Spector, L., H. Barnum, and H.J. Bernstein. 1998. Genetic Programming for Quantum Computers. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, edited by J.R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, and R.L. Riolo. pp. 365-374. San Francisco, CA: Morgan Kaufmann.

Genetic Programming for Quantum Computers

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

Howard Barnum
hbarnum@hampshire.edu

Herbert J. Bernstein
hbernstein@hampshire.edu

School of Natural Science
and Institute for Science and Interdisciplinary Studies (ISIS)
Hampshire College
Amherst, MA 01002

ABSTRACT

Genetic programming can be used to automatically discover algorithms for quantum computers that are more efficient than any classical computer algorithms for the same problems. In this paper we exhibit the first evolved better-than-classical quantum algorithm, for Deutsch's "early promise" problem. We also demonstrate a technique for evolving scalable quantum gate arrays and discuss other issues in the application of genetic programming to quantum computation and vice versa.

1. Quantum Computing

Quantum computers are computational devices that use atomic-scale objects, for example 2-state particles, to store and manipulate information (Steane, 1997; for an elementary on-line tutorial see Braunstein, 1995; for an introduction for the general reader see Milburn, 1997). The physics of these devices allows them to do things that common digital (henceforth "classical") computers cannot. Although quantum computers and classical computers appear to be bound by the same limits of Turing computability, physicists argue that quantum computers can solve certain problems using less resources (time and space) than classical computers are thought to require (Jozsa, 1997). For example, Shor's quantum algorithm finds the prime factors of an n-digit number in time $O(n^3)$, while the best known classical factoring algorithms require at least time $O(2^{n^{1/3}\log(n)^{2/3}})$ (Shor, 1994; Beckman et al. 1996). And Grover's quantum database search algorithm can find an item in an unsorted list of n items in $O(\sqrt{n})$ steps, while classical algorithms clearly require $O(n)$ (Grover, 1997). The full power of quantum computation is a subject of active investigation.

The smallest unit of quantum information is the *qubit*, which is analogous to the classical *bit*. Whereas a classical system of n bits is at any time in one of 2^n states, a quantum system of n qubits can be in any linear superposition of these 2^n states simultaneously. Although we cannot read the entire state (because measurement interferes with the system), it appears that this *quantum parallelism* can nonetheless be harnessed to perform real computational work.

In physical terms, a qubit can be thought of as a two-component wave, where each component represents a classical value, 0 or 1. The height of a component wave gives the probability that the qubit will be found in a particular classical state, and the phase controls how the wave will interfere with other waves. As usual, a wave with height and phase can be represented by a single complex number. Unlike a classical bit, a qubit can be in both states at the same time, and these states may be in phase, out of phase, or somewhere in between, leading to constructive or destructive interference.

To date, only very small quantum computers have been built, but the planning and construction of larger devices is in progress. Current experimental quantum computing hardware is based on the use of ion traps, cavity QED, and NMR techniques; many difficult problems must be solved before these techniques can be scaled up, but a discussion of these issues is beyond the scope of this paper (see Preskill, 1997).

Quantum computers are different from classical computers in several ways, and it is not obvious how, in general, software can be developed to take advantage of their non-classical power. A variety of basic questions about this power are still open, for example whether or not there exist polynomial time quantum algorithms for classically NP complete problems. In order to assess the wisdom of expending resources on the physical realization of quantum computers, it is important that we develop a more solid understanding of their real computational powers. One way to further this understanding is to develop more quantum algorithms, either by hand or by use of automatic programming techniques (as below). And there are additional reasons, aside from guidance for research and development efforts, for wanting to determine the real powers of quantum computation. For example, Penrose has argued that quantum effects play an important role in human brains (Penrose, 1994), although his claims have been disputed. Because complexity-theoretic arguments play a major role in cognitive science, a rethinking of the computational complexity limits of brain processes could have a significant impact on the study of human cognition.

