Evolutionary Computation

Evolutionary Algorithms: Concepts and Applications

Andrea G. B. Tettamanzi

This paper was first published, in its original Italian version, under the title "*Algoritmi evolutivi: concetti e applicazioni*", by ***Mondo Digitale*** (issue no. 3, March 2005, pp. 3-17, available at <http://www.mondodigitale.net/>). ***Mondo Digitale***, a founding member of **UP**ENET, is the digital journal of the CEPIS Italian society AICA (*Associazione Italiana per l'Informatica ed il Calcolo Automatico*, <http://www.aicanet.it/>.)

(Keywords added by the Editor of **UP**GRADE.)

Evolutionary algorithms are a family of stochastic problem-solving techniques, within the broader category of what we might call "natural-metaphor models", together with neural networks, ant systems, etc. They find their inspiration in biology and, in particular, they are based on mimicking the mechanisms of what we know as "natural evolution". During the last twenty-five years these techniques have been applied to a large number of problems of great practical and economic importance with excellent results. This paper presents a survey of these techniques and a few sample applications.

Keywords: Evolutionary Algorithms, Evolutionary Computation, Natural-metaphor Models.

1 What Are Evolutionary Algorithms?

If we think about living beings, including humans, and their organs, their complexity, and their perfection, we cannot help but wonder how it was possible for such sophisticated solutions to have evolved autonomously. Yet there is a theory, initially proposed by Charles Darwin and later refined by many other natural scientists, biologists and geneticists, which provides a satisfactory explanation for most of these biological phenomena by studying the mechanisms which enable species to adapt to mutable and complex environments. This theory is supported by a considerable body of evidence and has yet to be refuted by any experimental data. According to Darwin's theory, these wonderful creations are simply the result of a purposeless evolutionary process, driven on the one hand by randomness and on the other hand by the law of the survival of the fittest. Such is natural evolution.

If such a process has been capable of producing something as sophisticated as the eye, the immune system, and even our brain, it would seem only logical to try and do the same by simulating the process on computers to attempt to solve complicated problems in the real world. This is the idea behind the development of evolutionary algorithms (see the box entitled "Some History" for the birth and evolution of these algorithms).

1.1 The Underlying Metaphor

Evolutionary algorithms are thus bio-inspired computer-science techniques based on a metaphor which is schematically outlined in Table 1. Just as an individual in a population of organisms must adapt to its surrounding environment to survive and reproduce, so a candidate solution must be adapted to solving its particular problem. The problem is the environment in which a solution lives within a population of other candidate solutions. Solutions differ from one another in terms of their quality, i.e., their cost or merit, reflected by the evaluation of the objective function, in the same way as the individuals of a population of organisms differ from one another in terms of their degree of adaptation to the environment; what biologists refer to as *fitness*. If natural selection allows a population of organisms to adapt to its surrounding environment, when applied to a population of solutions to a problem, it should also be able to bring about the evolution of better and better, and eventually, given enough time, optimal solutions.

Based on this metaphor, the computational model borrows a number of concepts and their relevant terms from biology: every solution is coded by means of one or more *chromosomes*; the *genes* are the pieces of encoding responsible for one or more *traits* of a solution; the *alleles* are the possible configurations a gene can take on; the

Andrea Tettamanzi is an Associate Professor at the Information Technology Dept. of the University of Milan, Italy. He received his M.Sc. in Computer Science in 1991, and a Ph.D. in Computational Mathematics and Operations Research in 1995. In the same year he founded Genetica S.r.l., a Milan-based company specialising in industrial applications for evolutionary algorithms and soft computing. He is active in research into evolutionary algorithms and soft computing, where he has always striven to bridge the gap between theoretical aspects and practical and applicational aspects. <andrea.tettamanzi@unimi.it>

EVOLUTION	PROBLEM SOLVING
Environment	Object problem
Individual	Candidate solution
Fitness	Solution quality

Table 1: A Schematic Illustration of The Metaphor Underlying Evolutionary Algorithms.

exchange of genetic material between two chromosomes is called *crossover*, whereas a perturbation to the code of a solution is termed *mutation* (see the box entitled "A Genetic Algorithm at Work" for an example).

Although the computational model involves drastic simpli-fications compared to the natural world, evolutionary algorithms have proved capable of causing surprisingly complex and interesting structures to emerge. Given appropriate encoding, any individual can be the representation of a particular solution to a problem, the strategy for a game, a plan, a picture, or even a simple computer program.

1.2 The Ingredients of An Evolutionary Algorithm

Now we have introduced the concepts, let us take a look at what an evolutionary algorithm consists of in practice.

An evolutionary algorithm is a stochastic optimisation technique that proceeds in an iterative way. An evolutionary algorithms maintains a population (which in this context means a multiset or *bag*, i.e., a collection of elements not necessarily all distinct from one another) of individuals representing candidate solutions for the problem at hand (the *object* problem), and makes it evolve by applying a (usually quite small) number of stochastic operators: *mutation*, *recombination*, and *selection*.

Mutation can be any operator that randomly perturbs a solution. Recombination operators decompose two or more distinct individuals and then combine their constituent parts to form a number of new individuals. Selection creates copies of those individuals that represent the best solutions within the population at a rate proportional to their fitness.

The initial population may originate from a random sampling of the solution space or from a set of initial solutions found by simple local search procedures, if available, or determined by a human expert.

Stochastic operators, applied and composed according to the rules defining a specific evolutionary algorithm, determine a stochastic population-transforming operator. Based on that operator, it is possible to model the workings of an evolutionary algorithm as a Markov chain whose states are populations. It is possible to prove that, given some entirely reasonable assumptions, such a stochastic process will converge to the global optimum of the problem [16].

