Chapter 1
Evolutionary computation

It always is advisable to perceive clearly our ignorance.
Charles Robert Darwin

Evolution is the theory postulating that all the various types of living organisms have their origin in other preexisting types, and that the differences are due to modifications inherited through successive generations. *Evolutionary computation* is the offshoot of computer science focusing on algorithms inspired by the theory of evolution. The definition is deliberately vague since the boundaries of the field are not, and cannot be, defined clearly. Evolutionary computation is a branch of *computational intelligence*, and it is included into the broad framework of *bio-inspired heuristics*. We shall distinguish explicitly between *natural evolution* and *artificial evolution* to avoid confusion whenever necessary.

This chapter sketches the basics of evolutionary computation and introduces its terminology. A comprehensive compendium of the field is out of the scope of this book, and most concepts are defined only to the extent they are required in what follows. Interested readers will find several fascinating books on the topic, such as [6]. Moreover, a survey of evolutionary theories is beyond our knowledge. We can only suggest [5] and [12] as starting points into the vast and fascinating world of biology.

1.1 Natural and artificial evolution

Natural evolution is a cornerstone of modern biology, and scientists show a remarkable consensus on the topic. The original theories of *evolution* and *natural selection* proposed almost concurrently and independently by Charles Robert Darwin [4] and Alfred Russel Wallace [21] in 19th century, combined with *selectionism* by Charles Weismann [23] and genetics by Gregor Mendel [22], are accepted ubiquitously in the scientific community, as well as widespread among the general public. This coherent corpus, often named *Neo-Darwinism*, acts as a *grand unifying theory* for biology: it is able to explain the wonders of life, and, most noticeably, it does it starting from a limited number of relatively simple and intuitively plausible concepts. It describes the whole process of evolution through notions such as *reproduction*, *vari-*

E. Sanchez et al., *Evolutionary Optimization: the μGP toolkit,*
DOI 10.1007/978-0-387-09426-7_1, © Springer Science+Business Media, LLC 2011

ation, *competition*, and *selection*. Reproduction is the process of generating an offspring from parents where the progeny inherit traits of their predecessors. Variation is the unexpected alteration of a trait. Competition and selection are the inevitable results of the strive for survival caused by an environment with limited resources.

Evolution can be easily described as a sequence of steps, some mostly deterministic and some mostly random [15]. Such an idea of random forces shaped by deterministic pressures is inspiring and, not surprisingly, has been exploited to describe phenomena quite unrelated to biology. Notable examples include alternatives conceived during learning [3], ideas striving to survive in our culture [5], or even possible universes [24] [18].

Evolution may be seen as an improving process that perfect raw features. Indeed, this is a mistake that eminent biologists like Richard Dawkins and Stephen Jay Gould warn us not to do. Nevertheless, if evolution is seen as a force pushing toward a goal, another terrible misunderstanding, it must be granted that it worked quite well: in some billion years, it turned unorganized cells into wings, eyes, and other amazingly complex structures without requiring any *a priori* design. The whole neo-Darwinist paradigm may thus be regarded as a powerful optimization tool able to produce great results starting from scratch, not requiring a plan, and exploiting a mix of random and deterministic operators.

Dismissing biologists' complaints, evolutionary computation practitioners loosely mimic the natural process to solve their problems. Since they do not know how their goal could be reached, at least not in details, they exploit some neo-Darwinian principles to cultivate sets of solutions in artificial environments, iteratively modifying them in discrete steps. The problem indirectly defines the environment where solutions strive for survival. The process has a defined goal. The simulated evolution is simplistic, when not even implausible. Notwithstanding, successes are routinely reported in the scientific literature. Solutions in a given step inherit qualifying traits from solutions in the previous ones, and optimal results emerge from the artificial primeval soup.

In evolutionary computation, a single candidate solution is termed *individual*; the set of all candidate solutions that exists at a particular time is called *population*, and each step of the evolution process a *generation*. The ability of an individual to solve the given problem is measured by the *fitness function*, which ranks how likely one solution is to propagate its characteristics to the next generations. Most of the jargon of evolutionary computation mimics the precise terminology of biology. The word *genome* denotes the whole genetic material of the organism, although its actual implementation differs from one approach to another. The *gene* is the functional unit of inheritance, or, operatively, the smallest fragment of the genome that may be modified during the evolution process. Genes are positioned in the genome at specific positions called *loci*, the plural of *locus*. The alternative genes that may occur at a given locus are called *alleles*.

