Chapter 2
Evolutionary Algorithms and Other Randomized Search Heuristics

In our description of evolutionary algorithms we make use of terms that stem from biology, hinting at the roots of evolutionary algorithms. We adhere to these standard notions as long as they do not collide with standard notions in computer science. Evolutionary algorithms are structurally very simple. They work in rounds that are called *generations*. Evolutionary algorithms operate on some *search space* S, where S is a set. Points are assigned some quality via a function f. In the context of optimization, f is called an objective function. In the context of evolutionary algorithms, it is usually called *fitness function*. Sometimes one distinguishes between a fitness function f and an objective function f', where the fitness f is utilized by the evolutionary algorithm directly whereas the objective function f' is in some sense closer to the problem at hand. We will introduce this distinction later when we discuss how we can adapt evolutionary algorithms to a given problem (see Sect. 2.5). For now we are only dealing with the fitness function $f: S \to R$ and do not care if there is an objective function f' 'wrapped around.' The set R is the set of possible fitness values. It may be an arbitrary set; most often it is (some subset of) $\mathbb{R}$.

An evolutionary algorithm operates on a collection of points from the search space, called a *population* P. The members of the population, i.e., some points in the search space, are called *individuals*. We use $\mu \in \mathbb{N}$ to denote the size of the population, i.e., $\mu = |P|$. A population is a multiset over S, i.e., it may contain multiple copies of individuals. Since the population changes from generation to generation, we denote the tth population by P_t. Choosing the first population P_0 in the beginning is called *initialization*. Usually for each member x of the population its fitness $f(x)$ is computed and stored. If fitness values are stored this starts with the initialization, where it is done for the complete population. The first step in each generation is to select some individuals from the population that will be used to create new points in the search space. These individuals are referred to as *parents*. Their selection is called *selection for reproduction*. Often this selection is done fitness-based: the chances of individuals to become parents increase with their fitness. Then some random *variation* is applied to the parents where small changes are more likely than large changes. We distinguish two kinds of variation operators.

T. Jansen, *Analyzing Evolutionary Algorithms*, Natural Computing Series,
DOI 10.1007/978-3-642-17339-4_2, © Springer-Verlag Berlin Heidelberg 2013

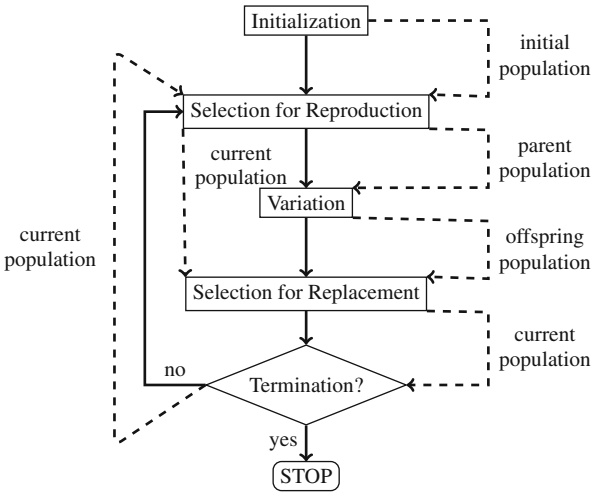

Fig. 2.1 Outline of a generic evolutionary algorithm. *Arrows* show control flow, *dashed arrows* show data flow

Variation operators that take one parent as input and randomly create one individual are called *mutation*. Variation operators that take at least two parents as input and randomly create (at least) one individual are called *crossover*. In any case, the newly created individuals are called *offspring*. It is not unusual to combine different variation operators, e.g., create an offspring via crossover and use this offspring as input for a mutation operator. Usually, the intermediate step (the offspring before mutation) is discarded; but this may be different in some evolutionary algorithms. There is hardly any limit to creativity when designing evolutionary algorithms and thus there are no really strict rules. After creating the offspring population most often there is some kind of *selection for replacement*. The reason is that the size of the population is typically not changed during the whole run. Thus, the new population P_{t+1} is selected from the old population P_t and the newly generated offspring. When describing this selection as selection for replacement, i.e., selecting which individuals are not going to be present in the next population P_{t+1}, it differs from selection for reproduction in preferring individuals with smaller fitness. One may as well describe this selection step as selection for survival where those individuals are selected that will be member of the next generation. In this description there is no conceptual difference to selection for reproduction. After the new generation P_{t+1} is produced (and the old population P_t is discarded), it is checked whether some *termination criterion* is met. If so, some output is produced and the algorithm terminates. Otherwise the next generation starts with the selection for reproduction. This *evolutionary cycle* is summarized in Fig. 2.1. Note that we do not explicitly include evaluating the fitness of individuals in the evolutionary cycle. We silently assume that whenever the fitness of an individual is needed (usually in selection) the value is either available or will be computed 'on the fly.'

Now we describe concrete realizations for each of the six modules: initialization, selection for reproduction, mutation, crossover, selection for replacement, and

termination criterion. Since variation operators depend on the structure of the individuals, we have to discuss different search spaces first.

In principle, the search space may be any set S. Often it is structured as Cartesian product of some other sets, i.e., $S = S_1 \times S_2 \times \cdots \times S_n$. The two most important cases for both theory and practical applications are $S = \{0,1\}^n$ and $S = \mathbb{R}^n$. In applications, combinations of some discrete or even Boolean search space and some real search space are also common. Another important search space is the permutation set, i.e., $S = S_n = \{\pi \mid \pi \text{ is permutation on } \{1, 2, \ldots, n\}\}$. The standard search spaces $\{0,1\}^n$, $\mathbb{R}^n$, and S_n all have in common that points in these search spaces have natural descriptions (and therefore natural implementations) of constant length: bit strings of length n, vectors of n real numbers, and n natural numbers, respectively. This is quite different from search spaces where natural descriptions have varying length. One example of such search spaces is the set of all trees that correspond to arithmetic terms. Most of the time, we will not deal with such search spaces where individuals are more flexible and have no fixed length. They are most often used in evolutionary algorithms that are counted to the subfield of genetic programming.

2.1 Modules of Evolutionary Algorithms

We describe the different modules that can be combined and 'plugged into' the algorithmic framework depicted in Fig. 2.1 to obtain a complete evolutionary algorithm. Where necessary, we give different descriptions for the three standard search spaces $\{0,1\}^n$, $\mathbb{R}^n$, and S_n. Our descriptions are formal and specific. This does not exclude the existence of quite different variants in actual evolutionary algorithms. As already pointed out in the introduction, there is hardly a limit to the creativity of designers of evolutionary algorithms. The descriptions given here will, however, be used throughout this text.

2.1.1 Initialization

In most cases, initialization is done 'purely at random.' For the search spaces $\{0,1\}^n$ and S_n this means uniformly at random. In $\mathbb{R}^n$, initialization is most often done uniformly in some restricted part of the search space.

In applications, it may make more sense to start with 'promising' individuals that were obtained by means of some heuristic, possibly in some previous run of the evolutionary algorithm. It has to be noted, though, that many evolutionary algorithms suffer if the members of a population are too similar to each other.

When dealing with problems with restrictions, it often makes sense to initialize the population with individuals that respect all restrictions. There are variation operators with the property that by using such feasible solutions as input only feasible points in the search space are produced as offspring. This may improve the

performance of an evolutionary algorithm significantly. It is most useful in problems where feasible solutions are difficult to find.