When talking about evolutionary algorithms, we often hear the phrase *implicit parallelism*. This term refers to the fact that each individual can be thought of as a representative of a multitude of solution *schemata*, i.e., of partially specified solutions, such that, while processing a single individual, the evolutionary algorithm will in fact be implicitly processing at the same time (i.e., in parallel) all the solution schemata of which that individual is a representative. This concept should not be confused with the *inherent parallelism* of evolutionary algorithms. This refers to the fact that they carry out a

Some History

The idea of using selection and random mutation for optimisation tasks goes back to the fifties at least and the work of the statistician George E. P. Box, the man who famously said "*all models are wrong, but some are useful*". Box, however, did not make use of computers, though he did manage to formulate a statistical methodology that would become widely used in industry, which he called *evolutionary operation* [1]. At around the same time, other scholars conceived the idea of simulating evolution on computers: Barricelli and Fraser used computer simulations to study the mechanisms of natural evolution, while the bio-mathematician Hans J. Bremermann is credited as being the first person to recognise an optimisation process in biological evolution [2].

As often happens with pioneering ideas, these early efforts met with considerable scepticism. Nevertheless, the time was evidently ripe for those ideas, in an embryonic stage at that point, to be developed. A decisive factor behind their development was the fact that the computational power available at that time in major universities broke through a critical threshold, allowing evolutionary computation to be put into practice at last. What we recognise today as the original varieties of evolutionary algorithms were invented independently and practically simultaneously in the mid sixties by three separate research groups. In America, Lawrence Fogel and colleagues at the University of California in San Diego laid down the foundations of *evolutionary programming* [3], while at the University of Michigan in Ann Arbor John Holland proposed his first *genetic algorithms* [4]. In Europe, Ingo Rechenberg and colleagues, then students at the Technical University of Berlin, created what they called "evolution strategies" (*Evolutionsstrategien*) [5]. During the following 25 years, each of these three threads developed essentially on its own, until in 1990 there was a concerted effort to bring about their convergence. The first edition of the PPSN (*Parallel Problem Solving from Nature*) conference was held that year in Dortmund. Since then, researchers interested in evolutionary computation form a single, albeit articulated, scientific

A Genetic Algorithm at Work

We can take a close look at how a genetic algorithm works by using an example. Let us assume we have to solve a problem, called *maxone*, which consists of searching for all binary strings of length *l* for the string containing the maximum number of ones. At first sight this might seem to be a trivial problem, as we know the solution beforehand: it will be the string made up entirely of ones. However, if we were to suppose that we had to make *l* binary choices to solve a problem, and that the quality of the solution were proportional to the number of correct choices we made, then we would have a problem of equivalent difficulty, by no means easy to solve. In this example we assume that all correct choices correspond to a one merely to make the example easier to follow. We can therefore define the fitness of a solution as the number of ones in its binary coding, set *l* = 10, which is a number small enough to make things manageable, and try to apply the genetic algorithm to this problem.

First of all, we have to establish the size of the population. A sensible choice to begin with might be 6 individuals. At this point, we need to generate an initial population: we will do this by tossing a fair coin 60 times (6 individuals times 10 binary digits) and writing 0 if the outcome is 'heads' and 1 if the outcome is 'tails'. The initial population thus obtained is shown in Table A. Note that the average fitness in the initial population is 5.67.

NO.	INDIVIDUAL	FITNESS
1)	1111010101	7
2)	0111000101	5
3)	1110110101	7
4)	0100010011	4
5)	1110111101	8
6)	0100110000	3

Table A: The Initial Population of The Genetic Algorithm to Solve The *maxone* Problem, Showing The Fitness for All Individuals.

The evolutionary cycle can now begin. To use fitness-proportionate selection, the simplest method is to simulate throwing a ball into a special roulette wheel which has as many slots as individuals in the population (6 in this case). Each slot has a width that is to the circumference of the wheel as the fitness of the corresponding individual is to the sum of the fitness of all the individuals in the population (36 in this case). Therefore, when we spin the wheel, the ball will have a 7/34 probability of coming to rest in the individual 1 slot, 5/34 of landing in the individual 2 slot, and so on. We will have to throw the ball exactly 6 times in order to put together an intermediate population of 6 strings for reproduction. Let us assume the outcomes are: 1, 3, 5, 2, 4, and 5 again. This means two copies of individual 5 and a single copy of the other individuals with the exception of individual 6 will be used for reproduction. Individual 6 will not leave descendants. The next operator to be applied is recombination. Couples are formed, the first individual extracted with the second, the third with the fourth, and so forth. For each couple, we decide with a given probability, say 0.6, whether to perform crossover. Let us assume that we perform crossover with only the first and the last couple, with cutting points randomly chosen after the second digit and after the fifth digit respectively. For the first couple, we will have

11.11010101	becoming	11.10110101
11.10110101	"	11.11010101.

We observe that, since the parts to the left of the cutting point are identical, this crossover will have no effect. This contingency is more common than you might imagine, especially when, after many generations, the population is full of equally good and nearly identical individuals. For the third couple we will have instead

01000.10011	becoming	01000.11101
11101.11101	"	11101.10011.

All that remains is to apply mutation to the six strings resulting from recombination by deciding with a probability of, say, 1/10 for each digit whether to invert it. As there are 60 binary digits in total, we would expect an average of 6 mutations randomly distributed over the whole population. After applying all the genetic operators, the new population might be the one shown in Table B, where the mutated binary digits have been highlighted in bold type.

In one generation, the average fitness in the population has changed from 5.67 to 6.17, with an 8.8% increase. By iterating the same process again and again, very quickly we reach a point at which an individual made entirely of ones appears, the optimal solution to our problem.

NO.	INDIVIDUAL	FITNESS
1)	11101**0**0101	6
2)	1111**1**1010**0**	7
3)	11101**0**111**1**	8
4)	0111000101	5
5)	0100011101	5
6)	1110110**0**01	6

Table B: The Population of The Genetic Algorithm to Solve The *maxone* Problem after One Generation, Showing The Fitness for All Individuals.

population-based search, which means that, although for the sake of convenience they are usually expressed by means of a sequential description, they are particularly useful and easy to implement on parallel hardware.

