PART II: ADAPTIVE COMPUTATION AT WORK

In our last issue we discussed some research in adaptive computation, most of it at the Santa Fe Institute: What happens when you build models and let them evolve based on internal fitness functions -- reproduction or ability to control resources? In this issue, we focus on more directed use of genetic algorithms (GAs) to solve specific problems, both research and practical, driven by externally imposed fitness functions. We also address a couple of side issues: viruses, and commercial software as an evolving if not living artifact. All of this, of course, refers back to the fundamental Darwinian principle of survival and reproduction of the fittest.

The first section of the newsletter is written by Lawrence Davis, a consultant and actual practitioner in the field of GAs. The second section covers the work of John Koza, an even more practical-minded guy (he founded a company and grew it to sales of $38 million before selling it in 1982) who has now turned to exploring the field of genetic programming (page 17).

Programming by evolution: A brief review

The basic idea of genetic algorithms is to drive the process of evolution electronically. The generic genetic algorithm manages a cycle: algorithms, designs or other possible solutions to a problem are represented electronically, typically as chromosome-like strings or as programs with well-defined substructures that can be swapped in and out. The first generation of possible solutions is evaluated by a fitness function which scores each possible solution. The best of the solutions are saved and reproduced for the next generation. Some of them are combined, in sex-like operations called crossovers, where the first portion of one is combined with the second portion of another to create a new entity possibly combining the best characteristics of both, or perhaps the worst of both. Others are reproduced intact or mutated.

The exact parameters -- how many members in the initial generation and how many generations, the rate of mutation, the rate of crossover and how it's accomplished, the way survivors are selected for the next generation according to their raw fitness scores, how many children per parent and so forth -- are all variable, and define a specific genetic algorithm.

The fitness function itself -- the way fitness is evaluated -- and its scoring, along with the problem-solving elements that make up the chromosomes of the initial population, are

BEST WISHES, DAVID AND RUTHANN!
problem-specific. While genetic algorithms automatically evolve good and better solutions, this is the point where the magic ends. As the examples here show, there's an art to designing good GAs -- mixing populations right, running optimizations in parallel and combining the results, and using GAs in conjunction with other techniques. Beyond that, you need wisdom and creativity in designing the evaluation function, which may be a single function, a simulation using values from the solution or even a set of problems ("fitness cases") for a GA-evolved program to solve. For now, there's a limited body of practice and experience to guide developers. This issue points to some of it.

GENETIC ALGORITHMS AT WORK

Lawrence Davis is the founder of Tice Associates of Cambridge, MA, a consulting firm specializing in the application of genetic algorithms. We invited him to contribute this section of the newsletter because he has 11 years of experience in the field, working for Bolt Beranek & Newman, a Cambridge, MA, high-tech consulting firm, and Texas Instruments before founding his own firm in 1990. So far as we know, he is the world's only commercial GA consultant, although he's surely the first of a growing breed. Earlier, he earned his doctorate in philosophy at UMass at Amherst and spent two years in Morocco with the Peace Corps designing irrigation and drinking water systems (without any help from GAs). He is the author/editor of Genetic Algorithms and Simulated Annealing and of The Handbook of Genetic Algorithms.

This article naturally enough focuses on GA success stories. There are some failures too, Davis acknowledges, but he refrains from citing them by name. As with expert systems and other exotic technologies, Davis says delicately, "many of the implementation problems are due to technology transfer and cultural issues, not to technical problems."

ABOUT GENETIC ALGORITHM APPLICATIONS by Lawrence Davis

John Holland invented genetic algorithms three decades ago. For the next two decades years few GA applications were fielded, and those mostly by researchers already familiar with GA technology rather than with the problems to be solved. In the last ten years things have changed. The number of applications is growing exponentially and the cadre of developers is broadening as GA tools and techniques become more widely available and understood.

What makes a problem appropriate for a genetic algorithm?

Determining whether a problem is a good candidate for a genetic algorithm approach requires both art and science. I take a rule-based approach. A problem may be a good one for genetic algorithms to solve if:

- The problem is an optimization problem. Genetic algorithms are being used in machine learning and artificial life, but I know of no commercial applications along these lines. Optimization is the most highly developed part of the genetic algorithm field.

- The problem is not already addressed by highly refined, domain-specific optimization algorithms. Genetic algorithms are robust algorithms that do well at finding approaches to difficult problems. Unless they are
tailored to a domain, however, they are unlikely to outperform algorithms that have been "living" in the domain for a long time, evolving (through programmers' efforts) to fit the domain's requirements. Domain-specific algorithms may be effectively hybridized by GA techniques, however...

- The problem is one in which there are existing algorithms and heuristics that may be combined with the genetic algorithm. Hybridizing the genetic algorithm with effective algorithms of other types produces offspring with hybrid vigor. The hybrid algorithms combine the global-search and population-management capabilities of the genetic algorithm with the domain-specific powers of the local algorithms, and do better than either side can do alone.

- The problem is one in which small amounts of optimization are worth a lot. Genetic algorithms tend to find small improvements over the results of other optimization algorithms, sometimes after expending greater amounts of CPU time. The best-suited problems are those where small increments of improvement -- 4 to 10 percent, for instance -- have dramatic impact. Examples of such problems include currency trading, scheduling of manufacturing processes and the design of America's Cup entrants. Examples of problems where such benefits are not obtained include optimization of message flow in a network that has significant excess capacity, layout of semiconductor components in a part of the chip to be run in parallel with a slower part, or the design of buggy whips. Long run, use of GAs may come to be a legal issue: "But judge, we did the best we could. We used a genetic algorithm!"

- The problem is one in which straightforward optimization algorithms do badly. With luck, GAs may do better. Such problems may have many local optima far from the global optimum, noisy or discontinuous evaluation functions, or such complexity (NP-hard, for example) that other techniques require unrealistic amounts of time to produce good solutions -- or simply cannot. Typically, in such problems each new possibility sends you all the way back to the drawing board to look for better solutions. Examples include the traveling sales rep problem, bin-packing or truck-loading (add one item, and you may have to rearrange the whole truck), IC board stuffing, and cutting fabric so as to minimize waste (especially tough when it has visible patterns).

The chromosome is a list of numbers

The best understood and most frequently used genetic algorithms solve problems in which solutions are represented by lists of numbers akin to chromosomes. The numbers may be represented as real values or, as was the case in Holland's original work, as strings of 0's and 1's. In fact, these numbers are really just tokens: place-holders or pointers for parameters, instructions or even, for example, different tasks to be scheduled. The genetic algorithm assesses not the list or string itself, but the performance of the design, program or sequence of tasks it represents (just as natural evolution assesses phenotypes, not genes). You could use abcd, 12345, #@%& or even Juan Alice Fred David Ruthann. It's simply that the strings are easier to manipulate. On the other hand, as John Koza illustrates below (page 16), you can even use lists of lists represented in LISP, for more complex, variegated structures such as programs.
Most commonly, as in chromosomes, the range of meanings of each item or variable in the list is determined by its place in the list. For example, the first element determines diameter, the second density, the third temperature and so on, even though they may all be represented by seemingly identical numbers. The range of genetic algorithm applications of this type is broad. The most obvious examples lie in the area of mathematical function optimization, in which one is looking for combinations of variables or parameter values that minimize or maximize the value of the function. In this type of problem, the GA manipulates chromosomes that encode the parameter settings, and the evaluation of a chromosome is the value of the function when the parameters are set to the values in the chromosome’s fields.

**USING HUMANS TO EVALUATE CHROMOSOMES**

Most GA applications use mathematical functions or computerized simulations to evaluate chromosomes, but this is not a necessary feature of GAs. In an experiment funded by the National Institute of Justice, Craig Caldwell and Victor Johnston, psychologists at New Mexico State University, built a GA system that uses human feedback.

