4

Representation, Mutation, and Recombination

As explained in Chapt. 3, there are two fundamental forces that form the basis of evolutionary systems: variation and selection. In this chapter we discuss the EA components behind the first one. Since variation operators work at the equivalent of the genetic level, that is to say they work on the representation of solutions, rather than on solutions themselves, this chapter is subdivided into sections that deal with different ways in which solutions can be represented and varied within the overall search algorithm.

4.1 Representation and the Roles of Variation Operators

The first stage of building any evolutionary algorithm is to decide on a genetic **representation** of a candidate solution to the problem. This involves defining the genotype and the mapping from genotype to phenotype. When choosing a representation, it is important to choose the right representation for the problem being solved. In many cases there will be a range of options, and getting the representation right is one of the most difficult parts of designing a good evolutionary algorithm. Often this only comes with practice and a good knowledge of the application domain. In the following sections, we look more closely at some commonly used representations, and the genetic operators that might be applied to them. It is important to stress, however, that while the representations described here are commonly used, they might not be the best representations for your application. Equally, although we present the representations and their associate operators separately, it frequently turns out in practice that using mixed representations is a more natural and suitable way of describing and manipulating a solution than trying to shoehorn different aspects of a problem into a common form.

Mutation is the generic name given to those variation operators that use only one parent and create one child by applying some kind of randomised change to the representation (genotype). The form taken depends on the choice of encoding used, as does the meaning of the associated parameter,

A.E. Eiben, J.E. Smith, *Introduction to Evolutionary Computing*,
Natural Computing Series, DOI 10.1007/978-3-662-44874-8_4

which is often introduced to regulate the intensity or magnitude of mutation. Depending on the given implementation, this can be mutation probability, mutation rate, mutation step size, etc. In the descriptions below we concentrate on the choice of operators rather than of parameters. However, the latter can make a significant difference in the behaviour of the evolutionary algorithm, and this is discussed in more depth in Chap. 7.

Recombination, the process whereby a new individual solution is created from the information contained within two (or more) parent solutions, is considered by many to be one of the most important features in evolutionary algorithms. A lot of research activity has focused on it as the primary mechanism for creating diversity, with mutation considered as a background search operator. However, different strands of EC historically emphasised different variation operators, and as these came together under the umbrella of evolutionary algorithms, this emphasis prompted a great deal of debate. Regardless of the merits of different viewpoints, the ability to combine partial solutions via recombination is certainly one of the features that most distinguishes EAs from other global optimisation algorithms.

Although the term recombination has come to be used for the more general case, early authors used the term **crossover**, motivated by the biological analogy to meiosis (see Sect. 2.3.2). Therefore we will occasionally use the terms interchangeably, although crossover tends to refer to the most common two-parent case. Recombination operators are usually applied probabilistically according to a **crossover rate** p_c. Usually two parents are selected and two offspring are created via recombination of the two parents with probability p_c; or by simply copying the parents, with probability $1 - p_c$.

Distinguishing variation operators by their arity a makes it a straightforward idea to go beyond the usual $a = 1$ (mutation) and $a = 2$ (crossover). The resulting **multiparent recombination** operators for $a = 3, 4, \ldots$ are simple to define and implement. This provides the opportunity to experiment with evolutionary processes using reproduction schemes that do not exist in biology. From the technical point of view this offers a tool for amplifying the effects of recombination. Although such operators are not widely used in EC, there are many examples that have been proposed during the development of the field, even as early as 1966 [67], see [126, 128] for an overview, and Sect. 6.6 for a description of how this idea is applied in differential evolution. These operators can be categorised by the basic mechanism used for combining the information of the parent individuals. This mechanism can be:

- based on allele frequencies, e.g., p-sexual voting [311] generalising uniform crossover;
- based on segmentation and recombination of the parents, e.g., the diagonal crossover in [139]; generalising n-point crossover
- based on numerical operations on real-valued alleles, e.g., the centre of mass crossover [434], generalising arithmetic recombination operators.

In general, it cannot be claimed that increasing the arity of recombination has a positive effect on the performance of an EA – this depends very much on the type of recombination and the problem at hand. However, systematic studies on landscapes with tuneable ruggedness [143] and a large number of experimental investigations on various problems clearly show that using more than two parents can accelerate evolutionary search and be advantageous in many cases.

4.2 Binary Representation

The first representation we look at is one of the simplest – the binary one used in Sect. 3.3. This is one of the earliest representations, and historically many genetic algorithms (GAs) have (mistakenly) used this representation almost independently of the problem they were trying to solve. Here the genotype consists simply of a string of binary digits – a bit-string.

For a particular application we have to decide how long the string should be, and how we will interpret it to produce a phenotype. In choosing the genotype–phenotype mapping for a specific problem, one has to make sure that the encoding allows that all possible bit strings denote a valid solution to the given problem[1] and that, vice versa, all possible solutions can be represented.

For some problems, particularly those concerning Boolean decision variables, the genotype–phenotype mapping is natural. One example is the knapsack problem described in Sect. 3.4.2, where for each possible item a Boolean decision was evolved, denoting whether it was included in the final solution. Frequently bit-strings are used to encode other nonbinary information. For example, we might interpret a bit-string of length 80 as 10 integers, each encoded as 8-bit integers (allowing for 256 possible values), or five 16-bit real numbers. Using bit-strings to encode nonbinary information is usually a mistake, and better results can be obtained by using the integer or real-valued representations directly.

One of the problems of coding numbers in binary is that different bits have different significance, and so the effect of a single bit mutation is very variable. Using standard binary code has the disadvantage that the Hamming distance between two consecutive integers is often not equal to one. If the goal is to evolve an integer number, you would like to have equal probabilities of changing a 7 into an 8 or a 6. However, changing 0111 to 1000 requires four bit-flips while changing it to 0110 takes just one. Thus with a mutation operator that randomly, and independently, changes each allele value with probability $p_m < 0.5$, the probability of changing 7 to 8 is much less than changing 7 to 6. This can be helped by using **Gray coding**, a variation on the way that integers are mapped on bit strings where consecutive integers always have Hamming distance one.

[1] In practice this restriction to validity in not always possible; see Chap. 13 for a more complete discussion of this issue.

4.2.1 Mutation for Binary Representation

Although a few other schemes have been occasionally used, the most common mutation operator for binary encodings considers each gene separately and allows each bit to flip (i.e., from 1 to 0 or 0 to 1) with a small probability p_m. The actual number of values changed is thus not fixed, but depends on the sequence of random numbers drawn, so for an encoding of length L, on average $L \cdot p_m$ values will be changed. In Fig. 4.1 this is illustrated for the case where the third, fourth, and eighth random values generated are less than the bitwise mutation rate p_m.

Fig. 4.1. Bitwise mutation for binary encodings

A number of studies and recommendations have been made for the choice of suitable values for the bitwise mutation rate p_m. Most binary coded GAs use mutation rates in a range such that on average between one gene per generation and one gene per offspring is mutated. However, it is worth noting at the outset that the most suitable choice to use depends on the desired outcome. For example, does the application require a population in which *all* members have high fitness, or simply that *one* highly fit individual is found? The former suggests a lower mutation rate, less likely to disrupt good solutions. In the latter case one might choose a higher mutation rate if the potential benefits of ensuring good coverage of the search space outweighed the cost of disrupting copies of good solutions[2].

4.2.2 Recombination for Binary Representation

Three standard forms of recombination are generally used for binary representations. They all start from two parents and create two children, although all of these have been extended to the more general case where a number of parents may be used [152], and there are also situations in which only one of the offspring might be considered (Sect. 5.1).

One-Point Crossover One-point crossover was the original recombination operator proposed in [220] and examined in [102]. It works by choosing a

[2] In fact this example illustrates that the algorithm's parameters cannot be chosen independently: in the second case we might couple higher mutation rates with a more aggressive selection policy to ensure the best solutions were not lost.

random number r in the range $[1, l-1]$ (with l the length of the encoding), and then splitting both parents at this point and creating the two children by exchanging the tails (Fig. 4.2, top). Note that by using the range $[1, l-1]$ the crossover point is prevented from falling before the first position ($r = 0$) or after the last position ($r = l$).

