Chapter 1: Genetic Algorithms: An Overview

Overview

Science arises from the very human desire to understand and control the world. Over the course of history, we humans have gradually built up a grand edifice of knowledge that enables us to predict, to varying extents, the weather, the motions of the planets, solar and lunar eclipses, the courses of diseases, the rise and fall of economic growth, the stages of language development in children, and a vast panorama of other natural, social, and cultural phenomena. More recently we have even come to understand some fundamental limits to our abilities to predict. Over the eons we have developed increasingly complex means to control many aspects of our lives and our interactions with nature, and we have learned, often the hard way, the extent to which other aspects are uncontrollable.

The advent of electronic computers has arguably been the most revolutionary development in the history of science and technology. This ongoing revolution is profoundly increasing our ability to predict and control nature in ways that were barely conceived of even half a century ago. For many, the crowning achievements of this revolution will be the creation—in the form of computer programs—of new species of intelligent beings, and even of new forms of life.

The goals of creating artificial intelligence and artificial life can be traced back to the very beginnings of the computer age. The earliest computer scientists—Alan Turing, John von Neumann, Norbert Wiener, 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 they 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 years, but since the early 1980s 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 are the most prominent example.

1.1 A BRIEF HISTORY OF EVOLUTIONARY COMPUTATION

In the 1950s and the 1960s several computer scientists independently studied evolutionary systems with the idea that evolution could be used as an optimization tool for engineering problems. The idea in all these systems was to evolve a population of candidate solutions to a given problem, using operators inspired by natural genetic variation and natural selection.

In the 1960s, Rechenberg (1965, 1973) introduced "evolution strategies" (*Evolutionsstrategie* in the original German), a method he used to optimize real−valued parameters for devices such as airfoils. This idea was further developed by Schwefel (1975, 1977). The field of evolution strategies has remained an active area of research, mostly developing independently from the field of genetic algorithms (although recently the two communities have begun to interact). (For a short review of evolution strategies, see Back, Hoffmeister, and Schwefel 1991.) Fogel, Owens, and Walsh (1966) developed "evolutionary programming," a technique in which candidate solutions to given tasks were represented as finite−state machines, which were evolved by randomly mutating their state−transition diagrams and selecting the fittest. A somewhat broader formulation of evolutionary programming also remains an area of active research (see, for example, Fogel and Atmar 1993). Together, evolution strategies, evolutionary programming, and genetic algorithms form the backbone of the field of evolutionary computation.

Several other people working in the 1950s and the 1960s developed evolution−inspired algorithms for optimization and machine learning. Box (1957), Friedman (1959), Bledsoe (1961), Bremermann (1962), and Reed, Toombs, and Baricelli (1967) all worked in this area, though their work has been given little or none of the kind of attention or followup that evolution strategies, evolutionary programming, and genetic algorithms have seen. In addition, a number of evolutionary biologists used computers to simulate evolution for the purpose of controlled experiments (see, e.g., Baricelli 1957, 1962; Fraser 1957 a,b; Martin and Cockerham 1960). Evolutionary computation was definitely in the air in the formative days of the electronic computer.

Genetic algorithms (GAs) were invented by John Holland in the 1960s and were developed by Holland and his students and colleagues at the University of Michigan in the 1960s and the 1970s. In contrast with evolution strategies and evolutionary programming, Holland's original goal was not to design algorithms to solve specific problems, but rather to formally study the phenomenon of adaptation as it occurs in nature and to 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* presented the genetic algorithm 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 ones and zeros, or "bits") to a new population by using a kind of "natural selection" together with the genetics−inspired operators of crossover, mutation, and inversion. Each chromosome consists of "genes" (e.g., bits), 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, and on average the fitter chromosomes produce more offspring than the 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. (Here, as in most of the GA literature, "crossover" and "recombination" will mean the same thing.)

Holland's introduction of a population−based algorithm with crossover, inversion, and mutation was a major innovation. (Rechenberg's evolution strategies started with a "population" of two individuals, one parent and one offspring, the offspring being a mutated version of the parent; many−individual populations and crossover were not incorporated until later. Fogel, Owens, and Walsh's evolutionary programming likewise used only mutation to provide variation.) Moreover, Holland was the first to attempt to put computational evolution on a firm theoretical footing (see Holland 1975). Until recently this theoretical foundation, based on the notion of "schemas," was the basis of almost all subsequent theoretical work on genetic algorithms

In the last several years there has been widespread interaction among researchers studying various evolutionary computation methods, and the boundaries between GAs, evolution strategies, evolutionary programming, and other evolutionary approaches have broken down to some extent. Today, researchers often use the term "genetic algorithm" to describe something very far from Holland's original conception. In this book I adopt this flexibility. Most of the projects I will describe here were referred to by their originators as GAs; some were not, but they all have enough of a "family resemblance" that I include them under the rubric of genetic algorithms.

1.2 THE APPEAL OF EVOLUTION

Why use evolution as an inspiration for solving computational problems? To evolutionary−computation researchers, the mechanisms of evolution seem well suited for some of the most pressing computational problems in many fields. Many computational problems require searching 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 or equations that will predict the ups and downs of a financial market, such as that for 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. For example, in searching for proteins with specified properties, rather than evaluate one amino acid sequence at a time it would be much faster to evaluate many simultaneously. What is needed is both computational parallelism (i.e., many processors evaluating sequences at the same time) and an intelligent strategy for choosing the next set of sequences to evaluate.

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, and by computer interfaces that must adapt to the idiosyncrasies of different users. Other problems require computer programs 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 on a program; expert systems were one result of this early optimism. Nowadays, many AI researchers believe that the "rules" underlying intelligence are too complex for scientists to encode by hand in a "top−down" fashion. Instead they believe that the best route to artificial intelligence is through a "bottom−up" paradigm in which humans write only very simple rules, and complex behaviors such as intelligence emerge from the massively parallel application and interaction of these simple rules. Connectionism (i.e., the study of computer programs inspired by neural systems) is one example of this philosophy (see Smolensky 1988); evolutionary computation is another. In connectionism the rules are typically simple "neural" thresholding, activation spreading, and strengthening or weakening of connections; the hoped−for emergent behavior is sophisticated pattern recognition and learning. In evolutionary computation the rules are typically "natural selection" with variation due to crossover and/or mutation; the hoped−for emergent behavior is the design of high−quality solutions to difficult problems and the ability to adapt these solutions in the face of a changing environment.

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 highly fit organisms—organisms well able to survive and reproduce in their environments. Evolution can also be seen as a method for *designing* innovative solutions to complex problems. For example, the mammalian immune system is a marvelous evolved solution to the problem of germs invading the body. 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 work 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 thought to be responsible, in large part, for the extraordinary variety and complexity we see in the biosphere.

1.3 BIOLOGICAL TERMINOLOGY

At this point it is useful to formally introduce some of the biological terminology that will be used throughout the book. In the context of genetic algorithms, these biological terms are used in the spirit of analogy with real biology, though the entities they refer to are much simpler than the real biological ones.

All living organisms consist of cells, and each cell contains the same set of one or more *chromosomes*—strings of DNA—that serve as a "blueprint" for the organism. A chromosome can be conceptually divided into *genes*— each of which encodes a particular protein. Very roughly, one can think of a gene as encoding a *trait*, such as eye color. The different possible "settings" for a trait (e.g., blue, brown, hazel) are called *alleles*. Each gene is located at a particular *locus* (position) on the chromosome.

