3

What Is an Evolutionary Algorithm?

The most important aim of this chapter is to describe what an evolutionary algorithm (EA) is. In order to give a unifying view we present a general scheme that forms the common basis for all the different variants of evolutionary algorithms. The main components of EAs are discussed, explaining their role and related issues of terminology. This is immediately followed by two example applications to make things more concrete. We then go on to discuss general issues concerning the operation of EAs, to place them in a broader context and explain their relationship with other global optimisation techniques.

3.1 What Is an Evolutionary Algorithm?

As the history of the field suggests, there are many different variants of evolutionary algorithms. The common underlying idea behind all these techniques is the same: given a population of individuals within some environment that has limited resources, competition for those resources causes natural selection (survival of the fittest). This in turn causes a rise in the fitness of the population. Given a quality function to be maximised, we can randomly create a set of candidate solutions, i.e., elements of the function's domain. We then apply the quality function to these as an abstract fitness measure – the higher the better. On the basis of these fitness values some of the better candidates are chosen to seed the next generation. This is done by applying recombination and/or mutation to them. Recombination is an operator that is applied to two or more selected candidates (the so-called parents), producing one or more new candidates (the children). Mutation is applied to one candidate and results in one new candidate. Therefore executing the operations of recombination and mutation on the parents leads to the creation of a set of new candidates (the offspring). These have their fitness evaluated and then compete – based on their fitness (and possibly age) – with the old ones for a place in the next generation. This process can be iterated until a candidate

A.E. Eiben, J.E. Smith, *Introduction to Evolutionary Computing*,
Natural Computing Series, DOI 10.1007/978-3-662-44874-8_3

with sufficient quality (a solution) is found or a previously set computational limit is reached.

There are two main forces that form the basis of evolutionary systems:

- Variation operators (recombination and mutation) create the necessary diversity within the population, and thereby facilitate novelty.
- Selection acts as a force increasing the mean quality of solutions in the population.

The combined application of variation and selection generally leads to improving fitness values in consecutive populations. It is easy to view this process as if evolution is optimising (or at least 'approximising') the fitness function, by approaching the optimal values closer and closer over time. An alternative view is that evolution may be seen as a process of adaptation. From this perspective, the fitness is not seen as an objective function to be optimised, but as an expression of environmental requirements. Matching these requirements more closely implies an increased viability, which is reflected in a higher number of offspring. The evolutionary process results in a population which is increasingly better adapted to the environment.