1.3 Genetic Algorithms

The best way to understand how evolutionary algorithms work is to consider one of their simplest versions, namely genetic algorithms [6]. In genetic algorithms, solutions are represented as fixed-length binary strings. This type of representation is by far the most general, although, as we shall see below, not always the most convenient, although the fact is that any data structure, no matter how complex and articulated, will always be encoded in binary in a computer's memory. A sequence of two symbols, 0 and 1, from which it is possible to reconstruct a solution, is very reminiscent of a DNA thread made up of a sequence of four bases, A, C, G, and T, from which it is possible to reconstruct a living organism! In other words, we can consider a binary string as the DNA of a solution to the object problem.

A genetic algorithm consists of two parts:

1. a routine that generates (randomly or by using heuristics) the initial population;
2. an evolutionary cycle, which at each iteration (or *generation*), creates a new population by applying the genetic operators to the previous population.

The evolutionary cycle of the genetic algorithms can be represented using the pseudocode in Table 2. Each individual is assigned a particular *fitness* value, which depends on the quality of the solution it represents. The first operator to be applied is selection, whose purpose is to simulate the Darwinian law of the survival of the fittest. In the original version of genetic algorithms, that law is implemented by means of what is known as the fitness-proportionate selection: to create a new intermediate population of *n* 'parent' individuals, *n* independent extractions of an individual from the existing population are carried out, where the probability for each individual to be extracted is directly proportional to its fitness. As a consequence, above-average individuals will be extracted more than once on average, whereas below-average individuals will face extinction.

```
generation = 0
Initialize population
while not <termination condition> do
   generation = generation + 1
   Compute the fitness of all individuals
   Selection
   Crossover(p_cross)
   Mutation(p_mut)
end while
```

Table 2: Pseudocode Illustrating A Typical Simple Genetic Algorithm.

Once *n* parents are extracted as described, the individuals of the next generation will be produced by applying a number of reproduction operators, which may involve one parent only (thus simulating a sort of asexual reproduction) in which case we speak of *mutation*, or more than one parent, usually two (sexual reproduction), in which case we speak of *recombination*. In genetic algorithms, two reproduction operators are used: crossover and mutation.

To apply crossover, the parent individuals are mated two by two. Then, with a certain probability p_{cross}, called the "crossover rate", which is a parameter of the algorithm, each couple undergoes crossover itself. This is done by lining up the two binary strings, cutting them at a randomly chosen point, and swapping the right-hand halves, thus yielding two new individuals, which inherit part of their characters from one parent and part from the other.

After crossover, all individuals undergo mutation, whose purpose is to simulate the effect of random transcription errors that can happen with a very low probability p_{mut} every time a chromosome is duplicated. Mutation amounts to deciding whether to invert each binary digit, independently of the others, with probability p_{mut}. In other words, every zero has probability p_{mut} of becoming a one and *vice versa*.

The evolutionary cycle, according to how it is conceived, could go on forever. In practice, however, one has to decide when to halt it, based on some user-specified termination criterion. Examples of termination criteria are:

· a fixed number of generations or a certain elapsed time;
· a satisfactory solution, according to some particular criterion, has been found;
· no improvement has taken place for a given number of generations.

1.4 Evolution Strategies

Evolution strategies approach the optimisation of a real-valued objective function of real variables in an *l*-dimensional space. The most direct representation is used for the independent variables of the function (the solution), namely a vector of real numbers. Besides encoding the independent variables, however, evolution strategies give the individual additional information on the probability distribution to be used for its perturbation (mutation operator). Depending on the version, this information may range from just the variance, valid for all independent variables, to the entire variance-covariance matrix **C** of a joint normal distribution; in other words, the size of an individual can range from $l + 1$ to $l(l + 1)$ real numbers.

In its most general form, the mutation operator perturbs an individual in two steps:

1. It perturbs the **C** matrix (or, more exactly, an equivalent matrix of rotation angles from which the **C** matrix can be easily calculated) with the same probability distribution for all individuals;
2. It perturbs the parameter vector representing the solution to the optimisation problem according to a joint normal probability distribution

having mean **0** and the perturbed **C** as its variance-covariance matrix.

This mutation mechanism allows the algorithm to evolve the parameters of its search strategy autonomously while it is searching for the optimal solution. The resulting process, called *self-adaptation*, is one of the most powerful and interesting features of this type of evolutionary algorithm.

Recombination in evolution strategies can take different forms. The most frequently used are *discrete* and *intermediate* recombination. In discrete recombination, each component of the offspring individuals is taken from one of the parents at random, while in intermediate recombination each component is obtained by linear combination of the corresponding components in the parents with a random parameter.

There are two alternative selection schemes defining two classes of evolution strategies: (n, m) and $(n + m)$. In (n, m) strategies, starting from a population of n individuals, $m > n$ offspring are produced and the n best of them are selected to form the population of the next generation. In $(n + m)$ strategies, on the other hand, the n parent individuals participate in selection as well. Of those $n + m$ individuals, only the best n make it to the population of the next generation. Note that, in both cases, selection is deterministic and works "by truncation", i.e., by discarding the worst individuals. In this way, it is not necessary to define a non-negative fitness, and optimisation can consider the objective function, which can be maximised or minimised according to individual cases, directly.

1.5 Evolutionary Programming

Evolution, whether natural or artificial, has nothing 'intelligent' about it, in the literal sense of the term: it does not understand what it is doing, nor is it supposed to. Intelligence, assuming such a thing can be defined, is rather an 'emergent' phenomenon of evolution, in the sense that evolution may manage to produce organisms or solutions endowed with some form of 'intelligence'.

Evolutionary programming is intended as an approach to artificial intelligence, as an alternative to symbolic reasoning techniques. Its goal is to evolve intelligent behaviours represented through finite-state machines rather than define them *a priori*. In evolutionary programming, therefore, the object problem determines the input and output alphabet of a family of finite-state machines, and individuals are appropriate representations of finite-states machines operating on those alphabets. The natural representation of a finite-state machine is the matrix that defines its state-transition and output functions. The definition of the mutation and recombination operators is slightly more complex than in the case of genetic algorithms or evolution strategies, as it has to take into account the structure of the objects those operators have to manipulate. The fitness of an individual can be computed by testing the finite-state machine it represents on a set of instances of the problem. For example, if we wish to evolve individuals capable of modelling a historical series, we need to select a number of pieces from the previous series and feed them into an individual. We can then interpret the symbols produced by the individual as predictions and compare them with the actual data to measure their accuracy.

