Genetic programming

Haym Hirsh
Rutgers University
hirsh@cs.rutgers.edu

A wide range of core concepts in AI—and in computer science in general—can be traced to computational metaphors inspired by a rich variety of phenomena in the natural world. One of the best examples is neural networks, whose core ideas are based on the functioning of systems of neurons in the brain. This issue's Trends and Controversies concerns genetic programming (GP), whose inspiration comes from computational analogs to Darwinian evolution.

GP is part of a more general field known as evolutionary computation. Evolutionary computation is based on the idea that basic concepts of biological reproduction and evolution can serve as a metaphor on which computer-based, goal-directed problem solving can be based. The general idea is that a computer program can maintain a population of artifacts represented using some suitable computer-based data structures. Elements of that population can then mate, mutate, or otherwise reproduce and evolve, directed by a fitness measure that assesses the quality of the population with respect to the goal of the task at hand.

For example, an element of a population might correspond to an arbitrary placement of eight queens on a chessboard, and the fitness function might count the number of queens that are not attacked by any other queens. Given an appropriate set of genetic operators by which an initial population of queen placements can spawn new collections of queen placements, a suitably designed system could solve the classic eight-queens problem. GP's uniqueness comes from the fact that it manipulates populations of structured programs—in contrast to much of the work in evolutionary computation in which population elements are represented using flat strings over some alphabet.

The concepts of Darwinian evolution explain the vast diversity of biological organisms in terms of reproductive and evolutionary processes. This issue's essays show that when we map these concepts onto computers, we get analogously complex and diverse computational artifacts. Wolfgang Banzhaf's lead-off essay surveys the area, both from a historical perspective and a practical standpoint, advancing the view of GP as a form of automatic programming. John Koza, the progenitor of modern GP, develops this theme in more detail, elaborating on the automatic-programming perspective of GP and cataloging some of the notable successes this area has already achieved. The essays by Conor Ryan and Lee Spector then discuss two very different areas—software reengineering and quantum computing—where these successes are now occurring. Finally, Christian Jacob concludes with an essay that ties GP back to its origins, asking whether it can in turn provide insights for developmental biology.

The essays show that underlying the Darwinian metaphors of evolutionary computation is a powerful technology that, when given sufficient computational resources, can yield results that compete with those of the best domain experts. To me, the key insight in evolutionary computation is that computer technology makes it possible to simulate the progress of time—the movement through consecutive generations of populations—on a time scale very different than that found in biology. Rather than waiting years for the effects of reproduction and evolution to materialize, we can create thousands of generations in a matter of hours on a computer. This insight is all the more notable given Moore's Law and the continuing geometric increases in computer speed, leaving me all the more intrigued about what this technology might yet produce. These essays do a nice job of surveying where we are and where we might find ourselves in the not-too-distant future.

— Haym Hirsh

The artificial evolution of computer code

Wolfgang Banzhaf, University of Dortmund, Germany

Over the past decade, the artificial evolution of computer code has become a rapidly spreading technology with many ramifications. Originally conceived as a means to enforce computer intelligence, it has now spread to all areas of machine learning and is starting to conquer pattern-recognition applications such as data mining and the human–computer interface.

In the long run, genetic programming[1,2] and its kin will revolutionize program development. Present methods are not mature enough for deployment as automatic programming systems. Nevertheless, GP has already made inroads into automatic programming and will continue to do so in the foreseeable future. Likewise, the application of evolution in machine-learning problems is one of the potentials we'll exploit over the coming decade.

In the beginning

Alan Turing was perhaps the first to express this vision in his seminal essay "Computing Machinery and Intelligence."[3] Besides proposing the Turing test, he ventured into unknown territory by proposing computing machines with constituting random elements. He then suggested that a computing machine could manipulate itself just as well as any other data. Self-modifying code—something generations of computer scientists were later taught to avoid—thus became an interesting possibility. By observing the results of behavior—of executing programs—the code could improve itself, perhaps when working on external data.

Turing hit on the learning problem when he noticed the combinatorial multitude of environments that require a machine to acquire approximately correct behavior rather than starting out with such behavior. Naturally, he concluded that we should enact a procession of learning stages during which a machine could "grow" in knowledge. He termed this a *child machine*, which would learn more or less quickly (depending on its construction) by following its own developmental process. Turing anticipated that a systematic process would produce the child machine, but he did not rule out a random element in its construction and indeed pointed to the natural evolutionary process

as a kind of search process that could (theoretically, if only nonefficiently) guide the search for a faster learning machine.

Later, but still early on in the history of computing, R.M. Friedberg invented a system, called Herman, designed to learn through random variations.[4,5] He defined a virtual assembler language space within which these variations took place. He tested the variations against a given task, and the system preferred instructions that, in a given location in the program, showed better performance over less performing instructions. Herman and its variants showed limited success in adapting to the prescribed task. Besides certain deficits in the variation-selection loop, the most limiting factor was computer time. Evaluating programs one after the other to measure their performance was very slow.

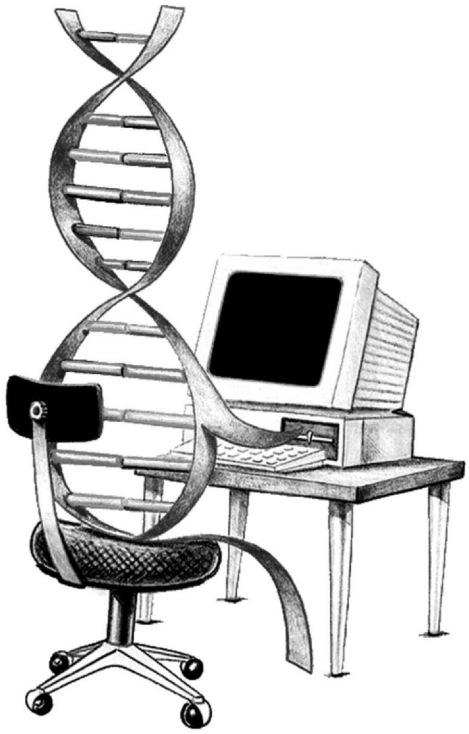
Moving into artificial evolution

Today, we are in a much better position because many evolutionary algorithms for optimization have emerged. Mostly driven by applications in problems that were difficult to approach using traditional means, artificial evolution is now routinely applied to combinatorial optimization problems of all varieties. The key insight was to center the algorithm around the variation-selection loop (see Figure 1). Sticking to this loop, we can contradict all probability arguments, such as, "There is only one in 10^{100} solutions that is of acceptable quality." By continuously varying the programs, we can set into motion a stream of improvements that will hopefully converge on a solution of satisfying quality. We now know that the so-called Bremermann limit, which states that there is a limit to the amount of information processing at approximately 2×10^{47} bits per second per gram that a machine can use to solve a problem,[6] is not relevant if we enact the variation selection loop.

The difference between combinatorial optimization problems and programming is not so much the brittleness of programs as the mere size of the search space. Brittleness or feasibility constraints are also a major problem in constraint-optimization problems, where evolutionary algorithms first learn to handle them. The techniques have been subsequently transferred to the realm of programming, where they can similarly be employed. However, having to search in a space that has routinely $10^{100,000}$ rather than 10^{100} search points is rather demanding. So, for a long time, researchers believed that generating computer code through artificial evolution would not work because of the search space's size. The only good news that comes in the realm of programming is that there are also many more satisfying solutions.

