PRE-PRESS version of: Spector, L., H. Barnum, and H.J. Bernstein. 1999. Quantum Computing Applications of Genetic Programming. In *Advances in Genetic Programming, Volume 3*, edited by L. Spector, U.-M. O'Reilly, W. Langdon, and P. Angeline, pp. 135-160. Cambridge, MA: MIT Press.

7 Quantum Computing Applications of Genetic Programming

Lee Spector, Howard Barnum, Herbert J. Bernstein, and Nikhil Swamy

Quantum computers are computational devices that use the dynamics of atomic-scale objects to store and manipulate information. Only a few, small-scale quantum computers have been built to date, but quantum computers can in principle outperform all possible classical computers in significant ways. Quantum computation is therefore a subject of considerable theoretical interest that may also have practical applications in the future.

Genetic programming can automatically discover new algorithms for quantum computers [Spector et al., 1998]. We describe how to simulate a quantum computer so that the fitness of a quantum algorithm can be determined on classical hardware. We then describe ways in which three different genetic programming approaches can drive the simulator to evolve new quantum algorithms. The approaches are standard tree-based genetic programming, stack-based linear genome genetic programming, and stackless linear genome genetic programming. We demonstrate the techniques on four different problems: the *two-bit early promise* problem, the *scaling majority-on* problem, the *four-item database search* problem, and the *two-bit and-or* problem. For three of these problems (all but *majority-on*) the automatically discovered algorithms are more efficient than any possible classical algorithms for the same problems. One of the better-than-classical algorithms (for the *two-bit and-or* problem) is in fact more efficient than any previously known quantum algorithm for the same problem, suggesting that genetic programming may be a valuable tool in the future study of quantum programming.

7.1 Quantum Computation

Quantum computers use the dynamics of atomic-scale objects, for example 2-state particles, to store and manipulate information ([Steane, 1998]; see [Braunstein, 1995] for an on-line tutorial; see [Milburn, 1997] for an introduction for the general reader). Devices at this scale are governed by the laws of quantum mechanics rather than by classical physics, and this makes it possible for a quantum computer to do things that a common digital ("classical") computer cannot. In particular, quantum computers can solve certain problems using less time and space resources than classical computers require [Jozsa, 1997]. The physical basis of a real quantum computer might take various forms. Current experimental hardware is based on the use of ion traps, cavity quantum electrodynamics, or nuclear magnetic resonance techniques, all of which appear to have weaknesses [Preskill, 1997], although some physicists are optimistic that new developments will allow for the construction of large-scale quantum computers.

Richard Feynman hinted at the possible power of quantum computation at least as early as 1981 [Milburn, 1997, page 164], but the idea didn't attract widespread attention until a few dramatic examples were discovered more than a decade later. Perhaps the most dramatic was Peter Shor's quantum factoring algorithm, which finds the prime factors of an *n*-digit number in time $O(n^2 \log(n) \log\log(n))$ [Shor, 1998]. The best currently known

classical factoring algorithms require at least time $O(2^{n^{\frac{1}{3}}\log(n)^{\frac{2}{3}}})$, so Shor's algorithm appears to provide a near-exponential speedup [Shor, 1994; Beckman et al., 1996]. This is not certain, however, because a classical lower bound for factoring has not yet been proven. Another intriguing result was provided by Lov Grover, who showed how a quantum computer can find an item in an unsorted list of *n* items in $O(\sqrt{n})$ steps; classical algorithms clearly require $O(n)$, so this is a case in which quantum computation clearly beats classical computation on a commonly occurring problem [Grover, 1997]. It is not yet clear exactly how powerful quantum computers are relative to classical computers, but this is a subject of active investigation by several research groups.

In the following section we describe how to build a virtual quantum computer to simulate the operation of a quantum computer on ordinary classical computer hardware. We then show how the virtual quantum computer can be used, in conjunction with genetic programming techniques, to evolve new quantum algorithms. This is followed by a presentation of results for four different problems and some concluding remarks.

7.2 A Virtual Quantum Computer

The smallest unit of information in a quantum computer is called a *qubit*, by analogy with the classical *bit*. A classical system of n bits is at any time in one of 2^n states. Quantum mechanics tells us, however, that we must think of a quantum system of n qubits as having a distinct probability of "being in" (that is, "being found in upon measurement") each of the 2^n classical states at any given time. Of course the probabilities must sum to 1—we will always find the system in some particular state when we measure it. The system is said to be in a "superposition" of all states for which there is non-zero probability.