1.6 Genetic Programming

Genetic programming [7] is a relatively new branch of evolutionary algorithms, whose goal is an old dream of artificial intelligence: automatic programming. In a programming problem, a solution is a program in a given programming language. In genetic programming, therefore, individuals represent computer programs.

Any programming language can be used, at least in principle. However, the syntax of most languages would make the definition of the genetic operators that preserve it particularly awkward and burdensome. This is why early efforts in that direction found a sort of restricted LISP to be an ideal expression medium of expression. LISP has the advantage of possessing a particularly simple syntax. Furthermore, it allows us to manipulate data and programs in a uniform fashion. In practice, approaching a programming problem calls for the definition of a suitable set of variables, constants, and primitive functions, thus limiting the search space which would otherwise be unwieldy. The functions chosen will be those that *a priori* are deemed useful for the purpose. It is also customary to try and arrange things so that all functions accept the results returned by all others as arguments, as well as all variables and predefined constants. As a consequence, the space of all possible programs from which the program that will solve the problem is to be found will contain all possible compositions of functions that can be formed recursively from the set of primitive functions, variables, and predefined constants.

For the sake of simplicity, and without loss of generality, a genetic programming individual can be regarded as the parse tree of the corresponding program, as illustrated in Figure 1. The

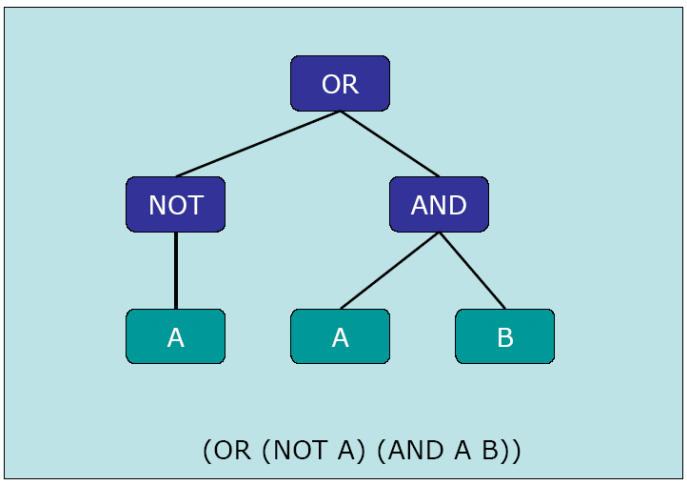

Figure 1: A Sample LISP Program with Its Associated Parse Tree.

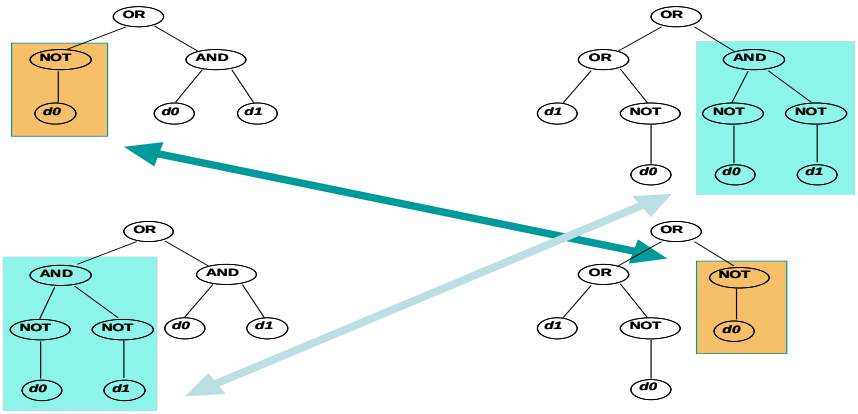

Figure 2: Schematic Illustration of Recombination in Genetic Programming.

recombination of two programs is carried out by randomly selecting a node in the tree of both parents and by swapping the subtrees rooted in the selected nodes, as illustrated in Figure 2. The importance of the mutation operator is limited in genetic programming, for recombination alone is capable creating enough diversity to allow evolution to work.

Computing the fitness of an individual is not so different from testing a program. A set of test cases must be given as an integral part of the description of the object problem. A test case is a pair (input data, desired output). The test cases are used to test the program as follows: for each case, the program is executed with the relevant input data; the actual output is compared with the desired output; and the error is measured. Finally, fitness is obtained as a function of the accumulated total error over the whole test set.

An even more recent approach to genetic programming is what is known as *grammatical evolution* [8], whose basic idea is simple but powerful: given the grammar of a programming language (in this case completely arbitrary, without limitations deriving from its particular syntax), consisting of a number of production rules, a program in this language is represented by means of a string of binary digits. This representation is decoded by starting from the target non-terminal symbol of the grammar and reading the binary digits from left to right – enough digits each time to be able to decide which of the applicable production rules should actually be applied. The production rule is then applied and the decoding continues. The string is considered to be circular, so that the decoding process never runs out of digits. The process finishes when no production rule is applicable and a well-formed program has therefore been produced, which can be compiled and executed in a controlled environment.

2 'Modern' Evolutionary Algorithms

Since the early eighties, evolutionary algorithms have been successfully applied to many real-world problems which are difficult or impossible to solve with exact methods and are of great interest to operations researchers. Evolutionary algorithms have gained a respectable place in the problem solver's toolbox, and this last quarter of a century has witnessed the coming of age of the various evolutionary techniques and their cross-fertilisation as well as progressive hybridisation with other technologies.

If there is one major trend line in this development process, it is the progressive separation from elegant representations, based on binary strings, of the early genetic algorithms, so suggestively close to their biological source of inspiration, and an increasing propensity for adopting representations closer to the nature of the object problem, ones which map more directly onto the elements of a solution, thus allowing all available information to be exploited to 'help', as it were, the evolutionary process to find its way to the optimum [9].

