Genetic Algorithms: An Overview[1]

Melanie Mitchell
Santa Fe Institute
1399 Hyde Park Road
Santa Fe, NM 87501
email: mm@santafe.edu

Complexity, 1 (1) 31–39, 1995.

Abstract

Genetic algorithms (GAs) are computer programs that mimic the processes of biological evolution in order to solve problems and to model evolutionary systems. In this paper I describe the appeal of using ideas from evolution to solve computational problems, give the elements of simple GAs, survey some application areas of GAs, and give a detailed example of how a GA was used on one particularly interesting problem—automatically discovering good strategies for playing the Prisoner's Dilemma. The paper concludes with a short introduction to the theory of GAs.

Introduction

The goals of creating artificial intelligence and artificial life stem from the very beginnings of the computer age. The earliest computer scientists—Alan Turing, John Von Neumann, Norbert Weiner, and others—were motivated in large part by visions of imbuing computer programs with intelligence, with the life-like ability to self-replicate, and with the adaptive capability to learn and to control their environments. These early pioneers of computer science were as much interested in biology and psychology as in electronics, and looked to natural systems as guiding metaphors for how to achieve their visions. It should be no surprise, then, that from the earliest days, computers were applied not only to calculating missile trajectories and deciphering military codes, but also to modeling the brain, mimicking human learning, and simulating biological evolution. These biologically motivated computing activities have waxed and waned over the last forty years, but in the last ten years they have all undergone a resurgence in the computation research community. The first has grown into the field of neural networks, the second into machine learning, and the third into what is now called "evolutionary computation," of which genetic algorithms (GAs) are the most prominent example.

GAs were first described by John Holland in the 1960s and further developed by Holland and his students and colleagues at the University of Michigan in the 1960s and 1970s. Holland's goal was to understand the phenomenon of "adaptation" as it occurs in nature and to

[1]Adapted from *An Introduction to Genetic Algorithms*, Chapter 1. MIT Press, forthcoming.

develop ways in which the mechanisms of natural adaptation might be imported into computer systems. Holland's 1975 book *Adaptation in Natural and Artificial Systems* (Holland, 1975/1992) presented the GA as an abstraction of biological evolution and gave a theoretical framework for adaptation under the GA. Holland's GA is a method for moving from one population of "chromosomes" (e.g., strings of "bits" representing candidate solutions to a problem) to a new population, using "selection" together with the genetics-inspired operators of crossover, mutation, and inversion. Each chromosome consists of "genes" (e.g., bits), with each gene being an instance of a particular "allele" (e.g., 0 or 1). The selection operator chooses those chromosomes in the population that will be allowed to reproduce, with fitter chromosomes producing on average more offspring than less fit ones. Crossover exchanges subparts of two chromosomes, roughly mimicking biological recombination between two single-chromosome ("haploid") organisms; mutation randomly changes the allele values of some locations in the chromosome; and inversion reverses the order of a contiguous section of the chromosome, thus rearranging the order in which genes are arrayed.

The Appeal of Evolution

Why use evolution as an inspiration for solving computational problems? The mechanisms of evolution seem well-suited for some of the most pressing computational problems in many fields. Many computational problems involve search through a huge number of possibilities for solutions. One example is the problem of computational protein engineering, in which an algorithm is sought that will search among the vast number of possible amino-acid sequences for a protein with specified properties. Another example is searching for a set of rules that will predict the ups and downs of a financial market such as foreign currency. Such search problems can often benefit from an effective use of parallelism, in which many different possibilities are explored simultaneously in an efficient way.

Many computational problems require a computer program to be *adaptive*—to continue to perform well in a changing environment. This is typified by problems in robot control in which a robot has to perform a task in a variable environment, or computer interfaces that need to adapt to the idiosyncrasies of an individual user. Other problems require computers to be innovative—to construct something truly new and original, such as a new algorithm for accomplishing a computational task, or even a new scientific discovery. Finally, many computational problems require complex solutions that are difficult to program by hand. A striking example is the problem of creating artificial intelligence. Early on, AI practitioners believed that it would be straightforward to encode the rules that would confer intelligence in a program; expert systems are a good example. Nowadays, many AI researchers believe that the "rules" underlying intelligence are too complex for scientists to encode in a "top-down" fashion, and that the best route to artificial intelligence is through a "bottom-up" paradigm. In such a paradigm, human programmers encode simple rules, and complex behaviors such as intelligence emerge from these simple rules. Connectionism (i.e., the study of computer programs inspired by neural systems) is one example of this philosophy (Smolensky, 1988); evolutionary computation is another.

Biological evolution is an appealing source of inspiration for addressing these problems.

