2

What is an Evolutionary Algorithm?

2.1 Aims of this Chapter

The most important aim of this chapter is to describe what an Evolutionary Algorithm is. This description is deliberately based on a unifying view presenting a general scheme that forms the common basis of all Evolutionary Algorithm variants. The main components of EAs are discussed, explaining their role and related issues of terminology. This is immediately followed by two example applications (unlike other chapters, where example applications are typically given at the end) to make things more concrete. Further on we discuss general issues for EAs concerning their working. Finally, we put EAs into a broader context and explain their relation with other global optimisation techniques.

2.2 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 the environmental pressure causes natural selection (survival of the fittest) and this 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, and apply the quality function as an abstract fitness measure – the higher the better. Based on this fitness, some of the better candidates are chosen to seed the next generation by applying recombination and/or mutation to them. Recombination is an operator applied to two or more selected candidates (the so-called parents) and results one or more new candidates (the children). Mutation is applied to one candidate and results in one new candidate. Executing recombination and mutation leads to a set of new candidates (the offspring) that 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 with sufficient quality (a solution) is found or a previously set computational limit is reached.

In this process there are two fundamental forces that form the basis of evolutionary systems.

- Variation operators (recombination and mutation) create the necessary diversity and thereby facilitate novelty, while
- selection acts as a force pushing quality.

The combined application of variation and selection generally leads to improving fitness values in consecutive populations. It is easy (although somewhat misleading) to see such a process as if the evolution is optimising, or at least "approximising", by approaching optimal values closer and closer over its course. Alternatively, evolution it is often 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, reflected in a higher number of offspring. The evolutionary process makes the population adapt to the environment better and better.

Let us note that many components of such an evolutionary process are stochastic. During selection fitter individuals have a higher chance to be selected than less fit ones, but typically even the weak individuals have a chance to become a parent or to survive. For recombination of individuals the choice of which pieces will be recombined is random. Similarly for mutation, the pieces that will be mutated within a candidate solution, and the new pieces replacing them, are chosen randomly. The general scheme of an Evolutionary Algorithm can is given in Figure 2.1 in a pseudo-code fashion; Figure 2.2 shows a diagram.

```
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. 2.1. The general scheme of an Evolutionary Algorithm in pseudo-code

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

- EAs are population based, i.e., they process a whole collection of candidate solutions simultaneously,
- EAs mostly use recombination to mix information of more candidate solutions into a new one,
- EAs are stochastic.

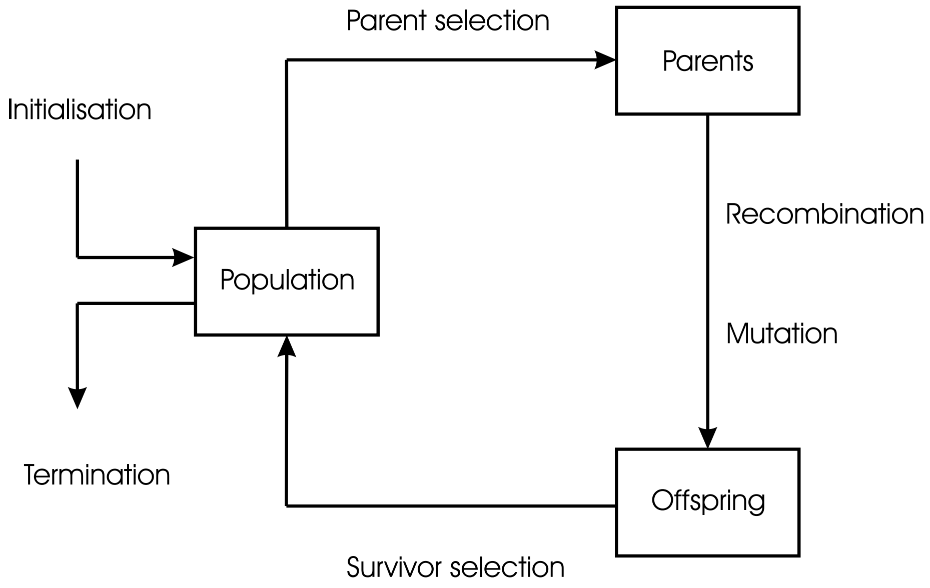

Fig. 2.2. The general scheme of an Evolutionary Algorithm as a flow-chart

The various dialects of evolutionary computing that we have mentioned previously all follow the above general outlines, and differ only in technical details, as is shown in the overview table in Chapter 15. For instance, the representation of a candidate solution is often used to characterise different streams. Typically, the candidates are represented by (i.e., the data structure encoding a solution has the form of) strings over a finite alphabet in Genetic Algorithms (GA), real-valued vectors in Evolution Strategies (ES), finite state machines in classical Evolutionary Programming (EP) and trees in Genetic Programming (GP). These differences