Chapter 2
Evolutionary Algorithms

Ágoston E. Eiben and James E. Smith

2.1 Motivation and Brief History

Developing automated problem solvers (that is, algorithms) is one of the central themes of mathematics and computer science. Similarly to engineering, where looking at Nature's solutions has always been a source of inspiration, copying 'natural problem solvers' is a stream within these disciplines. When looking for the most powerful problem solver of the universe, two candidates are rather straightforward:

- the human brain, and
- the evolutionary process that created the human brain.

Trying to design problem solvers based on these answers leads to the fields of neurocomputing and evolutionary computing respectively. The fundamental metaphor of evolutionary computing (EC) relates natural evolution to problem solving in a trial-and-error (a.k.a. generate-and-test) fashion.

In natural evolution, a given environment is filled with a population of individuals that strive for survival and reproduction. Their fitness – determined by the environment – tells how well they succeed in achieving these goals, i.e., it represents their chances to live and multiply. In the context of a stochastic generate-and-test style problem solving process we have a collection of candidate solutions. Their quality – determined by the given problem – determines the chance that they will be kept and used as seeds for constructing further candidate solutions.

Surprisingly enough, this idea of applying Darwinian principles to automated problem solving dates back to the forties, long before the breakthrough of computers [270]. As early as in 1948 Turing proposed "genetical or evolutionary search"

Ágoston E. Eiben
Free University, Amsterdam, The Netherlands
e-mail: gusz@cs.vu.nl

James E. Smith
UWE, Bristol, UK
e-mail: James.Smith@uwe.ac.uk

Table 2.1. The basic evolutionary computing metaphor linking natural evolution to problem solving

EVOLUTION		PROBLEM SOLVING
environment	$\longleftrightarrow$	problem
individual	$\longleftrightarrow$	candidate solution
fitness	$\longleftrightarrow$	quality

and already in 1962 Bremermann actually executed computer experiments on "optimization through evolution and recombination". During the sixties three different implementations of the basic idea have been developed at three different places. In the USA Fogel introduced evolutionary programming, [269, 271], while Holland called his method a genetic algorithm [325, 389, 645]. In Germany Rechenberg and Schwefel invented evolution strategies [761, 801]. For about 15 years these areas developed separately; it is since the early nineties that they are envisioned as different representatives ("dialects") of one technology that was termed evolutionary computing [32, 36, 37, 235, 596]. It was also in the early nineties that a fourth stream following the general ideas has emerged: Koza's genetic programming [41, 483]. The contemporary terminology denotes the whole field by evolutionary computing, or evolutionary algorithms, and considers evolutionary programming, evolution strategies, genetic algorithms, and genetic programming as sub-areas.

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 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, commonly called individuals. 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 individuals 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 individuals (the so-called parents) producing one or more new candidates (the children). Mutation is applied to one individual and results in one new individual. Therefore executing the operations of recombination and mutation on the parents leads to the creation of a set of new individuals (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 an individuals with sufficient quality (a solution) is found or a previously set computational limit is reached.

There are two fundamental 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. As opposed to variation operators, selection reduces diversity.

The combined application of variation and selection generally leads to improving fitness values in consecutive populations. It is easy (although somewhat misleading) 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. Thus, although during selection fitter individuals have a higher chance of being selected than less fit ones, typically even the weak individuals have a 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 Fig. 1 in a pseudocode fashion.

Algorithm 1. The general scheme of an evolutionary algorithm in pseudocode

```
1 INITIALISE population with random individuals;
2 EVALUATE each individual;
3 repeat
4     SELECT parents;
5     RECOMBINE pairs of parents;
6     MUTATE the resulting offspring;
7     EVALUATE new individuals;
8     SELECT individuals for the next generation;
9 until TERMINATION CONDITION is satisfied ;
```

It is easy to see that this scheme falls into 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 selection operators. Evolutionary algorithms possess a number of features that can help to 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.

- EAs mostly use recombination, mixing information from two or more candidate solutions to create a new one.
- EAs are stochastic.

The various dialects of evolutionary computing that we have mentioned previously all follow these general outlines, differing only in technical details as shown in the overview table (2.2) later on in this chapter. In particular, the representation of a candidate solution is often used to characterise different streams. Typically the representation (i.e., the data structure encoding an individual) 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, hence the appropriate EA would be a genetic algorithm. To evolve 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. 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.