Evolution is, in effect, a method of searching among an enormous number of possibilities for solutions. In biology, the enormous set of possibilities is the set of possible genetic sequences, and the desired "solutions" are high-fitness organisms—organisms able to survive and reproduce in their environments. Seen in this light, the mechanisms of evolution can inspire computational search methods. Of course the fitness of a biological organism depends on many factors—for example, how well it can weather the physical characteristics of its environment and how well it can compete with or cooperate with the other organisms around it. The fitness criteria continually change as creatures evolve, so evolution is searching a constantly changing set of possibilities. Searching for solutions in the face of changing conditions is precisely what is required for adaptive computer programs. Furthermore, evolution is a massively parallel search method: rather than working on one species at a time, evolution tests and changes millions of species in parallel. Finally, viewed from a high level the "rules" of evolution are remarkably simple: species evolve by means of random variation (via mutation, recombination, and other operators), followed by natural selection in which the fittest tend to survive and reproduce, thus propagating their genetic material to future generations. Yet these simple rules are responsible, in large part, for the marvelous complexity we see in the biosphere, including human intelligence.

Elements of Genetic Algorithms

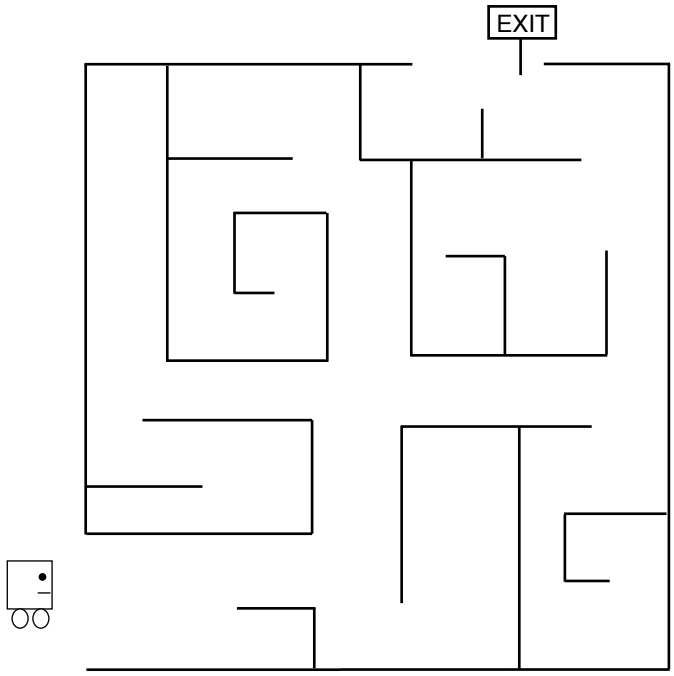
As was sketched above, a GA searches through a space of "chromosomes," each of which represents a candidate solution to a given problem (in some cases, a solution consists of a set of chromosomes). Most methods called "GAs" have at least the following elements in common: populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring.

The chromosomes in a GA population most often take the form of bit strings (i.e., strings of 1s and 0s); each bit position ("locus") in the chromosome has two possible values ("alleles"), 0 and 1. These biological terms are used in the spirit of analogy with real biology, though the entities they refer to are, of course, much simpler than the real biological ones. The search takes place by processing populations of chromosomes, changing from one such population to another. The GA most often requires a "fitness function" that assigns a score (fitness) to each chromosome in the current population. The fitness of the chromosome depends on how well that chromosome solves the problem at hand.

For example, a common application of GAs is function optimization, where the goal is to find a set of parameter values that maximizes, say, a complex multiparameter function. As a simple example, one might want to maximize the real-valued one-dimensional function

$$f(x) = x + |\sin(32x)|$$

over all values of x between 0 and π (Riolo, 1992). Here the candidate solutions are values of x, which can be encoded as bit strings representing real numbers. The fitness calculation translates a given bit string into a real number x and then evaluates the function at that value. The fitness of a string is the function value at that point.



0 0 = Backward; 0 1 = Left; 1 0 = Forward; 1 1 = Right

1 0 1 0 0 1 1 0 1 1 0 0 1 0 . . .
F F L F R B F

Figure 1: A robotic navigation problem, in which one candidate solution (a series of "forward," "backward," "left," and "right" moves) is encoded as a bit string.

As another simple example, consider the robotics problem illustrated in Figure 1. Here the problem is to find a sequence of, say, no more than 100 steps (or movements of the wheels) that will move the robot from the entrance to the exit.

As shown in the figure, each move (Forward, Backward, Left, or Right) can be represented by two bits, so a sequence of 100 moves can be represented by a bit string of length 200. A bit string encoding part of a candidate solution is also shown in the figure. The fitness of a particular sequence can be calculated by letting the robot follow the sequence, and then measuring the number of steps between its final position and the exit—the smaller the distance, the higher the fitness of the sequence. Maximum fitness is attained if the robot reaches the exit in 100 steps or less.

These examples show two different contexts in which candidate solutions to a problem are encoded as abstract bit-string "chromosomes," with fitness functions defined on the resulting space of bit strings. GAs are methods for searching a space of chromosomes for ones with high fitness.

The simplest form of GA involves three types of operators:

- **Selection:** This operator selects chromosomes in the population for reproduction. The fitter the chromosome, the more times it is likely to be selected to reproduce.

- **Crossover:** This operator exchanges subsequences of two chromosomes to create two offspring. For example, the strings

  10000100 and 11111111

  could be crossed over after the third locus in each to produce the two offspring

  10011111 and 11100100.