For tests and studies, it may make sense to start in some carefully chosen region of the search space or with individuals having some property of interest. This way one may observe and analyze how the evolutionary algorithm under consideration behaves under circumstances that may be unlikely to evolve in a normal run using plain random initialization.

2.1.2 Selection

Selection appears twice in the evolutionary cycle (Fig. 2.1): as selection for reproduction and as selection for replacement. Since selection for replacement can as well be described as selection for survival, then coinciding with selection for reproduction, we describe all selection mechanisms in a way appropriate for this kind of selection. If selection for replacement is desired, analogous remarks apply but 'the sign of fitness' changes: whereas larger fitness values are to be preferred in selection for reproduction, we prefer smaller fitness values in selection for replacement (since we are maximizing fitness).

Often, selection is based on the fitness of the individuals alone. Some variants do additionally take other properties of the individuals or even the population into account. All variants have in common that they do not select in favor of lower fitness values. While one may in principle devise such selection variants, we discard those variants as unreasonable—essentially being designed not in accordance with the spirit of evolutionary computation.

There are two quite different ways how selection may be performed. One way is to select single individuals and repeat this as many times as selected individuals are needed. This applies to almost all selection methods presented here. Such selection methods can be described by the underlying probability distribution. For each individual one provides the probability to be selected. Given these probabilities, selection is usually performed independently with replacement. The other way is selecting all individuals that are needed in one batch. This implies that no single individual can be selected more often than once. This corresponds to performing selection without replacement.

Uniform selection Select an individual uniformly at random. This is the weakest form of selection that we still consider to be reasonable.

Fitness-proportional selection This selection method assumes that all fitness values are positive. An individual s in the current population P is selected with probability $f(s)/\sum_{x \in P} f(x)$.

The most obvious disadvantage of fitness-proportional selection is the direct and strong dependence on the fitness values. This implies that changing the fitness function f to $f + c$ for some constant c (from an optimization point of view not really a change) changes the selection probabilities observably.

If differences between fitness values in the population are very large, fitness-proportional selection behaves almost deterministically. This may be something that is not wanted. If, on the other hand, differences between fitness values are very small, fitness-proportional selection behaves similar to uniform selection. This is typical for situations later in the run of an evolutionary algorithm: as all fitness values become larger, the relative differences become smaller and fitness-proportional selection becomes increasingly similar to uniform selection.

Variants of fitness-proportional selection Since fitness-proportional selection has such obvious drawbacks, there are a number of variants of fitness-proportional selection that aim at adjusting the selection mechanism and avoiding its difficulties without disregarding the main idea.

Scaled fitness-proportional selection Use fitness-proportional selection but replace the fitness function f by a scaled and translated version $m \cdot f + b$, where m and b are parameters. Sometimes these parameters are chosen adaptively depending on the current population. Clearly, this does not really solve the problem. With respect to $m \cdot f + b$, all our critical remarks about fitness-proportional selection still apply. Moreover, it may be difficult to find appropriate values for m and b.

Boltzmann selection Use fitness-proportional selection but replace the fitness function f by $e^{f/T}$ where T is a parameter (called temperature) that allows one to vary the influence of the actual fitness values. Typically, T varies with time, it is rather large in the beginning and gradually lowered.

Boltzmann selection comes with the immediate advantage that additive changes of the fitness function, i.e., going from f to $f + c$, have no influence on the probabilities to be selected. We observe that

$$\frac{e^{(f(s)+c)/T}}{\sum_{x \in P} e^{(f(x)+c)/T}} = \frac{e^{f(s)/T} \cdot e^{c/T}}{\sum_{x \in P} \left(e^{f(x)/T} \cdot e^{c/T}\right)} = \frac{e^{f(s)/T}}{\sum_{x \in P} e^{f(x)/T}}$$

holds for each constant c.

Rank selection Use fitness-proportional selection but replace the fitness value of individual $s \in P$ by its rank, i.e., its position in the list of all individuals of the current population sorted descending with respect to fitness, ties being broken uniformly at random.

It is not difficult to see that each direct dependence on the concrete fitness values is removed here. In particular, this selection method does not change its characteristics during a run as fitness-proportional selection does.

Tournament selection This selection method comes with a parameter $k \in \mathbb{N} \setminus \{1\}$, called tournament size. The selection works by first selecting k individuals uniformly at random. Then the one with maximal fitness is selected. Ties are broken uniformly at random.

The main advantage of tournament selection is that it works in a very local fashion. There is no need to compute the actual fitness values for all members

of the population. Only those individuals selected for the tournament need to be evaluated. Another advantage is that the parameter k allows for very simple tuning of the selection strength. Clearly, the larger k is, the more competitive tournament selection becomes. All advantages with respect to independence of the concrete fitness values that we listed for rank selection apply here, too. Finally, tournament selection guarantees that the $k - 1$ worst members of the population are never selected.

Truncation selection This is the only selection method that we discuss where all individuals are selected in one batch. The selection is simple: the individuals are selected descending with respect to fitness, and ties are broken uniformly at random. If k individuals are to be selected, the individuals with ranks in $\{1, 2, \ldots, k\}$ are selected.

We introduce two special variants of truncation selection that are both used for selection for replacement only. If all the individuals to be selected are from the offspring population (ignoring the parent population), we call the selection comma-selection. If we select λ offspring from a population of size μ, we write this as (μ, λ). If the individuals to be selected are from both the parent population and the offspring population, we call the selection plus-selection and write $(\mu+\lambda)$.

Selection mechanisms taking the actual individuals into consideration are typically sensitive to the similarity of the individuals. If this is done the selection mechanism tends to aim at increasing diversity in the population by avoiding the selection of individuals that are very similar. We will discuss some concrete examples later when we perform concrete analyses. Another type of selection mechanism takes the 'age' of individuals into account. Inspired from nature, something like a lifespan for individuals may be introduced. The most basic form of this mechanism uses age to break ties: if a parent individual and an offspring individual have the same fitness value, typically the offspring individual is favored.

2.1.3 Mutation

Mutation operators depend on the structure of the individuals and thus on the search space. All mutation operators have in common that they tend to create rather small changes. We describe mutation operators for the three standard search spaces $\{0, 1\}^n$, $\mathbb{R}^n$, and S_n.

Mutation operators for $\{0, 1\}^n$ The parent is $x \in \{0, 1\}^n$ with length n. The ith bit in x is denoted by $x[i]$. We want to favor small changes to x and measure the size of changes by using Hamming distance as our metric. The Hamming distance $\mathrm{H}(x, y)$ of two individuals $x, y \in \{0, 1\}^n$ is simply the number of positions where x and y differ, i.e., $\mathrm{H}(x, y) = \sum_{i=1}^{n} (x[i] + y[i] - 2x[i]y[i])$.

Standard bit mutation The offspring y is created as a copy of x, where for each bit the value of the bit is inverted (a bit-flip occurs) independently with probability p_m. The parameter p_m is called the mutation probability. The expected number of positions where the parent x and its offspring y differ equals $p_m \cdot n$. Since we want to favor small changes, we need $p_m \in (0, 1/2]$. Note that $p_m = 1/2$ implies that y is drawn from $\{0,1\}^n$ independently of x uniformly at random. The most common mutation probability is $p_m = 1/n$, flipping just one bit on average.

***b*-bit mutations** The offspring y is created as a copy of x where the value of exactly b bits is inverted. The positions of these b bits are chosen uniformly at random; $b \in \{1, 2, \ldots, n\}$ is a parameter. Typically, b is quite small: $b = 1$ is a common choice. In comparison with standard bit mutations with mutation probability $p_m = 1/n$, 1-bit mutations have less variance and facilitate analysis. But it has to be noted that the differences in performance induced by changing from 1-bit mutations to standard bit mutations with $p_m = 1/n$ can be enormous.