Adopting representations closer to the problem also necessarily implies designing mutation and recombination operators that manipulate the elements of a solution in an explicit, informed manner. On the one hand, those operators end up being less general, but on the other hand, the advantages in terms of performance are often remarkable and compensate for the increased design effort.

Clearly, the demand for efficient solutions has prompted a shift away from the coherence of the genetic model.

2.1 Handling Constraints

Real-world problems, encountered in industry, business, finance and the public sector, whose solution often has a significant economical impact and which constitute the main target of operations research, all share a common feature: they have complex and hard to handle constraints. In early work on evolutionary computation, the best way to approach constraint handling was not clear. Over time, evolutionary algorithms began to be appreciated as approximate methods for operations research and they have been able to take advantage of techniques and expedients devised within the framework of operations research for other approximate methods. Three main techniques emerged from this cross-fertilisation, which can be combined if needed, that enable nontrivial constraints to be taken into account in an evolutionary algorithm:

· the use of penalty functions;
· the use of decoders or repair algorithms;
· the design of specialised encodings and genetic operators.

Penalty functions are functions associated with each problem constraint that measure the degree to which a solution violates its relevant constraint. As the name suggests, these functions are combined with the objective function in order to penalise the fitness of individuals that do not respect certain constraints. Although the penalty function approach is a very general one, easy to apply to all kinds of problems, its use is not without pitfalls. If penalty functions are not accurately weighted, the algorithm can waste a

great deal of time processing infeasible solutions, or it might even end up converging to an apparent optimum which is actually impossible to implement. For instance, in a transportation problem, described by *n* factories and *m* customers to which a given quantity of a commodity has to be delivered, where the cost of transporting a unit of the commodity from every factory to any of the customers, a solutions that minimises the overall cost in an unbeatable way is the solution where absolutely nothing is transported! If the violation of the constraints imposing that the ordered quantity of the commodity is delivered to each customer is not penalised to a sufficient extent, the absurd solution of not delivering anything could come out as better than any solution that actually meets customers' orders. For some problems, called *feasibility problems*, finding a solution that doe not violate any constraint is almost as difficult as finding the optimum solution. For this kind of problems, penalty functions have to be designed with care or else the evolution may never succeed in finding any feasible solution.

Decoders are algorithms based on a parameterised heuristics, which aim to construct an optimal solution from scratch by making a number of choices. When such an algorithm is available, the idea is to encode the parameters of the heuristics into the individuals processed by the evolutionary algorithms, rather than the solution directly, and to use the decoder to reconstruct the corresponding solution from the parameter values. We have thus what we might call an *indirect* representation of solutions.

Repair algorithms are operators that, based on some heuristics, take an infeasible solution and 'repair' it by enforcing the satisfaction of one violated constraint, then of another, and so on, until they obtain a feasible solution. When applied to the outcome of genetic operators of mutation and recombination, repair algorithms can ensure that the evolutionary algorithm is at all times only processing feasible solutions. Nevertheless, the applicability of this technique is limited, since for many problems the computational complexity of the repair algorithm far outweighs any advantages to be gained from its use.

Designing specialised encodings and genetic operators would be the ideal technique, but also the most complicated to apply in all cases. The underlying idea is to try and design a solutions representation that, by its construction, is capable of encoding all and only feasible solutions, and to design specific mutation and recombination operators alongside it that preserve the feasibility of the solutions they are applied to. Unsurprisingly, as the complexity and number of constraints increases, this exercise soon becomes formidable and eventually impossible. However, when possible, this is the optimal way to go, for it guarantees the evolutionary algorithm processes feasible solutions only and therefore reduces the search space to the absolute minimum.

2.2 Combinations with Other Soft-Computing Techniques

Evolutionary algorithms, together with fuzzy logic and neural network, are part of what we might call *soft computing*, as opposed to traditional or *hard* computing, which is based on criteria like precision, determinism, and the limitation of complexity. Soft computing differs from hard computing in that it is tolerant of imprecision, uncertainty, and partial truth. Its guiding principle is to exploit that tolerance to obtain tractability, robustness, and lower solution costs.

Soft computing is not just a mixture of its ingredients, but a discipline in which each constituent contributes a distinct methodology for addressing problems in its domain, in a complementary rather than competitive way [10]. Thus evolutionary algorithms can be employed not only to design and optimise fuzzy systems, such as fuzzy rule bases or fuzzy decision trees, but also to improve the learning characteristics of neural networks, or even determine their optimal topology. Fuzzy logic can also be used to control the evolutionary process by acting dynamically on the algorithm parameters, to speed up convergence to the global optimum and escape from local optima, and to fuzzify, as it were, some elements of the algorithm, such as the fitness of individuals or their encoding. Meanwhile neural networks can help an evolutionary algorithm obtain an approximate estimate of the fitness of individuals for problems where fitness calculation requires computationally heavy simulations, thus reducing CPU time and improving overall performance.

The combination of evolutionary algorithms with other soft computing techniques is a fascinating research field and one of the most promising of this group of computing techniques.

3 Applications

Evolutionary algorithms have been successfully applied to a large number of domains. For purely illustrative purposes, and while this is not intended to be a meaningful classification, we could divide the field of application of these techniques into five broad domains:

- Planning, including all problems that require choosing the most economical and best performing way to use a finite set of resources. Among the problems in this domain are vehicle routing, transport problems, robot trajectory planning, production scheduling in an industrial plant, timetabling, determining the optimal load of a transport, etc.
- Design, including all those problem that require determining an optimal layout of elements (electronic or mechanic components, architectural elements, etc.) with the aim of meeting a set of functional, aesthetic, and robustness requirements. Among the problems in this domain are electronic circuit design, engineering structure design, information system design, etc.
- Simulation and identification, which requires determining how a given design or model of a system will behave. In some cases this needs to be done because we are not sure about how the system behaves, while in others its behaviour is known but the accuracy of a model has to be assessed. Systems under scrutiny may be chemical (determining the 3D structure of a protein, the equilibrium of a chemical reaction), economical (simulating the

dynamics of competition in a market economy), medical, etc.