A quantum mechanical system of n qubits can be modeled as a vector of 2^n complex numbers, one *probability amplitude* for each of the 2^n classical states. The probability of finding the system in a particular state is calculated as the square of the modulus of the corresponding amplitude. Computations in the system are modeled as linear transformations, often represented as matrices, applied to the vector of probability amplitudes. Some of these transformations simply move probability from one state to another, in a manner analogous to classical logic gates, but others "spread" or recombine probability between multiple states in more interesting ways (see below). Readers familiar with wave mechanics will recognize these phenomena as instances of quantum interference.

In the following subsections we present some useful notation and then describe the operation of the virtual quantum computer. We also trace the execution of an example quantum algorithm and make some brief observations about the power of quantum computation in light of the simulation.

7.2.1 State Representation and Notation

We represent the state of an n-qubit system as a unit vector of 2^n complex numbers $[\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{2^n-1}]$. Each of these numbers can be viewed as paired with one of the system's classical states. The classical states are called the "computational basis vectors" of the system and are labled by n-bit strings, represented as $|b_{n-1}b_{n-2}\ldots b_j\ldots b_0\rangle$ where each b_j is either 0 or 1.[1] The state labels can be abbreviated using the binary number formed by concatenating the bits; that is, we can write $|k\rangle$ in place of $|b_{n-1}b_{n-2}\ldots b_j\ldots b_0\rangle$ where $k = b_0 + 2b_1 + 4b_2 + \ldots + 2^{n-1}b_{n-1}$. For example we can write $|6\rangle$ in place of $|110\rangle$. The modulus squared of each amplitude α (for example $|\alpha_k|^2$) is the probability that measurement of the system will find it in the corresponding classical state ($|k\rangle$).

As an example, the complete state of a two-qubit system is represented in the following form:

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

If we measure the system's state, each of the computational basis vectors is a possible outcome. The probability that the state of the system is $|00\rangle$ is $|\alpha_0|^2$, the probability that we will find the state of the system to be $|01\rangle$ is $|\alpha_1|^2$, etc.

7.2.2 Quantum Gates

The primitive operations supported by a quantum computer are called *quantum logic gates*, by analogy with traditional digital logic gates. Several small sets of quantum logic gates are *universal* for quantum computation in almost the same sense that *NAND* is universal for classical computation; one can implement any quantum algorithm with at most polynomial slowdown using only primitive gates from one of these sets ([Barenco et al., 1995] and references therein). For example, all quantum computations can be implemented using only the U_2 and *CNOT* gates described below.

We will describe and represent quantum gates as matrices that operate on a quantum system via matrix multiplication with the vector of amplitudes. Gates representing physically possible dynamics (time-evolution) of a closed (or isolated) quantum system must be *unitary*—that is, each gate U must satisfy $U^\dagger U = UU^\dagger = 1$, where $U^\dagger$ is the Hermitean adjoint of U, obtained by taking the complex conjugate of each element of U and then transposing the matrix [Löwdin, 1998].

7.2.2.1 Quantum *NOT* and *SQUARE ROOT OF NOT*

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

[1] The "$|\ldots\rangle$" notation is for "ket" vectors; this notation is standard in the quantum computation literature and it will be used here even though the "bra-ket" notation system of which it is part is beyond the scope of this chapter (but see [Chester, 1987]).

on a single qubit. In a one-qubit system (which we represent with two amplitudes, one for $|0\rangle$ and one for $|1\rangle$) the quantum *NOT* operation simply swaps the values of the two amplitudes. That is, a single qubit system in the state $\alpha_0|0\rangle + \alpha_1|1\rangle$ will be transformed by quantum *NOT* into $\alpha_1|0\rangle + \alpha_0|1\rangle$. Quantum *NOT* can be represented in matrix form as $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$, and its operation on a one qubit system $\alpha_0|0\rangle + \alpha_1|1\rangle$, represented as a column vector $\left[\begin{smallmatrix} \alpha_0 \\ \alpha_1 \end{smallmatrix}\right]$, can be shown as:

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

Another interesting one-qubit gate is the *SQUARE ROOT OF NOT* (*SRN*) gate:

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

A single application of *SRN* will in effect randomize the state of a qubit that was previously in a "pure" state of 0 or 1. That is, it will transform a situation in which there is a probability of 1 for reading the state as "0" (or a situation in which there is a probability of 1 for reading the state as "1") into one in which there is a probability of $\frac{1}{2}$ for reading the state as "0" and a probability of $\frac{1}{2}$ for reading the state as "1". But applying this gate twice in succession will produce the same inverting effect as *NOT*, thereby extracting information from the seemingly randomized intermediate state.[2]