2.3 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. 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, run-able, algorithm, it is necessary to specify each of these components and to define the initialisation procedure and a termination condition.

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 takes place. 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.

Within the Evolutionary Computation literature many synonyms can be found for naming the elements of these two spaces.

- On the side of the original problem context the terms **candidate solution**, phenotype, and **individual** are all used to denote points in the space of possible solutions. This space itself 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 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.

2.3.2 Evaluation Function (Fitness Function)

The role of the **evaluation function** is to represent the requirements the population should adapt to. 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 a quality measure in the phenotype space and the inverse representation. 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 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, 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. 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 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. Alternatively, in some sophisticated EAs a population has an additional spatial structure, defined via a distance measure or a neighbourhood relation. This may be thought of as akin to the way that "real" populations evolve within the context of a spatial structure dictated by the individuals' geographical position on earth. In such cases the additional structure must also be defined in order to fully specify a population. In contrast to variation operators, that act on one or more parent individuals, 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 the given population* is chosen to be replaced by a new one. In almost all EA applications the population size is constant and does not change 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

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

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. The converse is, however, not true: if only one genotype is present then this implies only one phenotype and fitness value are.

2.3.4 Parent Selection Mechanism

The role of **parent selection** or **mating 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 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. In principle, there is no restriction on how such variation operators work. The variation operators in the traditional EA dialects are usually divided into two types based on their **arity**, distinguishing unary and n-ary ($n > 1$) operators. Such a division can also be made for the newest members of the EA family, such as differential evolution [733] or particle swarm optimisation methods [457].

2.3.5.1 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 an arbitrary unary operator is not necessarily seen as mutation. For example, it might be tempting to use the term mutation to describe a problem-specific heuristic operator which acts on one individual[2]. 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 has historically been different in various EC dialects. Thus, in genetic programming for instance, it is often not used at all, whereas in genetic algorithms it has traditionally been seen as a background

[2] Such operators are used frequently in memetic algorithms.

operator, used to fill the gene pool with "fresh blood", and in evolutionary programming it is the sole variation operator responsible for the whole search work.

It is worth noting that variation operators form the evolutionary implementation of 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 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 [236]. 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 with a nonzero probability. However, it should also be noted that many researchers feel these proofs have limited practical importance, and many implementations of EAs do not in fact possess this property.

2.3.5.2 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 [234].

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 a *very* few exceptions lower organisms reproduce asexually, and higher organisms always reproduce sexually [569, 570], suggests that recombination is the superior form of reproduction. However recombination operators in EAs are usually applied probabilistically, that is, with a non-zero 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 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 selection, but 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. 2.3.3, in EC the population size is almost always constant, which means that a choice has 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 **replacement** or the replacement strategy. In many cases the two terms can be used interchangeably, and so the choice of which to use is often arbitrary. A good reason to use the name survivor selection is to keep terminology consistent: steps 1 and 5 in Fig. 1 are both named selection, distinguished by an adjective. A preference for using replacement can be motivated if there is a large difference between 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 choosing the two old individuals that are to be deleted to make places 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.

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, 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 issue based on the so-called anytime behaviour of EAs. These are discussed in Sect. 2.4.

2.3.8 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 , albeit perhaps only within a given precision $\varepsilon > 0$. 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.

2.4 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. Fig. 2.1 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 phase, directly after initialisation, the individuals are randomly spread over the whole search space (Fig. 2.1, 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. 2.1, 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 definition of what the terms mean, these distinct phases of the search process 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. 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. **Premature convergence** is the well-known effect of losing population diversity too quickly, and

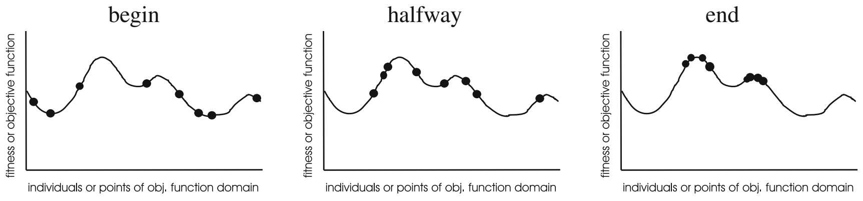

Fig. 2.1. Typical progress of an EA illustrated in terms of population distribution

getting trapped in a local optimum. This danger is generally present in evolutionary algorithms.

