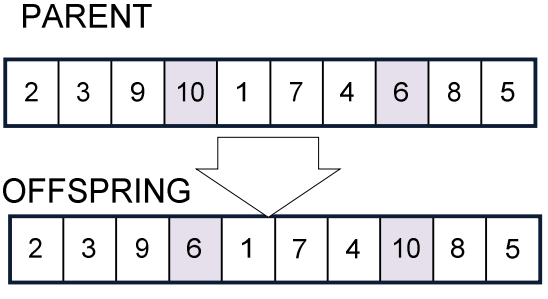

Fig. 14.28 Swap mutation.

Insert Mutation

This operator works by picking two alleles at random and moving one so that it is next to the other, shuffling along the others to make room.

An example of insert mutation is illustrated in Figure 14.29.

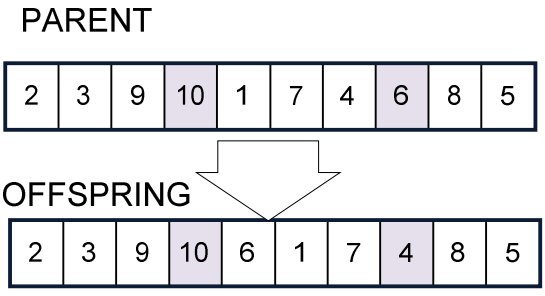

Fig. 14.29 Insert mutation.

Scramble Mutation

In this case the entire string or a subset of it (randomly chosen) has their values scrambled. An example is shown in Figure 14.30 where the selected subset is between positions 2 and 5.

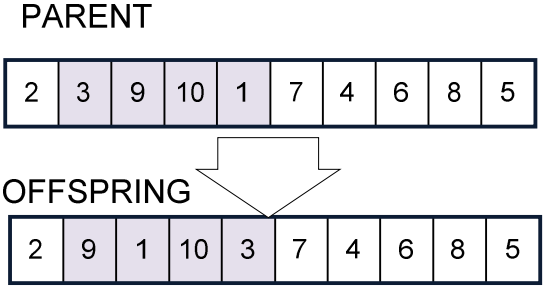

Fig. 14.30 Scramble mutation.

Inversion Mutation

This mutation operator works by randomly selecting two positions in the string and reversing the order in which the values appear between these positions. It breaks the string into three parts with all links inside a part being preserved and only the two links between the parts being broken. The inversion of a randomly chosen substring is the smallest change that can be made to an adjacency based problem.

An example is shown in Figure 14.31 with the selected positions 2 and 7.

14.3.4.2.4 Mutation for Integer Representation

There are two main forms of mutation used for integer representation; both mutate each gene independently with a defined probability:

- random resetting and
- creep mutation [2].

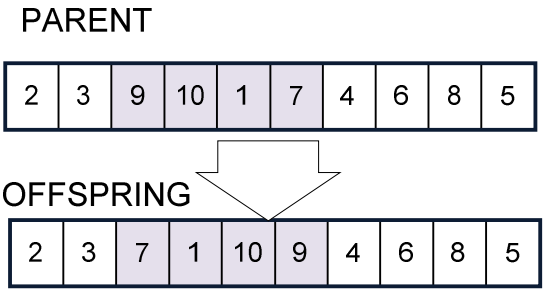

Fig. 14.31 Inversion mutation.

Random Resetting

The bit-flipping mutation of binary representation is extended to random resetting so that with a probability a new value is chosen at random from the set of permissible values in each position.

Creep Mutation

This type of mutation works by adding – with a given probability – a small (positive or negative) value to each gene. Usually, these values are sampled randomly for each gene from a distribution that is symmetric about zero and it is more likely to generate small changes than big ones.

14.3.5 Population Models

There are two main population models used by genetic algorithms:

1) Generational model and
2) Steady state model.

Generational Model

The generational model works as follows: in each generation the algorithm starts with a population of size N. A mating pool of size N is selected from this (some individuals will have multiple copies while other will not be selected at all). N offspring are further created by applying the variation operators. After each generation the whole population is replaced by the offspring population which will be the population of the next generation.

Steady-state model

In the steady-state model the population is not changed at once. In this case M ($M<N$) old individuals are replaced by M new individuals (from the offspring). The percentage of the population that is replaced is called generational gap and it

is equal to M/N. The steady state algorithm has been widely applied especially with M=1 and the corresponding generational gap $1/N$ [2].

