Genetic Algorithms and Evolutionary Programming

Ross Flieger-Allison

4 December 2015

Summary

This paper covers the intersection of biological evolution and computational optimization. Genetic algorithms and evolutionary programming are two types of approaches to solving computational optimization problems using the principles of natural selection (selection, mutation, and reproduction). In this paper I will discuss the foundation of this unique field of research, both these algorithms in detail, and consider the effectiveness of evolution as a viable problem-solving tactic and the direction I see this field going in. I ultimately conclude that these evolutionary algorithms are best suited for approximation of global optima and exploration of a diverse solution set.

Introduction

The Theory of Evolution states that all life on Earth descended from a single common ancestor. The birds and the trees, the ants and the bees — all related to each other, to one extent or another. The principle concept behind this is a purely naturalistic "descent with modification", where complex creatures evolve from more simplistic ancestors naturally over time [1]. To a biologist, this means that random genetic mutations occur within an organism's genetic code and the beneficial mutations are preserved across generations (as they aid in the organism's survival). Once enough mutations build up, entire new species can be created and, since the mutative process is inherently stochastic (random), many different species can be developed to accomplish the same overarching goal: survival. This is why biological ecosystems today are so incredibly diverse. But naturally, only the "fittest" organisms in an ecosystem will survive in order to reproduce. The less fit organisms die out over generations, while the more fit thrive and continue populating their species. When more fit organisms reproduce with *other* more fit organisms, their offspring receive the genetic fusion of their largely beneficial genotypes. This recombination and mutation of previously "good" genes leads to even *better* genes, and therefore better traits (phenotypes), in the offspring! As this process iterates over several generations, organisms that do survive produce better and better offspring until they reach a local or global

optimum within their environment. This evolutionary mechanism, and the foundation of Darwin's theory, is called *natural selection*.

In 1950, Alan Turing, the father and pioneer of computational science, proposed a "learning machine" that could leverage this principle of natural selection. The idea behind it was the following: given a problem space and a randomly generated set of machines, have each machine attempt the problem with a measurable level of success. Once every machine has attempted the problem, select out only the most successful machines and have them "reproduce" (via mutation and recombination of their binary code). This will produce new machines that are (on average) more capable of solving the problem than previous ones. Then, by repeating this process, one machine will eventually be capable of solving the problem. By following this fairly straightforward algorithm, Turing claimed, the machines can essentially *teach* themselves how to solve complex problems. Hence the ambitious title, "learning machines" [6].

After this paper was released in 1950, an entire new field of science was developed called *evolutionary computation*. This field was entirely defined by the algorithms it employed to solve problems, aptly named *evolutionary algorithms* (EA). What an evolutionary algorithm does is conceptually very simple: given a computational problem, try to find an optimal solution using the principles of evolution, namely *selection*, *mutation*, and *reproduction* [4].

To start, every EA requires an initial *population* to evolve. That population must consist of various *candidate solutions* to the problem at hand. The population size always depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions. In most problems, this set of starting solutions is randomly generated, allowing the entire range of possible solutions to be considered (the *search space*). Sometimes, however, solutions must be "seeded" in areas where optimal solutions are likely to be found. This initial stage of building a population is called *initialization*.

Next, every EA requires a concept of *fitness*, which is a measurable value representing the quality of a candidate solution. The fitness value should be computable via a predefined *fitness function*. This function is also totally dependent on the problem space. In some problems, the fitness may be computed as a simple sum of weighted values, while in others, it might be quantified as a candidate solution's ability to follow a red dot on a screen. This stage of fitness calculation is called *evaluation*.