Many organisms have multiple chromosomes in each cell. The complete collection of genetic material (all chromosomes taken together) is called the organism's *genome*. The term *genotype* refers to the particular set of genes contained in a genome. Two individuals that have identical genomes are said to have the same genotype. The genotype gives rise, under fetal and later development, to the organism's *phenotype*—its physical and mental characteristics, such as eye color, height, brain size, and intelligence.

Organisms whose chromosomes are arrayed in pairs are called *diploid*; organisms whose chromosomes are unpaired are called *haploid*. In nature, most sexually reproducing species are diploid, including human beings, who each have 23 pairs of chromosomes in each somatic (non−germ) cell in the body. During sexual reproduction, *recombination* (or *crossover*) occurs: in each parent, genes are exchanged between each pair of chromosomes to form a *gamete* (a single chromosome), and then gametes from the two parents pair up to create a full set of diploid chromosomes. In haploid sexual reproduction, genes are exchanged between the two parents' single−strand chromosomes. Offspring are subject to *mutation*, in which single nucleotides (elementary bits of DNA) are changed from parent to offspring, the changes often resulting from copying errors. The *fitness* of an organism is typically defined as the probability that the organism will live to reproduce (*viability*) or as a function of the number of offspring the organism has (*fertility*).

In genetic algorithms, the term *chromosome* typically refers to a candidate solution to a problem, often encoded as a bit string. The "genes" are either single bits or short blocks of adjacent bits that encode a particular element of the candidate solution (e.g., in the context of multiparameter function optimization the bits encoding a particular parameter might be considered to be a gene). An allele in a bit string is either 0 or 1; for larger alphabets more alleles are possible at each locus. Crossover typically consists of exchanging genetic material between two singlechromosome haploid parents. Mutation consists of flipping the bit at a randomly chosen locus (or, for larger alphabets, replacing a the symbol at a randomly chosen locus with a randomly chosen new symbol).

Most applications of genetic algorithms employ haploid individuals, particularly, single−chromosome individuals. The genotype of an individual in a GA using bit strings is simply the configuration of bits in that individual's chromosome. Often there is no notion of "phenotype" in the context of GAs, although more recently many workers have experimented with GAs in which there is both a genotypic level and a phenotypic level (e.g., the bit−string encoding of a neural network and the neural network itself).

1.4 SEARCH SPACES AND FITNESS LANDSCAPES

The idea of searching among a collection of candidate solutions for a desired solution is so common in computer science that it has been given its own name: searching in a "search space." Here the term "search space" refers to some collection of candidate solutions to a problem and some notion of "distance" between candidate solutions. For an example, let us take one of the most important problems in computational bioengineering: the aforementioned problem of computational protein design. Suppose you want use a computer to search for a protein—a sequence of amino acids—that folds up to a particular three−dimensional shape so it can be used, say, to fight a specific virus. The search space is the collection of all possible protein sequences—an infinite set of possibilities. To constrain it, let us restrict the search to all possible sequences of length 100 or less—still a huge search space, since there are 20 possible amino acids at each position in the sequence. (How many possible sequences are there?) If we represent the 20 amino acids by letters of the alphabet, candidate solutions will look like this:

A G G M C G B L.…

We will define the distance between two sequences as the number of positions in which the letters at corresponding positions differ. For example, the distance between A G G M C G B L and MG G M C G B L is 1, and the distance between A G G M C G B L and L B M P A F G A is 8. An algorithm for searching this space is a method for choosing which candidate solutions to test at each stage of the search. In most cases the next candidate solution(s) to be tested will depend on the results of testing previous sequences; most useful algorithms assume that there will be some correlation between the quality of "neighboring" candidate solutions—those close in the space. Genetic algorithms assume that high−quality "parent" candidate solutions from different regions in the space can be combined via crossover to, on occasion, produce high−quality "offspring" candidate solutions.

Another important concept is that of "fitness landscape." Originally defined by the biologist Sewell Wright (1931) in the context of population genetics, a fitness landscape is a representation of the space of all possible genotypes along with their fitnesses.

Suppose, for the sake of simplicity, that each genotype is a bit string of length l, and that the distance between two genotypes is their "Hamming distance"—the number of locations at which corresponding bits differ. Also suppose that each genotype can be assigned a real−valued fitness. A fitness landscape can be pictured as an (l + 1)−dimensional plot in which each genotype is a point in l dimensions and its fitness is plotted along the (l + 1)st axis. A simple landscape for $l = 2$ is shown in figure 1.1. Such plots are called landscapes because the plot of fitness values can form "hills," "peaks," "valleys," and other features analogous to those of physical landscapes. Under Wright's formulation, evolution causes populations to move along landscapes in particular ways, and "adaptation" can be seen as the movement toward local peaks. (A "local peak," or "local optimum," is not necessarily the highest point in the landscape, but any small

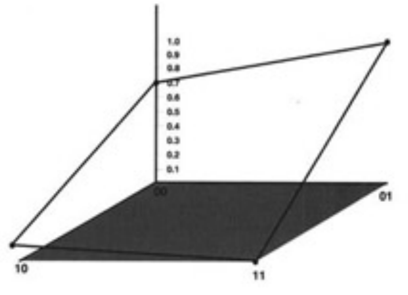


Figure 1.1: A simple fitness landscape for l = 2. Here f(00) = 0.7, f(01) = 1.0, f(10) = 0.1, and f(11) = 0.0.

movement away from it goes downward in fitness.) Likewise, in GAs the operators of crossover and mutation can be seen as ways of moving a population around on the landscape defined by the fitness function.

The idea of evolution moving populations around in unchanging landscapes is biologically unrealistic for several reasons. For example, an organism cannot be assigned a fitness value independent of the other organisms in its environment; thus, as the population changes, the fitnesses of particular genotypes will change as well. In other words, in the real world the "landscape" cannot be separated from the organisms that inhabit it. In spite of such caveats, the notion of fitness landscape has become central to the study of genetic algorithms, and it will come up in various guises throughout this book.

1.5 ELEMENTS OF GENETIC ALGORITHMS

It turns out that there is no rigorous definition of "genetic algorithm" accepted by all in the evolutionary−computation community that differentiates GAs from other evolutionary computation methods. However, it can be said that 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.Inversion—Holland's fourth element of GAs—is rarely used in today's implementations, and its advantages, if any, are not well established. (Inversion will be discussed at length in chapter 5.)

The chromosomes in a GA population typically take the form of bit strings. Each locus in the chromosome has two possible alleles: 0 and 1. Each chromosome can be thought of as a point in the search space of candidate solutions. The GA processes populations of chromosomes, successively replacing one such population with another. The GA most often requires a fitness function that assigns a score (fitness) to each chromosome in the current population. The fitness of a chromosome depends on how well that chromosome solves the problem at hand.

Examples of Fitness Functions

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

$$f(y) = y + |\sin(32y)|, \quad 0 \leq y < \pi$$

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

As a non−numerical example, consider the problem of finding a sequence of 50 amino acids that will fold to a desired three−dimensional protein structure. A GA could be applied to this problem by searching a population of candidate solutions, each encoded as a 50−letter string such as

IHCCVASASDMIKPVFTVASYLKNWTKAKGPNFEICISGRTPYWDNFPGI,

where each letter represents one of 20 possible amino acids. One way to define the fitness of a candidate sequence is as the negative of the potential energy of the sequence with respect to the desired structure. The potential energy is a measure of how much physical resistance the sequence would put up if forced to be folded into the desired structure—the lower the potential energy, the higher the fitness. Of course one would not want to physically force every sequence in the population into the desired structure and measure its resistance—this would be very difficult, if not impossible. Instead, given a sequence and a desired structure (and knowing some of the relevant biophysics), one can estimate the potential energy by calculating some of the forces acting on each amino acid, so the whole fitness calculation can be done computationally.