Mutation operators for $\mathbb{R}^n$ Most often, the offspring $y \in \mathbb{R}^n$ is created from its parent $x \in \mathbb{R}^n$ by adding some vector $m \in \mathbb{R}^n$. In principle, m may be chosen in arbitrary ways. In simple evolutionary algorithms, each component of m is chosen in the same way, i.e., independently and identically distributed. Thus, we now describe the way of choosing one component $m' \in \mathbb{R}$. Since we want to favor small changes, we typically choose m' as a random variable with mean value 0. We distinguish bounded from unbounded mutations. In bounded mutations, m' is chosen from some interval $[a, b]$, often uniformly from $[-a, a]$ for some fixed $a \in \mathbb{R}^+$. More commonly used are unbounded mutation operators, where m' is not bounded. For these unbounded mutations, however, the probability typically decreases strictly with the absolute value. The most common choice makes use of a normal distribution (Gaussian mutations), where we have $e^{-r^2/(2\sigma^2)}/\sqrt{2\pi\sigma^2}$ as probability density function with parameter σ. We know that the mean is 0 and the standard deviation is σ. In some sense, the choice of σ determines the size of the mutation. Typically, σ is not fixed but varies during a run. The idea is to have σ large when far away from optimal points and small when close by. Often, one chooses $\sigma = 1$ fixed and uses an additional parameter $s \in \mathbb{R}^+$ to scale the step size, using $s \cdot m'$ instead of m'.

Mutation operators for S_n For permutations a number of quite different mutation operators have been devised. Which ones make sense clearly depends on the fitness function and, more generally, the problem at hand. Some mutation operators are designed for specific problems. Here, we concentrate on only a few general mutation operators that can be used in different contexts.

Exchange Choose $(i, j) \in \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$ with $i \neq j$ uniformly at random. Generate the offspring y from its parent x by copying x and exchanging i and j.

Jump Choose $(i, j) \in \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$ with $i \neq j$ uniformly at random. Generate the offspring y from its parent x by copying x, moving i to position j, and shifting the other elements accordingly.

Combination of exchange and jump Both mutation operators for permutations, exchange and jump, have in common that they act quite locally. They are not capable of generating arbitrary permutations in a single mutation. But it is often desirable to have this ability since it makes sure that an algorithm cannot become trapped in a local optimum. Thus, we let ourselves be inspired by standard bit mutations with mutation probability $p_m = 1/n$. For these mutations, the number of mutated bits is approximately Poisson distributed with parameter 1. Here, we choose $k \in \mathbb{N}$ according to a Poisson distribution with parameter 1, i.e., $\text{Prob}(k = r) = 1/(e \cdot r!)$. Then we perform $k + 1$ local operations, choosing exchange or jump each time independently with equal probability.

2.1.4 Crossover

Crossover operators cannot be designed independently of the search space. In this way they are similar to mutation operators. The difference from mutation is that more than one parent is used. Most crossover operators make use of two parents, which is clearly close to the natural paradigm. But there are also crossover operators in use that operate on many more parents.

The idea of crossover is to generate an offspring that is in some way similar to its parents. We will define all crossover operators in a way that they produce exactly one offspring.

Crossover operators for $\{0, 1\}^n$ Let $x_1, x_2 \in \{0, 1\}^n$ be the two parents of length n. For crossover operators for $\{0, 1\}^n$ that make use of two parents, it is not unusual to produce two offspring. Such crossover operators produce one offspring by assembling pieces of the two parents. The second offspring is created by assembling exactly the unused pieces of the two parents. This way, for each position the numbers of 0-bits and 1-bits in the two offspring equal these numbers in their parents. Here we stick to describing the construction of one offspring only.

***k*-point crossover** Select k different positions from $\{1, 2, \ldots, n - 1\}$ uniformly at random. Let these positions be $p_1 < p_2 < \cdots < p_k$. Then the offspring y is defined by copying the first p_1 bits from x_1, the second $p_2 - p_1$ from x_2, the next $p_3 - p_2$ bits from x_1, and so on, alternating between x_1 and x_2. This method can be visualized as having the two parent bit strings cut into pieces after each p_ith position. Then the offspring is the concatenation of pieces taken alternately from the two parents. An example for $n = 9$, $k = 3$, and $p_1 = 2$, $p_2 = 5$, $p_3 = 6$ can be seen here.

$$\begin{array}{rccccc|c|ccc}
x_1 = x_1[1] & x_1[2] & x_1[3] & x_1[4] & x_1[5] & x_1[6] & x_1[7] & x_1[8] & x_1[9] \\
x_2 = x_2[1] & x_2[2] & x_2[3] & x_2[4] & x_2[5] & x_2[6] & x_2[7] & x_2[8] & x_2[9] \\
\hline
y = x_1[1] & x_1[2] & x_2[3] & x_2[4] & x_2[5] & x_1[6] & x_2[7] & x_2[8] & x_2[9]
\end{array}$$

We observe that the offspring is equal to both parents at all positions where the parents agree. Usually, only very small numbers of crossover points are used. The most common forms of k-point crossover are 2-point crossover and even 1-point crossover.

Uniform crossover The offspring is created from its parents by copying their bits in the following way. For each bit the value is copied from one of the parents and the decision among the parents is made independently for each bit and with equal probability. As is the case for k-point crossover, the offspring is equal to both parents at all positions where the parents agree. From the set of all individuals with this property, each is generated with equal probability by uniform crossover. This way the number of possible offspring is usually much larger for uniform crossover than for k-point crossover.

Gene pool crossover This crossover operates on an arbitrary number of parents. It is not unusual to use the complete population as parents. If we use $m \in \mathbb{N}$ parents $x_1, x_2, \ldots, x_m$, the offspring $y = y[1]y[2]\cdots y[n]$ is defined by setting $y[i] = 1$ with probability $\left(\sum_{j=1}^{m} x_j[i]\right)/m$, and $y[i] = 0$ otherwise.

Crossover for $\mathbb{R}^n$ For $\mathbb{R}^n$ it is possible to have k-point crossover and uniform crossover very similar to the corresponding operators for $\{0,1\}^n$. Instead of copying bits, real numbers are copied from the parents. In addition, there is another type of crossover that makes use of the natural interpretation of individuals from $\mathbb{R}^n$ as vectors.

Arithmetic crossover This crossover operates on an arbitrary number of parents like gene pool crossover. Again, it is not unusual to use the complete population as parents. If we use $m \in \mathbb{N}$ parents $x_1, x_2, \ldots, x_m$, the offspring $y \in \mathbb{R}^n$ is created as weighted sum $y = \sum_{i=1}^{n} \alpha_i \cdot x_i$, where the parameters α_i sum up to 1, i.e., $\sum_{i=1}^{n} \alpha_i = 1$. Typically, one sets $\alpha_i = 1/m$ for all $i \in \{1, 2, \ldots, n\}$. This is called intermediate recombination and defines the offspring to be the center of mass of its parents. Note that this is the only variation operator that has no random component.

Crossover for S_n There is a variety of crossover operators for permutations. Almost all of them create an offspring based on exactly two parents. Most often, two positions are selected uniformly at random and the elements between these two positions in the first parent are reordered according to the ordering in the second parent. Examples include order crossover, partially mapped crossover (PMX), and cycle crossover (CX). We name these examples without giving precise definitions. The reason is that for permutations no successful standard

crossover with convincing performance across a wide variety of different permutation problems is known. It is worth mentioning that for specific applications custom-tailored crossover operators have been suggested. Examples are edge recombination and inver-over, both designed for the traveling salesperson problem (TSP).