Once we have a fitness metric defined for our algorithm and our candidate solutions evaluated, we can rank and select out particularly fit solutions and use them to repopulate our search space in preparation for the next generation. This can be done in a number of different ways, the most common way being to directly correlate the probability of selection with a candidate's fitness value, so that fitter candidates have a higher probability of reproducing while unfit candidates are more likely to die out. This makes sense in nature since fitter organisms are more likely to survive and reproduce than unfit organisms, but no organism, however fit it may be, is ever *guaranteed* survival. However, many people searching for optimal solutions to their problems don't care about biological fidelity, so another popular selection method simply sorts the candidates by fitness and selects an arbitrary top-percentage (maybe top 50% or 10%) for reproduction. This strategy ensures that all sufficiently-fit candidates are preserved and none are lost to chance. Additionally,

note that the candidate solutions that breed new generations often breed the *entire* new population of solutions, but many algorithms alternatively use *elitist selection*, which allows top candidates to live on to the next generation with their "children". This strategy is often used to ensure that the solution quality obtained by the algorithm does not decrease from one generation to the next [4]. This stage is called *selection*.

Next, our fittest candidates undergo a "genetic mutation" where random pieces of their "genomes" are modified slightly with a given *mutation operator* and *mutation probability* or *mutation severity*. Again, these parameters are entirely problem-dependent, as different solution values should be mutated differently. If the solution representation is a binary string, for example, the only possible mutation would be flipping 0s and 1s. If the values are integers or float-values, however, the mutation operator might be a scalar multiplication, the addition/subtraction of a small Δ-value, or any number of other operations. In problems where solutions are finite, *mutation probability* is used to determine the probability that any given fit candidate undergoes a mutation before reproduction. This means that (usually) not all candidates are mutated. This is desirable, because excessive mutation would cause the search to devolve into a primitive random search, so in general, it's best to keep the mutation probability fairly low in order to ensure a "steady climb" towards an optimal solution. However, when problems have real-valued solutions, *mutation severity* is used instead. In this case, all solutions are mutated but with a normally-distributed severity distribution with expected value 0. This maintains the property that small mutations are more probable than large ones but now allows solutions to consist of real values. This segment of the algorithm is appropriately named the *mutation* stage.

The last stage of the algorithm, the *reproduction* stage, is the recursive step that allows the average quality of the search space to increase over time. As fit parents breed, they have a small probability of producing children even fitter than themselves, which in turn live on to breed and produce even *more* fit solutions. This generational process is repeated until a termination condition is reached. The most common termination conditions are the following: a solution is found that satisfies minimum criteria, a fixed number of generations is reached, an allocated "budget" (computation time) is reached, the highest ranking solution's fitness has reached a plateau such that successive iterations no longer produce better results, some form of manual inspection, or any combination of the above [4]. Once the algorithm terminates, the best candidate solution currently available in the population is returned to the user.

To summarize, here is a rough pseudo-code implementation of the *majority* of evolutionary algorithms:

1. Randomly generate an initial population of candidate solutions (first generation).
2. While the termination condition is not satisfied:
    (a) Evaluate the fitness of each candidate solution in the population.
    (b) Select the most-fit candidates for reproduction (parents).
    (c) Breed new candidate solutions via mutation (and optionally recombination) operations (children).
    (d) Replace either the least-fit population or entire population with child solutions.

3. Return: the most fit solution in the current generation.

The development of evolutionary algorithms to solve different optimization problems throughout the 20^{th} and 21^{th} centuries has produced numerous different search heuristics and subtypes of EA, each designed to solve problems with slightly different constraints. The two that I'd like to discuss here are *genetic algorithms* (GA), the most popular form of EA, and *evolutionary programming* (EP). Both approaches have their pros and cons, and each can be used to solve a number of different real-world optimization problems.

Genetic Algorithms

In nature, all organisms are entirely encoded by strands of deoxyribonucleic acid (DNA) in their cells' chromosomes. That DNA, in an overly-simplified world, merely consists of the four nucleotide base pairs cytosine (C), guanine (G), adenine (A), and thymine (T). Thus, any organism can be represented as a string (or *array*) of base-pair values. Those base pairs can in turn be encoded into binary: C $\leftrightarrow$ 00, G $\leftrightarrow$ 01, A $\leftrightarrow$ 10, T $\leftrightarrow$ 11. Therefore, any organism's genotype can be encoded as a simple array of binary digits.