The problem was to assist witnesses to criminal acts in producing images of the perpetrators. Witnesses are often able to recall the face of a perpetrator to some degree. However, their attempts to describe the image of a perpetrator, or to describe the differences between a trial image and the true image, are not as accurate as their recollections. Caldwell and Johnston’s system reduces loss of information between visual processing and verbalization. Their system encodes features of faces on virtual, electronic chromosomes. The genetic algorithm manipulates these encodings — lists of numbers representing eyebrow height, nose shape and so forth. The witnesses look at the images that these encoded lists generate and rank each image for its similarity to the face of the perpetrator. In effect, the witnesses perform the evaluation function. An initial population of randomly generated faces evolves over the course of generations to produce a face remarkably similar to the original one. (The system is not yet in use in any real cases, for procedural and financial reasons rather than technical difficulties.)

**Designer genes**

By extension, we can solve design problems in which the design can be expressed as the selection of a set of parameter settings, but the result is much more complex than a single function because of the interaction of the design elements. Typically, to evaluate such systems you need to simulate them, as with the air-injected hydrocyclone described across. In fact, a great many difficult optimization problems have solutions that can be represented as lists of numbers. Training a neural network of fixed architecture is such a problem: the solution consists entirely of finding weights for each synapse, which can be represented as lists of numbers.

Tuning the performance of expert systems can be such a problem; after human experts have produced the logic of the rules in the system, a genetic algorithm can be used to optimize critical threshold values or probabilities. In this type of application, the genetic algorithm searches for co-adapted sets of threshold values (the thing it is good at) and the human produces the higher-level logic of the rules (the thing some humans are good at).
CASE STUDY: THE AIR-INJECTED HYDROCYCLONE

The separation of precious metals from worthless rock is an important part of the hard-rock mining process. The machinery that accomplishes this task today is either a hydrocyclone, a centrifuge that uses specific gravity to accomplish the separation; or a flotation device, an air-injection chamber that uses hydrophobic properties of worthless rock to float it to the surface where it can be skimmed off and discarded.¹

Don Stanley, a research chemist at the Tuscaloosa Research Center of the United States Bureau of Mines, had patented an idea for combining these two techniques into a single device that would be smaller, cheaper, and better performing than the single-process devices in use today. However, he needed a prototype to have any hope of persuading manufacturers of mining equipment to license the patent. This was no trivial requirement. It’s an imposing challenge for a never-built machine whose empirical properties are not fully understood to beat the small size, low cost and high performance of machines that have been fielded, tested and redesigned for decades. But Stanley could not afford to experiment with a series of prototypes. Instead, in 1988 he went to the University of Alabama in search of someone to program a simulation of air-injected hydrocyclone performance reliable enough to eliminate the need for multiple prototypes.

The University of Alabama was no idle choice. UA’s David Goldberg, a student of John Holland’s and a leading GA man in his own right, brought Stanley and the Bureau of Mines together with Chuck Karr, a graduate student working in fluid dynamics.

Rigged rock rejecter

Over the space of a year, Karr developed a software simulator that accurately predicted the separation properties of air-injected hydrocyclones based on the settings of 11 design variables. The point here is that the real engineering problem was not the GA itself, but the prototype that could evaluate the algorithms as part of the process. With this simulator in hand, it was then a much easier matter to consider various settings of the design parameters. What was a good setting for the diameter of the intake? What was a good feed rate for the slurry? And all of these in combination?

Given a set of values of these variables, the simulator could predict performance. Karr started with an initial population of 21 randomly generated 51-bit chromosomes, each containing encodings of 11 design parameters (several bits per parameter). As it happened, each encoded an extremely low-performance hydrocyclone. But over 150 generations and through a series of runs, with the best chromosomes from early runs as "seeds" for future runs, they evolved to produce exceptional designs. Early on in the evolutionary

¹ Little-known facts department: By sheer luck, most kinds of worthless ores, in combination with certain chemicals (different for each kind), tend to seek air, whereas most valuable metals stick with water. Thus when bubbles are forced through the slurry, they attract the particles of ore, which then float to the surface. It's not software, but it helps the example make sense!

Release 1.0 28 May 1992
process, bubble size revealed itself to be important and rapidly converged to a single value. Chromosomes containing two or three other settings well-adapted to the bubble-size choice began to emerge, producing better and better simulated performance. These chromosomes recombined, producing new combinations with even higher performance. Mutations refined the performance of these combinations, until the algorithm converged on a combination of parameter settings that solved the design problem well.

Black art

Karr's approach was not pure: He did a number of runs, combined the best of those runs into new runs with some randomly generated chromosomes added in, in a heuristic process that has no formal rules as yet. (GAs still benefit from some human art or feel that can't yet be defined explicitly.)

The bottom line is that Karr's genetic algorithm technique outperformed traditional methods and even more modern techniques such as gradient descent (akin to hill-climbing), which find local but not global optima. The design produced by the genetic algorithm was predicted to yield 1 per cent of separation more than the design produced by any other algorithm, which was already better than that currently achieved by conventional devices. This difference is very significant in the mining industry, where an increase of one per cent in yield translates very nearly into an increase of one per cent in profit once fixed costs are covered.

Better yet, the air-injected hydrocyclone designed by Karr's genetic algorithm was built and achieved the predicted level of yield when field-tested. It is now the subject of discussion with mining equipment manufacturers, undergoing an extended process of technology transfer.

You ain't seen nothing yet

Karr is not the only developer to use the genetic algorithm to design devices with a chromosome representation. Several General Electric and Rensselaer Polytechnic researchers applied genetic algorithms and other conventional algorithms in a set of sequential, interoperating optimization modules. They used it to design an aircraft engine turbine, which is now being built at a GE plant. They have since developed a generic GA tool called Engineous and used it for the design of a wide range of products including motors, power generation plants, nuclear power satellites, electronic circuits, space superconductor generators, transformers, utility planning and operation, nuclear fuel, light bulbs, plastic bottles and combat simulation. In many cases, GAs have helped to produce unconventional and dramatically better designs, some of which are going into production.
PRODUCT: THE GENETIC ALGORITHM MEETS THE SPREADSHEET

The genetic algorithm tool most widely used today uses the facilities of a spreadsheet for evaluation of the algorithms, and runs its own subroutines, implemented as a Dynamic Link Library under Windows or a discrete code module on the Mac for the actual genetic algorithm functions of generation, recombination etc. The tool -- Evolver from Axcelis of Seattle -- is designed to be used on pcs in conjunction with Wingz or Excel spreadsheets.

Often the most difficult part of a genetic algorithm to design, and the most variable from problem to problem, is the evaluation function. Evolver allows users to create their evaluation functions in a familiar spreadsheet. Once installed, the Evolver package shows up on the spreadsheet menu. The user sets mathematical relationships among cells in the usual spreadsheet fashion, or uses certain templates provided with Evolver. When the user selects Evolver from the menu, a dialogue box (see illustration) pops up asking the user to indicate the cell to be optimized, which cells are variable and how, and the termination conditions. Thus, for example, you can vary the individual cells that total to a certain budget amount while keeping the total constant, as in the sample shown.

Evolver trades on the fact that many spreadsheet applications are de facto evaluation functions, and can easily be manipulated by the genetic algorithm to produce global optimization instead of the relatively puny backsolving functions some spreadsheets offer internally. Basically, it creates chromosomes with elements representing possible values of the variable cells, and evaluates them by running the spreadsheet with those values to find the value of the target cell, which it then compares to the user's goal. Using standard GA techniques, it tests and selects improving combinations of the variables until the termination conditions are reached.

More than 600 copies of Evolver have been sold, and its users have applied it to problems ranging from the grading of rare coins and the estimation of risks incurred from contact with hazardous substances, to high-dimensional curve-fitting for power stations estimating load demands. Evolver has even been used to train neural networks, with back-propagation algorithms incorporated in the spreadsheet equations.
The modern spreadsheet: now, with GAs!