have a mainly historical origin. Technically, a given representation might be preferable over others if it matches the given problem better, that is, it makes the encoding of candidate solutions easier or more natural. For instance, for solving a satisfiability problem the straightforward choice is to use bit-strings of length n, where n is the number of logical variables, hence the appropriate EA would be a Genetic Algorithm. For evolving a computer program that can play checkers trees are well-suited (namely, the parse trees of the syntactic expressions forming the programs), thus a GP approach is likely. It is important to note that 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. As opposed to variation operators, selection takes only the fitness information into account, hence it works independently from the actual representation. Differences in the commonly applied selection mechanisms in each stream are therefore rather a tradition than a technical necessity.

2.3 Components of Evolutionary Algorithms

In this section we discuss Evolutionary Algorithms in detail. EAs have 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 Figure 2.1, are:

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

Each of these components must be specified in order to define a particular EA. Furthermore, to obtain a running algorithm the initialisation procedure and a termination condition must be also defined.

2.3.1 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 problem solving space where evolution will take place. Objects forming possible solutions within the original problem context are referred to as **phenotypes**, their encoding, the individuals within the EA, are called **genotypes**. The 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 these phenotypes. For instance, given an optimisation problem on integers, the given set of integers would form the set of phenotypes. Then one could decide to represent them by their binary code, hence 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. To this end, it should hold that the (optimal) solution to the problem at hand – a phenotype – is represented in the given genotype space.

The common EC terminology uses many synonyms for naming the elements of these two spaces. On the side of the original problem context, **candidate solution**, phenotype, and **individual** are used to denote points of the space of possible solutions. This space itself is commonly called the **phenotype space**. On the side of the EA, genotype, **chromosome**, and again individual can be used for points in the space where the evolutionary search will actually take place. This space is often termed the **genotype space**. Also for the elements of individuals there are many synonymous terms. A place-holder is commonly called a variable, a **locus** (plural: loci), a position, or – in a biology oriented terminology – a **gene**. An object on 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 required that the representation be invertible: to each genotype there has to be 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 behind speaking about mutation operators for binary representation, for instance.

2.3.2 Evaluation Function (Fitness Function)

The role of the **evaluation function** is to represent the requirements to adapt to. It forms the basis for selection, and thereby it facilitates improvements. More accurately, it defines what improvement means. From the problem solving perspective, it represents the task to solve in the evolutionary context. Technically, it is a function or procedure that assigns a quality measure to genotypes. Typically, this function is composed from a quality measure in the phenotype space and the inverse representation. To remain with the above example, if we were to maximise x^2 on integers, the fitness of the genotype 10010 could be defined as the square of its corresponding phenotype: $18^2 = 324$.

The evaluation function is commonly called the **fitness function** in EC. This might cause a counterintuitive terminology if the original problem requires minimisation for 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 Section 12.2.1). 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.