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 over time (Fig. 2.2). This curve is characteristic 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 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.

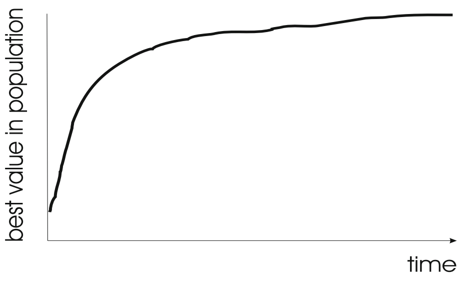

Fig. 2.2. Typical progress of an EA illustrated in terms of development over time of the highest fitness in the population

Based on this anytime curve we can make some general observations concerning initialisation and the termination condition for EAs. In Section 2.3.7 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 the typical progress curve of an evolutionary process makes it unnecessary. This is illustrated in Fig. 2.3. As the figure indicates, using 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 worth of extra effort questionable in general.

The anytime behaviour also gives some general indications regarding the choice of termination conditions for EAs. In Fig. 2.4 we divide the run into two equally long sections. As the figure indicates, the progress in terms of fitness increase in

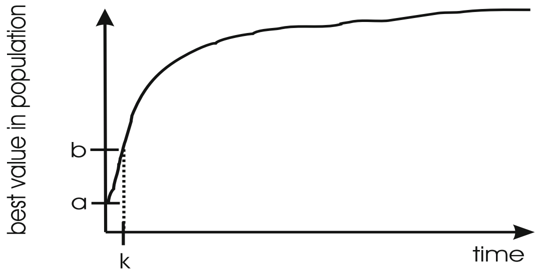

Fig. 2.3. Illustration of why heuristic initialisation might not be worth additional effort. Level a shows the best fitness in a randomly initialised population, level b belongs to heuristic initialisation

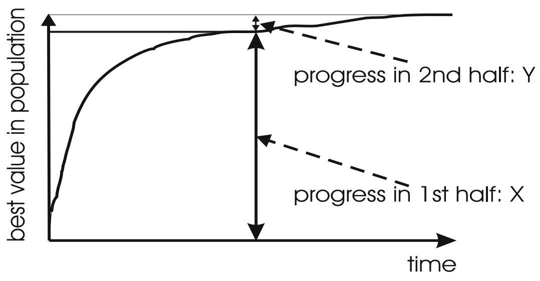

Fig. 2.4. Illustration of why long runs might not be worth performing. X shows the progress in terms of fitness increase in the first half of the run, while Y belongs to the second half

the first half of the run (X) is significantly greater than in the second half (Y). This provides a general suggestion 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) are 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. 2.5 shows the 1980s view after Goldberg [325]. What the figure indicates is that robust problem solvers –as EAs are claimed to be– 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 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 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 practise as well as from

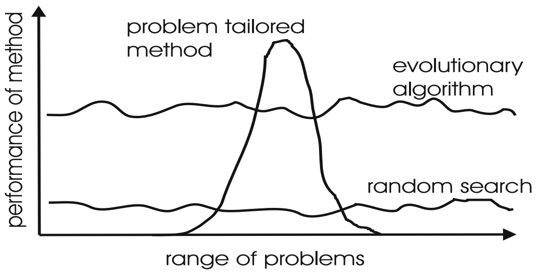

Fig. 2.5. 1980s view of EA performance after Goldberg [325]

theory. The contemporary view acknowledges the possibility of combining the two extremes into a hybrid algorithm. This insight is the main premise behind memetic algorithms that form the subject matter of the present book.

2.5 Evolutionary Algorithm Variants

Throughout this chapter we present evolutionary computing as *one* problem-solving paradigm, mentioning four historical types of EAs as "dialects". These dialects have emerged independently to some extent (except GP that grew out of GAs) and developed their own terminology, research focus, and technical solutions to realise particular evolutionary algorithm features. The differences between them, however, are not crisp – there are many examples of EAs that are hard to place into one of the historical categories. It is one of our main messages that such a division is not highly relevant, even though it may be helpful in some cases. Existing literature however, often uses the names of these dialects to position a particular method and we feel that a good introduction should also include some information about them. To this end, we provide a simple summary in Table 2.2.