In this paper we describe how genetic programming can be used to automatically find successful quantum algorithms. Space limitations prevent us from describing the basic genetic programming technique here; readers unfamiliar with genetic programming should consult (Koza 1992, 1994). In the following sections we first describe how quantum computers are simulated for the purposes of fitness evaluation in a genetic programming system. We then document the evolution of a better-than-classical quantum algorithm for Deutsch's early promise problem and show how our technique can also be used to evolve scalable quantum gate arrays. This is followed by a brief outline of our current research and comments about the prospects for future results.

2. Classical Simulation of Quantum Computation

The fitness of a program in a genetic programming system is assessed in part by running the program and by observing its behavior. Because we do not yet have physical quantum computers at our disposal, we can only assess the fitness of quantum algorithms by simulating a quantum computer. Unfortunately, this simulation requires computational resources that scale exponentially with the size of the simulated quantum system. For this reason simulation is only feasible for very small systems.

The state of an n-qubit system can be represented as a unit vector of 2^n complex numbers $[\alpha_0\alpha_1\alpha_2...\alpha_{2^n-1}]$. Each of these numbers can be viewed as paired with a computational basis vector of the form $|b_0b_1b_2...b_{n-1}\rangle$ where each b_i is an element of {0, 1}. (The "|...>" means that this is a "ket" vector; the mathematics of this are beyond the scope of this paper, but we are using the notation that is standard in the quantum computation literature.) The modulus squared of each α, i.e. $|\alpha_n|^2$, represents the probability that measurement of the system will find it in the state corresponding to $|n\rangle$. The unit-length condition ensures that these probabilities add to one. As an example, the complete state of a two-qubit system can be represented as a vector space of the following form:

$$\alpha_0|00\rangle + \alpha_1|01\rangle + \alpha_2|10\rangle + \alpha_3|11\rangle$$

If we measure the system the probability that we will read the state of the system as "01" is $|\alpha_1|^2$.

A quantum algorithm is run by preparing the system in an initial state, subjecting the system to a sequence of operators, and then reading the final state of the system. It is usually required that the initial state is a computational basis vector (that is, one in which one α is 1 and all others are 0), that the final measurement be done in the computational basis, and that each gate involves no more than a few qubits; this ensures that the number of gates in the quantum circuit is a reasonable measure of computational complexity. The final state is read by squaring the modulus of each alpha, summing those that correspond to the same values for the output bits, and reporting the output bit pattern with the highest sum. There will usually be some indeterminacy in the reading of the final state, but it is possible to sufficiently reduce this indeterminacy either by running the system multiple times or by designing quantum algorithms that minimize the uncertainty. The operators must all be *unitary*—that is, they must be linear transformations on the vector space which are bijective and length-preserving (Barenco et al., 1995). The operators are often called *quantum logic gates* and sequences of these operators are often called either *gate arrays* or *quantum algorithms*. A very small set of quantum logic gates can be *complete* for quantum computation in the same sense that *NAND* is complete for classical computation; one can implement any quantum algorithm using only these primitive gates (Barenco et al., 1995, and references therein).

A simple example of a quantum gate is the quantum counterpart of classical *NOT*. Classical *NOT* inverts the value of a single bit, changing 0 to 1 and 1 to 0. Quantum *NOT* operates on a single qubit. In a one-qubit system (which has two αs, one for $|0\rangle$ and one for $|1\rangle$) the quantum *NOT* operation simply swaps the values of the two αs. That is, a single qubit system in the state $\alpha_0|0\rangle+\alpha_1|1\rangle$ will be transformed by quantum *NOT* into $\alpha_1|0\rangle+\alpha_0|1\rangle$.

It is sometimes convenient to represent quantum gates via the matrix form of the operator in the computational basis (which completely determines the operator). Quantum *NOT* can be represented in matrix form as $\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$, and its operation on a one qubit system can be shown as:

$$\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\times\begin{bmatrix}\alpha_0\\ \alpha_1\end{bmatrix}|0\rangle = \begin{bmatrix}\alpha_1\\ \alpha_0\end{bmatrix}|1\rangle$$

When applied to qubit i of a multi-qubit system, quantum *NOT* swaps the αs of each pair of basis vectors that differ from one another only in the ith position. For example, in a two-qubit system the application of quantum *NOT* to the rightmost qubit will swap the α of $|00\rangle$ with that of $|01\rangle$, and the α of $|10\rangle$ with that of $|11\rangle$. This is represented in matrix form in Figure 1.