The natural processes that lead to mutations, reproduction, competition and selection are emulated by *operators*. Operators act on genes, single individuals, groups or entire populations, usually producing a modified version of the entity they manipulate.

Biologists need to distinguish between the *genotype* and the *phenotype*: the former is all the genetic constitution of an organism; the latter is the set of observable properties that are produced by the interaction between the genotype and the environment. In many implementations, evolutionary computation practitioners do not require such a precise distinction. The numerical value representing the fitness of an individual is sometimes assimilated to its phenotype.

To generate the offspring for the next generation, most evolutionary algorithms implement sexual and asexual reproduction. The former is usually named *recombination*; it necessitates two or more participants, and implies the possibility for the offspring to inherit different characteristics from different parents. When recombination is achieved through a simple exchange of genetic material between the parents, it often takes the name of *crossover*. The latter is named *replication*, to indicate that a copy of an individual is created, or, more commonly, *mutation*, to stress that the copy is not exact. Almost no evolutionary algorithm takes gender into account; hence, individuals do not have distinct reproductive roles. In some implementations, mutation takes place only after the sexual recombination. Noticeably, some evolutionary algorithms do not store a collection of distinct individuals, and therefore reproduction is performed modifying the statistical parameters that describe the current population. All operators exploited during reproduction can be cumulatively called *evolutionary operators*, or *genetic operators* stressing that they act at the genotypical level.

Mutation and recombination introduce variability in the population. *Parent selection* is also usually a stochastic process, albeit biased by the fitness. The population broadens and contracts rhythmically at each generation. First, it widens when the offspring are generated. Then, it shrinks when individuals are discarded. The deterministic step usually involves deciding which individuals are chosen for survival from one generation to the next. This step may be called *survivor selection*, or just *selection*.

Evolutionary algorithms may be defined local search algorithms since they sample a region of the search space dependent upon their actual state, and the offspring loosely define the concept of neighborhood. Since they are based on the trial and error paradigm, they are heuristic algorithms. They are not usually able to mathematically guarantee an optimal solution in a finite time, whereas interesting mathematical properties have been proven over the years.

If the current boundary of evolutionary computation may seem not clear, its inception is even more vague. The field does not have a single recognizable origin. Some scholars identify its starting point in 1950, when Alan Turing pointed out the similarities between learning and natural evolutions [20]. Others pinpoint the inspiring ideas that appeared in the end of the decade [11] [16] [1], despite the fact that the lack of computational power significantly impaired their diffusion in the broader scientific community. More commonly, the birth of evolutionary computation is set in the 1960s with the appearance of three independent research lines, namely: *genetic algorithms*, *evolutionary programming*, and *evolution strategies*. Despite some minor disagreements, the pivotal importance of these researches is unquestionable.

1.2 The classical paradigms

Genetic algorithm is probably the most popular term in evolutionary computation. It is abbreviated as GA, and it is so popular that in the non-specialized literature it is sometimes used to denote any kind of evolutionary algorithm. The fortune of the paradigm is linked to the name of John Holland and his 1975 book [14], but the methodology was used and described much earlier by several researchers, including many of Holland's own students [9] [10] [2]. Genetic algorithms have been proposed as a step in *classifier systems*, a technique also proposed by Holland. They have been originally exploited more to study the evolution mechanisms itself, rather than solving actual problems. Very simple test benches, as trying to set a number of bits to a specific value, were used to analyze different strategies and schemes.

In a genetic algorithm, the individual, i.e., the evolving entity, is a sequence of bits, and this is probably the only aspect common to all the early implementations. The number of offspring is usually larger than the size of the original population. Various crossover operators have been proposed by different researchers. The parents are chosen using a probability distribution based on their fitness. How much a highly fit individual is favored determines the *selective pressure* of the algorithm. After evaluating all new individuals, the population is reduced back to its original size. Several different schemes have been proposed to determine which individuals survive and which are discarded, but interestingly most schemes are deterministic. When all parents are discarded, regardless their fitness, the approach is called *generational*. Conversely, if parents and offspring compete for survival regardless their age, the approach is *steady-state*. Any mechanism that preserves the best individuals through generations is called *elitist*.