2.3.3 Population

The role of the **population** is to hold (the representation of) possible solutions. A population is a multiset[1] of genotypes. The population forms the unit of evolution. Individuals are static objects not changing or adapting, it is the population that does. Given a representation, defining a population can 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, with a distance measure or a neighbourhood relation. In such cases the additional structure has to be defined as well to fully specify a population. As opposed to variation operators that act on the one or two parent individuals, the selection operators (parent selection and survivor selection) work at population level. In general, they take the whole current population into account and choices are always made relative to what we have. For instance, the best individual *of the given population* is chosen to seed the next generation, or the worst individual *of the given population* is chosen to be replaced by a new one. In almost all EA applications the population size is constant, not changing during the evolutionary search.

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 only one fitness value does not necessarily imply only one phenotype is present, and in turn only one phenotype does not necessarily imply only one genotype. The reverse is however not true: one genotype implies only one phenotype and fitness value.

2.3.4 Parent Selection Mechanism

The role of **parent selection** or **mating selection** is to distinguish among individuals based on their quality, 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,

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

high quality individuals get a higher chance to become 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 get stuck in a local optimum.

2.3.5 Variation Operators

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**[2].

Mutation

A unary[3] variation operator is commonly called **mutation**. It is applied to one genotype and delivers a (slightly) modified mutant, the **child** or **offspring** of it. A mutation operator is always stochastic: its output – the child – depends on the outcomes of a series of random choices[4]. It should be noted that an arbitrary unary operator is not necessarily seen as mutation. A problem specific heuristic operator acting on one individual could be termed as mutation for being unary. 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. The role of mutation in EC is different in various EC-dialects, for instance in Genetic Programming it is often not used at all, in Genetic Algorithms it has traditionally been seen as a background operator to fill the gene pool with "fresh blood", while in Evolutionary Programming it is the one and only variation operator doing the whole search work.

It is worth noting that variation operators form the evolutionary implementation of the elementary steps within the search space. Generating a child amounts to stepping to a new point in this space. From this perspective, mutation has a theoretical role too: it can guarantee that the space is connected. This is important since theorems stating that an EA will (given sufficient time) discover the global optimum of a given problem often rely on the property that each genotype representing a possible solution can be reached by the variation operators [118]. The simplest way to satisfy this condition is to allow the mutation operator to "jump" everywhere, for example, by allowing that any allele can be mutated into any other allele with a non-zero probability. However it should also be noted that many researchers feel these proofs

[2] The arity of an operator is the number of objects that it takes as inputs
[3] An operator is **unary** if it applies to one object as input.
[4] Usually these will consist of using a pseudo-random number generator to generate a series of values from some given probability distribution. We will refer to these as "random drawings"

have limited practical importance, and many implementations of EAs do not in fact possess this property.

Recombination

A binary variation operator[5] is called **recombination** or **crossover**. As the names indicate such an operator merges information from two parent genotypes into one or two offspring genotypes. Similarly to mutation, recombination is a stochastic operator: the choice of what parts of each parent are combined, and the way these parts are combined, depend on random drawings. Again, the role of recombination is different in EC dialects: in Genetic Programming it is often the only variation operator, in Genetic Algorithms it is seen as the main search operator, and 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 [115].

The principal behind recombination is simple – that by mating two individuals with different but desirable features, we can produce an offspring which combines both of those features. This principal has a strong supporting case – it is one which has been successfully applied for millennia by breeders of plants and livestock, to produce species which give higher yields or have other desirable features. Evolutionary Algorithms create a number of offspring by random recombination, accept that some will have undesirable combinations of traits, most may be no better or worse than their parents, and hope that some have improved characteristics. Although the biology of the planet earth, (where with a *very* few exceptions lower organisms reproduce asexually, and higher organisms reproduce sexually see e.g., [268, 269]), suggests that recombination is the superior form of reproduction, recombination operators in EAs are usually applied probabilistically, that is, with an existing chance of not being performed.