It is worth to note that the borders between the four main EC streams have diminished over the last decade. Approaching EAs from a "unionist" perspective it is better not to distinguish different EAs by the traditional stream they belong to, but by their main algorithmic components: representation, recombination operator, mutation operator, parent selection operator, and survivor selection operator. Reviewing the details of the commonly used operators and related parameters exceeds the scope of this chapter. Hence, we are forced to use (the names of) them without further explanation here and refer to a modern text book, such as [239] or [193], for those details. Table 2.3 provides an illustration showing how particular choices can lead to a typical genetic algorithm or evolution strategy, thus linking the two perspectives.

Considering Table 2.3, one may notice that it does not provide all details needed for a complete specification of an evolutionary algorithm. For instance, the population size is not specified. This observation raises the issue of algorithm parameters and, one step further, the issue of algorithm design.

Table 2.2. Overview of the main EA dialects

Component or feature	EA Dialect			
	GA	ES	EP	GP
Typical problems	Combinatorial optimisation	Continuous optimisation	Optimisation	Modelling
Typical representation	Strings over a finite alphabet	Vectors of real numbers	Appl. specific often as in ES	Trees
Role of recombination	Primary variation operator	Important, but secondary	Never applied	Primary/only variation operator
Role of mutation	Secondary variation operator	Important, sometimes the only operator	The only variation operator	Secondary, sometimes not used at all
Parent selection	Random, biased by fitness	Random, uniform	Each individual creates one child	Random, biased by fitness
Survivor selection	Random, biased by fitness	Deterministic, biased by fitness	Random, biased by fitness	Random, biased by fitness

Table 2.3. A typical GA and ES as an instantiation of the generic EA scheme

	GA	ES
Representation	bit-strings	real-valued vectors
Recombination	1-point crossover	intermediary
Mutation	bit-flip	Gaussian noise by $N(0,\sigma)$
Parent selection	2-tournament	uniform random
Survivor selection	generational	(μ,λ)
Extra	none	self-adaptation of σ

In the broad sense, algorithm design includes all decisions needed to specify an algorithm for solving a given (type of) problem. A decision to use evolutionary algorithms implies a general algorithmic framework – the one described in the beginning of this chapter. Using such an algorithmic framework implies that the algorithm designer adopts many design decisions (that led to the framework) and only needs to specify a "few" details. The principal challenge for algorithm designers is caused by the fact that the design details largely influence the performance of the algorithm. A well designed EA can perform orders of magnitude better than one based on poor choices. Hence, algorithm design in general, and EA design in particular, is an optimization problem itself, where the objective to be optimised is the performance of the EA.

As stated above, designing an EA for solving a given problem requires filling in the details of the generic EA framework appropriately. To denote these details one can use the term *EA parameters*. Using this terminology, designing an EA for a given application amounts to selecting good values for the parameters. For instance, the definition of an EA might include setting the parameter `crossoveroperator` to `onepoint`, the parameter `crossoverrate` to 0.5, and the parameter `populationsize` to 100. In principle, this is a sound naming convention, but intuitively, there is a difference between choosing a good crossover operator from a given list, e.g., {`onepoint`, `uniform`, `averaging`}, and choosing a good value for the related crossover rate $p_c \in [0,1]$. This difference can be formalised if we distinguish parameters by their domains. The parameter `crossoveroperator` has a finite domain with no sensible distance metric or ordering, whereas the domain of the parameter p_c is a subset of $\mathbb{R}$ with the natural structure for real numbers. This difference is essential for searchability of the design space. For parameters with a domain that has a distance metric, or is at least partially ordered, one can use heuristic search and optimization methods to find optimal values. For the first type of parameters this is not possible because the domain has no exploitable structure. The only option in this case is sampling.

The difference between these two types of parameters has already been noted in evolutionary computing, but various authors use various naming conventions. For instance, [47] uses the names *qualitative* and *quantitative* parameters respectively, [951] distinguishes between *symbolic* and *numeric* parameters, while [67] calls them *categorical* and *numerical*. Furthermore, [819] calls unstructured parameters *components* and the elements of their domains operators and in the corresponding terminology a parameter is instantiated by a value, while a component is instantiated by allocating an operator to it. In the context of statistics and data mining one distinguishes two types of variables (rather than parameters) depending on the presence of an ordered structure, but a universal terminology is lacking here too. Commonly used names are *nominal* vs. *ordinal* and *categorical* vs. *ordered* variables. Looking at it from a technical perspective, the very essence of the matter is the presence/absence of a (partial) ordering which is pivotal to searchability. This aspect is best captured through the names *ordered* and *unordered* parameters.