It should be noted that many components of such an evolutionary process are stochastic. For example, during selection the best individuals are not chosen deterministically, and typically even the weak individuals have some chance of becoming a parent or of surviving. During the recombination process, the choice of which pieces from the parents will be recombined is made at random. Similarly for mutation, the choice of which pieces will be changed within a candidate solution, and of the new pieces to replace them, is made randomly. The general scheme of an **evolutionary algorithm** is given in pseudocode in Fig. 3.1, and is shown as a flowchart in Fig. 3.2.

```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```

Fig. 3.1. The general scheme of an evolutionary algorithm in pseudocode

It is easy to see that this scheme falls into the category of generate-and-test algorithms. The evaluation (fitness) function provides a heuristic estimate of solution quality, and the search process is driven by the variation and selection operators. Evolutionary algorithms possess a number of features that can help position them within the family of generate-and-test methods:

- EAs are population based, i.e., they process a whole collection of candidate solutions simultaneously.
- Most EAs use recombination, mixing information from two or more candidate solutions to create a new one.
- EAs are stochastic.

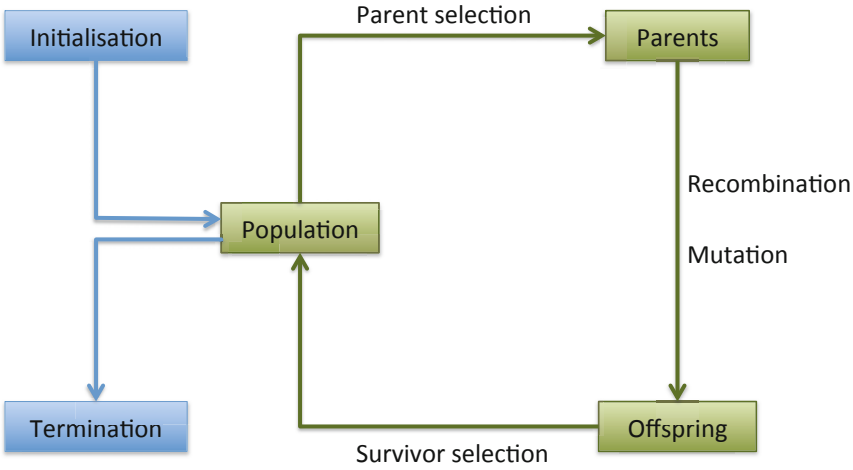

Fig. 3.2. The general scheme of an evolutionary algorithm as a flowchart

The various dialects of evolutionary computing we have mentioned previously all follow these general outlines, differing only in technical details. In particular, different streams are often characterised by the representation of a candidate solution – that is to say the data structures used to encode candidates. Typically this has the form of strings over a finite alphabet in genetic algorithms (GAs), real-valued vectors in evolution strategies (ESs), fi-

nite state machines in classical evolutionary programming (EP), and trees in genetic programming (GP). The origin of these differences is mainly historical. Technically, one representation might be preferable to others if it matches the given problem better; that is, it makes the encoding of candidate solutions easier or more natural. For instance, when solving a satisfiability problem with n logical variables, the straightforward choice is to use bit-strings of length n so that the contents of the ith bit would denote that variable i took the value $true$ (1) or $false$ (0). Hence, the appropriate EA would be a GA. To evolve a computer program that can play checkers, the parse trees of the syntactic expressions forming the programs are a natural choice to represent candidate solutions, thus a GP approach is likely. It is important to note two points. First, the recombination and mutation operators working on candidates must match the given representation. Thus, for instance, in GP the recombination operator works on trees, while in GAs it operates on strings. Second, in contrast to variation operators, the selection process only takes fitness information into account, and so it works independently from the choice of representation. Therefore differences between the selection mechanisms commonly applied in each stream are a matter of tradition rather than of technical necessity.

3.2 Components of Evolutionary Algorithms

In this section we discuss evolutionary algorithms in detail. There are a number of components, procedures, or operators that must be specified in order to define a particular EA. The most important components, indicated by italics in Fig. 3.1, are:

- representation (definition of individuals)
- evaluation function (or fitness function)
- population
- parent selection mechanism
- variation operators, recombination and mutation
- survivor selection mechanism (replacement)

To create a complete, runnable algorithm, it is necessary to specify each component and to define the initialisation procedure. If we wish the algorithm to stop at some stage[1], we must also provide a termination condition.

3.2.1 Representation (Definition of Individuals)

The first step in defining an EA is to link the 'real world' to the 'EA world', that is, to set up a bridge between the original problem context and the

[1] Note that this is not always this case. For instance, there are many examples of open-ended evolution of art on the Internet.

problem-solving space where evolution takes place. This often involves simplifying or abstracting some aspects of the real world to create a well-defined and tangible problem context within which possible solutions can exist and be evaluated, and this work is often undertaken by domain experts. The first step from the point of view of automated problem-solving is to decide how possible solutions should be specified and stored in a way that can be manipulated by a computer. We say that objects forming possible solutions within the original problem context are referred to as **phenotypes**, while their encoding, that is, the individuals within the EA, are called **genotypes**. This first design step is commonly called **representation**, as it amounts to specifying a mapping from the phenotypes onto a set of genotypes that are said to represent them. For instance, given an optimisation problem where the possible solutions are integers, the given set of integers would form the set of phenotypes. In this case one could decide to represent them by their binary code, so, for example, the value 18 would be seen as a phenotype, and 10010 as a genotype representing it. It is important to understand that the phenotype space can be very different from the genotype space, and that the whole evolutionary search takes place in the genotype space. A solution – a good phenotype – is obtained by decoding the best genotype after termination. Therefore it is desirable that the (optimal) solution to the problem at hand – a phenotype – is represented in the given genotype space. In fact, since in general we will not know in advance what that solution looks like, it is usually desirable that *all* possible feasible solutions can be represented[2].

The evolutionary computation literature contains many synonyms:

- On the side of the original problem context the terms **candidate solution**, phenotype, and **individual** are all used to denote possible solutions. The space of all possible candidate solutions is commonly called the **phenotype space**.
- On the side of the EA, the terms genotype, **chromosome**, and again individual are used to denote points in the space where the evolutionary search actually takes place. This space is often termed the **genotype space**.
- There are also many synonymous terms for the elements of individuals. A placeholder is commonly called a variable, a **locus** (plural: loci), a position, or – in a biology-oriented terminology – a **gene**. An object in such a place can be called a value or an **allele**.

It should be noted that the word 'representation' is used in two slightly different ways. Sometimes it stands for the mapping from the phenotype to the genotype space. In this sense it is synonymous with **encoding**, e.g., one could mention binary representation or binary encoding of candidate solutions. The inverse mapping from genotypes to phenotypes is usually called **decoding**, and it is necessary that the representation should be invertible so that for each

[2] In the language of generate-and-test algorithms, this means that the generator is *complete*.

genotype there is at most one corresponding phenotype. The word representation can also be used in a slightly different sense, where the emphasis is not on the mapping itself, but on the data structure of the genotype space. This interpretation is the one we use when, for example, we speak about mutation operators for binary representation.

3.2.2 Evaluation Function (Fitness Function)

The role of the **evaluation function** is to represent the requirements the population should adapt to meet. It forms the basis for selection, and so it facilitates improvements. More accurately, it defines what improvement means. From the problem-solving perspective, it represents the task to be solved in the evolutionary context. Technically, it is a function or procedure that assigns a quality measure to genotypes. Typically, this function is composed from the inverse representation (to create the corresponding phenotype) followed by a quality measure in the phenotype space. To stick with the example above, if the task is to find an integer x that maximises x^2, the fitness of the genotype 10010 could be defined by decoding its corresponding phenotype ($10010 \rightarrow 18$) and then taking its square: $18^2 = 324$.

The evaluation function is commonly called the **fitness function** in EC. This might cause a counterintuitive terminology if the original problem requires minimisation, because the term fitness is usually associated with maximisation. Mathematically, however, it is trivial to change minimisation into maximisation, and vice versa. Quite often, the original problem to be solved by an EA is an optimisation problem (treated in more technical detail in Sect. 1.1). In this case the name objective function is often used in the original problem context, and the evaluation (fitness) function can be identical to, or a simple transformation of, the given objective function.

3.2.3 Population

The role of the **population** is to hold (the representation of) possible solutions. A population is a multiset[3] of genotypes. The population forms the unit of evolution. Individuals are static objects that do not change or adapt; it is the population that does. Given a representation, defining a population may be as simple as specifying how many individuals are in it, that is, setting the population size. In some sophisticated EAs a population has an additional spatial structure, defined via a distance measure or a neighbourhood relation. This corresponds loosely to the way that real populations evolve within the context of a spatial structure given by individuals' geographical locations. In such cases the additional structure must also be defined in order to fully specify a population.

[3] A multiset is a set where multiple copies of an element are possible.

In almost all EA applications the population size is constant and does not change during the evolutionary search – this produces the limited resources need to create competition. The selection operators (parent selection and survivor selection) work at the population level. In general, they take the whole current population into account, and choices are always made relative to what is currently present. For instance, the best individual *of a given population* is chosen to seed the next generation, or the worst individual *of a given population* is chosen to be replaced by a new one. This population level activity is in contrast to variation operators, which act on one or more parent individuals.

The **diversity** of a population is a measure of the number of *different* solutions present. No single measure for diversity exists. Typically people might refer to the number of different fitness values present, the number of different phenotypes present, or the number of different genotypes. Other statistical measures such as entropy are also used. Note that the presence of only one fitness value in a population does not necessarily imply that only one phenotype is present, since many phenotypes may have the same fitness. Equally, the presence of only one phenotype does not necessarily imply only one genotype. However, if only one genotype is present then this implies only one phenotype and fitness value are present.

3.2.4 Parent Selection Mechanism

The role of **parent selection** or **mate selection** is to distinguish among individuals based on their quality, and, in particular, to allow the better individuals to become parents of the next generation. An individual is a **parent** if it has been selected to undergo variation in order to create offspring. Together with the survivor selection mechanism, parent selection is responsible for pushing quality improvements. In EC, parent selection is typically probabilistic. Thus, high-quality individuals have more chance of becoming parents than those with low quality. Nevertheless, low-quality individuals are often given a small, but positive chance; otherwise the whole search could become too greedy and the population could get stuck in a local optimum.

3.2.5 Variation Operators (Mutation and Recombination)

The role of **variation operators** is to create new individuals from old ones. In the corresponding phenotype space this amounts to generating new candidate solutions. From the generate-and-test search perspective, variation operators perform the generate step. Variation operators in EC are divided into two types based on their arity, distinguishing unary (mutation) and n-ary versions (recombination).

Mutation

A unary variation operator is commonly called **mutation**. It is applied to one genotype and delivers a (slightly) modified mutant, the **child** or **offspring**.

A mutation operator is always stochastic: its output – the child – depends on the outcomes of a series of random choices. It should be noted that not all unary operators are seen as mutation. For example, it might be tempting to use the term mutation to describe a problem-specific heuristic operator which acts systematically on one individual trying to find its weak spot and improve it by performing a small change. However, in general mutation is supposed to cause a random, unbiased change. For this reason it might be more appropriate not to call heuristic unary operators mutation. Historically, mutation has played a different role in various EC dialects. Thus, for example, in genetic programming it is often not used at all, whereas in genetic algorithms it has traditionally been seen as a background operator, providing the gene pool with 'fresh blood', and in evolutionary programming it is the only variation operator, solely responsible for the generation of new individuals.

Variation operators form the evolutionary implementation of elementary (search) steps, giving the search space its topological structure. Generating a child amounts to stepping to a new point in this space. From this perspective, mutation has a theoretical role as well: it can guarantee that the space is connected. There are theorems which state that an EA will (given sufficient time) discover the global optimum of a given problem. These often rely on this connectedness property that each genotype representing a possible solution can be reached by the variation operators [129]. The simplest way to satisfy this condition is to allow the mutation operator to jump everywhere: for example, by allowing any allele to be mutated into any other with a nonzero probability. However, many researchers feel these proofs have limited practical importance, and EA implementations often don't possess this property.

Recombination

A binary variation operator is called **recombination** or **crossover**. As the names indicate, such an operator merges information from two parent genotypes into one or two offspring genotypes. Like mutation, recombination is a stochastic operator: the choices of what parts of each parent are combined, and how this is done, depend on random drawings. Again, the role of recombination differs between EC dialects: in genetic programming it is often the only variation operator, and in genetic algorithms it is seen as the main search operator, whereas in evolutionary programming it is never used. Recombination operators with a higher arity (using more than two parents) are mathematically possible and easy to implement, but have no biological equivalent. Perhaps this is why they are not commonly used, although several studies indicate that they have positive effects on the evolution [126, 128].

The principle behind recombination is simple – by mating two individuals with different but desirable features, we can produce an offspring that combines both of those features. This principle has a strong supporting case – for millennia it has been successfully applied by plant and livestock breeders to

produce species that give higher yields or have other desirable features. Evolutionary algorithms create a number of offspring by random recombination, and we hope that while some will have undesirable combinations of traits, and most may be no better or worse than their parents, some will have improved characteristics. The biology of the planet Earth, where, with *very* few exceptions, lower organisms reproduce asexually and higher organisms reproduce sexually [288, 289], suggests that recombination is the superior form of reproduction. However recombination operators in EAs are usually applied probabilistically, that is, with a nonzero chance of not being performed.

It is important to remember that variation operators are representation dependent. Thus for different representations different variation operators have to be defined. For example, if genotypes are bit-strings, then inverting a bit can be used as a mutation operator. However, if we represent possible solutions by tree-like structures another mutation operator is required.

3.2.6 Survivor Selection Mechanism (Replacement)

Similar to parent selection, the role of **survivor selection** or **environmental selection** is to distinguish among individuals based on their quality. However, it is used in a different stage of the evolutionary cycle – the survivor selection mechanism is called after the creation of the offspring from the selected parents. As mentioned in Sect. 3.2.3, in EC the population size is almost always constant. This requires a choice to be made about which individuals will be allowed in to the next generation. This decision is often based on their fitness values, favouring those with higher quality, although the concept of age is also frequently used. In contrast to parent selection, which is typically stochastic, survivor selection is often deterministic. Thus, for example, two common methods are the fitness-based method of ranking the unified multiset of parents and offspring and selecting the top segment, or the age-biased approach of selecting only from the offspring.

Survivor selection is also often called the **replacement** strategy. In many cases the two terms can be used interchangeably, but we use the name survivor selection to keep terminology consistent: steps 1 and 5 in Fig. 3.1 are both named selection, distinguished by a qualifier. Equally, if the algorithm creates surplus children (e.g., 500 offspring from a population of 100), then using the term survivor selection is clearly appropriate. On the other hand, the term "replacement" might be preferred if the number of newly-created children is small compared to the number of individuals in the population. For example, a "steady-state" algorithm might generate two children per iteration from a population of 100. In this case, survivor selection means choosing the two old individuals that are to be deleted to make space for the new ones, so it is more efficient to declare that everybody survives unless deleted and to choose whom to replace. Both strategies can of course be seen in nature, and have their proponents in EC, so in the rest of this book we will be pragmatic about this issue. We will use survivor selection in the section headers for reasons of

generality and uniformity, while using replacement if it is commonly used in the literature for the given procedure we are discussing.

3.2.7 Initialisation

Initialisation is kept simple in most EA applications; the first population is seeded by randomly generated individuals. In principle, problem-specific heuristics can be used in this step, to create an initial population with higher fitness. Whether this is worth the extra computational effort, or not, very much depends on the application at hand. There are, however, some general observations concerning this question that we discuss in Sect. 3.5, and we also return to this issue in Chap. 10.

3.2.8 Termination Condition

We can distinguish two cases of a suitable **termination condition**. If the problem has a known optimal fitness level, probably coming from a known optimum of the given objective function, then in an ideal world our stopping condition would be the discovery of a solution with this fitness. If we know that our model of the real-world problem contains necessary simplifications, or may contain noise, we may accept a solution that reaches the optimal fitness to within a given precision $\epsilon > 0$. However, EAs are stochastic and mostly there are no guarantees of reaching such an optimum, so this condition might never get satisfied, and the algorithm may never stop. Therefore we must extend this condition with one that certainly stops the algorithm. The following options are commonly used for this purpose:

1. The maximally allowed CPU time elapses.
2. The total number of fitness evaluations reaches a given limit.
3. The fitness improvement remains under a threshold value for a given period of time (i.e., for a number of generations or fitness evaluations).
4. The population diversity drops under a given threshold.

Technically, the actual termination criterion in such cases is a disjunction: optimum value hit *or* condition X satisfied. If the problem does not have a known optimum, then we need no disjunction. We simply need a condition from the above list, or a similar one that is guaranteed to stop the algorithm. We will return to the issue of when to terminate an EA in Sect. 3.5.

3.3 An Evolutionary Cycle by Hand

To illustrate the working of an EA, we show the details of one selection–reproduction cycle on a simple problem after Goldberg [189], that of maximising the values of x^2 for integers in the range 0–31. To execute a full

evolutionary cycle, we must make design decisions regarding the EA components representation, parent selection, recombination, mutation, and survivor selection.

For the representation we use a simple five-bit binary encoding mapping integers (phenotypes) to bit-strings (genotypes). For parent selection we use a fitness proportional mechanism, where the probability p_i that an individual i in population P is chosen to be a parent is $p_i = f(i)/\sum_{j\in P} f(j)$. Furthermore, we can decide to replace the entire population in one go by the offspring created from the selected parents. This means that our survivor selection operator is very simple: all existing individuals are removed from the population and all new individuals are added to it without comparing fitness values. This implies that we will create as many offspring as there are members in the population. Given our chosen representation, the mutation and recombination operators can be kept simple. Mutation is executed by generating a random number (from a uniform distribution over the range $[0, 1]$) in each bit position, and comparing it to a fixed threshold, usually called the **mutation rate**. If the random number is below that rate, the value of the gene in the corresponding position is flipped. Recombination is implemented by the classic one-point crossover. This operator is applied to two parents and produces two children by choosing a random crossover-point along the strings and swapping the bits of the parents after this point.

String no.	Initial population	x Value	Fitness $f(x) = x^2$	$Prob_i$	Expected count	Actual count
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
Sum			1170	1.00	4.00	4
Average			293	0.25	1.00	1
Max			576	0.49	1.97	2

Table 3.1. The x^2 example, 1: initialisation, evaluation, and parent selection

After having made the essential design decisions, we can execute a full selection–reproduction cycle. Table 3.1 shows a random initial population of four genotypes, the corresponding phenotypes, and their fitness values. The cycle then starts with selecting the parents to seed the next generation. The fourth column of Table 3.1 shows the expected number of copies of each individual after parent selection, being $f_i/\bar{f}$, where $\bar{f}$ denotes the average fitness (displayed values are rounded up). As can be seen, these numbers are not integers; rather they represent a probability distribution, and the mating pool is created by making random choices to sample from this distribution. The

