Genetic Algorithms

Peter J. Denning

Biological analogies have been part of the science and the lore of computation since the 1940s. The early theory of automata, which assumed machines were made of neuron-like components, produced the first examples of self-reproducing machines. Over the years many debates in artificial intelligence have centered on biological metaphors—for example, whether machines can think, whether rule-based expert systems can be competent when judged by human standards, and whether neural networks can give machines the ability to see or hear. Recent biocomputational successes with robot insects and simulations of population dynamics have encouraged a growing number of adherents to a new field called artificial life. Computer scientists and molecular biologists have begun to explore collaborative research *(1)*. And the metaphor has even wider application: Computer viruses are routinely discussed as if they were a form of parasitic life inside a computer.

Analogies between computing and biology are more than coincidence: Both genes and computers record, copy, and disseminate information. Douglas Hofstadter of Indiana University showed this clearly by demonstrating that the action of DNA and RNA during the reproduction of the living cell can be interpreted as an example of a self-reproducing Turing machine *(2)*.

Nowhere have these analogies produced greater successes than with genetic algorithms, a family of methods that search for optimal solutions of difficult problems. The story begins in the late 1960s at the University of Michigan, where John Holland and his students investigated how to build machines that can learn. Holland noted that learning can occur not only by adaptation in a single organism but also by evolutionary adaptation over many generations of a species. He was inspired by a Darwinian notion of evolution in which only the fittest survive. He proposed that a learning machine's search for a good learning strategy be organized as the breeding of many strategies in a population of candidates, rather than as the construction and refinement of a single strategy. Holland and his students called their searches reproductive plans, but the name genetic algorithms became popular after Holland published a seminal book in 1975 *(3)*.

By the early 1980s genetic algorithms were showing broad promise. The leaders of the field began holding regular conferences every other year. In 1989 David Goldberg of the University of Alabama published a book that demonstrated a solid scientific basis for the field and cited no fewer than 73 successful applications *(4)*. In 1991 Lawrence Davis of Tica Associates published a handbook of genetic algorithms *(5)*. The field has turned into a gold mine of opportunity for exploring biological analogies in computing and information analogies in biology.

Peter J. Denning is associate dean and chair of computer science in the school of information technology and engineering at George Mason University, Fairfax, VA 22030. His internet address is pjd@cs.gmu.edu.

Exploring Parameter Space

For many practical problems in engineering and science the only sure way to find an optimal solution is to search through the entire set of all possible solutions. Such an exhaustive search is described as exploring the total "parameter space" of the problem. In many cases the parameter space is so large that only a minute fraction of it can be explored. The question then becomes: How can one organize the search so that there is a high likelihood of locating a near-optimal solution?

The usual approach is to iteratively refine a trial solution until the refinement heuristic produces no further improvements. Genetic search algorithms take a different approach. Inspired by biological evolution, they cross-breed trial solutions and allow only the "fittest" solutions (those accorded the highest value) to survive after several generations.

In its simplest form, a genetic search works as follows. First, the problem is formulated in such a way that any solution can be encoded in a string of binary digits. Each such string can be assigned a fitness value, based on how well the corresponding problem solution meets some stated goal. Starting with a population of strings, a new population of the same size is generated in two stages, called reproduction and mating. In the reproduction stage, each individual's probability of being reproduced is proportional to the string's fitness. One way to arrange for such proportional reproduction is to create a roulette wheel whose circumference is divided into as many segments as there are binary strings in the population. The length of each segment is made proportional to the fitness of the corresponding string. Reproduction proceeds by spinning the wheel many times, and each time selecting a string to carry forward into the next generation. In this way the reproduction step generates a list of copies of a subset of the starting population. The fittest individuals tend to produce the most copies.