As every programmer knows, there are a large variety of programs that fulfill a given task—a few simple ones but many more complicated ones. This is a key insight that helps explain why artificial evolution of computer code works. Above the complexity threshold, where a program starts to work, the ratio of acceptable to nonacceptable solutions doesn't change very much, but to reach this threshold, the evolutionary search process must be able to vary the complexity of programs. This is the case in all serious attempts to evolve programs, and it is one of the hallmarks of the approach.

Perhaps the most radical approach to the artificial evolution of computer code is to make binary machine code the object of evolution.[7] The evolutionary algorithm thus treats the binary numbers representing a program's machine code instructions as data, which the evolutionary operators of mutation and crossover then manipulate. Figure 2 shows an example of programs and their manipulation. Mutation randomly varies a program's instruction by flipping bits, and crossover generates variants by mixing two programs—for example, by taking the first half of program 2 and appending the second half of program 1. To avoid hitting unfeasible programs, the system runs a sanity check that filters all op-codes and register–memory addresses or constant values to avoid a system crash. If the system also performs this sanity check with the initial population of programs, it can guarantee it won't crash the computer on which it runs. The main advantage of a machine code GP system is its speed and memory efficiency. In a way, such a system speaks the computer's native language. As such, it guarantees the fastest realization possible.



Artwork courtesy of Genetic Programming, Inc.

Let's consider a few numbers: A present-day CPU can execute approximately 10^9 instructions per second. Without any structural precautions, GP might generate programs of a length between 10 and 200 instructions. A good maximum size for evolving programs is therefore 500 instructions. With this size, the system can evaluate 2×10^6 programs per second. This would make a population of about 1,000 programs for 2,000 generations or a population of 20,000 programs for 100 generations. In reality, the numbers are not this large due to the population's administration overhead, plus some display and supervision functions. Still, the amount of computational power is stunning.

Applications of automatic induction of machine code with GP range from generating small subroutines (to be decompiled into assembler or C code for export into other program environments), to real-time problems (for example, robot control or sound discrimination), to offline applications such

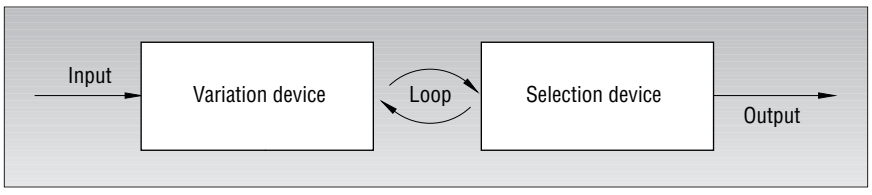


Figure 1. The variation-selection loop of a typical evolutionary algorithm.

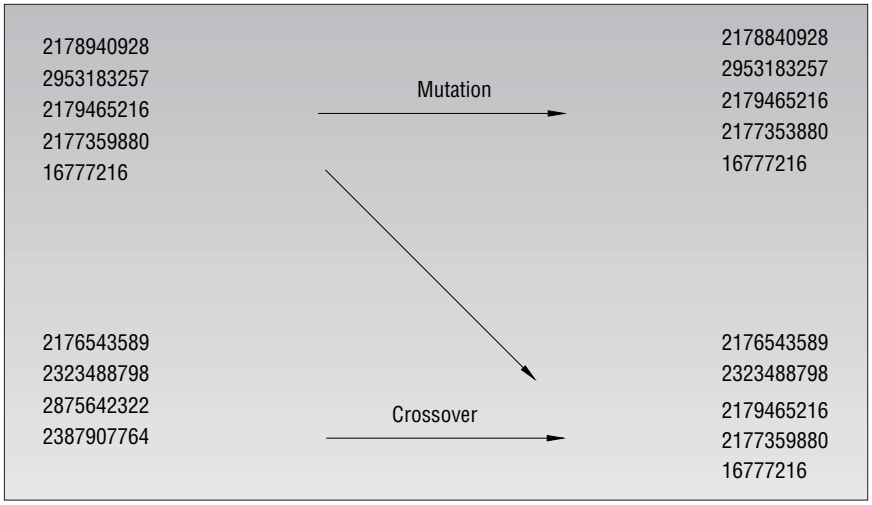


Figure 2. The operations of mutation and crossover applied to two machine code programs shown as integer numbers. Each number represents a single instruction. A frame of header and footer instructions (not shown here) is added to allow program execution.

as time-series analysis or data-mining tasks. All applications approached by neural networks are obvious candidates for a GP attack, as are more symbolic problems, such as logic circuits and the like. Even rule-based systems, such as automatic theorem proofing, are candidate targets for GP techniques.

Besides the raw speed of certain GP systems, researchers are actively exploring ways to modularize programs to gain nonlinear speedups resulting from reusing certain parts of the programs and improving the evolution of structures. Automatically defined functions, introduced by John Koza in the mid 1990s, are the most widespread technique in modularization. Modularization, however, will continue to be a focus of research in the coming years, and researchers expect new methods will turn up better structures to be subjected to evolution.

The 21st century—leaving medieval times?

We always seem to simultaneously create inventions at various places. Different researchers in the 1980s mulled over ideas for the evolution of computer code, notably S.F. Smith in 1980 and N. Cramer in 1985. With Koza's contributions, however, the field really took off in the 1990s.

Why do we believe GP will revolutionize computing? First, the stage is set: numerous computers are idle and ready to take up productive computation. We estimate the amount of computer power available to be in the petaflop range. The situation is similar to the middle ages, when Europe was flooded with paper.[8] In the 14th century, paper mills started to turn out large volumes of paper and prices continued to fall; the shortage was suddenly not the material for writing (as it had been for centuries past), but the number of scribes knowledgeable enough to make use of the resource both for copying and producing new texts. As pressure grew to invent a faster method for copying texts, the invention of the printing press by Johannes Gutenberg was bound to happen. The new technology subsequently spread through Europe like a bushfire, quickly putting the paper resource under pressure again.

Today, we seem to be in the medieval days of handcrafted coding of computers. A programmer must specify, write, and test every program. The flood of CPU cycles and tasks to be moved onto computers is so large that education cannot keep up with the demand.

However, as I've argued, the means are there through automatic program writing. Evolving and testing multiple variants of programs will make both an automatic and fault-tolerant production of programs possible. This is the second precondition for a successful revolution.

As the third precondition—the commercial drive for automating programming—is visibly present, artificial evolution of computer code seems also commercially viable. Certainly, GP will not remain the only way to generate programs. We're already applying simulated annealing and other heuristic search methods to problems in learning computer programs. But with its thirst for CPU power, GP will continue to thrive as more and more computers become available in the coming years. Perhaps we stand witness to one of the major culture revolutions, rivaling the invention of the printing press 400 years ago.