Evolver takes care of the parameter settings, population sizes and other features of genetic algorithms that occupy the minds of developers producing systems for more specialized purposes. The Evolver genetic algorithm is robust enough to perform reasonably well across varied domains. The thread that ties them together is the fact that, to a GA, the problems all look the same. In each case, the data structure to be manipulated is a list of numbers, and the operations of mutation and crossover do not change from application to application. The mysteries of the user's spreadsheet evaluation function do not trouble the genetic algorithm as it does its work.

In the next release due this summer, however, Evolver will have additional capabilities to handle a broader variety of specific problems. Although Evolver for now will still be marketed as a spreadsheet add-on and use spreadsheet cells for data, more and more of its power will migrate to its own library of capabilities and templates for users to express things such as schedule dependencies and constraints through problem-specific Evolver interfaces. Call this Trojan-horse marketing, where users are introduced to GAs through the familiar face of a spreadsheet. (We could certainly imagine a module attached to a project management tool or a database tool sometime in the future.) Users will be able to select from a list:

- **recipe/budget.** The items must amount to some total and generate maximum revenues or minimum costs.

- **recipe/distribution.** Resources must be combined to produce maximum output. For example, you want to maximize production of a variety of products from a variety of plants with specific capabilities and capacities, using a variety of raw materials of various costs.

- **grouping.** Items must be combined into groups. For example, a securities firm is using Evolver to group individual securities into same-value packages with similar risk and yield characteristics.

- **order/schedule.** Items must be scheduled among a variety of activities, as in assigning professors and students to classes, or scheduling break-out sessions into rooms with different AV set-ups at an industry Forum, with a soft constraint to avoid scheduling competitors into the same time slots.

- **order/project.** Tasks of varying lengths must be scheduled in sequence, reflecting a variety of dependencies and resource requirements.

- **algorithm.** This is a broad category, including examples such as optimizing SQL queries or creating more efficient equations. (Cf. John Koza's work, page 16.)

Of course, the user can also combine problem types, or add his own through an API. The second version of Evolver also lets the user fiddle with the GA parameters, such as crossover and mutation rates.

Release 1.0 28 May 1992
ORDER-BASED GENETIC ALGORITHMS

Now we consider the second most common form of genetic algorithm, in which solutions can be represented as permutations of a list of elements, or order-based genetic algorithms. The feature defined by each section on a chromosome in the approach described above is generally different for each location. It is also fixed at that location, although the values vary for each field. For example: The first field encodes the appetizer (to use the restaurant example), the second the drink, the third the salad and so on. What varies are the values of the fields, which together generate a total fitness score for each menu.

In an order-based genetic algorithm, by contrast, what varies is the order of the elements of each chromosome; each value has the same meaning wherever it is in the sequence. Here you might say: No dessert until you've finished your main course; or, let's try salad after the entree, in the French style. The items are the same, but the order in which they are handled is different -- which usually ends up affecting details of each job and the overall efficiency of the two sequences.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>10</td>
<td>Y</td>
<td>red</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>12</td>
<td>N</td>
<td>blue</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>14</td>
<td></td>
<td>green</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
<td>18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

genetic chromosome

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>10</td>
<td>Y</td>
<td>red</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>12</td>
<td>N</td>
<td>blue</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>14</td>
<td></td>
<td>green</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
<td>18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

order-based chromosome

Note that the values in the genetic chromosome on the left may be of different types, even though they will ultimately be encoded as similar-looking numbers. In the order-based (permutation) GA on the right, any item can appear anywhere in the sequence.

The underlying meaning of these order-based strings bears little relation to the natural encoding of biological chromosomes, but they have all the "visible" characteristics of a genetic algorithm; for example: E C A D B or B E A C D (except that usually no individual value is repeated). The GA evaluates the results of permutations much as it evaluates various combinations of parameters. To the GA, it's all chromosomes and fitness scores. Mutations (local permutations of the orderings on a chromosomes) and crossovers (combinations of the relative orderings on two parent chromosomes) are similar in flavor but not in detail to the mutation and crossover operators in the more familiar chromosome-style GA. Typically, there is a grammar of variation that restricts the allowable sequences to those that have one of each number (unless some jobs are intended to be performed several times).
Division of labor

Genetic-chromosome-style GAs are good at mathematical function optimization. Order-based genetic algorithms are good at another large and important class of optimization problems, that of combinatorial optimization. Not all combinatorial optimization problems can be solved using order-based GAs, but a significant and commercially important subset of them can be. Examples include the venerable traveling sales rep problem, in which tours of a graph are encoded as orderings of the cities on the chromosome; graph coloring problems, where color schemes are encoded as permutations of a list of graph nodes; and scheduling problems, in which the chromosome contains a permutation of the list of jobs to be scheduled and a simple scheduler does the actual scheduling of the jobs based on their order on the chromosome.

The order-based approach to optimization originated in the early 1980s in work done by a group I was a member of at Texas Instruments. Because this approach is newer, it has led to fewer applications than the chromosome approach, but this situation should change as developers -- and users! -- become aware of its power. At Texas Instruments we used it to solve some problems in semiconductor layout, finding no other algorithm that produced designs as good. At Colorado State University, a group led by Dr. Darrell Whitley is using this approach to schedule release of orders for a Coors warehouse, so that beer moves more quickly from breweries to sales outlets. In fact, it is using specific techniques developed at the SITS lab, described below, which have improved scheduling times from six hours to about four minutes. The result is less inventory to finance, and fresher beer for customers. This system is in intermediate stages of adoption at Coors.

At US West in Boulder, Colorado, I am collaborating with a simulation and modelling group led by Dr. Anthony Cox. We are experimenting with order-based GAs to solve problems of message routing in large telephone networks.

CASE STUDY: THE SITS LAB SCHEDULER

The two System Integration Test Station laboratories at the US Navy's Point Mugu Naval Airbase manage several F-14 airframes linked to monitoring equipment and simulation environments that model the effects of flying the aircraft, using radar and employing electronic countermeasures. Developers of systems for F-14 aircraft use the SITS labs to test new hardware and software systems targeted for installation in working aircraft without endangering an aircraft in actual flight or using valuable flight time.

Schedules for the facilities are produced on a weekly basis. Typically, the list of tasks requested for scheduling exceeds the time available, so every hour of test time is important.

Scheduling the labs is complicated by hard and soft constraints. Hard constraint violations result in an illegal schedule. For example, a particular flyby of an aircraft transmitting to the test system might not be possible on Wednesday; one hour of rollout (moving the airframe to an outdoor position overlooking the Pacific) is required for a test involving live radar; two experiments requiring particular hardware cannot be carried out simultaneously; and so forth.
Soft constraints can be violated, but are suboptimal. For example, a developer might prefer Wednesday morning; high-priority tasks should be placed in the schedule at the expense of low-priority tasks; and so on. The goal is to produce a legal schedule that maximizes the number of high-priority tasks scheduled while minimizing soft-constraint violations.

Previously, the lab was manually scheduled by a human expert who had been doing so for nearly 20 years. The cycle began each Thursday when the expert disappeared into his office with sheafs of paper listing task requests and constraints. By Friday, he would emerge with the next week's schedule.

Sam Wilson, head of the Avionics Lab division which runs the SITS labs, wanted to automate this scheduling process for three reasons. First, the human expert's abilities were stretched by the problem as it currently existed, and the amount of activity in the SITS labs was increasing dramatically. Second, the human expert was nearing retirement age and nobody else could approach his level of performance (performance went down whenever he went on vacation). Third, things would come up and schedules often needed changes during the week, a problem well-suited for automatic rescheduling. Wilson turned to Bolt Beranek and Newman for help.

First assist the expert...

