An Introduction to Evolutionary Computation

Wim Hordijk

Karunya Institute of Technology and Sciences
Deemed University
Karunya Nagar, Coimbatore – 641 114
wim@santafe.edu

Introduction

Evolutionary Computation (EC) deals with computational methods that incorporate ideas and principles from natural evolution. More specifically, it includes a class of so-called *evolutionary algorithms* (EAs), which are heuristic search algorithms that can be used to search for good solutions to difficult problems. These algorithms try to "evolve" better and better solutions, as opposed to constructing one from scratch, and are particularly suitable for problems for which no known efficient (polynomial-time) algorithm exists, and for which exhaustive search is impractical. These evolutionary algorithms are fairly simple and general, and can be applied to a wide range of search and optimization problems. However, to get an appreciation for how and why they work, it is useful to understand the basics of genetics and natural evolution.

This paper is organized as follows. First, a brief overview of genetics and evolution is presented. Then, the concepts of search spaces, representations, and fitness functions are explained. Next, one particular evolutionary algorithm, namely the genetic algorithm, is explained in some detail, after which its advantages and disadvantages are discussed. A list of applications is then presented to give an idea of the kinds of problems that a genetic algorithm can be applied to successfully. Finally, some pointers to more information about genetic algorithms and evolutionary computation are provided.

Genetics and Evolution

In biology, a distinction is made between the genotype and the phenotype of an organism. The *genotype* is the genetic encoding of an organism, in particular the DNA molecules that reside in each and every cell in the body of an organism (whether it is a bacteria, fungus, plant, or animal). These DNA molecules contain *genes,* small subsets of DNA that encode for certain *traits* (or characteristics) of the organism, for example skin or fur color, number of legs, brain size, etc. More specifically, these genes, or pieces of DNA, are translated into proteins which form the basic building blocks of our bodies, and which perform the necessary functions for staying alive (metabolism, muscle activity, defense against disease, etc.). Together, the appearance of an organism and its functionality or behavior, form the *phenotype*, or the actual organism as it lives in the real world. So, in short, there is a one-way process where the genetic information (genotype) determines (together with environmental influences) the shape and functionality (phenotype) of an organism.

When two organisms of the same species get together to mate and create offspring, what happens at the genetic level is the following. Copies of the DNA molecules of both parents are mixed up, and the children receive part of their genotype from the father, and part from the mother. In other words, the genotype of an offspring individual is literally a (random) mix of the genotypes of its parents, a different mix for each child. This process is called *crossover*. Furthermore, during the process of copying the genetic information from parents to offspring, sometime small errors occur, called *mutations*. These happen only very infrequently, but they do introduce some (possibly) new variation in the genetic information. As a result of these crossover and mutation processes, the offspring individuals (i.e., the next generation) will have some variation compared to their parents. Some traits of an individual in the next generation will have come from its father, and some from its mother. Moreover, occasional mutations might be expressed in some entirely new variation in a certain trait. In short, there is descent with variation, where the variation is introduced at the level of the genotypes (through crossover and mutation), and expressed at the level of the phenotypes.

In the real world, not all individuals that are being born will survive and create offspring themselves. For example, in a rabbit population, most baby rabbits will not survive until adulthood: they might get eaten by predators, they will have to compete with other rabbits for food sources, or they may encounter any other natural or accidental causes of death before they have a chance to mate. Consequently, only a small portion of each generation of rabbits will actually create offspring themselves, and pass on their genetic information to the next generation. In particular, those rabbits with "good" traits, such as those able to run fast or with a fur color that allows them to hide in the bushes to escape predators, will (on average) survive and create offspring. Individuals with "bad" traits (or "not-so-good" traits), however, will not be very successful (again on average) in surviving, and will not be able to create offspring. This process, summarized with the familiar expression "survival of the fittest", is called *natural selection*: advantageous traits (or variations in traits) will on average be preserved and proliferate in the population, and disadvantageous traits (or variations) will on average disappear again, because they are not passed on to subsequent generations. Over time, the population as a whole will thus slowly change, or evolve, due to genetic variation, and adapt to cope better with the environmental challenges ("survival") through natural selection.