It is important to note that variation operators are representation dependent. That is, for different representations different variation operators have to be defined. For example, if genotypes are bit-strings, then inverting a 0 to a 1 (1 to a 0) can be used as a mutation operator. However, if we represent possible solutions by tree-like structures another mutation operator is required.

2.3.6 Survivor Selection Mechanism (Replacement)

The role of **survivor selection** or **environmental selection** is to distinguish among individuals based on their quality. In that it is similar to parent

[5] An operator is **binary** if it applies to two objects as input.

selection, but it is used in a different stage of the evolutionary cycle. The survivor selection mechanism is called after having having created the offspring of the selected parents. As mentioned in section 2.3.3, in EC the population size is (almost always) constant, thus a choice has to to be made on which individuals will be allowed in the next generation. This decision is usually based on their fitness values, favouring those with higher quality, although the concept of age is also frequently used. As opposed to parent selection which is typically stochastic, survivor selection is often deterministic, for instance ranking the unified multiset of parents and offspring and selecting the top segment (fitness biased), or selecting only from the offspring (age-biased).

Survivor selection is also often called **replacement** or replacement strategy. In many cases the two terms can be used interchangeably. The choice between the two is thus often arbitrary. A good reason to use the name survivor selection is to keep terminology consistent: step 1 and step 5 in Figure 2.1 are both named selection, distinguished by an adjective. A preference for using replacement can be motivated by the skewed proportion of the number of individuals in the population and the number of newly created children. In particular, if the number of children is very small with respect to the population size, e.g., 2 children and a population of 100. In this case, the survivor selection step is as simple as to chose the two old individuals that are to be deleted to make place for the new ones. In other words, it is more efficient to declare that everybody survives unless deleted, and to choose whom to replace. If the proportion is not skewed like this, e.g., 500 children made from a population of 100, then this is not an option, so using the term survivor selection is appropriate. 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.

2.3.7 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 aiming at an initial population with higher fitness. Whether this is worth the extra computational effort or not is very much depending on the application at hand. There are, however, some general observations concerning this issue based on the so-called anytime behaviour of EAs. These will be discussed later on in Section 2.5 and we will also return to this issue in Chapter 10.

2.3.8 Termination Condition

As for a suitable **termination condition** we can distinguish two cases. If the problem has a known optimal fitness level, probably coming from a known optimum of the given objective function, then reaching this level (perhaps only with a given precision $\epsilon > 0$) should be used as stopping condition. However, EAs are stochastic and mostly there are no guarantees to reach an optimum, hence this condition might never get satisfied and the algorithm may never stop. This requires that this condition is extended with one that certainly stops the algorithm. Commonly used options for this purpose are the following:

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

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, simply a condition from the above list or a similar one that is guaranteed to stop the algorithm. Later on in Section 2.5 we will return to the issue of when to terminate an EA.

In the coming chapters we will describe various types of Evolutionary Algorithms by specifying how the EA components are implemented in the given type. That is we will give a treatment of the representation, variation, and selection operators, specific for that EA variant and give a one glance overview of the typical representatives in an EA tableau. However, we will not discuss the initialisation procedure and a termination condition, for they are usually not "dialect" specific, but implemented along the general considerations outlined above.

2.4 Example Applications

2.4.1 The 8-Queens Problem

In the 8-queens problem we are given a regular chess board (8 by 8) and eight queens that must be placed on the board in such a way that no two queens can check each other. This problem can be naturally generalised, yielding the N-queens problem. Many classical AI approaches to this problem work in a constructive, or incremental, fashion: one starts with placing one queen and after having placed n queens, one attempts to place the $(n+1)$th on a feasible position, i.e., a 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 where all eight queens are placed. The phenotype space

P is the set of all such configurations. Clearly, most elements of this space are infeasible, violating the condition of non-checking 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. By 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 possibilities to specify what kind of inverse we wish to use here. For instance, $1/q(p)$ is an option, but it has the disadvantage that division by zero can deliver a problem. We could circumvent this by adding that when this occurs we have a solution, or by adding a small value ϵ i.e., $1/(q(p) + \epsilon)$. Another option is to use $-q(p)$ or $M - q(p)$, where M is a sufficiently large number to make all fitness values positive, e.g., $M = max\{ \; q(p) \mid p \in P \; \}$. 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 elements of P represented as matrices directly as genotypes, meaning that we design variation operators acting such 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 n-th column contains exactly one queen placed on the i_n-th 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 chromosome 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 against 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, 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 those phenotypes from $P \setminus F(G)$ can never be solutions.