These examples show two different contexts in which candidate solutions to a problem are encoded as abstract chromosomes encoded as strings of symbols, with fitness functions defined on the resulting space of strings. A genetic algorithm is a method for searching such fitness landscapes for highly fit strings.

GA Operators

The simplest form of genetic algorithm involves three types of operators: selection, crossover (single point), and mutation.

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 randomly chooses a locus and exchanges the subsequences before and after that locus between 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. The crossover operator roughly mimics biological recombination between two single–chromosome (haploid) organisms.

Mutation This operator randomly flips some of the 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).

1.6 A SIMPLE GENETIC ALGORITHM

Given a clearly defined problem to be solved and a bit string representation for candidate solutions, a 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 until *n* offspring have been created:

    a. Select a pair of parent chromosomes from the current population, 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" or "crossover rate"), 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. (Note that here the crossover rate is defined to be the probability that two parents will cross over in a single point. There are also "multi–point crossover" versions of the GA in which the crossover rate for a pair of parents is the number of points at which a crossover takes place.)

c.

Mutate the two offspring at each locus with probability p_m (the mutation probability or mutation rate), and place the resulting chromosomes in the new population.

If *n* is odd, one new population member can be discarded at random.

4.

Replace the current population with the new population.

5.

Go to step 2.

Each iteration of this process is called a *generation*. A GA is typically iterated for anywhere from 50 to 500 or more generations. The entire set of generations 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 behaviors. GA researchers often report statistics (such as the best fitness found in a run and the generation at which the individual with that best fitness was discovered) averaged over many different runs of the GA on the same problem.

The simple procedure just described 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 more complicated versions of GAs (e.g., GAs that work on representations other than strings or GAs that have different types of crossover and mutation operators). Many examples will be given in later chapters.

As a more detailed example of a simple GA, suppose that *l* (string length) is 8, that $f(x)$ is equal to the number of ones in bit string *x* (an extremely simple fitness function, used here only for illustrative purposes), that *n*(the population size)is 4, that $p_c = 0.7$, and that $p_m = 0.001$. (Like the fitness function, these values of *l* and *n* were chosen for simplicity. More typical values of *l* and *n* are in the range 50–1000. The values given for p_c and p_m are fairly typical.)

The initial (randomly generated) population might look like this:

Chromosome label	Chromosome string	Fitness
A	00000110	2
B	11101110	6
C	00100000	1
D	00110100	3

A common selection method in GAs is *fitness–proportionate selection*, in which the number of times an individual is expected to reproduce is equal to its fitness divided by the average of fitnesses in the population. (This is equivalent to what biologists call "viability selection.")

A simple method of implementing fitness−proportionate selection is "roulette−wheel sampling" (Goldberg 1989a), which is conceptually equivalent to giving each individual a slice of a circular roulette wheel equal in area to the individual's fitness. The roulette wheel is spun, the ball comes to rest on one wedge−shaped slice, and the corresponding individual is selected. In the $n = 4$ example above, the roulette wheel would be spun four times; the first two spins might choose chromosomes B and D to be parents, and the second two spins might choose chromosomes B and C to be parents. (The fact that A might not be selected is just the luck of the draw. If the roulette wheel were spun many times, the average results would be closer to the expected values.)

Once a pair of parents is selected, with probability p_c they cross over to form two offspring. If they do not cross over, then the offspring are exact copies of each parent. Suppose, in the example above, that parents B and D cross over after the first bit position to form offspring E = 10110100 and F = 01101110, and parents B and C do not cross over, instead forming offspring that are exact copies of B and C. Next, each offspring is subject to mutation at each locus with probability p_m. For example, suppose offspring E is mutated at the sixth locus to form E' = 10110000, offspring F and C are not mutated at all, and offspring B is mutated at the first locus to form B' = 01101110. The new population will be the following:

Chromosome label	Chromosome string	Fitness
E'	10110000	3
F	01101110	5
C	00100000	1
B'	01101110	5

Note that, in the new population, although the best string (the one with fitness 6) was lost, the average fitness rose from 12/4 to 14/4. Iterating this procedure will eventually result in a string with all ones.

1.7 GENETIC ALGORITHMS AND TRADITIONAL SEARCH METHODS

In the preceding sections I used the word "search" to describe what GAs do. It is important at this point to contrast this meaning of "search" with its other meanings in computer science.

There are at least three (overlapping) meanings of "search":

Search for stored data Here the problem is to efficiently retrieve information stored in computer memory. Suppose you have a large database of names and addresses stored in some ordered way. What is the best way to search for the record corresponding to a given last name? "Binary search" is one method for efficiently finding the desired record. Knuth (1973) describes and analyzes many such search methods.

Search for paths to goals Here the problem is to efficiently find a set of actions that will move from a given initial state to a given goal. This form of search is central to many approaches in artificial intelligence. A simple example—all too familiar to anyone who has taken a course in AI—is the "8−puzzle," illustrated in figure 1.2. A set of tiles numbered 1–8 are placed in a square, leaving one space empty. Sliding one of the adjacent tiles into the blank space is termed a "move." Figure 1.2a illustrates the problem of finding a set of moves from the initial state to the state in which all the tiles are in order. A partial search tree corresponding to this problem is illustrated in figure 1.2b The "root" node represents the initial state, the nodes branching out from it represent all possible results of one move from that state, and so on down the tree. The search algorithms discussed in most AI contexts are methods for efficiently finding the best (here, the shortest) path in the tree from the initial state to the goal state. Typical algorithms are "depth−first search," "branch and bound," and "A*."

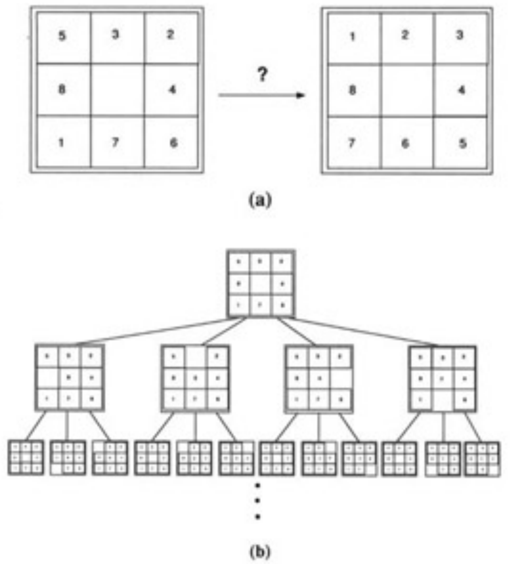


Figure 1.2: The 8−puzzle. (a) The problem is to find a sequence of moves that will go from the initial state to the state with the tiles in the correct order (the goal state). (b) A partial search tree for the 8−puzzle.

Search for solutions This is a more general class of search than "search for paths to goals." The idea is to efficiently find a solution to a problem in a large space of candidate solutions. These are the kinds of search problems for which genetic algorithms are used.