7.2.2.2 Applying quantum gates to multi-qubit systems

When applied to qubit j of a multi-qubit system, quantum *NOT* swaps the amplitudes of each pair of basis vectors that differ from one another only in the jth position. For example, in a two-qubit system the application of quantum *NOT* to qubit 0 will swap the amplitude of $|00\rangle$ with that of $|01\rangle$, and the amplitude of $|10\rangle$ with that of $|11\rangle$. This can be represented in matrix form as follows:

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

One typically describes only the minimal version of a gate, for example the 2×2 matrix for *NOT*, and expands it as needed for application to a larger system. For an n-qubit system the expansion will always be a $2^n \times 2^n$ matrix of complex numbers which, when multiplied

[2] The double application of *SRN* is not quite equivalent to *NOT* because there is a change in sign: $\left[\begin{smallmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{smallmatrix}\right] \left[\begin{smallmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{smallmatrix}\right] = \left[\begin{smallmatrix} 0 & -1 \\ 1 & 0 \end{smallmatrix}\right]$. But this change in sign has no immediate effect because we square the amplitudes when reading the state of the system.

by the vector of amplitudes, has the effect of applying the gate to the specified qubit or set of qubits.

To understand how quantum gates are applied in multi-qubit systems one must bear in mind that all amplitudes in the state representation encode part of the value for each qubit. For example, in a two-qubit system the amplitudes for $|00\rangle$ and $|10\rangle$ both contribute to the probability that the right-most qubit (qubit 0) is zero, and the amplitudes for $|01\rangle$ and $|11\rangle$ both contribute to the probability that qubit 0 is one. So a gate applied to a small subset of the qubits of a multi-qubit system may nonetheless change all of the amplitudes in the state representation.

To apply an m-qubit gate to a set Q of m qubits in an n-qubit system ($m \leq n$), one must in general operate on *all* 2^n amplitudes in the system. The $2^n \times 2^n$ matrix that one uses should have the effect of applying the $2^m \times 2^m$ minimal version of the gate to each of 2^{n-m} different column vectors. Each of these column vectors corresponds to a set of basis vectors that varies only with respect to Q and is constant in all other bit positions. For example, consider the 4×4 *NOT* matrix above, which is a *NOT* gate for qubit 0 in a two-qubit system. This 4×4 matrix has the effect of applying the 2×2 *NOT* matrix ($\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$) to the amplitudes for $|00\rangle$ and $|01\rangle$, that is to $\left[\begin{smallmatrix} \alpha_0 \\ \alpha_1 \end{smallmatrix}\right]$, to produce new amplitudes for $|00\rangle$ and $|01\rangle$, and also of applying the 2×2 *NOT* matrix to the amplitudes for $|10\rangle$ and $|11\rangle$, that is to $\left[\begin{smallmatrix} \alpha_2 \\ \alpha_3 \end{smallmatrix}\right]$, to produce new amplitudes for $|10\rangle$ and $|11\rangle$. This can be generalized for any m; one wants the $2^n \times 2^n$ matrix which, for each set of 2^m basis vectors that vary only with respect to Q, multiplies the $2^m \times 2^m$ minimal version of the gate by the corresponding set of amplitudes.

An implementation option is to build up the $2^n \times 2^n$ matrix that has the required effect explicitly, and to multiply this matrix by the vector of amplitudes. Alternatively one can operate on the amplitudes one at a time, dynamically computing for each one the necessary matrix elements. Because the full expanded matrices are large and mostly zero we generally take the latter approach. Note that in any case one must perform an exponentially large amount of work (with respect to n) in order to apply a single gate; this is the source of the exponential slowdown in the simulation of quantum computations.

7.2.2.3 Other Quantum Gates

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

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

CNOT flips the state with respect to its target qubit wherever its control qubit is 1. By making the condition on this flipping more complex, using more controlling qubits, we can construct analogous gates for any classical boolean function. For example, consider the classical *NAND* gate which takes two input bits and outputs 0 if both inputs are 1, and 1 otherwise. That is, it has the following truth table:

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

Such a truth table can be used to form a quantum gate by interpreting a 1 in the output (rightmost) column of a particular row as an instruction to swap amplitudes 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 quantum gate, called quantum *NAND*, that takes three qubit indices (these can be thought of as 2 inputs and 1 output[3]) and swaps amplitudes 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 1 and 2 as inputs and qubit 0 as output, this can be represented in matrix form as follows[4]:

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

[3]The designation of quantum gate arguments as "inputs" and "outputs" is convenient but may in some cases be misleading. When applied to certain states quantum *NAND* (and other gates described below) can affect "input" as well as "output" qubits. We retain the "input/output" terminology because it allows for more intuitive explanations.