String no.	Mating pool	Crossover point	Offspring after xover	x Value	Fitness $f(x) = x^2$
1	0 1 1 0 | 1	4	0 1 1 0 0	12	144
2	1 1 0 0 | 0	4	1 1 0 0 1	25	625
2	1 1 | 0 0 0	2	1 1 0 1 1	27	729
4	1 0 | 0 1 1	2	1 0 0 0 0	16	256
Sum					1754
Average					439
Max					729

Table 3.2. The x^2 example, 2: crossover and offspring evaluation

String no.	Offspring after xover	Offspring after mutation	x Value	Fitness $f(x) = x^2$
1	0 1 1 0 0	1 1 1 0 0	26	676
2	1 1 0 0 1	1 1 0 0 1	25	625
2	1 1 0 1 1	1 1 0 1 1	27	729
4	1 0 0 0 0	1 0 1 0 0	18	324
Sum				2354
Average				588.5
Max				729

Table 3.3. The x^2 example, 3: mutation and offspring evaluation

column "Actual count" stands for the number of copies in the mating pool, i.e., it shows *one* possible outcome.

Next the selected individuals are paired at random, and for each pair a random point along the string is chosen. Table 3.2 shows the results of crossover on the given mating pool for crossover points after the fourth and second genes, respectively, together with the corresponding fitness values. Mutation is applied to the offspring delivered by crossover. Once again, we show one possible outcome of the random drawings, and Table 3.3 shows the hand-made 'mutants'. In this case, the mutations shown happen to have caused positive changes in fitness, but we should emphasise that in later generations, as the number of 1's in the population rises, mutation will be *on average* (but not always) deleterious. Although manually engineered, this example shows a typical progress: the average fitness grows from 293 to 588.5, and the best fitness in the population from 576 to 729 after crossover and mutation.

3.4 Example Applications

3.4.1 The Eight-Queens Problem

This is the problem of placing eight queens on a regular 8×8 chessboard so that no two of them can check each other. This problem can be naturally

generalised, yielding the N-queens problem described in Sect. 1.3. There are many classical artificial intelligence approaches to this problem, which work in a constructive, or incremental, fashion. They start by placing one queen, and after having placed n queens, they attempt to place the $(n+1)$th in a feasible position where the new queen does not check any others. Typically some sort of backtracking mechanism is applied; if there is no feasible position for the $(n+1)$th queen, the nth is moved to another position.

An evolutionary approach to this problem is drastically different in that it is not incremental. Our candidate solutions are complete (rather than partial) board configurations, which specify the positions of all eight queens. The phenotype space P is the set of all such configurations. Clearly, most elements of this space are infeasible, violating the condition of nonchecking queens. The quality $q(p)$ of any phenotype $p \in P$ can be simply quantified by the number of checking queen pairs. The lower this measure, the better a phenotype (board configuration), and a zero value, $q(p) = 0$, indicates a good solution. From this observation we can formulate a suitable objective function to be minimised, with a known optimal value. Even though we have not defined genotypes at this point, we can state that the fitness (to be maximised) of a genotype g that represents phenotype p is some inverse of $q(p)$. There are many possible ways of specifying what kind of inverse we wish to use here. For instance, $1/q(p)$ is an easy option, but has the disadvantage that attempting division by zero is a problem for many computing systems. We could circumvent this by watching for $q(p) = 0$ and saying that when this occurs we have a solution, or by adding a small value ϵ, i.e., $1/(q(p) + \epsilon)$. Other options are to use $-q(p)$ or $M - q(p)$, where M is a sufficiently large number to make all fitness values positive, e.g., $M \geq max\{q(p) \mid p \in P\}$. This fitness function inherits the property of q that it has a known optimum M.

To design an EA to search the space P we need to define a representation of phenotypes from P. The most straightforward idea is to use a matrix representation of elements of P directly as genotypes, meaning that we must design variation operators for these matrices. In this example, however, we define a more clever representation as follows. A genotype, or chromosome, is a permutation of the numbers $1, \ldots, 8$, and a given $g = \langle i_1, \ldots, i_8 \rangle$ denotes the (unique) board configuration, where the nth column contains exactly one queen placed on the i_nth row. For instance, the permutation $g = \langle 1, \ldots, 8 \rangle$ represents a board where the queens are placed along the main diagonal. The genotype space G is now the set of all permutations of $1, \ldots, 8$ and we also have defined a mapping $F : G \to P$.

It is easy to see that by using such chromosomes we restrict the search to board configurations where horizontal constraint violations (two queens on the same row) and vertical constraint violations (two queens on the same column) do not occur. In other words, the representation guarantees half of the requirements of a solution – what remains to be minimised is the number of diagonal constraint violations. From a formal perspective we have chosen a representation that is not surjective since only part of P can be obtained

by decoding elements of G. While in general this could carry the danger of missing solutions in P, in our present example this is not the case, since we know a priori that those phenotypes from $P \setminus F(G)$ can never be solutions.

The next step is to define suitable variation operators (mutation and crossover) for our representation, i.e., to work on genotypes that are permutations. The crucial feature of a suitable operator is that it does not lead out of the space G. In common parlance, the offspring of permutations must themselves be permutations. Later, in Sects. 4.5.1 and 4.5.2, we will discuss such operators in great detail. Here we only describe one suitable mutation and one crossover operator for the purpose of illustration. For mutation we can use an operator that randomly selects two positions in a given chromosome, and swaps the values found in those positions. A good crossover for permutations is less obvious, but the mechanism outlined in Fig. 3.3 will create two child permutations from two parents.

1. Select a random position, the crossover point, $i \in \{1, \ldots, 7\}$
2. Cut both parents into two segments at this position
3. Copy the first segment of parent 1 into child 1 and the first segment of parent 2 into child 2
4. Scan parent 2 from left to right and fill the second segment of child 1 with values from parent 2, skipping those that it already contains
5. Do the same for parent 1 and child 2

Fig. 3.3. 'Cut-and-crossfill' crossover

The important thing about these variation operators is that mutation causes a small undirected change, and crossover creates children that inherit genetic material from both parents. It should be noted though that there can be large performance differences between operators, e.g., an EA using mutation A might find a solution quickly, whereas one using mutation B might never find a solution. The operators we sketch here are not necessarily *efficient*; they merely serve as examples of operators that are *applicable* to the given representation.

The next step in setting up an EA is to decide upon the selection and population update mechanisms. We will choose a simple scheme for managing the population. In each evolutionary cycle we will select two parents, producing two children, and the new population of size n will contain the best n of the resulting $n + 2$ individuals (the old population plus the two new ones).

Parent selection (step 1 in Fig. 3.1) will be done by choosing five individuals randomly from the population and taking the best two as parents. This ensures a bias towards using parents with relatively high fitness. Survivor selection (step 5 in Fig. 3.1) checks which old individuals should be deleted to make

place for the new ones – provided the new ones are better. Following the naming convention discussed from Sect. 3.2.6 we define a replacement strategy. The strategy we will use merges the population and offspring, then ranks them according to fitness, and deletes the worst two.

To obtain a full specification we can decide to fill the initial population with randomly generated permutations, and to terminate the search when we find a solution, or when 10,000 fitness evaluations have elapsed, whichever happens sooner. Furthermore we can decide to use a population size of 100, and to use the variation operators with a certain frequency. For instance, we always apply crossover to the two selected parents and in 80% of the cases apply mutation to the offspring. Putting this all together, we obtain an EA as summarised in Table 3.4.

Representation	Permutations
Recombination	'Cut-and-crossfill' crossover
Recombination probability	100%
Mutation	Swap
Mutation probability	80%
Parent selection	Best 2 out of random 5
Survival selection	Replace worst
Population size	100
Number of offspring	2
Initialisation	Random
Termination condition	Solution or 10,000 fitness evaluations

Table 3.4. Description of the EA for the eight-queens problem

3.4.2 The Knapsack Problem

The 0–1 knapsack problem, a generalisation of many industrial problems, can be briefly described as follows. We are given a set of n items, each of which has attached to it some value v_i, and some cost c_i. The task is to select a subset of those items that maximises the sum of the values, while keeping the summed cost within some capacity C_{max}. Thus, for example, when packing a backpack for a round-the-world trip, we must balance likely utility of the items against the fact that we have a limited volume (the items chosen must fit in one bag), and weight (airlines impose fees for luggage over a given weight).

It is a natural idea to represent candidate solutions for this problem as binary strings of length n, where a 1 in a given position indicates that an item is included and a 0 that it is omitted. The corresponding genotype space G is the set of all such strings with size 2^n, which increases exponentially with the number of items considered. Using this G, we fix the representation

in the sense of data structure, and next we need to define the mapping from genotypes to phenotypes.

The first representation (in the sense of a mapping) that we consider takes the phenotype space P and the genotype space to be identical. The quality of a given solution p, represented by a binary genotype g, is thus determined by summing the values of the included items, i.e., $q(p) = \sum_{i=1}^{n} v_i \cdot g_i$. However, this simple representation leads us to some immediate problems. By using a one-to-one mapping between the genotype space G and the phenotype space P, individual genotypes may correspond to invalid solutions that have an associated cost greater than the capacity, i.e., $\sum_{i=1}^{n} c_i \cdot g_i > C_{max}$. This issue is typical of a class of problems that we return to in Chap. 13, and a number of mechanisms have been proposed for dealing with it.

The second representation that we outline here solves this problem by employing a decoder function, that breaks the one-to-one correspondence between the genotype space G and the solution space P. In essence, our genotype representation remains the same, but when creating a solution we read from left to right along the binary string, and keep a running tally of the cost of included items. When we encounter a value 1, we first check to see whether including the item would break our capacity constraint. In other words, rather than interpreting a value 1 as meaning *include this item*, we interpret it as meaning *include this item IF it does not take us over the cost constraint.* The effect of this scheme is to make the mapping from genotype to phenotype space many-to-one, since once the capacity has been reached, the values of all bits to the right of the current position are irrelevant, as no more items will be added to the solution. Furthermore, this mapping ensures that all binary strings represent valid solutions with a unique fitness (to be maximised).

Having decided on a fixed-length binary representation, we can now choose off-the-shelf variation operators from the GA literature, because the bit-string representation is 'standard' there. A suitable (but not necessarily optimal) recombination operator is the so-called one-point crossover, where we align two parents and pick a random point along their length. The two offspring are created by exchanging the tails of the parents at that point. We will apply this with 70% probability, i.e., for each pair of parents there is a 70% chance that we will create two offspring by crossover and 30% that the children will be just copies of the parents. A suitable mutation operator is so-called bit-flipping: in each position we invert the value with a small probability $p_m \in [0, 1)$.

In this case we will create the same number of offspring as we have members in our initial population. As noted above, we create two offspring from each two parents, so we will select that many parents and pair them randomly. We will use a tournament for selecting the parents, where each time we pick two members of the population at random (with replacement), and the one with the highest value $q(p)$ wins the tournament and becomes a parent. We will institute a generational scheme for survivor selection, i.e., all of the population in each iteration are discarded and replaced by their offspring.

Finally, we should consider initialisation (which we will do by random choice of 0 and 1 in each position of our initial population), and termination. In this case, we do not know the maximum value that we can achieve, so we will run our algorithm until no improvement in the fitness of the best member of the population has been observed for 25 generations.

We have already defined our crossover probability as 0.7; we will work with a population size of 500 and a mutation rate of $p_m = 1/n$, i.e., that will *on average* change one value in every offspring. Our evolutionary algorithm to tackle this problem can be specified as below in Table 3.5.

Representation	Binary strings of length n
Recombination	One-point crossover
Recombination probability	70%
Mutation	Each value inverted with independent probability p_m
Mutation probability p_m	$1/n$
Parent selection	Best out of random 2
Survival selection	Generational
Population size	500
Number of offspring	500
Initialisation	Random
Termination condition	No improvement in last 25 generations

Table 3.5. Description of the EA for the knapsack problem

3.5 The Operation of an Evolutionary Algorithm

Evolutionary algorithms have some rather general properties concerning how they work. To illustrate how an EA typically works, we will assume a one-dimensional objective function to be maximised. Figure 3.4 shows three stages of the evolutionary search, showing how the individuals might typically be distributed in the beginning, somewhere halfway, and at the end of the evolution. In the first stage directly after initialisation, the individuals are randomly spread over the whole search space (Fig. 3.4, left). After only a few generations this distribution changes: because of selection and variation operators the population abandons low-fitness regions and starts to climb the hills (Fig. 3.4, middle). Yet later (close to the end of the search, if the termination condition is set appropriately), the whole population is concentrated around a few peaks, some of which may be suboptimal. In principle it is possible that the population might climb the wrong hill, leaving all of the individuals positioned around a local but not global optimum. Although there is no universally accepted rigorous definition of the terms exploration and exploitation, these notions are often used to categorize distinct phases of the search

process. Roughly speaking, **exploration** is the generation of new individuals in as-yet untested regions of the search space, while **exploitation** means the concentration of the search in the vicinity of known good solutions. Evolutionary search processes are often referred to in terms of a trade-off between exploration and exploitation. Too much of the former can lead to inefficient search, and too much of the latter can lead to a propensity to focus the search too quickly (see [142] for a good discussion of these issues). **Premature convergence** is the well-known effect of losing population diversity too quickly, and getting trapped in a local optimum. This danger is generally present in evolutionary algorithms, and techniques to prevent it are discussed in Chap. 5.

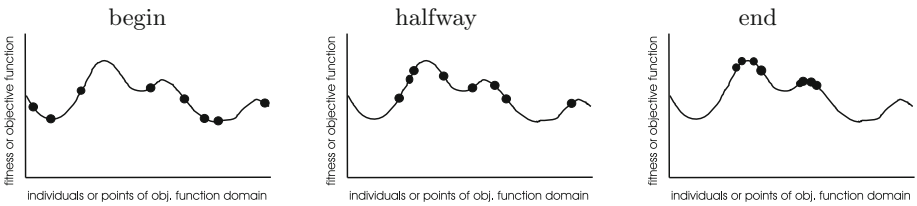

Fig. 3.4. Typical progress of an EA illustrated in terms of population distribution. For each point x in the search space y shows the corresponding fitness value.

The other effect we want to illustrate is the **anytime behaviour** of EAs by plotting the development of the population's best fitness value over time (Fig. 3.5). This curve shows rapid progress in the beginning and flattening out later on. This is typical for many algorithms that work by iterative improvements to the initial solution(s). The name 'anytime' comes from the property that the search can be stopped at any time, and the algorithm will have some solution, even if it is suboptimal. Based on this anytime curve we can

Fig. 3.5. Typical progress of an EA illustrated in terms of development over time of the highest fitness in the population

make some general observations concerning initialisation and the termination

condition for EAs. In Sect. 3.2.7 we questioned whether it is worth putting extra computational effort into applying intelligent heuristics to seed the initial population with better-than-random individuals. In general, it could be said that that the typical progress curve of an evolutionary process makes it unnecessary. This is illustrated in Fig. 3.6. As the figure indicates, using

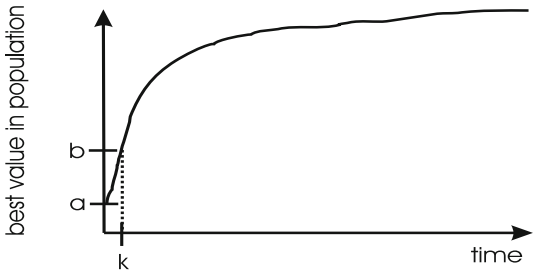

Fig. 3.6. Illustration of why heuristic initialisation might not be worth additional effort. Level a shows the best fitness in a randomly initialised population; level b belongs to heuristic initialisation

heuristic initialisation can start the evolutionary search with a better population. However, typically a few (k in the figure) generations are enough to reach this level, making the extra effort questionable. In Chap. 10 we will return to this issue.

The anytime behaviour also gives some general indications regarding the choice of termination conditions for EAs. In Fig. 3.7 we divide the run into two equally long sections. As the figure indicates, the progress in terms of fitness increase in the first half of the run (X) is significantly greater than in the second half (Y). This suggests that it might not be worth allowing very long runs. In other words, because of frequently observed anytime behaviour of EAs, we might surmise that effort spent after a certain time (number of fitness evaluations) is unlikely to result in better solution quality.

We close this review of EA behaviour by looking at EA performance from a global perspective. That is, rather than observing one run of the algorithm, we consider the performance of EAs for a wide range of problems. Fig. 3.8 shows the 1980s view after Goldberg [189]. What the figure indicates is that EAs show a roughly evenly good performance over a wide range of problems. This performance pattern can be compared to random search and to algorithms tailored to a specific problem type. EAs are suggested to clearly outperform random search. In contrast, a problem-tailored algorithm performs much better than an EA, but only on the type of problem for which it was designed. As we move away from this problem type to different problems, the problem-specific algorithm quickly loses performance. In this sense, EAs and problem-specific algorithms form two opposing extremes. This perception played an important

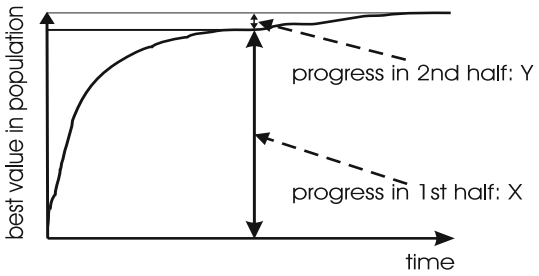

Fig. 3.7. Why long runs might not be worth performing. X shows the fitness increase in the first half of the run, while Y belongs to the second half

role in positioning EAs and stressing the difference between evolutionary and random search, but it gradually changed in the 1990s based on new insights from practice as well as from theory. The contemporary view acknowledges the possibility of combining the two extremes into a hybrid algorithm. This issue is treated in detail in Chap. 10, where we also present the revised version of Fig. 3.8. As for theoretical considerations, the No Free Lunch theorem has shown that (under some conditions) no black-box algorithm can outperform random walk when averaged over 'all' problems [467]. That is, showing the EA line always above that of random search is fundamentally incorrect. This is discussed further in Chap. 16.

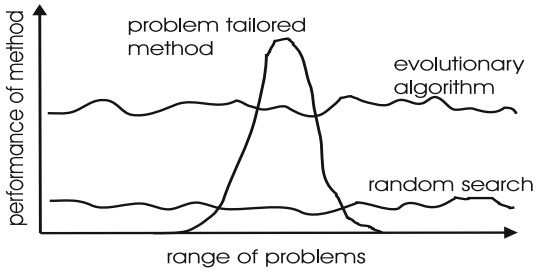

Fig. 3.8. 1980s view of EA performance after Goldberg [189]

3.6 Natural Versus Artificial Evolution

From the perspective of the underlying substrate, the emergence of evolutionary computation can be considered as a major transition of the evolutionary principles from wetware, the realm of biology, to software, the realm of computers. This was made possible by using computers as instruments for creating digital worlds that are very flexible and much more controllable than the

physical reality we live in. Together with the increased understanding of the genetic mechanisms behind evolution this brought about the opportunity to become active masters of evolutionary processes that are fully designed and executed by human experimenters from above.

It could be argued that evolutionary algorithms are not faithful models of natural evolution. However, they certainly are a form of evolution. As phrased by Dennett [116]: If you have variation, heredity, and selection, then you must get evolution. In Table 3.6 we compare natural evolution and artificial evolution as used in contemporary evolutionary algorithms.

	Natural evolution	Artificial evolution
Fitness	Observed quantity: *a posteriori* effect of selection ('in the eye of the observer').	Predefined *a priori* quantity that drives selection.
Selection	Complex multifactor force based on environmental conditions, other individuals of the same species and other species (e.g., predators). Viability is tested continually; reproducibility is tested at discrete times.	Randomized operator with selection probabilities based on given fitness values. Parent selection and survivor selection both happen at discrete times.
Genotype-phenotype mapping	Highly complex biochemical process influenced by the environment.	Relatively simple mathematical transformation or parameterised procedure.
Variation	Offspring created from one (asexual reproduction) or two parents (sexual reproduction).	Offspring may be generated from one, two, or many parents.
Execution	Parallel, decentralized execution; birth and death events are not synchronised.	Typically centralized with synchronised birth and death.
Population	Spatial embedding implies structured populations. Population size varies according to the relative number of death and birth events.	Typically unstructured and panmictic (all individuals are potential partners). Population size is kept constant by synchronising time and number of birth and death events.

Table 3.6. Differences between natural and artificial evolution

3.7 Evolutionary Computing, Global Optimisation, and Other Search Algorithms

In Chap. 2 we noted that evolutionary algorithms are often used for problem optimisation. Of course EAs are not the only optimisation technique known, so in this section we explain where EAs fall into the general class of optimisation methods, and why they are of increasing interest.