There is clearly a big difference between the first kind of search and the second two. The first concerns problems in which one needs to find a piece of information (e.g., a telephone number) in a collection of explicitly stored information. In the second two, the information to be searched is not explicitly stored; rather, candidate solutions are created as the search process proceeds. For example, the AI search methods for solving the 8−puzzle do not begin with a complete search tree in which all the nodes are already stored in memory; for most problems of interest there are too many possible nodes in the tree to store them all. Rather, the search tree is elaborated step by step in a way that depends on the particular algorithm, and the goal is to find an optimal or high−quality solution by examining only a small portion of the tree. Likewise, when searching a space of candidate solutions with a GA, not all possible candidate solutions are created first and then evaluated; rather, the GA is a method for finding optimal or good solutions by examining only a small fraction of the possible candidates.

"Search for solutions" subsumes "search for paths to goals," since a path through a search tree can be encoded as a candidate solution. For the 8−puzzle, the candidate solutions could be lists of moves from the initial state to some other state (correct only if the final state is the goal state). However, many "search for paths to goals" problems are better solved by the AI tree−search techniques (in which partial solutions can be evaluated) than by GA or GA−like techniques (in which full candidate solutions must typically be generated before they can be evaluated).

However, the standard AI tree−search (or, more generally, graph−search) methods do not always apply. Not all problems require finding a path from an initial state to a goal. For example, predicting the threedimensional structure of a protein from its amino acid sequence does not necessarily require knowing the sequence of physical moves by which a protein folds up into a 3D structure; it requires only that the final 3D configuration be predicted. Also, for many problems, including the protein−prediction problem, the configuration of the goal state is not known ahead of time.

The GA is a general method for solving "search for solutions" problems (as are the other evolution−inspired techniques, such as evolution strategies and evolutionary programming). Hill climbing, simulated annealing, and tabu search are examples of other general methods. Some of these are similar to "search for paths to goals" methods such as branch−and−bound and A*. For descriptions of these and other search methods see Winston 1992, Glover 1989 and 1990, and Kirkpatrick, Gelatt, and Vecchi 1983. "Steepest−ascent" hill climbing, for example, works as follows:

1. Choose a candidate solution (e.g., encoded as a bit string) at random. Call this string *current−string*.
2. Systematically mutate each bit in the string from left to right, one at a time, recording the fitnesses of the resulting one−bit mutants.
3. If any of the resulting one−bit mutants give a fitness increase, then set *current−string* to the one−bit mutant giving the highest fitness increase (the "steepest ascent").
4. If there is no fitness increase, then save *current−string* (a "hilltop") and go to step 1. Otherwise, go to step 2 with the new *current−string*.
5. When a set number of fitness−function evaluations has been performed, return the highest hilltop that was found.

In AI such general methods (methods that can work on a large variety of problems) are called "weak methods," to differentiate them from "strong methods" specially designed to work on particular problems. All the "search for solutions" methods (1) initially generate a set of candidate solutions (in the GA this is the initial population; in steepest−ascent hill climbing this is the initial string and all the one−bit mutants of it), (2) evaluate the candidate solutions according to some fitness criteria, (3) decide on the basis of this evaluation which candidates will be kept and which will be discarded, and (4) produce further variants by using some kind of operators on the surviving candidates.

The particular combination of elements in genetic algorithms—parallel population−based search with stochastic selection of many individuals, stochastic crossover and mutation—distinguishes them from other search methods. Many other search methods have some of these elements, but not this particular combination.

1.9 TWO BRIEF EXAMPLES

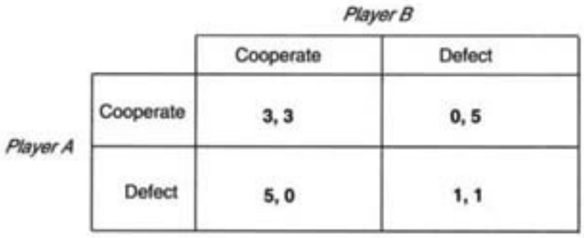
As warmups to more extensive discussions of GA applications, here are brief examples of GAs in action on two particularly interesting projects.

Using GAs to Evolve Strategies for the Prisoner's Dilemma

The Prisoner's Dilemma, a simple two–person game invented by Merrill Flood and Melvin Dresher in the 1950s, 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; Axelrod and Dion 1988). It can be formulated as follows: Two individuals (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 5 years. However, if at the same time Bob confesses and agrees to testify against Alice, her testimony will be discredited, and each will receive 4 years for pleading guilty. Alice is told that Bob is being offered precisely the same deal. Both Alice and Bob know that if neither testify against the other they can be convicted only on a lesser charge for which they will each get 2 years in jail.

Should Alice "defect" against Bob and hope for the suspended sentence, risking a 4–year sentence if Bob defects? Or should she "cooperate" with Bob (even though they cannot communicate), in the hope that he will also cooperate so each will get only 2 years, thereby risking a defection by Bob that will send her away for 5 years?

The game can be described more abstractly. Each player independently decides which move to make—i.e., whether to cooperate or defect. A "game" consists of each player's making a decision (a "move"). The possible results of a single game are summarized in a payoff matrix like the one shown in figure 1.3. Here the goal is to get as many points (as opposed to as few years in prison) as possible. (In figure 1.3, the payoff in each case can be interpreted as 5 minus the number of years in prison.) If both players cooperate, each gets 3 points. If player A defects and player B cooperates, then player A gets 5 points and player B gets 0 points, and vice versa if the situation is reversed. If both players defect, each gets 1 point. What is the best strategy to use in order to maximize one's own payoff? If you suspect that your opponent is going to cooperate, then you should surely defect. If you suspect that your opponent is going to defect, then you should defect too. No matter what the other player does, it is always better to defect. The dilemma is that if both players defect each gets a worse score than if they cooperate. If the game is *iterated* (that is, if the two players play several games in a row), both players' always defecting will lead to a much lower total payoff than the players would get if they 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).



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

Figure 1.3: 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 the given situation, with player A's payoff listed first in each pair.

Robert Axelrod of the University of Michigan has studied the Prisoner's Dilemma and related games extensively. 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 game, each program remembered what move (i.e., cooperate or defect) both it and its opponent had made in each of the three previous games that they had played with each other, and its strategy was based on this memory. The programs were paired in a round−robin tournament in which each played with all the other programs over a number of games. The first tournament consisted of 14 different programs; the second 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 Anatol Rapoport, cooperates in the first game and then, in subsequent games, does whatever the other player did in its move in the previous game with TIT FOR TAT. That is, it offers cooperation and reciprocates it. But if the other player defects, TIT FOR TAT punishes that defection with a defection of its own, and continues the punishment until the other player begins cooperating again.

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

- *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 in the previous game, case 2 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, TIT FOR TAT as played by player A is as follows:

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

If 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 in the previous game (e.g., *CD*), finds the case number i by looking up that case in a table of ordered cases like that given above (for *CD*, $i = 2$), and selects the letter in the ith position of the string as its move in the next game (for $i = 2$, the move is *D*).

Axelrod's tournaments involved strategies that remembered three previous games. There are 64 possibilities for the previous three games:

- *CC CC CC* (case 1),
- *CC CC CD* (case 2),
- *CC CC DC* (case 3),
    - ♦ î
- *DD DD DC* (case 63),
- *DD DD DD* (case 64).

Thus, a strategy can be encoded by a 64−letter string, e.g., *CDCCCDDCC CDD*…. Since using the strategy requires the results of the three previous games, Axelrod actually used a 70−letter string, where the six extra letters encoded three hypothetical previous games used by the strategy to decide how to move in the first actual game. Since each locus in the string has two possible alleles (*C* and *D*), the number of possible strategies is 2^{70}. The search space is thus far too big to be searched exhaustively.