In a genetic algorithm, the same concept applies to encoding solutions in a problem space. All GAs use chromosome-like fixed-length linear data structures to encode their candidate solutions as strings of *bits* (1s and 0s). What this allows the algorithm to do is utilize a new genetic operator, after the traditional mutation stage, called *crossover*.

During the crossover stage, pairs of (now mutated) fit candidates replicate a biological process during reproduction called "chromosomal crossover" where the "mother" and "father" chromosomes of an solution are spliced and segments of their genetic material are randomly swapped, creating a new genotype with some segments from the mother and some from the father. As with mutation, there are many possible techniques for performing crossover, the most common being one-point crossover and two-point crossover. In one-point crossover, a single crossover point on both parents' bit strings (chromosomes) is selected. All data beyond that point on either string is swapped between the two parents and the resulting pair of strings are encodings for two child solutions. Two-point crossover calls for two points to be selected on the parents' bit strings, where everything between the two points is swapped and the result is two children [7].

This extra stage in genetic algorithms allows parent solutions in a population to breed child solutions such that their children might possess the so called "best of both worlds". The idea behind this being that mutation *in combination* with crossover allows problem domains in a "complex fitness landscape" to move the population away from the local optima that a traditional hill-climbing algorithm might get stuck on. However, it has also been argued that crossover is simply a generalization of several mutations performed at once and therefore "provides no significant benefit" [5]. Much of this is still up for debate, but no one has argued yet that crossover *hinders* the optimization process in any way, so GAs still include it for biological fidelity (genetic algorithms are given their name because of this emulation of genetic operators observed in nature).

The list of computational optimization problems that have been solved by genetic algorithms is extensive. Probably the simplest and easiest problem to explain is the Knapsack Problem, where you are given a finite set of weighted valued objects and a "knapsack"

(or bag) that has a predefined weight threshold. The goal of the problem is to find the combination of objects that maximizes the total value in the bag while still staying below the weight limit. To solve this with a genetic algorithm, we would first encode the list of possible objects in the knapsack as a bit string where each bit represents the presence of an object in the solution (1 meaning included, 0 meaning excluded). The fitness function for the problem would then just be the sum of the values of the objects included in the solution (i.e., having a 1 at their predefined index in the solution string). Once we have this representation, the algorithm does the rest (using the steps explained above: initialization, evaluation, selection, mutation, *crossover*, and reproduction) [8].

Other problems particularly appropriate for genetic algorithms include timetabling and scheduling problems. I won't go into much detail here, but a common gene encoding for a problem like this would be an array of start and end times for a sequence of tasks. The goal is to maximize the number of completed tasks in a minimum amount of time with a minimum number of defects such that all the contraints of the system are met (in production systems such as factories, many constraints are often involved). GAs are well suited for solving problems like these because, unlike heuristic methods, genetic algorithms operate on a population of solutions rather than a single solution, allowing the solutions to converge to a small number of optimal results.

As a general rule of thumb, if a problem has solutions that can be encoded as fixed-length binary strings and the quality of those solutions can be evaluated reasonably efficiently, the problem can be solved by a genetic algorithm. It might not be the best approach, depending on the landscape of possible solutions, but it's probably worth a shot, at least for approximation.

Evolutionary Programming

EP, another subset of evolutionary algorithms, differs slightly from genetic algorithms because it places extra emphasis on the *behavioral* linkage between parents and their children, rather than seeking to emulate specific genetic operators observed in nature. EP is known as a *phenotypic* algorithm while GA is a *genotypic* algorithm. Phenotypic algorithms operate directly on "the parameters of the system itself", whereas genotypic algorithms operate on strings representing the system (genes). The analogy in biology to phenotypic algorithms is a direct change in an organism's behavior or body and the analogy to genotypic is a change in the organism's genetic makeup, which lies behind the behavior or body [2].