References

1. J. Koza, *Genetic Programming*, MIT Press, Cambridge, Mass., 1992
2. W. Banzhaf et al., *Genetic Programming—An Introduction*, Morgan Kaufmann, San Francisco, 1998.
3. A. Turing, "Computing Machinery and Intelligence," *Mind 59*, Vol. 59, 1950, pp. 433–460.
4. R.M. Friedberg, "A Learning Machine: Part I," *IBM J. Research and Development*, Vol. 2, 1958, pp. 2–13.
5. R.M. Friedberg, B. Dunham, and J.H. North, "A Learning Machine: Part II," *IBM J. Research and Development*, Vol. 3, 1959, pp. 183–191.
6. H.J. Bremermann, "Optimization through Evolution and Recombination," *Self-Organizing Systems*, M.C. Yovits, G.T. Jacobi, and G.D. Goldstine, eds., Spartan Books, Washington, D.C., 1962, pp. 93–106.
7. P. Nordin, *Evolutionary Program Induction of Binary Machine Code and its Application*, PhD thesis, Univ. of Dortmund, Germany, 1997.
8. J. Burke, *Connections*, Little, Brown and Co., Boston, 1995.

Human-competitive machine intelligence by means of genetic programming

John R. Koza, Stanford University

The central challenge and common goal of AI and machine learning is to get a computer to solve a problem without explicitly programming it. This challenge envisions an automatic system whose input is a high-level statement of a problem's requirements and whose output is a satisfactory solution to the given problem. To be useful, the system must routinely achieve this goal at levels that equal or exceed the human level of performance.

Genetic programming has achieved this goal. It starts with a high-level statement of a problem's requirements and produces a computer program that solves the problem. It has produced results that are competitive with human-produced results (25 currently known) in areas such as control, design, classification, pattern recognition, game playing, and algorithm design. Six of these automatically created results are improvements to previously published, human-created scien-

Related work

The 1992 book *Genetic Programming: On the Programming of Computers by Means of Natural Selection*[1] and accompanying videotape[2] demonstrate the breadth of genetic programming by successfully applying it, in a uniform and consistent manner, to a wide variety of problems taken from AI and machine-learning literature of the late 1980s and early 1990s. The 1994 book *Genetic Programming II: Automatic Discovery of Reusable Programs* focuses on the automatic discovery and creation of reusable and parameterized subroutines and argues that code reuse is essential for solving nontrivial problems in a scalable fashion.[3,4] *Genetic Programming III: Darwinian Invention and Problem Solving* contains numerous examples of human-competitive results and describes the Genetic Programming Problem Solver.[5,6]

Over 1,500 papers have been published on GP. You can find additional information in edited collections of papers such as the *Advances in Genetic Programming* series of books from the MIT Press,[7] in the proceedings of the annual Genetic Programming Conference (currently combined into the Genetic and Evolutionary Computation Conference) and the annual Euro-GP conference, in the *Genetic Programming and Evolvable Machines* journal, and at Web sites such as www.genetic-programming.org.

References

1. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, Mass., 1992.
2. J.R. Koza and J.P. Rice, *Genetic Programming: The Movie*, MIT Press, Cambridge, Mass., 1992.
3. J.R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge, Mass., 1994.
4. J.R. Koza, *Genetic Programming II Videotape: The Next Generation*, MIT Press, Cambridge, Mass., 1994.
5. J.R. Koza et al., *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufmann, San Francisco, 1999.
6. J.R. Koza et al., *Genetic Programming III Videotape: Human-Competitive Machine Intelligence*, Morgan Kaufmann, San Francisco, 1999.
7. L. Spector, H. Barnum, and H.J. Bernstein, "Quantum Computing Applications of Genetic Programming," *Advances in Genetic Programming 3*, L. Spector et al., eds., MIT Press, Cambridge, Mass., 1999, pp. 135–160.

tific results. Seven of them infringe on previously issued patents, one improves on a previously issued patent, and nine duplicate the functionality of previously patented inventions in a novel way (for more information on GP, see the "Related work" sidebar).

How GP works

Recognizing that the solution to a wide variety of problems can easily be recast as a search for a computer program, GP conducts its search in the space of computer programs. It operates by progressively breeding a population of computer programs over a series of generations using the Darwinian principles of evolution and natural selection. It extends the genetic algorithm[1] to the arena of computer programs.[2]

A run of GP starts with a primordial ooze of thousands of randomly created computer programs. It evaluates pairs of programs from the population to determine which one is better for solving the problem at hand. It then probabilistically selects programs from the population based on this partial order and modifies the programs using crossover (sexual recombination), mutation, and architecture-altering operations. The architecture-altering operations automatically add and delete subroutines, subroutine parameters, iterations, loops, recursions, and memory in a manner patterned after gene duplication and gene deletion in nature. These operations automatically arrange the program's elements into a hierarchy.

GP as automatic programming

GP has all the following attributes of what might be called automatic programming (or automatic program induction or automatic program synthesis). It

- starts with "what needs to be done";
- tells us "how to do it";
- produces a computer program;
- automatically determines the number of steps in the program;
- supports code reuse (subroutines);
- supports parameterized code reuse;
- supports code reuse in the form of iterations, loops, and recursions;
- supports reuse of the results of executing code in the form of memory and internal storage;
- automatically determines the use of subroutines, iterations, loops, recursions, and memory;
- automatically determines the hierarchical arrangement of subroutines, iterations, loops, recursions, and memory;
- supports a wide range of useful programming constructs;
- is well-defined;
- is problem-independent;
- applies to a wide variety of problems;
- is scalable; and
- produces human-competitive results.

The last of these attributes is especially important because it reminds us that the ultimate goal of AI and machine learning is to produce useful results—not to make steady progress toward solving toy problems. The human-competitive results that GP produces include the automatic synthesis of the PID controller topology Albert Callender and Allan Stevenson patented in 1939 and the PID-D2 topology controller Harry Jones patented in 1942. The results also include the automatic creation of

- filter circuits that infringe on patents issued to George Campbell, Otto Zobel, and Wilhelm Cauer between 1917 and 1935;
- amplifiers and 12 other circuits infringing on Sidney Darlington's 1953 patent for emitter-follower sections;
- a sorting network that is better than one in the 1962 O'Connor and Nelson patent on sorting networks;
- a crossover filter circuit infringing on Otto Zobel's 1925 patent;
- a quantum computing algorithm for the Deutsch–Jozsa "early promise" problem, the Grover's database search problem, and the depth-2 or query problem that are better than previously published algorithms;[3]
- a cellular-automata rule for the majority classification problem that is better than the Gacs–Kurdyumov–Levin rule and all other known human-written rules;[4]
- a soccer-playing program that ranked in the middle of the field of 34 human-written programs in the Robo Cup 1998 competition;[5]
- an algorithm for the transmembrane

segment identification problem for proteins that is better than previously published algorithms;
- motifs that detect protein families that are equal to or better than previously known human-discovered motifs; and
- amplifier, computational, voltage reference, Nand, digital-to-analog, and analog-to-digital circuits that duplicate the functionality of previously patented circuits in novel ways.

The rapidly decreasing cost of computing power and the amenability of GP to nearly 100%-efficient parallelization should let GP continue to deliver human-competitive results on increasingly difficult problems. Because evolutionary search is not channeled down preordained paths, I anticipate that researchers will routinely use GP to produce new and useful inventions.