2.1.5 Termination Criterion

The termination criterion is the final step in the evolutionary cycle (see Fig. 2.1). It decides if the algorithm is to be stopped or if another generation is to be started. In the case of stopping, usually an output is produced. In most cases evolutionary algorithms are used for maximizing the fitness function f, and consequently some $x \in S$ with maximal fitness among all visited search points is presented. To achieve this, the current best is usually stored in addition to and independent of the current population. Termination criteria can be more or less flexible. We describe different classes of termination criteria without going into detail. For our goals it is helpful to consider the most basic and simple termination criterion, i.e., avoiding the issue altogether by using no termination criterion at all.

Adaptive termination criteria These termination criteria may take anything into account. Being completely unrestricted implies that they are the most flexible way of deciding about termination. This may depend on directly observable properties like the population and its fitness values as well as on statistics based on such properties. A typical example for such an adaptive termination criterion is the number of generations since the last improvement. Also, simpler criteria, like stopping once the best fitness value found is beyond some predetermined value, fall into this category.

Fixed termination criteria In practice, in many cases much simpler stopping criteria are used. Fixed termination criteria stop the algorithm after a predefined number of steps or computations regardless of the actual run. Concrete examples include stopping after a predefined number of generations or a predefined number of fitness evaluations.

No termination criterion When we consider evolutionary algorithms from a theoretical point of view we avoid the topic of choosing a termination criterion and simply let the evolutionary cycle continue without stopping. In a formal sense, we let the algorithms run forever. What we are interested in is the first point of time T when a global maximum of f is found, where time is measured by the number of function evaluations. We call this point of time the *optimization time* and are mostly interested in its mean value. For practical purposes this is an important measure. It tells us how long we have to wait, on average, in order to find an optimal solution.

2.2 Parameters of Evolutionary Algorithms

It is already apparent that evolutionary algorithms have a number of parameters that need to be set. Apart from the parameters belonging to some module as defined in the previous section, there are also more global parameters. We list these parameters here and briefly discuss their role. The main purpose is to introduce the notation that is used throughout the text. Note that in the evolutionary computation community there is no common notation in use. Here, we only introduce the most basic and important parameters. Some evolutionary algorithms make use of additional parameters. These will be introduced as needed.

Dimension of the search space n We use n to measure the size of the search space for $\{0,1\}^n$, $\mathbb{R}^n$, and S_n. Clearly, n is not really a parameter of the evolutionary algorithm but a property of the search space and thus the fitness function. When analyzing (randomized) algorithms, one usually considers the performance of the algorithm depending on the size of the input. Most often, results are expressed in an asymptotic form (using the Landau notation, see Appendix A.1) and one assumes that the size of the input grows to infinity. For us, n plays the role of the input size. Therefore, we analyze the performance of evolutionary algorithms for $n \to \infty$. Another important role of the dimension of the search space is to determine the value of parameters of the evolutionary algorithms. Often such parameters are fixed with respect to n. One example is the mutation probability p_m, which is most often set to $p_m = 1/n$.

Population size μ The number of individuals in the population, in particular right after initialization, is called the population size μ. There are no clear rules for setting the population size. Clearly, μ needs to be bounded above in a reasonable way. If we adopt the usual perspective of complexity theory, it makes sense to have μ polynomially bounded, i.e., $\mu = n^{O(1)}$. Typical choices include $\mu = O(n)$ or $\mu = O(\sqrt{n})$, but also much smaller populations like $\mu = O(1)$ or even $\mu = 1$ are not uncommon.

Offspring population size λ The number of offspring generated in each generation is called the offspring population size. Clearly, $\lambda = n^{O(1)}$ is reasonable. Again, a wide variety of different values for the offspring population size are commonly used. In particular, $\lambda = 1$ is a very common choice (even more common than $\mu = 1$), but also having $\lambda \gg \mu$ (like $\lambda = n \cdot \mu$) is not unusual. Of course, what values for λ make sense also depends on the selection mechanism employed.

Crossover probability p_c We denote the probability that crossover is used to produce an offspring as crossover probability p_c. Any choice $p_c \in [0,1]$ is possible; the most often recommended choices are quite large constant values like $p_c \in [0.5, 0.8]$. In some evolutionary algorithms either crossover or mutation is applied. In these cases we apply crossover with probability p_c and consequently mutation with probability $1 - p_c$. Here we do not do that. Instead, we decide about the application of crossover and mutation independently and have offspring created by means of crossover undergo a subsequent mutation with a certain probability that is called the probability for mutation.

Probability for mutation Probability for mutation is the probability to produce an offspring via application of a mutation operator. We mention this parameter without actually using it. In the algorithms that we consider we always set the probability for mutation to 1. It is important not to mix this up with the mutation probability p_m, a parameter of standard bit mutation. Using the most commonly used mutation probability $p_m = 1/n$ implies that with probability $(1 - 1/n)^n \approx 1/e$ no bit is flipped. Thus, there is no need to ensure that there are steps without actual mutations by introducing a probability for mutation. It is important to note that some authors use 'mutation probability' to denote the probability for mutation and introduce the notion of 'mutation strength' or 'mutation rate' for our mutation probability.

There are different ways of setting the parameters in evolutionary algorithms. The simplest and most common way is to set all parameters to some fixed values in advance and keep the parameters unchanged during the run. Usually, one experiments a little to find settings that lead to acceptable performance. It is not clear, however, that such static parameter settings are able to deliver satisfactory results at all. It may well be the case that in the beginning of the run the fitness function presents the evolutionary algorithm with a landscape that makes certain values for some parameters mandatory while at some later time the population searches in other regions of the search space where other values for the parameters are to be preferred. This motivates having the parameter settings vary with time. This way of setting the parameters is called dynamic parameter setting. In implementations, the number of generations or the number of fitness evaluations are used as measure for time. While dynamic parameter setting is quite common for some randomized search heuristics, it is seldom used in evolutionary algorithms. One example, however, that we already mentioned is the temperature T in Boltzmann selection. In evolutionary algorithms, if parameter values are to change during the run, in most cases a more complicated mechanism is used. In principle, any information available may be used to decide upon the new values of parameters. Typical examples of the kind of information used include the current population, the population's distribution in the search space, the current fitness values, or measures based on a history of the population. A concrete example would be the number of generations since an offspring with better fitness than its parent has been produced. A common strategy is to increase the mutation probability if such an event has not occurred for a predefined number of generations. Generally, any method depending on any individual created and any random experiment performed during the complete run is permissible. This complex way of setting the parameters is called adaptive parameter settings.

We observe that these methods of setting the parameters form a hierarchy as depicted in Fig. 2.2. Adaptive parameter settings form the most flexible and largest class of methods for setting the parameters. Dynamic parameter settings are restricted methods that may only depend on the time but not on the many other available data. It may therefore be adequately described as a special case of adaptive parameter setting. In the same way, static parameter settings can be described as a

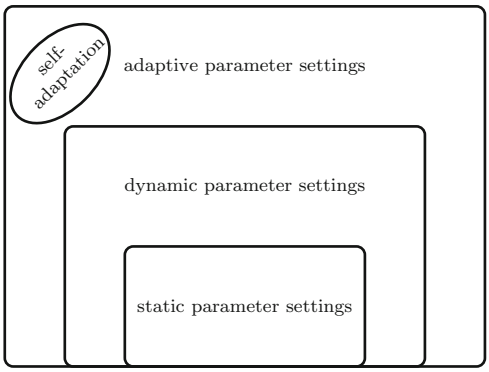

Fig. 2.2 Hierarchy of parameter setting methods

special case of dynamic parameter settings, where the function that depends on time is constant. One may say that static parameter settings are a degenerate form of dynamic parameter settings.