Another useful quantum gate is *controlled NOT* (or *CNOT*), which takes two qubit indices as arguments; we will call these arguments *controller* and *controlled*. *CNOT* is an identity operation for basis vectors with 0 in the controller position, but it acts like quantum *NOT* applied to the controlled position for basis vectors with 1 in the controller position. For the case of a two-qubit system, with qubit 0 as the controller and qubit 1 as the controlled (we start counting with 0 from the leftmost position in the ket vectors), this can be notated in matrix form as in figure 2.

One can also look at *CNOT* as a gate with one input qubit (controller) and one output qubit (controlled). *CNOT* flips the state with respect to its output wherever its input is 1. By making the condition on this flipping more complex, possibly using more input qubits, we can construct analogous unitary transforms for any classical boolean function. For example, consider classical *NAND*, which takes two input bits and outputs 0 if both inputs are 1, and 1 otherwise. That is, it has the truth table shown in Table 1.

Such a truth table can be used as the basis of a unitary transformation by interpreting a 1 in the output (rightmost) column of a particular row as an instruction to swap αs between each pair of basis vectors that match that row's values for the input qubits and differ only in their values for the output qubit. That is, we can construct a unitary transformation, called quantum *NAND*, that takes three qubit indices (2 inputs and 1 output) and swaps αs of all pairs of basis vectors that are equivalent with respect to their input qubits but differ in their output qubit, except for those for which both input qubits are 1 (the bottom row of the truth table). For a three-qubit system, with qubits 0 and 1 as inputs and qubit 2 as output, this can be notated in matrix form as shown in Figure 3.

The work described in this paper uses *CNOT*, quantum *NAND*, and the additional *Hadamard* and rotation quantum gates shown in Figures 4 and 5. To apply these operators to a particular qubit (in the case of Hadamard and the rotation) or to a pair or triple of qubits (*CNOT*, *NAND*) in a *n*-qubit system one first builds a 2^nx2^n matrix by taking the appropriate tensor products of the operator matrix and the identity matrix. One then multiplies the system state by the resulting matrix. We do not actually build the matrices in our implementation as they are large and mostly zeros, but this matrix formulation may be the clearest way to see what the operators do.

Table 1. The truth table for classical *NAND*.

A	B	A *NAND* B
0	0	1
0	1	1
1	0	1
1	1	0

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 1. The matrix form of a quantum *NOT* gate that operates on the second qubit of a two-qubit system.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 2. The matrix form of a quantum *CNOT* gate for a two-qubit system, using qubit 0 as the controller and qubit 1 as the controlled.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 3. The matrix form of a quantum *NAND* gate in a three-qubit system, using qubits 0 and 1 for input and qubit 2 for output.

$$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Figure 4. *H*: Hadamard gate on a single qubit.

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

Figure 5. U_θ: Unitary rotation of a single qubit by the angle θ.

3. Evolution of a Better-Than-Classical Quantum Algorithm for Deutsch's Early Promise Problem

Suppose you are given an oracle that computes an unknown binary function of n input bits. That is, it takes n bits as input and returns one bit output. Suppose further that you are promised that the function is either *uniform*, meaning that it always returns 0 or always returns 1, or that it is *balanced*, meaning that it will return an equal number of 0s and 1s if called on all possible inputs. The *early promise problem* is the problem of determining whether such an oracle is uniform or balanced.

It is easy to see that a deterministic classical algorithm will in the worst case require $\frac{2^n}{2}+1$ oracle calls to solve this problem. If the first $\frac{2^n}{2}$ calls all return the same value, then it is still possible that the oracle is either uniform or balanced, but the answer will be known for certain after one more oracle call. A probabilistic classical algorithm can do somewhat better, because it is unlikely that a balanced oracle will produce $\frac{2^n}{2}$ of the same value in sequence. Nonetheless, it is also clear that a *single* call to the oracle on a classical computer produces *no* information that can be helpful in solving the problem, whether deterministically or probabilistically—0 and 1 are both equally likely outputs from such a call whether the oracle is uniform or balanced.