Performance evaluation

GP delivers its results with a high A/I ratio—that is, there is a great deal of "artificial" compared to the amount of "intelligence" provided by the human user. Specifically, a GP run is launched after the human user performs several preparatory steps. These clearly defined preparatory steps provide a bright line that distinguishes between what the human user provides and what GP delivers. These steps involve specifying the ingredients (functions and terminals) of the to-be-created programs, a fitness measure that decides if one program is better than another in satisfying the problem's high-level objectives, administrative parameters, and termination criteria. A crisp evaluation of GP's performance is possible because the GP algorithm is problem-independent, has no hidden steps, and has no backdoors for embedding (consciously or unconsciously) additional problem-specific knowledge.

Comparison to other approaches

GP differs from all other approaches to AI and machine learning in many important ways. First, it differs in its representation. GP overtly conducts its search for a solution to the given problem in program space. Second, it doesn't require an explicit knowledge base, and third, it doesn't use the inference methods of formal logic. Fourth, it does not conduct its search by transforming a single point in the search space into another single point but, instead, transforms a set (population) of points into another set of points. Fifth, GP does not rely exclusively on greedy hill climbing to conduct its search; instead, it allocates a certain number of trials, in a principled way, to choices that are known to be inferior. The population-based search and avoidance of hill climbing lets innovative exploration occur, while simultaneously exploiting immediately gratifying paths. Lastly, GP conducts its search probabilistically. Probability permits the search to avoid becoming trapped on points that are locally, but not globally, optimal and to make innovative leaps.

As John Holland said in l997:

> Genetic programming *is* automatic programming. For the first time since the idea of automatic programming was first discussed in the late '40s and early '50s, we have a set of nontrivial, non-tailored, computer-generated programs that satisfy Samuel's exhortation: 'Tell the computer what to do, not how to do it.'

References

1. J.H. Holland, *Adaptation in Natural and Artificial Systems*, Univ. of Michigan Press, Ann Arbor, Mich., 1975 (Second edition, MIT Press, Cambridge, Mass., 1992).
2. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, Mass., 1992.
3. L. Spector, H. Barnum, and H.J. Bernstein, "Quantum Computing Applications of Genetic Programming," *Advances in Genetic Programming 3*, L. Spector et al., eds., MIT Press, Cambridge, Mass., 1999, pp. 135–160.
4. D. Andre, F.H. Bennett III, and J.R. Koza, "Discovery by Genetic Programming of a Cellular Automata Rule that is Better than Any Known Rule for the Majority Classification Problem," *Proc. First Ann. Conf.*, J.R. Koza et al., eds., MIT Press, Cambridge, Mass., 1996, pp. 3–11.
5. D. Andre and A. Teller, "Evolving Team Darwin United," *RoboCup-98: Robot Soccer World Cup II. Lecture Notes in Computer Science*, M. Asada and H. Kitano, eds., Vol. 1604, Springer-Verlag, Berlin, 1999, pp. 346–352.

Genetic programming tools have the answers

Conor Ryan, University of Limerick, Ireland

There has long been some confusion over what kinds of tasks genetic programming can handle. Those unsure, or even suspicious of GP capabilities have often been known to ask, "Can GP evolve a word processor?" The answer is, of course, no. Another answer, however, could be, "No, but why would anyone want to do such a thing?" A better question might be whether GP could help design a word processor. The answer is yes.

GP as a tool, not a replacement

GP is not intended to be a fully automatic programming system that can generate elaborate code for all manner of exotic applications, with merely a gentle prod or vague description from a human. GP is an excellent problem solver, a superb function approximater, and an effective tool for writing functions to solve specific tasks. However, despite all these areas in which it excels, it still doesn't replace programmers; rather, it helps them. A human still must specify the fitness function and identify the problem to which GP should be applied.

This essay is concerned with using GP as an automatic software-reengineering tool—in particular, the problem of transforming serial code into functionally equivalent parallel code. If you asked, "Can GP convert a program into parallel?" I would probably say it can't. However, if your question were more GP-friendly—"How can GP help convert a program into parallel form?"—then my answer would be for you to read on.

Software maintenance

Parallel computing is becoming an increasingly important paradigm as hardware —such as Beowulf systems—have made powerful computational resources available to those with even the most humble budgets. However, to effectively use parallel hardware, the quality of your software is of utmost importance. Poorly designed, communication-intensive programs can even run more slowly on parallel machines than their serial counterparts. Furthermore, there are currently many institutions that run intensive applications on serial machines and, despite being the kind of user who stands to benefit most from parallel architectures, these insti-

tutions face the expensive task of rewriting the code from scratch. This essay is based on experience I gained while working on SCARE, a GP-based automatic parallelization system.

Software maintenance consumes a significant portion of the time, effort, and expense of many data-processing departments. Up to 70% of total expenditure is directed toward this activity. These tasks vary from meeting users' changing needs and desires to improving a program's control structure (for example, removing GOTOs to make subsequent modifications easier). Other reengineering tasks include the ubiquitous Y2K problem and the Euro-conversion, which involves adapting software to display monetary values in both local currency and the new European currency.

Those involved in software maintenance first face the task of understanding the code. This task is difficult because those involved in the initial writing of that code are usually long gone, and sparse or nonexistent documentation further complicates the task. Given the scarcity of available information, it is not surprising that a major part of the effort in reengineering—up to 50% of the project costs—is spent trying to understand the code.

The difficulty and level of concern—which often bordered on panic with the recent portents of doom accompanying the new millennium—have caused many to look for a third party to reengineer their code. The scale of their problems is evidenced by the existence of reengineering companies whose sole service was to provide Y2K solutions. The most successful reengineering companies have developed tools that automate, or at least semiautomated, the reengineering process. In general, the greater the level of automation, the greater the success rate and, thus, the less testing required.

It is reasonable to believe that automated software-reengineering tools will be increasingly important to the software industry as a whole. As the shortage of trained computer personnel increases, fewer programmers will be concerned with code comprehension. GP is potentially useful in this task because it requires no understanding of the code and can fully automate both reengineering and testing.

Applying GP

The traditional way we would expect programmers to apply GP to the task would be to identify the part of the program that needs modification or rewriting. They then would use it to evolve a program function to replace its original function. It is possible to use the original code as a fitness function for the evolved code, because we always need to preserve the functionality of any reengineered code. However, many mainstream programmers are somewhat skeptical or even suspicious of code produced by an apparently random method such as GP. Even vast amounts of regression testing are sometimes not enough to convince customers that there is no lurking pathological problem. Another problem is that GP usually insists on executing the code many times to evaluate its fitness. If the reengineered code is particularly time consuming, this might not be possible.

