Abstract

The impact of evolutionary theory on biology cannot be underestimated. But evolutionary theory extends beyond an ordering principle of life. The fundamental processes in evolution can be simulated on a computer and used for good engineering purpose. Evolutionary computation is the field of research investigating simulations of natural evolution. This paper offers a brief introduction to three main avenues of investigation: Genetic algorithms, evolution strategies, and evolutionary programming. The methods are broadly similar, but each adopts a slightly different level of abstraction. Some simple examples of each technique are offered.

Modeling Evolutionary Optimization

Darwinian evolution provides an essential framework for biology [29]. Evolution can be viewed as an optimization process: Organisms iteratively improve their ability to predict environmental circumstances so as to survive and reproduce. The challenges that individuals and species face are typified by nonlinearities, chaos, random and temporal variation, and other difficulties, and yet natural evolution has invented "organs of extreme perfection" [4] to cope with these challenges. These same general characteristics have also posed significant challenges to conventional optimization algorithms. Capturing the essential aspects of Darwinian evolution in algorithmic form can lead to robust optimization techniques capable of addressing problems that are currently intractable.

The most widely accepted collection of evolutionary theories is the neo-Darwinian paradigm. These arguments assert that the vast majority of the history of life can be accounted for by physical processes operating on and within populations and species [23]. These processes are reproduction, mutation, competition, and selection. Reproduction is an obvious property of extant species. Mutation, in any positively entropic universe, is guaranteed in that errors during replication of genetic information will necessarily occur. Competition is a consequence of expanding populations in a finite resource space. Selection is the inevitable result of competitive replication as species fill the available space. Evolution is then an inescapable result of these basic interacting physical processes [46].

Evolved individuals and species can be viewed as a duality of their genetic program, the genotype, and their expressed behavioral traits, the phenotype (Figure 1). The genotype provides a mechanism for the storage of experiential evidence, of historically acquired information. But the resulting phenotypic expression of a genetic structure is generally unpredictable due to the universal effects of pleiotropy and polygeny. Pleiotropy is the effect that a single
gene can affect multiple phenotypic traits, and polygeny is the effect that a single phenotypic trait can be affected by many genes (Figure 2). Different genetic structures may imply equivalent behaviors and the realized phenotype also depends on the interaction between the organism and its environment.

Selection acts only on the expressed behaviors of individuals and species [29]. This is perhaps best illustrated using the idea of an adaptive topography (response surface) as proposed by Wright [47]. A population of genotypes maps to respective phenotypes which in turn mapped onto the adaptive topography. The topography indicates the fitness of the individual or population. Evolution probabilistically proceeds up the slopes of the topography, which may contain multiple optima, toward peaks as selection culls inappropriate variants. This process may be equivalently viewed from an inverted position, with selection eliminating those species with greatest predictive error relative to current environmental demands (Figure 3). Evolution then probabilistically descends slopes on the topography searching for troughs.

Viewed in this manner, evolution is an obvious optimization, problem-solving search process. Selection drives phenotypes as close to the optimum as possible, given initial conditions and environmental constraints. But the environment continually changes and species lag behind, evolving toward new optima. Ultimately, suboptimal behaviors must be expected in any dynamic environment, but selection continues to operate regardless of a population’s position on the adaptive topography.

This view of evolution leads naturally to two fundamentally different methods of simulation. The first emphasizes the genotype and the interactions that can be observed at this level. Such procedures have become known as genetic algorithms. The second emphasizes the phenotype and suggests that instead of incorporating genetic operators as observed in nature, trial solutions can be perturbed with various operators such that there is a continuous normally distributed variation in the observed behaviors of new trials (or nearly so in discrete problems). Such procedures have become known as evolutionary algorithms (although some use this term synonymously with evolutionary computation). Within evolutionary algorithms, attention can be focused on the behavior of individuals, or on the behavior of entire species. These methods are described as evolution strategies and evolutionary programming, respectively. Each of these methods is described in greater detail in the following sections.

Genetic Algorithms

Many researchers have proposed simulations of genetic systems [3, 20, 24, 36]. These genetic algorithms are typically implemented as follows:

1. A problem to be addressed is defined and captured in
an objective function that indicates the fitness of any potential solution.

2. A population of candidate solutions is initialized subject to applicable constraints. Typically, each solution is coded as a vector \( \mathbf{x} \), termed a chromosome, with elements of the vector being called genes, and alternative values at specific positions termed alleles. Often, these solutions are coded in binary, following suggestions in [24]. If the problem involves continuous-valued optimization then the degree of required precision determines the length of the binary coding.