In Axelrod's first experiment, the GA had a population of 20 such strategies. The fitness of a strategy in the population was determined as follows: Axelrod had found 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 individual in the population played iterated games with each of the eight fixed strategies, and the individual's fitness was taken to be its average score over all the games it played.

Axelrod performed 40 different runs of 50 generations each, using different random−number seeds for each run. Most of the strategies that evolved were similar to TIT FOR TAT in that they reciprocated cooperation and punished defection (although not necessarily only on the basis of 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 discovered strategies that are "better" than any human−designed strategy. The performance of a strategy depends very much on its environment—that is, on the strategies with which it is playing. Here the environment was fixed—it consisted of eight human−designed strategies that did not change over the course of a run. The resulting fitness function is an example of a static (unchanging) fitness landscape. 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 an individual was determined by allowing the individuals in the population to play with one another rather than with the fixed set of eight strategies. Now the environment changed from generation to generation because the opponents themselves were evolving. At every generation, each individual played iterated games with each of the 19 other members of the population and with itself, and its fitness was again taken to be its average score over all games. Here the fitness landscape was not static—it was a function of the particular individuals present in the population, and it changed as the population changed.

In this second set of experiments, Axelrod observed that the GA initially evolved uncooperative strategies. 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 10–20 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 one another and were not completely defeated by less cooperative strategies, as were the initial cooperative strategies. Because the reciprocators scored above average, they spread in the population; this resulted in increasing cooperation and thus increasing fitness.

Axelrod's experiments illustrate how one might use a GA both to evolve solutions to an interesting problem and to model evolution and coevolution in an idealized way. One can think of many additional possible experiments, such as running the GA with the probability of crossover set to 0—that is, using only the selection and mutation operators (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 or decrease (Lindgren 1992).

Hosts and Parasites: Using GAs to Evolve Sorting Networks

Designing algorithms for efficiently sorting collections of ordered elements is fundamental to computer science. Donald Knuth (1973) devoted more than half of a 700−page volume to this topic in his classic series *The Art of Computer Programming*. The goal of sorting is to place the elements in a data structure (e.g., a list or a tree) in some specified order (e.g., numerical or alphabetic) in minimal time. One particular approach to sorting described in Knuth's book is the *sorting network*, a parallelizable device for sorting lists with a fixed number n of elements. Figure 1.4 displays one such network (a "Batcher sort"—see Knuth 1973) that will sort lists of *n* = 16 elements (e_0–e_{15}). Each horizontal line represents one of the elements in the list, and each vertical arrow represents a comparison to be made between two elements. For example, the leftmost column of vertical arrows indicates that comparisons are to be made between e_0 and e_1, between e_2 and e_3, and so on. If the elements being compared are out of the desired order, they are swapped.

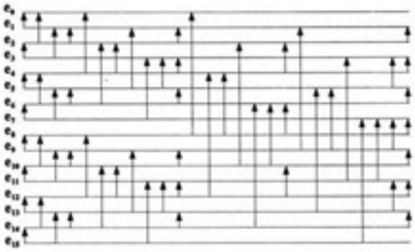


Figure 1.4: The "Batcher sort" n=16 sorting network (adapted from Knuth 1973). Each horizontal line

represents an element in the list, and each vertical arrow represents a comparison to be made between two elements. If the elements being compared are out of order, they are swapped. Comparisons in the same column can be made in parallel.

To sort a list of elements, one marches the list from left to right through the network, performing all the comparisons (and swaps, if necessary) specified in each vertical column before proceeding to the next. The comparisons in each vertical column are independent and can thus be performed in parallel. If the network is correct (as is the Batcher sort), any list will wind up perfectly sorted at the end. One goal of designing sorting networks is to make them correct *and* efficient (i.e., to minimize the number of comparisons).

An interesting theoretical problem is to determine the minimum number of comparisons necessary for a correct sorting network with a given *n*. In the 1960s there was a flurry of activity surrounding this problem for *n* = 16 (Knuth 1973; Hillis 1990,1992). According to Hillis (1990), in 1962

Bose and Nelson developed a general method of designing sorting networks that required 65 comparisons for *n* = 16, and they conjectured that this value was the minimum. In 1964 there were independent discoveries by Batcher and by Floyd and Knuth of a network requiring only 63 comparisons (the network illustrated in figure 1.4). This was again thought by some to be minimal, but in 1969 Shapiro constructed a network that required only 62 comparisons. At this point, it is unlikely that anyone was willing to make conjectures about the network's optimality—and a good thing too, since in that same year Green found a network requiring only 60 comparisons. This was an exciting time in the small field of *n* = 16 sorting−network design. Things seemed to quiet down after Green's discovery, though no proof of its optimality was given.

In the 1980s, W. Daniel Hillis (1990,1992) took up the challenge again, though this time he was assisted by a genetic algorithm. In particular, Hillis presented the problem of designing an optimal *n* = 16 sorting network to a genetic algorithm operating on the massively parallel Connection Machine 2.

As in the Prisoner's Dilemma example, the first step here was to figure out a good way to encode a sorting network as a string. Hillis's encoding was fairly complicated and more biologically realistic than those used in most GA applications. Here is how it worked: A sorting network can be specified as an ordered list of pairs, such as

(2,5),(4,2),(7,14)….

These pairs represent the series of comparisons to be made ("first compare elements 2 and 5, and swap if necessary; next compare elements 4 and 2, and swap if necessary"). (Hillis's encoding did not specify which comparisons could be made in parallel, since he was trying only to minimize the total number of comparisons rather than to find the optimal *parallel* sorting network.) Sticking to the biological analogy, Hillis referred to ordered lists of pairs representing networks as "phenotypes." In Hillis's program, each phenotype consisted of 60–120 pairs, corresponding to networks with 60–120 comparisons. As in real genetics, the genetic algorithm worked not on phenotypes but on genotypes encoding the phenotypes.

The genotype of an individual in the GA population consisted of a set of chromosomes which could be decoded to form a phenotype. Hillis used diploid chromosomes (chromosomes in pairs) rather than the haploid chromosomes (single chromosomes) that are more typical in GA applications. As is illustrated in figure 1.5a, each individual consists of 15 pairs of 32−bit chromosomes. As is illustrated in figure 1.5b, each chromosome consists of eight 4−bit "codons." Each codon represents an integer between 0 and 15 giving a position in a 16−element list. Each adjacent pair of codons in a chromosome specifies a comparison between two list elements. Thus each chromosome encodes four comparisons. As is illustrated in figure 1.5c, each pair of chromosomes encodes between four and eight comparisons. The chromosome pair is aligned and "read off" from left to right. At each position, the codon pair in chromosome A is compared with the codon pair in chromosome B. If they encode the same pair of numbers (i.e., are "homozygous"), then only one pair of numbers is inserted in the phenotype; if they encode different pairs of numbers (i.e., are "heterozygou"), then both pairs are inserted in the phenotype. The 15 pairs of chromosomes are read off in this way in a fixed order to produce a phenotype with 60–120 comparisons. More homozygous positions appearing in each chromosome pair means fewer comparisons appearing in the resultant sorting network. The goal is for the GA to discover a minimal correct sorting network—to equal Green's network, the GA must discover an individual with all homozygous positions in its genotype that also yields a correct sorting network. Note that under Hillis's encoding the GA cannot discover a network with fewer than 60 comparisons.