This, in a nutshell, is how evolution and natural selection happen in the real world. Of course this is a somewhat simplified picture, but these main principles are what is of importance here. One particular view of evolution that has become very popular is that of evolution as a *problem solver*. The "problem" is that of survival and reproduction, and evolution through natural selection is the mechanism of "solving" the problem. In other words, evolution searches through the space of possible DNA molecules (genotypes) for those that encode for successful phenotypes that are able to survive and create offspring. Those genotypes that encode for successful (fit) phenotypes will, on average, be used to produce new genotypes (through genetic variation) to make up the next generation. This way, over subsequent generations, the individuals in the population will become more and more successful at "solving the problem."

Search Spaces, Representations, and Fitness Functions

To use the principle of evolution in computer search algorithms, first we need to introduce the concept of search spaces, (genetic) representations, and fitness functions.

Search spaces

The *search space* of a given problem is simply the space of all possible (candidate) solutions to that problem. For example, consider the subset sum problem: given a set of N numbers x_i, is there a subset S of these numbers such that the sum of the numbers in this subset add up to some given number M? Or in the optimization version of this problem one could ask for a subset for which the sum is as close as possible to M. Obviously, the collection of all candidate solutions to this problem (the search space) is the set of all possible subsets of N numbers, of which there are 2^N (each number is either in a subset, or it is not). As another example, consider the traveling salesman problem (TSP). Here, the object is to find the shortest tour along N cities (which are linked by roads with certain distances), such that each city is visited exactly once. Here, the search space is the collection of all possible tours along these N cities, of which there are $N!$.

Representations

Sometimes it is convenient to represent candidate solutions to some problem in a simple way, e.g. with strings of characters, which can be easily manipulated using a computer program. For example, in the subset sum problem each candidate solution can be represented by a bit string (a string of 0s and 1s) of length N. Given a particular bit string, the corresponding candidate solution can be easily reconstructed by including the number x_i in the subset if its corresponding bit value (i.e., the i^{th} bit in the string) is 1. Similarly, possible tours along N cities can be represented by permutations of the numbers 1 to N. The order of the numbers in a given permutation simply indicates the order in which to visit the cities in the corresponding candidate solution, or tour. This way of representing candidate solutions to a given problem is analogous to the genotype-phenotype distinction in biology. The representations (bit strings, permutations) are the genotypes and the corresponding candidate solutions (subsets, tours) are the phenotypes. Thus, the search space of subsets is represented by the space of bit strings (of length N), and the search space of tours is represented by the space of permutations (of length N).

Fitness functions

In order to apply an evolutionary algorithm to a given problem, we need some way of evaluating candidate solutions. This is done with a fitness function. A *fitness function* is a function that takes as input a representation (or genotype), translates this into the corresponding candidate solution (or phenotype), tests this candidate solution on the given problem, and then returns a number that indicates how good this solution is, i.e., its *fitness*. For example, in the subset sum problem, the fitness function takes as input a bit string, from this it constructs the corresponding subset of numbers, and compares the sum of this subset to the given number M. In the TSP problem, the input is a permutation, which gets translated into the corresponding tour, and the length of this tour is then calculated.

Since we generally view evolution as a *maximization* problem, i.e., fitness should be maximized, a fitness function is usually constructed in such a way that a higher fitness value is assigned to better solutions. In the examples given, we are dealing with *minimization* problems (as is often the case in real-world problems), so better solutions will have lower values (such as a smaller difference with the given number *M,* or shorter tour length). However, every minimization problem can be easily converted into a maximization problem, and the fitness function could, for example, return the inverse of the calculated fitness values, or some large number minus the calculated value. So, here we will only consider evolutionary algorithms as *maximizing* the fitness values, in analogy with biology. The fitness function for a given problem (whether it is a minimization or maximization problem) can always be written in the appropriate form.