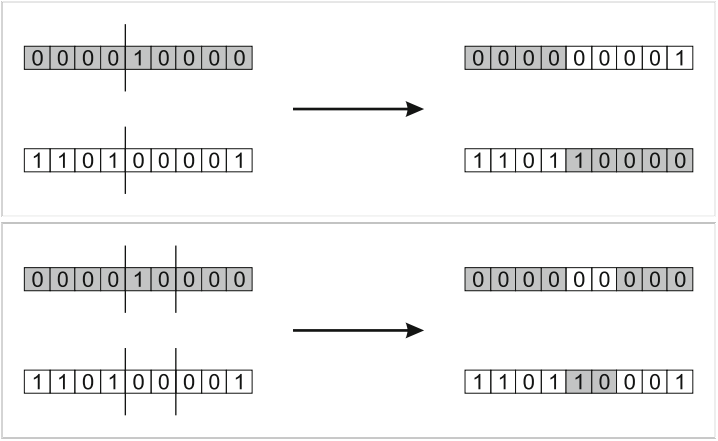

Fig. 4.2. One-point crossover (top) and n-point crossover with $n = 2$ (bottom)

n-Point Crossover One-point crossover can easily be generalised to n-point crossover, where the chromosome is broken into more than two segments of contiguous genes, and the offspring are created by taking alternative segments from the parents. In practice this means choosing n random crossover points in $[1, l-1]$, which is illustrated in Fig. 4.2 (bottom) for $n = 2$.

Uniform Crossover The previous two operators worked by dividing the parents into a number of sections of contiguous genes and reassembling them to produce offspring. In contrast to this, uniform crossover [422] works by treating each gene independently and making a random choice as to which parent it should be inherited from. This is implemented by generating a string of l random variables from a uniform distribution over [0,1]. 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. This is illustrated in Fig. 4.3.

In our discussion so far, we have suggested that in the absence of prior information, recombination worked by randomly mixing parts of the parents. However, as Fig. 4.2 illustrates, n-point crossover has an inherent bias in that it tends to keep together genes that are located close to each other in

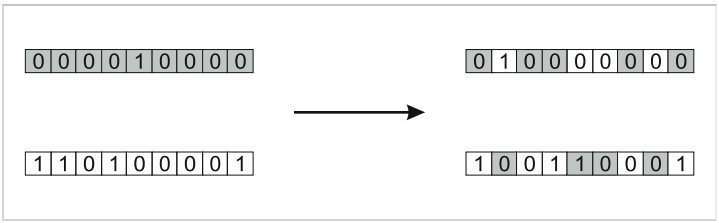

Fig. 4.3. Uniform crossover. The array [0.3, 0.6, 0.1, 0.4, 0.8, 0.7, 0.3, 0.5, 0.3] of random numbers and $p = 0.5$ were used to decide inheritance for this example.

the representation. Furthermore, when n is odd (e.g., one-point crossover), there is a strong bias against keeping together combinations of genes that are located at opposite ends of the representation. These effects are known as **positional bias** and have been extensively studied from both a theoretical and experimental perspective [157, 412] (see Sect. 16.1 for more details). In contrast, uniform crossover does not exhibit any positional bias. However, unlike n-point crossover, uniform crossover does have a strong tendency towards transmitting 50% of the genes from each parent and against transmitting an offspring a large number of coadapted genes from one parent. This is known as **distributional bias**.

The general nature of these algorithms (and the No Free Lunch theorem [467], Sect. 16.10) make it impossible to state that one or the other of these operators performs best on any given problem. Nevertheless, an understanding of the types of bias exhibited by different recombination operators can be invaluable when designing an algorithm for a particular problem, particularly if there are known patterns or dependencies in the chosen representation that can be exploited. To use the knapsack problem as an example, it might make sense to use an operator that is likely to keep together the decisions for the first few heaviest items. If the items are ordered by weight (cost) in our representation, then we could make this more likely by using n-point crossover with its positional bias. However, if we used a random ordering this might actually make it less likely that co-adapted values for certain decisions were transmitted together, so we might prefer uniform crossover.

4.3 Integer Representation

As we hinted in the previous section, binary representations are not always the most suitable if our problem more naturally maps onto a representation where different genes can take one of a set of values. One obvious example of when this might occur is the problem of finding the optimal values for a set of variables that all take integer values. These values might be unrestricted (i.e., any integer value is permissible), or might be restricted to a finite set: for example, if we are trying to evolve a path on a square grid, we might restrict the

values to the set {0,1,2,3} representing {*North, East, South, West*}. In either case an integer encoding is probably more suitable than a binary encoding. When designing the encoding and variation operators, it is worth considering whether there are any natural relations between the possible values that an attribute can take. This might be obvious for **ordinal attributes** such as integers (2 is more like 3 than it is 389), but for **cardinal attributes** such as the compass points above, there may not be a natural ordering.[3]

To give a well-known example of where there is no natural ordering, let us consider the graph k-colouring problem. Here we are given a set of points (vertices) and a list of connections between them (edges). The task is to assign one of k colours to each vertex, so that no two vertices which are connected by an edge share the same colour. For this problem there is no natural ordering: 'red' is no more like 'yellow' than 'blue', as long as they are different. In fact, we could assign the colours to the k integers representing them in any order, and still get valid equivalent solutions.

4.3.1 Mutation for Integer Representations

For integer encodings there are two principal forms of mutation used, both of which mutate each gene independently with user-defined probability p_m.

Random Resetting Here the bit-flipping mutation of binary encodings is extended to random resetting: in each position independently, with probability p_m, a new value is chosen at random from the set of permissible values. This is the most suitable operator to use when the genes encode for cardinal attributes, since all other gene values are equally likely to be chosen.

Creep Mutation This scheme was designed for ordinal attributes and works by adding a small (positive or negative) value to each gene with probability p. Usually these values are sampled randomly for each position, from a distribution that is symmetric about zero, and is more likely to generate small changes than large ones. It should be noted that creep mutation requires a number of parameters controlling the distribution from which the random numbers are drawn, and hence the size of the *steps* that mutation takes in the search space. Finding appropriate settings for these parameters may not be easy, and it is sometimes common to use more than one mutation operator in tandem from integer-based problems. For example, in [98] both a "big creep" and a "little creep" operator are used. Alternatively, random resetting might be used with low probability, in conjunction with a creep operator that tended to make small changes relative to the range of permissible values.

[3] There are various naming conventions used to distinguish these two types of attributes. These are discussed further in Chap. 7 and displayed in Table 7.1.

4.3.2 Recombination for Integer Representation

For representations where each gene has a finite number of possible allele values (such as integers) it is normal to use the same set of operators as for binary representations. On the one hand, these operators are valid: the offspring would not fall outside the given genotype space. On the other hand, these operators are also sufficient: it usually does not make sense to consider 'blending' allele values of this sort. For example, even if genes represent integer values, averaging an even and an odd integer yields a non-integral result.

4.4 Real-Valued or Floating-Point Representation

Often the most sensible way to represent a candidate solution to a problem is to have a string of real values. This occurs when the values that we want to represent as genes come from a continuous rather than a discrete distribution — for example, if they represent physical quantities such as the length, width, height, or weight of some component of a design that can be specified within a tolerance smaller than integer values. A good example would be the satellite dish holder boom described in Sect. 2.4, where the design is encoded as a series of angles and spar lengths. Another example might be if we wished to use an EA to evolve the weights on the connections beween the nodes in an artificial neural network. Of course, on a computer the precision of these real values is actually limited by the implementation, so we will refer to them as floating-point numbers. The genotype for a solution with k genes is now a vector $\langle x_1, \ldots, x_k \rangle$ with $x_i \in \mathbb{R}$.

4.4.1 Mutation for Real-Valued Representation

For floating-point representations, it is normal to ignore the discretisation imposed by hardware and consider the allele values as coming from a continuous rather than a discrete distribution, so the forms of mutation described above are no longer applicable. Instead it is common to change the allele value of each gene randomly within its domain given by a lower L_i and upper U_i bound,[4] resulting in the following transformation:

$$\langle x_1, \ldots, x_n \rangle \to \langle x'_1, \ldots, x'_n \rangle, \quad \text{where} \quad x_i, x'_i \in [L_i, U_i].$$

As with integer representations, two types can be distinguished according to the probability distribution from which the new gene values are drawn: uniform and nonuniform mutation.