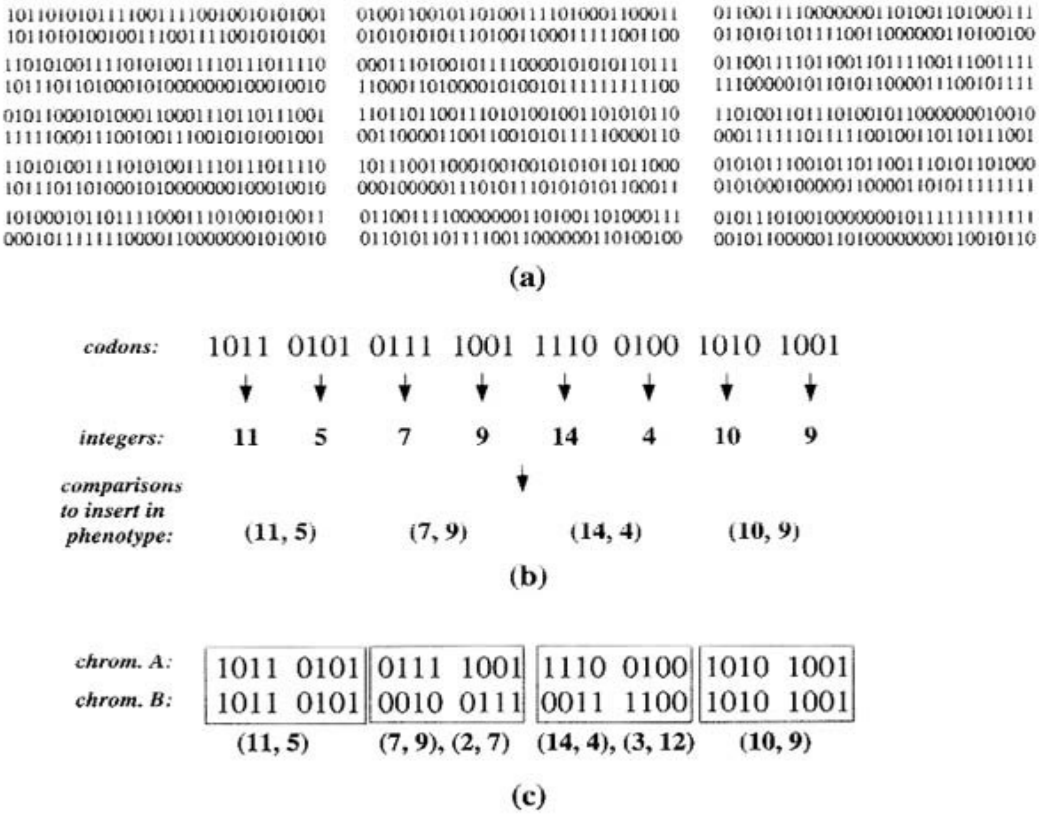


Figure 1.5: Details of the genotype representation of sorting networks used in Hillis's experiments. (a) An example of the genotype for an individual sorting network, consisting of 15 pairs of 32–bit chromosomes. (b) An example of the integers encoded by a single chromosome. The chromosome given here encodes the integers 11,5,7,9,14,4,10, and 9; each pair of adjacent integers is interpreted as a comparison. (c) An example of the comparisons encoded by a chromosome pair. The pair given here contains two homozygous positions and thus encodes a total of six comparisons to be inserted in the phenotype: (11,5), (7,9), (2,7), (14,4), (3,12), and (10,9).

In Hillis's experiments, the initial population consisted of a number of randomly generated genotypes, with one noteworthy provision: Hillis noted that most of the known minimal 16–element sorting networks begin with the same pattern of 32 comparisons, so he set the first eight chromosome pairs in each individual to (homozygously) encode these comparisons. This is an example of using knowledge about the problem domain (here, sorting networks) to help the GA get off the ground.

Most of the networks in a random initial population will not be correct networks—that is, they will not sort all input cases (lists of 16 numbers) correctly. Hillis's fitness measure gave partial credit: the fitness of a network was equal to the percentage of cases it sorted correctly. There are so many possible input cases that it was not practicable to test each network

exhaustively, so at each generation each network was tested on a sample of input cases chosen at random.

Hillis's GA was a considerably modified version of the simple GA described above. The individuals in the initial population were placed on a two–dimensional lattice; thus, unlike in the simple GA, there is a notion of spatial distance between two strings. The purpose of placing the population on a spatial lattice was to foster "speciation" in the population—Hillis hoped that different types of networks would arise at different spatial locations, rather than having the whole population converge to a set of very similar networks.

The fitness of each individual in the population was computed on a random sample of test cases. Then the half of the population with lower fitness was deleted, each lower–fitness individual being replaced on the grid with a copy of a surviving neighboring higher–fitness individual.

That is, each individual in the higher–fitness half of the population was allowed to reproduce once.

Next, individuals were paired with other individuals in their local spatial neighborhoods to produce offspring. Recombination in the context of diploid organisms is different from the simple haploid crossover described above. As figure 1.6 shows, when two individuals were paired, crossover took place within each chromosome pair inside each individual. For each of the 15 chromosome pairs, a crossover point was chosen at random, and a single "gamete" was formed by taking the codons before the crossover point from the first chromosome in the pair and the codons after the crossover point from the second chromosome in the pair. The result was 15 haploid gametes from each parent. Each of the 15 gametes from the first parent was then paired with one of the 15 gametes from the second parent to form a single diploid offspring. This procedure is roughly similar to sexual reproduction between diploid organisms in nature.

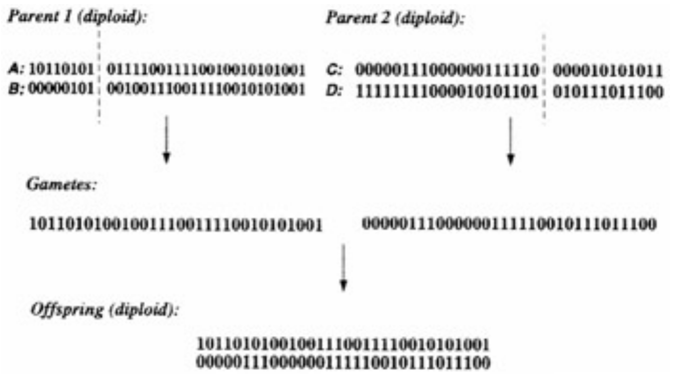


Figure 1.6: An illustration of diploid recombination as performed in Hillis's experiment. Here an individual's genotype consisted of 15 pairs of chromosomes (for the sake of clarity, only one pair for each parent is shown). A crossover point was chosen at random for each pair, and a gamete was formed by taking the codons before the crossover point in the first chromosome and the codons after the crossover point in the second chromosome. The 15 gametes from one parent were paired with the 15 gametes from the other parent to make a new individual. (Again for the sake of clarity, only one gamete pairing is shown.)

Such matings occurred until a new population had been formed. The individuals in the new population were then subject to mutation with $p_m = 0.001$. This entire process was iterated for a number of generations.

Since fitness depended only on network correctness, not on network size, what pressured the GA to find minimal networks? Hillis explained that there was an indirect pressure toward minimality, since, as in nature, homozygosity can protect crucial comparisons. If a crucial comparison is at a heterozygous position in its chromosome, then it can be lost under a crossover, whereas crucial comparisons at homozygous positions cannot be lost under crossover. For example, in figure 1.6, the leftmost comparison in chromosome B (i.e., the leftmost eight bits, which encode the comparison (0, 5)) is at a heterozygous position and is lost under this recombination (the gamete gets its leftmost comparison from chromosome A), but the rightmost comparison in chromosome A (10, 9) is at a homozygous position and is retained (though the gamete gets its rightmost comparison from chromosome B). In general, once a crucial comparison or set of comparisons is discovered, it is highly advantageous for them to be at homozygous positions. And the more homozygous positions, the smaller the resulting network.