The next step is to define suitable variation operators (mutation and crossover), fitting our representation, i.e., working on genotypes being permutations. The crucial feature of a suitable operator is that it does not lead out of the space G. In common parlance, offspring of a permutation must be permutations as well. Later on in Sections 3.4.4 and 3.5.4 we will treat such operators in much detail. Here we only give one suitable mutation and one crossover operator for illustration purposes. As for mutation we can use an operator that selects two positions in a given chromosome randomly and swaps the values standing on those positions. A good crossover for permutations is less obvious, but the mechanism outlined in Figure 2.3 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 in two segments after 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 are already contained in it
5. do the same for parent 1 and child 2

Fig. 2.3. "Cut-and-crossfill" crossover

The important thing about these variation operators is that mutation will cause 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 could find a solution quickly, while using mutation B can result in an algorithm never finding 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 of setting up an EA is deciding about selection and the population update mechanism. As for managing the population we choose for a simple scheme. In each evolutionary cycle we select two parents delivering 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 Figure 2.1) will be done by choosing 5 individuals randomly from the population and taking the best two as parents that undergo crossover. This ensures a bias towards using parents with relatively high fitness. Survivor selection (step 5 in Figure 2.1) 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 Section 2.3.6 we are to 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 terminate the search if we find a solution or 10.000 fitness evaluations have elapsed. We can furthermore decide to use a population size of 100, and using the variation operators with a certain frequency. For instance we always apply crossover to the two selected parents and in 80% of the cases applying mutation to the offspring. Putting this all together we obtain an EA as summarised in Table 2.1.

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 evaluation

Table 2.1. Tableau describing the EA for the 8-queens problem

2.4.2 The Knapsack Problem

The “0–1 Knapsack” problem, a generalisation of many industrial problems, can be briefly described as follows: Given a set of n of items, each of which has some value v_i attached to it and some cost c_i, how do we select a subset of those items that maximises the value whilst keep the summed cost within some capacity C_{max}? Thus for example when packing a back-pack for a “round the world” trip, we must balance likely utility of the items we wish to take 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 that increases exponentially with the number of items considered. By 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 which 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 will return to in Chapter 12, 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 if including the item would break our capacity constraint, i.e., 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 value of all bits to the right of the current position is 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 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 we will select a random value with uniform probability between 0 and 1, and if it is below 0.7 then we will create two offspring by crossover, otherwise we will make 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 our initial population, and 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 twenty five 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 2.2:

Representation	binary strings of length n
Recombination	One point crossover
Recombination probability	70%
Mutation	each value inverted with independent probability p_m per position
Mutation probability p_m	$1/n$
Parent selection	best out of random two
Survival selection	generational
Population size	500
Number of offspring	500
Initialisation	random
Termination condition	no improvement in last 25 generations

Table 2.2. Tableau describing the EA for the Knapsack Problem

2.5 Working of an Evolutionary Algorithm

Evolutionary Algorithms have some rather general properties concerning their working. To illuminate how an EA typically works we assume a one dimensional objective function to be maximised. Figure 2.4 shows three stages of the evolutionary search, exhibiting how the individuals are distributed in the beginning, somewhere halfway and at the end of the evolution. In the first phase, directly after initialisation, the individuals are randomly spread over the whole search space, see Figure 2.4, left. Already after a few generations this distribution changes: caused by selection and variation operators the population abandons low fitness regions and starts to "climb" the hills as shown in Figure 2.4, 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, where some of these peaks can be sub-optimal. In principle it is possible that the population "climbs the wrong hill" and all individuals are positioned around a local, but not global optimum. Although there is no universally accepted definition of what the terms mean, these distinct phases of search are often categorised in terms of **exploration** (the generation of new individuals in as-yet untested regions of the search space), and **exploitation** (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, with to much of the former leading to inefficient search, and too much of the latter leading to a propensity to focus the search too quickly (see e.g., [131] 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; techniques to prevent it will be discussed in Chapter 9.