  This operator roughly mimics biological recombination between two single-chromosome (haploid) organisms. (Most higher organisms have chromosomes in pairs and are thus "diploid.")

- **Mutation:** This operator randomly flips some bits in a chromosome. For example, the string 00000100 might be mutated in its second position to yield 01000100. Mutation can occur at each bit position in a string with some probability, usually very small (e.g., 0.001).

A Simple Genetic Algorithm

Given a clearly defined problem to be solved and a bit-string representation for candidate solutions, the simple GA works as follows:

1. Start with a randomly generated population of N L-bit chromosomes (candidate solutions to a problem).
2. Calculate the fitness $F(x)$ of each chromosome x in the population.
3. Repeat the following steps (a)–(c) until N offspring have been created:
   (a) Select a pair of parent chromosomes from the current population, with the probability of selection being an increasing function of fitness. Selection is done "with replacement," meaning that the same chromosome can be selected more than once to become a parent.
   (b) With probability p_c (the crossover probability), cross over the pair at a randomly chosen point (chosen with uniform probability) to form two offspring. If no crossover takes place, form two offspring that are exact copies of their respective parents.
   (c) Mutate the two offspring at each locus with probability p_m (the mutation probability), and place the resulting chromosomes in the new population.
4. Replace the current population with the new population.
5. Go to step 2.

(This assumes that N is even; if N is odd, one offspring can be discarded at random.)

Each iteration of this process is called a "generation." A GA is typically iterated for anywhere from 50 to 500 or more generations, which is called a "run." At the end of a run, there are often one or more highly fit chromosomes in the population. Since randomness plays a large role in each run, two runs with different random number seeds will generally produce different detailed behavior. GA researchers often report statistics (such as the best fitness found and generation at which best fitness was found) averaged over many different runs of the GA on the same problem.

This simple procedure is the basis for most applications of GAs. There are a number of details to fill in, such as the size of the population and the probabilities of crossover and mutation, and the success of the algorithm often depends greatly on these details. There are also far more complicated versions of GAs, for example, GAs that work on representations other than bit strings or GAs that have different types of crossover and mutation operators.

Some Applications of Genetic Algorithms

The algorithm described above is very simple, but variations on this basic theme have been used in a large number of scientific and engineering problems and models, including the following:

- **Optimization**: GAs have been used in a wide variety of optimization tasks, including numerical optimization as well as combinatorial optimization problems such as circuit layout and job-shop scheduling.
- **Automatic Programming**: GAs have been used to evolve computer programs for specific tasks, and to design other computational structures, such as cellular automata and sorting networks.
- **Machine learning**: GAs have been used for many machine-learning applications, including classification and prediction tasks such as the prediction of weather or protein structure. GAs have also been used to evolve aspects of particular machine-learning systems, such as weights for neural networks, rules for learning classifier systems or symbolic production systems, and sensors for robots.
- **Economic models**: GAs have been used to model processes of innovation, the development of bidding strategies, and the emergence of economic markets.
- **Immune system models**: GAs have been used to model various aspects of the natural immune system including somatic mutation during an individual's lifetime and the discovery of multi-gene families during evolutionary time.
- **Ecological models**: GAs have been used to model ecological phenomena such as biological arms races, host-parasite co-evolution, symbiosis, and resource flow in ecologies.

		Player B: Cooperate	Player B: Defect
Player A	Cooperate	**3, 3**	**0, 5**
	Defect	**5, 0**	**1, 1**

Figure 2: The payoff matrix for the Prisoner's Dilemma (adapted from Axelrod, 1987). The two numbers given in each box are the payoffs for players A and B in that situation, with player A's payoff listed first in each pair.

- **Population genetics models**: GAs have been used to study questions in population genetics, such as "Under what conditions will a gene for recombination be evolutionarily viable?"
- **Interactions between evolution and learning**: GAs have been used to study how individual learning and species evolution affect one another.
- **Models of social systems**: GAs have been used to study evolutionary aspects of social systems, such as the evolution of cooperation, the evolution of communication, and trail-following behavior in ants.

This list is by no means exhaustive, but it gives the flavor of the kinds of things GAs have been used for, both in problem-solving and scientific contexts. Papers describing these and other applications will be found in the references listed in the "General References" section.

A Detailed Example: Using Genetic Algorithms to Evolve Strategies for the Prisoner's Dilemma

Let me now give a detailed example of a GA in action on one particularly interesting problem: automatically discovering good strategies for playing the Prisoner's Dilemma game.

The Prisoner's Dilemma is a simple two-person game, invented by Merrill Flood and Melvin Dresher in the 1950s, that has been studied extensively in game theory, economics, and political science because it can be seen as an idealized model for real-world phenomena such as arms races (Axelrod, 1984). It can be formulated as follows. Two people (call them Alice and Bob) are arrested for committing a crime together and are held in separate cells with no communication possible between them. Alice is offered the following deal: If she confesses and agrees to testify against Bob, she will receive a suspended sentence with probation, and Bob will be put away for five years. However, if at the same time Bob confesses and agrees to testify against her, her testimony will be discredited, and both will

receive four years for pleading guilty. She is also told that Bob is being offered precisely the same deal. Both Alice and Bob know that if neither testify against each other, they can be convicted only on a lesser charge that will get two years in jail for each of them.