As a first step, computer scientist Gilbert Syswerda and his team at BBN (to which I was an occasional advisor) developed a computer-based, interactive schedule editor that could be used by the human expert or other schedulers to produce schedules much more quickly and accurately than with stacks of paper. Once requests and constraints were entered into the computer, the tool displayed a blank week along with the task list. When the human scheduler selected an activity from the list to schedule it, the week's time slots changed color. Time periods where hard constraints prevented the selected activity from being scheduled turned red, time slots with soft constraints against the activity showed yellow, and time slots especially good for the activity were green. Alternatively, the user could select a time slot, and tasks prohibited by hard constraints from occupying that slot turned red, tasks with soft constraints against the slot turned yellow, and especially appropriate tasks turned green. With this tool, the human expert was able to build schedules in an hour that had previously taken a day.

Then replace the expert...

The final phase was to reduce scheduling time -- and human effort -- even further through computer generation of high-performance schedules. Syswerda tested several implementations of order-based GA techniques on the scheduling task and eventually settled on the approach described below. This is what helped to improve performance at Coors, and BBN has filed a patent application on it. (At first glance it might seem to make sense to use a literal encoding: The first time slot is the first field on the chromosome, and so forth, and each task is assigned a numerical ID. It "ought" to work, but it doesn't.)

...with an automatic system

The SITS system works in two sections. The GA generates ordered lists of tasks to be scheduled, but the scheduler, part of the evaluation function as
broadly defined, actually schedules the tasks, taking each from the GA-generated list in order and placing it in the first legal spot on the schedule. The schedule builder is a dumbed-down version of the scheduling tool built for the human expert -- without the interface, of course. It makes no choices and implements no priorities or soft constraints. As Syswerda puts it, "If we put intelligence into the schedule builder, different lists would produce the same schedule and would take information away from the GA. The point is to let the GA do its work of producing the best sequence of tasks to be scheduled." The task of the GA is to find an order for the tasks that leads this simple scheduler to construct a good schedule. (In fact, it intersperses low-priority tasks with important ones, whereas a more "clever" approach would probably schedule the important tasks first. In practice this approach seems to work best in filling the schedule optimally, as it might for packing a space with objects, says Syswerda.)

The constraints and priorities are expressed only in the evaluation function which rates the schedule produced by each task list. The schedule evaluator takes as input a completed schedule and evaluates it, trading off the scheduling of high-priority tasks against the degree to which soft constraints and priorities have been violated. Given a schedule as input, the schedule evaluator returns a single number, a measure of the schedule's worth. Each ordered list produces a single schedule with a fitness score.

The system typically considers about 3000 schedules before the population converges on a near-optimal schedule. On a TI Explorer LISP machine, the whole process takes an hour or two a week; rescheduling an afternoon takes only a few minutes when someone cancels or some equipment goes down. The system is currently in use, under the direction of the human expert. He has much more free time now. (A second version for the second of the two labs will run in C++ on a SPARCstation.)

Mindless manipulator

In the context of the genetic algorithm, evaluation of a chromosome -- a list of tasks -- is carried out by using the schedule builder as a decoder that takes a permutation of the list of tasks and creates a schedule from it, and then using the schedule evaluator to return a number that is the chromosome's evaluation. The combination of a decoder that places tasks into a partially-scheduled week, an evaluator that mimics the human's preferences among different schedules, and a genetic algorithm that searches for orderings of tasks that result in high-performance schedules is highly successful. The genetic algorithm has no idea that it's manipulating tasks in the context of a schedule. It is simply manipulating a population of permutations with various performances, attempting to breed permutations with evaluations as high as possible.

This case study is typical in that the most time-consuming part of the genetic algorithm to create is its evaluation function -- which here includes not just the evaluation of the schedules, but the generation of the schedules from the ordered lists of task. In the SITS lab application, the evaluation function was carefully crafted to mimic the way a human might think about the schedule.

BBN's Systems & Technologies is working on other applications of its expertise in this field.
VARIABLE-LENGTH CHROMOSOMES

Most genetic algorithm applications have been built using the genetic chromosome representation or the order-based representation, but there are many other ways to represent solutions in genetic algorithm optimization, and with time and practice these representation techniques will become standards like their predecessors.

Many of these new representations involve chromosomes of variable length. Later in this issue, Esther Dyson discusses John Koza's genetic programming systems which manipulate tree structures encoding LISP programs. Koza's approach has led to some interesting successes on machine learning problems that are used as research benchmarks.

John Grefenstette and his group at the Naval Research Laboratory have had a some impressive successes with SAMUEL, a system that encodes production rule sets on chromosomes. (Samuel is named in honor of Art Samuel, who wrote a checkers-learning program back in the Fifties.) Rule-based expert systems have no restrictions on the number of rules they contain; SAMUEL chromosomes likewise have no fixed length. Grefenstette and his collaborators have been clever about clustering rules that work together so that they don't get destroyed during crossover, and about allocating credit to rules for successful runs. Their results include laboratory systems that control autonomous vehicles in simulations involving evasion and pursuit, movement through mine fields and other hostile environments.

GENETIC ALGORITHMS AND...

Most of the GA systems we have considered so far have operated in conjunction with a variety of other tools and algorithm-optimization techniques. Generally, significant benefits can be obtained by combining genetic algorithms with other algorithms; usually, the hybrid offspring do better than either of their parents. This technique is important, and in my experience it has produced the best optimization algorithms we know of for a wide variety of problems. Since it involves combining two algorithms with different characteristics, and since the characteristics of algorithms on different domains tend to differ widely, there is much more to learn about creating such algorithms than about the more standard genetic algorithms we have already considered.

Hybridization of genetic algorithms is a wonderful development in the history of the field, principally because there is a deep resonance between the principles of evolution that underlie the field and the way that the algorithms within it are themselves evolving. In 10 or 20 years we are likely to see optimization and machine learning algorithms routinely employed in real-world applications. Their GA and other antecedents will blur together through years of hybridization so that chromosomal boundaries can be distinguished only by historians. (We ourselves are all just part of a virtual fitness function working on the population of GA techniques.)

...neural networks

Genetic algorithms and neural networks have already been hybridized in many ways. On June 6 a day-long conference on Combinations of Genetic Algorithms and Neural Networks (COGAN) precedes the 1992 International Joint Conference
on Neural Networks in Baltimore (page 27), and GA and neural network conferences generally have at least one session devoted to such combinations.

One advantage of combining genetic algorithms and neural networks is that their search techniques are complementary. Neural networks tend to search through gradient-descent techniques, which follow local gradients from a random starting point to an optimal configuration. Genetic algorithms don't follow gradients, since they deal with discontinuous spaces, but they are good at global search.

One natural approach is to use genetic algorithms to find high-performance neural network architectures for specific problem types, encoding the architecture on a chromosome and evaluating chromosomes by running a network of the type encoded on the target problem. Design of neural networks can be as hard as training them, and the results of a GA approach seem positive.

Another common tack is to use GAs to search for high-performance weights on links in networks of fixed architecture with discontinuous evaluation functions. Darrell Whitley at U of Colorado and his students are among the best-known advocates of this approach.

...and fuzzy logic

The 1991 International Conference on Genetic Algorithms presented three papers on the combination of genetic algorithms and fuzzy logic applied to control problems. In one, Chuck Karr (page 5), still busy after his hydrocyclone adventure, has created a prototype fuzzy-logic system for controlling pH in mining applications. He uses a genetic algorithm operating on chromosomes that encode the parameters of the fuzzy-logic set membership functions (e.g., the fuzzy criteria for acidity). The GA is capable of adjusting to changes in the mining environment, such as drift in the acidity of chemicals added to a solution, while the fuzzy logic portion of the system remains constant.

...simulated annealing

There has been much discussion lately about the relative performance of GAs and simulated annealers. They are similar in that both use stochastic techniques inspired by natural processes in order to solve real-world problems.