One special way of adaptively setting the parameters deserves to be mentioned: self-adaptive parameter settings. Self-adaptation means that the parameter settings are evolved together with the population in some way. Typically, the parameter values are encoded as part of the individuals, formally increasing the size of the search space. We can describe this by having our new individuals live in $S \times Q$, where S is the original search space and Q is the set of possible parameter settings. Note that the fitness $f : S \to \mathbb{R}$ is still defined on S, and therefore the parameter settings have no direct influence on the fitness. In each generation, first variation operators are applied to the parameter settings. Then these offspring parameter values are used to parameterize the variation operators applied to points in S. Selection is done in the usual way. The idea is that good values for the parameters have the tendency to create good offspring. Since good offspring are typically selected, good values for the parameters indirectly have better chances to survive and be present in the next generation. This is the reason why the parameter values have to be subject to variation first. Otherwise we have random changes of parameter values that have no influence at all on the selection, and self-adaptation would probably not work that well.

2.3 Typical Evolutionary Algorithms

The 'invention' of evolutionary algorithms dates back to the 1960s. Different research groups suggested independently of each other quite similar randomized search heuristics all inspired by natural evolution. All had the idea to devise interesting and useful new algorithms by mimicking natural processes in algorithmic form. For us it is not difficult to identify these different algorithms as variants of the same algorithm family, as different kinds of evolutionary algorithms. And yet it was

a long and sometimes difficult process for the researchers involved to recognize that others developed very similar ideas. This can still be seen today and is apparent in different names for certain kinds of evolutionary algorithms. One may consider having separate names for these different variants useful since it seems to allow us to describe rather complicated evolutionary algorithms in a very short and precise way. This, however, is actually not an accurate description of the status quo. Nobody really uses the algorithmic variants devised nearly 50 years ago. There are so many variations of all kinds of evolutionary algorithms around that it is practically impossible to infer the details of an evolutionary algorithm just from its name. Since the historic names are still around, we give a short description and describe typical variants. We conclude this section by giving precise definitions of an evolutionary algorithm that we will consider in great detail in the chapter on methods for the analysis of evolutionary algorithms (Chap. 5). We describe all algorithms in a way that make them suitable for maximization of a fitness function f. This agrees with the intuitive idea that fitness should be maximized. Clearly, minimization of $-f$ is equivalent to maximization of f and thus considering only maximization is no restriction.

In the United States, Holland [50] devised genetic algorithms (GAs). Most often they are defined for the search space $\{0,1\}^n$; the variation operator that is considered to be the most important one is crossover. Often k-point crossover with $k \in \{1,2\}$ is used with quite high probability, i.e., $p_c \geq 1/2$. Mutation, typically standard bit mutations with small mutation probability, is considered to be a kind of 'background operator' that merely generates enough diversity in the population to facilitate crossover. Selection for reproduction is usually done using fitness-proportional selection. In his original work Holland does not mention selection for replacement; in fact, (μ, μ) selection is applied.

In Germany, Schwefel [114] and Rechenberg [107] devised evolution strategies (ESs). Most often they are defined for the search space $\mathbb{R}^n$; the most important variation operator is mutation. Often there is no crossover at all, and if crossover is to be used it is usually intermediate crossover. In the beginning there was no explicit discussion about selection for reproduction; uniform selection is used for this purpose. Selection for replacement is usually implemented either as comma- or plus-selection. For evolution strategies, non-static ways of setting the parameters have been in use almost from the very beginning. In particular, self-adaptive methods were used very early on.

In the United States, Fogel et al. [41] devised evolutionary programming (EP). Originally, the set of finite state automata was used as search space. This may be seen as the most ambitious approach. The complex structure of the search space led to an emphasis of mutation (of course, mutation operators suitable for mutating finite state automata had to be developed). Later developments led evolutionary programming increasingly close to evolution strategies.

Quite some time later, Koza [73, 74] and Koza et al. [75, 76] introduced genetic programming (GP). Genetic programming uses evolutionary algorithms that are modeled after genetic algorithms but use a different search space (often the set of all s-expressions) and aim at developing programs as individuals.

Considering the different streams that together led to the class of evolutionary algorithms, it comes as no surprise that there are many different variants of evolutionary algorithms. Some variants, however, are more common than others. We consider some of the more basic, well-known, and therefore in some sense important ones here.

One algorithm that is often singled out is the so-called simple GA. It is a genetic algorithm operating on the search space $\{0,1\}^n$, fitness-proportional selection for reproduction, 1-point crossover with crossover probability $p_c \in [0.5, 0.9]$, standard bit mutations with mutation probability $p_m \leq 1/n$, and (μ, μ) selection for replacement. Sometimes crossover is described as producing two offspring but this varies from author to author. The simple GA was first described by Goldberg [46] and later analyzed by Vose [125].

An evolutionary algorithm that we are going to analyze in considerable detail is a simple algorithm that we call $(\mu+\lambda)$ EA. It operates on the search space $\{0,1\}^n$, has population size μ, offspring population size λ, uniform selection for reproduction, no crossover, standard bit mutation with mutation probability $p_m = 1/n$, and $(\mu+\lambda)$-selection for replacement. Since this algorithm is central for our analysis, we give a precise formal definition.

Algorithm 1 ($(\mu+\lambda)$ EA).

1. **Initialization**
   Choose $x_1, x_2, \ldots, x_\mu \in \{0,1\}^n$ uniformly at random.
   Collect $x_1, x_2, \ldots, x_\mu$ in P_0. $t := 0$.
2. Repeat for $i \in \{1, 2, \ldots, \lambda\}$
3. **Selection for Reproduction**
   Select $y \in P_t$ uniformly at random.
4. **Variation**
   Create y_i by standard bit mutation of y with $p_m = 1/n$.
6. **Selection for Replacement**
   Sort all $x_1, x_2, \ldots, x_\mu \in P_t$ and $y_1, y_2, \ldots, y_\lambda$ in descending order according to fitness, breaking ties first by preferring offspring and breaking still unresolved ties uniformly at random.
   Collect the first μ individuals in P_{t+1}.
7. $t := t + 1$. Continue at line 2.

The $(\mu+\lambda)$ EA implements plus-selection with a slight preference for the offspring: if parents and offspring have equal fitness, the offspring is preferred. So, if we have three parents x_1, x_2, x_3 with fitness values $f(x_1) = 2$, $f(x_2) = 7$, $f(x_3) = 4$, and three offspring y_1, y_2, y_3 with fitness values $f(y_1) = 4$, $f(y_2) = 6$, $f(y_3) = 1$, then the ordering is x_2, y_2, y_1, x_3, x_1, y_3.

We know that evolutionary algorithms are particularly difficult to analyze since they are not designed with analysis in mind. Thus, it makes sense to start with particularly simple evolutionary algorithms. This way we can hope to have manageable objects of studies and develop tools for their analysis that turn out to be suitable for the analysis of more complex evolutionary algorithms, too.