What should Alice do? Should she "defect" against Bob and hope for the suspended sentence, risking getting a four-year sentence if Bob defects as well? Or should she "cooperate" with Bob (even though they cannot communicate), in the hope that he will also cooperate so each will only get two-year sentences, risking a defection by Bob that will send her away for five years?

The game can be described more abstractly. Each player independently decides whether to cooperate or defect. The game is summarized the payoff matrix shown in Figure 2—here, the goal is to get as many points—as opposed to as few years in prison—as possible. (In Figure 2, you can interpret the payoff in each case as 5 minus the number of years in prison.) If both players cooperate, they each get three points. If player A defects and player B cooperates, then player A gets five points and player B gets zero points, and vice versa if the situation is reversed. Finally, if both players defect, they each get one point. What is the best strategy to take in order to maximize one's own payoff? Clearly the best strategy is to defect: the worst consequence for a defector is to get one point and the best is to get five points, which are better than the worst score and the best score, respectively, for a cooperator. If you suspect your opponent is going to cooperate, then you should surely defect! And if you suspect your opponent is going to defect, then you should defect as well! In other words, no matter what the other player does it is always better to defect. The dilemma is that if both players defect, they get each get worse scores than if both players cooperate. If the game is *iterated*, that is, if two players play several turns in a row, both players always defecting will lead to a much lower total payoff than the players would get if they both cooperated. How can reciprocal cooperation be induced? This question takes on special significance when the notions of "cooperating" and "defecting" correspond to actions in, say, a real-world arms race (e.g., reducing or increasing one's arsenal).

Robert Axelrod of the University of Michigan has extensively studied the Prisoner's Dilemma and related games. His interest in determining what makes for a good strategy led him to organize two Prisoner's Dilemma tournaments (described in Axelrod, 1984). He solicited strategies from researchers in a number of disciplines. Each participant submitted a computer program that implemented a particular strategy, and the various programs played iterated games with each other. During each play, each program remembered what move both it and its opponent made on the three previous turns that they had played with each other, and its strategy was based on this memory. The programs were paired in a round-robin tournament, where each played with all of the other programs over a number of turns. The first tournament consisted of 14 different programs while the second tournament consisted of 63 programs (including one that made random moves). Some of the strategies submitted were rather complicated, using techniques such as Markov processes and Bayesian inference to model the other players in order to determine the best move. However, in both tournaments the winner (the strategy with the highest average score) was the simplest of the submitted strategies: TIT FOR TAT. This strategy, submitted by Anatole Rapoport, cooperates on the first turn and then, on subsequent turns, does whatever the other player

did on its last turn. That is, it offers cooperation and reciprocates it. But if the other player defects, TIT FOR TAT punishes that defection with defection of its own, and continues the punishment until the other player begins cooperating again.

After the two tournaments, Axelrod decided to see if a GA could *evolve* strategies to play this game successfully (Axelrod, 1987). The first problem was figuring out how to encode a strategy as a bit string. The encoding used by Axelrod is as follows. Suppose the memory of each player is one previous turn. There are four possibilities for the previous turn:

CC (case 1)
CD (case 2)
DC (case 3)
DD (case 4)

where C denotes "cooperate" and D denotes "defect." Case 1 is when both players cooperated on the previous turn, case two is when player A cooperated and player B defected, and so on. A strategy is simply a rule that specifies an action in each of these cases. For example, assuming that the strategy is for Player A, TIT FOR TAT is the following strategy:

If CC (case 1), then C.
If CD (case 2), then D.
If DC (case 3), then C.
If DD (case 4), then D.

Assuming that the cases are ordered in this canonical way, this strategy can be expressed compactly as the string $CDCD$. To use the string as a strategy, the player records the moves made on the previous turn (e.g., CD), finds the case number n by looking up that case in a table of ordered cases like that given above (for CD, $n = 2$), and selects the letter in the nth position of the bit string as its move on the next turn (for $n = 2$, the move is D).

Axelrod's tournaments involved strategies that remembered three previous turns. There are 64 ($2^2 \times 2^2 \times 2^2$) possibilities for the previous three turns:

$CC\ CC\ CC$ (case 1)
$CC\ CC\ CD$ (case 2)
$CC\ CC\ DC$ (case 3)
etc.

Thus a strategy can be encoded by a 64-bit string, e.g., $CDCCCDDCCCDD\ldots$ Since using the strategy requires the results of the three previous turns, Axelrod actually used a 70-bit string, where the six extra bits (Cs or Ds) encoded three hypothetical previous turns used by the strategy to decide what to do on the first actual turn of the game. Since each locus in the string has two possible alleles (C or D), the number of possible strategies is 2^{70}. The search space is thus far too big to search exhaustively.