- Control, including all problems that require a control strategy to be established for a given system;
- Classification, modelling and machine learning, whereby a model of the underlying phenomenon needs to be built based on a set of observations. Depending on the circumstances, such a model may consist of simply determining which of a number of classes an observation belongs to, or building (or learning) a more or less complex model, often used for prediction purposes. Among the problems in this domain is *data mining*, which consists of discovering regularities in huge amounts of data that are difficult to spot "with the naked eye".

Of course the boundaries between these five application domains are not clearly defined and the domains themselves may in some cases overlap to some extent. However, it is clear that together they make up a set of problems of great economic importance and enormous complexity.

In the following sections we will try to give an idea of what it means to apply evolutionary algorithms to problems of practical importance, by describing three sample applications in domains that differ greatly from one another, namely school timetabling, electronic circuit design, and behavioural customer modelling.

3.1 School Timetabling

The timetable problem consists of planning a number of meetings (e.g., exams, lessons, matches) involving a group of people (e.g., students, teachers, players) for a given period and requiring given resources (e.g., rooms, laboratories, sports facilities) according to their availability and respecting some other constraints. This problem is known to be NP-complete: that is the main reason why it cannot be approached in a satisfactory way (from the viewpoint of performance) with exact algorithms, and for a long time it has been a testbed for alternative techniques, such as evolutionary algorithms. The problem of designing timetables, in particular for Italian high schools, many of which are distributed over several buildings, is further complicated by the presence of very strict constraints, which makes it very much a feasibility problem.

Operation	Code	operand 1	operand 2	Description
Input	I	not used	not used	Copy input
Delay	D	*n*	not used	Delay *n* cycles
Left shift	L	*n*	*p*	multiply by 2^p
Right shift	R	*n*	*p*	divide by 2^p
Add	A	*n*	*m*	add
Subtract	S	*n*	*m*	subtract
Complement	C	*n*	not used	complement input

Table 3: Primitive Operations for The Representation of Digital Filters. (The format of the primitives is fixed, with two operands, of which only the required operands are used. The integers *n* and *m* refer to the inputs at cycles *t* – *n* and *t* – *m* respectively.)

An instance of this problem consists of the following entities and their relations:

- rooms, defined by their type, capacity, and location;
- subjects, identified by their required room type;
- teachers, characterised by the subjects they teach and their availability;
- classes, i.e., groups of students following the same curriculum, assigned to a given location, with a timetable during which they have to be at school;
- lessons, meaning the relation <*t*, *s*, *c*, *l*>, where *t* is a teacher, *s* is a subject, *c* is a class and *l* is its duration expressed in *periods* (for example, hours); in some cases, more than one teacher and more than one class can participate in a lesson, in which case we speak of *grouping*.

This problem involves a great many constraints, both hard and soft, too many for us to go into now in this article. Fortunately, anybody who has gone to a high school in Europe should at least have some idea of what those constraints might be.

This problem has been approached by means of an evolutionary algorithm, which is the heart of a commercial product, *EvoSchool* [11]. The algorithm adopts a 'direct' solution representation, which is a vector whose components correspond to the lessons that have to be scheduled, while the (integer) value of a component indicates the period in which the corresponding lesson is to begin. The function that associates a fitness to each timetable, one of the critical points of the algorithm, is in practice a combination of penalty functions with the form

$$f(x) = 1/\Sigma_i\, \alpha_i\, h_i + \gamma/\Sigma_j \beta_j\, s_j$$

where h_i is the penalty associated with the violation of the *i*th hard constraint, s_j is the penalty associated with the violation of the *j*th soft constraint, and parameters α_i and β_j are appropriate weightings associated with each constraint. Finally, γ is an indicator whose value is 1 when all hard con-

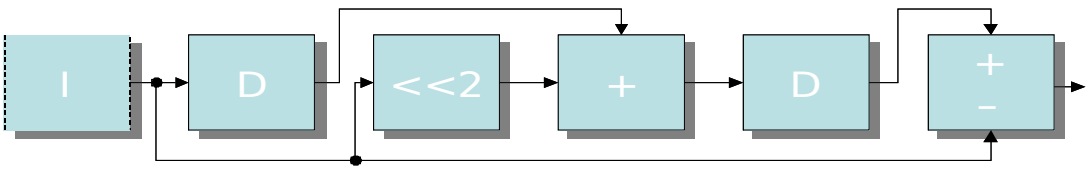

Figure 3: A Schematic Diagram of A Sample Circuit Obtained by Composition of 6 Primitive Operations.

straints are satisfied and zero otherwise. In effect, this means that soft constraints are taken into consideration only after all hard constraints have been satisfied.

All other ingredients of the evolutionary algorithm are fairly standard, with the exception of the presence of two mutually exclusive perturbation operators, called by the mutation operator, each with its own probability:
· intelligent mutation;
· improvement.

Intelligent mutation, while preserving its random nature, is aimed at performing changes that do not decrease the fitness of the timetable to which it is applied. In particular, if the operator affects the *i*th lesson, it will propagate its action to all the other lessons involving the same class, teacher or room. The choice of the "action range" of this operator is random with any given probability distribution. In practice, the effect of this operator is to randomly move some interconnected lessons in such a way as to decrease the number of constraint violations.

Improvement, in contrast, restructures an individual to a major extent. Restructuring commences by randomly selecting a lesson and concentrates on the partial timetables for the relevant class, teacher, or room. It compacts the existing lessons to free up enough space to arrange the selected lesson without conflicts.

A precisely balanced interaction between these two operators is the secret behind the efficiency of this evolutionary algorithm, which has proven capable of generating high quality timetables for schools with thousands of lessons to schedule over different buildings scattered over several sites. A typical run takes a few hours on a not so powerful PC of the kind to be found in high schools.