That means that there is no constraint on the representation of solutions using EP. While in GAs, solutions must be representable as fixed-length binary vectors, in EP, the representation follows entirely from the problem. A neural network, for example, can be represented in the same manner as it is implemented because the algorithm (namely, the mutation operation) does not require a linear encoding. This means that, when implemented to solve real-valued function optimization problems, EP is able to operate directly on the real values themselves rather than any encoding of the values.

Additionally, EP does not involve any genetic crossover, because EP is an abstraction of evolution at the level of reproductive populations (i.e., *species*) and thus no recombination mechanisms are typically used because recombination does not occur between species (by

definition). This means that mutation is the only operator, so it therefore assumes a new level of importance (in GA, mutation is really a background operator designed to get the solution out of local minima and introduce new solutions not initially present, rather than to comprehensively explore a search space).

During mutation in EP, we take *all* the system parameters (an example of this would be edge weights, if we're training a neural network) and change them by a random amount according to a normally-distributed severity distribution, which I briefly described earlier, that weights minor mutations as highly probable and substantial mutations as increasingly unlikely. This means the random number generator will produce mostly small numbers (which are more likely to produce a good result — the "gradual climb" technique) with just the occasional large number to perturb the system and stop it from settling in a local minimum. The reason that *all* the parameters are changed is that, in biology, changing just a single gene often changes the behavior of the whole organism, which we don't want here because this is a *phenotypic* algorithm [2]. Furthermore, the severity of mutation (or the spread of the mutation distribution) is often reduced as the global optimum is approached, because only fine adjustments to the system parameters become required to bring the solution to its final value. But how can the severity be dampened as the population approaches the global optimum if we don't know what the global optimum is in the first place? Various techniques have been proposed and implemented which address this challenge, the most widely studied technique being the "Meta-Evolutionary" technique in which (roughly speaking) the variance of the mutation distribution is subject to change by a variance mutation operator and evolves over time, along with the solutions of the population (that operator is very often some inverse of the average fitness of the population — as solution quality increases, variance in the mutation distribution should decrease) [3].

Another distinction EP has is in its selection stage. When selecting the best solution, EP often uses *tournament* selection where one solution is played off against a preselected number of opponents and receives a "win" if it is at least as good as its opponent in each encounter. Selection then eliminates those solutions with the least wins and carries the rest through to the next generation. This is different than the deterministic GA selection methods where the worst solutions are simply purged from the population based purely on fitness function evaluation.

As with GAs, the applications of EP are numerous. The most popular application of EP, however, is the optimization of continuous functions that are non-convex and non-differentiable. This is a natural application because EP is built upon the mutation of system parameters, where in this case, the system is the real-valued function and the solutions are the different points on its complex solution surface.

In the next section, I will delve more deeply into the benefits and disadvantages of each of these EA specializations (GA and EP) and discuss the limitations of evolutionary algorithms as a whole. Additionally, I'll introduce some new algorithmic variants that have come into play in the past couple years and ultimately give my two cents on the future of EA as a means of optimization.

Discussion

Let's start by debating GAs versus EP while the two concepts are still fresh in our minds. Do you think one approach might be better than the other?

Here are few ways in which GAs might be considered superior:

1. **Conceptually non-mathematical.** A major benefit of GA over EP is that it's very easy to describe the algorithm in an easy-to-understand and non-mathematical way. This in turn has led to many books which take a very practical look at GAs and are useful for the practitioner who simply wants to get a system running without getting bogged down in theory. On the other hand, the literature on EP tends to be littered with obscure math and Greek symbols and no really good practical guide exists to designing EP solutions.

2. **Speed of convergence.** One benefit of crossover, and therefore GAs over EP, is speed. As we said before, crossover is essentially many simultaneous mutations. Pure mutation, while still effective, is just a slow way to search for good solutions.

