# Evolutionary computing: the most powerful problem solver in the universe?

A.E. Eiben
http://www.cs.vu.nl/~gusz/
Faculty of Sciences
Vrije Universiteit Amsterdam

**Abstract**

In this paper we review evolutionary computing (EC) – the science of building, applying and studying algorithms based on the Darwinian principles of natural selection. We briefly introduce the main concepts behind evolutionary computing, and sketch the history of EC. Then we present the main components all evolutionary algorithms (EA) share and sketch differences between different types of EAs. Thereafter we consider two representatives, genetic algorithms and evolution strategies, in more detail. Finally, we discuss what EAs are good for, that is, we touch upon their strengths and weaknesses as well as some application areas ranging from combinatorial problems to entertainment and artificial life simulations.

## 1 Motivation and 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, respectively evolutionary computing. The fundamental metaphor of evolutionary computing 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 –

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

Table 1: The basic evolutionary computing metaphor linking natural evolution to problem solving

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 [14]. As early as in 1948 Turing proposed "genetical or evolutionary search" 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, [15, 13], while Holland called his method a genetic algorithm [16, 17, 21]. In Germany Rechenberg and Schwefel invented evolution strategies [22, 23]. 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 [1, 2, 3, 10, 20]. It was also in the early nineties that a fourth stream following the general ideas has emerged: Koza's genetic programming [4, 18]. 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 What is an evolutionary algorithm?

As the history of the field suggests there are many different variants of evolutionary algorithms. The common underlying idea behind all these techniques is the same: given a population of individuals the environmental pressure causes natural selection (survival of the fittest) and hereby the fitness of the population is growing. It is easy to see such a process as optimization. Given an objective function to be maximized we can randomly create a set of candidate solutions, i.e., elements of the objective function's domain, and

apply the objective function as an abstract fitness measure – the higher the better. Based on this fitness, some of the better candidates are chosen to seed the next generation by applying recombination and/or mutation to them. Recombination is a binary operator applied to two selected candidates (the so-called parents) and results one or two new candidates (the children). Mutation is a unary operator, it is applied to one candidate and results in one new candidate. Executing recombination and mutation leads to a set of new candidates (the offspring) that compete – based on their fitness – with the old ones for a place in the next generation. This process can be iterated until a solution is found or a previously set computational limit is reached. In this process selection acts as a force pushing quality, while variation operators (recombination and mutation) create the necessary diversity. Their combined application leads to improving fitness values in consecutive populations, that is, the evolution is optimizing.[1]

Let us note that many components of such an evolutionary process are stochastic. So is selection, where fitter individuals have a higher chance to be selected than less fit ones, but typically even the weak individuals have a chance to become a parent or to survive. For recombination of two individuals it holds that the choice on which pieces will be recombined is random. Similarly for mutation, the pieces that will be mutated within a candidate solution and the new pieces replacing the old ones are chosen randomly. The general scheme of an evolutionary algorithm can be given as follows.

```
INITIALIZE population with random candidate solutions
COMPUTE FITNESS of each candidate
    while not STOP-CRITERION do
        SELECT parents
        RECOMBINE pairs of parents
        MUTATE the resulting offspring
        COMPUTE FITNESS of new candidates
        REPLACE some parents by some offspring
    od
```

For the sake of completeness, let us note that the replacement step is often called survivor selection. It is easy to see that the above scheme falls in the category of generate-and-test algorithms. The fitness function represents a heuristic estimation of solution quality and the search process is driven by the variation and the selection operators. Evolutionary algorithms (EA) posses a number of features that can help to position them within in the family of generate-and-test methods:

[1] Actually, evolution is not "optimizing" as there are no general guarentees for finding an optimum, it is rather "approximizing", by approaching optimal values closer and closer over its course.