This motivates considering the $(\mu+\lambda)$ EA with minimum population size and offspring population size, i.e., $\mu = \lambda = 1$. These settings lead to a simpler formal description of the (1+1) EA.

Algorithm 2 ((1+1) EA).

1. **Initialization**
   Choose $x_0 \in \{0,1\}^n$ uniformly at random.
   $t := 0$.
2. **Variation**
   Create y by standard bit mutation of x_t with $p_m = 1/n$.
3. **Selection for Replacement**
   If $f(y) \geq f(x_t)$, then $x_{t+1} := y$ else $x_{t+1} := x_t$.
4. $t := t + 1$. Continue at line 2.

Note that we described these two evolutionary algorithms without any stopping criterion. Remember that this is the most common approach when analyzing evolutionary algorithms. We are interested in the optimization time T, i.e., $T := \min\{t \in \mathbb{N}_0 \mid f(x_t) = \max\{f(x') \mid x' \in \{0,1\}^n\}\}$ for the (1+1) EA.

2.4 Other Simple Randomized Search Heuristics

Evolutionary algorithms are by no means the only kind of general randomized search heuristics. There is a plethora of different randomized search heuristics ranging from very simple (like pure random search) to quite complicated and sophisticated (like particle swarm optimization or ant colony optimization). Since our interest is in evolutionary algorithms, we will not discuss other randomized search heuristics in great detail. It makes sense, however, to compare the performance of evolutionary algorithms with that of other randomized search heuristics in order to get a better understanding of their specific strengths and weaknesses. We consider five of the simpler randomized search heuristics for this purpose. Another motivation for discussing these other randomized search heuristics is to develop an understanding of the way a borderline could be drawn separating evolutionary algorithms from other randomized search heuristics. Due to the flexibility of the algorithmic concept 'evolutionary algorithm', it is impossible to come to a clear and indisputable distinction. But we will be able to give reasons for calling some simple randomized search algorithm like the (1+1) EA an evolutionary algorithm while we consider quite similar algorithms to be of a different kind. Finally, structurally simpler randomized search heuristics can serve as a kind of stepping stone. Proving results about their behavior can provide valuable insights about how a proof for the more complex evolutionary algorithms can be obtained. We describe all randomized search heuristics for the maximization of some function $f: \{0,1\}^n \to \mathbb{R}$.

Pure random search Search starts with some $x \in \{0,1\}^n$ chosen uniformly at random. In each step, another point $y \in \{0,1\}^n$ is chosen uniformly at random that replaces x.

This kind of blind search may, of course, be described as an (1, 1) EA with standard bit mutation with mutation probability $p_m = 1/2$. But this would be misleading. It differs in two important points from evolutionary algorithms: the search is not guided by the fitness values encountered, and the search has no locality whatsoever.

Pure random search is a very simple and almost always very bad search heuristic, of course. Its only advantage is that it is extremely easy to analyze. We may use it as a very weak competitor. If an evolutionary algorithm is not even able to clearly outperform pure random search, it is definitely not doing much good for the fitness function under consideration.

Random local search Search starts with some $x \in \{0,1\}^n$ chosen uniformly at random. In each step, another point $y \in N(x)$ is chosen uniformly at random where $N(x)$ denotes some neighborhood of x. Then, y replaces x if $f(y) \geq f(x)$ holds.

Often, the neighborhood $N(x) = \{x' \in \{0,1\}^n \mid \mathrm{H}(x,x') = 1\}$ is used. We call this the *direct Hamming neighborhood*. Sometimes, larger neighborhoods like $N(x) = \{x' \in \{0,1\}^n \mid \mathrm{H}(x,x') = 2\}$ are more useful. Almost always one has neighborhoods of at most polynomial size, i.e., $|N(x)| = n^{O(1)}$.

Random local search with the direct Hamming neighborhood, i.e., the neighborhood $N(x)$ consists of only the Hamming neighbors of x, could be described as (1+1) EA with 1-bit mutations instead of standard bit mutations. This difference, however, is crucial. Random local search can be trapped in local optima of the fitness landscape where all Hamming neighbors have smaller fitness values. Random local search cannot move anywhere from such a point. For the (1+1) EA, no such traps exist. It can reach any point in the search space by one mutation with positive (yet very small) probability. This ability to perform global search is typical for evolutionary algorithms. This is why we draw a borderline here.

Nevertheless, random local search and the (1+1) EA (as defined as Algorithm 2) are very similar. Often random local search is much easier to analyze. Since the performance of random local search and the (1+1) EA is often similar, analyzing random local search can be a helpful first step toward an analysis of the (1+1) EA.

Iterated local search In iterated local search we carry out several local search runs subsequently. Each random local search is carried out as described above. Every time the search gets stuck in a local optimum it is restarted with some new starting point x chosen uniformly at random from $\{0,1\}^n$.

We note that iterated local search requires considering the complete neighborhood $N(x)$ in order to decide whether the search got stuck. Clearly, this implies increased computation time for something that we consider one 'round' of the algorithm. However, this extension to random local search is so simple and yet such a significant improvement that it is worth mentioning. Moreover, such restarts may be added to any kind of random search heuristics; in particular they may be used in combination with evolutionary algorithms. There, however, it is more difficult to find an appropriate criterion for triggering a restart.

Metropolis algorithm Search starts with some $x \in \{0,1\}^n$ chosen uniformly at random. In each step another point $y \in N(x)$ is chosen uniformly at random, where $N(x)$ denotes some neighborhood of x. Then, y replaces x with probability $\min\left\{1, e^{(f(y)-f(x))/T}\right\}$.

As for random local search, the most common neighborhood consists just of the Hamming neighbors of x. The parameter $T \in \mathbb{R}^+$ is called temperature; it is fixed in advance and held constant during the complete run. The term $\min\left\{1, e^{(f(y)-f(x))/T}\right\}$ equals 1 if $f(y) \geq f(x)$ holds. So, improvements in fitness are always accepted here. This coincides with random local search. But for $f(y) < f(x)$, the two search heuristics differ. While such a move from x to y is never done with random local search, it may be done in the Metropolis algorithm. The probability, however, depends on the parameter T and the difference in fitness values $f(x) - f(y)$. With increasing difference the probability for such a step decreases exponentially. The selection mechanism helps prevent getting stuck in local optima.

Simulated annealing Simulated annealing is almost identical to the Metropolis algorithm, but the fixed temperature T is replaced by some function $T: \mathbb{N}_0 \to \mathbb{R}^n$ that is called annealing schedule and that depends on time, i.e., on the current generation.

Clearly, we may describe the Metropolis algorithm as simulated annealing with fixed temperature. Thus, simulated annealing is an example of an algorithm making use of a dynamic parameter-setting scheme, whereas the Metropolis algorithm is the same algorithm but utilizing a static parameter setting.

Usually, one uses strictly decreasing annealing schedules T. In fact, simulated annealing is inspired by the process of annealing in metallurgy. There, metal that is heated beyond its recrystallization temperature is cooled sufficiently slowly such that it is allowed to enter an energy-minimizing state. When using simulated annealing for finding points with large fitness values in the search space, the idea is the following. In the beginning, fitness values are quite bad, and it should be simple to escape from local optima. This is analogous to using a high temperature. Later on, the algorithm has probably found much more promising regions of the search space that should not be left easily. This is analogous to lower temperatures. Given a certain optimization problem, to find an appropriate annealing schedule is as crucial for the performance of simulated annealing as it is difficult to achieve.