In Axelrod's first experiment, the GA had a population of twenty such strategies. The fitness of a strategy in the population was determined as follows. Axelrod had found earlier

that eight of the human-generated strategies from the second tournament were representative of the entire set of strategies, in the sense that a given strategy's score playing with these eight was a good predictor of the strategy's score playing with all 63 entries. This set of eight strategies (which did *not* include TIT FOR TAT) served as the "environment" for the evolving strategies in the population. Each strategy in the population played iterated games with each of the eight fixed strategies, and the strategy's fitness was taken to be its average score over all the games that it played.

Axelrod performed forty different runs of fifty generations each, with different random number seeds used for each run. Most of the strategies that evolved were similar to TIT FOR TAT, in that the reciprocated cooperation and punished defection (although not necessarily based only on the immediately preceding move). However, the GA often found strategies that scored substantially higher than TIT FOR TAT. This is a striking result, especially in view of the fact that in a given run the GA is testing only $20 * 50 = 1000$ individuals out of a huge search space of 2^{70} possible individuals.

It would be wrong to conclude that the GA-evolved strategies are "better" than any human-designed strategy. The performance of a strategy depends very much on its environment—that is, on the strategies that it is playing with. Here the environment was fixed—it consisted of eight human-designed strategies that did not change over the course of a run. The highest-scoring strategies produced by the GA were "designed" to exploit specific weaknesses of several of the eight fixed strategies. It is not necessarily true that these high-scoring strategies would also score well in a different environment. TIT FOR TAT is a generalist, whereas the highest-scoring evolved strategies were more specialized to the given environment. Axelrod concluded that the GA is good at doing what evolution often does: developing highly specialized adaptations to specific characteristics of the environment.

To see the effects of a *changing* (as opposed to fixed) environment, Axelrod carried out another experiment in which the fitness of a strategy was determined by allowing the strategies in the population to play with *each other* rather than with the fixed set of eight strategies. Now the environment changed from generation to generation because the strategies themselves were evolving. At every generation, each strategy played iterated games with each of the nineteen other members of the population as well as with itself, and its fitness was again taken to be its average score over all games.

In this second set of experiments, Axelrod observed that the GA initially evolved uncooperative strategies, because in the first few generations strategies that tended to cooperate did not find reciprocation among their fellow population members and thus tended to die out. But after about 10 to 20 generations, the trend started to reverse: the GA discovered strategies that reciprocated cooperation and that punished defection (i.e., variants of TIT FOR TAT). These strategies did well with each other and were not completely defeated by other, less cooperative strategies, as were the initial cooperative strategies. The reciprocators scored above average, so they spread in the population, which resulted in more and more cooperation and increasing fitness.

This example illustrates how one might use a GA both to evolve solutions to an interesting problem and to model evolution and co-evolution in an idealized way. One can think of many

additional possible experiments, such as running the GA with the probability of crossover set to zero—that is, using only the selection and mutation operators (such an experiment was reported in Axelrod, 1987) or allowing a more open-ended kind of evolution in which the amount of memory available to a given strategy is allowed to increase with evolution (such an experiment was reported in Lindgren, 1991).

How and Why do Genetic Algorithms Work?

GAs are simple to describe and program but their behavior can be complicated, and many open questions exist about how and why they work and what they are good for. Much work has been done on the foundations of GAs (see, for example, Holland, 1975/1992; Goldberg, 1989a; Rawlins, 1991; Whitley, 1993; Whitley & Vose, to appear). The traditional theory of GAs (first formulated by Holland, 1975/1992) assumes that, at a very general level of description, GAs work by discovering, emphasizing, and recombining good *building blocks* of solutions in a highly parallel fashion. The idea here is that good solutions tend to be made up of good building blocks—combinations of bit values that often confer higher fitness to the strings in which they are present.

Most studies of the theory of GAs start with the notion of *schemas* (or "schemata") (Holland, 1975/1992), which formalizes the informal notion of "building blocks." A schema is a set of bit strings that can be described by a template made up of 1s, 0s, and $*$s, where the $*$s represent wild cards (or "don't cares"). For example, the schema $s = 1 * * * * 1$ represents the set of all 6-bit strings that begin and end with 1. The strings that fit this template (e.g., 100111 and 110011) are said to be *instances* of s. The schema s is said to have two *defined* bits (the number of non-$*$s) or, equivalently, to be of *order* 2. Its *defining length* (the distance between its outermost defined bits) is 5. Here I use the term "schema" to denote both a subset of strings represented by such a template as well as the template itself. In the following, the term's meaning should be clear from context.

Note that not every possible subset of the set of length-L bit strings can be described as a schema; in fact, the huge majority cannot. There are 2^L possible bit strings of length L and thus 2^{2^L} possible subsets of strings, but there are only 3^L possible schemas. However, a central tenet of traditional GA theory is that schemas are—implicitly—-the building blocks that the GA processes effectively under the operators of selection, mutation, and single-point crossover.