Evolutionary programming, abbreviated as EP, was proposed by Lawrence J. Fogel in a series of works in the beginning of 1960s [7] [8]. Fogel highlighted that an intelligent behavior requires the ability to forecast changes in the environment, and therefore focused his work on the evolution of predictive capabilities. He chose finite state machines as evolving entities, and the predictive capability measured the ability of an individual to anticipate the next symbol in the input sequence provided to it. Later, the technique was successfully applied to diverse combinatorial problems.

Fogel's original algorithm considered a set of P automata. Each individual in such population was tested against the current sequence of input symbols, i.e., its environment. Different payoff functions could be used to translate the predictive capability into a single numeric value called fitness, including a penalty for the complexity of the machine. Individuals were ranked according to their fitness. Then, P new automata were added to the population. Each new automaton was created by modifying one existing automaton. The type and extent of the mutation was random and followed certain probability distributions. Finally, half of the population was retained and half discarded, thus the size of the population remained constant. These steps were iterated until a specific number of generations has passed, at which point the best finite state machine was used to predict the actual next symbol. That symbol was added to the environment and process repeated.

In his basic algorithm, each automaton generated exactly one descendant through a mutation operator, but there was no firm constraint that only one offspring had to be created from each parent. After all the offspring are added to the population, half of the individuals are discarded. Survivals were chosen at random, with a probability influenced by their fitness. Thus, how much a highly fit individual is likely to survive in the next generation represent the selective pressure is evolutionary programming.

The third approach is *evolution strategies*, ES for short, and was proposed by Ingo Rechenberg and Hans-Paul Schwefel in early 1960s [13] [17]. It has been originally developed as an optimization tool to solve a practical optimization problem. In evolution strategies, the individual is a set of parameters, usually encoded as numbers, either discrete or continuous. Mutation simply consists in the simultaneous modification of all parameters, with small alterations being more probable than larger ones. On the other hand, recombination can implement diverse strategies, like copying different parameters from different parents, or averaging them. Remarkably, the very first experiments with evolution strategies used a population of one individual, and dice tossed by hands.

Scholars developed a unique formalism to describe the characteristics of their evolution strategies. The size of the population is commonly denoted with the Greek letter *mu* (μ), and the size of the offspring with the Greek letter *lambda* (λ). When the offspring is added to the current population before choosing which individuals survive in the next generation, the algorithm is denoted as a $(\mu + \lambda)$-ES. In this case, a particularly fit solution may survive through different generations as in steady-state genetic algorithms or evolutionary programming. Conversely, when the offspring replace the current population before choosing which individuals survive in the next generation, the algorithm is denoted as a (μ, λ)-ES. This approach resembles a generational genetic algorithm or evolutionary programming, and the optimum solution may be discarded during the run. For short, the two approaches are called *plus* and *comma* selection, respectively. And in the 2000s, these two terms can be found in the descriptions of completely of different evolutionary algorithms. When comma selection is used, $\mu < \lambda$ must hold. No matter the selection scheme, the size of the offspring is much larger than the size of the population in almost all implementations of evolution strategies.

When recombination is implemented, the number of parents required by the crossover operator is denoted with the Greek letter rho (ρ) and the algorithm written as $(\mu/\rho \stackrel{+}{,} \lambda)$-ES. Indeed, the number of parents is smaller than the number of individuals in the population, i.e., $\rho < \mu$. $(\mu \stackrel{+}{,} 1)$-ES are sometimes called *steady-state evolution strategies*.

Evolution strategies may be nested. That is, instead of generating the offspring using conventional operators, a new evolution strategy may be started. The result of the sub-strategy is used as the offspring of the parent strategy. This scheme can be found referred as *nested evolution strategies*, or *hierarchical evolution strategies*, or *meta evolution strategies*. The inner strategy acts as a tool for local optimizations and commonly has different parameters from the outer one. An algorithm that runs for γ generations a sub-strategy is denoted with $(\mu/\rho \stackrel{+}{,} (\mu/\rho \stackrel{+}{,} \lambda)^{\gamma})$-ES. Where γ is also called isolation time. Usually, there is only one level of recursion, although a

deeper nesting is theoretically possible. Such a recursion is rarely used in evolutionary programming or genetic algorithms, although it has been successfully exploited in peculiar approaches, such as [19].