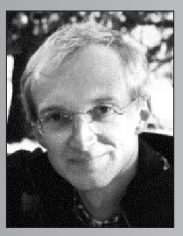
Programmers can execute certain reengineering tasks by carrying out standard transformations on the original code—for example, transforming two-digit dates to four digits, removing sections of code that are clones, and so forth. These tasks can only be automated if there are rules that govern which transformations can legally be applied and, if order-dependent, the order in which they should occur. If there are no clear rules to help order the application of the transfor-

Wolfgang Banzhaf is an associate professor of applied computer science at the University of Dortmund, Germany. His research interests lie in the areas of computational intelligence, molecular computing, artificial life, and self-organization. Before going to Dortmund, he conducted research on computers and synergetics at the University of Stuttgart. He holds a PhD in physics and is the editor in chief of the newly founded journal *Genetic Programming and Evolvable Machines*, published by Kluwer. Contact him at the Dept. of Computer Science, Univ. of Dortmund, 44221 Dortmund, Germany; banzhaf@cs.uni-dortmund.de; ls11-www.informatik.unidortmund.de/people/banzhaf.

John R. Koza is a consulting professor in the Section of Medical Informatics in the School of Medicine and in the Department of Electrical Engineering in the School of Engineering at Stanford University. At Stanford, he has taught a course on genetic algorithms and genetic programming since 1988 and he coteaches a course on computational molecular biology. He received his PhD in computer science from the University of Michigan. He is member of the IEEE, ACM, and International Society for Genetic and Evolutionary Computation. Contact him at Standford Medical Informatics, Dept. of Medicine, Stanford Univ., Stanford, CA 94305; koza@stanford.edu; www.smi.stanford.edu/people/koza.

Conor Ryan is a lecturer at the University of Limerick, in Ireland, where he is also the director of the Soft Computing and Reengineering Group. His research interests include genetic programming, evolutionary algorithms, neural networks, simulated annealing, parallel programming, scheduling, and software reengineering. He is particularly interested in promoting the use of AI techniques in modern industry. He received his BA in computer science and economics and his PhD in computer science, both from the University College Cork, Ireland. Contact him at conor.ryan@ul.ie.

Lee Spector is an associate professor of computer science in the School of Cognitive Science at Hampshire College. His research interests include genetic programming, quantum computation, and interdisciplinary connections among artificial intelligence, philosophy, psychology, linguistics, neuroscience, and the arts. He received his BA in philosophy from Oberlin College and his PhD in computer science from the University of Maryland, College Park. He is a member of the AAAI and the Int'l Society for Genetic and Evolutionary Computation, and he chaired the genetic programming track of the GECCO 2000 conference. Contact him at the School of Cognitive Science, Hampshire College, Amherst, MA 01002; lspector@ hampshire.edu; hampshire.edu/lspector.



Christian Jacob is an assistant professor in the Department of Computer Science at the University of Calgary. His research interests focus on bio-inspired computing techniques and their concepts and applications. He is particularly interested in the interplay of evolutionary and developmental processes in nature. He received his MSc and PhD in computer science, both from the University of Erlangen-Nuremberg, Germany. He is a member of the IEEE and ACM. Contact him at the Dept. of Computer Science, Univ. of Calgary, 2500 University Dr. N.W., Calgary, Alberta, T2N 1N4 Canada; jacob@cpsc.ucalgary.ca; www.cpsc.ucalgary.ca/~jacob.

mations, finding the order can be quite a difficult task.

This task, although extremely difficult even for humans with an intimate knowledge of the code, is extremely suitable for GP. Programmers have evolved sequences of these transformations, rather than actual code, with considerable success. GP is particularly suitable for generating parallel code, because it eagerly embraces the maze of transformations required for reengineering. Furthermore, the often lateral approach required for parallel program design, although a foreign way of thinking for many programmers, is tailor made for GP's bottom-up approach.

Like other systems that evolve sequences of transformations, such as John Koza's electrical circuits and Frederic Gruau's neural nets, we can look at Paragen as embryonic. That is, it starts with a serial program, and progressive application of the transformations modify it until eventually the system produces a parallel version of the program. It is only after this embryonic stage that an individual (that is, a potential solution) is tested.

All the transformations GP users employ are the standard parallel type and syntax with one caveat—the area of the program that a transformation affects does not contain any data dependencies. If a transformation violates that condition while being applied, it might change the semantics, and any transformation that runs this risk reduces an individual's fitness.

When calculating an individual's fitness, the fitness function examines all the transformations to test if they have caused any dependency clashes. A rough measure of the program's speed produced by an individual is also calculated, which is simply the number of time steps it takes to run the program. We can only assume a rough measure because it is very difficult, if not impossible, to exactly measure a program's speed. However, one major advantage of a system such as this is that we can subsequently examine the transformations employed and prove that they maintain the correctness of the program.

GP is suitable for a task such as this because when we attempt to understand the logic of a program and then convert that logic to an equivalent parallel form, the complexity grows enormously as the program increases in size. However, as GP applies its transformations without any understanding of the logic of the program it is modifying, it scales far more graciously than many other methods.

Moreover, while human programmers might have the advantage of being able to take a more holistic view of a program, GP is not only able to spot and exploit any patterns in the code but can also take advantage of its bottom-up approach. It can move statements around, swap the location of loops, and even join previously transformed loops together in ways that human programmers are unlikely to think of. The system can concentrate solely on the program's layout, because it doesn't need to solve the functionality problem. That is, a human programmer has already implemented the logic required to carry out the task with which the code is concerned.

Due to the arbitrary way in which GP cuts and divides the code and, in particular, the loops, we can't guarantee the readability of the modified code. In general, though, people are no more concerned about this than they are about being able to interpret the object code files from a compiler, so this is not a problem. There might be cases, however, where the code's end user will want to be able to read it—for example, if the programmer wants to add some extra, hardware-specific optimizations to the code. If this is strictly necessary, we can direct the loop transformations to insert comments wherever the system modifies a loop, thus providing an automatically documented piece of code.

Transformation-based GP differs from traditional GP in that it assumes the existence of some sort of embryonic structure that it can progressively modify into a more acceptable state. In some instances, such as in the work of Koza and Gruau, the embryo contains little if any functionality. However, there is no reason why the embryo can't be a fully functioning program, or possibly even a reasonably satisfactory solution generated by hand, and then passed to GP for further optimization.

All of this is a further pointer to the potential of GP to be looked at by mainstream programmers as yet another tool at their disposal. GP does not claim to replace programmers nor is it restricted to certain problem domains that are conducive to its success. We can use GP for virtually any reengineering task; it is simply a matter of correctly isolating the part of the code that is most likely to benefit from its application.

The evolution of arbitrary computational processes

Lee Spector, Hampshire College

We can view genetic programming as the use of genetic algorithms to evolve computational processes in the form of computer programs. GAs solve problems by manipulating populations of potential solutions that are interpreted in different ways depending on the application, while GP usually treats the individuals in the population as explicit computer programs written in a subset or variant of a conventional programming language. GP maintains the GA's overall algorithm: the search process proceeds by iteratively evaluating the fitness of the individuals in the population and by applying genetic operators such as crossover and mutation to the higher-fitness individuals so as to explore other promising areas of the search space. In GAs more generally the fitness evaluation step can take many forms, while in GP an individual is evaluated for fitness at least in part by executing the program and assessing the quality of its outputs.

GP techniques have proven valuable for the evolution of structures other than computer programs (for example, neural networks and analog electrical circuits), but the emphasis on individuals as literal computer programs is one of GP's central defining features.