Deutsch has shown that quantum computers can do better here (Deutsch, 1985; Costantini and Smeraldi, 1997). If the oracle is implemented as a unitary operator on a quantum computer's state, then information useful in solving the problem can be obtained using fewer oracle calls than would be required by any classical algorithm. In this section we use genetic programming to automatically discover a quantum algorithm that provides information on the two-bit early promise problem using only one oracle call.

Genetic Programming could be applied to the construction of quantum algorithms in a variety of ways. In the present work we have opted to evolve classical programs which, when executed, construct quantum gate arrays. The quantum gate arrays are then run a number of times (in this case once for each of the 8 two-bit oracles that are either uniform or balanced) to assess each program's fitness. This scheme is convenient for several reasons, one of which is that it naturally extends to the evolution of scalable gate arrays (see next section).

For the two-bit early promise problem we evolved gate arrays for a three-qubit quantum computer in which the qubits are referred to with the indices 0, 1, and 2. For each run the quantum computer is prepared in the initial state of 1|000>, the gate array is executed, and the result is then read from qubit 2. The gate arrays could include *H*, U_θ, *CNOT*, and *NAND* gates as described above, along with an *ORACLE* gate implemented analogously to *NAND*, but with a truth table corresponding to the function that the oracle computes. The genetic programming function set consisted of the following gate-array-construction and auxiliary functions:

`H-GATE`: takes 1 argument, which is coerced to a valid qubit index (0–2). An *H* gate is added to the end (output side) of the quantum gate array.

`UN-THETA-GATE`: takes 2 arguments, the first of which is coerced to a valid qubit index (0–2), and the second of which is interpreted as an angle in radians. A U_θ gate is added to the end of the quantum gate array.

`CNOT-GATE`: takes 2 arguments, both of which are coerced to valid qubit indices (0–2). A *CNOT* gate is added to the end of the quantum gate array, unless the two qubit indices are the same (in which case no action is taken).

`NAND-GATE`: takes 3 arguments, all of which are coerced to valid qubit indices (0–2). A *NAND* gate is added to the end of the quantum gate array, unless any of the qubit indices are the same (in which case no action is taken).

`ORACLE`: takes no arguments. A call to the oracle function, using qubits 0 and 1 as inputs and qubit 2 as output, is added to the end of the quantum gate array unless the gate array already contains a call to the oracle (in which case no action is taken).

`ITERATE`: an iteration control structure. Takes 2 arguments, the first of which is coerced to a positive integer and determines the number of iterations that the second argument, a body of code, will be executed. If a (large) bound on the number of iterations is exceeded the calling program immediately halts.

`IQ`: an iteration control structure that takes one argument: a body of code. This is equivalent to a call to `ITERATE` with a first argument equal to the number of qubits in the system (in this case 3).

`IVAR`: takes one argument, which is coerced to a positive integer. `(IVAR 0)` returns the value of the loop counter of the immediately enclosing iteration structure. `(IVAR 1)` returns the value of the loop counter for the next iteration structure out, etc. The argument is taken modulo the number of iteration structures that enclose the call to `IVAR`. Calls to `IVAR` outside of all iteration structures return 0.

+: returns the sum of its two arguments.

1+: returns the sum of its single argument and 1.

—: returns the difference of its two arguments.

1—: returns the difference of its single argument and 1.

*: returns the product of its two arguments.

*2: returns the product of its single argument and 2.

%p: protected division: returns the quotient of its two arguments; returns 1 if its second argument is 0.

%2: returns the quotient of its single argument and 2.

1/x: returns the quotient of 1 and its single argument; returns 1 if its argument is 0.

The genetic programming terminal set consisted of *NUM-QUBITS* (in this case 3), *NUM-INPUT-QUBITS* (in this case 2), *NUM-OUTPUT-QUBITS* (in this case 1), and the constants 0, 1, 2, PI (3.1415...), and i ($\sqrt{-1}$).