[4] We assume here that the domain of each variable is a single interval $[L_i, U_i] \subseteq \mathbb{R}$. The generalisation to a union of disjoint intervals is straightforward.

Uniform Mutation For this operator the values of x_i' are drawn uniformly randomly from $[L_i, U_i]$. This is the most straightforward option, analogous to bit-flipping for binary encodings and the random resetting for integer encodings. It is normally used with a positionwise mutation probability.

Nonuniform Mutation Perhaps the most common form of nonuniform mutation used with floating-point representations takes a form analogous to the creep mutation for integers. It is designed so that usually, but not always, the amount of change introduced is small. This is achieved by adding to the current gene value an amount drawn randomly from a Gaussian distribution with mean zero and user-specified standard deviation, and then curtailing the resulting value to the range $[L_i, U_i]$ if necessary. This distribution, shown in Eq. 4.1, has the feature that the probability of drawing a random number with any given magnitude is a rapidly decreasing function of the standard deviation σ. Approximately two thirds of the samples drawn will lie within plus or minus one standard deviation, which 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. Thus the σ value is a parameter of the algorithm that determines the extent to which given values x_i are perturbed by the mutation operator. For this reason σ is often called the **mutation step size**. It is normal practice to apply this operator with probability one per gene, and instead the mutation parameter is used to control the standard deviation of the Gaussian and hence the probability distribution of the step sizes taken.

$$p(\Delta x_i) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(\Delta x_i - \xi)^2}{2\sigma^2}}. \tag{4.1}$$

An alternative to the Gaussian distribution is the use of a Cauchy distribution, which has a 'fatter' tail. That is, the probabilities of generating larger values are slightly higher than for a Gaussian with the same standard deviation [469].

4.4.2 Self-adaptive Mutation for Real-Valued Representation

As described above, non-uniform mutation applied to continuous variables is usually done by adding some random variables from a Gaussian distribution, with zero mean and a standard deviation which controls the mutation step size. The concept of **self-adaptation** represents a solution to the problem of how to adapt the step-sizes, which has been successfully demonstrated in many domains, not only for real-valued, but also for binary and integer search spaces [24]. The essential feature is that the step sizes are also included in the chromosomes and they themselves undergo variation and selection.

Details on how to mutate the value of σ are given below. The key concept is that the mutation step sizes are not set by the user; rather the σ coevolves with the solutions (the $\bar{x}$ part). In order to achieve this behaviour it is essential

to modify the value of σ first, and then mutate the x_i values with the new σ value. The rationale behind this is that a new individual $\langle \bar{x}', \sigma' \rangle$ is effectively evaluated twice. Primarily, it is evaluated directly for its viability during survivor selection based on $f(\bar{x}')$. Second, it is evaluated for its ability to create good offspring. This happens indirectly: a given step size evaluates favourably if the offspring generated by using it prove viable (in the first sense). Thus, an individual $\langle \bar{x}', \sigma' \rangle$ represents both a good $\bar{x}'$ that survived selection and a good σ' that proved successful in generating this good $\bar{x}'$ from $\bar{x}$.

The alert reader may have noticed that there is an important underlying assumption behind the idea of using varying mutation step sizes. Namely, we assume that under different circumstances different step sizes will behave differently: some will be better than others. These circumstances can be given various interpretations. For instance, we might consider time and distinguish different stages within the evolutionary search process and expect that different mutation strategies would be appropriate in different stages. Self-adaptation can then be a mechanism adjusting the mutation strategy as the search is proceeding. Alternatively, we can consider space and observe that the local vicinity of an individual, i.e., the shape of the fitness landscape in its neighbourhood, determines what good mutations are: those that jump into the direction of fitness increase. Assigning a separate mutation strategy to each individual, which coevolves with it, opens the possibility to learn and use a mutation operator suited for the local topology. Issues related to these considerations are treated extensively in the chapter on parameter control, Chap. 8. In the following we describe three special cases of self-adaptive mutation in more detail.

Uncorrelated Mutation with One Step Size In the case of uncorrelated mutation with one step size, the same distribution is used to mutate each x_i, therefore we only have one strategy parameter σ in each individual. This σ is mutated each time step by multiplying it by a term e^{Γ}, with Γ a random variable drawn each time from a normal distribution with mean 0 and standard deviation τ. Since $N(0, \tau) = \tau \cdot N(0, 1)$, the mutation mechanism is thus specified by the following formulas:

$$\sigma' = \sigma \cdot e^{\tau \cdot N(0,1)}, \tag{4.2}$$

$$x_i' = x_i + \sigma' \cdot N_i(0, 1). \tag{4.3}$$

Furthermore, since standard deviations very close to zero are unwanted (they will have on average a negligible effect), the following boundary rule is used to force step sizes to be no smaller than a threshold:

$$\sigma' < \varepsilon_0 \Rightarrow \sigma' = \varepsilon_0.$$

In these formulas $N(0, 1)$ denotes a draw from the standard normal distribution, while $N_i(0, 1)$ denotes a separate draw from the standard normal

distribution for each variable i. The proportionality constant τ is an external parameter to be set by the user. It is usually inversely proportional to the square root of the problem size:

$$\tau \propto 1/\sqrt{n}.$$

The parameter τ can be interpreted as a kind of **learning rate**, as in neural networks. Bäck [22] explains the reasons for mutating σ by multiplying with a variable with a lognormal distribution as follows:

- Smaller modifications should occur more often than large ones.
- Standard deviations have to be greater than 0.
- The median (0.5-quantile) should be 1, since we want to multiply the σ.
- Mutation should be neutral on average. This requires equal likelihood of drawing a certain value and its reciprocal value, for all values.

The lognormal distribution satisfies all these requirements.

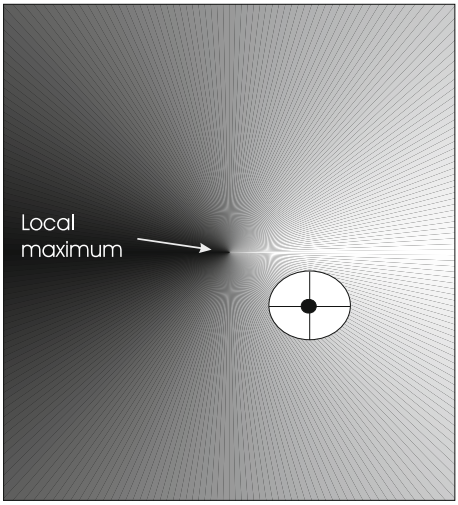

Fig. 4.4. Mutation with $n = 2, n_\sigma = 1, n_\alpha = 0$. Part of a fitness landscape with a *conical shape* is shown. The *black dot* indicates an individual. Points where the offspring can be placed with a given probability form a *circle*. The probability of moving along the y-axis (little effect on fitness) is the same as that of moving along the x-axis (large effect on fitness)

Figure 4.4 shows the effects of mutation in two dimensions. That is, we have an objective function $\mathbb{R}^2 \to \mathbb{R}$, and individuals are of the form $\langle x, y, \sigma \rangle$. Since there is only one σ, the mutation step size is the same in each direction and the points in the search space where the offspring can be placed with a given probability form a circle around the individual to be mutated.

Uncorrelated Mutation with n Step Sizes The motivation behind using n step sizes is the wish to treat dimensions differently. In particular, we want to be able to use different step sizes for different dimensions $i \in \{1, \ldots, n\}$. The reason for this is the trivial observation that the fitness landscape can have a different slope in one direction (along axis i) than in another direction (along axis j). The solution is straightforward: each basic chromosome $\langle x_1, \ldots, x_n \rangle$ is extended with n step sizes, one for each dimension, resulting in $\langle x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n \rangle$. The mutation mechanism is now specified as follows:

$$\sigma_i' = \sigma_i \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)}, \tag{4.4}$$

$$x_i' = x_i + \sigma_i' \cdot N_i(0,1), \tag{4.5}$$

where $\tau' \propto 1/\sqrt{2n}$, and $\tau \propto 1/\sqrt{2\sqrt{n}}$. Once again a boundary rule is applied to prevent standard deviations very close to zero.

$$\sigma_i' < \varepsilon_0 \Rightarrow \sigma_i' = \varepsilon_0.$$

Notice that the mutation formula for σ is different from that in Eq. (4.2). The present mutation mechanism is based on a finer granularity. Instead of the individual level (each individual $\bar{x}$ having its own σ) it works on the coordinate level (one σ_i for each x_i in $\bar{x}$). The corresponding straightforward modification of Eq. (4.2) is

$$\sigma_i' = \sigma_i \cdot e^{\tau \cdot N_i(0,1)},$$

but ES use Eq. (4.4). Technically, this is correct since the sum of two normally distributed variables is also normally distributed, hence the resulting distribution is still lognormal. The conceptual motivation is that the common base mutation $e^{\tau' \cdot N(0,1)}$ allows for an overall change of the mutability, guaranteeing the preservation of all degrees of freedom, while the coordinate-specific $e^{\tau \cdot N_i(0,1)}$ provides the flexibility to use different mutation strategies in different directions.