Computational universality

The computational power of the set of elements from which GP can construct programs—the *function set* and *terminal set* in the terminology of the field or the *primordial ooze* in less formal parlance—determines the range of computational processes that GP can potentially evolve. In the most frequently cited examples, this range is quite narrow. For example, in standard symbolic regression problems, in which the goal is to evolve a program that fits a provided set of numerical data, the evolving programs draw their components from an ooze that contains numerical functions but no mechanisms for conditional or iterative execution. In many other frequently cited problems, researchers make only a small set of domain-specific functions available, providing nothing approaching computational universality.

However, early work in the field showed how we can generalize the potential computational structures by including conditionals,

implicit iteration (in which the entire evolved program is executed repeatedly), and explicit iteration (with time-out bounds and other mechanisms to prevent infinite looping).[1] In 1994, Astro Teller showed that we could achieve Turing completeness by adding a potentially unbounded indexed memory; we could then, in principle, evolve any Turing-computable function.[2]

A different dimension along which we can generalize programs concerns not the absolute computational power of the representations but rather the ease with which commonly employed programming paradigms can be expressed. Most human programmers are not content to program with machine code, even though it is Turing complete. Key items in human programmers' toolkits are mechanisms that let them easily create reusable subroutines, specialized control structures, and data structures.

Recent work in GP has shown how all of these elements can be brought under evolutionary control. *Automatically defined functions* let evolving programs define subroutines and call them from within the main program or from within other ADFs. *Architecture altering operations* let the evolutionary process dynamically explore different program architectures (where *architecture* means the number of subroutines and parameters for each subroutine) as evolution proceeds.[3] *Automatically defined macros* let evolving programs define new iterative and conditional control structures in a manner analogous to ADFs.[4] Other work shows how GP can use rich type systems[5] and how the GP process can implement new data structures during evolution.[6] Further research has explored recursion as an alternative to iteration and various ways in which other elements of the functional programming paradigm can be brought under evolutionary control.

With all of these enhancements, it would seem that the computational world is GP's oyster and that arbitrary computational processes should be well within its reach. However, there are two problems with this optimistic assessment:

- Just because we can construct a desired program out of the provided raw materials does not mean that the evolutionary process will produce it. Adding unnecessary computational power or flexibility generally increases the search-space size, thus also increasing the work that we must do to find the desired program. On the other hand, it is often difficult to determine the minimum required power or flexibility.
- Recent results from physics and the theory of computation show that Turing completeness, as traditionally defined, does not capture the full range of physically possible computational processes. In particular, quantum computers can perform certain computations with lower computational complexity than they can be performed on a Turing machine or on any other classical computer. For full complexity-theoretic universality, we must let the evolving programs perform quantum computations.

With all of these enhancements, it would seem that the computational world is GP's oyster and that arbitrary computational processes should be well within its reach.

Although the first problem is fundamental and the subject of much current research, it is beyond the scope of this essay. The second problem has recently been tackled, and I outline its solution here.

Evolving quantum programs

Quantum computers are devices that use the dynamics of atomic-scale objects to store and manipulate information. Only a few small-scale quantum computers have been built so far, and there is debate about when, if ever, large-scale quantum computers will become a reality. Quantum computing is nonetheless the subject of widespread interest and active research. The primary reason for this interest is that quantum computers, if built, will be able to compute certain functions more efficiently than is possible on any classical computer. For example, Peter Shor's quantum factoring algorithm finds the prime factors of a number in polynomial time, but the best known classical factoring algorithms require exponential time. Lov Grover provided another important example, showing how a quantum computer can find an item in an unsorted list of n items in $O(\sqrt{n})$ steps, while classical algorithms require $O(n)$ steps. An earlier Trends & Controversies provided a brief introduction to the core ideas of quantum computing,[7] and Julian Brown has written a more complete, book-length introduction that is accessible to general readers.[8]

Because practical quantum computer hardware is not yet available, we must test the fitness of evolving quantum algorithms using a quantum computer simulator that runs on conventional computer hardware. My research group at Hampshire College, consisting of myself and physicists Herbert J. Bernstein and Howard Barnum, has developed a quantum computer simulator specifically for this purpose. Classical simulation of a quantum computer necessarily entails an exponential slowdown, so we must be content to simulate relatively small systems—but even with small systems much can be accomplished.

Our simulator, QGAME (quantum gate and measurement emulator), represents quantum algorithms using the *quantum gate array* formalism. In this formalism, computations are performed at the quantum bit (qubit) level, so they are similar in some ways to Boolean logic networks. A major difference, however, is that the state of the quantum system at any given time can be a superposition of all possible states of the corresponding Boolean system. For each classical state, we store a complex-valued *probability amplitude*, which we can use to determine the probability that we will find the system to be in the given classical state if we measure it. (In accordance with quantum mechanics, squaring the absolute value of the amplitude determines the probability.) Quantum gates are implemented as matrices that multiply the vector of probability amplitudes for the entire quantum system.[9,10]

QGAME also permits a program to measure the value of a qubit and branch to different code segments depending on the measurement result. Such measurements necessarily collapse the superposition of the measured qubit. QGAME always follows both branches, collapsing the superpositions appropriately in each branch and keeping track of the probabilities that the computer would reach each subsequent string of gates.