We used a standardized fitness function (for which lower values mean "more fit") with 3 components: a *hits* component, a *correctness* component, and an *efficiency* component. The hits component is the total number of fitness cases (8 here) minus the number of cases for which the program's gate array produces the correct answer (a "hit") with an error less than 0.48 (which is far enough from 0.5 to be sure that it is not due to roundoff). This is always a non-negative integer. The correctness component is the error for all cases, calculated as:

$$\frac{\sum_{i=1}^{numCases} \max(0, error_i - 0.48)}{\max(hits, 1)}$$

Note that any errors below 0.48 are ignored. This keeps the focus of the genetic search on the attainment of correct answers rather than on the improvement of errors for cases that are already being answered correctly. Note also that the correctness component gets smaller as the number of hits increases, so reasonably fit programs will be compared primarily with respect to the number of correct answers, and only secondarily with respect to the magnitudes of the errors on the incorrect cases. The efficiency component is simply the number of gates in the final quantum gate array (a non-negative integer) divided by 100,000 so that it will always be far below 1. For programs that do not get all fitness cases correct, only the hits and correctness components are used (they are summed). For programs that do get all fitness cases correct, only the efficiency component is used (and the other components are zero anyway). This causes the search to focus initially on the production of correct gate arrays, and later on the production of more efficient gate arrays. As a result the fitness function approximates lexicographic fitness (Ben-Tal, 1979), with the components ordered: hits (most significant), correctness, efficiency (least significant).

One run of this system, using Koza's non-ADF Lisp genetic programming code (Koza, 1992) and the parameters shown in Table 2, produced the program shown in Figure 6 at generation 46. (The parameters in Table 2 were chosen by intuition and have not been optimized.) When executed, this program produces the gate array shown in Figure 7. Using notation similar to that in the quantum computation literature, this can be represented as in Figure 8.

This is not minimal—at least the final H can be removed, although interestingly the *NAND* cannot (because quantum gates can affect their "inputs" as well as their "outputs"). This gate array solves or provides information useful in solving the two-bit early promise problem for all 8 possible two-bit oracles, using only one call to the oracle in each case. The probabilities of error for the 8 cases are (rounded to two decimal places): 0.02, 0.29, 0.23, 0.13, 0.13, 0.23, 0.30, and 0.04.

Table 2. Genetic programming parameters for a run on the two-bit early promise problem.

max number of generations	1,001
size of population	10,000
max depth of new individuals	6
max depth of new subtrees for mutants	4
max depth after crossover	12
reproduction fraction	0.2
crossover at any point fraction	0.1
crossover at function points fraction	0.5
selection method	tournament (size=5)
generation method	ramped half-and-half
randomizer seed	1.5

```
(IQ(NAND-GATE
      (+ (* (1- 0) (ITERATE PI PI)) (UN-THETA-GATE -1 (*2 *NUM-INPUT-QUBITS*)))
      (%2 (+  (H-GATE (IQ (IQ (1- PI))))
              (ITERATE
                (1- (SQRT (CNOT-GATE (UN-THETA-GATE1(IVAR*NUM-QUBITS*))
                                     (IVAR(ITERATEPI*NUM-OUTPUT-QUBITS*)))))
                (1/X
                  (NAND-GATE  (* (SQRT -1)
                                 (- (%P (IVAR 0) *NUM-QUBITS*) *NUM-INPUT-QUBITS*))
                              (1/X *NUM-INPUT-QUBITS*)
                              PI)))))
      (NAND-GATE (IQ (1- (IQ (%2 (%2 (IQ (*2 *NUM-INPUT-QUBITS*)))))))
                 (IQ (IVAR PI))
                 (SQRT (%2 (1- (ORACLE-GATE)))))))
```

Figure 6. An evolved program that produces a quantum gate array for the two-bit early promise problem.

```
(UN-THETA 2 4)  ;; qubit 2, θ=4 radians
(HADAMARD 0)    ;; qubit 0
(UN-THETA 1 1)  ;; qubit 1, θ=1 radian
(ORACLE)        ;; acts on qubit 2
(NAND 2 1 0)    ;; inputs are 2 & 1, output is 0
(UN-THETA 2 4)  ;; qubit 2, θ=4 radians
(HADAMARD 0)    ;; qubit 0
(UN-THETA 1 2)  ;; qubit 1, θ=2 radians
(CNOT 1 2)      ;; qubit 1 is controller, 2 controlled
```

Figure 7. An quantum gate array for the two-bit early promise problem, produced by the program in Figure 6.