14.3.6 Survivor Selection and Reinsertion

Once the offspring have been produced by selection, recombination and mutation of individuals from the old population, the fitness of the offspring may be determined. If less offspring are produced than the size of the original population then to maintain the size of the original population, the offspring have to be reinserted into the old population. Similarly, if not all offspring are to be used at each generation or if more offspring are generated than the size of the old population then a reinsertion scheme must be used to determine which individuals are to exist in the new population [12].

There are two main reinsertion strategies:

1) local reinsertion and
2) global reinsertion.

14.3.6.1 Local Reinsertion

In local selection, individuals are selected in a bounded neighborhood. The reinsertion of offspring takes place in exactly the same neighborhood. Thus, the locality of the information is preserved. The parent of an individual is the first selected parent in this neighborhood.

For the selection of parents to be replaced and for selection of offspring to reinsert the following schemes are possible [12]:

- insert every offspring and replace individuals in neighborhood uniformly at random;
- insert every offspring and replace weakest individuals in neighborhood;
- insert offspring fitter than weakest individual in neighborhood and replace weakest individuals in neighborhood;
- insert offspring fitter than weakest individual in neighborhood and replace parent;
- insert offspring fitter than weakest individual in neighborhood and replace individuals in neighborhood uniformly at random;
- insert offspring fitter than parent and replace parent.

14.3.6.2 Global Reinsertion

Different schemes of global reinsertion exist:

- produce as many offspring as parents and replace all parents by the offspring (pure reinsertion);
- produce less offspring than parents and replace parents uniformly at random (uniform reinsertion);

- produce less offspring than parents and replace the worst parents (elitist reinsertion);
- produce more offspring than needed for reinsertion and reinsert only the best offspring (fitness-based reinsertion).

Pure reinsertion is the simplest reinsertion scheme. Every individual lives one generation only. This scheme is used in the simple genetic algorithm. However, it is very likely, that very good individuals are replaced without producing better offspring and thus, good information is lost.

The elitist combined with fitness-based reinsertion prevents losing of information and is the recommended method. At each generation, a given number of the least fit parents are replaced by the same number of the most fit offspring. The fitness-based reinsertion scheme implements a truncation selection between offspring before inserting them into the population (i.e. before they can participate in the reproduction process). On the other hand, the best individuals can live for many generations. However, with every generation some new individuals are inserted. It is not checked whether the parents are replaced by better or worse offspring.

Because parents may be replaced by offspring with a lower fitness, the average fitness of the population can decrease. However, if the inserted offspring are extremely bad, they will be replaced with new offspring in the next generation [12].

14.3.7 The Basic Genetic Algorithm

The basic form of a general genetic algorithm is:

Step 1 Generate random population of *N* chromosomes.

Step 2 Evaluate each chromosome in the population using the fitness function

Step 3 Create a new population by repeating following steps until the new population is complete

Step 3.1 Selection

Select two parent chromosomes from a population according to their fitness (the better fitness, the higher the chance to be selected)

Step 3.2 Crossover

With a crossover probability cross over the parents to form new offspring (children). If no crossover was performed, offspring is the exact copy of parents.

Step 3.3 Mutation

With a mutation probability mutate new offspring at each locus (position in chromosome).

Step 3.4 replacement

Place new offspring in the new population

Step 4 Use new generated population for a further generation (iteration) of the algorithm

Step 5 If the termination condition is satisfied, stop, and return the best solution in current population

Step 6 Go to Step 2

Summaries

This chapter presented an evolutionary computation method with a focus on genetic algorithms. Genetic algorithms represent the most used techniques in practice among all of them. They can work with any representation and can be applied for a large variety of problems. There are any ways to speed up and improve a GA-based application as knowledge about problem domain is gained.

Some of the GAs advantages are (and these are also valid for the other EAs):

- concept is easy to understand and implement;
- modular, separate from application;
- supports multi-objective optimization;
- can be easily adapted for parallel machines;
- good for noisy and dynamic environments;
- easy to exploit previous or alternate solutions;
- flexible building blocks for hybrid applications.

When to use GAs:

- alternate solutions are too slow or overly complicated;
- need an exploratory tool to examine new approaches;
- problem is similar to one that has already been successfully solved by using a GA;
- want to hybridize with an existing solution;
- benefits of the GA technology meet key problem requirements.