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

The aforementioned dialects of evolutionary computing follow the above general outlines and differ only in technical details. For instance, the representation of a candidate solution is often used to characterize different streams. Traditionally, the candidates are represented by (i.e., the data structure encoding a solution has the form of) bit-strings in genetic algorithms (GA), real-valued vectors in evolution strategies (ES), finite state machines in evolutionary programming (EP) and trees in genetic programming (GP). These differences have a mainly historical origin. Technically, a given representation might be preferable over others if it matches the given problem better, that is, it makes the encoding of candidate solutions easier or more natural. For instance, for solving a satisfiability problem the straightforward choice is to use bit-strings of length $n$, where $n$ is the number of logical variables, hence the appropriate EA would be a genetic algorithm. For evolving a computer program that can play checkers trees are well-suited (namely, the parse trees of the syntactic expressions forming the programs), thus a GP approach is likely. It is important to note that the recombination and mutation operators working on candidates must match the given representation. That is, for instance in GP the recombination operator works on trees, while in GAs it operates on bit-strings. As opposed to variation operators, selection takes only the fitness information into account, hence it works independently from the actual representation. Differences in the commonly applied selection mechanisms in each stream are therefore rather a tradition than a technical necessity.

It is worth to note that the borders between the four main EC streams are diminishing in the last decade. This "unionism" has a technical and a psychological aspect. On the one hand, many EAs have been proposed that are hard to classify along the traditional lines. On the other hand, more and more EC researchers and practitioners follow a pragmatic attitude choosing whichever type of representation, variation operators, or selection procedures are appropriate, without bothering much about whether the resulting combination fits in one of the traditional categories.

# 3 Genetic algorithms

In this section we go into more details and illustrate the working of EAs by discussing a well-known type: genetic algorithms.

### 3.1 The simple genetic algorithm

Although during the last two decades several GA variants have been introduced the oldest version, named simple GA, can be easily specified by the particular instantiations of the EA components as shown in Table 2.

| | |
|---|---|
| Representation | bit-strings |
| Recombination | 1-point crossover |
| Mutation | bit-flip |
| Selection | fitness-proportional |
| Replacement | generational |

Table 2: Sketch of the simple GA

In the sequel we will consider these components one by one. The first step in handling a problem by an EA is to define the representation. In evolutionary terms, one needs to specify which genotypes, also called chromosomes, represent (encode) the given phenotypes, the candidate solutions. To illustrate this matter let us take an extremely simple problem: maximizing $f(x) = x^2$ on $\mathbb{N}$ between 0 en 31. The simple GA representation would use bit-strings of length 5 with the obvious encoding, e.g. 11000 denoting 24. This defines the genotype space $\{0,1\}^5$, where the genetic search will take place. The fitness value of a given genotype (bit-string) is the square of the uniquely defined phenotype (integer) it represents.

Fitness proportional selection assigns selection probabilities to chromosomes proportionally with their fitness: $Prob(c_i) = f(x(c_i))/\sum_{j=1}^{m} f((x(c_j))$, where $m$ is the population size, $c_j \in \{1,\ldots,m\}$ denote the chromosomes, and $x(c_i)$ stands for the integer encoded by $c_i$. Then $m$ independent drawings with replacement, based on these selection probabilities, are performed to select $m$ chromosomes that will undergo crossover and mutation. The selected chromosomes form the mating pool. Note that this mechanism is biased, in the sense that chromosomes with a higher fitness get a higher selection probability and expectedly deliver more copies into the mating pool. After the mating pool has been established recombination, implemented by 1-point crossover, and mutation are executed.

For recombination, the mating pool is divided into randomly selected pairs and 1-point crossover is applied to each pair. An algorithm parameter called crossover rate $p_c$ prescribes the probability of actually executing the operator on a given pair – with a chance of $1 - p_c$ it will not be performed and the two children are simply identical copies of the parents. When executing 1-point crossover, first a crossover point is selected randomly, telling after which bit position the strings will be crossed. Then the two strings are broken at this point and the "tails" are exchanged. A straightforward generalization of this operator is the n-point crossover that uses $n$ crossover