In Fig. 4.5 the effects of mutation are shown in two dimensions. Again, we have an objective function $\mathbb{R}^2 \to \mathbb{R}$, but the individuals now have the form $\langle x, y, \sigma_x, \sigma_y \rangle$. Since the mutation step sizes can differ in each direction (x and y), the points in the search space where the offspring can be placed with a given probability form an ellipse around the individual to be mutated. The axes of such an ellipse are parallel to the coordinate axes, with the length along axis i proportional to the value of σ_i.

Correlated Mutations The second version of mutation discussed above introduced different standard deviations for each axis, but this only allows ellipses orthogonal to the axes. The rationale behind correlated mutations is to allow the ellipses to have any orientation by rotating them with a rotation (covariance) matrix C.

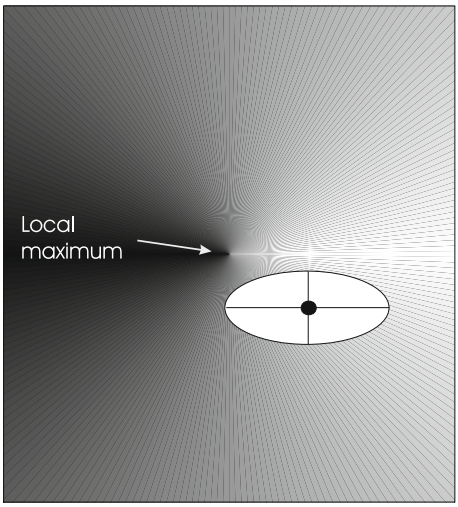

Fig. 4.5. Mutation with $n = 2, n_\sigma = 2, n_\alpha = 0$. Part of a fitness landscape with a *conical shape* is shown. The *black dot* indicates an individual. Points where the offspring can be placed with a given probability form an *ellipse*. The probability of moving along the x-axis (large effect on fitness) is larger than that of moving along the y-axis (little effect on fitness)

The probability density function for $\overline{\Delta x}$ replacing Eq. (4.1) now becomes

$$p(\overline{\Delta x}) = \frac{e^{-\frac{1}{2}\overline{\Delta x}^T \cdot C^{-1} \cdot \overline{\Delta x}}}{(\det C \cdot (2\pi)^n)^{1/2}},$$

with C the covariance matrix with entries

$$c_{ii} = \sigma_i^2, \tag{4.6}$$

$$c_{ij,i\neq j} = \begin{cases} 0 & \text{no correlations,} \\ \frac{1}{2}(\sigma_i^2 - \sigma_j^2)\tan(2\alpha_{ij}) & \text{correlations.} \end{cases} \tag{4.7}$$

The relation between covariance and rotation angle is as follows:

$$\tan(2\alpha_{ij}) = \frac{2c_{ij}}{\sigma_i^2 - \sigma_j^2},$$

which explains Eq. (4.7). This formula is derived from the trigonometric properties of rotations. A rotation in two dimensions is a multiplication with the matrix

$$\begin{pmatrix} \cos(\alpha_{ij}) & -\sin(\alpha_{ij}) \\ \sin(\alpha_{ij}) & \cos(\alpha_{ij}) \end{pmatrix}.$$

A rotation in more dimensions can be performed by a successive series of 2D rotations, i.e., matrix multiplications.

The complete mutation mechanism is described by the following equations:

$$\begin{aligned}\sigma_i' &= \sigma_i \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)},\\ \alpha_j' &= \alpha_j + \beta \cdot N_j(0,1),\\ \overline{x}' &= \overline{x} + \overline{N}(\overline{0}, C'),\end{aligned}$$

where $n_\alpha = \frac{n \cdot (n-1)}{2}$, $j \in 1, \ldots, n_\alpha$. The other constants are usually taken as: $\tau \propto 1/\sqrt{2\sqrt{n}}$, $\tau' \propto 1/\sqrt{2n}$, and $\beta \approx 5^o$.

The object variables $\overline{x}$ are now mutated by adding $\overline{\Delta x}$ drawn from an n-dimensional normal distribution with covariance matrix C'. The C' in the formula is the old C after mutation of the α values (and recalculation of covariances). The σ_i are mutated in the same way as before: with a multiplication by a log-normal variable, which consists of a global and an individual part. The α_j are mutated with an additive, normally distributed variation, similar to mutation of object variables.

We also have a boundary rule for the α_j values. The rotation angles should lie in the range $[-\pi, \pi]$, so the new value is simply mapped circularly into the feasible range:

$$|\alpha_j'| > \pi \Rightarrow \alpha_j' = \alpha_j' - 2\pi \text{ sign}(\alpha_j').$$

Fig. 4.6 shows the effects of correlated mutations in two dimensions. The individuals now have the form $\langle x, y, \sigma_x, \sigma_y, \alpha_{x,y} \rangle$, and the points in the search space where the offspring can be placed with a given probability form a rotated ellipse around the individual to be mutated, where again the axis lengths are proportional to the σ values.

Table 4.1 summarises three possible common settings for self-adaptive mutation regarding the length and structure of the individuals. Simply considering the size of the representation of the individuals in each scheme, i.e., the number of values that need to be learned by the algorithm as it evolves (let alone their complex interrelationships) brings home an important point: we can get nothing for free! In other words, what we must consider is that as the ability of the algorithm to adapt the nature of its search according to the local topology increases, so too does the scale of the learning task. To simplify matters a little, as we increase the precision with which we can specify the shape of the lines of equiprobable mutations, so we increase the number of different options which should be tried. Since the merits of these different possibilities are evaluated indirectly, i.e., by applying them and gauging the relative fitness of the individuals created, it is reasonable to conclude that an increased number of function evaluations will be needed to learn good search strategies as the complexity of the mutation operator increases.

While this may sound a little pessimistic, it is also worth noting that it is easy to imagine a situation where the extra complexity is required, for example, if the landscape contains a 'ridge' of increasing fitness, perhaps running at

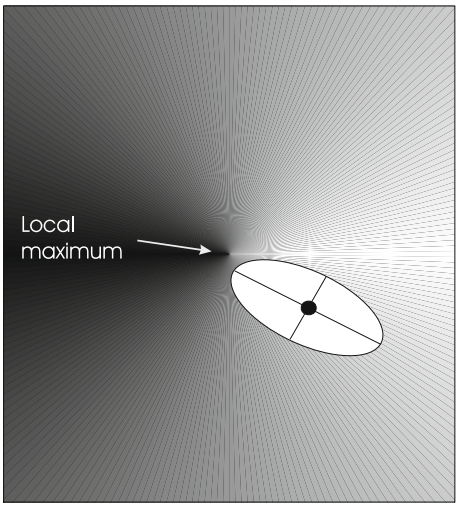

Fig. 4.6. Correlated mutation: $n = 2, n_\sigma = 2, n_\alpha = 1$. Part of a fitness landscape with a *conical shape* is shown. The *black dot* indicates an individual. Points where the offspring can be placed with a given probability form a *rotated ellipse*. The probability of generating a move in the direction of the steepest ascent (largest effect on fitness) is now larger than that for other directions

an angle to the co-ordinate axis. In short, there are no fixed recommendations about which scheme to use, but a common approach is to start with uncorrelated mutation with n σ values and then try moving to a simpler model if good results are obtained but too slowly (or if the σ_i all evolve to similar values), or to the more complex model if the results are not of good enough quality.

n_σ	n_α	Structure of individuals	Remark
1	0	$\langle x_1, \ldots, x_n, \sigma \rangle$	Standard mutation
n	0	$\langle x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n \rangle$	Standard mutations
n	$n \cdot (n-1)/2$	$\langle x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n, \alpha_1, \ldots, \alpha_{n \cdot (n-1)/2} \rangle$	Correlated mutations

Table 4.1. Some possible settings of n_σ and n_α for different mutation operators

Self-adaptive mutation mechanisms have been used and studied for decades in EC. Besides experimental evidence, showing that an EA with self-adaptation outperforms the same algorithm without self-adaptation, there are also theoretical results showing that self-adaptation works [52]. Theoretical and experimental results can neatly complement each other in this area if experimentally obtained mutation step sizes show a good match with the theoretically derived optimal values. Unfortunately, for a complex problem

and/or algorithm a theoretical analysis is infeasible. However, for simple objective functions theoretically optimal mutation step sizes can be calculated (in light of some performance criterion, e.g., progress rate during a run) and compared to step sizes obtained during a run of the EA in question.