How does the GA process schemas? Any given bit string of length L is an instance of 2^L different schemas. For example, the string 11 is an instance of ** (all four possible bit strings of length 2), $*1$, $1*$, and 11 (the schema that contains only one string, 11). Thus any given population of N strings contains instances of between 2^L and $N \times 2^L$ different schemas. If all the strings are identical, then there are instances of exactly 2^L different schemas; otherwise, the number is less than or equal to $N \times 2^L$. This means that, at a given generation, while the GA is explicitly evaluating the fitnesses of the N strings in the population, it is actually *implicitly* estimating the average fitness of a much larger number of schemas, where the average fitness of a schema is defined to be the average fitness of all possible instances of that schema. For example, in a randomly generated population of N strings, on average

half the strings will be instances of $1***\ldots*$ and half will be instances of $0***\ldots*$. The evaluations of the approximately $N/2$ strings that are instances of $1***\ldots*$ give an estimate of the average fitness of that schema (this is an estimate because the instances evaluated in a typical-size population are only a small sample of all possible instances). In evaluating a population of N strings, the GA is implicitly estimating the average fitnesses of all schemas that are present in the population. This simultaneous evaluation of large numbers of schemas in a population of N strings is known as *implicit parallelism* (Holland, 1975/1992). The idea is that the degree of parallelism in a GA depends not on the number of strings in the population, but on the much larger number of schemas whose instances are present. The effect of selection is to gradually bias the sampling procedure toward instances of schemas whose fitness is estimated to be above average. Over time, the estimate of a schema's average fitness should in principle become more and more accurate since the GA is sampling more and more instances of that schema (some possible problems with this assumption are discussed by Grefenstette, 1991, among others).

We can calculate the approximate dynamics of this sample biasing as follows. Let s be a schema with at least one instance present in the population at time t. Let $N(s,t)$ be the number of instances of s at time t, and let $\hat{u}(s,t)$ be the observed average fitness of s at time t (i.e., the average fitness of instances of s in the population at time t). We want to calculate $E(N(s,t+1))$, the expected number of instances of s at time $t+1$. Assume that selection is carried out in proportion to fitness: the expected number of copies of a string x is equal to $F(x)/\overline{F}(t)$, where $F(x)$ is the fitness of string x in the population and $\overline{F}(t)$ is the average fitness of the population at time t. (For now, we'll ignore the effects of crossover and mutation.) Then, letting $x \in s$ denote "x is an instance of s,"

$$\begin{aligned} E(N(s,t+1)) &= \sum_{x \in s} F(x)/\bar{F}(t) \\ &= \frac{\hat{u}(s,t)}{\bar{F}(t)} N(s,t) \end{aligned} \tag{1}$$

by definition, since $\hat{u}(s,t) = (\sum_{x \in s} F(x))/N(s,t)$ for x in the population at time t.

Crossover and mutation can both destroy and create instances of s, so if we include the effects of crossover and mutation the right side of Eq. 1 can be modified to give a lower bound on $E(N(s,t+1))$. Let us consider the disruptive effects of crossover. Let p_c be the probability that single-point crossover will be applied to a string, and suppose that an instance of schema s is picked to be a parent. Schema s is said to "survive" under single-point crossover if one of the offspring is also an instance of schema s. We can give a lower bound on the probability $S_c(s)$ of s surviving under single-point crossover (which, you will recall, chooses a single point with uniform probability at which to perform the crossover):

$$S_c(s) \geq 1 - p_c \left(\frac{d(s)}{L-1} \right),$$

where $d(s)$ is the defining length of s and L is the length of bit strings in the search space. That is, crossovers occurring within the defining length of s can destroy s (i.e., produce offspring that are not instances of s), so we multiply the fraction of the string that s occupies

by the crossover probability to obtain an upper bound on the probability that it will be destroyed. (The value is an upper bound because some crossovers inside a schema's defined positions will not destroy it, e.g., if two identical strings cross with each other.) In short, the probability of survival under crossover is higher for shorter schemas.

The disruptive effects of mutation can be quantified as follows. Let p_m be the probability of any bit being mutated. Then $S_m(s)$, the probability that schema s will survive under mutation, is the following:

$$S_m(s) = (1 - p_m)^{o(s)},$$

where $o(s)$ is the order of s (i.e., the number of defined bits in s). That is, for each bit, the probability that it will not be mutated is $1 - p_m$, so the probability that no bits of schema s will be mutated is this quantity multiplied by itself $o(s)$ times. In short, the probability of survival under mutation is higher for lower-order schemas.

These disruptive effects can be used to amend Eq. 1:

$$E(N(s, t+1)) \geq \frac{\hat{u}(s,t)}{\overline{F}(t)} N(s,t) \left[1 - p_c \frac{d(s)}{l-1}\right] [(1 - p_m)^{o(s)}]. \quad (2)$$

This is known as the Schema Theorem (Holland, 1975/1992; see also Goldberg, 1989a). It can be interpreted to say that short, low-order schemas whose average fitness remains above the mean will receive exponentially increasing numbers of samples (i.e., instances evaluated) over time, since the number of samples of those schemas that are not disrupted and remain above average in fitness increases by a factor of $\hat{u}(s,t)/\overline{F}(t)$ at each generation.