[4]A reviewer suggested that would normally be called "NOT Controlled-Controlled-NOT" in the quantum computation literature.

The work described in this chapter also uses a *Hadamard* gate which can be used to split the amplitude between opposite values for a single qubit:

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

and simple rotation by an angle θ:

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

and a conditional phase gate that takes a single real parameter α:

$$CPHASE = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & e^{i\alpha} \\ 0 & 0 & e^{-i\alpha} & 0 \end{bmatrix}$$

and generalized rotation with four real parameters α, θ, ϕ, and ψ:

$$U_2 = \begin{bmatrix} e^{-i\phi} & 0 \\ 0 & e^{i\phi} \end{bmatrix} \times \begin{bmatrix} \cos(\theta) & \sin(-\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} e^{-i\psi} & 0 \\ 0 & e^{i\psi} \end{bmatrix} \times \begin{bmatrix} e^{i\alpha} & 0 \\ 0 & e^{i\alpha} \end{bmatrix}$$

The U_2 gate can emulate any other single bit gate, at the cost of taking four real-valued parameters. A generalized 2-qubit gate with sixteen real-valued parameters (U_4) also exists, but we did not use this in the work described here.

7.2.3 Running a Quantum Algorithm

A quantum algorithm is run by putting the system into a known initial state, subjecting it to a sequence of gates, and then reading out (i.e., measuring) the final state of the system. The initial state is usually a computational basis vector—that is, a state in which a single amplitude is 1 and all others are 0; in the work reported here the system is always started in state $|00\ldots0\rangle$, meaning that the amplitude for $|00\ldots0\rangle$ is initially 1. The final measurement is usually made in the computational basis, and each gate usually involves no more than a few qubits. These conditions ensure that the number of gates in a quantum circuit is a reasonable measure of its computational complexity. The final state is read by squaring the modulus of each amplitude, summing those that correspond to the same values for the output bits, and reporting the output bit pattern with the highest sum. This is the output that would most likely be produced if the same sequence of operators was run on a real quantum computer. The simulation can also report the actual probability of obtaining this most-likely result; this is just the sum of $|\alpha|^2$ for the states having the most probable output pattern. If the probability for returning the correct answer is less than 1 but greater

than $\frac{1}{2}$, the algorithm may nonetheless be useful. This is because one can often show that re-running the algorithm some small number of times will reduce the indeterminacy to any required level. A quantum algorithm is said to compute a function with *2-sided-error* if it always returns an answer which is correct with probability at least p, where $\frac{1}{2} < p < 1$ [Beals et al., 1998]. For algorithms intended to scale to systems of any number of qubits n, p must not depend on n, or at least not decrease too rapidly with n.

7.2.4 Example Execution Trace

To clarify the way in which quantum algorithms are executed we will trace the execution of a simple, arbitrary algorithm in some detail. Consider the following quantum algorithm for a two qubit system:

```
Hadamard qubit:0
Hadamard qubit:1
U-theta qubit:0 theta:pi/5
Controlled-not control:1 target:0
Hadamard qubit:1
```

Execution starts in the state $1|00\rangle + 0|01\rangle + 0|10\rangle + 0|11\rangle$. The *Hadamard* gate on qubit 0 is then applied by means of two matrix multiplications:

- The *Hadamard* matrix is multiplied by a column vector made from the amplitudes for $|00\rangle$ and $|01\rangle$, and the new values for the amplitudes of $|00\rangle$ and $|01\rangle$ are taken from the resulting column vector.

- The *Hadamard* matrix is multiplied by a column vector made from the amplitudes for $|10\rangle$ and $|11\rangle$, and the new values for the amplitudes of $|10\rangle$ and $|11\rangle$ are taken from the resulting column vector.