The other effect we want to illustrate is the **anytime behaviour** of EAs. We show this by plotting the development of the population's best fitness (objective function) value in time, see Figure 2.5. This curve is characteristic

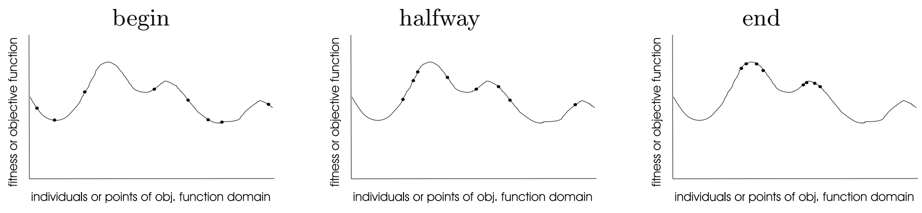

Fig. 2.4. Typical progress of an EA illustrated in terms of population distribution.

for Evolutionary Algorithms, showing rapid progress in the beginning and flattening out later on. This is typical for many algorithms that work by iterative improvements on the initial solution(s). The name "any time" comes from the property that the search can be stopped at any time, the algorithm will have some solution, be it suboptimal.

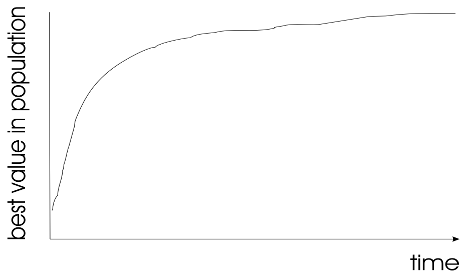

Fig. 2.5. Typical progress of an EA illustrated in terms of development of the best fitness (objective function to be maximised) value within population in time.

Based on this anytime curve we can make some general observations concerning initialisation and the termination condition for Evolutionary Algorithms. As for initialisation, recall the question from Section 2.3.7 whether it is worth to put extra computational efforts into applying some intelligent heuristics to seed the initial populations 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 Figure 2.6. As the figure indicates, using heuristic initialisation can start the evolutionary search with a better population. However, typically a few (in the Figure: k) generations are enough to reach this level, making the worth of extra effort questionable. Later on in Chapter 10 we will return to this issue.

The anytime behaviour also has some general indications regarding termination conditions of EAs. In Figure 2.7 we divide the run into two equally long sections, the first and the second half. As the figure indicates, the progress in terms of fitness increase in the first half of the run, X, is significantly

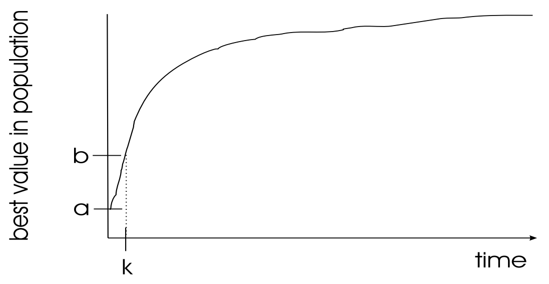

Fig. 2.6. Illustrating why heuristic initialisation might not be worth. Level a shows the best fitness in a randomly initialised population, level b belongs to heuristic initialisation.

greater than the achievements in the second half, Y. This provides a general suggestion that it might not be worth to allow very long runs: due to the anytime behaviour on EAs, efforts spent after a certain time (number of fitness evaluations) may not result in better solution quality.