The Schema Theorem as stated in Eq. 2 is a lower bound since it deals only with the destructive effects of crossover and mutation. However, crossover is believed to be a major source of the GA's creative power, taking the instances of high-average-fitness schemas that are emphasized in the population and recombining them to form instances of even fitter higher-order schemas that are themselves then emphasized via selection. The supposition that this is the process by which GAs work is known as the "Building-Block Hypothesis" (Holland, 1975/1992; Goldberg, 1989a). There has been some work done in the GA community on quantifying the "constructive" power of crossover (e.g., see Holland, 1975/1992, Chapter 6, and Spears, 1992).

The Schema Theorem and the Building-Block Hypothesis deal with the roles of selection and crossover in GAs. What is the role of mutation? Holland (1975/1992) proposed that mutation is what prevents the loss of diversity at a given bit position. For example, without mutation, all the strings in the population might come to have a 1 at the first bit position, and there would then be no way to obtain a string beginning with a zero. Mutation provides a kind of "insurance policy" against such fixation.

The reader may have noticed that the Schema Theorem given in Eq. 1 applies not only to schemas but to any subset of strings in the search space. The reason for specifically focusing on schemas is that they (in particular, short, high-average-fitness schemas) are a good description of the types of building blocks that are combined effectively by single-point crossover. A belief underlying this formulation of the GA is that schemas will be a good description of the relevant building blocks of a good solution. GA researchers have defined

other types of crossover operators that deal with different types of building blocks and have analyzed the generalized "schemas" that a given crossover operator effectively manipulates (e.g., Radcliffe, 1991; Vose, 1991).

The interpretation of the Schema Theorem as stated above makes a number of assumptions that may not hold for real GAs, and it has recently been the subject of much critical discussion in the GA community (e.g., see Grefenstette, 1991, 1993). Some other approaches to GA theory include modeling GAs with Markov chains (e.g., Vose, 1993), population genetics approaches (e.g., Booker, 1992; Altenberg, to appear), statistical mechanics approachs (e.g., Prügel-Bennett & Shapiro), and developing characterizations of problems on GAs are likely to perform well or outperform other search methods (e.g., Goldberg, 1989b, 1989c; Manderick et al., 1991; Forrest & Mitchell, 1993; Eshelman & Schaffer, 1993). The major problems of GA theory—predicting the behavior of GAs on realistic problems, characterizing the classes of problems for which GAs are appropriate search methods, developing GAs in which problem encodings and control parameters can themselves adapt, and using what we learn about GAs to understand evolutionary systems in nature—are still very much open.

Acknowledgments

Many thanks to Robert Axelrod, John Casti, Stephanie Forrest, Terry Jones, and Richard Palmer for comments on an earlier draft of this paper. I am also grateful to the Santa Fe Institute, the Alfred P. Sloan Foundation (grant B1992-46), the National Science Foundation (grant IRI-9320200), and the Department of Energy (grant DE-FG03-94ER25231) for providing research support.

General References

The following are some general references in the field of GAs.

Books

Holland, J. H. (1975/1992). *Adaptation in Natural and Artificial Systems.* Cambridge, MA: MIT Press. Second edition (First edition, 1975).

Goldberg, D. E (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, MA: Addison-Wesley.

Davis, L. D., ed. *Handbook of Genetic Algorithms.* New York: Van Nostrand Reinhold, 1991.

Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs.* Artificial Intelligence Series. Berlin: Springer-Verlag, 1992.

Review Articles

Holland, J. H. (1992). Genetic algorithms. *Scientific American*, July, 114–116.

Forrest, S. (1993). Genetic algorithms: Principles of natural selection applied to computation. *Science, 261*, 872-878.

Mitchell, M. (1993). Genetic algorithms. In Nadel, L. and Stein, D. L. (Eds.), *1992 Lectures in Complex Systems*, 3–87. Reading, MA: Addison-Wesley.

Goldberg, D. E. (1994). Genetic and evolutionary algorithms come of age. *Communications of the ACM, 37*(3), 113–119.

Conference Proceedings

Grefenstette, J. J. (Ed.) (1985). *Proceedings of an International Conference on Genetic Algorithms and Their Applications*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Grefenstette, J. J. (Ed.) (1987). *Proceedings of the Second International Conference on Genetic Algorithms*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Schaffer, J. D. (ed.) (1989). *Proceedings of the Third International Conference on Genetic Algorithms*. Los Altos, CA: Morgan-Kaufmann.

Belew, R. K. and Booker, L. B. (Eds.) (1991) *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann.

Forrest, S. (Ed.) (1993) *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann.

Langton, C. G. (Ed.) (1989). *Artificial Life*. Reading, MA: Addison-Wesley.

Langton, C. G., Taylor, C., Farmer, J. D., and Rasmussen, S. (Eds.) (1992). *Artificial Life II*. Reading, MA: Addison-Wesley.

Langton, C. G. (Ed.) (1993). *Artificial Life III*. Reading, MA: Addison-Wesley.