This transforms the state to $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle + 0|10\rangle + 0|11\rangle$. At this point there is an equal probability of finding the state $|00\rangle$ or the state $|01\rangle$, but the other states have zero probability. This means that one could find qubit 0 to be 0 or 1 (each with equal probability), but one would definitely find qubit 1 to be 0. The subsequent *Hadamard* gate on qubit 1 transforms the state to $\frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle$. At this point all states have the same probability. The U_θ gate rotates qubit 0 by $\frac{\pi}{5}$, using the same procedure as described for the *Hadamard* gates but with a different matrix, producing approximately $0.698|00\rangle + 0.111|01\rangle + 0.698|10\rangle + 0.111|11\rangle$. The subsequent *CNOT* flips qubit 0 for basis vectors in which qubit 1 is "1", producing $0.698|00\rangle + 0.111|01\rangle + 0.111|10\rangle + 0.698|11\rangle$. The final *Hadamard* again manipulates qubit 1, producing a final state of approximately $0.572|00\rangle + 0.572|01\rangle + 0.416|10\rangle - 0.416|11\rangle$. The probabilities of finding the system in each of the possible classical states upon measurement are approximately as follows:

state	probability
$\|00\rangle$	0.33
$\|01\rangle$	0.33
$\|10\rangle$	0.17
$\|11\rangle$	0.17

We can measure the system and read the output from either or both of the qubits. If we read only qubit 1 there is a probability of $0.33 + 0.33 = 0.66$ that we will find it to be "0" and a probability of $0.17 + 0.17 = 0.34$ that we will find it to be "1". Qubit 0's value will be completely random, with a probability of 0.5 for each state. So the example algorithm takes the state $|00\rangle$ to a state with a random value for qubit 0 and a biased value for qubit 1, with probability 0.66 to be "0".

7.2.5 The Power of Quantum Computation

Having examined the mechanics of a virtual quantum computer and traced its execution we may now be in a better position to see the source of the power of quantum computation. The vector of 2^n complex amplitudes clearly contains more information than a classical state of n bits, but it is maintained with only n quantum mechanical bit registers (e.g., n spin-$\frac{1}{2}$ particles). We cannot read the entire state because the measured result is always a single computational basis vector and there will generally be some uncertainty about which computational basis vector we will actually read. Further, measurement interferes with the system so it can only be read once. But the additional information in the state can nonetheless sometimes be extracted and harnessed to perform real computational work. In some cases the extra information can be used to perform computations on several different somewhat probable states simultaneously, and clever manipulations allow us to extract useful information from all of them. Of course we are paying for this in the simulator with exponential resources, so we can only work with relatively small systems. Fortunately, small systems are adequate for the evolution of some new algorithms, including some algorithms that can be scaled up to work on larger problem instances when real quantum computing hardware becomes available.

7.3 Evolving Quantum Algorithms

Given a simulator for a quantum computer, one can use genetic programming techniques to evolve quantum algorithms [Spector et al., 1998]. Genetic programming systems evolve programs (algorithms), and one can use a genetic programming system to evolve quantum algorithms by ensuring that the programs in the population have the proper form and by assessing their fitness on the simulated quantum computer.

Many open questions in quantum computation concern the computational resources required to scale algorithms up to larger problem instances. For this reason it would be most useful if we could evolve *scalable* quantum algorithms that work on problem instances of any size. Scaling is also important because classical simulation of quantum computers consumes an amount of resources that grows exponentially with respect to the number of qubits in the system, and this limits us to simulating quantum algorithms for small systems. But quantum computation is most interesting when applied to much larger problems, for which their exponential savings in resource requirements really pays off. We can ameliorate the problem by using small cases of several sizes for fitness evaluation during evolution. In some cases this will produce algorithms that scale correctly to all sizes; scaled-up versions of these algorithms could be run on much larger problem instances on real quantum computer hardware in the future.

The scaling results reported in this chapter are modest, but the technique that we use was designed to allow for the evolution of scalable algorithms and we show an example of this in section 7.4.2 below. One should also note that new quantum algorithms may be of significant interest even if they do not scale, although it is obviously preferable to find scaling algorithms.

Our technique for finding scalable algorithms involves evolving classical programs that, when executed, construct the actual quantum algorithms. Because the classical programs can include iteration structures and constants related to the size of a particular problem instance, a single evolved program can produce different quantum algorithms for problem instances of different sizes. This technique is related to a theoretical construction used by Peter Shor to define quantum complexity classes [Shor, 1998, pages 473–474]. It is also similar to "second-order encoding" techniques, in which evolved programs must be run to produce the sought-after executable structures, that have previously been used to evolve neural networks and electrical circuits [Gruau, 1994; Koza and Bennett, 1999]. The use of such second-order encodings to provide scaling appears to be novel with this work.