Genetic Algorithms

As one particular example of an evolutionary algorithm, we will now give an overview of the *genetic algorithm* (GA). GAs were originally developed by John Holland in the 60s and 70s [1]. For a detailed introduction, including examples and applications, see [2-4]. A GA searches for good solutions to a problem by maintaining a population of candidate solutions (suitably encoded), and creating subsequent generations by selecting (on average) the current best solutions and using operators like crossover and mutation to create new candidate solutions. Thus, better and better solutions are "evolved" over time.

To apply a GA, first a suitable encoding (a set of genotypes) needs to be chosen to represent candidate solutions to the given problem. Examples of some possible representations were given in the previous section. Next, an appropriate fitness function needs to be constructed. As described above, this function is used to translate a genotype to its corresponding candidate solution, and then evaluate this solution on the given problem. A "better" solution will have a higher fitness value. The choice of representation and the construction of the fitness function depend, of course, largely on the problem at hand. Given a representation and fitness function, the algorithm now roughly works as follows (implementation details are discussed below):

Genetic Algorithm

1. **Initialization:** Initialize a population of *P* individuals (genotypes) at random.
2. **Fitness assignment:** Evaluate the individuals in the current population (calculate their fitness values) using the fitness function.
3. **Selection:** Create a (temporary) mating pool of "parents" by selecting (with replacement) *P* individuals from the current population, placing a copy of each selected individual in the mating pool. The probability of an individual being selected is proportional to its fitness relative to the rest of the population (highly fit ones might get selected more than once, less fit ones less often or not at all).
4. **Reproduction:** While there are still parents in the mating pool, do:
    i. Take the next two parents from the mating pool.
    ii. Perform **crossover** on these two parents, creating two "children".
    iii. With a small probability, apply **mutation** to the children.
    iv. Place the children in the new population.

5. **Replacement:** Replace the old population with the new population.
6. **Repeat:** Go back to step 2, unless some stopping criterion is reached (e.g., a certain number of generations or a certain level of fitness).
7. **Stop:** Return the best individual in the final population.

During the initialization step, individuals are created completely at random. For example, when the chosen representation is bit strings, P (the population size) random bit strings are constructed by simply choosing for each bit position in each individual either a 0 or a 1 with equal probability. Random permutations can be created by filling an array with N random numbers chosen uniformly between 0 and 1, and assigning values from 1 to N to each position in the array according to the rank of its random number. In other words, the position with the largest (random) number gets value 1, the second largest number gets 2, and so on, until the position with the smallest number, which gets value N.

Next, the individuals in the population are evaluated, and their fitness values are calculated. Based on these fitness values, individuals are now selected to contribute their genetic information to the next generation. The idea is that individuals with a high fitness, compared to the population as a whole, will have a high chance of being selected, and individuals with a low fitness value will have a low chance. One way of implementing this is using *roulette wheel selection*. In this selection method, each individual will have a chance $f_i / \sum_i f_i$ of being selected, where f_i is the fitness of individual i. One can imagine this as dividing a roulette wheel into P slices, where the size of each slice i is proportional to the fitness of individual i relative to the population total. Now a roulette ball is rolled P times, and each time the individual corresponding to the slice in which the ball ends up is copied to the (temporary) mating pool. Obviously, individuals with larger slices (high fitness values), have a higher chance of having the ball end up in their slice, i.e., of being selected. Consequently, individuals with above average fitness values (compared to the current population average) will most likely end up with multiple copies in the mating pool, and individuals with below average fitness values only with a few or none, in proportion (statistically) to their fitness values relative to the population total. So, selection, or "survival of the fittest", is implemented based on how "good" (fit) the current solutions in the population are as compared to each other. Of course there are other ways of implementing the selection method, such as tournament selection, where each time two individuals are picked from the population (with equal probabilities), and the most fit of the two is copied into the mating pool. This is then repeated P times. Still other methods exist and have been used as well, but the main idea is that individuals are selected based on their fitness values.