3. **Established community and extensive literature.** One thing that is often overlooked when looking for practical solutions to problems is the size of the community established around known techniques. In the age of the internet, almost everything that has been done can be made available over the internet and since GAs are historically very popular, a lot of research has been done, a lot of problems have been solved, and a lot of literature has been written, all of which is all at your disposal when addressing your own unique problems (maybe someone has already solved your particular problem and has done most of the work for you!).

4. **Biological fidelity.** This might not be viewed as an advantage to some, but while our understanding of complex biological systems remains limited, the processes involved in biology that we *do* understand maintain a massive amount of credibility, due to the incredible results that biological algorithms have produced in nature throughout Earth's history. Therefore, it's easy to believe that an algorithm like GA that emulates such a historically successful formula is likely also on a track to success.

Now, consider the ways in which EP might be considered superior:

1. **No encodings needed.** EP uses a simple and direct method of representing system parameters. The real values of the variables can be used in the algorithm (for example, weight values, as shown earlier). There can also be several different types of variables composing the parameters of a solution (for example, weight values and number of neurons in the network). This is compared to GAs, where everything in a solution needs to be encoded into a single string of binary digits, which might be complicated to achieve and time-consuming to decode.

2. **Variable-length solutions.** In some variants, solution sizes can also increase and decrease unlike the standard GA. This means that we can add neurons or layers onto a neural network without too much problem (this may be useful in the definition of network or circuit structures). In the standard GA, on the other hand, everything must be fixed-length to support efficient crossover operations.

3. **More flexibility.** In general, we can consider EP algorithms more flexible than GAs, based on some of the points I've already made.

So, while EP algorithms tend to be more flexible problem-solvers, GAs have a large established following and more intuitive theory behind them. But for me personally, while this discussion has its place in understanding the advantages of certain algorithmic approaches over others, this is ultimately a moot debate because the two algorithms do totally different things in order to address totally different problems. EP will always be phenotypic while GAs are genotypic and their fundamental properties differ because of the unique problem spaces they're designed to confront.

Instead, let's step back and analyze evolutionary optimization as a concept. Here are some of the advantages we've encountered so far:

1. **Intuitive and easy to understand.** Pretty much anyone with a high school education who hasn't been shielded from the concept of evolution understands how organisms in nature have evolved. Therefore, since the concepts of evolution are so tightly coupled to evolutionary algorithms (naturally), learning and implementing one is fairly straightforward, even for a newcomer.

2. **Good approximations with decent efficiency.** Evolutionary algorithms often perform well approximating solutions because they can converge to an optimum (possibly only local) within a matter of a couple generations if the algorithm's parameters are set properly.

3. **Great for problems with rich fitness landscapes.** Since randomization is so firmly engrained in the different stages of evolutionary algorithms (initialization, mutation, recombination), the range of resulting solutions tends to be very diverse, allowing EAs to navigate very complex search spaces and consider a variety of possible optima.

Now, here are the disadvantages:

1. **Global optima aren't guarenteed.** In some situations, EAs may have a tendency to get "stuck" on local optima or even arbitrary points rather than the global optimum as a result of the problem's "hilly" fitness landscape.

2. **Hard to recognize a global optimum vs. a local one.** The "better" solution is only in comparison to other solutions. As a result, the stop criterion is not clear in every problem and it is impossible to known if the "best solution" has been found without providing other insights into the problem.

3. **Can't be applied to all optimization problems.** EAs cannot effectively solve problems in which the only fitness measure is a single right/wrong measure (like decision problems), for example, as there is no way to converge on the solution (i.e., no hill to climb).

4. **It's not always the most effective approach.** For specific optimization problems and problem instances, other optimization algorithms may be more efficient than genetic algorithms in terms of speed of convergence.

5. **Lack of scalability.** EAs do not scale well with complexity (where the number of elements which are exposed to mutation is large there is often an exponential increase in search space size). This makes it extremely difficult to use the technique on problems such as designing an engine, a house or plane. In order to make such problems tractable to evolutionary search, they must be broken down into the simplest representation possible. Hence we typically see evolutionary algorithms encoding designs for fan blades instead of engines, building shapes instead of detailed construction plans, airfoils instead of whole aircraft designs.