7.3.1 Standard Tree-based Genetic Programming

To evolve quantum algorithms using a standard (weakly typed) tree-based genetic programming engine [Koza, 1992] we start with a set of functions that add gates to an initially empty quantum algorithm. These functions are parameterized by numbers, so the *closure* type for the function set is `number` (which includes integers, ratios, floating point numbers, and complex numbers). The algorithm-building functions include the following:

`H-GATE` Takes 1 argument, which is coerced to a valid qubit index by taking the truncated real part of the argument modulo the number of qubits in the system. (All coercions specified below are performed in a similar way.) A *Hadamard* gate on the given qubit is added to the end (output side) of the quantum algorithm. The function call returns the argument (un-coerced).

U-THETA-GATE Takes 2 arguments, the first of which is coerced to a valid qubit index, and the second of which is interpreted as an angle in radians. A rotation (U_θ) gate is added to the end of the quantum algorithm. The function call returns the first argument.

CNOT-GATE Takes 2 arguments, both of which are coerced to valid qubit indices. A *CNOT* gate is added to the end of the quantum algorithm, using the first argument as the control qubit and the second argument as the target qubit, unless the two qubit indices are the same (in which case no action is taken). The function call returns the first argument.

NAND-GATE Takes 3 arguments, all of which are coerced to valid qubit indices. A *NAND* gate is added to the end of the quantum algorithm, using the first two arguments as inputs and the third argument as output, unless any of the qubit indices are the same (in which case no action is taken). The function call returns the third argument.

Similar functions may be added for the other quantum gates. We also include iteration control structures that help to evolve scalable quantum algorithms:

ITERATE An iteration control structure. Takes 2 arguments, the first of which is coerced to a non-negative integer, and determines the number of iterations that the second argument, a body of code, will be executed. If a (typically very large) bound on the number of iterations is exceeded the calling program immediately halts. The number of iterations is returned as the value of the control structure expression.

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.

IVAR Takes one argument, which is coerced to a non-negative 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 reduced modulo the number of iteration structures that enclose the call to `IVAR`. Calls to `IVAR` outside of all iteration structures return 0.

We also include a collection of arithmetic functions: `+` (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).

In the genetic programming terminal set we include `*NUM-QUBITS*` (a constant equal to the number of qubits in the system), `*NUM-INPUT-QUBITS*` (a constant equal to the number of qubits used for input), `*NUM-OUTPUT-QUBITS*` (a constant equal to the number of qubits used for output), and a variety of useful constants, sometimes including 0, 1, 2, π (3.1415...), and i ($\sqrt{-1}$). In some runs we also include an ephemeral random constant specifier [Koza, 1992] that can produce random floating point constants (usually in the range [-10.0 to $+10.0$], although we have experimented with various ranges).

7.3.2 Stack-Based, Linear Genome Genetic Programming

Although standard tree-based genetic programming (*TGP*) can be used to evolve quantum algorithms, other approaches may have certain advantages. The tree structure of *TGP* representations plays several roles; for example it provides an elegant mechanism for adaptive determination of program size and shape [Langdon et al., 1999] and it also allows for natural expression of *functional* programming constructs, in which sub-expressions return values that are used for various purposes in the larger expressions within which they are nested. But the tree representations come at a cost (time, space, complexity), and there is no guarantee that they will be the most appropriate representations for all problems.

Notice that the algorithm-building functions described in the previous section all work by "side effect"; that is, they do their useful work by making changes to the quantum algorithm that is being constructed, and they return uninformative values (copies of their arguments). This suggests that the function set is ill-suited to the functional programming paradigm, and that the tree structure of *TGP* representations will therefore have diminished utility in this domain. It is possible that the tree structure is actually a liability in cases such as this, since a sub-expression's contribution to its enclosing expression (its return value) is related to its function (side effect) in arbitrary ways. One could argue that it would be difficult for evolution to untangle return values and side effects, for example to preserve an important return value while modifying a side effect of the same sub-expression, and that this would put unnecessary burdens on the genetic programming system.

While the reliance on side effects is due in part to our specific design choices, *any* representation of quantum algorithms is likely to have similar features. This is because *measurement* (accessing values of variables) in a quantum system changes the system, which will usually destroy prepared superpositions and ruin the computation. In more concrete terms, we cannot access the amplitudes in our state representation and use the values to influence the choice of future computational steps, because any such access on a real quantum computer would change the system's state. So steps in a quantum algorithm must always be blind, to a certain extent, to the values (amplitudes) produced by earlier steps.

In a stack-based, linear genome genetic programming (*SBGP*) system, programs are represented not as trees but rather as linear sequences of functions that can communicate via a global stack [Perkis, 1994; Stoffel and Spector, 1996]. This eliminates the conflation of return value and side effect, since functions with no meaningful return values can simply

be coded not to "return" values onto the stack. It is well suited to the evolution of sequential, side-effect-based programs because the program structure is itself sequential and less biased toward functional (return-value-based) programming style. For example, in SBGP programs a side-effect-producing function can be replaced with another without the danger that a different (possibly arbitrary) return value will change the behavior of an enclosing expression.