The mating stage simulates the recombination of genetic elements made possible by sexual modes of reproduction. Mating begins with the selection of a random integer larger than zero and less than the string length, defining thereby a crossover point. Two strings are mated by joining the prefix of one string with the suffix of the other string relative to the crossover point. For example, suppose the prefix length is 3 and the two individuals selected for mating are:

```
010 | 110
110 | 011
```

(the vertical line marks the crossover point). Then the binary strings resulting from the crossover operation are:

```
010011
110110
```

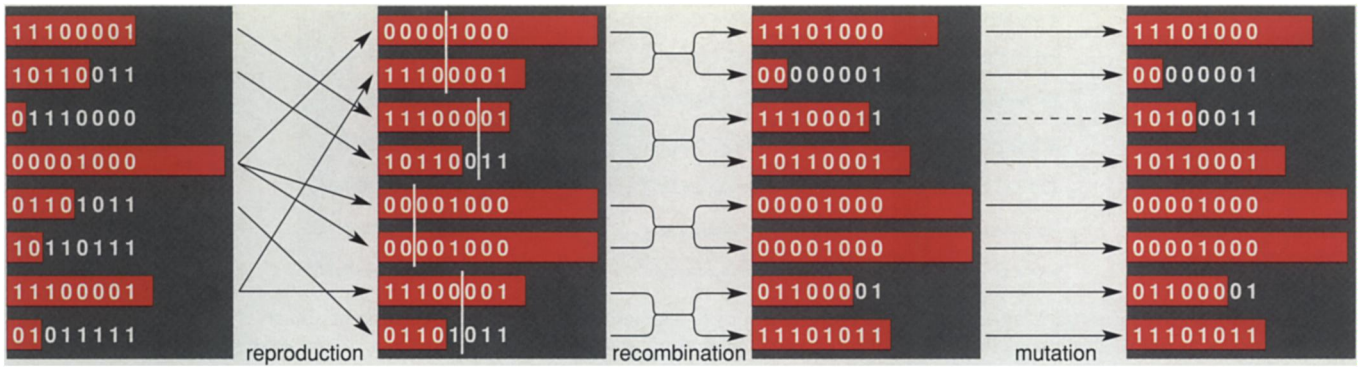

Figure 1. Genetic algorithms employ the mechanisms of evolution to solve optimization problems. Each candidate solution is encoded in a string of binary digits and assigned a "fitness," which is represented here by the length of the red bar superimposed on the string. A population of strings evolves through selective reproduction, recombination and mutation.

Avoiding Local Optima

With any sparse search through a large parameter space there is a danger of converging on a solution that is only locally rather than globally optimal. To avoid such traps, mutations are introduced during the mating stage: Each binary digit has some small probability of being reversed during the genetic recombination.

Kenneth DeJong of George Mason University performed extensive experimental studies of genetic algorithms beginning in the late 1970s. He reports that populations of 50 to 100 individuals taken through 10 to 20 generations have a high probability of including optimal or near-optimal individuals *(6)*. This finding holds for a wide variety of problem spaces. DeJong says that a mutation probability on the order of 0.001 per bit is enough to prevent the search from locking onto a local optimum.

The type of search outlined above works when the encodings of solutions are all the same length and when any binary string defines a valid solution of the problem. However, there are many situations in which some binary strings do not define valid solutions. In such cases, a crossover operation could produce meaningless strings.

As an example, consider the problem of mapping the points of a grid to the nodes of a massively parallel computer so as to minimize the average communication distance between neighboring grid points. This distance is important because in many programs—such as weather simulations—calculating a new value at each grid point requires communication with nearby points. Suppose there are K computing nodes in the machine. An assignment of grid points to machine nodes is encoded as a long vector of integers $(n_1, ..., n_K)$, in which n_i is the machine node to which grid point i is assigned; this vector is a permutation of the integers 1 through K; in other words each integer from 1 through K appears in the vector exactly once. Most crossover operations are likely to produce vectors that are not permutations of 1 through K.

Cases of this kind can be brought into the paradigm of genetic search by defining crossover operators that preserve the validity of the encoding. Here the need is for a crossover procedure that permutes elements of a vector rather than replacing them. One such operator was proposed and used successfully by Ophir Frieder of George Mason University and Hava Tova Siegelmann of Rutgers University *(7)* for assigning documents to the nodes of a multimachine database, a problem similar to the grid assignment problem.