We can diagram QGAME programs in a manner analogous to classical logic circuits,

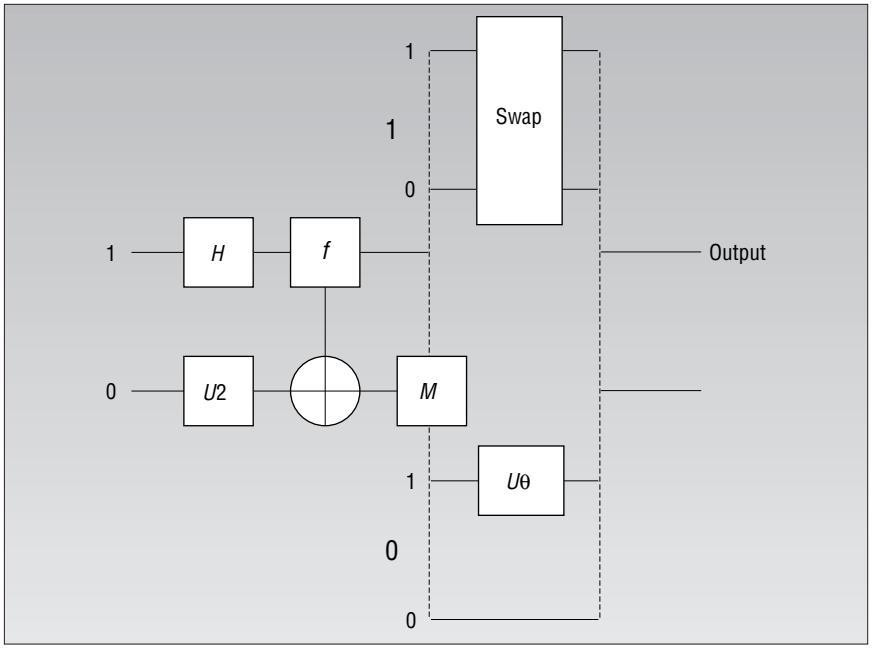


Figure 3. The gate array diagram for an evolved quantum program for the *OR* problem of determining whether *f*, a black-box Boolean function, answers "1" (and thereby performs a "not" on qubit 0) for the "0" input, the "1" input, or both. *H* is a Hadamard gate, *U*2 and *Uq* are single-qubit rotations (the parameters of which are shown in Figure 4), and Swap is a 2-qubit gate that swaps the values of its two input qubits. *M* is a measurement gate; if measurement yields a value of 1, then the upper branch will be executed, otherwise the lower branch will be executed. More details on the implementation of the gates are available in the cited references.[9,10]

```
  ;; start in the state |00>
   ;; apply a Hadamard gate to qubit 1
(HADAMARD 1)
   ;; apply a U2 rotation to qubit 0, with parameters:
   ;;  PHI=-pi THETA=9.103027 PSI=pi/7 ALPHA=0
(U2 0 ,(- pi) 9.103027 ,(/ pi 7) 0)
   ;; call f with qubit 1 as input and qubit 0 as output
(F 1 0)
   ;; measure qubit 0, collapsing the superposition
(MEASURE 0)
   ;; this is the branch for qubit 0 measured as "1"
   ;; swap qubits 0 and 1
(SWAP 1 0)
   ;; this marks the end of the "1" branch
(END)
   ;; this is the branch for qubit 0 measured as "1"
   ;; apply a U-THETA rotation to qubit 1 with THETA=pi/4
(U-THETA 1 ,(/ pi 4))
   ;; end of evolved algorithm
   ;; read result from qubit 1
```

Figure 4. Textual listing of the evolved quantum program in Figure 1.

as Figure 3 shows. Such diagrams can be deceptive, however; unlike classical logic gate arrays, in quantum gate arrays, the values traveling on different wires may be entangled with one another so that measurement of one can change the value of another. Textually, QGAME programs are represented as sequences of gate descriptions and structuring primitives, as Figure 4 shows.

To apply GP to the evolution of quantum programs, we provide QGAME elements as the raw materials and use QGAME as the execution engine for fitness evaluation. The program shown in Figures 3 and 4 is a simplified version of a program our GP system produced in this way. This program solves the *OR* problem of determining whether the black-box one-input Boolean function *f* answers "1" for the "0" input, the "1" input, or both. It does this using only one call to *f* and with a probability of error of only 1/10, which is impossible using only classical computation. (Classical probabilistic computation can achieve an error probability no lower than 1/6.) This result, that quantum computing is capable of solving the *OR* problem with only one call to *f* and an error probability of only 1/10, was first discovered with the aid of GP (using an earlier version of our system).

Evolving arbitrary computational processes

With the addition of quantum computing primitives and quantum simulation for fitness assessment, GP is, in principle, capable of evolving any physically implementable computational process. It has already "rediscovered" several better-than-classical quantum algorithms and made a couple of new discoveries about the nature of quantum computing. The full range of physically computable functions is now within the scope of GP, which is beginning to find interesting new programs that humans had not previously discovered.

But, as mentioned earlier, computational power is a double-edged sword. Even within classical domains, we are often risking long periods of evolutionary drift and stagnation if too much computational power is provided —for example, in the form of unnecessary memory capacity or control structures. Quantum computation provides power even beyond that of a Turing machine, and the dangers are therefore even greater. The task remains to sufficiently understand the evolutionary dynamics of GP so we can avoid getting lost in the enormous search space of possible quantum computations.

The practical strategy that most GP practitioners follow is to use our intuitions to make reasonable guesses about the demands of the problems we are attacking and to provide little more than the required computational power. For example, many researchers limit the arithmetic functions in the function set for symbolic regression problems to those that they think might be needed, and few would include iteration structures, conditionals, or dynamic structuring mechanisms such as ADFs or ADMs

unless they have good reason to believe that they would be well utilized. Similarly, we often limit the number and type of quantum gates that can be included in evolving quantum programs, and we have begun to work on hybrid classical and quantum algorithms with limited quantum components.

This tension between universality and constraint, between the potential to produce any arbitrary computational process and the need to limit the evolutionary search space to one that we can explore in a reasonable amount of time, is a critical issue for GP's future. Any advances that reduce the need for human intuition in resolving this tension will significantly increase GP's applicability, particularly in application areas (such as quantum computing) for which the representational and computational power requirements are not immediately obvious.

References

1. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, Mass., 1992.
2. A. Teller, "The Evolution of Mental Models," *Advances in Genetic Programming*, K.E. Kinnear, Jr., ed., MIT Press, Cambridge, Mass., 1994, pp. 199–219.
3. J.R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge, Mass., 1994.
4. L. Spector, "Simultaneous Evolution of Programs and their Control Structures," *Advances in Genetic Programming 2*, P. Angeline and K. Kinnear, eds., MIT Press, Cambridge, Mass., 1996, pp. 137–154.
5. T. D. Haynes, D.A. Schoenefeld, and R.L. Wainwright, "Type Inheritance in Strongly Typed Genetic Programming," *Advances in Genetic Programming 2*, P. Angeline and K. Kinnear, eds., MIT Press, Cambridge, Mass., 1996, pp. 359–375.
6. W.B. Langdon, "Data Structures and Genetic Programming," *Advances in Genetic Programming 2*, P. Angeline and K. Kinnear, eds., MIT Press, Cambridge, Mass., 1996, pp. 395–414.
7. H. Hirsh, "A Quantum Leap for AI," *IEEE Intelligent Systems*, Vol. 14, No. 4, July/Aug. 1999, pp. 9–16.
8. J. Brown, *Minds, Machines, and the Multiverse: The Quest for the Quantum Computer*, Simon & Schuster, New York, 2000.
9. L. Spector et al., "Quantum Computing Applications of Genetic Programming," *Advances in Genetic Programming 3*, L. Spector et al., eds., MIT Press, Cambridge, Mass., 1999.
10. L. Spector et al., "Finding a Better-than-Classical Quantum AND/OR Algorithm using Genetic Programming," *Proc. 1999 Congress Evolutionary Computation*, IEEE Press, Piscataway, N.J., 1999, pp. 2239–2246.

The art of genetic programming

Christian Jacob, University of Calgary, Canada

During the past 10 years, there has been a revolution in the understanding of natural development processes—the processes whereby a simple, undifferentiated egg turns into a complex adult organism—also known as *epigenesis*. Only 15 years ago, this process was identified as "one of the major problem areas of biology along with how the brain works."[1] This progress in reproduction and development results, to a large extent, from applying the ideas and techniques of genetics. Genes carry the instructions (the "developmental programs," in computer science terms) for making an organism. The Human Genome Project and related projects are currently providing key information for understanding development and evolution. Soon, we will know the complete contents of what is actually written on selected natural genomes (for example, the yeast, the fruit fly Drosophila, the frog Xenopus, and the human genome).

Unfortunately, being able to identify only words, sentences, and paragraphs without knowing the semantics and sentence structure does not put us in an advanced position to increase our understanding of developmental processes. Therefore, getting to know the syntax is a major step toward improving our knowledge of genome programs.