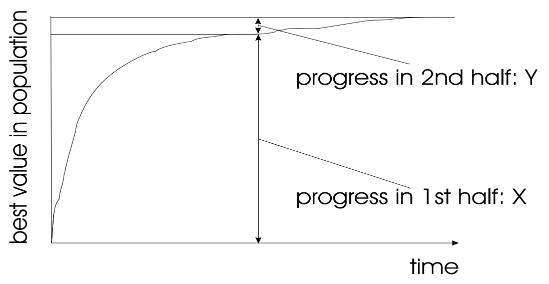

Fig. 2.7. Illustrating why long runs might not be worth. X shows the progress in terms of fitness increase in the first half of the run, Y belongs to the second half.

We close this review of EA behaviour with looking at EA performance from a global perspective. That is, rather than observing one run of the algorithm, we consider the performance of EAs on a wide range of problems. Figure 2.8 shows the 80's view after Goldberg [179]. What the figure indicates is that robust problem solvers –as EAs are claimed to be– show a roughly even 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 clearly outperform random search. A problem tailored algorithm, however, performs much better than an EA, but only on that type of problem where it was designed for. As we move away from this problem type to different problems, the problem specific algorithm quickly looses performance. In this sense, EAs and problem specific algorithms form two antagonistic extremes. This perception has played an important role in positioning Evolutionary

Algorithms and stressing the difference between evolutionary and random search, but it gradually changed in the 90's based on new insights from practice as well as from theory. The contemporary view acknowledges the possibility to combine the two extremes into a hybrid algorithm. This issue will be treated in detail in Chapter 10, where we also present the revised version of Figure 2.8. 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 [434]. That is, showing the EA line always above that of random search is fundamentally incorrect. This will be discussed further in Chapter 11.

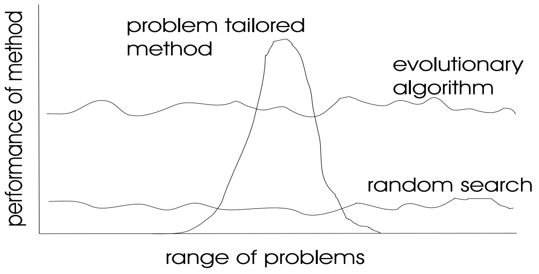

Fig. 2.8. 1980's view on EA performance after Goldberg [179].

2.6 Evolutionary Computing and Global Optimisation

In Chapter 1 we mentioned that there has been a steady increase in the complexity and size of problems that are desired to be solved by computing methods. We also noted that Evolutionary Algorithms are often used for problem optimisation. Of course EAs are not the only optimisation technique known, and 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 exist: an exhaustive enumeration of all of the possible solutions to our problem is clearly such an algorithm. 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 (which states that the available computing power doubles every one and a half year), it is a sad fact of life that 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 exponentially with the number of variables considered. For example many problems in transportation can be reduced to the well known "Travelling Sales Person" problem, i.e., given a list of destinations, to 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 given by $n!/2 = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3$, which is exponential in n. Whilst exact methods whose time complexity scales linearly (or at least polynomially) with the number of variables, exist for some of these problems (see e.g., [197] for an overview), it is widely accepted that for many types of problems often encountered, no such algorithms exist. 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.

We will use the term **Global Optimisation** to refer to the process of attempting to find the solution x^* out of a set of possible solutions S which has the optimal value for some fitness function f. In other words, if we are trying to find the solution x^* such that $x \neq x^* \Rightarrow f(x^*) \geq f(x)$ (here we have assumed a maximisation problem, the inequality is simply reversed for minimisation).

As noted above, a number of *deterministic* algorithms exist which 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 exist (collectively known as Box Decomposition) which 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 (due to searching in a suboptimal order) the time complexity of the algorithms is still the same as complete enumeration.