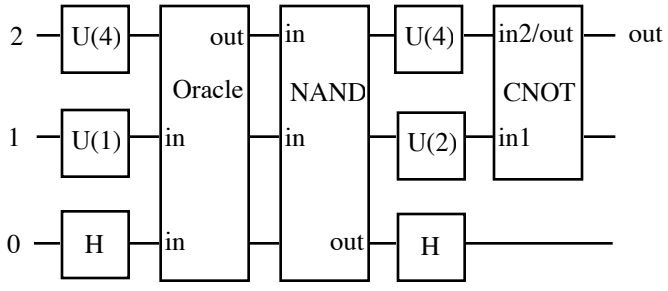

Figure 8. A graphic view of the quantum gate array (see Figure 7) for the two-bit early promise problem.

4. Evolution of Scalable Quantum Gate Arrays

Our framework for evolving quantum gate arrays, described above, can also be used to evolve *scalable* gate arrays. That is, we can use this framework to evolve a single program which, when run with different values for `*NUM-QUBITS*`, `*NUM-INPUT-QUBITS*`, and `*NUM-OUTPUT-QUBITS*`, produces a gate array that solves the given problem at the given size. This is important because the unavailability of quantum computation hardware, coupled with the exponential resources required for classical simulation of quantum computers, limits us to simulating quantum algorithms for small numbers of qubits. But quantum computation is most interesting when applied to much larger problems, for which their exponential savings in resource requirements really pays off. The reason to evolve scalable gate arrays is that we can use small cases for fitness evaluation during evolution. If we manage to evolve gate arrays that do in fact scale correctly, then we will be able to run scaled-up versions of the gate arrays on much larger problem instances on real quantum computer hardware in the future.

Our technique for evolving scalable gate arrays is to evolve programs that construct gate arrays (rather than evolving gate arrays more directly) and to allow the problem size (e.g. `*NUM-QUBITS*`, `*NUM-INPUT-QUBITS*`, and `*NUM-OUTPUT-QUBITS*`) and iterative control structures (e.g. `ITERATE` and `IQ`) to appear in the evolved programs. It may also be useful to allow novel iterative control structures to evolve simultaneously with the programs that use them (Spector, 1996).

As an example, consider an oracle version of the *majority-on* problem. (Genetic programming is applied to the standard non-oracle version of majority-on by Koza, 1992.) This problem is the same as the early promise problem, discussed above, except that all binary oracles are allowed (there is no

promise that the oracles will be either balanced or uniform) and the program's job is to determine if the majority of the oracle's outputs are 1 or not. Using a similar function set and fitness function to those described above we evolved a program that produces the following gate arrays for this problem:

For one-bit oracles:
```
(HADAMARD 0)
(ORACLE)
```

For two-bit oracles:
```
(HADAMARD 1)
(HADAMARD 0)
(ORACLE)
```

For three-bit oracles:
```
(HADAMARD 1)
(HADAMARD 2)
(HADAMARD 0)
(ORACLE)
```

For four-bit oracles:
```
(HADAMARD 1)
(HADAMARD 2)
(HADAMARD 3)
(HADAMARD 0)
(ORACLE)
```

Etc.

This works by spreading the probability out among all basis vectors and then using a single oracle call, which can be thought of as operating on the superposition of all oracle inputs simultaneously, to compute the output. This works quite well for oracles that produce mostly 1s or mostly 0s, but for exactly balanced oracles (for which the answer should be 0—a majority is not on) the output error will be 0.5. This means that there will be a 50% chance of getting the wrong answer for balanced oracles, but this can be remedied by running the program multiple times; if the answer is 1 50% of the time then we know that the oracle is balanced and that the real answer is therefore 0.