3. Each chromosome, \( \mathbf{x}_i, i = 1, \ldots, P \), in the population is decoded into a form appropriate for evaluation and is assigned a fitness score \( F(\mathbf{x}) \) according to the objective.

4. Each chromosome is assigned a probability for reproduction, \( p_i, i = 1, \ldots, P \), so that its likelihood of being selected is proportional to its fitness relative to the other chromosomes in the population.

5. According to the assigned probabilities \( p_i, i = 1, \ldots, P \), a new population of chromosomes is generated by probabilistically selecting solutions from the current population. The selected chromosomes generate "offspring" via the use of specific genetic operators, such as crossover and bit mutation. Crossover operates on two parents and creates two new offspring by selecting a random position along the coding and splicing the section that appears before the selected position in the first solution with the section that appears after the selected position in the second solution, and vice versa. Bit mutation simply offers the possibility of flipping any bit in the coding of a new solution (Figure 4). Typical ranges for the probabilities for crossover and mutation are 0.6-0.95 and 0.001-0.01, respectively [38].

6. The process is halted if a suitable solution has been found, or if the available computing time has expired, otherwise the process continues to step 3, where the new chromosomes are scored and the cycle is repeated.

This general procedure has been used to address many difficult combinatorial optimization problems, including the traveling salesman problem [45], protein secondary and tertiary structure prediction [28], automatic programming [26], and others [19].

There are many issues that must be addressed using a genetic algorithm. For example, the necessity for binary coding has received considerable attention and many researchers have used other representations with success. Selection in proportion to fitness can be problematic for negative objective values, but methods for rescaling the data have been suggested (e.g., fitness based on ranking). The repeated application of crossover and selection tends to homogenize the population, leading potentially to convergence at suboptimal solutions (i.e., premature convergence). Specific tricks have been invented to maintain diversity such as only accepting new offspring into a population if they are structurally significantly different from their parents [6]. While the application of genetic algorithms to optimization problems may not be straight-forward, the method has demonstrated potential.

**Evolutionary Algorithms**

An alternative method for simulating evolution was independently adopted by Rechenberg and Schwefel [33, 39] collaborating in Germany, and L. Fogel [15] in the United States. These models, commonly described as evolution strategies or evolutionary programming, or more broadly as evolutionary algorithms [8, 32], emphasize the behavioral link between parents and offspring, or between reproductive populations, rather than the genetic link. When applied to real-valued function optimization, the most simple method is implemented as follows:

1. The problem is defined as finding the real-valued \( n \)-dimensional vector \( \mathbf{x} \) that is associated with the extremum of a function \( F(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R} \). Without loss of generality, let the procedure be implemented as a minimization process.

2. An initial population of parent vectors, \( \mathbf{x}_i, i = 1, \ldots, P \), is selected at random from a feasible range in each dimension. The distribution of initial trials is typically uniform.

3. An offspring vector, \( \mathbf{x}'_i, i = 1, \ldots, P \), is created from each parent \( \mathbf{x} \) by adding a zero mean Gaussian random variable with preselected standard deviation to each component.

This general procedure has been used to address many difficult combinatorial optimization problems, including the traveling salesman problem [45], protein secondary and tertiary structure prediction [28], automatic programming [26], and others [19].

<table>
<thead>
<tr>
<th>Crossover Point</th>
<th>Mutation Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent 1: 1 1 1 1 1 1 1</td>
<td>Parent: 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>Parent 2: 0 0 0 0 0 0 0</td>
<td>X</td>
</tr>
<tr>
<td>Offspring 1: 1 1 1 1 0 0 0</td>
<td>Offspring: 1 1 1 1 0 1 1</td>
</tr>
<tr>
<td>Offspring 2: 0 0 0 0 0 1 1</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.** In genetic algorithms, crossover exchanges segments of two strings, while bit mutation alters a single component of one string.
of $x$.

4. Selection then determines which of these vectors to maintain by comparing the errors $F(x_i)$ and $F(x_i')$, $i = 1, \ldots, P$. The $P$ vectors that possess the least error become the new parents for the next generation.

5. The process of generating new trials and selecting those with least error continues until a sufficient solution is reached or the available computation is exhausted.