In my experience, the natures of the evaluation function and the mutation operator tend to determine which kind of algorithm is more effective in optimization. Simulated annealers are most successful when the impact of the mutations they introduce can be evaluated locally, without global recomputation of the evaluation function. The overhead of global re-evaluation associated with crossover typically outweighs the benefits of crossover for problems of this type. But if you need to perform a global re-evaluation anyway, then GAs give you a better search technique at little added cost.

So far no one has studied the effects of weighting the probability of crossover in a genetic algorithm inversely to the time taken to evaluate the result of crossover. If this is done, we might see genetic algorithms become more similar to simulated annealers, in that crossover probabilities (and resultant large-scale evaluations) would be greatly diminished. In effect, the solution approach would adjust to the cost of evaluating solutions.
PROMISING AREAS FOR GENETIC ALGORITHM APPLICATIONS

Where will we find the next areas of commercial activity in genetic algorithms? There are many obvious candidates. Among them:

Financial management. There are already a number of applications in this area. KiQ of London markets a loan evaluation application that uses a GA component. The April issue of Release 1.0 discusses a system produced by Brian Arthur, John Holland and Richard Palmer that simulates the behavior of trading agents on a simulated market with genetic algorithms to evolve trading strategies, which may well lead to real-world applications. I have heard of several currency-trading systems that use GAs to select characteristics of sequences of financial data in order to predict their future behavior and to trade on that basis. Perhaps the best example of this is an application produced by Andrew Colin of Citicorp Investment Bank in London. Colin uses a GA to build algebraic combinations of leading indicators and features of financial data streams. These combinations are fed into a neural network that guides trading decisions. Other developers are not always so willing to discuss their methods, but there seem to be more and more of them every year; the amounts of money traded under the control of such algorithms amaze the uninitiated observer (who can't be specific).

Design. We have looked at some examples of design with genetic-chromosome GAs. GAs will be an excellent tool for design of all kinds of things, including especially all kind of electronics, from integrated circuits and boards to high-level network layouts.

Scheduling. We have looked at some examples of scheduling with genetic algorithms. As factories and job shops become more computerized, one or more of the leading commercial scheduling packages should soon include GA components (or at least add-ins such as Evolver) to optimize manufacturing schedules. These are domains in which a little optimization can be worth a lot, and we already know how GAs can produce such optimization.

Molecular engineering. At the 1991 International Conference on Genetic Algorithms, seven conference attendees who had never before met discovered that they were all interested in the use of genetic algorithms to investigate problems of molecular conformation and design and are now keeping in touch. Others are joining them. Genetic algorithms are well-suited to solve some of the myriad optimization problems in molecular engineering.

---

Release 1.0 is published 12 times a year by EDventure Holdings, 375 Park Ave., New York, NY 10152; (212) 758-3434. It covers pcs, software, CASE, groupware, text management, connectivity, artificial intelligence, intellectual property law. A companion publication, Rel-EAST, covers emerging technology markets in Central Europe and the former Soviet units. Editor: Esther Dyson; publisher: Daphne Kis; circulation & fulfillment manager: Robyn Sturm; executive secretary: Denise DuBois; editorial & marketing communications consultant: William M. Kutik. Copyright 1992, EDventure Holdings Inc. All rights reserved. No material in this publication may be reproduced without written permission; however, we gladly arrange for reprints or bulk purchases. Subscriptions cost $495 per year, $575 overseas.

Release 1.0 28 May 1992
Despite some advice from Lawrence Davis and a fair amount of scientific research, the people described above are working in uncharted territory. There's no shelf of handbooks, no place to be an apprentice, no compendium of success stories, few role models. In the very long run, it will all boil down to the same issues as AI -- return on investment, interoperability with existing tools, project management skills and the like. Meanwhile, there's a large body of practice waiting to develop. Still, the world of genetic algorithms in general is well explored in comparison with the subset of genetic programming. For a long time, one book will probably fill the gap and take its place next to every genetic programmer's computer: John Koza's "Genetic Programming: On the programming of computers by means of natural selection and genetics," due this August from MIT Press.

Koza's book is neither theoretical nor commercial, but more of an argument from examples that manages to impart a great deal of practical wisdom along the way. He makes an important distinction between genetic algorithms and genetic programming. The standard GA approach uses selection (genetic algorithms) to act on chromosome strings, but generally produces results whose form is fairly close to what the system started with. This is not inherent in the genetic algorithm itself, but in the practice of representing the individuals that are generated and evolved as strings (like chromosomes), with the bits corresponding to instructions. The length and overall structure of the string individuals is fixed from the start. During crossover, the strings may exchange sequences, but they're generally of the same length.

In genetic programming, by contrast, you work with trees of varying sizes and shapes: A single expression, or node, could be replaced with a complex tree of several branches (subroutines) during crossover (page 21). The branches and nodes in a tree indicate the natural points for effective crossover. Thus the structure and length of the resulting program are not predetermined; they too comprise attributes to be evolved. Of course, you could do this with strings as long as you could divide them properly into coherent sections or clauses; representing the programs as trees with nodes and branches simply makes the problem easier to conceptualize, represent and execute.2 These trees are the parse trees used internally by every compiler, but which the programmer usually never sees.

While chromosome-like GAs solve specific problems one at a time -- transforming a particular list of tasks into a schedule or designing a single model -- genetic programming creates a single program to solve multiple problems of a similar nature, problems that are represented (but not fully covered) by the test problems (fitness cases) used to guide the program's evolution. In other words, GAs catch fish; genetic programming makes fishing poles. Programs have the benefit of far more flexibility. To quote from Koza's book:

"...existing methods of machine learning, artificial intelligence, self-improving systems, self-organizing systems, neural networks and induc-

2 Somehow this all fits in with Koza's thesis on induction of grammars (page 24); here's a man who understands the importance of structure -- and the flexibility it allows when it's explicit.

Release 1.0 28 May 1992
tion do not seek solutions in the form of computer programs. Instead, existing paradigms involve specialized structures which are nothing like computer programs (e.g., weight vectors for neural networks, decision trees, formal grammars, frames, schemata, conceptual clusters, coefficients for polynomials, production rules, chromosome strings in the conventional genetic algorithm, and concept sets). Each of these specialized structures can facilitate the solution of particular problems... [but] human programmers do not regard these specialized structures as having the flexibility necessary... If we are interested in getting computers to solve problems without being explicitly programmed, the structures that we really need are computer programs."

The techniques listed above are valuable components and may be part of a program, but they lack the flexibility of true programs, with hierarchies, structures, subprograms that can be reused, etc. The structure of the answer is part of the answer, not part of the problem specification.

The basic idea is that you're searching a large space of possible programs for the right one (or few) that might address the problem set. You can't just do a random search; you have to generate the possibilities nonrandomly and test them against the problem. The early attempts give guidance and feedback for further attempts; the assumption is that even partially successful attempts contain some things worth keeping, and other things worth abandoning. Hence the tactic of combining parts of the successful answers.

Heresies

Of course, every experienced programmer reacts to the whole concept of genetic programming with alarm, since it violates seven basic precepts of science and programming: correctness, consistency, justifiability, certainty, orderliness, parsimony and decisiveness.

Basically, genetic programming is pragmatic. It's not certifiably correct, or rational, or orderly. It comes up with an answer by hook or by crook. To use just one example, take a problem where the solution is \( ax^2 + bx^5 \). A solution such as \( ax^2 + .000000001x^3 + bx^5 \) is not correct in any scientific sense, but it generally works; the error is much smaller than your average computation or measurement error. Such an answer is certainly useful in solving the problem, but in scientific terms it would be wrong, excessively complex (unparsimonious), unjustifiable, disorderly and so forth. And it's probably not certain, either. It might just as easily have been \( ax^2 + .000000002x^3 + bx^5 \).

In the same way, of course, we have a lot of unnecessary sequences in our genes. We also have deviations and variations from eye color to appendices (for solving some earlier problem no longer relevant to us), to deficiencies of character, which are probably counterproductive. It's the entirety of the solution that counts. And it is not necessarily optimal.

