The evolution of arbitrary computational processes

To appear in *IEEE Intelligent Systems*

Lee Spector
School of Cognitive Science
Hampshire College
Amherst, MA 01002, USA
lspector@hampshire.edu

May 1, 2000

Genetic programming (GP) can be viewed as the use of genetic algorithms (GAs) to evolve computational processes in the form of computer programs. In GAs more generally, the individuals in a population may be represented and interpreted in a variety of ways, while in GP the individuals are usually treated as explicit computer programs written in a subset or variant of a conventional programming language. The overall algorithm of the GA is maintained in GP: search 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 in order 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 by assessing the quality of its outputs. GP techniques have proven valuable for the evolution of structures other than computer programs (e.g. neural networks [1] and analog electrical circuits [2]), but the emphasis on individuals as literal computer programs is the most central defining feature of GP.

1 Computational Universality

The computational power of the set of elements out of which programs may be constructed — 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 can potentially be evolved by GP. In the most frequently cited examples this range is actually 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 only a small set of domain-specific functions are made available, providing nothing approaching computational universality. But early work in the field showed how one could 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) [3]. In 1994 Teller showed that Turing completeness could be achieved with the addition of a potentially unbounded indexed memory; any Turing-computable function could then be evolved in principle [4].

A different dimension along which programs may be generalized concerns not the absolute computational power of the representations but rather the ease with which commonly employed programming paradigms can be evolved. Most human programmers are not content to program with machine code, Turing complete though it may be. Key items in the human programmer's toolkit are mechanisms that allow for the creation of reusable subroutines, specialized control structures, and data structures. More recent work in GP has shown how all of these elements can be brought under evolutionary control. *Automatically defined functions* (ADFs) allow evolving programs to define subroutines and to call them from within the main program or from within other ADFs, and *architecture altering operations* allow the evolutionary process to dynamically explore different program architectures (where "architecture" means the number of subroutines and the number of parameters for each subroutine) as evolution proceeds [5]. *Automatically defined macros* (ADMs) allow evolving programs to define new iterative and conditional control structures in a manner analogous to ADFs [6]. Additional work has shown how GP can make use of rich type systems [7] and how new data structures can be implemented by the GP process during evolution [8]. Further research has explored the use of recursion as an alternative to iteration and various ways in which other elements of the functional programming paradigm can be brought under evolutionary control [9].

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. But there are two problems with this optimistic assessment:

1. Just because a desired program *can* be constructed out of the provided raw materials it does not follow that it *will* be produced by the evolutionary process. In fact one will generally increase the size of the search space, and therefore also the amount of work that must be done to find the desired program, when one adds unnecessary additional computational power or flexibility. On the other hand it is often difficult to determine the minimum required power or flexibility.

2. 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. If one wants full complexity-theoretic universality then one must allow the evolving programs to perform quantum computations.

Problem 1 is deep, fundamental, the subject of much current research, and beyond the scope of this essay to discuss in any detail (but see, for example, many sections of [10]). Problem 2 has recently been tackled and I outline its solution in the following sections.

2 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. But 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, Shor's quantum factoring algorithm finds the prime factors of a number in polynomial time, while the best known classical factoring algorithms require exponential time [11]. Another important example was provided by Grover, who showed how a quantum computer could find an item in an unsorted list of *n* items in $O(\sqrt{n})$ steps, while classical algorithms require $O(n)$ steps [12]. A brief introduction to the core ideas of quantum computing was provided in an earlier *Trends & Controversies* section of *IEEE Intelligent Systems* [13]; a more complete book-length introduction that is nonetheless accessible to general readers was recently written by Brown [14].

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. This entails an exponential slowdown, so we must be content to simulate relatively small systems.

Our quantum 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* that can be used 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 the probability is determined by squaring the absolute value of the amplitude.) Quantum gates are implemented as matrices that are multiplied by the vector of probability amplitudes for the entire quantum system; see [15, 16] for details.

QGAME also allows one to measure the value of a qubit and to 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 gate.

QGAME programs can be diagrammed in a manner analogous to classical logic cir-

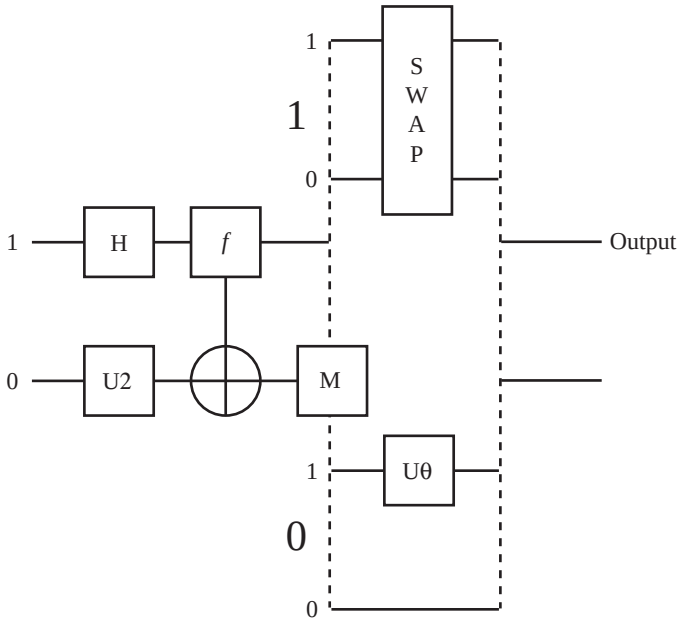

Figure 1: The gate array diagram for an evolved quantum program for the *OR* problem.