It should be noted that in contrast to the early promise algorithm exhibited above, this majority-on quantum algorithm is not better than classical. A probabilistic classical algorithm for majority-on can simply call the oracle with a random input; if the output is 1 then it should answer 1, otherwise it should answer 0. This too will have a 50% chance of being wrong for balanced oracles (and some smaller chance of being wrong for other oracles), and this too can be remedied with multiple runs. In this case the genetic programming system found a quantum algorithm that works in the same way as a probabilistic classical algorithm, although the early promise algorithm above shows that it can in some cases do better.

The strategy of running a Hadamard gate on all qubits at the beginning of the computation, found here by the genetic programming process, is useful in general. It puts the system into an in-phase equal superposition of all inputs, and thereby allows for computation on all inputs simultaneously (with interference). In some of our subsequent experiments we have found it useful to automatically run a Hadamard gate on each qubit whenever the simulated quantum computer is initialized; that is, we evolve a gate array that takes an equal superposition as input.

5. Future Work

The work described in this paper points to several directions for future research, several of which we are actively pursuing.

5.1. Search for Better-than-Classical Quantum Algorithms for Other Problems

We are currently applying the techniques described in this paper to several other problems, including problems for which better-than-classical quantum algorithms are already known (for example, Grover's database search algorithm) and classically NP complete problems (including Hamiltonian Circuit and Clique).

5.2. Exploration of the Quantum Complexity of Boolean Functions

Recent results (Beals et al., 1998) establish tight bounds on the quantum complexity of certain classes of boolean functions. But there are still several open questions about the speedup that quantum computers may be able to achieve for boolean functions (e.g. for asymmetric boolean functions), and we are currently using genetic programming to look for quantum algorithms that may shed light on these questions.

5.3. Evolution of Deterministic Quantum Gate Arrays

In the work described here we ignored errors of less than 0.48, meaning that we were satisfied when the correct answer was the most probable one, even if there was still significant indeterminacy. While this can be remedied via repeated runs, and while one can often nonetheless prove that the quantum algorithm is better than any possible classical algorithm (as with the early promise problem), it would be more desirable to produce deterministic algorithms that always produce the correct answers. It is straightforward to adapt our methods to this goal, either by counting all error all of the time or by lowering the acceptable error threshold as the genetic programming runs progress.

5.4. Evolution of Hybrid Quantum/Classical Algorithms

Several known quantum algorithms use quantum computation for a central calculation but rely on classical computation either to pre-process the quantum computer's inputs or to post-process the quantum computer's outputs (or both). It is straightforward to adapt our methods to evolve hybrid quantum/classical algorithms. For example, one can use a genetic programming engine that supports automatically defined functions (Koza, 1994) to evolve a program with three branches: a classical pre-processing branch, a quantum gate array producing branch, and a classical post-processing branch.

5.5. Evolution of Useful Gates from Physically Simple Gates

Depending on the physical means used to implement a quantum computer, certain kinds of quantum gates may be more or less easy to build. Because the physically simple (or even feasible) gates may not be most convenient gates for human theoreticians/programmers or for automatic programming processes, we may wish to evolve gate arrays that implement the theoretically convenient gates in terms of the physically simple ones.

5.6. Genetic Programming on Quantum Computers

It has not escaped our attention that a slight modification of this paper's title, "Genetic Programming *on* Quantum Computers" may also hold significant promise. Grover's work demonstrates the existence of better-than-classical search algorithms, and it is possible that this result could be extended to the search of the space of computer programs. It is also possible that quantum parallelism and the intrinsic probabilistic processing provided by quantum computers may provide better-than-classical speedups for fitness evaluation. For genetic algorithms with binary chromosomes, it may be particularly easy to speed up recombination or to determine the fitness of entire *schemata* (in the sense of the Schema Theorem of Holland, 1992) using quantum parallelism.

6. Conclusion

We have used genetic programming to automatically discover a quantum algorithm that is more efficient than any possible classical algorithm for Deutsch's early promise problem. We also showed how our technique can be used to evolve scalable quantum gate arrays, and we exhibited a scalable quantum gate array for the oracle version of the majority-on problem. We sketched extensions to our technique, including methods by which deterministic quantum gate arrays and hybrid quantum/classical algorithms may be evolved.