Once the mating pool has been created, the individuals in this pool are allowed to reproduce and create offspring. As long as there are still individuals in the mating pool, the next pair of individuals, or "parents", is removed from the pool, and the crossover and mutation operators are performed on them, creating two "children". These children are then placed in a new population. For bit string representations, the crossover operator works as follows. Consider two parents as in the example below (the first parent is indicated in bold, to show the result of the crossover operator more clearly). First, a crossover point (the vertical bar) is randomly chosen somewhere between the first and

last bit positions, in this case at the 4th position. Next, the first part of the first parent (up to the crossover point) is combined with the last part of the second parent (after the crossover point) to create the first child, and vice versa for the second child.

0010|**111001** → **0010**|001110
1001|001110 → 1001|**111001**

Often, crossover is only applied with a certain probability p_c, which is usually set in the range [0.6;1.0]. In this case, for each pair of parents, a random number between 0 and 1 is drawn first. If this number is smaller than the crossover probability, then a random crossover point is chosen and crossover is applied as explained. Otherwise, the two children will be identical to the parents (i.e., no crossover is applied). Of course there are many variations on this basic crossover operator (known as one-point crossover). We can also use two or more crossover points, or even decide (randomly) for each individual bit in the children from which parent it will come. And, obviously, for different kinds of representations (such as permutations), different crossover operators are required to ensure that the resulting children are still valid representations for candidate solutions (for example, applying one-point crossover to two permutations does generally not result in two new valid permutations, but other crossover methods are available for permutations).

Next, with some probability p_m mutation is applied to the children, where a bit is flipped (i.e., a 0 changed into a 1 or vice versa). However, this is usually only done with a very low probability, so that (on average) only a few children actually undergo a mutation in each generation. This operator is mostly performed to prevent the population from converging too quickly (i.e., to maintain some diversity), and also to introduce some new variation, if possible. For example, if all the individuals in the population have a 1 at a certain bit position, then by using crossover alone it is impossible to get a 0 at that position. However, with mutation, such a change is possible. Here, too, different representations will require different types of mutation operators.

This reproduction process (crossover and mutation) is then repeated for the next two individuals (parents) in the mating pool, and then the next pair, and so on, until P new individuals (children) have been created. This new population then replaces the previous one, thus forming the next generation. Now, the fitness values of the new individuals can be calculated again, and the process of selection and reproduction is repeated to create subsequent generations. This way, the algorithm searches through the space of candidate solutions (by "manipulating" genotypes), trying to create better and better ones based on the current best individuals in the population and recombining (mixing) their genetic information. The algorithm usually stops when a given number of generations or some time limit is reached, or when no significant improvements are found anymore after some amount of time or number of generations.

In short, GAs are "just another" class of heuristic search algorithms that can be applied to problems for which there are no efficient algorithms available, and for which the search space is too large to perform an exhaustive search. However, in practice it turns out that they perform quite well, often finding better solutions than other search algorithms (such

as, for example, simulated annealing or hill climbing) or greedy methods. Their success is often ascribed to the fact that they use a population of candidate solutions (instead of looking at one candidate solution at a time), and that they combine global search (crossover) with local search (mutation). Thus, GAs provide a class of widely applicable, yet powerful search algorithms that are inspired by natural evolution, and form an important part of what is known more generally as evolutionary computation.

Advantages and Disadvantages of GAs

Even though GAs seem to work quite well in practice, there are several disadvantages attached to them as well. In particular, since they are fairly general search algorithms, it can be difficult to "fine tune" the algorithm to work well on any specific problem. The more complicated a problem is, the more work needs to be done to make the GA perform well. As already mentioned earlier, the choice of representation and the construction of the fitness function depend highly on the problem at hand. A bad choice for any of these can make the algorithm perform poorly. Unfortunately, it is still not known very well how to make the best choices, or how to construct the best fitness function given a problem. Most of this relies on trial and error, and often much thinking and testing needs to be done before the algorithm performs reasonably well. Furthermore, there are many different operators to choose from (different selection, crossover, and mutation methods), and several parameters to set (population size, crossover and mutation rates, etc.). Again, it is not always obvious what the best choices are, and here, too, trial and error are often the only option. Finally, there is no convergence guarantee. Whatever the best solution is that the GA finds, it is generally unknown how close this "best" solution is to the global optimum (if we knew what the global optimum is, we would not need a GA!). So, it is advisable to run the GA several times, each time starting with a different (random) initial population, and perhaps using different parameter settings. The best solution over all these runs is then taken as "the solution". If this is better than all other solutions known so far, then at least we have achieved something…