cuits, as shown in Figure 1. Such diagrams can be deceptive, however; unlike classical logic gate arrays, in quantum gate arrays the values travelling 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 shown in Figure 2.

To apply GP to the evolution of quantum programs we simply provide QGAME elements as the raw materials and use QGAME as the execution engine for fitness evaluation. The program shown in Figures 1 and 2 is a simplified version of a program that was produced by our GP system in this way. This program solves the *OR* problem of determining whether the "black box" one-input Boolean function *f* answers "1" for either the input of "0" or the input of "1" or both. It does this using only one call to *f* and with a probability of error of only $\frac{1}{10}$, which is impossible using only classical computation. (Classical probabilistic computation can achieve an error probability no lower than $\frac{1}{6}$). This result, that quantum computing is capable of solving the *OR* problem with only one call to *f* and an error probability of only $\frac{1}{10}$, was first discovered with the aid of GP (using an earlier version of our system).

```
     ;; 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 2: Textual listing of the evolved quantum program in Figure 1.

3 The evolution of 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. In fact it has already "rediscovered" several better-than-classical quantum algorithms and it has 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, and GP is beginning to find interesting new programs that had not previously been discovered by humans.

But as mentioned above, computational power is a double-edged sword. Even within classical domains one is 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 understand the evolutionary dynamics of GP sufficiently well that we can avoid getting lost in the enormous search space of possible quantum computations.

The practical strategy that we have followed, and 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/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 can explored in a reasonable amount of time, is a critical issue for the future of GP. Any advances that reduce the need for human intuition in resolving this tension will significantly increase the applicability of GP, particularly in application areas (like quantum computing) for which the representational and computational power requirements are not immediately obvious.

References

[1] F. Gruau, "Genetic micro programming of neural networks," in *Advances in Genetic Programming*, K. E. Kinnear Jr., Ed., pp. 495–518. MIT Press, 1994.

[2] J. R. Koza and F. H Bennett, III, "Automatic synthesis, placement, and routing of electrical circuits by means of genetic programming," in *Advances in Genetic Programming 3*, Spector, Langdon, O'Reilly, and Angeline, Eds. MIT Press, 1999.

[3] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[4] A. Teller, "The evolution of mental models," in *Advances in Genetic Programming*, K. E. Kinnear Jr., Ed., pp. 199–219. MIT Press, 1994.

[5] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, 1994.

[6] L. Spector, "Simultaneous evolution of programs and their control structures," in *Advances in Genetic Programming 2*, P. Angeline and K. Kinnear, Eds., pp. 137–154. MIT Press, 1996.

[7] T. D. Haynes, D. A. Schoenefeld, and R. L. Wainwright, "Type inheritance in strongly typed genetic programming," in *Advances in Genetic Programming 2*, P. Angeline and K. Kinnear, Eds., pp. 359–375. MIT Press, 1996.

[8] W. B. Langdon, "Data structures and genetic programming," in *Advances in Genetic Programming 2*, P. Angeline and K. Kinnear, Eds., pp. 395–414. MIT Press, 1996.

[9] G. T. Yu, *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming*, Ph.D. thesis, University of London, 1999.

[10] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*, Morgan Kaufmann, dpunkt.verlag, 1998.

[11] P. W. Shor, "Quantum computing," *Documenta Mathematica*, vol. Extra Volume ICM, pp. 467–486, 1998.

[12] L. K. Grover, "Quantum mechanics helps in searching for a needle in a haystack," *Physical Review Letters*, pp. 325–328, 1997.

[13] H. Hirsh, "A quantum leap for AI," *IEEE Intelligent Systems*, pp. 9–16, July/August 1999.

[14] J. Brown, *Minds, Machines, and the Multiverse: The quest for the quantum computer*, Simon & Schuster, 2000.

[15] L. Spector, H. Barnum, H. J. Bernstein, and N. Swamy, "Quantum computing applications of genetic programming," in *Advances in Genetic Programming 3*, Spector, Langdon, O'Reilly, and Angeline, Eds., pp. 135–160. MIT Press, 1999.

[16] L. Spector, H. Barnum, H. J. Bernstein, and H. Swamy, "Finding a better-than-classical quantum and/or algorithm using genetic programming," in *Proceedings of the 1999 Congress on Evolutionary Computation*. 1999, pp. 2239–2246, IEEE Press.