Theoretical and experimental results agree on the fact that for a successful run the σ values must decrease over time. The intuitive explanation for this is that in the beginning of a search process a large part of the search space has to be sampled in an explorative fashion to locate promising regions (with good fitness values). Therefore, large mutations are appropriate in this phase. As the search proceeds and optimal values are approached, only fine tuning of the given individuals is needed; thus smaller mutations are required.

Another kind of convincing evidence for the power of self-adaptation is provided in the context of changing fitness landscapes. In this case, where the objective function is changing, the evolutionary process is aiming at a moving target. When the objective function changes, the given individuals may have a low fitness, since they have been adapted to the old objective function. Thus, the present population needs to be reevaluated, and the search space re-explored. Often the mutation step sizes will prove ill-adapted: they are too low for the new exploration phase required. The experiment presented in [217] illustrates how self-adaptation is able to reset the step sizes after each change in the objective function (Fig. 4.7).

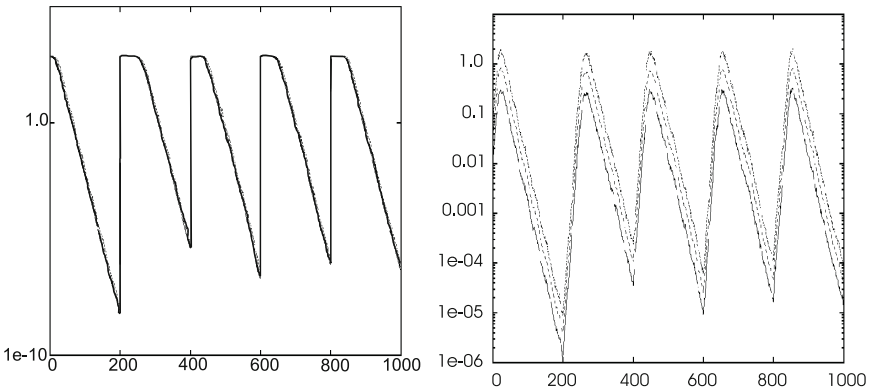

Fig. 4.7. Moving optimum ES experiment on the sphere function with $n = 30$, $n_\sigma = 1$. The location of the optimum is changed after every 200 generations (x-axes) with a clear effect on the average best objective function values (y-axis, left) in the given population. Self-adaptation is adjusting the step sizes (y-axis, right) with a small delay to larger values appropriate for exploring the new fitness landscape, whereafter the values of σ start decreasing again as the population approaches the new optimum

Over recent decades much experience has been gained over self-adaptation in Evolutionary Algorithms, in particular in Evolution Strategies. The accumulated knowledge has identified necessary conditions for self-adaptation:

1. $\mu > 1$ so that different strategies are present
2. generation of an offspring surplus: $\lambda > \mu$
3. a not too strong selective pressure (heuristic: $\lambda/\mu = 7$, e.g., (15,100))
4. (μ, λ)-selection (to guarantee extinction of misadapted individuals)
5. recombination, usually intermediate, of strategy parameters

4.4.3 Recombination Operators for Real-Valued Representation

In general, we have three options for recombining two floating-point strings. First, using an analogous operator to those used for bit-strings, but now split between floats. In other words, an allele is one floating-point value instead of one bit. This has the disadvantage (shared with all of the recombination operators described above) that only mutation can insert new values into the population, since recombination only gives us new combinations of existing values. Recombination operators of this type for floating-point representations are known as **discrete recombination** and have the property that if we are creating an offspring z from parents x and y, then the allele value for gene i is given by $z_i = x_i$ or y_i with equal likelihood.

Second, using an operator that, in each gene position, creates a new allele value in the offspring that lies between those of the parents. Using the terminology above, we have $z_i = \alpha x_i + (1 - \alpha)y_i$ for some α in [0,1]. In this way, recombination is now able to create new gene material, but it has the disadvantage that as a result of the averaging process the range of the allele values in the population for each gene is reduced. Operators of this type are known as **intermediate** or **arithmetic recombination**.

Third, using an operator that in each position creates a new allele value in the offspring which is close to that of one of the parents, but may lie outside them (i.e., bigger than the larger of the two values, or smaller than the lesser). Operators of this type can create new material without restricting the range. Operators of this type are known as **blend recombination**.

Three types of arithmetic recombination are described in [295]. In all of these, the choice of the parameter α is sometimes made at random over [0,1], but in practice it is common to use a constant value, often 0.5 (in which case we have **uniform arithmetic recombination**).

Simple Arithmetic Recombination First pick a recombination point k. Then, for child 1, take the first k floats of parent 1 and put them into the child. The rest is the arithmetic average of parent 1 and 2:

$$\text{Child 1: } \langle x_1, \ldots, x_k, \alpha \cdot y_{k+1} + (1 - \alpha) \cdot x_{k+1}, \ldots, \alpha \cdot y_n + (1 - \alpha) \cdot x_n \rangle.$$

Child 2 is analogous, with x and y reversed (Fig. 4.8, top).

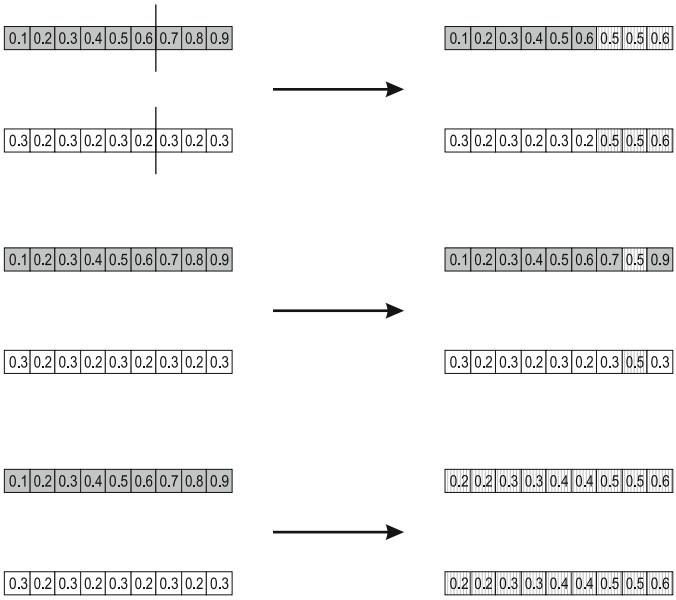

Fig. 4.8. Simple arithmetic recombination with $k = 6, \alpha = 1/2$ (top), single arithmetic recombination with $k = 8, \alpha = 1/2$ (middle), whole arithmetic recombination with $\alpha = 1/2$ (bottom).

Single Arithmetic Recombination Pick a random allele k. At that position, take the arithmetic average of the two parents. The other points are the points from the parents, i.e.:

$$\text{Child 1: } \langle x_1, \ldots, x_{k-1}, \alpha \cdot y_k + (1-\alpha) \cdot x_k, x_{k+1}, \ldots, x_n \rangle.$$

The second child is created in the same way with x and y reversed (Fig. 4.8, middle).

Whole Arithmetic Recombination This is the most commonly used operator and works by taking the weighted sum of the two parental alleles for each gene, i.e.:

$$\text{Child 1} = \alpha \cdot \bar{x} + (1-\alpha) \cdot \bar{y}, \qquad \text{Child 2} = \alpha \cdot \bar{y} + (1-\alpha) \cdot \bar{x}.$$

This is illustrated in Fig. 4.8, bottom. As the example shows, if $\alpha = 1/2$ the two offspring will be identical for this operator.