Suppose these two vectors are possible assignments:

1532 | 704 | 968
7409 | 651 | 328

The standard crossover procedure might suggest, say, exchanging the central segments of the vectors (between the vertical lines); this transformation would not be legal, however, since neither vector would then be a permutation of the integers 0 through 9. But there is a method of making the exchange while preserving the permutations. The idea is to use the corresponding pairs of integers within the selected segments of the two vectors to define a series of exchanges that can then be carried out separately within each vector. In this case the pairs are 7:6, 0:5 and 4:1. When these exchanges are carried out in both vectors, the desired crossover is achieved without sacrificing the essential character of the vectors as permutations:

4032 | 651 | 978
6159 | 704 | 328

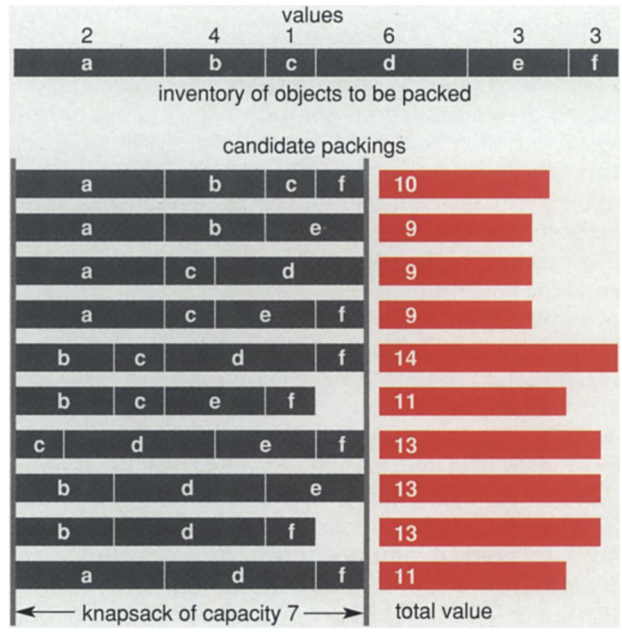

Figure 2. Knapsack problem is an example of an optimization task that can be undertaken by genetic search algorithms. The problem is to pack a knapsack of limited volume with a selection of items that maximizes some measure of value. In this one-dimensional version of the problem the knapsack has a length of seven units, and there are six objects with lengths ranging from one to three. Some possible packings are shown, including one that yields the optimal value of 14.

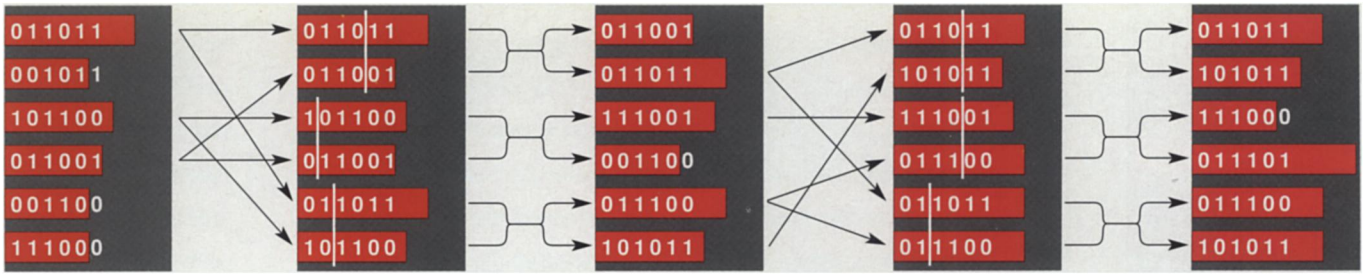

Figure 3. Optimum knapsack packing emerges after a few iterations of a genetic algorithm. Each selection of objects is represented by a binary string; for example, 011011 designates the choices of *b*, *c*, *e* and *f*. Total population fitness rises from 49 to 63, and an optimum solution appears.