3.2 Digital Electronic Circuit Design

One of the problems that has received considerable attention from the international evolutionary computation community is the design of finite impulse response digital filters. This interest is due to their presence in a large number of electronic devices that form part of many consumer products, such as cellular telephones, network devices, etc.

The main criterion of traditional electronic circuit design methodologies is minimising the number of transistors used and, consequently, production costs. However, another very significant criterion is power absorption, which is a function of the number of logic transitions affecting the nodes of a circuit. The design of minimum power absorption digital filters has been successfully approached by means of an evolutionary algorithm [12].

A digital filter can be represented as a composition of a very small number of elementary operations, like the primitives listed in Table 3. Each elementary operation is encoded by means of its own code (one character) and two integers, which represent the relative offset (calculated backwards from the current position) of the two operands. When all offsets are positive, the circuit does not contain any feedback and the resulting structure is that of a finite impulse response filter. For example, the individual

(I 0 2) (D 1 3) (L 2 2) (A 2 1) (D 1 0) (S 1 5)

corresponds to the schematic diagram in Figure 3.

The fitness function has two stages. In the first stage, it penalises violations of the filter frequency response specifications, represented by means of a 'mask' in the graph of frequency response. In the second stage, which is activated when the frequency response is within the mask, fitness is inversely proportional to the circuit activity, which in turn is directly proportional to power absorption.

The evolutionary algorithm which solves this problem requires a great deal of computing power. For this reason, it has been implemented as a distributed system, running on a cluster of computers according to an island model, whereby the population is divided into a number of islands, residing on distinct machines, which evolve independently, except that, every now and then, they exchange 'migrant' individuals, which allow genetic material to circulate while at the same time keeping the required communication bandwidth as small as we wish.

A surprising result of the above evolutionary approach to electronic circuit design has been that the digital filters discovered by evolution, besides having a much lower power absorption in comparison with the corresponding filters obtained using traditional design techniques, as was intended, they also bring about a 40% to 60% reduction in the number of logic elements and, as a consequence, in area and speed as well. In other words, the decrease in consumption has not been achieved at the expense of production cost and speed. On the contrary, it has brought about an overall increase in efficiency in comparison with traditional design methods.

3.3 Data Mining

A critical success factor for any business today is its ability to use information (and knowledge that can be extracted from information) effectively. This strategic use of data can result in opportunities presented by discovering hidden, previously undetected, and frequently extremely valuable facts about consumers, retailers, and suppliers, and business trends in general. Knowing this information, an organisation can formulate effective business, marketing, and sales strategies; precisely target promotional activity; discover and penetrate new markets; and successfully compete in the marketplace from a position of informed strength. The task of sifting information with the aim of obtaining such a competitive advantage is known as *data mining* [13]. From a technical point of view, data mining can be defined as the search for correlations, trends, and patterns that are difficult to perceive "ith the naked eye" by digging into large amounts of data stored in warehouses and large databases, using statistical, artificial intelligence, machine learning, and soft computing techniques. Many large companies and organisations, such as banks, insurance companies, large retailers, etc., have a huge amount of information about their customers' behaviour. The possibility

of exploiting such information to infer behaviour models of their current and prospective customers with regard to specific products or classes of products is a very attractive proposition for organisations. If the models thus obtained are accurate, intelligible, and informative, they can later be used for decision making and to improve the focus of marketing actions,.

For the last five years the author has participated in the design, tuning, and validation of a powerful data mining engine, developed by Genetica S.r.l. and Nomos Sistema S.p.A (now an Accenture company) in collaboration with the University of Milan, as part of two Eureka projects funded by the Italian Ministry of Education and University.

The engine is based on a genetic algorithm for the synthesis of predictive models of customer behaviour, expressed by means of sets of fuzzy IF-THEN rules. This approach is a clear example of the advantages that can be achieved by combining evolutionary algorithms and fuzzy logic.

The approach assumes a data set is available: that is, a set as large as we like of records representing observations or recordings of past customer behaviour. The field of applicability could be even wider: the records could be observations of some phenomenon, not necessarily related to economy or business, such as the measurement of free electrons in the ionosphere [14].

A record consists of m attributes, i.e., values of variables describing the customer. Among these attributes, we assume that there is an attribute measuring the aspect of customer behaviour we are interested in modelling. Without loss of generality, we can assume there is just one attribute of this kind — if we were interested in modelling more than one aspect of behaviour, we could develop distinct models for each aspect. We could call this attribute 'predictive', as it is used to predict a customer's behaviour. Within this conceptual framework, a model is a function of $m - 1$ variables which returns the value of the predictive attribute depending on the value of the other attributes.

The way we choose to represent this function is critical. Experience proves that the usefulness and acceptability of a model does not derive from its accuracy alone.

Accuracy is certainly a necessary condition, but more important is the model's intelligibility for the expert who will have to evaluate it before authorising its use. A neural network or a LISP program, to mention just two alternative 'languages' that others have chosen to express their models, may provide killer results when it comes to accuracy. However, organisations will be reluctant to 'trust' the results of the model unless they can understand and explain how the results have been obtained.

This is the main reason for using sets of fuzzy IF-THEN rules as the language for expressing models. Fuzzy IF-THEN rules are probably the nearest thing to the intuitive way experts express their knowledge, due to the use of rules that express relationships between linguistic variables (which take on linguistic values of the type LOW, MEDIUM, HIGH). Also, fuzzy rules have the desirable property of behaving in an interpolative way, i.e., they do not jump from one conclusion to the opposite because of a slight change in the value of a condition, as is the case with crisp rules.

The encoding used to represent a model in the genetic algorithm is quite complicated, but it closely reflects the logical structure of a fuzzy rule base. It allows specific mutation and recombination operators to be designed which operate in an informed way on their constituent blocks. In particular, the recombination operator is designed in such a way as to preserve the syntactic correctness of the models. A child model is obtained by combining the rules of two parent models: every rule in the child model may be inherited from either parent with equal probability. Once inherited, a rule takes on all the definitions of the linguistic values (fuzzy sets) of the source parent model that contribute to determining its semantics.