In an ideal world, we would possess the technology and algorithms that could provide a provably optimal solution to any problem that we could suitably pose to the system. In fact such algorithms do exist: an exhaustive enumeration of all of the possible solutions to a problem is clearly such an algorithm. Moreover, for many problems that can be expressed in a suitably mathematical formulation, much faster, exact techniques such as branch and bound search are well known. However, despite the rapid progress in computing technology, and even if there is no halt to Moore's Law, all too often the types of problems posed by users exceed in their demands the capacity of technology to answer them.

Decades of computer science research have taught us that many real-world problems can be reduced in their essence to well-known abstract forms, for which the number of potential solutions grows very quickly with the number of variables considered. For example, many problems in transportation can be reduced to the well-known travelling salesperson problem (TSP): given a list of destinations, construct the shortest tour that visits each destination exactly once. If we have n destinations, with symmetric distances between them, the number of possible tours is $n!/2 = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3$, which is exponential in n. For some of these abstract problems exact methods are known whose time complexity scales linearly (or at least polynomially) with the number of variables (see [212] for an overview). However, it is widely accepted that for many types of problems encountered, no such algorithms exist — as was discussed in Sect. 1.4. Thus, despite the increase in computing power, beyond a certain size of problem we must abandon the search for provably optimal solutions, and look to other methods for finding good solutions.

The term **global optimisation** refers to the process of attempting to find the solution with the optimal value for some fitness function. In mathematical terminology, we are trying to find the solution x^* out of a set of possible solutions S, such that $x \neq x^* \Rightarrow f(x^*) \geq f(x)\ \forall x \in S$. Here we have assumed a maximisation problem – the inequality is simply reversed for minimisation.

As noted above, a number of *deterministic* algorithms exist that, if allowed to run to completion, are guaranteed to find x^*. The simplest example is, of course, complete enumeration of all the solutions in S, which can take an exponentially long time as the number of variables increases. A variety of other techniques, collectively known as box decomposition, are based on ordering the elements of S into some kind of tree, and then reasoning about the quality of solutions in each branch in order to decide whether to investigate its elements. Although methods such as branch and bound can sometimes make very fast

progress, in the worst case (caused by searching in a suboptimal order) the time complexity of the algorithms is still the same as complete enumeration.

Another class of search methods is known as *heuristics*. These may be thought of as sets of rules for deciding which potential solution out of S should next be generated and tested. For some *randomised* heuristics, such as **simulated annealing** [2, 250] and certain variants of EAs, convergence proofs do in fact exist, i.e., they are guaranteed to find x^*. Unfortunately these algorithms are fairly weak, in the sense that they will not identify x^* as being globally optimal, rather as simply the best solution seen so far.

An important class of heuristics is based on the idea of using operators that impose some kind of structure onto the elements of S, such that each point x has associated with it a set of neighbours $N(x)$. In Fig. 2.2 the variables (traits) x and y were taken to be real-valued, which imposes a natural structure on S. The reader should note that for those types of problem where each variable takes one of a finite set of values (so-called **combinatorial optimisation**), there are many possible neighbourhood structures. As an example of how the landscape 'seen' by a local search algorithm depends on its neighbourhood structure, the reader might wish to consider what a chessboard would look like if we reordered it, so that squares that are possible next moves for the knight piece were adjacent to each other. Thus points which are locally optimal (fitter than all their neighbours) in the landscape induced by one neighbourhood structure may not be for another. However, by its definition, the **global optimum** x^* will always be fitter than all of its neighbours *under any neighbourhood structure*.

So-called **local search** algorithms [2] and their many variants work by taking a starting solution x, and then searching the candidate solutions in $N(x)$ for one x' that performs better than x. If such a solution exists, then this is accepted as the new incumbent solution, and the search proceeds by examining the candidate solutions in $N(x')$. This process will eventually lead to the identification of a **local optimum**: a solution that is superior to all those in its neighbourhood. Such algorithms (often referred to as **hill climbers** for maximisation problems) have been well studied over the decades. They have the advantage that they are often quick to identify a good solution to the problem, which is sometimes all that is required in practical applications. However, the downside is that problems will frequently exhibit numerous local optima, some of which may be significantly worse than the global optimum, and no guarantees can be offered for the quality of solution found.

A number of methods have been proposed to get around this problem by changing the search landscape, either by changing the neighbourhood structure (e.g., variable neighbourhood search [208]), or by temporarily assigning low fitness to already-seen good solutions (e.g., Tabu search [186]). However the theoretical basis behind these algorithms is still very much in gestation.

There are a number of features of EAs that distinguish them from local search algorithms, relating principally to their use of a population. The population provides the algorithm with a means of defining a nonuniform prob-

ability distribution function (p.d.f.) governing the generation of new points from S. This p.d.f. reflects possible interactions between points in S which are currently represented in the population. The interactions arise from the recombination of partial solutions from two or more members of the population (parents). This potentially complex p.d.f. contrasts with the globally uniform distribution of blind random search, and the locally uniform distribution used by many other stochastic algorithms such as simulated annealing and various hill-climbing algorithms.

The ability of EAs to maintain a diverse set of points provides not only a means of escaping from local optima, but also a means of coping with large and discontinuous search spaces. In addition, as will be seen in later chapters, if several copies of a solution can be generated, evaluated, and maintained in the population, this provides a natural and robust way of dealing with problems where there is noise or uncertainty associated with the assignment of a fitness score to a candidate solution.

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

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 ($r = l$).

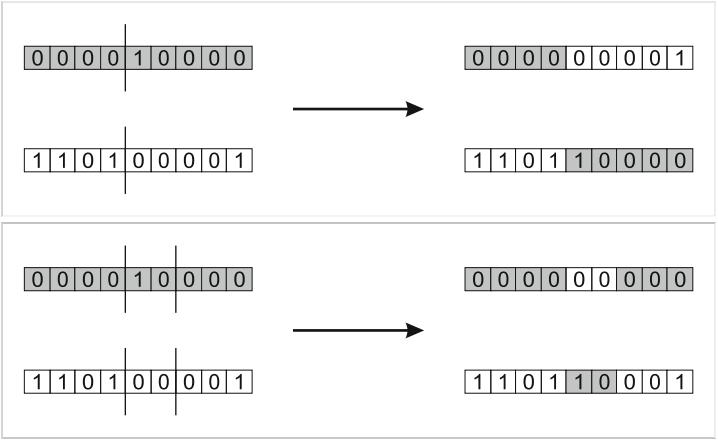

Fig. 4.2. One-point crossover (top) and n-point crossover with $n = 2$ (bottom)

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

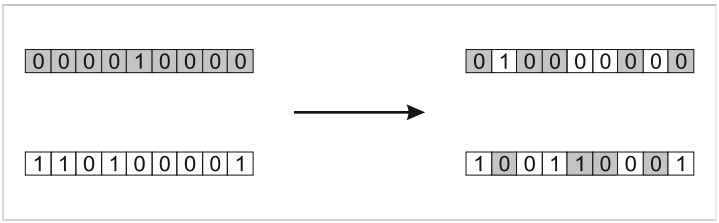

Fig. 4.3. Uniform crossover. The array [0.3, 0.6, 0.1, 0.4, 0.8, 0.7, 0.3, 0.5, 0.3] of random numbers and $p = 0.5$ were used to decide inheritance for this example.

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.

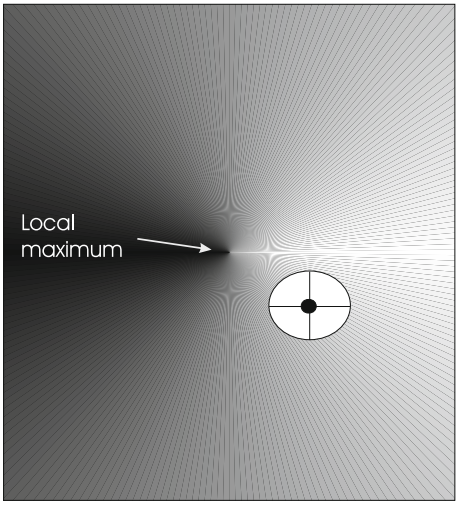

Fig. 4.4. Mutation with $n = 2, n_\sigma = 1, n_\alpha = 0$. Part of a fitness landscape with a *conical shape* is shown. The *black dot* indicates an individual. Points where the offspring can be placed with a given probability form a *circle*. The probability of moving along the y-axis (little effect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

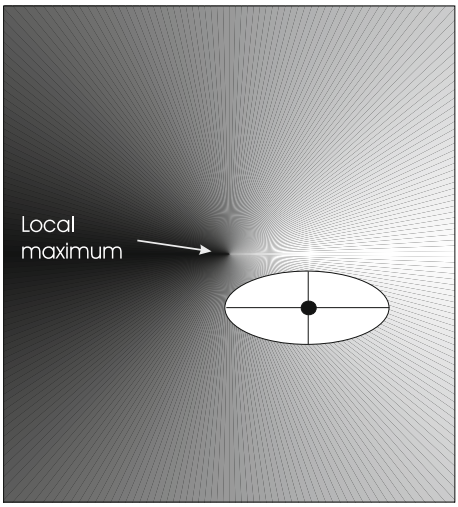

Fig. 4.5. Mutation with $n = 2, n_\sigma = 2, n_\alpha = 0$. Part of a fitness landscape with a *conical shape* is shown. The *black dot* indicates an individual. Points where the offspring can be placed with a given probability form an *ellipse*. The probability of moving along the x-axis (large effect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

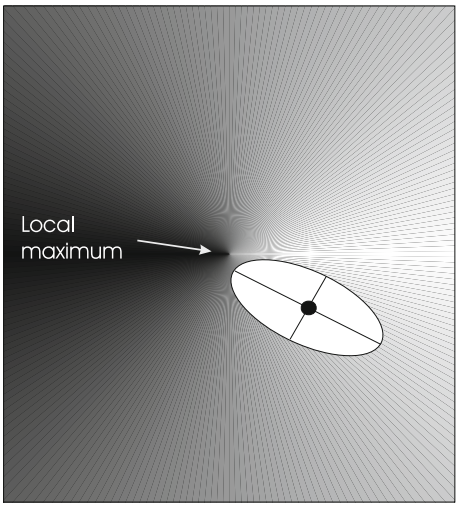

Fig. 4.6. Correlated mutation: $n = 2, n_\sigma = 2, n_\alpha = 1$. Part of a fitness landscape with a *conical shape* is shown. The *black dot* indicates an individual. Points where the offspring can be placed with a given probability form a *rotated ellipse*. The probability of generating a move in 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).

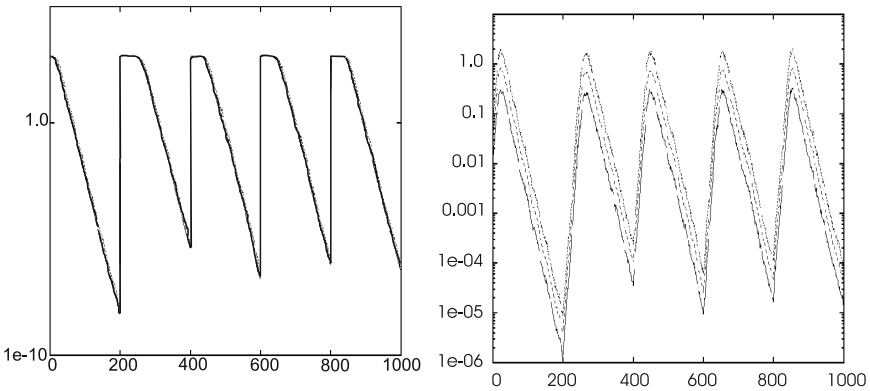

Fig. 4.7. Moving optimum ES experiment on the sphere function with $n = 30$, $n_\sigma = 1$. The location of the optimum is changed after every 200 generations (x-axes) with a clear effect on the average best objective function values (y-axis, left) in the given population. Self-adaptation is adjusting 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).

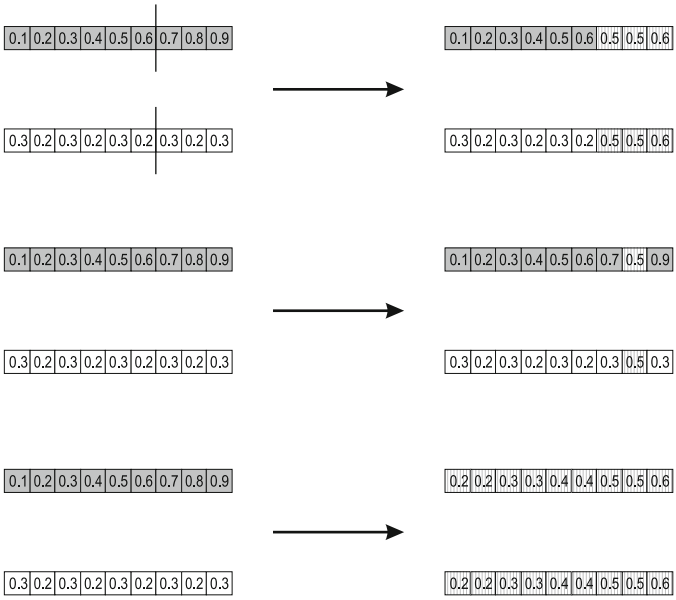

Fig. 4.8. Simple arithmetic recombination with $k = 6, \alpha = 1/2$ (top), single arithmetic recombination with $k = 8, \alpha = 1/2$ (middle), whole arithmetic recombination with $\alpha = 1/2$ (bottom).

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

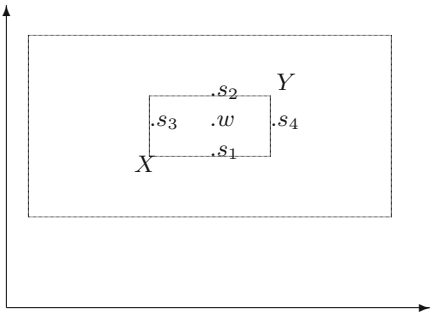

Fig. 4.9. Possible offspring from different recombination operators for two real-valued parents X and Y. $\{s_1, \ldots, s_4\}$ are the four possible offspring from single arithmetic recombination with $\alpha = 0.5$. w is the offspring from whole arithmetic recombination with $\alpha = 0.5$ and the *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.

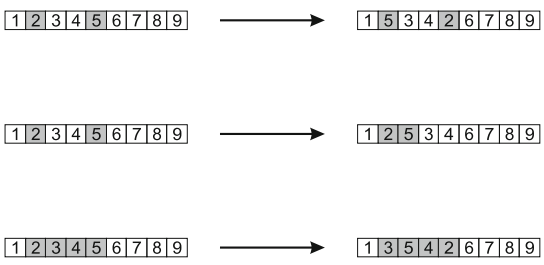

Fig. 4.10. Swap (top), insert (middle), and scramble mutation (bottom).

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.

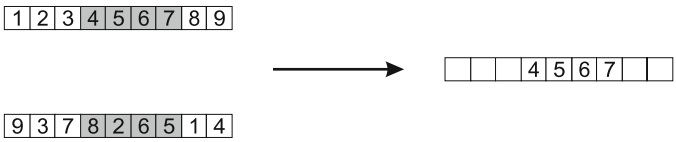

Fig. 4.12. PMX, step 1: copy randomly selected segment from first parent into offspring

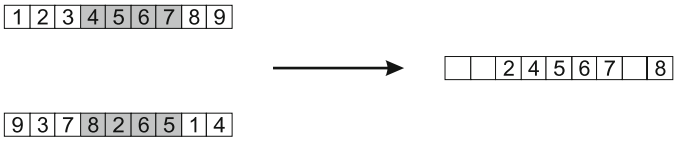

Fig. 4.13. PMX, step 2: consider in turn the placement of the elements that occur in the middle segment of parent 2 but not parent 1. The position that 8 takes in P2 is occupied by 4 in the offspring, so we can put the 8 into the position vacated by the 4 in P2. The position of the 2 in P2 is occupied 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.

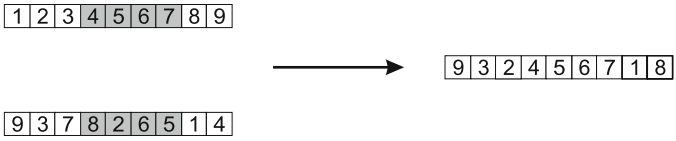

Fig. 4.14. PMX, step 3: copy remaining elements from second parent into same positions in offspring

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.

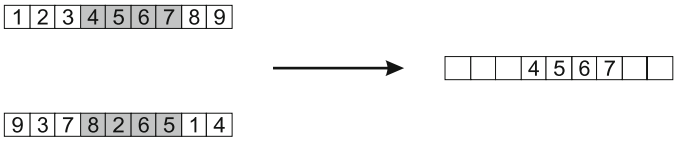

Fig. 4.15. Order crossover, step 1: copy randomly selected segment from first parent into offspring

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

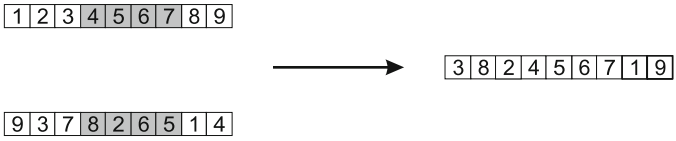

Fig. 4.16. Order crossover, step 2: copy rest of alleles in order they appear in second parent, treating string as toroidal

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.

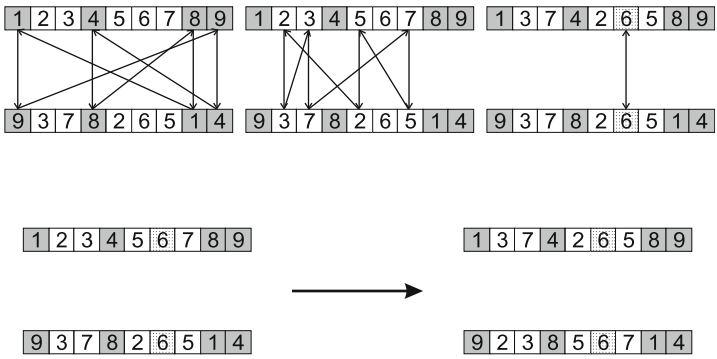

Fig. 4.17. Cycle crossover. Top: step 1- identification of cycles. Bottom: step 2- construction of offspring

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

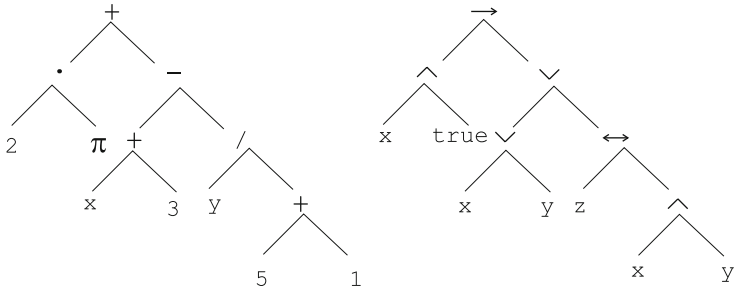

Fig. 4.18. Parse trees belonging to Eqs. (4.8) (left) and (4.9) (right)

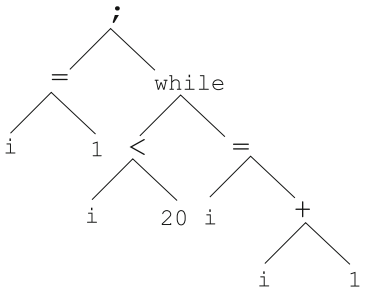

Fig. 4.19. Parse tree belonging to the above program

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.

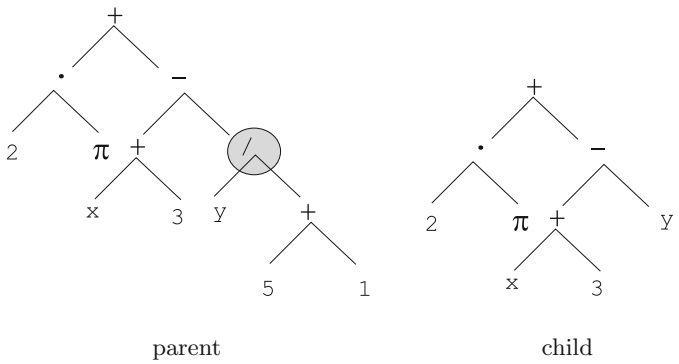

Fig. 4.20. Tree-based mutation illustrated: the node designated by a *circle* in the tree *on the left* is selected for mutation. The subtree staring at that node is replaced by a randomly generated tree, which is a leaf here

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

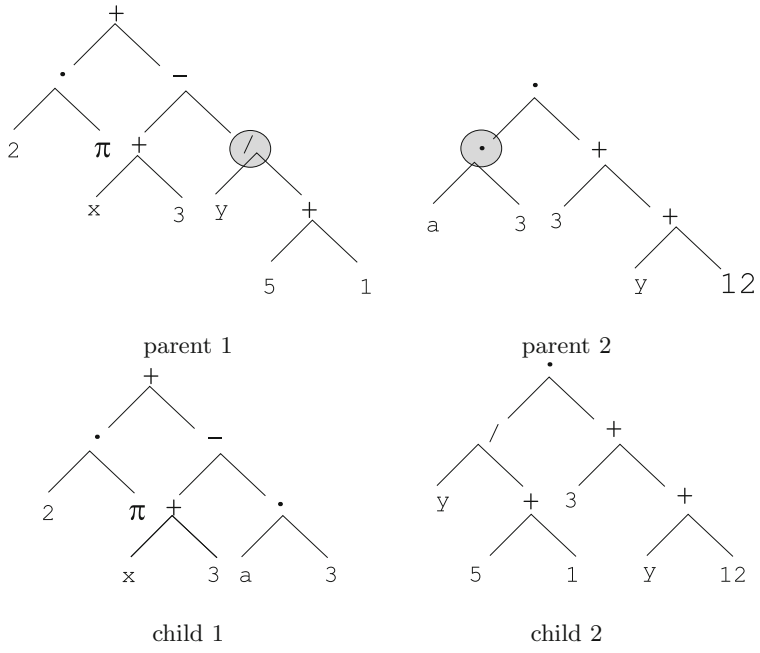

Fig. 4.21. Tree-based crossover illustrated: the nodes designated by a *circle* in the parent trees are selected to serve as crossover points. The subtrees staring at those nodes are swapped, resulting in two new trees, which are the children

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

5

Fitness, Selection, and Population Management

As explained in Chap. 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 second one. Having discussed some typical population management models, and selection operators, we then go on to explicitly look at some situations where diversity is needed, such as multimodal problems, and some approaches to population management, and altering selection, that have been proposed to increase useful diversity.

5.1 Population Management Models

In the previous chapter we have focused on the way that potential solutions are represented to give a population of diverse individuals, and on the way that variation (recombination and mutation) operators work on those individuals to yield offspring. These offspring will generally inherit some of their parents' properties but also differ slightly from them, providing new potential solutions to be evaluated. We now turn our attention to the second important element of the evolutionary process – the differential survival of individuals to compete for resources and take part in reproduction, based on their relative fitness.

Two different models of population management are found in the literature: the **generational model** and the **steady-state model**. The generational model is the one used in the example in Sect. 3.3. In each generation we begin with a population of size μ, from which a **mating pool** of parents is selected. Every member of the pool is a copy of something in the population, but the proportions will probably differ, with (usually) more copies of the 'better' parents. Next, λ offspring are created from the mating pool by the application of variation operators, and evaluated. After each generation, the whole population is replaced by μ individuals selected from its offspring, which is called the next generation. In the model typically used within the Simple Genetic Algorithm, the population, mating pool and offspring are all the same size, so that each generation is replaced by all of its offspring. This restriction

A.E. Eiben, J.E. Smith, *Introduction to Evolutionary Computing*,
Natural Computing Series, DOI 10.1007/978-3-662-44874-8_5

is not necessary: for example in the (μ, λ) Evolution Strategy, an excess of offspring is created (typically λ/μ is in the range 5–7) from which the next generation is selected on the basis of fitness.

In the steady-state model, the entire population is not changed at once, but rather a part of it. In this case, λ ($< \mu$) old individuals are replaced by λ new ones, the offspring. The proportion of the population that is replaced is called the **generational gap**, and is equal to λ/μ. Since its introduction in Whitley's GENITOR algorithm [460], the steady-state model has been widely studied and applied [105, 354, 442], often with $\lambda = 1$.

At this stage it is worth reiterating that the operators that are responsible for this competitive element of population management work on the basis of an individual's fitness. As a direct consequence, these selection and replacement operators work *independently* of the problem representation chosen. As was seen in the general description of an evolutionary algorithm at the start of Chap. 3, there are two points in the evolutionary cycle at which fitness-based competition can occur: during selection to take part in mating, and during the selection of individuals to survive into the next generation. We begin by describing the most commonly used methods for parent selection, but note that many of these can also be applied during the survival selection phase. As a final preliminary, please note that we will adopt a convention that we are trying to maximise fitness, and that fitness values are not negative. Often problems are expressed in terms of an objective function to be minimised, and sometimes negative fitness values occur. However, in all cases these can be mapped into the desired form by using an appropriate transformation.

5.2 Parent Selection

5.2.1 Fitness Proportional Selection

The principles of **fitness proportional selection** (FPS) were described in the simple example in Sect. 3.3. Recall that for each choice, the probability that an individual i is selected for mating depends on its *absolute* fitness value compared to the *absolute* fitness values of the rest of the population. Observing that the sum of the probabilities over the whole population must equal 1 the selection probability of individual i using FPS is $P_{FPS}(i) = f_i / \sum_{j=1}^{\mu} f_j$.

This selection mechanism was introduced in [220] and has been the topic of intensive study ever since, not least because it happens to be particularly amenable to theoretical analysis. However, it has been recognised that there are some problems with this selection mechanism:

- Outstanding individuals take over the entire population very quickly. This tends to focus the search process, and makes it less likely that the algorithm will thoroughly search the space of possible solutions, where better

solutions may exist. This phenomenon is often observed in early generations, when many of the randomly created individuals will have low fitness, and is known as **premature convergence**.

- When fitness values are all very close together, there is almost no **selection pressure**, so selection is almost uniformly random, and having a slightly better fitness is not very 'useful' to an individual. Therefore, later in a run, when some convergence has taken place and the worst individuals are gone, it is typically observed that the mean population fitness only increases very slowly.
- The mechanism behaves differently if the fitness function is transposed.

This last point is illustrated in Table 5.1, which shows three individuals and a fitness function with $f(A) = 1, f(B) = 4$, and $f(C) = 5$. Transposing this fitness function changes the selection probabilities, while the shape of the fitness landscape, and hence the location of the optimum, remains the same.

Individual	Fitness for f	Sel. prob. for f	Fitness for $f + 10$	Sel. prob. for $f + 10$	Fitness for $f + 100$	Sel. prob. for $f + 100$
A	1	0.1	11	0.275	101	0.326
B	4	0.4	14	0.35	104	0.335
C	5	0.5	15	0.375	105	0.339
Sum	10	1.0	40	1.0	310	1.0

Table 5.1. Transposing the fitness function changes selection probabilities for fitness-proportionate selection

To avoid the second two problems with FPS, a procedure known as **windowing** is often used. Under this scheme, fitness differentials are maintained by subtracted from the raw fitness $f(x)$ a value β^t, which depends in some way on the recent search history, and so can change over time (hence the superscript t). The simplest approach is just to subtract the value of the least-fit member of the current population P^t by setting $\beta^t = min_{y \in P^t} f(y)$ This value may fluctuate quite rapidly, so one alternative is to use a running average over the last few generations.

Another well-known approach is **sigma scaling** [189], which incorporates information about the mean $\bar{f}$ and standard deviation σ_f of fitnesses in the population:

$$f'(x) = max(f(x) - (\bar{f} - c \cdot \sigma_f), 0),$$

where c is a constant value, usually set to 2.

5.2.2 Ranking Selection

Rank-based selection is another method that was inspired by the observed drawbacks of fitness proportionate selection [32]. It preserves a constant selection pressure by sorting the population on the basis of fitness, and then

allocating selection probabilities to individuals according to their rank, rather than according to their actual fitness values. Let us assume that the ranks are numbered so that an individual's rank notes how many worse solutions are in the population, so the best has rank μ-1 and the worst has rank 0. The mapping from rank number to selection probability can be done in many ways, for example, linearly or exponentially decreasing. As with FPS above, and any selection scheme, we insist that the sum over the population of the selection probabilities must be unity – that we must select *one* of the parents.

The usual formula for calculating the selection probability for linear ranking schemes is parameterised by a value s ($1 < s \leq 2$). In the case of a generational EA, where $\mu = \lambda$, this can be interpreted as the *expected* number of offspring allotted to the fittest individual. Since this individual has rank $\mu - 1$, and the worst has rank 0, then the selection probability for an individual of rank i is:

$$P_{lin-rank}(i) = \frac{(2-s)}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}.$$

Note that the first term will be constant for all individuals (it is there to ensure the probabilities add to one). Since the second term will be zero for the worst individual (with rank $i = 0$), it can be thought of as the 'baseline' probability of selecting that individual.

In Table 5.2 we show an example of how the selection probabilities differ for a population of $\mu = 3$ different individuals with fitness proportionate and rank-based selection with different values of s.

Individual	Fitness	Rank	P_{selFP}	P_{selLR} $(s = 2)$	P_{selLR} $(s = 1.5)$
A	1	0	0.1	0	0.167
B	4	1	0.4	0.33	0.33
C	5	2	0.5	0.67	0.5
Sum	10		1.0	1.0	1.0

Table 5.2. Fitness proportionate (FP) versus linear ranking (LR) selection

When the mapping from rank to selection probabilities is linear, only limited selection pressure can be applied. This arises from the assumption that, on average, an individual of median fitness should have one chance to be reproduced, which in turn imposes a maximum value of $s = 2$. (Since the scaling is linear, letting $s > 2$ would require the worst to have a negative selection probability if the probabilities are to sum to unity.) If a higher selection pressure is required, i.e., more emphasis on selecting individuals of above-average fitness, an exponential ranking scheme is often used, of the form:

$$P_{exp-rank}(i) = \frac{1 - e^{-i}}{c}.$$

The normalisation factor c is chosen so that the sum of the probabilities is unity, i.e., it is a function of the population size.

5.2.3 Implementing Selection Probabilities

The description above provides two alternative schemes for deciding a probability distribution that defines the likelihood of each individual in the population being selected for reproduction. In an ideal world, the mating pool of parents taking part in recombination would have exactly the same proportions as this selection probability distribution. This would mean that the number of any given individual would be given by its selection probability, multiplied by the size of the mating pool. However, in practice this is not possible because of the finite size of the population, i.e., when we do this multiplication, we find typically that some individuals have an *expected* number of copies which is noninteger – whereas of course in practice we need to select complete individuals. In other words, the mating pool of parents is *sampled* from the selection probability distribution, but will not in general accurately reflect it, as was seen in the example in Sect. 3.3.

The simplest way of achieving this sampling is known as the **roulette wheel** algorithm. Conceptually this is the same as repeatedly spinning a one-armed roulette wheel, where the sizes of the holes reflect the selection probabilities. In general, the algorithm can be applied to select λ members from the set of μ parents into a mating pool. To illustrate the workings of this algorithm, we will assume some order over the population (ranking or random) from 1 to μ, so that we can calculate the *cumulative probability distribution*, which is a list of values $[a_1, a_2, \ldots, a_\mu]$ such that $a_i = \sum_1^i P_{sel}(i)$, where $P_{sel}(i)$ is defined by the selection distribution — fitness proportionate or ranking. Note that this implies $a_\mu = 1$. The outlines of the algorithm are given in Fig. 5.1.