2.5 Design of Evolutionary Algorithms

Our main focus when considering evolutionary algorithms is analysis. We aim at considering common evolutionary algorithms and finding out how they perform on different problems. When one wants to apply evolutionary algorithms, the perspective is necessarily different. In this case, one wants to design an evolutionary algorithm that is appropriate and efficient for a given problem class. We consider this

situation here and discuss aspects that stem from a theoretical perspective and that should be taken into account.

We restricted our description of modules for evolutionary algorithms to the three search spaces $\{0,1\}^n$, $\mathbb{R}^n$, and S_n. In principle, we could even restrict the discussion to the search space $\{0,1\}^n$. Evolutionary algorithms are (like all algorithms nowadays) implemented on computers using binary representations internally. Thus, any nonbinary data like real numbers has to be mapped to a binary representation at some level. One may therefor argue that we may as well 'lift' this mapping to the topmost level, perform it ourselves, and deal only with binary representations from now on. However, such reasoning does usually not take the complexity of the actual mappings involved into account. It is similar to arguing that a modern computer is in principle nothing more than a finite automata with a huge state space. While this is true in a formal sense, we know that it is much more appropriate to describe a modern computer as a Turing machine. Properties and complexity are better captured this way. Similarly, it makes more sense to consider evolutionary algorithms on different search spaces.

We are interested in evolutionary algorithms because they have proven to be very useful in many practical applications. Without practical applications there would be no point in doing theory. For some practical problems it is actually possible to find an appropriate formalization as function $f: \{0,1\}^n \to \mathbb{R}$, $f: \mathbb{R}^n \to \mathbb{R}$, or $f: S_n \to \mathbb{R}$ where f is to be maximized. One particularly good example is the satisfiability problem SAT that is of practically importance in many applications and at the same time one of the most prominent problems in theoretical computer science. If we consider SAT for n variables it is easy for any concrete SAT instance to describe a corresponding function $f_{\text{SAT instance}}: \{0,1\}^n \to \mathbb{N}_0$ that yields the number of clauses satisfied by an assignment $x \in \{0,1\}^n$ of the n variables. Similarly, for each instance of the traveling salesperson problem (TSP) with n cities, we can define a corresponding function $f_{\text{TSP instance}}: S_n \to \mathbb{R}$ that yields the length of the tour described by a permutation π for each permutation $\pi \in S_n$. But here we encounter a first (easy to solve) problem. While evolutionary algorithms aim at maximizing a fitness function, the function $f_{\text{TSP instance}}$ needs to be minimized. As discussed above, this can easily be fixed by replacing $f_{\text{TSP instance}}$ by $-f_{\text{TSP instance}}$. But this yields a fitness function with only negative function values, making, for example, the use of fitness proportional selection impossible. A different way of dealing with a minimization problem would be to replace the kind of selection employed, using selection for replacement where selection for reproduction is intended, and vice versa. But this would in some sense be against one of the main ideas of evolutionary algorithms. Evolutionary algorithms are attractive to practitioners since there is no need for modifications of the algorithm, since they are easy off-the-shelf solutions to a wide variety of different problems. Such ad hoc modifications of the algorithm become even more complicated if the fitness function is not defined over one of the standard search spaces. Adopting a novel kind of search space implies that new variation operators need to be defined. Then the design of new evolutionary algorithms becomes as complicated and as time consuming as the design and implementation of any new problem-specific search heuristic. Thus, it is advisable to

follow a different route. In the following, we discuss such a solution to our problem that, from the point of view of computer science, is more structured than ad hoc modifications of the algorithm. Moreover, it has the additional charming property of having a corresponding mechanism in the natural paradigm.

Let us assume that we are dealing with an optimization problem that is modeled as either maximization or minimization of some function $g\colon A \to B$. Here, A may be an arbitrary set, B is some set that allows for some way of evaluating solutions. Thus, there needs to be at least a partial ordering defined on B. We want to solve this optimization problem by means of some evolutionary algorithm. In order to do so, we define two mappings $h_1\colon S \to A$ and $h_2\colon B \to \mathbb{R}$, where S is the search space of our evolutionary algorithm. The idea is to have the search space S equal to some standard search space so that we have an evolutionary algorithm operating on S ready to use. We define a fitness function f for our evolutionary algorithm by $f := h_2 \circ g \circ h_1$. This implies that in order to compute the fitness for some point $s \in S$, we first map s via h_1 to $h_1(s) \in A$, compute its value $g(h_1(s)) \in B$, and, by means of h_2, map this value to a fitness value $h_2(g(h_1(s))) \in \mathbb{R}$.

Following the natural paradigm, the set S is often called phenotype space and A is called genotype space. Accordingly, h_1 is called genotype–phenotype mapping. The idea is that our 'genetic' algorithms operate on the genotypes, whereas in nature fitness (in the sense of survivability) is, of course, expressed on the level of phenotypes. So having some mapping from the genotypes to the phenotypes that is involved in determining the fitness of some individual is something that we may see in nature.

In principle, we are free to choose h_1 and h_2 any way we like. Obviously, we have to make sensible choices in order to arrive at an evolutionary algorithm that works well for our problem. Probably the most basic requirement is that h_2 is a function that needs to be maximized in order to find optimal solutions to g. Moreover, h_1 and h_2 need to be computable efficiently, evaluation via h_2 needs to have a good correspondence to evaluation via g, and h_1 needs to map to as much of A as possible. If we choose h_1 in a unfavorable way, it may happen that optimal solutions in A have no preimage in S and thus cannot be found be the evolutionary algorithm at all.

All this advice is basically trivial. In practice, however, it may be highly nontrivial to follow this advice. Nevertheless, we discuss even more guidelines that all aim at delivering a well-functioning evolutionary algorithm. These guidelines come with the advantage of being less trivial: they are useful advice that is substantial and could hardly be found with just a few minutes of thinking. While following them in practice may be difficult, it pays to know about them in order to avoid making mistakes that have been made many times before by others.

As we pointed out when discussing different variation operators, the main idea in evolutionary algorithms is to search for promising new search points quite close to the points of the current population. Since our variation operators work in genotype space S but fitness assessment is done in phenotype space A, it is desirable that small changes in genotype space correspond to small changes in phenotype space. If there is no such correspondence between changes in S and A, we have departed (at least implicitly, perhaps unknowingly) from the idea of evolutionary algorithms.

It is possible that the evolutionary algorithm that we obtain still works—but from a fundamental point of view it should not. In order to make our ideas more precise, we need a way of measuring changes. This is done by means of some metric d. For the sake of completeness, we recall the definition of a metric.

Definition 2.1. For a set M a mapping $d: M \times M \to \mathbb{R}_0^+$ is called a *metric* on M if it has the following three properties:

1. Positivity	$\forall x, y \in M: x \neq y \Leftrightarrow d(x, y) > 0$
2. Symmetry	$\forall x, y \in M: d(x, y) = d(y, x)$
3. Triangle inequality	$\forall x, y, z \in M: d(x, y) + d(y, z) \geq d(x, z)$

We assume that some metric d_A for A is known. This metric is a formal way to express domain knowledge that a user has. For candidate solutions $a_1, a_2 \in A$ to the practical problem $g: A \to B$, the user is expected to be able to describe their similarity. If our genotype–phenotype mapping h_1 happens to be injective, then we obtain a metric d_S on S by means of h_1 via $d_S(x, y) := d_A(h_1(x), h_1(y))$ for all $x, y \in S$. If h_1 is not injective, we cannot define d_S this way since this would violate the positivity constraint. In this case a metric d_S that reflects d_A has to be defined some other way. From now on we simply assume that some metric d_S on S is defined, and we demand that for all $x, x', x'' \in S$

$$\begin{aligned} & d_S(x, x') \leq d_S(x, x'') \\ \Rightarrow \quad & d_A(h_1(x), h_1(x')) \leq d_A(h_1(x), h_1(x'')) \end{aligned}$$

holds. We call this property monotonicity. Clearly, monotonicity guarantees the connection between the magnitudes of changes that we desire.