AI men of straw

Traditionally, human intelligence has been the main source of feedback. You write a program, look at it (or test it, if you're that far along), figure out what could be improved, and try again. "Anyone who has ever written and debugged a computer program probably thinks of programs as very brittle,
nonlinear, and unforgiving and probably thinks [it] unlikely that computer programs can be progressively modified and improved in a mechanical and domain-independent way," Koza acknowledges at the outset of his book.

Question: What practical use is genetic programming in building, say, a spreadsheet or a database application?

Probably very little, but those after all are two unsuitable kinds of problems: The spreadsheet is a general tool, consisting of myriads of individual programs and routines -- and an even larger portion of user-interface and environment-specific code for file formats, output devices, etc. (Just compare the size of 1-2-3 with any current Windows spreadsheet.) You could use GAs to optimize parts of a spreadsheet tool, or to evolve specific models (see Evolver, page 7), but it probably doesn’t make practical sense to evolve one.

On the other hand, building a standard database/transaction application is pretty simple nowadays, once you know what you want. If you can specify the problem, you can probably use a variety of tools to generate the application automatically, with all the necessary accoutrements of transaction security, pretty interfaces and the like.

But beyond these, there is a large (infinite?) number of one-off problems that require a solution and were not previously cost-effective to solve.

Moreover, people may confuse genetic programming with AI, another once-heralded solution to a variety of problems. Indeed, if you look at all of AI, with its alleged flexibility, conditionality, fuzziness and all the other terms applied to it, it’s hard not to anticipate disappointment for genetic programming. AI works very well for specific problem domains explicitly stated: rules for granting credit, for controlling power plants, for determining fares. Neural nets can recognize patterns, but the patterns must be presented in the proper way.

"AI programs" are indeed brittle, unforgiving and rigid, for all their flexibility within their limited domains. Moreover, writing these programs is the real challenge. Each must be written specifically for its problem set (although shells, objects, rule sets and templates can certainly be reused). Their flexibility, based on things such as synonym lists and additional rules, is built in by hand. And frequently, such additions can cause breakdowns elsewhere, as when a new rule has unexpected side-effects. But this is all a fundamentally irrelevant concern: AI comprises a variety of kinds of programs/applications, such as expert systems, case-based reasoning, and natural-language systems using a variety of approaches.

By contrast, genetic programming is a kind of programming, a way of generating all kinds of programs. AI is basically an attempt to imitate or implement human reasoning and other intellectual techniques, whereas adaptive
computation is, in extremis, an attempt to emulate the process that produced human intellectual capacity. The issue is finding a domain-independent way to search the space of all possible computer programs to find the right program (or one close enough) for a given kind of problem. Obviously, there are some tricks. In the same way, nature has searched all possible forms of life through evolution to find the ones capable of coexisting on this planet. Obviously, it didn't try them all out either...

Koza's two major points

Remember these for the quiz!

- A wide variety of seemingly different problems are all, underneath, problems of program induction. That is, they require "the discovery of a computer program that produces some desired output when presented with particular inputs." He addresses the point by listing a variety of problems, from planning, game-playing strategies, symbolic regression, optimal control and the like, and describing them in terms of a matrix of inputs, outputs, program elements.

- Genetic programming is a useful, broadly applicable way of doing program induction. It can search the space of possible computer programs to find the program(s) to solve the particular program. Moreover, it can produce the requisite program structure.

This point takes the rest of the book to illustrate, as he describes 68 out of a total of 112 experiments performed over the past three years. The book is not an argument so much as a broad survey of examples. In fact, jokes Koza, one reviewer took so long over the book that it grew almost 200 pages in the meantime, from about 45 examples to 68. None of these experiments is conclusive or unique, of course, but each involves setting up the problem and producing solutions through genetic programming. As he notes:

"The reader may, at some point, come to feel that the examples presented from numerous fields in this book are merely repetitions of the same thing. Indeed, they are! That is precisely the point. When the reader begins to see that optimal control, symbolic regression, planning, solving differential equations, discovery of game-playing strategies, evolving emergent behavior, empirical discovery, classification, pattern recognition, evolving subsumption architectures, and induction are all the same thing and when the reader begins to see that all these problems can be solved the same way, this book will have succeeded in communicating its main point: that genetic programming provides a way to search the space of possible computer programs for an individual computer program that is highly fit to solve a wide variety of problems from many different fields."

The framework

Koza has a helpful habit of describing things with tables and lists. Genetic programming is such a broad, widely applicable field that at first it seems impossible to say much concrete about it, or other than to provide examples. However, the problems have a number of commonalities, starting with a basic way of describing them. Each has an input and outputs, and a computer program that transforms the inputs into outputs. For example:

Release 1.0	 28 May 1992
<table>
<thead>
<tr>
<th>Problem area</th>
<th>Computer program</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forecasting and modeling</td>
<td>Model (equations)</td>
<td>Independent variables (data)</td>
<td>Forecast (dependent vars.)</td>
</tr>
<tr>
<td>Optimal control*</td>
<td>Control strategy</td>
<td>State variables</td>
<td>Control variables</td>
</tr>
<tr>
<td>Classification</td>
<td>Decision tree</td>
<td>Values of attributes</td>
<td>Class of object</td>
</tr>
<tr>
<td>Emergent behavior</td>
<td>Set of rules</td>
<td>Sensory input</td>
<td>Actions</td>
</tr>
<tr>
<td>Robotic planning</td>
<td>Plan</td>
<td>Sensor values</td>
<td>Actions</td>
</tr>
</tbody>
</table>

*Tasks such as balancing a broom, managing a process plant or keeping a pool at the proper temperature

These may seem somewhat esoteric, but they cover a wide range of useful problems (or problems with potentially useful solutions!). In fact, they cover problems generally not solved by computers... because it simply wasn’t possible until recently. Many complex problems of design and engineering are covered by control strategy (getting something to control something else to achieve desired results) or some form of regression for generating mathematical expressions or rules, so that independent values will generate the proper results.

Thus for each problem there’s a standard framework to use in setting up the problem and the specific GA and evaluation functions for its solution. The following items occur in each problem and vary in detail:

- objective, terminal set, function set, fitness cases, raw fitness measure or hits (number of fitness cases accomplished successfully), standardized fitness (to use in selection), forms of variation (mutation and crossover rules and rates), population size, number of generations, success predicate.

However, simple tables and lists -- or even the truism that the solutions to most problems may be viewed as computer programs -- don’t guarantee much in the way of a practical problem-solving approach unless there is a way to generate such a program for a particular problem. That’s the second part (and the majority) of the book. How do you represent all these problems in such a way that the generic genetic-programming approach can solve them?

**Answer components**

With the information above as a guide, you can start assembling some basic components that are probably part of the program you want. However, this is not a case of pre-rigging the solution. These components are not themselves full programs, typically, but functions and selected elements (terminals) appropriate to the problem domain. The functions are standard, low-level programming functions such as arithmetic operations, standard programming operations (including iteration and if-then), logical/Boolean functions, or possibly domain-specific functions such as sines or cosines.
The terminals are the program's interaction with the problem or the real world -- numbers, sensor values, true or false, actions to be taken by a machine, dollar values in a data series or forecast, commands for running a swimming pool heater, etc. (If you start with extraneous functions and terminals, they will probably eventually drop out of the population. On the other hand, some missing ones can be created, such as squaring a number from multiply operations.) The terminals are the leaves on the tree.

On the left: Two parent computer programs, each with a crossover point marked with a scissor.

On the right: After crossover, the offspring on the left is the well-known solution for one of the roots of the quadratic equation $ax^2 + bx + c = 0$.