Since evolution strategies are based on mutations, the search for the optimal amplitude of the perturbations kept researchers busy throughout the years. In real-valued search spaces, the mutation is usually implemented as a random perturbation that follows a normal probability distribution centered on the zero. Small mutations are more probable than larger ones, as desired, and the variance may be used as a knob to tweak the average magnitude. The variance used to mutate parameters, and the parameters themselves may also be evolved concurrently. Furthermore, because even the same problem may call for different amplitudes in different loci, a dedicated variance can be associated to each parameter. This *variance vector* is modified using a fixed scheme, while the *object parameter vector*, i.e., the values that should be optimized, are modified using the variance vector. Both vectors are then evolved concurrently as parts of a single individual. Extending the idea, the optimal magnitudes of mutation may be correlated. To take into account this phenomenon, modern evolution strategies implement a *covariance matrix*.

All evolutionary algorithms show the capacity to adapt to different problems, thus they can sensibly be labeled as *adaptive*. An evolutionary algorithm that also adapts the mechanism of its adaptation, i.e., its internal parameters, is called *self adaptive*. Parameters that are self adapted are sometimes named *endogenous*, borrowing the term describing the hormones synthesized within an organism. Self adaptation mechanisms have been routinely exploited both in the evolution strategies and evolutionary programming paradigms, and sometimes used in genetic algorithms.

Since the 2000s, evolution strategies have been used mainly as a numerical optimization tool for continuous problems. Several implementations, written either in general-purpose programming languages or commercial mathematical toolboxes, like MatLab, are freely available. And they are sometimes exploited by practitioners overlooking their bio-inspired origin. Evolutionary programming is also mostly used for numerical optimization problems. The practical implementations of the two approaches have mostly converged, although the scientific communities remain deeply distinct.

Over the years, researchers have also broadened the scope of genetic algorithms. They have been used for solving problems whose results are highly structured, like the traveling salesman problem where the solution is a permutation of the nodes in a graph. However, the term genetic algorithm remained strongly linked to the idea of fixed-length bit strings.

If not directly applicable within a different one, the ideas developed by researchers for one paradigm are at least inspiring for the whole community. The various approaches may be too different to directly interbreed, but many key ideas are now shared. Moreover, over the year a great number of minor and hybrid algorithms, not simply classifiable, have been described.

1.3 Genetic programming

The fourth and last evolutionary algorithm sketched in this is introduction is *genetic programming*, abbreviated as GP. Whereas μGP shares with it more in its name than in its essence, the approach presented in this book owes a deep debit to its underlying ideas.

Genetic programming was popularized by John Koza, who described it after having applied for a patent in 1989. The ambitious goal of the methodology is to create computer programs in a fully automated way, exploiting neo-Darwinism as an optimization tool. The original version was developed in *Lisp*, an interpreted computer language that dates back to the end of the 1950s. The Lisp language has the ability to handle fragments of code as data, allowing a program to build up its subroutines before evaluating them. Everything in Lisp is a prefix expression, except variables and constants. Genetic programming individuals were lisp programs, thus, they were prefix expressions too. Since the Lisp language is as flexible as inefficient, in the following years, researchers moved to alternative implementations, mostly using compiled language. Indeed, the need for computational power and the endeavor for efficiency have been constant pushes in the genetic programming research since its origin. While in Lisp the difference between an expression and a program was subtle, it became sharper in later implementations. Many algorithms proposed in the literature clearly tackle the former, and are hardly applicable to the latter.

Regardless of the language used, in genetic programming individuals are almost always represented internally as trees. In the simplest form, leaves, or terminals, are numbers. Internal nodes encode operations. More complex variations may take into account variables, complex functions, and programming structures. The offspring may be generated applying either recombination or, in recent implementations, mutation. The former is the exchange of sub-trees between the two parents. The latter is the random modification of the tree. Original genetic programming used huge populations, and emphasized recombination, with no, or very little, mutations. In fact, the substitution of a sub-tree is highly disruptive operation and may introduce a significant amount of novelty. Moreover, a large population ensures that all possible symbols are already available in the gene pool. Several mutations have been proposed, like promoting a sub-tree to a new individual, or collapsing a sub-tree to a single terminal node.

The genetic-programming paradigm attracted many researchers. Results were used as test benches for new practical techniques, as well as theoretical studies. It challenged and stimulated new lines of research. The various topics tackled included: representation of individuals; behavior of selection in huge populations; techniques to avoid the growth of trees; type of initializations. Some of this research has been inspiring for the development μGP.