Models are evaluated by applying them to a portion of the data set. This yields a fitness value gauging their accuracy. As is customary in machine learning, the remaining portion of the data set is used to monitor the generalisation capability of the models and avoid overfitting, which happens when a model learns one by one the examples it has seen, instead of capturing the general rules which can be applied to cases never seen before.

The engine based on this approach has been successfully applied to credit scoring in the banking environment, to estimating customer lifetime value in the insurance world [15], and to the collection of consumer credit receivables.

4 Conclusions

With this short survey on evolutionary algorithms we have tried to provide a complete, if not exhaustive - for obvious reasons of space -, overview of the various branches into which they are traditionally divided (genetic algorithms, evolution strategies, evolutionary programming and genetic programming). We have gone on to provide some information about the most significant issues concerning the practical application of evolutionary computing to problems of industrial and economic importance, such as solution representation and constraint handling, issues in which research has made substantial progress in the last few years. Finally, we have completed the picture with a more in-depth, but concise, illustration of three sample applications to "real-world" problems, chosen for being in domains which are as different from one another as possible, with the idea of providing three complementary views on the criticalities and the issues that can be encountered when implementing a software system that works. Readers should appreciate the versatility and the enormous potential of these techniques which are still coming of age almost forty years after their introduction. Unfortunately, this survey necessarily lacks an illustration of the theoretical foundations of evolutionary computing, which includes the schema theorem (with its so-called *building block* hypothesis) and the convergence theory. These topics have been omitted on purpose, since they would have required a level of formality unsuited to a survey. Interested

Evolutionary Algorithms on The Internet

Below are a few selected websites where the reader can find introductory or advanced information about evolutionary algorithms:

- <http://www.isgec.org/>: the portal of the International Society for Genetic and Evolutionary Computation;
- <http://evonet.lri.fr/>: the portal of the European network of excellence on evolutionary algorithms;
- <http://www.aic.nrl.navy.mil/galist/>: the GA Archives, originally the "GA-List" mailing list archives, now called the "EC Digest"; it contains up-to-date information on major events in the field plus links to other related web pages;
- <http://www.fmi.uni-stuttgart.de/fk/evolalg/index.html>: the EC Repository, maintained at Stuttgart University.

readers can fill this gap by referring to the bibliography below. Another aspect that has been overlooked because it is not really an 'application', although it is of great scientific interest, is the impact that evolutionary computation has had on the study of evolution itself and of complex systems in general (for an example, see the work by Axelrod on spontaneous evolution of co-operative behaviours in a world of selfish agents [18]).

Readers wishing to look into the field of evolutionary computation are referred to some excellent introductory books [6][9][17][19] or more in-depth treatises [20][21], or can browse the Internet sites mentioned in the box "Evolutionary Algorithms on the Internet".

References

[1] George E. P. Box, N. R. Draper. *Evolutionary Operation: Statistical Method for Process Improvement.* John Wiley & Sons, 1969.

[2] Hans J. Bremermann. "Optimization through Evolution and Recombination". In M. C. Yovits, G. T. Jacobi and G. D. Goldstein (editors), *Self-Organizing Systems 1962*, Spartan Books, Washington D. C., 1962.

[3] Lawrence J. Fogel, A. J. Owens, M. J. Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley & Sons, New York, 1966.

[4] John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

[5] Ingo Rechenberg. *Evolutions strategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzboog, Stuttgart, 1973.

[6] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[7] John R. Koza. *Genetic Programming.* MIT Press, Cambridge, Massachusetts, 1992.

[8] Michael O'Neill, Conor Ryan. *Grammatical Evolution. Evolutionary automatic programming in an arbitrary language.* Kluwer, 2003.

[9] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs, 3rd Edition.* Springer, Berlin, 1996.

[10] Andrea G. B. Tettamanzi, Marco Tomassini. *Soft Computing. Integrating evolutionary, neural, and fuzzy systems.* Springer, Berlin, 2001.

[11] Calogero Di Stefano, Andrea G. B. Tettamanzi. " An Evolutionary Algorithm for Solving the School Time-Tabling Problem". In E. Boers *et al.*, *Applications of Evolutionary Computing. EvoWorkshops 2001*, Springer, 2001. Pages 452–462.

[12] Massimiliano Erba, Roberto Rossi, Valentino Liberali, Andrea G. B. Tettamanzi. "Digital Filter Design Through Simulated Evolution". Proceedings of ECCTD'01 - European Conference on Circuit Theory and Design, August 28-31, 2001, Espoo, Finland.

[13] Alex Berson, Stephen J. Smith. *Data Warehousing, Data Mining & OLAP*, McGraw Hill, New York, 1997.

[14] Mauro Beretta, Andrea G. B. Tettamanzi. "Learning Fuzzy Classifiers with Evolutionary Algorithms". In A. Bonarini, F. Masulli, G. Pasi (editors), *Advances in Soft Computing*, Physica-Verlag, Heidelberg, 2003. Pagg. 1–10.

[15] Andrea G. B. Tettamanzi *et al.* "Learning Environment for Life-Time Value Calculation of Customers in Insurance Domain". In K. Deb *et al.* (editors), Proceedings of the Genetic and Evolutionary Computation Congress (GECCO 2004), Seattle, June 26–30, 2004. Pages II-1251–1262.

[16] Günter Rudolph. *Finite Markov Chain Results in Evolutionary Computation: A Tour d'Horizon*. Fundamenta Informaticae, vol. 35, 1998. Pages 67–89.

[17] Melanie Mitchell. *An Introduction to Genetic Algorithms*. Bradford, 1996.

[18] Robert Axelrod. *The Evolution of Cooperation*. Basic Books, 1985.

[19] David B. Fogel. *Evolutionary Computation: Toward a new philosophy of machine intelligence, 2nd Edition*. Wiley-IEEE Press, 1999.

[20] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996.

[21] Thomas Bäck, David B. Fogel, Zbigniew Michalewicz (editors). *Evolutionary Computation* (2 volumes). IoP, 2000.