**Generation 0**

Genetic programming starts with a random population of programs represented as trees and made up of random collections of elements; however, the varieties of elements are nonrandomly selected to be appropriate for the problem at hand. The programs are varied in size and structure as well as in constituent elements. These original programs are short, closer to program fragments, but large enough to do real work so that their performance ("fitness") can be measured and the better ones selected. The inputs, intermediate results, functions and outputs are expressed in terms appropriate to the particular domain; there's little need for pre- or post-processing. But there is certainly a need to hook the programs up to sensors, data sources, remote controls or other applications.

Fitness is measured by the evolving programs' ability to produce the desired results, whether that's in terms of minimizing error; minimizing the use of specified resources such as time, fuel or money to achieve a result such as hitting a sales goal or backing a truck up to a loading dock correctly; or maximizing some specified measure of fitness such as recognizing or classifying items correctly. "Fitness cases" are the specific problems used in the aggregate to evaluate the individual programs as they evolve.
Behind the scenes

The most fit individual from any generation (usually it's the last generation) is the answer. But let's explore how we get there. How many fitness cases do you need? How large an initial population? How many generations? How much crossover of program components to produce the next generation?

There are no clear answers to any of these questions, and they vary according to the problem type. Partly for simplicity, and partly so that it would not appear that he had tailored an approach to each problem type, Koza pretty much stuck to a standard (but probably not optimal) procedure, with populations of 500 and 51 generations (the initial random sample plus 50 succeeding generations). The proportion of crossover is 90 percent; that is, out of each new population of 500, there are 450 individuals generated from two parents through crossover (225 pairings of parents). Another 50 individuals have been reproduced intact.

Note of course that some of these individuals may be duplicates of especially fit individuals from the previous generation, and some parents may be represented in several pairings while other, unfit would-be parents are removed from the population and do not reproduce at all. The likelihood of any particular individual reproducing or participating in crossover is proportional to its fitness, as defined.

The crossover points are determined randomly, except that 90 percent occur internally, among the function points, rather than simply involving switching of terminals (or values), which is closer to mutation than actual crossover. (Mutations are not included, although they could easily be added.)

--------------
If a population hasn't come close to the goal after 50 generations, it probably won't get much better over the next 150. On the other hand, at least one of four runs is likely to come close to the goal after only 50 generations.
--------------

Parallel progress

Overall, the process is extremely parallelizable at a high level (not just in the sense that you could evaluate individuals concurrently). In practice, Koza has found it more productive to do several short runs rather than a single long run; for example, four runs of 50 generations each rather than a single run of 200 generations. Generally, the progress is fastest in the first 50 generations than thereafter (although the precise parameters vary by problem); short runs have a better payoff proportional to the resources allotted. Thus you can easily parallelize at the macro level simply by running four (or seven or ten) GAs simultaneously, and picking the best of the results they come up with.

Fitness of the fitness cases: There's the rub

The big computation burden in genetic programming is not the generation of new populations, which is fairly simple and straightforward, but measuring the fitness of the individuals in each generation. Usually, the programs are run against a variety of cases -- independent variables, problems, con-
straights or other representative situations. The fitness of an individual is the average performance of that individual in a variety of fitness cases.

This is where the "but of course it's not magic" caveats come to roost. The selection of fitness cases is like the astute selection of test cases for a debugger: If you don't cover your universe broadly enough, you're likely to get unpleasant surprises later. The idea is to cover a range of possibilities so that the genetically engineered program gets tested at solving the problem type, broadly defined, rather than just some specific problems that may be skewed in some way. Picking and expressing these cases well, of course, requires a good understanding of the problem domain and an ability to pick both representative cases and ones that properly exercise the resulting program's ability to handle likely or possible variations.

How Koza was generated

John Koza, now 48, has proven himself to be a natural genetic programmer through a variety of fitness cases. He got his PhD in 1972 under John Holland (father of genetic algorithms) at the University of Michigan. His topic seemed to have little to do with GAs; it was "Inducing a nontrivial and parsimonious grammar from a sample of sentences." However, it actually had a lot to do with his later work. Genetic programming, in a sense, is the ultimate technique in program induction, and grammar is a basic notion in the field. And a grammar, after all, is a specification of allowable sequences and combinations of elements.

After that experience, which involved decks of punched cards and was both exciting and tedious, Koza went on to found Scientific Games, an Atlanta company which developed lottery systems and was sold to Bally Manufacturing in 1982, when it had sales of $38 million. He stayed on for five years.

With some of the money from the sale he created the Third Millennium Venture Capital Fund, and started spending time at the Stanford computer science department, then run by Nils Nilsson. "I went to an orgy of conferences that year, including IJCAI in Italy and AAAI in Seattle," recalls Koza.

An idea gelled in August 1987, at the AAAI meeting (the one with the debate on "Should AI run for president?"). He ended up buying an Explorer LISP machine from TI that October after taking a vendor course in September, and got some initial results in the first few weeks. "It dawned on me that you could generalize it to more and more problems," he says. Nilsson encouraged him not to waste his time on a formal proof of the concept, but to go for a selection of benchmark problems in a variety of areas.

He produced his first paper in Detroit at IJCAI in 1989 -- "a restrained lunch buffet of five or six problems," he recalls. His book, by contrast, is a full-scale smorgasbord. As consulting associate professor, which means what, he teaches courses on GAs and artificial life in the computer science department at Stanford. He currently uses a four-processor LISP machine, TI's Explorer MP, one of only about a dozen such parallel LISP machines around. Ironically, genetic programming has appeared just as LISP machines seem to be going extinct -- or it may be the fitness case that finally proves their utility.
Although it's not polite to say so, it's inevitable that some evolving viruses will show up soon. (If we thought we were the only person to think about this, we'd hesitate to mention it, but that's hardly the case.) The proper method of counterattack is not to stop research in the hopes of stopping viruses, but rather to work harder on co-evolving immune systems to foil them.

Sensible, unexcitable folks point out correctly that evolving software viruses will find it hard to take hold, since actual machine code is too brittle to mutate effectively. Any errors will generally kill the virus rather than strengthen it. (This was the reasoning behind Tom Ray's Tierra, covered in our April issue, which created a special limited-set language so that mutations and flaws would still generate something meaningful.) Thus a virus would probably have to carry around either its own language and interpreter/compiler, or a mutation module that would be easy to detect. (Most viruses are detected by scanners that recognize distinctive pieces of code.) Unlike physical viruses, computer viruses can usually be destroyed once they are detected. And anything as bulky and specific as the mutation module could not wreak much havoc before being detected -- we think!

The United States spends about 15 percent of its GNP on health care, while we spend only about 5 percent of our packaged software budget on utilities (virus countermeasures, backup tools, etc.). Are we spending too much on health or too little on software hygiene -- or is this a spurious parallel?

However, environments have a habit of changing. In our current world of discrete pcs and massive centrally controlled mainframes, the above is true. In the future world of networks, distributed data and roaming black-box objects (courtesy of interoperability, object request brokers and component software), there will be much more foreign matter floating around the global network, and it will be harder to track. The global network is not a discrete, tidy system, but more like the global road network, with byways and highways, dead ends and washed-out bridges. Things can appear out of nowhere, and vanish as easily. Objects hide themselves and what lurks inside them. Each system on the network can have its own moats and filters, but they cut off the outside world.

Rather than a romantic view, this is an appropriate one. From thinking of computers and electronic systems as precise, digital, deterministic systems, we will come to think of them as alive, biological, unpredictable. Just as we can't eradicate disease, we won't be able to eradicate viruses. But if we understand these systems, we might be able to do better at fighting both viruses and poverty.

So how do we protect against viruses in the digital/biological world of the future? A number of thoughts come to mind.

First of all, there's the value of diversity. We could legislate a few standards and then outlaw anything that didn't conform, thus eliminating all viruses. In theory...
But aside from being impossible anyway, that would forestall progress, creativity and other good things. Secondly, it would render us extremely vulnerable to any virus that did manage to overcome the defenses, since there would be a vulnerable, homogeneous population of systems to attack.