After exact methods, we find a class of search methods known as *heuristics* which 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, 231] and (certain variants of) Evolutionary Algorithms, 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 Figure 1.1 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 many 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 chess board would look like if we re-ordered it so that squares which are possible next moves for a knight are adjacent to each other. Note that 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 $N(x')$. Eventually this process will lead to the identification of a **local optimum**: a solution which 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, and have the advantage that they are often quick to identify a good solutions to the problem (which is in fact sometimes all that is required in practical applications). However, the downside is that frequently problems will exhibit numerous local optima, some of which may be significantly worse than the global optimum, and no guarantees can be offered in the quality of solution found.

A number of methods have been proposed to get around this problem by changing the search landscape, either by reordering it through a change of neighbourhood function (e.g., Variable Neighbourhood Search [191]) or by temporally assigning low fitness to already seen good solutions (e.g., Tabu Search [171]). However the theoretical basis behind these algorithms is still very much in gestation.

There are a number of features of Evolutionary Algorithms which distinguish them from Local Search algorithms, relating principally to their use of a population. It is the population which provides the algorithm with a means of defining a non-uniform probability distribution function (p.d.f.) governing the generation of new points from S. This p.d.f. reflects possible interactions between points in the population, arising from the recombination of partial solutions from two (or more) members of the population (parents). This contrasts with the globally uniform distribution of blind random search, or the locally uniform distribution used by many other stochastic algorithms such as simulated annealing and various hill-climbing algorithms.

The ability of Evolutionary Algorithms to maintain a diverse set of points not only provides a means of escaping from one local optimum: it provides a means of coping with large and discontinuous search spaces, and if several copies of a solution can be maintained, 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, as will be seen in later chapters.

2.7 Exercises

1. How big is the phenotype space for the 8-queens problem as discussed in section 2.4.1?
2. Try to design an Evolutionary Algorithm for the 8-queens problem that is incremental. That is, a solution must represent a way to place the queens on the chess board one by one. How big is the search space in your design?
3. Find a problem where EAs would certainly perform very poorly compared to to alternative approaches. Explain why do you expect this would be the case.

2.8 Recommended Reading for this Chapter

1. T. Bäck. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, New York, 1996.
   A book giving a formal treatment of evolutionary programming, evolution strategies, and genetic algorithms (no genetic programming) from a perspective of optimisation.
2. Th. Bäck and H.-P. Schwefel. An overview of Evolutionary Algorithms for parameter optimisation. *Evolutionary Computation*, 1(1):1–23, 1993.
   A classical paper (with formalistic algorithm descriptions) that "unified" the field.
3. A.E. Eiben. Evolutionary computing: the most powerful problem solver in the universe? *Dutch Mathematical Archive (Nederlands Archief voor Wiskunde)*, 5/3(2):126–131, 2002.
   A gentle introduction to evolutionary computing with details over GAs and ES. To be found at `http://www.cs.vu.nl/~gusz/papers/ec-intro-naw.ps`
4. D.B. Fogel. *Evolutionary Computation.* IEEE Press, 1995.
   A book covering evolutionary programming, evolution strategies, and genetic algorithms (no genetic programming) from a perspective of achieving machine intelligence through evolution.
5. M.S. Hillier and F.S. Hillier. Conventional optimization techniques. Chapter 1, pages 3–25.in R. Sarker, M. Mohammadian, and X. Yao, editors. *Evolutionary Optimization.* Kluwer Academic Publishers, 2002. *Gives a nice overview of Operations Research techniques for optimisation, including linear-, nonlinear-, goal-, and integer programming.*
6. X. Yao. Evolutionary computation: A gentle introduction. Chapter 2, pages 27–53 in R. Sarker, M. Mohammadian, and X. Yao, editors. *Evolutionary Optimization.* Kluwer Academic Publishers, 2002. *Indeed a smooth introduction presenting all dialects and explicitly discussing EAs in relation to generate-and-test methods.*