Table 2.4. Possible pairs of terms to distinguish the two types of EA parameters

Type I	Type II
qualitative parameter	quantitative parameter
symbolic parameter	numeric parameter
categorical parameter	numerical parameter
component	parameter
nominal variable	ordinal variable
categorical variable	ordered variable
unordered parameter	ordered parameter

For a clear distinction between these cases we propose to use the terms *qualitative parameter* and *quantitative parameter* and to call the elements of the parameter's domain *parameter values*.[3] In practice, quantitative parameters are mostly numerical values, e.g., the parameter crossover rate uses values from the interval $[0,1]$, and qualitative parameters are often symbolic, e.g., `crossoveroperator`. However, in general, quantitative parameters and numerical parameters are not the same, because it is possible to have an ordering on a set of symbolic values - for example colours may be ordered by how they appear in the rainbow. Note that the terminology we propose here does not refer to the presence/absence of the (partial) ordering. In this respect, ordered vs. unordered could have been be better, but we prefer quantitative and qualitative for non-technical reasons, feeling that their use is more natural.

It is important to note that the number of parameters of EAs is not specified in general. Depending on particular design choices one might obtain different numbers of parameters. For instance, instantiating the qualitative parameter `parentselection` by `tournament` implies a new quantitative parameter `tournamentsize`. However, choosing for `roulettewheel` does not add any parameters. This example also shows that there can be a hierarchy among parameters. Namely, qualitative parameters may have quantitative parameters "under them". If an unambiguous treatment requires we can call such parameters *sub-parameters*, always belonging to a qualitative parameter.

Distinguishing qualitative and quantitative parameters naturally leads to distinguishing two levels in designing a specific EA for a given problem. In the resulting terminology we say that the high-level qualitative parameters define the EA, while the low-level quantitative parameters define a variant of this EA. Table 2.5 illustrates this matter.

Adopting this naming convention we can give a detailed answer to the question that forms the title of this chapter: What are Evolutionary Algorithms? An evolutionary algorithm is a partial instantiation of the generic EA framework where the values to instantiate qualitative parameters are defined, but the quantitative parameters are not. After specifying all details, including the values for all parameters, we obtain *an EA instance*. This terminology enables precise formulations, meanwhile it enforces care with phrasing. Clearly, this distinction between EAs and EA instances is similar to distinguishing problems and problem instances. For example, "TSP" represents the set of all possible problem configurations of the travelling salesman problem, whereas an instance is one specific problem, e.g., the 10 cities TSP with a given distance matrix D and Euclidean metric. If rigorous terminology is required then the right phrasing is "to apply an EA instance to a problem instance".

[3] Parameter values belonging to qualitative parameters, e.g., one-point-crossover, uniform-crossover, or tournament-selection, ranked-biased-selection, are usually called operators. This is fully consistent with our proposal here and can be seen as a matter of an additional naming convention.

Table 2.5. Three EA instances specified by the qualitative parameters: Representation, recombination, mutation, parent selection, survivor selection, and the quantitative parameters : mutation rate (p_m), mutation step size (σ), crossover rate (p_c), population size (μ), offspring size (λ), and tournament size. The EA instances in columns EA_1 and EA_2 are just variants of the same EA. The EA instance in column EA_3 belongs to a different EA.

	EA_1	EA_2	EA_3
Representation	bitstring	bitstring	real-valued
Recombination	1-point	1-point	averaging
Mutation	bit-flip	bit-flip	Gaussian $N(0,\sigma)$
Parent selection	tournament	tournament	uniform random
Survivor selection	generational	generational	(μ,λ)
p_m	0.01	0.1	0.05
σ	n.a.	n.a	0.1
p_c	0.5	0.7	0.7
μ	100	100	10
λ	n.a.	n.a.	70
tournament size	2	4	n.a.

2.6 Designing and Tuning Evolutionary Algorithms

As mentioned above, designing an good EA is in fact an optimisation problem. This problem is far from trivial, because there is very little known in general about the influence of EA parameters on EA performance. Most researchers and practitioners agree that the parameters of EAs interact with each other in a complex, non-linear way and even after 30 years of research there are only vague heuristics for designing a good EA instance for a given problem. In practice, the EA is often chosen intuitively or driven by habits, e.g., one may have a personal preference for GAs, while others' default could be ES. After that, parameter values are mostly selected by conventions (mutation rate should be low), ad hoc choices (why not use uniform crossover), and experimental comparisons on a limited scale (testing combinations of three different crossover rates and three different mutation rates).