Evolution and development

What mechanisms in nature produced the developmental programs that convert an egg into an adult organism? It is evolution that "on an enormously longer time scale, [has] converted simple single-celled ancestors into the existing array of multicellular animals and plants."[1] Consequently, in both developmental and evolutionary processes, a single cell—nature's primary building block—converts into a complex 3D structure, an organism with many highly differentiated cells. Although the final results of developmental and evolutionary processes are similar, their working mechanisms seem to be entirely different. Natural selection shapes rules, which in turn drive development. Therefore, there is an intricate connection between evolution and development in nature—an interplay that has not yet been sufficiently integrated into evolutionary algorithms and, especially, into genetic programming.

Since the 1950s, evolutionary algorithms have been implementing a wide range of different aspects of natural evolution, from evolution on a species level down to the level of genes. In particular, it has always been one of the primary features of genetic algorithms (GAs) to glean key mechanisms from genomes, their way of representing information (encoded by a discrete alphabet), their techniques to mutate and recombine information, and the selection procedures under which they have to adjust for better survival strategies. Could GP, as an extension of GAs, help accomplish a better understanding of gene–gene interactions, genotype–phenotype mappings, and epigenesis?

GP might lead us to more sophisticated models of gene–gene interactions, interactions among cells, pattern and structure formation in 2D and 3D, and, hence, decentralized control mechanisms that lead to self-organization and evolution of complexity.

GP and development

Let us compare the way in which nature builds its organisms and how engineers and computer scientists build their products. Nature uses a developmental approach—instead of assembling separate parts, it grows its organisms. We use a compositional approach: we assemble premanufactured parts into a working machine; we do not use developmental procedures. In essence, we do not (yet) know how to grow a complex device like a computer. Starting from a blueprint, we use an instruction manual to perform the assembling.

Within the last decade, a few approaches, combining concepts of evolution and epigenesis, have appeared in evolutionary com-

putation, most of them closely related to GP. One of the first was Frederic Gruau's GP system to evolve modular neural network architectures.[2] The novelty of this approach lies in the application of growth programs, instead of using parametrized blueprints to specify neural network architectures. Starting from an initial embryonic network, the structure is elaborated in a step-by-step manner, following an evolvable set of instructions for subdivision, reconnection, or modularization of the network, for example.

John Koza and his coworkers apply a similar embryonic technique to evolve analog electrical circuits.[3] It is also possible to use Lindenmayer systems to encode pattern formation and growth processes in branching structures to simulate the evolution and coevolution of plant ecosystems.[4,5] Again, the key point is the evolution of growth programs (for example, dynamic, morphogenetic rules instead of a static blueprint approach). In the context of self-replicating systems, much research has been performed over the last 50 years.[6] In one of the latest applications, using *cellular automata*, James Reggia, Jason Lohn, and Hui-Hsien Chou demonstrate the evolvability of simple but arbitrary structures that replicate from nonreplicating components.[7] Here, too, locally interacting rules implicitly describe emergent pattern formation of the overall system.

From a holistic point of view, complex patterns can emerge in dynamic systems without any specific global instructions regulating the development of particular parts. This alternative approach toward an emergence of complex structural and computational patterns is represented by the notion of self-organization, an active research area today.[8,9] Agent-based, massive, parallel, and decentralized systems might provide an appropriate level of abstraction, where local interaction rules determine agent behavior, from which the overall system behavior emerges.[10] In conjunction with GP used to evolve agent-behavior programs, these "swarm intelligence" systems have the potential to enhance our understanding of developmental processes and their evolution. This proposed Swarm GP approach would also lead us toward alternative ways of modeling biological systems, from the level of complex ecosystems, individuals, and cells, down to the level of genomes.

The "programming" of development

Returning to the gene level, we must clarify key questions in developmental biology to justify simulation models of epigenesis. Can we interpret genes as a set of instructions? Do genes actually correspond to programs? As the developmental biologist Enrico Coen pointed out in his recent book *The Art of Genes*, the notion of pure instructions, encoded on a genome, does not work for genes.[11] For epigenetic processes in nature, the maker (the developmental process) and the made (the organism) are interdependent. While genes are building on the activities of other genes, in the course of the development, the changing frames of reference become particularly important. Therefore, developing an organism is a process of elaboration and refinement, a highly interwoven process with intricate feedback loops and no clear separation between the plan (program) and its execution.

If we intend to design GP-based models that incorporate developmental processes, we must rethink our notion that a program and its execution are separate.

What does that mean for GP? It means that if we intend to design GP-based models that incorporate developmental processes, we must rethink our notion that a program and its execution are separate. The self-reproducing cellular-automata systems incorporate key characteristics that follow the picture of the maker and the made as interconnected. CA cells change their states in response to the states of other, neighboring cells. That is, cells act as programs. On the other hand, cells also provide the environment for other cells. So, there is a constant interplay between the maker (the CA rules) and the made (the structure developing on the CA grid).

If we succeed in a raising these notions to a higher abstraction level, where cells are not fixed in size, shape, and location, where agents that implement any developing units replace cells, and where we can have more than local interactions, we might have the right computational tools to design complex, developing systems using evolution.

Still a lot to learn from nature

Growing designs for complex machines, like today's computers, will dramatically change the way of interacting and programming these "evo-computers." Evolvable hardware systems already set the pace in this direction.[12] GP in conjunction with the applied principles of the "art of genes" in natural systems are key ingredients toward building exciting new computational tools, both in software and in hardware. It is time to get them growing and evolving. ■

References

1. J.M. Smith, *Shaping Life—Genes, Embryos and Evolution*, Weidenfeld & Nicolson, London, 1998.
2. F. Gruau, and D. Whitley, "Adding Learning to the Cellular Development of Neural Networks: Evolution and the Baldwin Effect," *Evolutionary Computation*, Vol. 1, No. 3, 1993, pp. 213–233.
3. J.R. Koza et al., *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufmann, San Francisco, 1999.
4. C. Jacob, "Evolving Evolution Programs: Genetic Programming and L-Systems," *Genetic Programming 1996*: *First Ann. Conf.*, MIT Press, Cambridge, Mass.1996.
5. C. Jacob, *Illustrating Evolutionary Computation with Mathematica*, Morgan Kaufmann, San Francisco, 2000.
6. M. Sipper, "Fifty Years of Research on Self-Replication: An Overview," *Artificial Life*, Vol. 4, No. 3 1998, pp. 237–257.
7. J.A. Reggia et al., "Self-Replicating Structures: Evolution, Emergence, and Computation," *Artificial Life*, Vol. 4, No. 3, 1998, pp. 283–302.
8. P. Bak, *How Nature Works—The Science of Self-Organized Criticality*, Springer Verlag, New York, 1996.
9. J.H. Holland, *Emergence: From Chaos to Order*, Addison-Wesley, Reading, Mass., 1998.
10. E.M. Bonabeau et al., *From Natural to Artificial Swarm Intelligence*, Oxford Univ. Press, Oxford, 1999.
11. E. Coen, *The Art of Genes—How Organisms Make Themselves*, Oxford Univ. Press, Oxford, 1999.
12. M. Sipper and E.M.A. Ronald, "A New Species of Hardware," *IEEE Spectrum*, Mar. 2000, pp. 59–64.