6. **A complicated concept of fitness makes problems intractable.** Even though the fitness function may seem relatively harmless, it is very often the crutch that makes theoretically-solvable evolutionary problems intractable — when the fitness calculation is too complex, the function evaluation becomes the most limiting and time-consuming segment of the algorithm. Note: in order to counteract this, some genetic algorithms will only evaluate a sample of candidate solutions, leaving the other solutions to die off until they are repopulated by the selected sample in the next generation.

So, what's our verdict? Does evolution still have it's place in the realm of computational optimization? Or are man-made mathematical algorithms always going to be more effective?

In short, yes, it does have it's place. But it is certainly not the end-all be-all approach to solving optimization. In my opinion, EAs excel in two areas in particular: *approximation* within complex problem spaces and *discovery* of novel solutions where several different solutions might suffice. What EAs ultimately provide is an easy-to-understand and easy-to-implement means of optimization where not too much needs to be assumed about the problem space and iteration over just a few hundred generations can usually provide very effective approximations of global optima.

But naturally, we can't stop there. Researchers in recent years have thought up a number of different EA *variants* that have been proven to make our favorite canonical examples ("standard" GA and EP) even more effective than they already are. For example, new approaches to chromosome representation in GAs are currently being tested, including the use of real-valued numbers instead of bit strings. Additionally, reproduction using *more than two* parents during crossover in GAs has been experimented with, yielding surprisingly successful results. The most promising variant, however, (notice all these variants are for GAs because of their large following) is the idea of "adaptive" GAs where the probabilities of crossover and mutation are *adaptively adjusted* in order to maintain the population *diversity* (this is key for "climbing all hills") as well as sustain the convergence capacity. As I said before, GAs tend to be quite good at finding generally good global solutions, but very inefficient at finding the last few mutations to find the absolute optimum. This new concept of adaptive GAs (when done correctly) solves the issue in the majority of problem cases [4].

So as these algorithms evolve, just like the populations they develop (think of the algorithmic variants as "mutations"), the field of evolutionary computation will continue to improve and become more practical as computing power becomes more readily available and our understanding of biological evolution becomes more clear. I'm sure we will converge on a few "local optima" along the way, but with time, all the "hills" will be climbed

and the optimal evolutionary approach will become evident. I believe this field of computation, along with our understanding of the biology behind it, is still very much in its infancy.

References

[1] Darwin, Charles. *The Origin of Species, 1876*. Washington Square, NY: New York UP, 1988. Web.

[2] "Evolutionary Programming and Evolutionary Strategies." *Computational Intelligence* (2007): 187-211. Web.

[3] Fogel, D. B., L. J. Fogel, and J. W. Atmar. "Meta-evolutionary programming." *Conference Record of the Twenty-Fifth Asilomar Conference on Signals, Systems & Computers* (1991): n. pag. Web.

[4] "Genetic algorithm." *Wikipedia*. Wikimedia Foundation, 2 Nov. 2015. Web. 28 Nov. 2015.

[5] Senaratna, Nuwam I. "Genetic Algorithms: The Crossover-Mutation Debate." *A literature survey (CSS3137-B) submitted in partial fulfilment of the requirements for the Degree of Bachelor of Computer Science (special) of the University of Colombo*. (n.d.): n. pag. 15 Nov. 2005. Web. 29 Nov. 2015.

[6] Turing, A. M. "Computing Machinery and Intelligence." *Mind* LIX.236 (1950): 433-60. Web.

[7] Vekaria, Kanta, and Chris Clack. "Selective Crossover in Genetic Algorithms: An Empirical Study." *Lecture Notes in Computer Science Parallel Problem Solving from Nature — PPSN V* (1998): 438-47. Web.

[8] Whitley, Darrell. "A genetic algorithm tutorial." *Statistics and Computing (1994)* 4.2 (1994): n. pag. Web.