Below is a brief summary of the advantages and disadvantages of GAs:

Advantages

- GAs are fairly easy to understand and implement.
- GAs are applicable to a wide range of problems.
- GAs are easy to parallelize (e.g., simultaneous calculation of fitness values).

Disadvantages

- Choosing a good representation and constructing a good fitness function depend on the problem at hand, and can be difficult.
- There are many operators (selection, crossover, mutation) to choose from, and several parameters to set, and it is often not clear what the best choices are.
- There is no convergence guarantee. Just run the GA many times and hope for the best!

Applications

To give an idea of the kinds of problems that GAs have been applied to successfully, here is an overview of some applications (not meant to be exhaustive though!):

- Aircraft wing design (optimizing parameters that determine the shape of a wing)
- Optimal pipeline flow (optimizing, e.g., the flow of oil through a network of pipelines to satisfy customer demand)
- Scheduling problems (job-shop scheduling, etc.)
- Routing problems (TSP, network routing)
- VLSI technology, chip design (minimizing connections and communications)
- Telecommunications networks (optimal location of access points, etc.)
- Coding methods for data transmission (optimizing coding methods and their parameters)
- Robotics (evolution of robot controllers)
- Stock market prediction

(making money with money…and evolution!)

- …

And, of course, there are many, many more applications. It is impossible to list all existing GA applications here. However, the next section gives some pointers to where more information on genetic algorithms and evolutionary computation can be found.

More Information

The original book on genetic algorithms by Holland [1] is fairly technical and mathematical, and is perhaps not the best introduction to GAs. However, his later article in *Scientific American* [2] is highly recommended, as it is written for a general audience and introduces GAs at a high level that is very readable. Furthermore, two standard textbooks with good introductions but also with much detail and many examples are [3,4]. In [5], a good overview of the different GA methods (such as selection, crossover, and mutation) and their implementations can be found. As mentioned above, it is impossible to list all the different problems that GAs have been applied to. However, the various proceedings of the international conferences on genetic algorithms provide one of the most complete collections of GA applications. And of course the internet is a rich source of information. One good starting point is the GA Archives, maintained by the US navy: `www.aic.nrl.navy.mil/galist`.

Information on evolutionary computation in general is also widespread. An excellent source is the *Evolutionary Computation* journal, published by the MIT Press, or equivalently, the *IEEE Transactions on Evolutionary Computation*. Another useful source is [6], and many application areas can be found in the proceedings of the international conferences on evolutionary computation. On the internet, the *Hitch-Hiker's Guide to Evolutionary Computation* is a wonderful starting point, with many links to other pages: `www.cs.bham.ac.uk/Mirrors/ftp.de.uu.net/EC/clife/www`.

References

[1] J. H. Holland, "*Adaptation in Natural and Artificial Systems*", University of Michigan Press, 1975.

[2] J. H. Holland, "Genetic Algorithms", *Scientific American,* vol. 267, no. 1, pp 66-72, 1992.

[3] D. E. Goldberg, "*Genetic Algorithms in Search, Optimization, and Machine Learning*", Addison-Wesley, 1989.

[4] M. Mitchell, "*Introduction to Genetic Algorithms*", MIT Press, 1996.

[5] L. Davis (editor), "*Handbook of Genetic Algorithms*", Van Nostrand-Reinhold, 1991.

[6] T. Bäck, D. B. Fogel, and Z. Michalewicz (editors), "*Handbook of Evolutionary Computation*", Oxford University Press, 1997.