```
BEGIN
  /*  Given the cumulative probability distribution a */
  /*  and assuming we wish to select λ members of the mating pool */
  set current_member = 1;
  WHILE ( current_member ≤ λ ) DO
    Pick a random value r uniformly from [0,1];
    set i = 1;
    WHILE (  a_i < r ) DO
      set i = i + 1;
    OD
    set mating_pool[current_member] = parents[i];
    set current_member = current_member + 1;
  OD
END
```

Fig. 5.1. Pseudocode for the roulette wheel algorithm

Despite its inherent simplicity, it has been recognised that the roulette wheel algorithm does not in fact give a particularly good sample of the required distribution. Whenever more than one sample is to be drawn from the distribution – for instance λ – the use of the **stochastic universal sampling** (SUS) algorithm [32] is preferred. Conceptually, this is equivalent to making one spin of a wheel with λ equally spaced arms, rather than λ spins of a one-armed wheel. Given the same list of cumulative selection probabilities $[a_1, a_2, \ldots, a_\mu]$, it selects the mating pool as described in Fig. 5.2.

```
BEGIN
  /*  Given the cumulative probability distribution a */
  /*  and assuming we wish to select λ members of the mating pool */
  set current_member = i = 1;
  Pick a random value r uniformly from [0, 1/λ];
  WHILE ( current_member ≤ λ ) DO
    WHILE (  r ≤ a[i] ) DO
      set mating_pool[current_member] = parents[i];
      set r = r + 1/λ;
      set current_member = current_member + 1;
    OD
    set i = i + 1;
  OD
END
```

Fig. 5.2. Pseudocode for the stochastic universal sampling algorithm making λ selections

Since the value of the variable r is initialised in the range $[0, 1/\lambda]$ and increases by an amount $1/\lambda$ every time a selection is made, it is guaranteed that the number of copies made of each parent i is at least the integer part of $\lambda \cdot P_{sel}(i)$ and is no more than one greater. Finally, we should note that with minor changes to the code, SUS can be used to make any number of selections from the parents, and in the case of making just one selection, it is the same as the roulette wheel.

5.2.4 Tournament Selection

The previous two selection methods and the algorithms used to sample from their probability distributions relied on a knowledge of the entire population. However, in certain situations, for example, if the population size is very large, or if the population is distributed in some way (perhaps on a parallel system), obtaining this knowledge is either highly time consuming or at worst impossible. Furthermore, both methods assume that fitness is a quantifiable

measure (based on some explicit objective function to be optimised), which may not be valid. Think, for instance, of an application evolving game playing strategies. In this case we might not be able to quantify the strength of a given individual (strategy) in isolation, but we can compare any two of them by simulating a game played by these strategies as opponents. Similar situations occur also in evolutionary design and evolutionary art applications [48, 49]. In these the user typically makes a subjective selection by comparing individuals representing designs or pieces of art, rather than using a quantitative measure to assign fitness, cf. Sect. 14.1.

Tournament selection is an operator with the useful property that it does not require any global knowledge of the population, nor a quantifiable measure of quality. Instead it only relies on an ordering relation that can compare and rank any two individuals. It is therefore conceptually simple and fast to implement and apply. The application of tournament selection to select λ members of a pool of μ individuals works according to the procedure shown in Fig. 5.3.