The simple operations of reproduction, crossover and mutation on populations are the essence of genetic algorithms. Traditional methods for locating optimal solutions use heuristics that explore the neighborhood around a single trial solution; even when augmented with occasional jumps to distant parts of the solution space, these heuristics tend to get snagged on local optima. By maintaining a multipoint perspective on many regions of the space, genetic algorithms have a much higher chance of locating a global optimum. They do this even when the function defining fitness is discontinuous, irregular or noisy. David Goldberg and Lawrence Davis document these claims empirically *(4, 5)*.

Machines That Learn

Genetic algorithms have an impressive record of progress as a heuristic search method. Their progress in the domain that originally inspired their development, machine learning, is equally impressive.

The objective of machine learning is easily stated: to build a machine capable of performing certain actions even though the builders do not know algorithms for those actions. Examples of behaviors such a machine might learn are walking toward darkness, grasping objects, and recognizing images. Internally, the machine is organized around a controller that receives sensory inputs from one or more detectors and generates actions through one or more output devices. After each action the machine receives feedback about the effectiveness of the action; this feedback is called the payoff. The controller uses the payoff to adjust its internal program and database so that future actions are more likely to produce high payoffs. The payoff function is not known to the machine.

The simplest type of learning machine is an associator between input patterns (from sensors) and output patterns (driving motors). The machine is shown a series of examples, each consisting of an input and a corresponding ideal output. On the basis of the examples the machine adjusts internal parameters of an associative memory so that it minimizes its errors in generating outputs for a given series of inputs. After the training session, the machine is judged by its ability to "predict" outputs in real time. An example of such a machine is an optical character recognizer whose inputs are images of handwritten characters and whose outputs are ASCII codes (the standard numeric representations of alphanumeric data). Genetic searches have been used successfully to identify internal parameters of the associative memory that minimize error during training.

Learning machines of this kind employ a payoff function that is known in advance; the payoff is determined by the difference between the output for a given setting of the parameters and the correct output. A more advanced form of learning machine is needed for the case when payoffs are not known in advance but are received in real time as the machine performs actions. Now the machine's internal structure can be represented by a set of rules telling it how to respond to given inputs. After each action, the machine uses the resulting payoff feedback to modify its rule set so that either an effective behavior is reinforced or an ineffective behavior is dropped. Machines of this type include the robot insects under study at the Massachusetts Institute of Technology and elsewhere, which can learn to walk, hide from light, locate doorways, and hunt for simple targets *(8)*. Although not used in the MIT insectoids, genetic search is under study as a new method of modifying the rule set of such machines when new payoff information is provided.

The power of genetic algorithms derives from their emulation of nature's principle of evolution over generations of a species. In the case of searches for optima, a population of candidates is allowed to evolve over several generations, with the fittest individuals having the best chances of survival. In the case of learning machines, a population of rules evolves over time, and the rules producing the highest payoffs come to dominate the population.

Through most of their history, genetic algorithms were used as optimizers and searchers. They are now well understood in this role. A major shift is under way: Genetic algorithms are being used as the builders of programs inside learning machines. One might speculate about using such machines to design organisms whose genetic codes endow them with desirable characteristics. This is a development worth watching closely.

References

1. Eric S. Lander, Robert Landridge and Damian M. Saccocio. 1991. Mapping and interpreting biological information. *Communications of the ACM* 34 (September): 33–39.
2. Douglas R. Hofstadter. 1985. The genetic code: arbitrary? In *Metamagical Themas*, 671–699. New York: Basic Books.
3. John H. Holland. 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor: University of Michigan Press.
4. David E. Goldberg. 1989. *Genetic Algorithms*. Reading, Mass.: Addison-Wesley.
5. Lawrence Davis (ed.). 1991. *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold.
6. Kenneth DeJong. 1988. Learning with genetic algorithms: An overview. *Machine Learning* 3:121–138.
7. Ophir Frieder and Hava Tova Siegelmann. 1991. On the allocation of documents in multiprocessor information retrieval systems. *Proceedings of the 14th Annual ACM/SIGIR Conference on Research and Development in Information Retrieval*, 230–239. New York: ACM Press.
8. Paul Wallich. 1991. Silicon babies. *Scientific American* 265 (December):124*ff*.