We're better off with a variety of systems. A virus might kill off one or two variants, but that would be a small proportion of the population at large. Moreover, viruses don't spread through ether; they spread from system to system. Thus a population of 100 vulnerable Xes among 1000 systems of all kinds is not one-tenth as vulnerable as a population of 1000 Xes, but far less so, since the interspersed other systems stem the spread of an X-specific virus. (This argues for keeping the Xes and Ys talking to each other, rather than segregating each kind in its own group.)

**Immunity isn't complete, but it's adaptable**

Diversity makes it harder for the virus to gain a hold and easier to find resistant systems. However, while diversity may keep a population from getting wiped out, it's not a sufficient defense against viruses for individual systems.

Ultimately, we're going to have to fight back with immune systems -- adaptive software trained to recognize foreign objects. Viruses, of course, will be disguising themselves as familiar, encapsulated objects, but at some point in some way they must behave differently, or else they could cause no harm. We're not sure how to design tools to catch this errant behavior, but it's the kind of challenge GAs should be good at. It takes one to catch one.

**RELEASE 0.5 -- LIVING SOFTWARE: COMMERCIAL EVOLUTION**

Viruses are only one example of life in the primordial soup of electronic networks. They compete for resources against the not-quite-living useful programs also built by humans. It seems inevitable that we'll soon have living things on the Internet.

Here's a thought experiment, not to be taken too seriously: Let's posit that software packages are alive, and are influencing their environment to replicate and perpetuate themselves. This living software bends humans to its will by being "useful." VisiCalc, for example, managed to pass on its (altered) genes to 1-2-3, while other mutations, such as Context MBA, went extinct. Those genes (and the phenotype) also were combined into Excel and Quattro.

Software packages are like the first few molecules in the primordial soup; not quite alive, perhaps, but somehow self-replicating. Commercial software simply uses different methods.

But now it may become more alive. Rather than duplicate itself in separate disk-duplication machines run by obedient humans, code now starts to live on networks, and can replicate itself in "network" versions designed to download copies onto client machines.
RESOURCES & PHONE NUMBERS

Matt Jensen, Axcelis, Inc., (206) 632-0885; fax, (206) 632-3681
Gilbert Syswerda, Bolt Beranek and Newman Inc., (617) 873-8234; fax, (617) 873-3776
David Barrow, KIQ Ltd., 44 (71) 233-7173
Susan Wider, Santa Fe Institute, (505) 894-8800; fax, (505) 873-3776
Dean Cerys, TSP (The Software Partnership), no phone on purpose; write to Box 991, Melrose MA 02176, USA
John Koza, Third Millennium Fund, (415) 941-9137; fax, (415) 941-9430
Lawrence Davis, Tica Associates, (617) 864-2393; fax, (617) 494-4050
Tom Schwartz, Tom Schwartz Associates, (415) 965-4561; fax (415) 968 0834
Darrell Whitley, University of Colorado and COGAN, (303) 491-5373
Chuck Karr, US Bureau of Mines, Tuscaloosa Research Center, (205) 759-9478

For further reading or hacking:

Davis, Lawrence, editor. Handbook of genetic algorithms, Van Nostrand Reinhold, 1991. A tutorial and case studies, in much more detail (386 pages!) than presented in this newsletter. John Grefenstette and Lawrence Davis have produced a 16-hour video lecture series on genetic algorithms that comes in a package with this book and Goldberg's, below, along with two formerly shareware GA software tools called OOGA and GENESIS. The full package is available for $975 from Tom Schwartz Associates; The Software Partnership sells the software only for $50.

Goldberg, David E. Genetic algorithms in search, optimization and machine learning, Addison-Wesley, 1989.


Koza, John. Genetic Programming: On the programming of computers by means of natural selection and genetics, due in August from MIT Press. The book includes software (which needs a LISP compiler) for the user to experiment with, and a videotape illustrating actual computer runs.

The International Society for Genetic Algorithms has been formed to support the GA field's major conference, held in North America in the summer of odd-numbered years. Currently just a legal entity, it will soon become a genuine society and start accepting members. Information on these matters is available through GA-LIST-REQUEST@AIC.NRL.NAVY.MIL, an electronic mailing list accessible by sending it a message with your return electronic address.

COMING SOON

• Newton.
• Customer resource planning.
• Infrastructure for mail and groupware.
• Commercialization of the internet.
• Pen stuff.
• Constraint-based reasoning.
• And much more... (If you know of any good examples of the categories listed above, please let us know.)
**Release 1.0 Calendar**

### June

**June 3-5**

**June 5**
**Symposium and open house of the Human-Computer Interaction Laboratory - College Park, MD.** Sponsored by the University of Maryland Center for Automation Research. With Professor Ben Shneiderman. Call Gary Marchionini, (301) 405-2053.

**June 6**
**COGAN - Baltimore, MD.** Combinations of Genetic Algorithms and Neural Networks. Call Darrell Whitley, (303) 491-5373.

**June 7-11**
**International joint conference on neural networks '92 - Baltimore.** The big one. Sponsored by the International Neural Network Society and IEEE. Call Gail Reed, (619) 453-6222.

**June 15-19**
**Artificial Life III - Santa Fe.** Sponsored by the Santa Fe Institute. Lots of life, real and artificial. How to grow your own. See our April issue. Call Christopher Langton, (505) 984-8800.

**June 23-25**
**PC EXPO - New York City.** Sponsored by Bruno Blenheim. Call Annie Scully, (201) 346-1400 or (800) 829-3976.

**June 23-25**
**Digital World '92 - Beverly Hills.** Sponsored by Seybold Seminars. Call Beth Sadler or Kevin Howard, (310) 457-5850.

**June 29-July 3**
**ECOOP '92 - Utrecht, Netherlands.** Sponsored by Software Engineering Research Center. Contact: Gert Florijn, 31 (30) 322640; fax, 31 (30) 341249; e-mail, ecoop92@serc.nl.

**June 30-July 1**
**First international conference & exhibition on advanced service and HelpDesk automation - Strasbourg, France.** Sponsor: Applied Workstations and ServiceWare. Contact: Jeff Pepper, (412) 826-1158; Tim Lewis, 44 (306) 77331; fax, (306) 77696.

### July

**July 6-10**
**CASE '92 - Montreal.** Sponsored by International Workshop on CASE and IEEE. Contact: Francois Coallier, (514) 468-5523; fax, (514) 647-3163.

**July 14-16**
**AAAT/IAAT '92 - San Jose.** AI's AE (annual event). Sponsor: American Association for Artificial Intelligence. Call Mary Livingston, (415) 328-3123.

**July 14-17**

**July 20-21**

**July 20-23**
**Object World - San Francisco.** Co-sponsored by The Object Management Group and World Expo Corp. Businesspeople's answer to OOPSLA. Call Bill Hoffman, (508) 820-4300.

**August 10-14**
**LUV-92, LISP users and vendors conference - San Diego.** Sponsored by the Association of LISP Users. Call Laura Lotz, (215) 651-2990.

*Please let us know about any other events we should include.* — Denise DuBois

---

**Release 1.0**

28 May 1992
SUBSCRIPTION FORM

Please enter my subscription to Release 1.0 at the rate of $495 per year in the U.S. and Canada. Overseas subscriptions are $575, airmail postage included. Payment must be enclosed. Multiple-copy rates available on request. Satisfaction guaranteed or your money back.

Name
Title
Company
Address
City State Zip
Telephone

☐ Check enclosed.
☐ Charge my
   ☐ American Express ☐ MasterCard ☐ Visa
Card Number Expired Date
Name on Card Signature

☐ Please send me information on your multiple-copy rate.

Please fill in the information above and send to:

EDVENTURE HOLDINGS INC.
375 PARK AVENUE, SUITE 2503
NEW YORK, NEW YORK 10152

If you have any questions, please call us at (212) 758-3434.

Daphne Kis
Publisher