```
BEGIN
  /* Assume we wish to select λ members of a pool of μ individuals */
  set current_member = 1;
  WHILE ( current_member ≤ λ ) DO
    Pick k individuals randomly, with or without replacement;
    Compare these k individuals and select the best of them;
    Denote this individual as i;
    set mating_pool[current_member] = i;
    set current_member = current_member + 1;
  OD
END
```

Fig. 5.3. Pseudocode for the tournament selection algorithm

Because tournament selection looks at relative rather than absolute fitness, it has the same properties as ranking schemes in terms of invariance to translation and transposition of the fitness function. The probability that an individual will be selected as the result of a tournament depends on four factors, namely:

- Its rank in the population. Effectively this is estimated without the need for sorting the whole population.
- The **tournament size** k. The larger the tournament, the greater the chance that it will contain members of above-average fitness, and the less that it will consist entirely of low-fitness members. Thus the probability of selecting a high-fitness member increases, and that of selecting a low-

fitness member decreases, as k is increased. Hence we say that increasing k increases the selection pressure.

- The probability p that the most fit member of the tournament is selected. Usually this is 1 (*deterministic tournaments*), but stochastic versions are also used with $p < 1$. Since this makes it more likely that a less-fit member will be selected, decreasing p will decrease the selection pressure.
- Whether 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, since the other member of the tournament will be fitter. However, if the tournament candidates are picked with replacement, it is always possible for even the least-fit member of the population to be selected, since with probability $1/\mu^k > 0$ all tournament candidates will be copies of that member.

These properties of tournament selection were characterised in [20, 58], and it was shown [190] that for binary ($k = 2$) tournaments with parameter p the expected time for a single individual of high fitness to take over the population is the same as that for linear ranking with $s = 2p$. However, since λ tournaments are required to produce λ selections, it suffers from the same problems as the roulette wheel algorithm, in that the outcomes can show a high variance from the theoretical probability distribution. Despite this drawback, tournament selection is perhaps the most widely used selection operator in some EC dialects (in particular, Genetic Algorithms), due to its extreme simplicity and the fact that the selection pressure is easy to control by varying the tournament size k.

5.2.5 Uniform Parent Selection

In some dialects of EC it is common to use mechanisms such that each individual has the same chance to be selected. At first sight this might appear to suggest that there is no selection pressure in the algorithm, which would indeed be true if this was not coupled with a strong fitness-based survivor selection mechanism.

In Evolutionary Programming, usually there is no recombination, only mutation, and parent selection is deterministic. In particular, each parent produces exactly one child by mutation. Evolution Strategies are also usually implemented with uniform random selection of parents into the mating pool, i.e., for each $1 \leq i \leq \mu$ we have $P_{uniform}(i) = 1/\mu$.

5.2.6 Overselection for Large Populations

In some cases it may be desirable to work with extremely large populations. Sometimes this could be for technical reasons – for example, there has been a lot of interest in implementing EAs using graphics cards (GPUs), which offer similar speed-up to clusters or supercomputers, but at much lower cost.

However, achieving the maximum potential speed-up typically depends on having a large population on each processing node.

Regardless of the implementation details, if the potential search space is enormous it might be a good idea to use a large population to avoid 'missing' promising regions in the initial random generation, and thereafter to maintain the diversity needed to support exploration. For example, in Genetic Programming it is not unusual to use population sizes of several thousands: in 1994 [254] used 1000; in 1996 [7] used 128,000; and in 1999 [255] used 1,120,000 individuals. In the latter case, often a method called **over-selection** is used for population sizes of 1000 and above.

In this method, the population is first ranked by fitness and then divided into two groups, the top $x\%$ in one and the remaining $(100-x)\%$ in the other. When parents are selected, 80% of the selection operations choose from the first group, and the other 20% from the second. Koza [252] provides rule of thumb values for x depending on the population size as shown in Table 5.3. As can be seen, the number of individuals from which the majority of parents are chosen stays constant, i.e., the selection pressure increases dramatically for larger populations.

Population size	Proportion of population in fitter group (x)
1000	32%
2000	16%
4000	8%
8000	4%

Table 5.3. Rule of thumb values for overselection: Proportion of ranked population in fitter subpopulation from which majority of parents are selected

5.3 Survivor Selection

The survivor selection mechanism is responsible for managing the process of reducing the working memory of the EA from a set of μ parents and λ offspring to a set of μ individuals forming the next generation. In principle, any of the mechanisms introduced for parent selection could be also used for selecting survivors. However, over the history of EC a number of special survivor selection strategies have been suggested and are widely used.

As explained in Sect. 3.2.6, this step in the main evolutionary cycle is also called replacement. In the present section we often use this latter term to be consistent with the literature. Replacement strategies can be categorised according to whether they discriminate on the basis of the fitness or the age of individuals.

5.3.1 Age-Based Replacement

The basis of these schemes is that the fitness of individuals is not taken into account during the selection of which individuals to replace in the population. Instead, they are designed so that each individual exists in the population for the same number of EA iterations. This does not preclude the possibly that *copies* of highly-fit individuals might persist in the population, but for this to happen they must be chosen at least once in the selection phase and then survive the recombination and mutation stages without being modified. Note that since fitness is not taken into account, the mean, and even best fitness of any given generation, may be lower than that of its predecessor. While slightly counterintuitive, this is not a problem as long as it does not happen too often, and may even be beneficial if the population is concentrated around a local optimum. A net increase in the mean fitness over time therefore relies on (i) having sufficient selection pressure when selecting parents into the mating pool, and (ii) using variation operators that are not too disruptive.

Age-based replacement is the strategy used in the simple Genetic Algorithm. Since the number of offspring produced is the same as the number of parents ($\mu = \lambda$), each individual exists for just one cycle, and the parents are simply discarded, to be replaced by the entire set of offspring. This is the generational model, but in fact this replacement strategy can also be implemented in a steady-state with overlapping populations ($\lambda < \mu$), right to the other extreme where a single offspring is created and inserted in the population in each cycle. In this case the strategy takes the form of a first-in-first-out (FIFO) queue.

An alternative method of age-based replacement for steady-state GAs is to randomly select a parent for replacement. A straightforward mathematical argument based on the population size being fixed tells us that this probabilistic strategy has the same mean effect – that is, *on average* individuals live for μ iterations. De Jong and Sarma [105] investigated this strategy experimentally, and found that the algorithm showed higher variance in performance than a comparable generational GA. Smith and Vavak [400] showed that this was because the random strategy is far more likely to lose the best member of the population than a delete-oldest (FIFO) strategy. For these reasons the random replacement strategy is not recommended.

5.3.2 Fitness-Based Replacement

A wide number of strategies based on fitness have been proposed for choosing which μ of the μ parents + λ offspring should go forward to the next generation. Some also take age into account.

Replace worst (GENITOR) In this scheme the worst λ members of the population are selected for replacement. Although this can lead to very rapid improvements in the mean population fitness, it can also lead to premature convergence as the population tends to rapidly focus on the fittest member

currently present. For this reason it is commonly used in conjunction with large populations and/or a "no duplicates" policy.

Elitism This scheme is commonly used in conjunction with age-based and stochastic fitness-based replacement schemes, to prevent the loss of the current fittest member of the population. In essence a trace is kept of the current fittest member, and it is always kept in the population. Thus if it is chosen in the group to be replaced, and none of the offspring being inserted into the population has equal or better fitness, then it is kept and one of the offspring is discarded.

Round-robin tournament This mechanism was introduced within Evolutionary Programming, where it is applied to choose μ survivors. However, in principle, it can also be used to select λ parents from a given population of μ. The method works by holding pairwise tournament competitions in round-robin format, where each individual is evaluated against q others randomly chosen from the merged parent and offspring populations. For each comparison, a "win" is assigned if the individual is better than its opponent. After finishing all tournaments, the μ individuals with the greatest number of wins are selected. Typically, $q = 10$ is recommended in Evolutionary Programming. It is worth noting that this stochastic variant of selection allows for less-fit solutions to be selected if they had a lucky draw of opponents. As the value of q increases this chance becomes more and unlikely, until in the limit it becomes deterministic $\mu + \mu$.

$(\mu + \lambda)$ **Selection** The name and the notation of the $(\mu + \lambda)$ selection comes from Evolution Strategies. In general, it refers to the case where the set of offspring and parents are merged and ranked according to (estimated) fitness, then the top μ are kept to form the next generation. This strategy can be seen as a generalisation of the GENITOR method $(\mu > \lambda)$ and the round-robin tournament in Evolutionary Programming $(\mu = \lambda)$. In Evolution Strategies $\lambda > \mu$ with a great offspring surplus (typically $\lambda/\mu \approx 5 - 7$) that induces a large selection pressure.

(μ, λ) **Selection** The (μ, λ) strategy used in Evolution Strategies where typically $\lambda > \mu$ children are created from a population of μ parents. This method works on a mixture of age and fitness. The age component means that all the parents are discarded, so no individual is kept for more than one generation (although of course *copies* of it might exist later). The fitness component comes from the fact that the λ offspring are ranked according to the fitness, and the best μ form the next generation.

In Evolution Strategies, (μ, λ) selection, is generally preferred over $(\mu + \lambda)$ selection for the following reasons:

- The (μ, λ) discards all parents and is therefore in principle able to leave (small) local optima. This may be advantageous in a multimodal search space with many local optima.
- If the fitness function is not fixed, but changes in time, the $(\mu+\lambda)$ selection preserves outdated solutions, so it is not able to follow the moving optimum well.
- $(\mu + \lambda)$ selection hinders the self-adaptation mechanism used to adapt strategy parameters, cf. Sect. 6.2.

5.4 Selection Pressure

Throughout this chapter we have referred rather informally to the notion of selection pressure, using an intuitive description that as selection pressure increases, so fitter solutions are more likely to survive, or be chosen as parents, and less-fit solutions are correspondingly less likely.

A number of measures have been proposed for quantifying this, and studied theoretically, of which the best known is the takeover time. The **takeover time** τ^* of a given selection mechanism is defined as the number of generations it takes until the application of selection completely fills the population with copies of the best individual, given one copy initially. Goldberg and Deb [190] showed that

$$\tau^* = \frac{\ln \lambda}{\ln(\lambda/\mu)}.$$

For a typical evolution strategy with $\mu = 15$ and $\lambda = 100$, this results in $\tau^* \approx 2$. For fitness proportional selection in a genetic algorithm it is

$$\tau^* = \lambda \ln \lambda,$$

resulting in $\tau^* = 460$ for population size $\lambda = 100$.

Other authors have extended this analysis to other strategies in generational and steady-state population models [79, 400]; Rudolph applied it to different population structures such as rings [360], and also to consider a range of other measures of selection operators' performance, such as the 'Diversity Indicator'. Other measures of selection pressure have been proposed, including the 'Expected Loss of Diversity' [310], which is the expected change in the number of diverse solutions after μ selection events; and from theoretical biology, the 'Selection Intensity', which is the expected relative increase in mean population fitness after applying a selection operator.

While these measures can help in understanding the effect of different strategies, they can also be rather misleading since they consider selection alone, rather than in the context of variation operators providing diversity. Smith [390] derived mathematical expressions for a number of these indicators considering a wide range of replacement strategies in steady-state EAs. Experiments bore out the analytic results, and a benchmark comparison using well-known test problems showed that both the mean and variance of the takeover

time could correctly predict the *relative* ordering of the mean and variance of the time taken to first locate the global optimum. However, for many applications of EAs the most important measure is the quality of the best solution found and also possibly the diversity of good solutions discovered. Smith's results showed that in fact none of the theoretical measures were particularly indicative of the relative performance of different algorithms in these terms.

5.5 Multimodal Problems, Selection, and the Need for Diversity

5.5.1 Multimodal Problems

In Sects. 2.3.1 and 3.5 we introduced the concept of multimodal search landscapes and local optima. We discussed how effective search relies on the preservation of sufficient diversity to allow both exploitation of learned information (by investigating regions contained high fitness solutions discovered) and exploration in order to uncover new high-fitness regions.

Multimodality is a typical aspect of the type of problems for which EAs are often employed, either in attempt to locate the global optimum (particularly when a local optimum has the largest basin of attraction), or to identify a number of high–fitness solutions corresponding to various local optima. The latter situation can often arise, for example, when the fitness function used by the EA does not completely specify the underlying problem. An example of this might be in the design of a new widget, where the parameters of the fitness function may change during the design process, as progressively more refined and detailed models are used as decisions such as the choice of materials, etc., are made. In this situation it is valuable to be able to examine a number of possible options, first so as to permit room for human aesthetic judgements, and second because it is probably desirable to use solutions from niches with broader peaks rather than from a sharp peak. This is because the latter may be overfitted (that is, overly specialised) to the current fitness function and may not be as good once the fitness function is refined.

The population-based nature of EAs holds out much promise for identifying multiple optima, however, in practice the finite population size, when coupled with recombination between *any* parents (known as **panmictic** mixing) leads to the phenomenon known as **genetic drift** and eventual convergence around one optimum. The reasons for this can easily be seen: imagine that we have two equally fit niches, and a population of 100 individuals originally equally divided between them. Eventually, because of the random effects in selection, it is likely that we will obtain a parent population consisting of 49 of one sort and 51 of the other. Ignoring the effects of recombination and mutation, in the next generation the probabilities of selecting individuals from the two niches are now 0.49 and 0.51 respectively, i.e., we are increasingly likely to select

individuals from the second niche. This effect increases as the two subpopulations become unbalanced, until eventually we end up with only one niche represented in the population.

5.5.2 Characterising Selection and Population Management Approaches for Preserving Diversity

A number of mechanisms have been proposed to aid the use of EAs on multimodal problems. These can be broadly separated into two camps: *explicit* approaches, in which specific changes are made to operators in order to preserve diversity, and *implicit* approaches, in which a framework is used that permits, *but does not guarantee*, the preservation of diverse solutions. Before describing these it is useful to clarify exactly what we mean by 'diversity' and 'space'. Just as biological evolution takes place on a geographic surface, but can also be considered to occur on an adaptive landscape, so we can define a number of spaces within which the evolutionary algorithms operate:

- **Genotype Space:** We may perceive the set of representable solutions as a genotype space and define some distance metrics. This can be a natural distance metrics in that space (e.g., the Manhattan distance) or based on some fundamental move operator. Typical move operators include a single bit-flip for binary spaces, a single inversion for adjacency-based permutation problems and a single swap for order-based permutations problems.
- **Phenotype Space:** This is the end result: a search space whose structure is based on distance metrics between solutions. The neighbourhood structure in this space may bear little relationship to that in the genotype space according to the complexity of the representation–solution mapping.
- **Algorithmic Space:** This is the equivalent of the geographical space on which life on Earth has evolved. Effectively we are considering that the working memory of the EA, that is, the population of candidate solutions, can be structured in some way. This spatial structure could be either a conceptual division, or real: for example, a population might be split over a number of processors or cores.

Explicit approaches to diversity maintenance based on measures of either genotype or phenotypic space include Fitness Sharing (Sect. 5.5.3), Crowding (Sect. 5.5.4), and Speciation (Sect. 5.5.5), all of which work by affecting the probability distributions used by selection. Implicit approaches to diversity maintenance based on the concept of algorithmic space include Island Model EAs (Sect. 5.5.6) and Cellular EAs (Sect. 5.5.7).

5.5.3 Fitness Sharing

This scheme is based upon the idea that the number of individuals within a given niche is controlled by sharing their fitness immediately prior to selection, in an attempt to allocate individuals to niches *in proportion to the niche*

fitness [193]. In practice the scheme considers each possible pairing of individuals i and j within the population (including i with itself) and calculates a distance $d(i, j)$ between them according to some distance metric (phenotypic is preferred if possible, else genotypic, e.g., Hamming distance for binary representations). The fitness F of each individual i is then adjusted according to the number of individuals falling within some prespecified distance σ_{share} using a power-law distribution:

$$F'(i) = \frac{F(i)}{\sum_j sh(d(i,j))},$$

where the sharing function $sh(d)$ is a function of the distance d, given by

$$sh(d) = \begin{cases} 1 - (d/\sigma_{share})^{\alpha} & \text{if } d \leq \sigma_{share}, \\ 0 & \text{otherwise .} \end{cases}$$

The constant value α determines the shape of the sharing function: for α=1 the function is linear, but for values greater than this the effect of similar individuals in reducing a solution's fitness falls off more rapidly with distance.

The value of the share radius σ_{share} decides both how many niches can be maintained and the granularity with which different niches can be discriminated. Deb [114] gives some suggestions for how this might be set if the number of niches is known *in advance*, but clearly this is not always the case. In [110] he suggests that a default value in the range 5–10 should be used.

We should point out that the use of fitness proportionate selection is implicit within the fitness-sharing method. In this case there exists a stable distribution of solutions amongst the niches when solutions from each peak have the same effective fitness F'. Since the niche fitness $F'_k = F_k/n_k$, in this stable distribution each niche k contains a number of solutions n_k proportional to the niche fitness F_k[1]. This point is illustrated in Fig. 5.4. Studies have indicated that the use of alternative selection methods does not lead to the formation and preservation of stable subpopulations in niches [324].

5.5.4 Crowding

The crowding algorithm was first suggested in De Jong's thesis [102] as a way of preserving diversity by ensuring that new individuals replaced *similar* members of the population. The original scheme worked in a steady-state setting (the number of new individuals generated in each step was 20% of the population size). When a new offspring is inserted into the population, a number [2] of members of the parent population are chosen at random, and then the offspring replaces the most similar of those parents. A number of problems

[1] This assumes for the sake of ease that all solutions within a given niche lie at its optimal point, at zero distance from each other.

[2] called the Crowding Factor (CF) - De Jong used CF=2

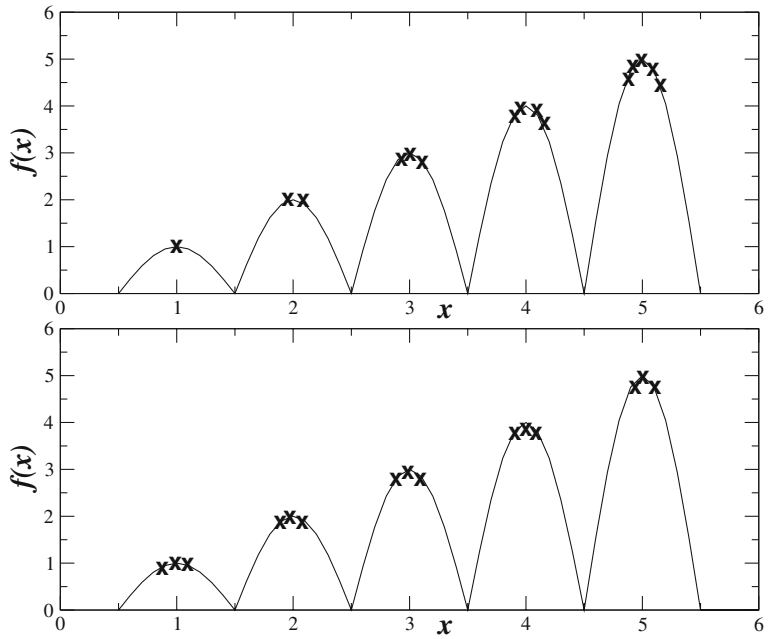

Fig. 5.4. Idealised population distributions under fitness sharing (top) and crowding (bottom). There are five peaks in the landscape with fitnesses (5,4,3,2,1) and the population size is 15. Fitness sharing allocates individuals to peaks in proportion to their fitness, whereas crowding distributes the population evenly amongst the peaks

were found with this approach, and Mahfoud has suggested an improvement called **deterministic crowding** [278]. This algorithm relies on the fact that offspring are likely to be similar to their parents as follows:

1. The parent population is randomly paired.
2. Each pair produces two offspring via recombination.
3. These offspring are mutated and then evaluated.
4. The four pairwise distances between offspring and parents are calculated.
5. Each offspring then competes for survival in a tournament with one parent, so that the intercompetition distances are minimised. In other words, denoting the parents as p, the offspring as o, and using the subscript to indicate tournament pairing, $d(p_1, o_1) + d(p_2, o_2) < d(p_1, o_2) + d(p_2, o_1)$.

The net result of all this is that offspring tend to compete for survival with the most similar parent, so subpopulations are preserved in niches but their size does not depend on fitness; rather it is equally distributed amongst the peaks available. Fig. 5.4 illustrates this point in comparison with the distribution achieved under crowding.

5.5.5 Automatic Speciation Using Mating Restrictions

The automatic speciation approach imposes mating restrictions based on some aspect of the candidate solutions (or their genotypes) defining them as belonging to different species. The population contains multiple species, and during parent selection for recombination individuals will only mate with others from the same (or similar) species. The biological analogy becomes particularly clear when we note that some authors refer to the aspect controlling reproductive opportunities as an individual's 'plumage' [401].

A number of schemes have been proposed to implement speciation, which can be divided into two main approaches. In the first speciation is based on the solution (or its representation), e.g., Deb's phenotype (genotype)-restricted mating [109, 114, 401]. The alternative approach is to add some elements such as tags to the genotype that code for the individual's species, rather than representing part of the solution. See [62, 109, 409] for implementations, noting that many of these ideas were previously suggested by other authors. These are usually randomly initialised and subject to recombination and mutation. Common to both approaches is the idea that once an individual has been selected to be a parent, then the choice of mate involves the use of a pairwise distance metric (in phenotype or genotype space as appropriate), with potential mates being rejected beyond a certain distance.

Note that in the tag scheme, there is initially no guarantee that individuals with similar tags will represent similar solutions, although after a few generations selection will usually take care of this problem. Neither is there any guarantee that different species will contain different solutions, although Spears goes some way towards rectifying this by also using the tags to perform fitness sharing [409], and even without this Deb reported improved performance compared to a standard GA [109]. Similarly, although the phenotype-based speciation scheme does not guarantee diversity maintenance, when used in conjunction with fitness sharing, it was reported to give better results than fitness sharing on its own [114].

5.5.6 Running Multiple Populations in Tandem: Island Model EAs

The idea of evolving multiple populations in tandem is also known as **island model EAs**, **parallel EA**, and, more precisely **coarse-grain parallel EAs**. These schemes attracted great interest in the 1980s when parallel computing became popular [87, 88, 274, 339, 356, 425] and are still applicable on MIMD systems such as computing clusters. Of course, they can equally well be implemented on a single-processor architecture, without the time speed-up.

The essential idea is to run multiple populations in parallel, in some kind of communication structure. The communication structure is usually a ring or a torus, but in principle any form is possible, and sometimes this is determined by the architecture of the parallel system, e.g., a hypercube [425]. After a (usually fixed) number of generations (known as an **epoch**), a number of

individuals are selected from each population to be exchanged with others from neighbouring populations – this can be thought of as **migration**.

In [284] this approach is discussed in the context of Eldredge and Gould's theory of punctuated equilibria [154] and the exploration–exploitation trade-off. They suggest that during the epochs between communication, when each subpopulation is evolving independently of the others, exploitation occurs, so that the subpopulations each explore the search space around the fitter solutions that they contain. When communication takes place, the injection of individuals of potentially high fitness, and with (possibly) radically different genotypes, facilitates exploration, particularly as recombination happens between the two different solutions.

Whilst extremely attractive in theory, it is obvious that there are no guarantees per se that the different subpopulations are actually exploring different regions of the search space. One possibility is clearly to achieve a start at this through a careful initialisation process, but even if this is used, there are a number of parameters that have been shown to affect the ability of this technique to explore different peaks and obtain good results even when only a single solution is desired as the end result.

A number of detailed studies have been made of the effects of different parameters and implementations of this basic scheme (see, e.g., earlier references in this section, and [276] for a more recent treatment), but of course we must bear in mind that the results obtained may be problem dependent, and so we will restrict ourselves to commenting on a few important facets:

- How often to exchange individuals? The essential problem here is that if the communication occurs too frequently, then all subpopulations will converge to the same solution. Equally if it is done too infrequently, and one or more subpopulations has converged quickly in the vicinity of a peak, then significant amounts of computational effort may be wasted. Most authors have used epoch lengths of the range 25–150 generations. An elegant alternative strategy proposed in [284] is to organise communication adaptively, that is to say, to stop the evolution in each subpopulation when no improvement has been observed for, say, 25 generations.
- How many, and which individuals to exchange? Many authors have found that in order to prevent too rapid convergence to the same solution, it is better to exchange a small number of solutions between subpopulations – usually 2–5. Once the amount of communication has been decided, it is necessary to specify *which* individuals are selected from each population to be exchanged. Clearly this can be done either by some fitness-based selection mechanism (e.g., "copy-best" [339], "pick-from-fittest-half" [425]) or at random [87]. It must also be decided whether the individuals being exchanged are effectively moved from one population to another, thus (assuming a symmetrical communication structure) maintaining subpopulation sizes, or whether they are merely copied, in which case each subpopulation must then undergo some kind of survivor selection mechanism.

The choices of how many and which individuals to exchange will evidently affect the tendency of the subpopulations to converge to the same solution. Random, rather than fitness-based, selection strategy is less likely to lead to takeover of one population by a new high-fitness migrant, and exchanging more solutions also leads to faster mixing and possible takeover. However, the extent to which these factors affect the behaviour is clearly tied to the epoch length, since if this is long enough to permit fitness convergence then all of the solutions contained within a given subpopulation are likely to be genotypically very similar, so the selection method used becomes less important.

- How to divide the population into subpopulations? The general rule here appears to be that provided a certain (problem-dependent) minimum subpopulation size is respected, then more subpopulations usually gives better results. This clearly fits in with our understanding, since if each subpopulation is exploring a different peak (the ideal scenario), the more peaks explored, the likely it is that one of them will contain the global optimum.

Finally, it is worth mentioning that it is perfectly possible to use different algorithmic parameters on different islands. Thus in the **injection island models** the subpopulations are arranged hierarchically with each level operating at a different granularity of representation. Equally, parameters such as the choice of recombination or mutation operator and associated parameters, or even subpopulation sizes, might be different between different subpopulations [148, 367].

5.5.7 Spatial Distribution Within One Population: Cellular EAs

In the previous section we described the implementation of a population structure in the form of a number of subpopulations with occasional communication. In this section we describe an alternative model whereby a single population is considered to be split into a larger number of smaller overlapping subpopulations (demes) by being distributed within algorithmic space. We can consider this to be equivalent to the situation whereby biological individuals are separated, only mating and competing for survival with those within a certain distance to them. To take a simple example from the days of less-rapid transport, a person might only have been able to marry and have children with someone from their own or surrounding villages. Thus should a new gene for say, telekinesis, evolve, even if it offers huge evolutionary advantage, at first it will only spread to surrounding villages. In the next generation it might spread to those surrounding them, and so on, only slowly diffusing or percolating throughout the society.

This effect is implemented by considering each member of the population to exist on a different point on a grid, and only permitting recombination and selection with neighbours, hence the common names of parallel EAs [195, 311], **fine-grain parallel EAs** [281], **diffusion model EA** [451], **distributed**

EAs [225] and, more commonly nowadays **cellular EAs** [456, 5]. There have been a great many differing implementations of this form of EA, but we can broadly outline the algorithm as follows:

1. The current population is conceptually distributed on a (usually toroidal) grid, with one individual per node.
2. For each node we have defined a deme (neighbourhood). This is usually the same for all nodes, e.g., for a neighbourhood size of nine on a square lattice, we take the node and all of its immediate neighbours.
3. In each generation we consider each deme in turn and perform the following operations within it:
   - Select two solutions from the nodes in the deme that will act as parents.
   - Generate an offspring via recombination.
   - Mutate, then evaluate the offspring.
   - Select one solution residing on a node in the deme and replace it with the new offspring.

Within this general structure there is scope for considerable differences in implementation. The ASPARAGOS algorithm [195, 311] uses a ladder topology rather than a lattice, and also performs a hill-climbing step after mutation. Several algorithms implemented on massively parallel SIMD or SPMD machines use asynchronous updates in step 3 rather than the sequential mode suggested in the third step above (a good discussion of this issue can be found in [338]). The selection of parents might be fitness-based [95] or random (or one of each [281]), and often one parent is taken to be that residing on the central node of the deme. When fitness-based selection is used it is usually a local implementation of a well-known global scheme such as fitness proportionate or tournament. De Jong and Sarma [106] analysed a number of such schemes and found that local selection techniques generally exhibited less selection pressure than their global versions. While it is common to replace the central node of the deme, again fitness-based or random selection have been used to select the individual to be replaced, or a combination such as "replace current solution if better" [195]. White and Pettey reported results suggesting that the use of fitness in the survivor selection is preferred [451]. A good recent treatment and discussion can be found in the book [5].

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

6

Popular Evolutionary Algorithm Variants

In this chapter we describe the most widely known evolutionary algorithm variants. This overview serves a twofold purpose: On the one hand, it introduces those historical EA variants without which no EC textbook would be complete together with some more recent versions that deserve their own place in the family tableau. On the other hand, it demonstrates the diversity of realisations of the same basic evolutionary algorithm concept.

6.1 Genetic Algorithms

The genetic algorithm (GA) is the most widely known type of evolutionary algorithm. It was initially conceived by Holland as a means of studying adaptive behaviour, as suggested by the title of the book describing his early research: *Adaptation in Natural and Artificial Systems* [220]. However, GAs have largely (if perhaps mistakenly – see [103]) been considered as function optimisation methods. This is perhaps partly due to the title of Goldberg's seminal book: *Genetic Algorithms in Search, Optimization and Machine Learning* [189] and some very high-profile early successes in solving optimisation problems. Together with De Jong's thesis [102] this work helped to define what has come to be considered as the classical genetic algorithm — commonly referred to as the 'canonical' or 'simple GA' (SGA). This has a binary representation, fitness proportionate selection, a low probability of mutation, and an emphasis on genetically inspired recombination as a means of generating new candidate solutions. It is summarised in Table 6.1. Perhaps because it is so widely used for teaching EAs, and is the first EA that many people encounter, it is worth re-iterating that many features that have been developed over the years are missing from the SGA — most obviously that of elitism.

While, the table does not indicate this, GAs traditionally have a fixed workflow: given a population of μ individuals, parent selection fills an intermediary population of μ, allowing duplicates. Then the intermediary population is shuffled to create random pairs and crossover is applied to each consecutive

A.E. Eiben, J.E. Smith, *Introduction to Evolutionary Computing*,
Natural Computing Series, DOI 10.1007/978-3-662-44874-8_6

pair with probability p_c and the children replace the parents immediately. The new intermediary population undergoes mutation individual by individual, where each of the l bits in an individual is modified by mutation with independent probability p_m. The resulting intermediary population forms the next generation replacing the previous one entirely. Note that in this new generation there might be pieces, perhaps complete individuals, from the previous one that survived crossover and mutation without being modified, but the likelihood of this is rather low (depending on the parameters μ, p_c, p_m).

Representation	Bit-strings
Recombination	1-Point crossover
Mutation	Bit flip
Parent selection	Fitness proportional - implemented by Roulette Wheel
Survival selection	Generational

Table 6.1. Sketch of the simple GA

In the early years of the field there was significant attention paid to trying to establish suitable values for GA parameters such as the population size, crossover and mutation probabilities. Recommendations were for mutation rates between $1/l$ and $1/\mu$, crossover probabilities around 0.6-0.8, and population sizes in the fifties or low hundreds, although to some extent these values reflect the computing power available in the 1980s and 1990s.