Blend Crossover Blend Crossover ($BLX - \alpha$) was introduced in [160] as a way of creating offspring in a region that is bigger than the (n-dimensional) rectangle spanned by the parents. The extra space is proportional to the

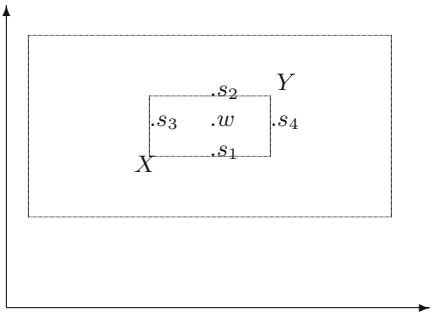

Fig. 4.9. Possible offspring from different recombination operators for two real-valued parents X and Y. $\{s_1, \ldots, s_4\}$ are the four possible offspring from single arithmetic recombination with $\alpha = 0.5$. w is the offspring from whole arithmetic recombination with $\alpha = 0.5$ and the *inner* box represents all the possible offspring positions as α is varied. The *outer dashed* box shows all possible offspring positions for blend crossover with $\alpha = 0.5$ ($BLX - 0.5$), each position being equally likely.

distance between the parents and it varies per coordinate. If we have two parents x and y and assume that in position i the value $x_i < y_i$ then the difference $d_i = y_i - x_i$ and the range for the ith value in the child z is $[x_i - \alpha \cdot d_i, x_i + \alpha \cdot d_i]$. To create a child we can sample a random number u uniformly from $[0, 1]$, calculate $\gamma = (1 - 2\alpha)u - \alpha$, and set:

$$z_i = (1 - \gamma)x_i + \gamma y_i$$

Interestingly, the original authors reported best results with $\alpha = 0.5$, where the chosen values are equally likely to lie inside the two parent values as outside, so balancing exploration and exploitation.

Figure 4.9 illustrates the difference between single arithmetic recombination, whole arithmetic combination and Blend Crossover, with in each case the value of α set to 0.5. More recent methods such as Simulated Binary Crossover [111, 113] have built on Blend Crossover, so that rather than selecting offspring values uniformly from a range around each parent values, they are selected from a distribution which is more likely to create small changes, and the distribution is controlled by the distance between the parents.

4.5 Permutation Representation

Many problems naturally take the form of deciding on the order in which a sequence of events should occur. While other forms do occur (for example, decoder functions based on unrestricted integer representations [28, 201] or "floating keys" based on real-valued representations [27, 44]), the most natural representation of such problems is as a permutation of a fixed set of values

that can be represented as integers. One immediate consequence is that while a binary, or simple integer, representation allows numbers to occur more than once, such sequences of integers will not represent valid permutations. It is clear therefore that when choosing or designing variation operators to work with solutions that are represented as permutations, we require them to preserve the permutation property that each possible allele value occurs exactly once in the solution. We previously described one example, when we designed an EA for solving the N-queens problem efficiently, by representing each solution as a list of the rows on which each queen was positioned (with each on a different column), and insisted that these be a permutation so that no two queens shared the same row.

When choosing variation operators it is worth bearing in mind that there are actually two classes of problems that are represented by permutations. In the first of these, the *order* in which events occur is important. This might happen when the events use limited resources or time, and a typical example of this sort of problem is the production scheduling problem. This is the common problem of deciding in which order a series of times should be manufactured on a set of machines, where there may be dependencies between products, for example, there might be different set-up times between products, or one might be a component of another. As an example, it might be better for widget 1 to be produced before widgets 2 and 3, which in turn might be preferably produced before widget 4, no matter how far in advance this is done. In this case it might well be that the sequences [1,2,3,4] and [1,3,2,4] have similar fitness, and are much better than, for example, [4,3,2,1].

Another type of problem depends on *adjacency*, and is typified by the travelling salesperson problem (TSP). The problem is to find a complete tour of n given cities of minimal length. The search space for this problem is huge: there are $(n\text{-}1)!$ different routes possible for n given cities (for the asymmetric case counting back and forth as two routes).[5] For $n = 30$ there are approximately 10^{32} different tours. Labelling the cities $1, 2, \ldots, n$, a complete tour is a permutation, so that for $n = 4$, the routes [1,2,3,4] and [3,4,2,1] are both valid. The vital point here is that it is the links between cities that are important. The difference from order-based problems can clearly be seen if we consider that the starting point of the tour is also not important, thus [1,2,3,4], [2,3,4,1], [3,4,1,2], and [4,1,2,3] are all equivalent. Many examples of this class are also symmetric, so that [4,3,2,1] and so on are also equivalent.

Finally, we should mention that there are two possible ways to encode a permutation. In the first (most commonly used) of these the ith element of the representation denotes the event that happens in that place in the sequence (or the ith destination visited). In the second, the value of the ith element denotes the position in the sequence in which the ith event happens. Thus for the four cities [A,B,C,D], and the permutation [3,1,2,4], the first encoding denotes the tour [C,A,B,D] and the second [B,C,A,D].

[5] These comments about problem size apply to all permutation problems.

4.5.1 Mutation for Permutation Representation

For permutation representations, it is no longer possible to consider each gene independently, rather finding legal mutations is a matter of moving alleles around in the genome. This has the immediate consequence that the mutation parameter is interpreted as the probability that the *chromosome* undergoes mutation, rather than that a single gene in the chromosome is altered. The three most common forms of mutation used for order-based problems were first described in [423]. Whereas the first three operators below (in particular insertion) work by making small changes to the order in which allele values occur, for adjacency-based problems these can cause huge numbers of links to be broken, and so inversion is more commonly used.

Swap Mutation Two positions (genes) in the chromosome are selected at random and their allele values swapped. This is illustrated in Fig. 4.10 (top), where the values in positions two and five have been swapped.
Insert Mutation Two alleles are selected at random and the second moved next to the first, shuffling along the others to make room. This is illustrated in Fig. 4.10 (middle), where the values two and five have been chosen.
Scramble Mutation Here the entire chromosome, or some randomly chosen subset of values within it, have their positions scrambled. This is illustrated in Fig. 4.10 (bottom), where the values from two to five have been chosen.

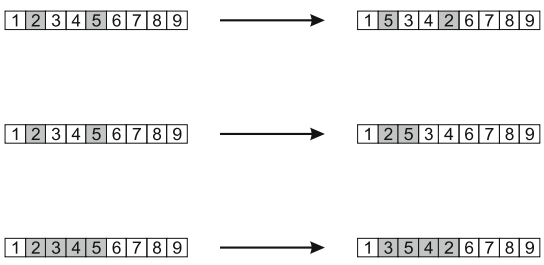

Fig. 4.10. Swap (top), insert (middle), and scramble mutation (bottom).

Inversion Mutation Inversion mutation works by randomly selecting two positions in the chromosome and reversing the order in which the values appear between those positions. It effectively breaks the chromosome 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 thus smallest change that can be made to an adjacency-based problem, and all other changes can be easily constructed as a series of inversions. The

ordering of the search space induced by this operator thus forms a natural basis for considering this class of problems, equivalent to the Hamming space for binary problem representations. It is the basic move behind the 2-opt search heuristic for TSP [271], and by extension k-opt. This operator is illustrated in Fig. 4.11, where the substring between positions two and five was inverted.

Fig. 4.11. Inversion mutation

4.5.2 Recombination for Permutation Representation

At first sight, permutation-based representations present particular difficulties for the design of recombination operators, since it is not generally possible simply to exchange substrings between parents and still maintain the permutation property. However, this situation is alleviated when we consider what it is that the solutions actually represent, i.e., either an order in which elements occur, or a set of moves linking pairs of elements. A number of specialised recombination operators have been designed for permutations, which aim at transmitting as much as possible of the information contained in the parents, especially that held in common. We shall concentrate here on describing two of the best known and most commonly used operators for each subclass of permutation problems.

Partially Mapped Crossover (PMX) was first proposed by Goldberg and Lingle as a recombination operator for the TSP in [192], and has become one of the most widely used operators for adjacency-type problems. Over the years many slight variations of PMX appeared in the literature; here we use Whitley's definition from [452], which works as follows (Figs. 4.12–4.14).