In order to take advantage of the massive parallelism of the Connection Machine, Hillis used very large populations, ranging from 512 to about 1 million individuals. Each run lasted about 5000 generations. The smallest correct network found by the GA had 65 comparisons, the same as in Bose and Nelson's network but five more than in Green's network.

Hillis found this result disappointing—why didn't the GA do better? It appeared that the GA was getting stuck at local optima—local "hilltops" in the fitness landscape—rather than going to the globally highest hilltop. The GA found a number of moderately good (65−comparison) solutions, but it could not proceed further. One reason was that after early generations the randomly generated test cases used to compute the fitness of each individual were not challenging enough. The networks had found a strategy that worked, and the difficulty of the test cases was staying roughly the same. Thus, after the early generations there was no pressure on the networks to change their current suboptimal sorting strategy.

To solve this problem, Hillis took another hint from biology: the phenomenon of host−parasite (or predator−prey) coevolution. There are many examples in nature of organisms that evolve defenses to parasites that attack them only to have the parasites evolve ways to circumvent the defenses, which results in the hosts' evolving new defenses, and so on in an ever−rising spiral—a "biological arms race." In Hillis's analogy, the sorting networks could be viewed as hosts, and the test cases (lists of 16 numbers) could be viewed as parasites. Hillis modified the system so that a population of networks coevolved on the same grid as a population of parasites, where a parasite consisted of a set of 10–20 test cases. Both populations evolved under a GA. The fitness of a network was now determined by the parasite located at the network's grid location. The network's fitness was the percentage of test cases in the parasite that it sorted correctly. The fitness of the parasite was the percentage of its test cases that stumped the network (i.e., that the network sorted incorrectly).

The evolving population of test cases provided increasing challenges to the evolving population of networks. As the networks got better and better at sorting the test cases, the test cases got harder and harder, evolving to specifically target weaknesses in the networks. This forced the population of networks to keep changing—i.e., to keep discovering new sorting strategies—rather than staying stuck at the same suboptimal strategy. With coevolution, the GA discovered correct networks with only 61 comparisons—a real improvement over the best networks discovered without coevolution, but a frustrating single comparison away from rivaling Green's network.

Hillis's work is important because it introduces a new, potentially very useful GA technique inspired by coevolution in biology, and his results are a convincing example of the potential power of such biological inspiration. However, although the host−parasite idea is very appealing, its usefulness has not been established beyond Hillis's work, and it is not clear how generally it will be applicable or to what degree it will scale up to more difficult problems (e.g., larger sorting networks). Clearly more work must be done in this very interesting area.

1.10 HOW DO GENETIC ALGORITHMS WORK?

Although genetic algorithms are simple to describe and program, their behavior can be complicated, and many open questions exist about how they work and for what types of problems they are best suited. Much work has been done on the theoretical foundations of GAs (see, e.g., Holland 1975; Goldberg 1989a; Rawlins 1991; Whitley 1993b; Whitley and Vose 1995). Chapter 4 describes some of this work in detail. Here I give a brief overview of some of the fundamental concepts.

The traditional theory of GAs (first formulated in Holland 1975) 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 confer higher fitness on the strings in which they are present.

Holland (1975) introduced the notion of *schemas* (or *schemata*) to formalize the informal notion of "building blocks." A schema is a set of bit strings that can be described by a template made up of ones, zeros, and asterisks, the asterisks representing wild cards (or "don't cares"). For example, the schema $H = 1 * * * * 1$ represents the set of all 6–bit strings that begin and end with 1. (In this section I use Goldberg's (1989a) notation, in which H stands for "hyperplane." H is used to denote schemas because schemas define hyperplanes—"planes" of various dimensions—in the $l$dimensional space of length–l bit strings.) The strings that fit this template (e.g., 100111 and 110011) are said to be*instances* of H.The schema H is said to have two *defined* bits (non–asterisks) 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 and 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***···* and half will be instances of 0 ***···*. The evaluations of the approximately $n/2$ strings that are instances of 1***···* give an estimate of the average fitness of that schema (this is an estimate because the instances evaluated in typical–size population are only a small sample of all possible instances). Just as schemas are not explicitly represented or evaluated by the GA, the estimates of schema average fitnesses are not calculated or stored explicitly by the GA. However, as will be seen below, the GA's behavior, in terms of the increase and decrease in numbers of instances of given schemas in the population, can be described as though it actually were calculating and storing these averages.

We can calculate the approximate dynamics of this increase and decrease in schema instances as follows. Let H be a schema with at least one instance present in the population at time t. Let $m(H,t)$ be the number of instances of H at time *t*, and let *Û* (*H*,*t*) be the observed average fitness of *H* at time *t* (i.e., the average fitness of instances of *H* in the population at time *t*). We want to calculate E(*m*(*H*, *t* + 1)), the expected number of instances of *H* at time *t* + 1. Assume that selection is carried out as described earlier: the expected number of offspring of a string *x* is equal to *f*(*x*)/

$\bar{f}(t)$ where *f*(*x*)is the fitness of *x* and $\bar{f}(t)$ is the average fitness of the population at time *t*. Then, assuming *x* is in the population at time *t*, letting *x* Î *H* denote "*x* is an instance of *H*," and (for now) ignoring the effects of crossover and mutation, we have

(1.1)
$$E(m(H,t+1)) = \sum_{x\in H} f(x)/\bar{f}(t)$$
$$= (\hat{u}(H,t)/\bar{f}(t))m(H,t)$$

by definition, since *Û*(*H*, *t*) = ($£_{xÎH}$ *f*(*x*))/*m*(*H*,*t*)for *x* in the population at time *t*. Thus even though the GA does not calculate *Û*(*H*,*t*) explicitly, the increases or decreases of schema instances in the population depend on this quantity.

Crossover and mutation can both destroy and create instances of *H*. For now let us include only the destructive effects of crossover and mutation—those that decrease the number of instances of *H*. Including these effects, we modify the right side of equation 1.1 to give a lower bound on *E*(*m*(*H*,*t* + 1)). Let p_c be the probability that single–point crossover will be applied to a string, and suppose that an instance of schema *H* is picked to be a parent. Schema *H* is said to "survive" under singlepoint crossover if one of the offspring is also an instance of schema *H*. We can give a lower bound on the probability $S_c(H)$ that *H* will survive single–point crossover:

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

where *d*(*H*) is the defining length of *H* and *l* is the length of bit strings in the search space. That is, crossovers occurring within the defining length of *H* can destroy *H* (i.e., can produce offspring that are not instances of *H*), so we multiply the fraction of the string that *H* 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.) Subtracting this value from 1 gives a lower bound on the probability of survival $S_c(H)$. 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 $S_m(H)$, the probability that schema *H* will survive under mutation of an instance of *H*, is equal to $(1 \;\; p_{m})^{o(H)}$, where *o*(*H*) is the order of *H* (i.e., the number of defined bits in *H*). That is, for each bit, the probability that the bit will not be mutated is 1 p_{m}, so the probability that no defined bits of schema *H* will be mutated is this quantity multiplied by itself *o*(*H*) times. In short, the probability of survival under mutation is higher for lower–order schemas.