Figure 2.6 shows the general scheme of the EA design process attempting to optimise algorithm performance on a given problem.[4] The designer is testing different parameter values, whose utility is determined by the performance of the corresponding EA instance on the given problem instance. Formally, such a design session is a trial-and-error (a.k.a. generate-and-test) procedure, resulting in specific values for the parameters of the EA in question. Given that all parameters of an EA must be specified before it can be applied, finding good parameter values is an absolutely necessary condition for any application, hence an immediate need for all researchers

[4] To be very precise: optimise the performance of an EA instance on a given problem instance.

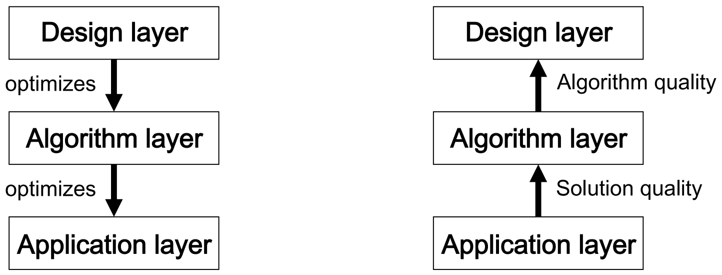

Fig. 2.6. Illustration of 3-tier hierarchy behind EA design showing the control flow (left), and the information flow (right).

and practitioners. In this light, it is odd that the evolutionary computing community has not adopted algorithmic optimisers to solve the EA parameter tuning problem. Ironically, it has been noted long ago that the EA tuning problem falls in the problem class where EAs are claimed to be competitive solvers:

- the given problem has many parameters leading to a large search space,
- the problem has parameters of different types (e.g., reals, integers, symbolic values),
- there are complex non-linear interactions between the parameters leading to a complex non-linear objective function,
- the objective function has many local optima,
- there is noise in the data hindering exact calculations.

This insight has motivated the so-called meta-EAs, whose first representatives, meta-GAs, have been developed already in the late eighties to tune GA parameters [335]. However, meta-GAs or meta-ES [381] have never been used on a large scale. This really suboptimal situation is slowly changing over the last couple of years. The new development takes place along different research lines. First, meta-EAs are being "unearthed", enriched with additional features and tested for their ability to find good EA parameters, see, for instance, [951]. Another line of research concerns generic parameter tuners, developed and used to optimise EA parameters. The *Sequential Parameter Optimization Toolbox* (SPOT), [47, 48, 49] uses an iterative procedure, repeatedly testing parameter vectors and using the results to fit a model to predict the utility of other parameter vectors. Over the course of a run, SPOT simultaneously improves the prediction model and the parameter vectors. The *Relevance Estimation and VAlue Calibration* method (REVAC) implicitly creates probability distributions regarding the parameters (one probability distribution per parameter) in such a way that parameter values that proved to be good in former trials have a higher probability then poor ones. Initially, all distributions represent a uniform random variable and after each new test they are updated based on the new information. After terminating the tuning process, i.e., stopping REVAC, these distributions can be retrieved and analysed, showing not only the range of promising parameter values, but also disclosing information about the relevance of each parameter, [650, 651, 819]. As for the (near) future, it seems safe to predict that the increasing maturity of such parameter tuners will lead to their adoption in the EC community. This, in turn, can increase the performance of EAs on a large scale and deliver novel insights and knowledge about the relationships between EA parameters and EA performance.

2.7 Concluding Remarks

We have described the basic evolutionary paradigm and how it encompasses a wide range of iterative population-based global search methods. Representatives from this class of methods have now been successfully applied to a huge range of different application domains as can be witnessed by the ever increasing volume of papers, conferences and journals. The prime difference between evolutionary and memetic algorithms (MAs) is that, as we have described them, EAs do not consider a step of self-improvement within the cycle - they just work on the outcome of randomised variation. In contrast Memetic Algorithms introduce a stage of individual (rather than population) learning, so that a solution (or its genotype) is (often systematically) perturbed and replaced by the new solution (or possibly its genotype) if that has higher fitness, *independently* of the rest of the population.