In this model, each component of a trial solution is viewed as a behavioral trait, either of an individual or species, not as a gene. A genetic source for these phenotypic traits is presumed but the nature of the linkage is not detailed. It is assumed that whatever genetic transformations occur, the resulting change in each behavior will follow a Gaussian distribution with zero mean difference and some standard deviation. Specific genetic alterations can affect many phenotypic characteristics due to pleiotropy and polygeny. It is therefore appropriate to simultaneously vary all of the components of a parent in the creation of a new offspring.

The original efforts in evolution strategies examined the main variations of evolutionary computation. Each technique is applied to a very simple problem. Extensions of these applications are discussed below.

Operations have also been added to adapt the distribution of normal variation around each parent [40]. These self-adaptive mechanisms are associated with each parent, so that each solution carries parameter values to be optimized, but also incorporates information on how to change itself when creating new offspring. This mechanism allows for trials to be adaptively aligned with arbitrary contours on a response surface, rather than being forced to search mainly along coordinate axes (Figure 5). Extensions have also been made to include recombinatory operators on these self-adaptive parameters. This is reasonable in evolution strategies because the solutions are abstracted individuals. Under evolutionary programming, however, no recombination operators are applied because the solutions are abstracted species and no sexual recombination occurs by definition.

The original evolutionary programming approach was similar to that of evolution strategies, but involved a more complex problem: Creating artificial intelligence. Fogel [16, 17] proposed that intelligent behavior requires the composite ability to predict one’s environment coupled with a translation of the predictions into a suitable response in light of a given goal. Environments were simulated as growing sequences from an alphabet of finite symbols and finite state machines were evolved to predict these sequences.

The prediction problem is a sequence of static optimization problems in which the adaptive topography (fitness function) is time-varying. The process can be easily extended to abstract situations in which the payoffs for individual behaviors depend not only on an extrinsic payoff function, but also on the behavior of other individuals in the population (e.g., the iterated prisoner’s dilemma) [9].

Evolutionary programming has also been applied to real-valued continuous optimization problems, such as training neural networks [1, 30], and automatic control [10, 43], and is virtually equivalent in many cases to the procedures of evolution strategies, despite their independent development. The extension to using self-adaptive mechanisms for creating offspring was offered in [13, 14], although with slightly different procedures. Comparisons between the various self-adaptive methods is an open area of research; initial comparisons [37] appear to favor the use of the methods proposed in [40]. Further investigations in evolution strategies and evolutionary programming can be found in [5, 42].

**Examples**

The following examples illustrate the basic operations of the main variations of evolutionary computation. Each technique is applied to a very simple problem. Extensions of these applications are discussed below.

**Genetic Algorithms**

Consider the problem of finding the string of 100 bits
{0,1} such that the sum of the bits in the string is maximized. The objective function may be written as

$$F(x) = \sum x_i, \quad i = 1, \ldots, 100.$$ 

An initial population of binary strings is taken at random with each bit having an equal probability of taking the value 1 or 0. Let the population size be 100 parent strings. Each string is evaluated in light of $F(x)$ and receives a fitness score, which in this case is simply the number of 1's in the string. A new population is created by probabilistically selecting strings from the initial population in proportion to their relative fitness. Thus a string with a fitness of 60 would be twice as likely to be selected as a string with a fitness of 30. The new population is created from the selected parents using the genetic operators of crossover and mutation. Let the crossover rate be 0.8 and the bit mutation rate be 0.01. The process of evaluating the new offspring and selecting new parents is then continued. Figure 6 indicates the rate of optimization of the best score in the population and the mean of all parents' scores at each iteration. The genetic algorithm quickly converges on strings with all components set to 1.

**Evolution Strategies**

Consider the problem of finding the minimum of the quadratic function of three variables:

$$F(x) = \sum x_i^2, \quad i = 1, 2, 3.$$ 

Rather than use any binary transformation on the evolving solutions, the real values are manipulated directly. Let the initial population consist of 30 parents, with each component varying in accordance with a uniform distribution over $[-5.12, 5.12]$ (after [6]). Let one offspring be created from each parent by adding a Gaussian random variable with mean zero and variance equal to the error score of the parent divided by the square of the number of dimensions to each component (Rechenberg [34] indicated that this is nearly optimal for the quadratic function at hand). Let selection simply retain the best 30 vectors in the population of parents and offspring. Figure 7 indicates the rate of optimization of the best vector in the population as a function of the number of generations. The process rapidly converges close to the unique global optimum.