Brooks, R. A. and Maes, P. (Eds.) (1994). *Artificial Life IV*. Cambridge, MA: MIT Press.

Varela, F. J. and Bourgine, P. (Eds.) (1992). *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*. Cambridge, MA: MIT Press.

Schwefel, H.-P. and Männer, R. (Eds.) (1990). *Parallel Problem Solving From Nature*. Berlin: Springer-Verlag (Lecture Notes in Computer Science, Vol. 496).

Männer, R. and B. Manderick, (Eds.) (1992). *Parallel Problem Solving From Nature 2*. Amsterdam: North Holland.

Davidor, Y., Schwefel, H.-P. and Männer, R., (Eds.) (1994). *Parallel Problem Solving From Nature — PPSN III*. Berlin: Springer-Verlag (Lecture Notes in Computer Science, Vol. 866).

Meyer, J.-A. and Wilson, S. W. (Eds.) (1991). *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*. Cambridge, MA: MIT Press

Meyer, J.-A., Roitblatt, H. L., and Wilson, S. W. (Eds.) (1993). *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*. Cambridge, MA: MIT Press

Bibliography

Altenberg, L. (To appear). The Schema Theorem and Price's Theorem. To appear in Whitley, L. D. and Vose, M. D. (Eds.) *Foundations of Genetic Algorithms 3*. San Mateo, CA: Morgan Kaufmann.

Axelrod, R. (1984). *The Evolution of Cooperation*. New York: Basic Books.

Axelrod, R. (1987). The evolution of strategies in the iterated Prisoner's Dilemma. In *Genetic Algorithms and Simulated Annealing*, Davis, L. D. (editor). Research Notes in Artificial Intelligence. Los Altos, CA: Morgan Kaufmann,

Booker, L. B. (1993). Recombination distributions for genetic algorithms. In L. D. Whitley (editor), *Foundations of Genetic Algorithms 2*. Morgan Kaufmann.

Eshelman, L. J. and Schaffer, J. D. (1993). Crossover's niche. In S. Forrest (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms*, 9–14. Morgan Kaufmann, San Mateo, CA.

Forrest, S. and Mitchell, M. (1993b) Relative building block fitness and the building block hypothesis. In L. D. Whitley (editor), *Foundations of Genetic Algorithms 2*. Morgan Kaufmann.

Goldberg, D. E. (1989a) *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison Wesley.

Goldberg, D. E. (1989b). Genetic algorithms and Walsh functions: Part I, A gentle introduction. *Complex Systems 3*, 129–152.

Goldberg, D. E. (1989c). Genetic algorithms and Walsh functions: Part II, Deception and its analysis. *Complex Systems 3*, 153–171.

Grefenstette, J. J. (1991). Conditions for implicit parallelism. In G. Rawlins (editor), *Foundations of Genetic Algorithms*. Morgan Kaufmann.

Grefenstette, J. J. (1993). Deception considered harmful. In L. D. Whitley (editor), *Foundations of Genetic Algorithms 2*. Morgan Kaufmann.

Holland, J. H. (1975/1992). *Adaptation in Natural and Artificial Systems*. Cambridge, MA: MIT Press. Second edition (1992). (First edition, University of Michigan Press, 1975).

Lindgren, K. (1991). Evolutionary phenomena in simple dynamics. In Langton, C. G., Taylor, C., Farmer, J. D., and Rasmussen, S. (editors), *Artificial Life II*, 295–312. Santa Fe Institute Studies in the Sciences of Complexity, Proc. Vol. X, Reading, MA: Addison-Wesley

Manderick, B., de Weger, M., and Spiessens, P. (1991). The genetic algorithm and the structure of the fitness landscape. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, R. K. Belew and L. B. Booker (Eds.). San Mateo, CA: Morgan Kaufmann.

Prügel-Bennett, A. and Shapiro, J. L. (1994). An analysis of genetic algorithms using sta-

tistical mechanics. *Physical Review Letters, 72*(9), 1305–1309.

Radcliffe, N. J. (1991). Equivalence class analysis of genetic algorithms. *Complex Systems 5* (2), 183–205.

Rawlins, G. (Ed.) (1991) *Foundations of Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann.

Riolo, R. L. (1992). "Survival of the fittest bits." *Sci. Am.* July, 1992: 114–116.

Smolensky, P. (1988). On the proper treatment of connectionism. *Behavioral and Brain Sciences* 11(1): 1–14.

Vose, M. D. (1991). Generalizing the notion of schema in genetic algorithms. *Artificial Intelligence 50*, 385–396.

Vose, M. D. (1993). Modeling simple genetic algorithms. In L. D. Whitley (editor), *Foundations of Genetic Algorithms 2*, 63–73. Morgan Kaufmann.

Whitley, L. D. (Ed.) (1993) *Foundations of Genetic Algorithms 2.* San Mateo, CA: Morgan Kaufmann.

Whitley, L. D. and Vose, M. D. (Eds.) (To appear). *Foundations of Genetic Algorithms 3.* San Mateo, CA: Morgan Kaufmann.