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.

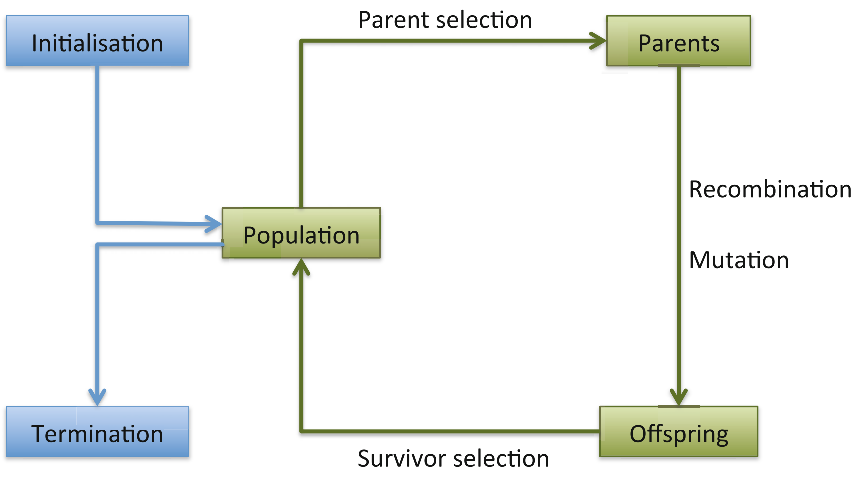

Fig. 3.2. The general scheme of an evolutionary algorithm as a flowchart

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), finite 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.

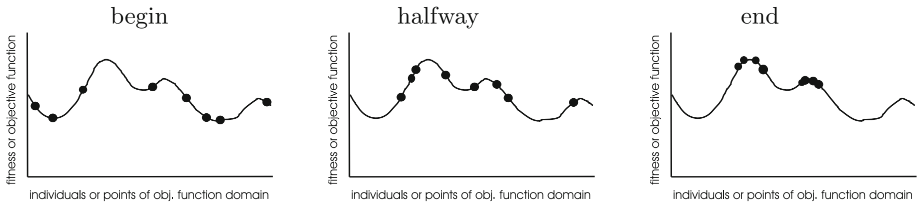

Fig. 3.4. Typical progress of an EA illustrated in terms of population distribution. For each point x in the search space y shows the corresponding fitness value.

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

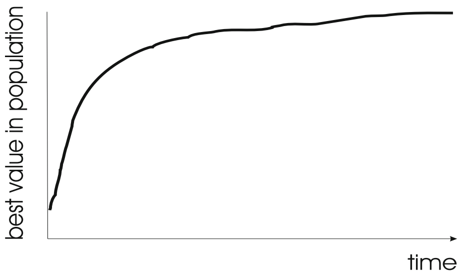

Fig. 3.5. Typical progress of an EA illustrated in terms of development over time of the highest fitness in the population

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

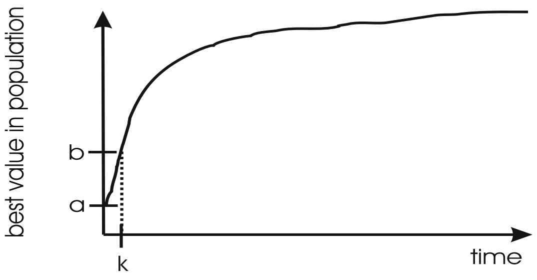

Fig. 3.6. Illustration of why heuristic initialisation might not be worth additional effort. Level a shows the best fitness in a randomly initialised population; level b belongs to heuristic initialisation

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

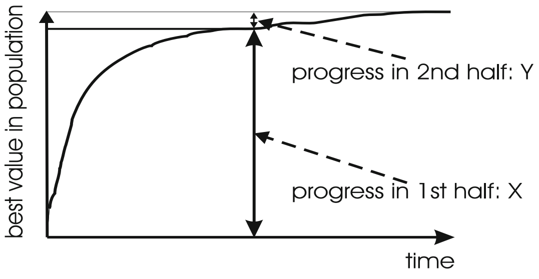

Fig. 3.7. Why long runs might not be worth performing. X shows the fitness increase in the first half of the run, while Y belongs to the second half

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.

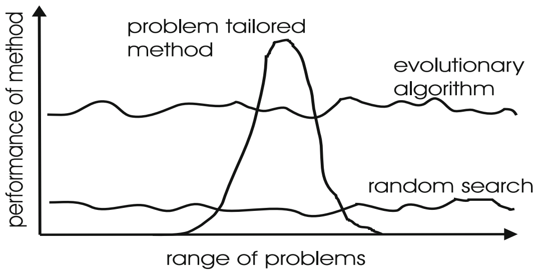

Fig. 3.8. 1980s view of EA performance after Goldberg [189]

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 probability 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`.