References

1. Bäck, T., Fogel, D., Michalewicz, Z.: Handbook of evolutionary computation. IOP Publishing and Oxford University Press, New York (1997)
2. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer, Heidelberg (2003)
3. Eiben, A.E., Aarts, E.H.L., van Hee, K.M.: Global convergence of genetic algorithms: a markov chain analysis. In: Schwefel, H.-P., Männer, R. (eds.) PPSN 1990. LNCS, vol. 496, pp. 4–12. Springer, Heidelberg (1991)
4. Bäck, T.: Evolutionary algorithms in theory and practice. Oxford University Press, New York (1996)
5. Fogel, D.B.: Evolutionary Computation. IEEE Press, Los Alamitos (1995)

6. Bäck, T.: Generalized convergence models for tournament and (μ, λ) selection. In: Proceedings of the 6th International Conference on Genetic Algorithms, pp. 2–8 (1995)
7. Blickle, T., Thiele, L.: A comparison of selection schemes used in genetic algorithms. Evolutionary Computation 4(4), 361–394 (1996)
8. Goldberg, D.E., Deb, K.: A comparative analysis of selection schemes used in genetic algorithms. Foundations of Genetic Algorithms 1, 69–93 (1991)
9. Holland, J.H.: Adaptation in natural and artificial systems. MIT Press, Cambridge (1992)
10. Goldberg, D.E.: Generic algorithms in search, optimization and machine learning. Addison-Wesley, Reading (1989)
11. Baker, J.E.: Reducing Bias and Inefficiency in the Selection Algorithm. In: Proceedings of the Second International Conference on Genetic Algorithms and their Application, pp. 14–21. Lawrence Erlbaum Associates, Hillsdale (1987)
12. Evolutionary algorithms tutorial, `http://www.geatbx.com/`
13. Bäck, T., Hoffmeister, F.: Extended Selection Mechanisms in Genetic Algorithms. In: Bäck, T., Hoffmeister, F. (eds.) Proceedings of the Fourth International Conference on Genetic Algorithms, San Mateo, California, USA, pp. 92–99 (1991)
14. Whitley, D.: The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best. In: Proceedings of the Third International Conference on Genetic Algorithms, San Mateo, California, USA, pp. 116–121 (1989)
15. Pohlheim, H.: Ein genetischer Algorithmus mit Mehrfachpopulationen zur Numerischen Optimierung. at-Automatisierungstechnik 3, 127–135 (1995)
16. Syswerda, G.: Uniform crossover in genetic algorithms. In: Proceedings of the Third International Conference on Genetic Algorithms, San Mateo, California, USA, pp. 2–9 (1989)
17. Spears, W.M., De Jong, K.A.: On the Virtues of Parameterised Uniform Crossover. In: Proceedings of the Fourth International Conference on Genetic Algorithms, San Mateo, California, USA, pp. 230–236 (1991)
18. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. Springer, Berlin (1996)
19. Goldberg, D.E., Lingle, R.: Alleles, loci and the traveling salesman problem. In: Proceedings of the First International Conference on Genetic Algorithms and Their Applications, pp. 154–159. Lawrence Erlbaum, Hillsdale (1985)
20. Whitley, D.: Permutations, In Evolutionary Computation 1: Basic Algorithms and Operators. In: Bäck, T., Fogel, D.B. (eds.), pp. 274–284. Institute of Physics Publishing, Bristol (2000)
21. Davis, L. (ed.): Handbook of Genetic Algorithms. Van Nostrand Reinhold, New York (1991)
22. Olivier, L.M., Smith, D.J., Holland, J.: A study of permutation crossover operators on the traveling salesman problem. In: Proceedings of the Second International Conference on Genetic Algorithms and Their Applications, pp. 224–230. Lawrence Erlbaum, Hillsdale (1987)
23. Mühlenbein, H., Schlierkamp-Voosen, D.: Predictive Models for the Breeder Genetic Algorithm: I. Continuous Parameter Optimization. Evolutionary Computation 1(1), 25–49 (1993)
24. Yao, X., Liu, Y.: Fast evolutionary programming, In. In: Proceedings of the Fifth Annual Conference on Evolutionary Programming, pp. 451–460. The MIT Press, Cambridge (1996)