1. Choose two crossover points at random, and copy the segment between them from the first parent (P1) into the first offspring.
2. Starting from the first crossover point look for elements in that segment of the second parent (P2) that have not been copied.
3. For each of these (say i), look in the offspring to see what element (say j) has been copied in its place from P1.
4. Place i into the position occupied by j in P2, since we know that we will not be putting j there (as we already have it in our string).
5. If the place occupied by j in P2 has already been filled in the offspring by an element k, put i in the position occupied by k in P2.

6. Having dealt with the elements from the crossover segment, the remaining positions in this offspring can be filled from P2, and the second child is created analogously with the parental roles reversed.

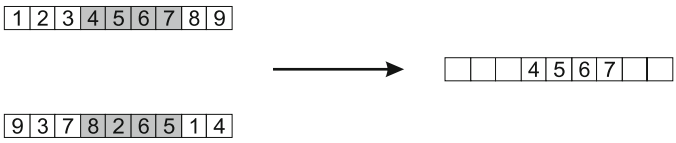

Fig. 4.12. PMX, step 1: copy randomly selected segment from first parent into offspring

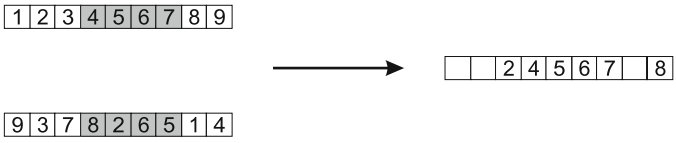

Fig. 4.13. PMX, step 2: consider in turn the placement of the elements that occur in the middle segment of parent 2 but not parent 1. The position that 8 takes in P2 is occupied by 4 in the offspring, so we can put the 8 into the position vacated by the 4 in P2. The position of the 2 in P2 is occupied by the 5 in the offspring, so we look first to the place occupied by the 5 in P2, which is position 7. This is already occupied by the value 7, so we look to where this occurs in P2 and finally find a slot in the offspring that is vacant – the third. Finally, note that the values 6 and 5 occur in the middle segments of both parents.

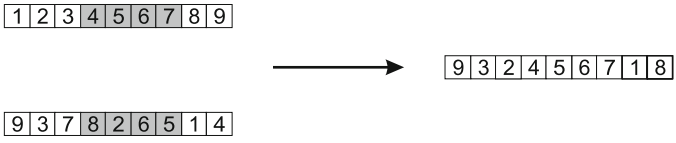

Fig. 4.14. PMX, step 3: copy remaining elements from second parent into same positions in offspring

Inspection of the offspring created shows that in this case six of the nine links present in the offspring are present in one or more of the parents. However, of the two edges {5–6} and {7–8} common to both parents, only the first is present in the offspring. Radcliffe [350] suggests that a desirable property

of any recombination operator is that of *respect*, i.e., that any information carried in both parents should also be present in the offspring. A moment's reflection tells us that this is clearly true for all of the recombination operators described above for binary and integer representations, and for discrete recombination for floating-point representations, but as the example above shows, is not necessarily true of PMX. With this issue in mind, several other operators have been designed for adjacency-based permutation problems, of which the best known is described next.

Edge crossover is based on the idea that offspring should be created as far as possible using only edges that are present in (one of) the parents. It has undergone a number of revisions over the years. Here we describe the most commonly used version: edge-3 crossover after Whitley [452], which is designed to ensure that common edges are preserved.

In order to achieve this, an edge table (also known as an adjacency 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 operator works as follows:

1. Construct the edge table
2. Pick an initial element at random and put it in the offspring
3. Set the variable $current\_element = entry$
4. Remove all references to $current\_element$ from the table
5. Examine list for $current\_element$
   - If there is a common edge, pick that to be the next element
   - Otherwise pick the entry in the list which itself has the shortest list
   - 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

Clearly only in the last case will so-called foreign edges be introduced.

Edge-3 recombination is illustrated by the following example where the parents are the same two permutations used in the PMX example [1 2 3 4 5 6 7 8 9] and [9 3 7 8 2 6 5 1 4], giving the edge table seen in Table 4.2 and the construction illustrated in Table 4.3. Note that only one child per recombination is created by this operator.

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

Table 4.2. Edge crossover: example edge table

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 choice (both have two items in list)	[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 item in list	[1 5 6 2 8 7 3]
4,9	9	Random choice	[1 5 6 2 8 7 3 9]
4	4	Last element	[1 5 6 2 8 7 3 9 4]

Table 4.3. Edge crossover: example of permutation construction

Order crossover This operator was designed by Davis for order-based permutation problems [98]. It begins in a similar fashion to PMX, by copying a randomly chosen segment of the first parent into the offspring. However, it proceeds differently because the intention is to transmit information about *relative order* from the second parent.

1. Choose two crossover points at random, and copy the segment between them from the first parent (P1) into the first offspring.
2. Starting from the second crossover point in the second parent, copy the remaining unused numbers into the first child in the order that they appear in the second parent, wrapping around at the end of the list.
3. Create the second offspring in an analogous manner, with the parent roles reversed.

This is illustrated in Figs. 4.15 and 4.16.

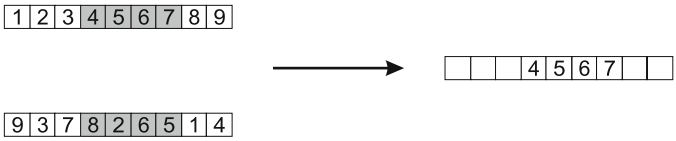

Fig. 4.15. Order crossover, step 1: copy randomly selected segment from first parent into offspring

Cycle Crossover The final operator that we will consider in this section is cycle crossover [325], which is concerned with preserving as much information as possible about the *absolute* position in which elements occur. The operator

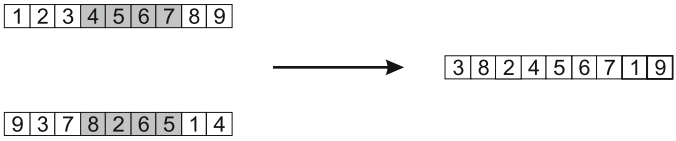

Fig. 4.16. Order crossover, step 2: copy rest of alleles in order they appear in second parent, treating string as toroidal

works by dividing the elements into *cycles*. A cycle is a subset of elements that has the property that each element always occurs paired with another element of the same cycle when the two parents are aligned. Having divided the permutation into cycles, the offspring are created by selecting alternate cycles from each parent. The procedure for constructing cycles is as follows:

1. Start with the first unused position and allele of P1
2. Look at the allele in the *same position* in P2
3. Go to the position with the *same allele* in P1
4. Add this allele to the cycle
5. Repeat steps 2 through 4 until you arrive at the first allele of P1

The complete operation of the operator is illustrated in Fig. 4.17.

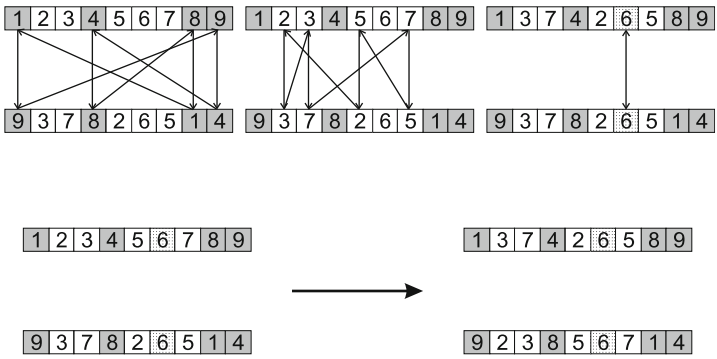

Fig. 4.17. Cycle crossover. Top: step 1- identification of cycles. Bottom: step 2- construction of offspring

4.6 Tree Representation

Trees are among the most general structures for representing objects in computing, and form the basis for the branch of evolutionary algorithms known as genetic programming (GP). In general, (parse) trees capture expressions in a given formal syntax. Depending on the problem at hand, and the users' perceptions on what the solutions must look like, this can be the syntax of arithmetic expressions, formulas in first-order predicate logic, or code written in a programming language. To illustrate the matter, let us consider one of each of these types of expressions.