Genetic programming appears to be a useful tool for exploring the power of quantum computation, and perhaps for developing software for the quantum computers of the future. One significant problem is the simulation speed for quantum gates—this may be improved but the problem will not go away because of the very powers of quantum computation that we wish to harness. This means that research should be focused on problems for which small instances are nonetheless significant, on the evolution of scalable quantum gate arrays, and on genetic programming techniques that reduce the required number of fitness evaluations.

Genetic programming *on* quantum computers, using better-than-classical search algorithms that are already in the literature, is also likely to be a fruitful area for future research.

Acknowledgments

Supported in part by the John D. and Catherine T. MacArthur Foundation's MacArthur Chair program at Hampshire College, by National Science Foundation grant #PHY-9722614, and by a grant from the Institute for Scientific Interchange (ISI), Turin. Thanks also to Jim Hendler for CPU time on University of Maryland computers.

Bibliography

Barenco, Adriano, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John Smolin, and Harald Weinfurter. 1995. Elementary gates for quantum computation. Submitted to *Physical Review A*, March 22, 1995. http://xxx.lanl.gov/abs/quant-ph/9503016.

Beals, R., H. Buhrman, R. Cleve, M. Mosca, and R. de Wolf. 1998. Tight Quantum Bounds by Polynomials (Preliminary Version). http://xxx.lanl.gov/abs/quant-ph/9802049.

Beckman, David, Amalavoyal N. Chari, Srikrishna Devabhaktuni, and John Preskill. 1996. Efficient networks for quantum factoring. California Institute of Technology Report CALT-68-2021, http://xxx.lanl.gov/abs/quant-ph/9602016.

Ben-Tal, A. 1979. Characterization of Pareto and Lexicographic Optimal Solutions. In *Multiple Criteria Decision Making Theory and Application*, Springer-Verlag Lecture Notes in Economics and Mathematical Systems, No. 177, Fandel and Gal, Editors. pp. 1–11.

Braunstein, Samuel L. 1995. Quantum computation: a tutorial. http://chemphys.weizmann.ac.il/~schmuel/comp/comp.html

Costantini, Giovanni, and Fabrizio Smeraldi. 1997. A Generalization of Deutsch's Example. http://xxx.lanl.gov/abs/quant-ph/9702020.

Deutsch, D. 1985. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London A* **400**, 97–117.

Grover, L.K. 1997. Quantum mechanics helps in searching for a needle in a haystack. *Physical Review Letters* **79** pp. 325–328.

Holland, J.H. 1992. *Adaptation in Natural and Artificial Systems.* Cambridge, MA: The MIT Press.

Jozsa, Richard. 1997. Entanglement and Quantum Computation. In *Geometric Issues in the Foundations of Science,* edited by S. Huggett, L. Mason, K.P. Tod, S.T. Tsou, and N.M.J. Woodhouse. Oxford University Press. http://xxx.lanl.gov/abs/quant-ph/9707034.

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press.

Koza, John R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: The MIT Press.

Milburn, Gerard J. 1997. *Schrödinger's Machines: The Quantum Technology Reshaping Everyday Life*. New York: W.H. Freeman and Company.

Penrose, Roger. 1994. *Shadows of the Mind*. Oxford University Press.

Preskill, John. 1997. Quantum Computing: Pro and Con. California Institute of Technology Report CALT-68-2113, http://xxx.lanl.gov/abs/quant-ph/9705032.

Shor, P.W. 1994. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science,* edited by S. Goldwasser. IEEE Computer Society Press. Los Alamitos, CA. p. 124.

Spector, Lee. 1996. Simultaneous Evolution of Programs and their Control Structures. In *Advances in Genetic Programming 2*, edited by P. Angeline and K. Kinnear. Cambridge, MA: MIT Press.

Steane, Andrew. 1997. Quantum computing. *Reports on Progress in Physics*. http://xxx.lanl.gov/abs/quant-ph/9708022.