Based on this metric d_S, we can now formalize our requirements for variation operators. This helps not only to check whether our genotype–phenotype mapping h_1 and the metric d_S are appropriate when applying some evolutionary algorithm to some practical optimization problem. It also enables us to discuss the appropriateness of newly designed variation operators in an objective way.

For our discussion we want to describe mutation and crossover as randomized functions that we define in the following way. Assume we have a mapping $r: X \to Y$, where the image $r(x) \in Y$ depends not only on $x \in X$ but also on some random experiment, i.e., $r(x)$ is a random variable. The random experiment is defined on some probability space (Ω, p). We define the mapping r as $r: X \times \Omega \to Y$, where $r(x, \omega) = y$ if applying r to x together with $\omega \in \Omega$ as result of a random experiment yields y. For $x \in X$ and $y \in Y$, we let $\text{Prob}\,(r(x) = y) = \sum_{\omega \in \Omega: r(x,\omega) = y} p(\omega)$. We simplify our notation by omitting ω and assume that the probability space (Ω, p) is clear from the context.

For the sake of clarity, we discuss a simple example and consider 1-bit mutation as a randomized mapping $m: \{0, 1\}^n \to \{0, 1\}^n$. Since we flip exactly one bit and choose the position of this bit uniformly at random, (Ω, p) with $\Omega = \{1, 2, \ldots, n\}$ and $p(i) = 1/n$ for all $i \in \Omega$ is an appropriate probability space. For $x \in \{0, 1\}^n$ and

$i \in \Omega$ we define $m(x,i) = x \oplus \left(0^{i-1}10^{n-i}\right)$, where we use the following notation. For $b \in \{0,1\}$ and $j \in \mathbb{N}_0$ let b^j denote the concatenation of j times the letter b. In particular, let b^0 denote the empty word. Thus, $0^2 1 0^3 = 001000$ holds. The operation $\oplus$ applied on $\{0,1\}^n$ stands for bitwise XOR, where for $a, b \in \{0,1\}$ XOR is defined via $\mathrm{XOR}(a,b) = a+b-2ab$. It is easy to see that this way m corresponds exactly to our understanding of 1-bit mutation.

Using this notation as we did in the example, we can proceed and define our requirements for variation operators. We begin with mutation operators that we describe as randomized mappings $m\colon S \to S$.

We want a mutation operator m to favor small changes. We express this by demanding that

$$\begin{aligned} \forall x, x', x'' \in S\colon \quad & d_S(x,x') < d_S(x,x'') \\ \Rightarrow \quad & \mathrm{Prob}\left(m(x) = x'\right) > \mathrm{Prob}\left(m(x) = x''\right) \end{aligned}$$

holds. If x' is closer to x than x'' is, than it should be more likely to obtain x' as offspring of x then x''.

A second reasonable requirement is to have mutation operators not induce a search bias. We want the search to be guided by the fitness values which are taken into account by selection. This can be expressed via

$$\begin{aligned} \forall x, x', x'' \in S\colon \quad & d_S(x,x') = d_S(x,x'') \\ \Rightarrow \quad & \mathrm{Prob}\left(m(x) = x'\right) = \mathrm{Prob}\left(m(x) = x''\right) \end{aligned}$$

in a formal and precise way.

For crossover, we can proceed in a similar way. Crossover is described as a randomized mapping $c\colon S \times S \to S$. We consider crossover operating on two parents and producing exactly one offspring. It is not difficult to generalize this to crossover operators using more parents. We express the idea that an offspring should be similar to its parent by having

$$\begin{aligned} \forall x, x', x'' \in S\colon \quad & \mathrm{Prob}\left(c(x,x') = x''\right) > 0 \\ \Rightarrow \quad & \max\left\{d_S(x,x''), d_S(x',x'')\right\} \leq d_S(x,x') \end{aligned}$$

for crossover. The distance from the offspring to any of its parents is bounded above by the distance of the parents.

Clearly, we do not want crossover to induce a search bias. We express this by requiring that

$$\forall x, x' \in S\colon \forall \alpha \in \mathbb{R}_0^+\colon \mathrm{Prob}\left(d_S(x, c(x,x')) = \alpha\right) = \mathrm{Prob}\left(d_S(x', c(x,x')) = \alpha\right)$$

holds. The offspring is required to be symmetric with respect to its parents in terms of distance.

It is important to understand that these requirements are not 'true' in the sense that following these rules necessarily leads to better evolutionary algorithms or that

violating a requirement implies poorer performance. And yet they should be taken seriously since they have the benefit of formalizing our intuitive understanding of evolutionary computation. Following them helps us not to depart from the paradigm of evolutionary computation. When applied during the creative act of defining new variation operators, they guide us and help us by presenting us with a formalism that leads us to more objectively justified and, in any case, more conscious design decisions. Moreover, they facilitate the formal analysis of evolutionary algorithms that are designed respecting these guidelines.

2.6 Remarks

While the description of the evolutionary cycle gives an accurate picture of the structure of evolutionary algorithms, our actual list of modules for evolutionary algorithms is very short and contains only the most basic examples. This is due to our emphasis on the analysis of evolutionary algorithms. A much more comprehensive view is presented in [9] and, more recently, in [110]. With respect to the mutation operators for permutations, it is worth mentioning that jump and exchange, while quite general in nature, have been designed for the problem of sorting [132].

With respect to tournament selection we remarked that there is no need to have all members of the current population evaluated as is the case for all selection mechanisms. Poli [98] suggests an evolutionary algorithm that takes advantage of this fact and reduces the number of function evaluations if the tournament size is small and the number of generations is not too large. This is one concrete example where the explicit inclusion of function evaluations within the evolutionary cycle would contradict an evolutionary algorithm as suggested to improve efficiency.

The analytical framework we pursue is inspired by the classical analysis (and design) of efficient randomized algorithms. Excellent textbooks providing introduction and overview include [15, 88].

In practical applications, finding good parameter settings is crucial. Bartz-Beielstein [10] provides a framework for a systematic approach to this problem. The classification of mechanisms to set and control parameters during a run (presented in Fig. 2.2) is structured in a way that a hierarchy is formed. This makes self-adaptation a special case of adaptive parameter settings. Historically, self-adaptation has been described as an alternative to adaptive parameter settings (see [8, 38]). While this may be useful to make a point and propagate self-adaptive parameter settings as ‘natural’ for evolutionary algorithms, it is, logically speaking, misleading.

Local search is a very well known and popular randomized search heuristic by itself. It is not covered in this text in any depth. Interested readers may want to consult [2, 83] for a more appropriate exposition of local search. The Metropolis algorithm and simulated annealing are also two popular randomized search heuristics. Useful references include [1, 55, 63, 72, 82]. The design methodology for evolutionary algorithms based on a metric was first presented by Droste and Wiesmann [29].

http://www.springer.com/978-3-642-17338-7

Analyzing Evolutionary Algorithms
The Computer Science Perspective
Jansen, Th.
2013, X, 258 p., Hardcover
ISBN: 978-3-642-17338-7