- an arithmetic formula:

$$2 \cdot \pi + ((x + 3) - \frac{y}{5 + 1}), \tag{4.8}$$

- a logical formula:

$$(x \wedge true) \rightarrow ((x \vee y) \vee (z \leftrightarrow (x \wedge y))), \tag{4.9}$$

- the following program:

```
i = 1;
while (i < 20)
{
        i = i+1;
}
```

Figures. 4.18 and 4.19 show the parse trees belonging to these expressions. These examples illustrate generally how parse trees can be used and interpreted.

Technically speaking, the specification of how to represent individuals boils down to defining the syntax of the trees, or equivalently the syntax of the symbolic expressions (**s-expressions**) they represent. This is commonly done by defining a **function set** and a **terminal set**. Elements of the terminal set are allowed as leaves, while symbols from the function set are internal nodes. For example, a suitable function and terminal set that allow the expression in Eq. (4.8) as syntactically correct is given in Table 4.4.

Function set	$\{+, -, \cdot, /\}$
Terminal set	$\mathbb{R} \cup \{x, y\}$

Table 4.4. Function and terminal set that allow the expression in Eq. (4.8) as syntactically correct

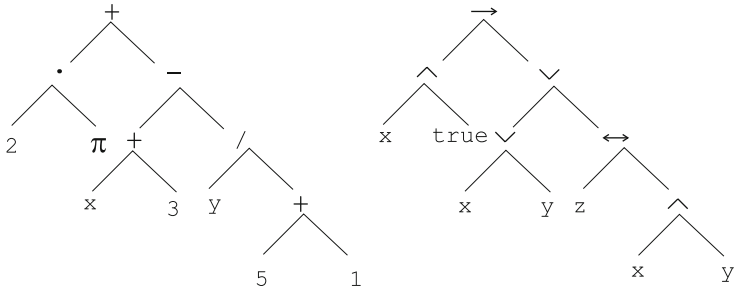

Fig. 4.18. Parse trees belonging to Eqs. (4.8) (left) and (4.9) (right)

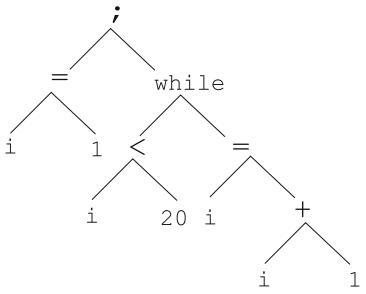

Fig. 4.19. Parse tree belonging to the above program

Strictly speaking, we should specify the arity (the number of attributes it takes) for each function symbol in the function set, but for standard arithmetic or logical functions this is often omitted. Similarly, a definition of correct expressions (trees) based on the function and terminal set should be given. However, as this follows the general way of defining terms in formal languages it is also often omitted. For the sake of completeness we provide it below:

- All elements of the terminal set T are correct expressions.
- If $f \in F$ is a function symbol with arity n and $e_1, \ldots, e_n$ are correct expressions, then so is $f(e_1, \ldots, e_n)$.
- There are no other forms of correct expressions.

Note that in this definition we do not distinguish different types of expressions; each function symbol can take any expression as argument. This feature is known as the **closure property**.

In practice, function symbols and terminal symbols are often typed and impose extra syntactic requirements. For instance, one might need both arithmetic and logical function symbols, e.g., to allow $(N = 2) \wedge (S > 80.000))$ as a correct expression. In this case it is necessary to enforce that an arithmetic (logical) function symbol only has arithmetic (logical) arguments, e.g., to exclude $N \wedge 80.000$ as a correct expression. This issue is addressed in strongly typed genetic programming [304].

4.6.1 Mutation for Tree Representation

The most common implementation of **tree-based mutation** works by selecting a node at random from the tree, and replacing the subtree starting there with a randomly generated tree. This newly created subtree is usually generated the same way as in the initial population, (Sect. 6.4), and so is subject to conditions on maximum depth and width. Figure 4.20 illustrates how the parse tree belonging to Eq. (4.8) (left) is mutated into one standing for $2 \cdot \pi + ((x + 3) - y)$. Note that since a node is selected at random to be the replacement point, and that as one goes down through a tree there are potentially more nodes at any given depth, the size (tree depth) of the child can exceed that of the parent tree.

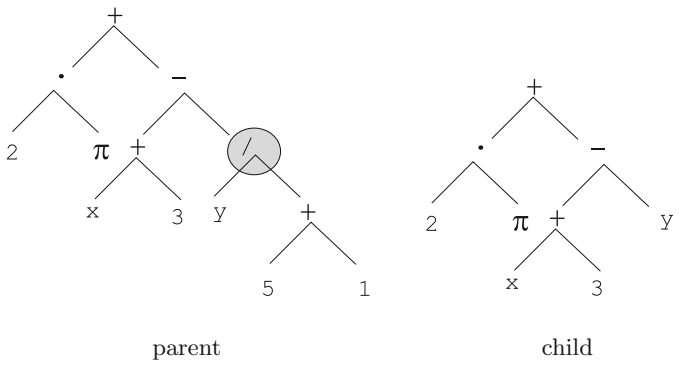

Fig. 4.20. Tree-based mutation illustrated: the node designated by a *circle* in the tree *on the left* is selected for mutation. The subtree staring at that node is replaced by a randomly generated tree, which is a leaf here

Tree-based mutation has two parameters:

- the probability of choosing mutation at the junction with recombination
- the probability of choosing an internal point within the parent as the root of the subtree to be replaced

It is remarkable that Koza's classic book on GP from 1992 [252] advises users to set the mutation rate at 0, i.e., it suggests that GP works *without* mutation. More recently Banzhaf et al. recommended 5% [37]. In giving mutation such a limited role, GP differs from other EA streams. The reason for this is the generally shared view that crossover has a large shuffling effect, acting in some sense as a macromutation operator [9]. The current GP practice uses low, but positive, mutation frequencies, even though some studies indicate that the common wisdom favouring an (almost) pure crossover approach might be misleading [275].

4.6.2 Recombination for Tree Representation

Tree-based recombination creates offspring by swapping genetic material among the selected parents. In technical terms, it is a binary operator creating two child trees from two parent trees. The most common implementation is **subtree crossover**, which works by interchanging the subtrees starting at two randomly selected nodes in the given parents. This is illustrated in Fig. 4.21. Note that the size (tree depth) of the children can exceed that of the parent trees. In this, recombination within GP differs from recombination in other EC dialects. Tree-based recombination has two parameters:

- the probability of choosing recombination at the junction with mutation
- the probability of choosing internal nodes as crossover points

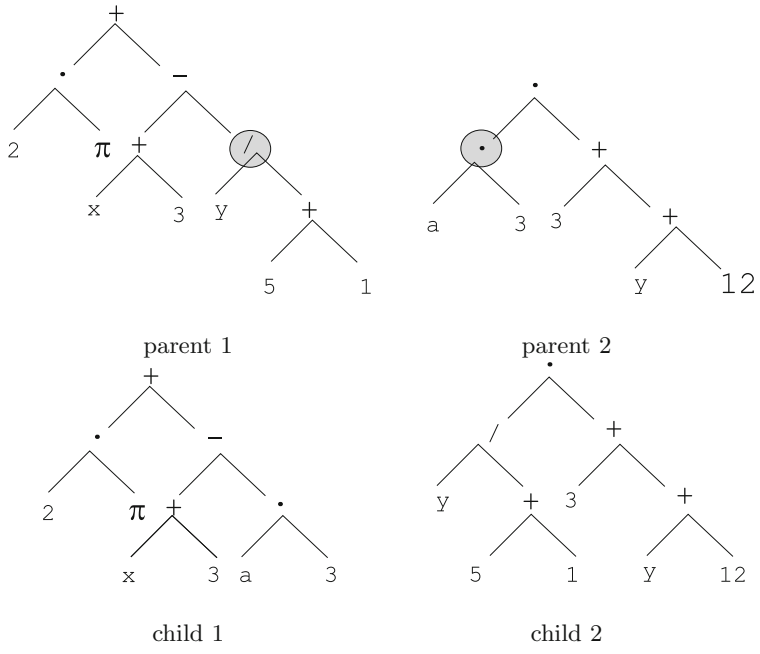

Fig. 4.21. Tree-based crossover illustrated: the nodes designated by a *circle* in the parent trees are selected to serve as crossover points. The subtrees staring at those nodes are swapped, resulting in two new trees, which are the children

For exercises and recommended reading for this chapter, please visit www.evolutionarycomputation.org.