More recently it has been recognised that there are some flaws in the SGA. Factors such as elitism, and non-generational models were added to offer faster convergence if needed. As discussed in Chap. 5, SUS is preferred to roulette wheel implementation, and most commonly rank-based selection is used, implemented via tournament selection for simplicity and speed. Studying the biases in the interplay between representation and one-point crossover (e.g. [411]) led to the development of alternatives such as uniform crossover, and a stream of work through 'messy-GAs' [191] and 'Linkage Learning' [209, 395, 385, 83] to Estimation of Distribution Algorithms (see Sect. 6.8). Analysis and experience has recognised the need to use non-binary representations where more appropriate (as discussed in Chap. 4). Finally the problem of how to choose a suitable fixed mutation rate has largely been solved by adopting the idea of self-adaptation, where the rates are encoded as extra genes in an individuals representation and allowed to evolve [18, 17, 396, 383, 375].

Nevertheless, despite its simplicity, the SGA is still widely used, not just for teaching purposes, and for benchmarking new algorithms, but also for relatively straightforward problems in which binary representation is suitable. It has also been extensively modelled by theorists (see Chap. 16). Since it has provided so much inspiration and insight into the behaviour of evolutionary processes in combinatorial search spaces, it is fair to consider that if OneMax is the *Drosophilia* of combinatorial problems for researchers, then the SGA is the *Drosophilia* of evolutionary algorithms.

6.2 Evolution Strategies

Evolution strategies (ES) were invented in the early 1960s by Rechenberg and Schwefel, who were working at the Technical University of Berlin on an application concerning shape optimisation (see [54] for a brief history). The earliest ES's were simple two-membered algorithms denoted (1+1) ES's (pronounce: one plus one ES), working in a vector space. An offspring is generated by the addition of a random number independently to each to the elements of the parent vector and accepted if fitter. An alternative scheme, denoted as (1,1) ES (pronounce: one comma one ES) always replaces the parent by the offspring, thus forgetting the previous solutions by definition. The random numbers are drawn from a Gaussian distribution with mean zero and a standard deviation σ, where σ is called the mutation step size. One of the key early breakthroughs of ES research was to propose a simple mechanism for online adjustment of step sizes by the famous **1/5 success rule** of Rechenberg [352] as described in Sect. 8.2.1. In the 1970s the concept of multi-membered evolution strategies was introduced, with the naming convention based on μ individuals in the population and λ offspring generated in one cycle. The resulting $(\mu+\lambda)$ and (μ, λ) ES's gave rise to the possibility of more sophisticated forms of step-size control, and led to the development of a very useful feature in evolutionary computing: **self-adaptation** of strategy parameters, see Sect. 4.4.2. In general, self-adaptivity means that some parameters of the EA are varied during a run in a specific manner: the parameters are included in the chromosomes and coevolve with the solutions. Technically this means that an ES works with extended chromosomes $\langle \overline{x}, \overline{p} \rangle$, where $\overline{x} \in \mathbb{R}^n$ is a vector from the domain of the given objective function to be optimised, while $\overline{p}$ carries the algorithm parameters. Modern evolution strategies always self-adapt the mutation step sizes and sometimes their rotation angles. That is, since the procedure was detailed in 1977 [372] most ESs have been self-adaptive, and other EAs have increasingly adopted self-adaptivity. Recent forms of ES such as the CMA [207] are among the leading algorithms for optimisation of complex real-valued functions. A summary of ES is given in Table 6.2.

Representation	Real-valued vectors
Recombination	Discrete or intermediary
Mutation	Gaussian perturbation
Parent selection	Uniform random
Survivor selection	Deterministic elitist replacement by (μ, λ) or $(\mu + \lambda)$
Speciality	Self-adaptation of mutation step sizes

Table 6.2. Sketch of ES

The basic recombination scheme in evolution strategies involves two parents that create one child. To obtain λ offspring recombination is performed λ

times. There are two recombination variants distinguished by the manner of recombining parent alleles. Using **discrete recombination** one of the parent alleles is randomly chosen with equal chance for either parents. In **intermediate recombination** the values of the parent alleles are averaged. An extension of this scheme allows the use of more than two recombinants, because the two parents are drawn randomly for each position $i \in \{1, \ldots, n\}$ in the offspring anew. These drawings take the whole population of μ individuals into consideration, and the result is a recombination operator with possibly more than two individuals contributing to the offspring. The exact number of parents, however, cannot be defined in advance. This multiparent variant is called **global recombination**. To make terminology unambiguous, the original variant is called **local recombination**. Evolution strategies typically use global recombination. Interestingly, different recombination is used for the object variable part (discrete is recommended) and the strategy parameters part (intermediary is recommended). This scheme preserves diversity within the phenotype (solution) space, allowing the trial of very different combinations of values, whilst the averaging effect of intermediate recombination assures a more cautious adaptation of strategy parameters.

The selection scheme that is generally used in evolution strategies is (μ, λ) selection, which is preferred over $(\mu + \lambda)$ selection for the following reasons:

- The (μ, λ) discards all parents and so can in principle leave (small) local optima, which is advantageous for multimodal problems.
- If the fitness function changes over time, the $(\mu + \lambda)$ selection preserves outdated solutions, so is less able to follow the moving optimum.
- $(\mu + \lambda)$ selection hinders the self-adaptation, because misadapted strategy parameters may survive for a relatively large number of generations. For example, if an individual has relatively good object variables but poor strategy parameters, often all of its children will be bad. Thus they will be removed by an elitist policy, while the misadapted strategy parameters in the parent may survive for longer than desirable.

The selective pressure in evolution strategies is very high because λ is typically much higher than μ (traditionally a 1/7 ratio is recommended, although recently values around 1/4 seem to gain popularity). The **takeover time** τ^* of a given selection mechanism is defined as the number of generations it takes until the application of selection completely fills the population with copies of the best individual, given one copy initially. Goldberg and Deb [190] showed that

$$\tau^* = \frac{\ln \lambda}{\ln(\lambda/\mu)}.$$

For a typical evolution strategy with $\mu = 15$ and $\lambda = 100$, this results in $\tau^* \approx 2$. By way of contrast, for fitness proportional selection in a genetic algorithm with $\mu = \lambda = 100$ it is $\tau^* = \lambda \ln \lambda = 460$. This indicates that an ES is a more aggressive optimizer than a (simple) GA.

6.3 Evolutionary Programming

Evolutionary programming (EP) was originally developed by Fogel et al. in the 1960s to simulate evolution as a learning process with the aim of generating artificial intelligence [166, 174]. Intelligence, in turn, was viewed as the capability of a system to adapt its behaviour in order to meet some specified goals in a range of environments. Adaptive behaviour is the key term in this definition, and the capability to predict the environment was considered to be a prerequisite. The classic EP systems used finite state machines as individuals.

Nowadays EP frequently uses real-valued representations, and so has almost merged with ES. The principal differences lie perhaps in the biological inspiration: in EP each individual is seen as corresponding to a distinct *species*, and so there is no recombination. Furthermore, the selection mechanisms are different. In ES parents are selected stochastically, then the selection of the μ best from the union of $\mu + \lambda$ offspring is deterministic. By contrast, in EP each parent generates exactly one offspring (i.e., $\lambda = \mu$), but these parents and offspring populations are then merged and compete in stochastic round-robin tournaments for survival, as described in Sect. 5.3.2. The field now adopts a very open, pragmatic approach that the choice of representation, and hence mutation, should be driven by the problem; Table 6.3 is therefore a representative rather than a standard algorithm variant.

Representation	Real-valued vectors
Recombination	None
Mutation	Gaussian perturbation
Parent selection	Deterministic (each parent creates one offspring via mutation)
Survivor selection	Probabilistic $(\mu + \mu)$
Speciality	Self-adaptation of mutation step sizes (in meta-EP)

Table 6.3. Sketch of EP

The issue of the advantage of using a mutation-only algorithm versus a recombination and mutation variant has been intensively discussed since the 1990s. Fogel and Atmar [170] compared the results of EP algorithms with and without recombination on a series of linear functions with parameterisable interactions between genes. They concluded that improved performance was obtained from the version without recombination. This led to intensive periods of research in both the EP and the GA communities to try and establish the circumstances under which the availability of a recombination operator yielded improved performance [159, 171, 222, 408]. The current state of thinking has moved on to a stable middle ground. The latest results [232] confirm that the ability of both crossover or Gaussian mutation to produce new offspring of superior fitness to their parents depends greatly on the state of the

search process, with mutation better initially but crossover gaining in ability as evolution progresses. These conclusions agree with theoretical results developed elsewhere and discussed in more depth in Chap. 7. In particular it is stated that: "the traditional practice of setting operator probabilities at constant values, ... is quite limiting and may even prevent the successful discovery of suitable solutions." However, it is perhaps worth noting that even in these studies the authors did not detect a difference between the performance of different crossover operators, which they claim casts significant doubt on the building block hypothesis (Sect. 16.1), so we are not entirely without healthy scientific debate!

Since the 1990s EP variants for optimisation of real-valued parameter vectors have become more frequent and even positioned as 'standard' EP [22, 30]. During the history of EP a number of mutation schemes, such as one in which the step size is inversely related to the fitness of the solutions, have been proposed. Since the proposal of **meta-EP** [165, 166], self-adaptation of step sizes has become the norm, using the scheme in Eq. (4.4). A variety of schemes have been proposed, including mutation variables first then strategy parameters (which violates the rationale explained in Sect. 4.4.2). Tracing the literature on this issue, the paper by Gehlhaar and Fogel [182] seems to be a turning point. Here the authors explicitly compare the 'sigma first' and 'sigma last' strategies and conclude that the first one – the standard ES manner – offers a consistent general advantage over the second one. Notably in a paper and book [81, 168], Fogel uses the lognormal adaptation of n standard deviations σ_i, *followed* by the mutation of the object variables x_i themselves, suggesting that EP is practically merging with ES regarding this aspect. Other ideas from ES have also informed the development of EP algorithms, and a version with self-adaptation of covariance matrices, called **R-meta-EP** is also in use. Worthy of note is Yao's improved fast evolutionary programming algorithm (IFEP) [470], whereby two offspring are created from each parent, one using a Gaussian distribution to generate the random mutations, and the other using the Cauchy distribution. The latter has a fatter tail (i.e., more chance of generating a large mutation), which the authors suggest gives the overall algorithm greater chance of escaping from local minima, whilst the Gaussian distribution (if small step sizes evolve) gives greater ability to fine-tune the current parents.

6.4 Genetic Programming

Genetic programming is a relatively young member of the evolutionary algorithm family. It differs from other EA strands in its application area as well as the particular representation (using trees as chromosomes). While the EAs discussed so far are typically applied to optimisation problems, GP could instead be positioned in machine learning. In terms of the different problem types as discussed in Chapter 2, most other EAs are for finding some input

realising maximum payoff (Fig. 1.1), whereas GP is used to seek models with maximum fit (Fig. 1.2). Clearly, once maximisation is introduced, modelling problems can be seen as special cases of optimisation. This, in fact, is the basis of using evolution for such tasks: models — represented as parse trees — are treated as individuals, and their fitness is the model quality to be maximised. The summary of GP is given in Table 6.4.

Representation	Tree structures
Recombination	Exchange of subtrees
Mutation	Random change in trees
Parent selection	Fitness proportional
Survivor selection	Generational replacement

Table 6.4. Sketch of GP

The parse trees used by GP as chromosomes 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, cf. Sect. 4.6. In particular, they can be envisioned as executable codes, that is, programs. The syntax of functional programming, e.g., the language LISP, very closely matches the so-called Polish notation of expressions. For instance, the formula in Eq. (4.8) can be rewritten in this Polish notation as

$$+(\cdot(2,\pi),-(+(x,3),/(y,+(5,1)))),$$

while the executable LISP code[1] looks like:

$$(+\ (\cdot\ 2\ \pi)\ (-\ (+\ x\ 3)\ (/\ y\ (+\ 5\ 1)))).$$

Based on this perception, GP can be positioned as the "programming of computers by means of natural selection" [252], or the "automatic evolution of computer programs" [37].

There are a few other issues that are specific to tree-based representations, and hence (but not exclusively) to genetic programming.

Initialisation can be carried out in different ways for trees. The most common method used in GP is the so-called **ramped half-and-half** method. In this method a maximum initial depth D_{max} of trees is chosen, and then each member of the initial population is created from the sets of functions F and terminals T using one of the two methods below with equal probability:

- *Full method*: here each branch of the tree has depth D_{max}. The contents of nodes at depth d are chosen from F if $d < D_{max}$ or from T if $d = D_{max}$.

[1] To be precise we should use PLUS, etc., for the operators.

- *Grow method*: here the branches of the tree may have different depths, up to the limit D_{max}. The tree is constructed beginning from the root, with the contents of a node being chosen stochastically from $F \cup T$ if $d < D_{max}$.

Stochastic Choice of a Single Variation Operator. In GP offspring are typically created by either recombination *or* mutation, rather than recombination *followed by* mutation, as is more common in other variants. This difference is illustrated in Fig. 6.1 (inspired by Koza [252]), which compares the loop for filling the next generation in a generational GA with that of GP.

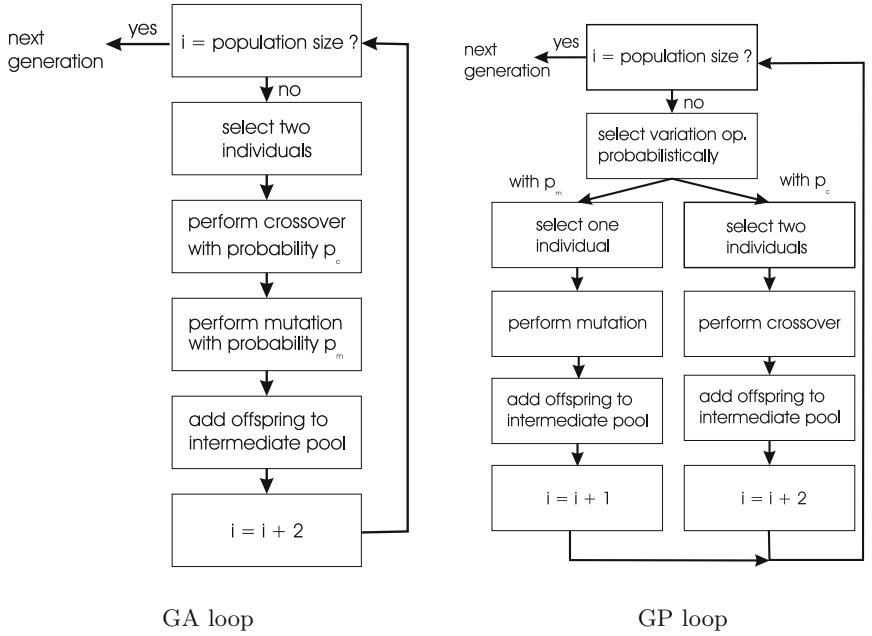

Fig. 6.1. GP flowchart versus GA flowchart. The two diagrams show two options for filling the intermediary population in a generational scheme. In a conventional GA mutation and crossover are used to produce the next offspring (left). Within GP, a new individual is created by either mutation or crossover (right).

Low or Zero Mutation Probabilities. 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 mutation frequencies, even though some studies indicate that an (almost) pure crossover approach might be inferior [275].

Over-selection is often used to deal with the typically large population sizes (population sizes of several thousands are not unusual in GP). The method first ranks the population, then divides it into two groups, one containing the top $x\%$ and the other containing the other $(100-x)\%$. When parents are selected, 80% of the selection operations come from the first group, and the other 20% from the second group. The values of x used are found empirically by rule of thumb and depend on the population size with the aim that the number of individuals from which the majority of parents are chosen stays constant in the low hundreds, i.e., the selection pressure increases dramatically for larger populations.

Bloat (sometimes called the 'survival of the fattest') is a phenomenon observed in GP whereby average tree sizes tend to grow during the course of a run. There are many studies devoted to understanding why bloat occurs and to proposing countermeasures, see for instance [268, 407]. Although the results and discussions are not conclusive, one primary suspect is the sheer fact that we have chromosomes with variable length, meaning that the *possibility* for chromosome sizes to grow along the evolution already implies that they will *actually* do so. Probably the simplest way to prevent bloat is to introduce a maximum tree size and forbid a variation operator if the child(ren) resulting from its application would exceed this maximum size. In this case, this threshold can be seen as an additional parameter of mutation and recombination in GP. Several advanced techniques have also been proposed, but the only one that is widely acknowledged is that of **parsimony pressure**. Such a pressure towards parsimony (i.e., being 'stingy' or ungenerous) is achieved through introducing a penalty term in the fitness formula that reduces the fitness of large chromosomes [228, 406] or using multiobjective techniques [115].

6.5 Learning Classifier Systems

Learning Classifier Systems (LCS) represent an alternative evolutionary approach to model building based on the use of rule sets, rather than parse trees, to represent knowledge [270, 269]. LCS are used primarily in applications where the objective is to evolve a system that will respond to the current state of its environment (i.e., the inputs to the system) by suggesting a response that in some way maximises future reward from the environment.

An LCS is therefore a combination of a classifier system and a learning algorithm. The classifier system component is typically a set of rules, each mapping certain inputs to actions. The whole rule set therefore constitutes a model that covers the space of possible inputs and suggests the most appropriate actions for each. The learning algorithm component of an LCS is implemented by an evolutionary algorithm, whose population members either represent individual rules, or complete rule sets, known respectively as the Michigan and Pittsburgh approaches. The fitness driving the evolutionary process may be driven by many different forms of learning, here we restrict

ourselves to 'supervised' learning, where at each stage the system receives a training signal (reward) from the environment in response to the output it proposes. This helps emphasise the difference between the Michigan and Pittsburgh approaches. In the former, data items are presented to the system one-by-one and individual rules are rewarded according to their predictions. By contrast, in a Pittsburgh approach each individual represents a complete model, so the fitness would normally be calculated by presenting the entire data set and calculating the mean accuracy of the predictions.

The **Michigan-style LCS** was first described by Holland in 1976 as a framework for studying learning in condition/action rule-based systems, using genetic algorithms as the principal method for the discovery of new rules and the reinforcement of successful ones [219]. Typically each member of the population was a single rule representing a partial model – that is to say it might only cover a region of the decision space. Thus it is the entire population that together represents the learned model. Each rule is a tuple {`condition:action:payoff`}. The condition specifies a region of the space of possible inputs in which the rule applies. The condition parts of rules may contain wildcard, or 'don't-care' characters for certain variables, or may describe a set of values that a given variable may take – for example, a range of values for a continuous variable. Rules may be distinguished by the number of wildcards they contain, and one rule is said to be more specific than another if it contains fewer wildcards, or if the ranges for certain variables are smaller — in other words if it covers a smaller region of the input space. Given this flexibility, it is common for the condition parts of rules to overlap, so a given input may match a number of rules. In the terminology of LCS, the subset of rules whose condition matches the current inputs from the environment is known as the **match set**. These rules may prescribe different actions, of which one is chosen. The action specifies either the action to be taken (for example, if controlling robots or on-line trading agents) or the system's prediction (such as a class label or a numerical value). The subset of the match set advocating the chosen action is known as the **action set**. Holland's original framework maintained lists of which rules have been used, and when a reward was received from the environment a portion was passed back to recently used rules to provide information for the selection mechanism. The intended effect is that the strength of a rule predicts the value of the reward that the system will gain for undertaking the action. However the framework proved unwieldy and difficult to make work well in practice.

LCS research was reinvigorated in the mid-1990s by Wilson who removed the concept of memory and stripped out all but the essential components in his minimalist ZCS algorithm [464]. At the same time several authors were noting the conceptual similarity between LCS and reinforcement learning algorithms which attempt to learn, for each input state, an accurate mapping from possible actions to expected rewards. The XCS algorithm [465] firmly established this link by extending rule-tuples to {`condition:action:payoff,accuracy`}, where the accuracy value reflects the system's experience of how well the pre-

dicted payoff matches the reward received. Unlike ZCS, the EA is restricted at each cycle — originally to the match set, latterly to the action set, which increases the pressure to discover generalised conditions for each action. As per ZCS, a credit assignment mechanism is triggered by the receipt of rewards from the environment to update the predicted pay-offs for rules in the previous action set. However, the major difference is that these are not used directly to drive selection in the evolution process. Instead selection operates on the basis of accuracy, so the algorithm can in principle evolve a *complete* mapping from input space to actions.

Table 6.5 gives a simple overview of the major features of Michigan-style classifiers for a problem with a binary input and output space. The list below summarizes the main workflow of the algorithm.

1. A new set of inputs are received from the environment.
2. The rule base is examined to find the match-set of rules.
   - If the match set is empty, a 'cover operator' is invoked to generate one or more new matching rules with a random action.
3. The rules in the match-set are grouped according to their actions.
4. For each of these groups the mean accuracy of the rules is calculated.
5. An action is chosen, and its corresponding group noted as the action set.
   - If the system is an 'exploit' cycle, the action with the highest mean accuracy is chosen.
   - If the system is in an 'explore' cycle, an action is chosen randomly or via fitness-proportionate selection, acting on the mean accuracies.
6. The action is carried out and a reward is received from the environment.
7. The estimated accuracy and predicted payoffs are then updated for the rule in the current and previous action sets, based on the rewards received and the predicted pay-offs, using a Widrow–Hoff style update mechanism.
8. If the system is in an 'explore' cycle, an EA is run within the action-set, creating new rules (with pay-off and accuracies set to the mean of their parents), and deleting others.

Representation	tuple of {`condition:action:payoff,accuracy`} conditions use {0,1,#} alphabet
Recombination	One-point crossover on conditions/actions
Mutation	Binary/ternary resetting as appropriate on action/conditions
Parent selection	Fitness proportional with sharing within environmental niches
Survivor selection	Stochastic, inversely related to number of rules covering same environmental niche
Fitness	Each reward received updates predicted payoff and accuracy of rules in relevant action sets by reinforcement learning.

Table 6.5. Sketch of a Michigan-style LCS for a binary input and action space

The **Pittsburgh-style LCS** predates, but is similar to the better-known GP: each member of the evolutionary algorithm's population represents a complete model of the mapping from input to output spaces. Each gene in an individual typically represents a rule, and again a new input item may match more than one rule, in which case typically the first match is taken. This means that the representation should be viewed as an ordered list, and two individuals which contain the same rules, but in a different order on the genome, are effectively different models. Learning of appropriately complex models is typically facilitated by using a variable-length representation so that new rules can be added at any stage. This approach has several conceptual advantages — in particular, since fitness is awarded to complete rule sets, models can be learned for complex multi-step problems. The downside of this flexibility is that, like GP, Pittsburgh-style LCS suffers from bloat and the search space becomes potentially infinite. Nevertheless, given sufficient computational resources, and effective methods of parsimony to counteract bloat, Pittsburgh-style LCS has demonstrated state-of-the-art performance in several machine learning domains, especially for applications such as bioinformatics and medicine, where human-interpretability of the evolved models is vital and large data-sets are available so that the system can evolve off-line to minimise prediction error. Two recent examples winning Humies Awards for better than human performance are in the realms of prostate cancer detection [272] and protein structure prediction [16].

6.6 Differential Evolution

In this section we describe a young, but powerful member of the evolutionary algorithm family: differential evolution (DE). Its birth can be dated to 1995, when Storn and Price published a technical report describing the main concepts behind a "new heuristic approach for minimizing possibly nonlinear and nondifferentiable continuous space functions" [419]. The distinguishing feature that delivered the name of this approach is a twist to the usual reproduction operators in EC: the so-called **differential mutation**. Given a population of candidate solution vectors in $\mathbb{R}^n$ a new mutant vector $\overline{x}'$ is produced by adding a **perturbation vector** to an existing one,

$$\overline{x}' = \overline{x} + \overline{p},$$

where the perturbation vector $\overline{p}$ is the scaled vector difference of two other, randomly chosen population members

$$\overline{p} = F \cdot (\overline{y} - \overline{z}), \tag{6.1}$$

and the scaling factor $F > 0$ is a real number that controls the rate at which the population evolves. The other reproduction operator is the usual uniform crossover, subject to one parameter, the crossover probability $Cr \in [0, 1]$ that

defines the chance that for any position in the parents currently undergoing crossover, the allele of the first parent will be included in the child. (Remember that in GAs the crossover rate $p_c \in [0, 1]$ is defined for any given pair of individuals and it is the likelihood of actually executing crossover.) DE also has a slight twist to the crossover operator: at one randomly chosen position the child allele is taken from the first parent without making a random decision. This ensures that the child does not duplicate the second parent.

In the main DE workflow populations are lists, rather than (multi)sets, allowing references to the i-th individual by its position $i \in \{1, \ldots, \mu\}$ in this list. The order of individuals in such a population $P = \langle \overline{x}_1, \ldots, \overline{x}_i, \ldots, \overline{x}_\mu \rangle$ is not related to their fitness values. An evolutionary cycle starts with creating a mutant vector population $M = \langle \overline{v}_1, \ldots, \overline{v}_\mu \rangle$. For each new mutant $\overline{v}_i$ three vectors are chosen randomly from P, a base vector to be mutated and two others to define a perturbation vector. After making the mutant vector population, a so-called trial vector population $T = \langle \overline{u}_1, \ldots, \overline{u}_\mu \rangle$ is created, where $\overline{u}_i$ is the result of applying crossover to $\overline{v}_i$ and $\overline{x}_i$. (Note, that it is guaranteed that $\overline{u}_i$ does not duplicate $\overline{x}_i$.) In the last step deterministic selection is applied to each pair $\overline{x}_i$ and $\overline{u}_i$: the i-th individual in the next generation is $\overline{u}_i$ if $f(\overline{u}_i) \leq f(\overline{x}_i)$ and $\overline{x}_i$ otherwise.

Representation	Real-valued vectors
Recombination	Uniform crossover
Mutation	Differential mutation
Parent selection	Uniform random selection of the 3 necessary vectors
Survival selection	Deterministic elitist replacement (parent vs. child)

Table 6.6. Sketch of differential evolution

In general, a DE algorithm has three parameters, the scaling factor F, the population size μ (usually denoted by NP in the DE literature), and the crossover probability Cr. It is worth noting that despite mediating a crossover process, Cr can also be thought of as a mutation rate, i.e., the approximate probability that an allele will be inherited from a mutant [343]. The DE community also emphasises another aspect of uniform crossover: The number of inherited mutant alleles follows a binomial distribution, since allele origins are determined by a finite number of independent trials having two outcomes with constant probabilities.

Over the years, several DE variants have been invented and published. One of the modifications concerns the choice of the base vector when building the mutant population M. It can be randomly chosen for each v_i, as presented here, but it can be fixed, always using the best vector in the population and only varying the perturbation vectors. Another extension is obtained by allowing more than one difference vector to define the perturbation vector in the mutation operator. For example, using two difference vectors, Equation

6.1 becomes

$$\overline{p} = F \cdot (\overline{y} - \overline{z} + \overline{y}' - \overline{z}') \tag{6.2}$$

where $\overline{y}, \overline{z}, \overline{y}', \overline{z}'$ are four randomly chosen population members.

In order to classify the different variants, the notation DE/a/b/c has been introduced in the literature, where a specifies the base vector, e.g., "rand" or "best", b is the number of difference vectors used to define the perturbation vector, and c denotes the crossover scheme, e.g., "bin" stands for using uniform crossover (because of the binomial distribution of donor alleles it generates). Using this notation, the basic version described above is DE/rand/1/bin.

6.7 Particle Swarm Optimisation

The algorithm we describe here deviates somewhat from other evolutionary algorithms in that it is inspired by social behavior of bird flocking or fish schooling, while the name and the technical terminology are grounded in physical particles [340, 248]. Seemingly there is no evolution in a particle swarm optimizer, but algorithmically it does fit in the general EA framework. Particle swarm optimisation (PSO) was launched in 1995, when Kennedy and Eberhart published their seminal paper about a "concept for the optimization of nonlinear functions using particle swarm methodology" [247]. Similarly to DE, the distinguishing feature of PSO is a twist to the usual reproduction operators in EC: PSO does not use crossover and its mutation is defined through a vector addition. However, PSO differs from DE and most other EC dialects in that every candidate solution $\overline{x} \in \mathbb{R}^n$ carries its own perturbation vector $\overline{p} \in \mathbb{R}^n$. Technically, this makes them quite similar to evolution strategies that use the mutation step sizes in the perturbation vector parts, cf. Sect. 4.4.2 and Sect. 6.2. However, the PSO mindset and terminology is based on a spatial metaphor of particles with a location and velocity, rather than a biological one of individuals with a genotype and mutation.

To simplify the explanation and to emphasise the similarities to other evolutionary algorithms, we present PSOs in two steps. First we give a description that captures the essence of the system using the notion of perturbation vectors $\overline{p}$. Second, we provide the technical details in terms of vectors for velocity $\overline{v}$ and a personal best $\overline{b}$ in line with the PSO literature.

On a conceptual level every population member in a PSO can be considered as a pair $\langle \overline{x}, \overline{p} \rangle$, where $\overline{x} \in \mathbb{R}^n$ is a candidate solution vector and $\overline{p} \in \mathbb{R}^n$ is a perturbation vector that determines how the solution vector is changed to produce a new one. The main idea is that a new pair $\langle \overline{x}', \overline{p}' \rangle$ is produced from $\langle \overline{x}, \overline{p} \rangle$ by first calculating a new perturbation vector $\overline{p}'$ (using $\overline{p}$ and some additional information) and adding this to $\overline{x}$. That is,

$$\overline{x}' = \overline{x} + \overline{p}'$$

The core of the PSO perspective is to consider a population member as a point in space with a position and a velocity and use the latter to determine a new position (and a new velocity). Thus, looking under the hood of a PSO we find that a perturbation vector is a velocity vector $\overline{v}$ and a new velocity vector $\overline{v'}$ is defined as the weighted sum of three components: $\overline{v}$ and two vector differences. The first points from the current position $\overline{x}$ to the best position $\overline{y}$ the given population member ever had in the past, and the second points from $\overline{x}$ to the best position $\overline{z}$ the whole population ever had. Formally, we have

$$\overline{v}' = w \cdot \overline{v} + \phi_1 U_1 \cdot (\overline{y} - \overline{x}) + \phi_2 U_2 \cdot (\overline{z} - \overline{x})$$

where w and ϕ_i are the weights (w is called the inertia, ϕ_1 is the learning rate for the personal influence and ϕ_2 is the learning rate for the social influence), while U_1 and U_2 are randomizer matrices that multiply every coordinate of $\overline{y} - \overline{x}$ and $\overline{z} - \overline{x}$ by a number drawn from the uniform distribution.

It is worth noting that this mechanism requires some additional book keeping. In particular, the personal best $\overline{y}$ and the global best $\overline{z}$ must be kept in memory. This requires a unique identifier for population members that keeps the 'identity' of the given individuals and allows it to maintain the personal memory. To this end, PSO populations are lists, rather than (multi)sets, allowing references to the i-th individual. Similarly to DE, the order of individuals in such a population is not related to their fitness values. Furthermore, the perturbation vector $\overline{p}_i \in \mathbb{R}^n$ for any given $\overline{x}_i \in \mathbb{R}^n$ is not stored directly, as the notation $\langle \overline{x}, \overline{p} \rangle$ in the previous paragraph would indicate, but indirectly by the velocity vector $\overline{v}_i$ and the personal best $\overline{b}_i$ of the i-th population member. Thus, technically, the i-th individual is a triple $\langle \overline{x}_i, \overline{v}_i, \overline{b}_i \rangle$, where $\overline{x}_i$ is the solution vector (perceived as a position), $\overline{v}_i$ is its velocity vector, and $\overline{b}_i$ is its personal best. During an evolutionary cycle each triple $\langle \overline{x}_i, \overline{v}_i, \overline{b}_i \rangle$ is replaced by the mutant triple $\langle \overline{x}'_i, \overline{v}'_i, \overline{b}'_i \rangle$ using the following formulas

$$\overline{x}'_i = \overline{x} + \overline{v}'_i$$
$$\overline{v}'_i = w \cdot \overline{v}_i + \phi_1 U_1 \cdot (\overline{b}_i - \overline{x}_i) + \phi_2 U_2 \cdot (\overline{c} - \overline{x}_i)$$

where $\overline{c}$ denotes the population's global best (champion) and

$$\overline{b}'_i = \begin{cases} \overline{x}'_i & \text{if } f(\overline{x}'_i) < f(\overline{b}_i) \\ \overline{b}_i & \text{otherwise} \end{cases}$$

The rest of the basic PSO algorithm is actually quite simple, since parent selection and survivor selection are trivial. An overview is given in Table 6.7.

6.8 Estimation of Distribution Algorithms

Estimation of distribution algorithms (EDA) are based on the idea of replacing the creation of offspring by 'standard' variation operators (recombination

Representation	Real-valued vectors
Recombination	None
Mutation	Adding velocity vector
Parent selection	Deterministic (each parent creates one offspring via mutation)
Survival selection	Generational (offspring replace parents)

Table 6.7. Sketch of particle swarm optimisation

and mutation) by a three-step process. First a 'graphical model' is chosen to represent the current state of the search in terms of the dependencies between variables (genes) describing a candidate solution. Next the parameters of this model are estimated from the current population to create a conditional probability distribution over the variables. Finally, offspring are created by sampling this distribution.

Probabilistic Graphical Models (PGMs) have been used as models of uncertainty in artificial intelligence systems since the early 1980s. In this approach models are considered to be graphs $G = (V, E)$, where each vertex $v \in V$ represents a single variable, and each directed edge $e \in E$ represents a dependency between two variables. Thus, for example, the presence of an edge $e = \{i, j\}$ denotes that the probability of obtaining a particular value for variable j depends on the value of variable i. Usually graphs are restricted to be acyclic to avoid difficulties with infinite loops.

The pseudocode in Figure 6.2 illustrates the way that offspring are created via the processes of **model selection**, **model fitting** and **model sampling**.