**Evolutionary programming**

For ease of description, attention is given to a problem considered in Fogel et al. [17]. Consider the use of finite state machines (Figure 8) to represent the logic underlying a sequence of observed symbols. Let the environment be the nonstationary sequence generated by classifying each of the increasing integers as being prime (symbol 1) or nonprime (symbol 0). Thus the environment consists of the sequence 01101010001,.., where each symbol depicts the primeness of the positive integers 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ..., respectively. Let the payoff function be all-or-none, that is, given a sequence of known symbols, one point is awarded if the finite state machine predicts the next symbol in the sequence correctly and no points are awarded if the prediction is incorrect. Further, let a penalty for complexity of 0.01 multiplied by the number of states of the machine be imposed on the machine's score, so as to maintain parsimonious machines.

Fogel et al. [17] used five finite state machines as the initial population. The initial machines possessed five states. Each machine generated a single offspring through simple mutation, with equal probabilities of either (1) adding a state, (2) deleting a state, (3) changing the start state, (4) changing an output symbol, or (5) changing a next-state transition. The parents and offspring were evaluated in light
of how well they fit the observed sequence of 1’s and 0’s and 10 iterations of mutation and selection were conducted before the best finite state machine in the population was used to predict the next, as yet unseen, symbol. The accuracy of the prediction was recorded and then the new symbol was added to the observed sequence and the process iterated.

Figure 9 shows the cumulative percentage of correct predictions in the first 200 symbols. After the initial fluctuation (due to the small sample size), the prediction score increased to 78 percent at the 115th symbol and then essentially remained constant until the 200th prediction. At this point, the best machine possessed four states. At the 201st prediction, the best machine possessed three states, and at the 202nd prediction, the best machine possessed only one state with both output symbols being 0. After 719 symbols, the process was halted with the cumulative percentage of correct predictions reaching 81.9 percent. The asymptotic worth of this machine would be 100 percent correct because the prime numbers become increasingly infrequent and the machine continues to predict “nonprime.” Additional experiments in Fogel et al. [17] considered forcing the payoff to reflect the rarity of the event, thereby increasing the value for correctly predicting prime numbers (see [17, 12] for details).

Extensions

Each of these techniques has been extended beyond its original formulation. For example, as genetic algorithms often prematurely converge to suboptimal solutions and the provided precision is restricted by the length of the coding, Schraudolph and Belew [38] invented an essentially iterative method of rescaling the available range using dynamic parameter encoding. When the population is believed to have converged to a collection of similar solutions, the minimum and maximum values in each dimension are recalculated based on the converged population and the search is reinitialized within that new range. Thus the eventual precision of the algorithm is not dependent on the initial coding length.

Other efforts in genetic algorithms have used real-valued representations as opposed to binary coding and have indicated such efforts to be more efficient and effective at discovering improved solutions [31]. In addition, Koza [26] has used genetic algorithms to evolve S-expressions as computer programs. Recent efforts in this area of genetic programming can be found in Kinnear [25].

Further, evolution strategies and evolutionary programming have been applied to a wide variety of combinatorial and discrete optimization problems, including the design and training of neural networks [12], fuzzy control systems [22], and image processing [21]. For a more detailed introduction to these efforts, the reader is encouraged to review [2, 7, 8, 11, 18, 27, 35, 41, 44].

Discussion

Simulated evolution has a long history. Similar ideas and implementations have been invented numerous times independently. There are currently three main lines of investigation: Genetic algorithms, evolution strategies, and evolutionary programming. These methods share many similarities. Each maintains a population of trial solutions, imposes random changes to these solutions, and incorporates the use of selection to determine which solutions to maintain into future generations and which to cull from the pool of trials. But these methods also have important differences. Genetic algorithms emphasize models of genetic operators as observed in nature, such as crossing over and point muta-
tion, and apply these to abstracted chromosomes. Evolution strategies and evolutionary programming emphasize mutational transformations that maintain behavioral linkage between each parent and its offspring, respectively at the level of the individual or population. Recombination may be appropriately applied to individuals, but is not applicable for species.

No model can be a complete description of a true system. All models are incomplete. But each of the above models has been demonstrated to be a useful method for addressing difficult optimization problems. The greatest potential for the application of evolutionary computation will come from implementation on parallel machines, for evolution is an inherently parallel process. Although such efforts will undoubtedly be of practical utility, the ultimate advancement of the field will always rely on the careful observation and abstraction of the natural process of evolution.

Acknowledgments

The author would like to thank L.J. Fogel for his comments and criticisms of this paper, and Y. Attikiouzel for his encouragement and support.

References