These disruptive effects can be used to amend equation 1.1:

(1.2)
$$E(m(H,t+1)) \geq \frac{\hat{u}(H,t)}{\bar{f}(t)}m(H,t)\left(1 - p_c\frac{d(H)}{l-1}\right)[(1-p_m)^{o(H)}].$$

This is known as the Schema Theorem (Holland 1975; see also Goldberg 1989a). It describes the growth of a schema from one generation to the next. The Schema Theorem is often interpreted as implying 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 $\hat{U}(H,t)/f(t)$ at each generation. (There are some caveats on this interpretation; they will be discussed in chapter 4.)

The Schema Theorem as stated in equation 1.2 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 power, with the ability to recombine instances of good schemas to form instances of equally good or better higher−order schemas. The supposition that this is the process by which GAs work is known as the Building Block Hypothesis (Goldberg 1989a). (For work on quantifying this "constructive" power of crossover, see Holland 1975, Thierens and Goldberg 1993, and Spears 1993.)

In evaluating a population of n strings, the GA is implicitly estimating the average fitnesses of all schemas that are present in the population, and increasing or decreasing their representation according to the Schema Theorem. This simultaneous implicit evaluation of large numbers of schemas in a population of n strings is known as *implicit paralelism* (Holland 1975). 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 counterexamples to this notion of increasing accuracy will be discussed in chapter 4.)

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

The Schema Theorem given in equation 1.1 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 (Radcliffe 1991; Vose 1991).

The Schema Theorem and some of its purported implications for the behavior of GAs have recently been the subject of much critical discussion in the GA community. These criticisms and the new approaches to GA theory inspired by them will be reviewed in chapter 4.

THOUGHT EXERCISES

1. How many Prisoner's Dilemma strategies with a memory of three games are there that are behaviorally equivalent to TIT FOR TAT? What fraction is this of the total number of strategies with a memory of three games?

2.

What is the total payoff after 10 games of TIT FOR TAT playing against (a) a strategy that always defects; (b) a strategy that always cooperates; (c) ANTI–TIT–FOR–TAT, a strategy that starts out by defecting and always does the opposite of what its opponent did on the last move? (d) What is the expected payoff of TIT FOR TAT against a strategy that makes random moves? (e) What are the total payoffs of each of these strategies in playing 10 games against TIT FOR TAT? (For the random strategy, what is its expected average payoff?)

3. How many possible sorting networks are there in the search space defined by Hillis's representation?

4. Prove that any string of length l is an instance of 2^l different schemas.

5. Define the fitness f of bit string x with $l = 4$ to be the integer represented by the binary number x. (e.g., $f(0011) = 3$, $f(1111) = 15$). What is the average fitness of the schema 1*** under f? What is the average fitness of the schema 0*** under f?

6. Define the fitness of bit string x to be the number of ones in x. Give a formula, in terms of l (the string length) and k, for the average fitness of a schema H that has k defined bits, all set to 1.

7. When is the union of two schemas also a schema? For example, {0*}*{1*} is a schema (**), but {01}*{10} is not. When is the intersection of two schemas also a schema? What about the difference of two schemas?

8. Are there any cases in which a population of n l–bit strings contains *exactly* $n \times 2^l$ different schemas?

COMPUTER EXERCISES

(Asterisks indicate more difficult, longer–term projects.)

1. Implement a simple GA with fitness–proportionate selection, roulettewheel sampling, population size 100, single–point crossover rate $p_c = 0.7$, and bitwise mutation rate $p_m = 0.001$. Try it on the following fitness function: $f(x)$ = number of ones in x, where x is a chromosome of length 20. Perform 20 runs, and measure the average generation at which the string of all ones is discovered. Perform the same experiment with crossover turned off (i.e., $p_c = 0$). Do similar experiments, varying the mutation and crossover rates, to see how the variations affect the average time required for the GA to find the optimal string. If it turns out that mutation with crossover is better than mutation alone, why is that the case?

2. Implement a simple GA with fitness–proportionate selection, roulettewheel sampling, population size 100, single–point crossover rate $p_c = 0.7$, and bitwise mutation rate $p_m = 0.001$. Try it on the fitness function $f(x)$ = the integer represented by the binary number x, where x is a chromosome of length 20.

Run the GA for 100 generations and plot the fitness of the best individual found at each generation as well as the average fitness of the population at each generation. How do these plots change as you vary the population size, the crossover rate, and the mutation rate? What if you use only mutation (i.e., $p_c = 0$)?

3. Define ten schemas that are of particular interest for the fitness functions of computer exercises 1 and 2 (e.g., 1*···* and 0*···*). When running the GA as in computer exercises 1 and 2, record at each generation how many instances there are in the population of each of these schemas. How well do the data agree with the predictions of the Schema Theorem?

4. Compare the GA's performance on the fitness functions of computer exercises 1 and 2 with that of steepest–ascent hill climbing (defined above) and with that of another simple hill–climbing method, "random–mutation hill climbing" (Forrest and Mitchell 1993b):

   1. Start with a single randomly generated string. Calculate its fitness.

   2. Randomly mutate one locus of the current string.

   3. If the fitness of the mutated string is equal to or higher than the fitness of the original string, keep the mutated string. Otherwise keep the original string.

   4. Go to step 2.

   Iterate this algorithm for 10,000 steps (fitness–function evaluations). This is equal to the number of fitness–function evaluations performed by the GA in computer exercise 2 (with population size 100 run for 100 generations). Plot the best fitness found so far at every 100 evaluation steps (equivalent to one GA generation), averaged over 10 runs. Compare this with a plot of the GA's best fitness found so far as a function of generation. Which algorithm finds higher–fitness chromosomes? Which algorithm finds them faster? Comparisons like these are important if claims are to be made that a GA is a more effective search algorithm than other stochastic methods on a given problem.

5. *

   Implement a GA to search for strategies to play the Iterated Prisoner's Dilemma, in which the fitness of a strategy is its average score in playin 100 games with itself and with every other member of the population. Each strategy remembers the three previous turns with a given player. Use a population of 20 strategies, fitness–proportional selection, single–point crossover with $p_c = 0.7$, and mutation with $p_m = 0.001$.

   a. See if you can replicate Axelrod's qualitative results: do at least 10 runs of 50 generations each and examine the results carefully to find out how the best–performing strategies work and how they change from generation to generation.

   b.

Turn off crossover (set $p_c = 0$) and see how this affects the average best fitness reached and the average number of generations to reach the best fitness. Before doing these experiments, it might be helpful to read Axelrod 1987.

c.

Try varying the amount of memory of strategies in the population. For example, try a version in which each strategy remembers the four previous turns with each other player. How does this affect the GA's performance in finding high−quality strategies? (This is for the very ambitious.)

d.

See what happens when noise is added—i.e., when on each move each strategy has a small probability (e.g., 0.05) of giving the opposite of its intended answer. What kind of strategies evolve in this case? (This is for the even more ambitious.)

6.

*

a.

Implement a GA to search for strategies to play the Iterated Prisoner's Dilemma as in computer exercise 5a, except now let the fitness of a strategy be its score in 100 games with TIT FOR TAT. Can the GA evolve strategies to beat TIT FOR TAT?

b.

Compare the GA's performance on finding strategies for the Iterated Prisoner's Dilemma with that of steepest−ascent hill climbing and with that of random−mutation hill climbing. Iterate the hill−climbing algorithms for 1000 steps (fitness−function evaluations). This is equal to the number of fitness−function evaluations performed by a GA with population size 20 run for 50 generations. Do an analysis similar to that described in computer exercise 4.