SBGP systems offer several other advantages. Their linear program structure simplifies the expression of genetic operators (one can use operators from traditional string-based genetic algorithms), reduces memory requirements (since there are no growing trees—one can use fixed-length programs with non-functional `noop` operators to allow for shorter programs), and allows for very high performance genetic programming engines [Stoffel and Spector, 1996]. In addition, anecdotal reports suggest that SBGP may require less computational effort than TGP for many problems [Perkis, 1994; Stoffel and Spector, 1996].

We have found SBGP to be preferable to TGB for our quantum algorithm problems, though we have not conducted a careful comparison of the techniques. With SBGP we have low memory requirements per program (allowing for larger populations), we are freed from concerns about tree growth dynamics and return-value/side-effect interactions, and results appear to emerge more quickly than in our prior TGP work. Further research may provide a more scientific comparison of TGB and SBGP for evolution of quantum algorithms.

We used the *MidGP* SBGP system [Spector, 1997], a simple, flexible Lisp-based system derived from HiGP [Stoffel and Spector, 1996]. The translation of quantum algorithm-building and arithmetic functions from TGP to *MidGP* is straightforward, but the translation of the iteration structures can be done in various ways. We have experimented with both structured and unstructured (GOTO-based) iteration mechanisms, and in principle one could use any control structures from other stack-based languages such as FORTH and Postscript. In SBGP one generally also includes a collection of stack-related functions, for example to duplicate (`dup`) or remove (`pop`) the top stack element, to swap the top two elements, etc. Because of the ease with which they can be written for *MidGP*, we have also used several new genetic operators in our *MidGP*-based work on quantum algorithms. For example, we have used program rotation operators, crossover operators that concatenate randomly selected chunks from parent programs, and a mutation operator that adds small random floating-point values to numeric constants.

7.3.3 Stackless Linear Genome Genetic Programming

Although the scaling of quantum algorithms is in many cases important, there are other cases for which it suffices to find a single quantum algorithm that works for a single problem size. In these cases there may (depending on the structure of the problem) be no need for iteration in the quantum algorithm-building program, since the primary role of the iteration structures in the function set is to allow for scaling of the quantum algorithms. If, in

addition, there is no compelling reason for the gates to be able to share parameter values, then there may be no need for any sort of storage (return values or the stack) at all.

For such cases we have found it useful to use a very simple technique, in which a quantum algorithm is represented as a linear sequence that includes *only* `noop` functions and *encapsulated gates*. An encapsulated gate is a package that includes, internally, the type of quantum gate and values for all required parameters. The quantum algorithm is run by executing each of the gates in sequence, and no global value stack is required.

We have implemented this approach within *MidGP*, generating all gates and parameters randomly via *MidGP*'s ephemeral random constant mechanism. We use a function set containing only `noop` and `ephemeral-random-quantum-gate`; when `ephemeral-random-quantum-gate` is selected a new encapsulated gate is created with a random choice of gate type and all necessary parameters. It might also be useful to include a mutation operator that manipulates the parameters encapsulated within a gate, but we have not yet found it necessary to do so; our current mutation mechanism simply adds a new random gate (with random parameters) at a random location in the program. Surprisingly, this very simple mechanism suffices to evolve some interesting quantum algorithms, including the algorithm for the *and-or* problem discussed below (Section 7.4.4).

7.3.4 Fitness Function

We use a standardized fitness function (for which lower values mean "more fit") with three components: a *misses* component that records the number of fitness cases for which the program misbehaves, an *error* component that records the total error for all cases in which the program misbehaves, and a *length* component that records the total number of gates in the quantum algorithms built by the program for all fitness cases. A program is said to "misbehave" on a case if the probability that the quantum algorithm it produces will give the correct answer for the case is less than 0.48; this allows for the evolution of non-deterministic algorithms with *2-sided-error* (see section 7.2.3) and is sufficiently far from 0.5 to ensure that errors below the threshold are not due to roundoff errors in the quantum computer simulator.

The three components could be combined in various ways. We recommend a *lexicographic* [Ben-Tal, 1979] ordering, with the components ordered: misses (most significant), error, length (least significant). This means that programs will be compared first with respect to misses, with error being used only to break ties. Length will be used only to break ties between programs with identical misses scores and identical error scores.