25. Mühlenbein, H., Pass, G.: From recombination of genes to the estimation of distributions I. Binary parameters. In: Proceedings of the 4th Conference on Parallel Problems Solving from Nature, pp. 188–197 (1996)
26. Rechenberg, I.: Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Fromman-Holzboog, Stuttgart (1973)
27. Schwefel, H.P.: Numerische Optimierung von Computermodellen mittels der Evolutionsstrategie. Birkhaeuser, Basel (1977)
28. Fogel, L.J., Owens, A.J., Walsh, M.J.: Artificial intelligence through simulated evolution. Wiley, Chichester (1966)
29. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)

Verification Questions

1. What are the main steps in building an evolutionary algorithm?
2. Enumerate a few standard representations.
3. Enumerate and explain the selection mechanisms
4. Define the main variants of crossover operators for different representations.
5. Explain the crossover operators for permutations.
6. Define the main variants of the mutation operator.
7. Explain the mutation operator for permutations.
8. Explain the main population models which can be used by a generic algorithm.
9. Explain survival selection and reinsertion mechanisms.
10. Explain local and global reinsertion.
11. Present and explain the main structure of a genetic algorithm.

Exercises

We propose a list of problems, which can be easily approached with GA.

1. **Vertex Coloring**
Given a graph $G(V, E)$, with n vertex and the connections between them and a set of k colors, color the vertices of a graph such that no two adjacent vertices share the same color.

2. **Edge Coloring**
Given a graph $G(V, E)$, with n vertex and the connections between them and a set of k colors, color the edges of a graph such that no two adjacent edges share the same color.

3. **Monochromatic triangle**
Given a graph $G(V, E)$, with n vertex and the connections between them partition it into two disjoint sets $E1$ and $E2$, such that neither of the two graphs $G1(V,E1)$ and $G2(V,E2)$ contain a triangle. That is: for all nodes in $E1$ or $E2$ there does not exist a set $\{u, v, w\}$ such that $\{u, v\}$, $\{u, w\}$, $\{v, w\}$ are all edges.

4. **Graph partitioning problem**
Given a graph $G(V, E)$ and an integer $k > 1$, partition V into k parts (subsets) V_1, V_2, ... V_k such that the parts are disjoint and have equal size, and the number of edges with endpoints in different parts is minimized.

5. **Traveling salesman problem (TSP)**
Given a list of cities and their pairwise distances, find a shortest possible tour that visits each city exactly once.

6. **Quadratic Assignment Problem (QAP)**
There are a set of n facilities and a set of n locations. For each pair of locations, a *distance* is specified and for each pair of facilities a *weight* or *flow* is specified (e.g., the amount of supplies transported between the two facilities). The problem is to assign all facilities to different locations with the goal of minimizing the sum of the distances multiplied by the corresponding flows.

Given two sets, P (facilities) and L (locations), of equal size, together with a weight function $w : P \times P \to \boldsymbol{R}$ and a distance function $d : L \times L \to \boldsymbol{R}$. Find the bijection $f : P \to L$ ("assignment") such that the cost function:

$$\sum_{a,b \in P} w(a,b) \cdot d\big(f(a), f(b)\big)$$

is minimized.

7. **Subset sum problem**
Given a set of n integers, find a subset whose sum equals S (for S given).

8. **Knapsack problem (one of the variants)**
Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible.

9. **Partition problem**
Given a multiset S of integers, find a way to partition S into two subsets S_1 and S_2 such that the sum of the numbers in S_1 equals the sum of the numbers in S_2. The subsets S_1 and S_2 must form a partition in the sense that they are disjoint and they cover S.

10. **Shortest common supersequence**
Given two sequences $X = < x_1,...,x_m >$ and $Y = < y_1,...,y_n >$, a sequence $U = < u_1,...,u_k >$ is a common supersequence of X and Y if U is a supersequence of both X and Y.

The shortest common supersequence is a common supersequence of minimal length. For X and Y given find the shortest common supersequence.

11. Evolutionary algorithm for **sudoku** game.

12. Evolutionary algorithm for **magic squares**.

13. **Crossword puzzle**
Given a crossword square and an alphabet which can be used (whose size is much higher that the number of words to be filled in the puzzle), find a valid solution for the crossword.

14. ***n*-Queens problem**
Given an $n \times n$ chess board, place n queens on it so that none of them can hit any other.

15. **Coin Problem**
Let there be $n \geq 0$ integers $0 < a_1 < \ldots < a_n$. The values a_i represent the denominations of n different coins, where these denominations have greatest common divisor of 1. Find a way to pay the sum S by using the smallest number of coins.