```
BEGIN
  INITIALISE population P^0 with μ random candidate solutions;
  set t = 0;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    EVALUATE each candidate in P^t;
    SELECT subpopulation P_s^t to be used in the modeling steps;
    MODEL SELECTION creates graph G by dependencies in P_s^t;
    MODEL FITTING creates Bayesian Network BN with G and P_s^t;
    MODEL SAMPLING produces a set Sample(BN) of μ candidates;
    set t = t + 1;
    P^t = Sample(BN);
  OD
END
```

Fig. 6.2. Pseudocode for generic estimation of distribution algorithm

In principle, any standard selection method may be used to select P_s^t, but it is normal practice to use truncation selection and to take the fittest subset of the current population as the basis for the subsequent modelling process. Survivor selection in EDAs typically uses a generational model: newly generated offspring replaces the old population.

Model selection is the critical element in using a PGM approach to data modelling (in our case modelling a population in an EDA). In essence, it amounts to finding the appropriate structure to capture the conditional (in)dependencies between variables. In the context of genetics, and also of evolutionary computation, this process is also known as the "Linkage Learning" problem. Historically the pattern of research in this area has progressed via permitting increasing complexity for the types of structural dependencies examined. The earliest univariate algorithms such as PBIL [35] assumed variables behaved independently. The second generation of EDAs such as MIMIC [59] and BMDA [337] selected structures from the possible pairwise interactions, and current methods such as BOA [336] select structures from the set of all trees of some prespecified maximum size. Of course the number of possible combinations of variables expands extremely rapidly, meaning that some form of search is needed to identify good structural models. Broadly speaking there are two approaches to this. The first is direct estimation that has the reputation of being complex and in most cases impractical. More widely used is the "score + search" approach, which is effectively heuristic search in the space of possible graphical models. This relies heavily on the use of metrics which reflect how well a model, and hence the induced Bayesian Network, captures the underlying structure of the data. A variety of metrics have been proposed in the literature, mostly based on the Kullback–Liebler Divergence metric which is closely related to entropy. A full description of these different quality metrics is beyond the scope of this book.

The process of model-fitting may be characterised as per the pseudocode in Figure 6.3, where we use the notation $P(x, i, c)$ to denote the probability of obtaining allele value i for variable x given the set of conditions c. For the unconditional case we will use $P(x, i, -)$. It should be noted that this code is intended to illustrate the general concept, and that much more efficient implementations exist. For discrete data these parameters form a Bayesian Network, and for continuous data it is common to use a mixture of Gaussian models. Both of these processes are relatively straightforward.

The process of model sampling follows a similar pattern, but in this case within each subgraph we draw random variables to select the parent–node alleles, and then to select allele values for the other nodes using the probabilities from the appropriate partition.

Most of the discussion above has tacitly assumed discrete combinatorial problems and the models fitted have been Bayesian networks. For continuous variable problems a slightly different approach is needed to measure and describe probabilities, which is typically based on normal (Gaussian) distributions. There have been a number of developments, such as the **Iterated Den-**

```
BEGIN
  /*  Let P(x,i,c) denote the probability of generating */
  /*  allele value i for variable x given conditions c */
  /*  Let D denote the set of selected parents */
  /*  Let G denote the model selected */

  FOR EACH unconnected subgraph g ⊂ G DO
    FOR EACH node x ∈ g with no parents DO
      FOR EACH possible allele value i for variable x DO
        set P(x,i,−) = Frequency_In_Subpop(x,i,D);
      OD
    OD
    FOR EACH child node x ∈ g DO
      Partition D according to allele values in x's parents;
      FOR EACH partition c DO
        set P(c) = Sizeof(c)/Sizeof(D);
        FOR EACH possible allele value i for variable x DO
          set P(x,i,c) = Frequency_In_Subpop(x,i,c)/P(c);
        OD
      OD
    OD
  OD
END
```

Fig. 6.3. Pseudocode for generic model fitting

sity Estimation Algorithm [64], and also continuous versions for EDAs. Depending on the complexity of the models permitted, the result is either a univariate or multivariate normal distribution. These are very similar to the correlation matrix in evolution strategies, and a comparison of these approaches may be found in [206].

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