The actual fitness function that we used in our runs approximates the lexicographic fitness function described above, but because it was developed and modified in an ad hoc fashion during the course of our work it varies from this fitness function in several minor ways. It is documented in full in [Spector et al., 1998].

7.4 Results

7.4.1 Deutsch's Early Promise Problem

In 1985 David Deutsch presented a problem for which quantum computers can clearly out-perform classical computers. This problem, like many in the quantum computation literature, involves determining properties of an unknown function. We can call the unknown function in programs that we write (or evolve) but we aren't given access to the function's code. For this reason the unknown function is often called a "black box" function or an "oracle."

Suppose you are given an oracle that computes an unknown binary function of n input bits. 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. Deutsch's *early promise problem* is the problem of determining whether such an oracle is uniform or balanced.

It is easy to see that the best deterministic classical algorithm for this problem will in the worst case require $\frac{2^n}{2} + 1$ oracle calls. 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 showed that quantum computers can do better here [Deutsch, 1985; Deutsch and Jozsa, 1992; Costantini and Smeraldi, 1997]. If the oracle is implemented as an 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. Note that this does not imply that we must know anything about the implementation of the oracle except that it is a well-behaved quantum mechanical operator.

We used the standard tree-based genetic programming techniques described above (Section 7.3.1) to automatically discover a quantum algorithm that provides information on the two-bit early promise problem using only one oracle call. We evolved quantum algorithms for a three-qubit quantum computer, using two qubits for the oracle's input and one for its output. The qubits are referred to with the indices 0, 1, and 2. For each fitness case the quantum computer was prepared in the initial state of $|000\rangle$, the algorithm was executed, and the result was then read from qubit 2. The algorithms 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. Each fitness case uses a different oracle function—the goal is to find a single quantum algorithm which puts qubit 2 into the "1" state if the oracle is uniform or into the "0" state if the

Table 7.1
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
function set	`+`, `-`, `*`, `%p`, `sqrt`, `1+`, `1-`, `*2`, `%2`, `1/x`, `iterate`, `ivar`, `iq`, `H-gate`, `U-theta-gate`, `CNOT-gate`, `NAND-gate`, `ORACLE-gate`
terminal set	`*num-qubits*`, `*num-input-qubits*`, `*num-output-qubits*`, 0, 1, 2, π, i

```
(IQ
 (NAND-GATE
  (+ (* (1- 0) (ITERATE PI PI))
     (U-THETA-GATE -1 (*2 *NUM-INPUT-QUBITS*)))
  (%2
   (+ (H-GATE (IQ (IQ (1- PI))))
      (ITERATE
        (1- (SQRT (CNOT-GATE (U-THETA-GATE 1 (IVAR *NUM-QUBITS*))
                             (IVAR (ITERATE PI *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 7.1
An evolved program that produces a quantum algorithm for the two-bit early promise problem.

```
U-theta qubit:2 theta:4
Hadamard qubit:0
U-theta qubit:1 theta:1
Oracle (input-qubits:0,1 output-qubit:2)
NAND input-qubits:2,1 output-qubit: 0
U-theta qubit:2 theta:4
Hadamard qubit:0
U-theta qubit:1 theta:2
Controlled-not control:1 target:2
(read output from qubit 2)
```

Figure 7.2
An quantum algorithm for the two-bit early promise problem, produced by the program in Figure 7.1. The system is initialized to the state $|000\rangle$ and then the algorithm is run, leaving qubit 2 in the "1" state with high probability if the provided oracle is uniform, or in the "0" state with high probability if the provided oracle is balanced. The final *Hadamard* gate on qubit 0 is unnecessary and can be removed.

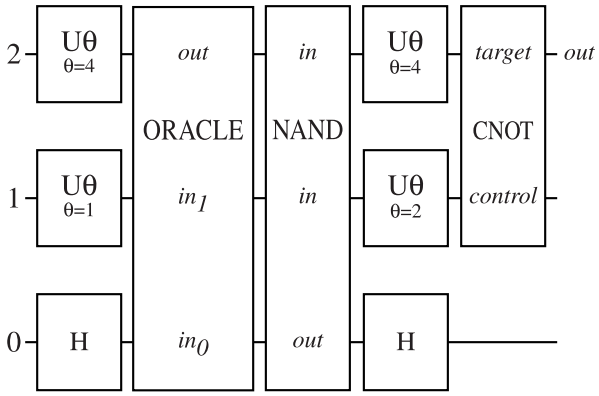

Figure 7.3
A graphic view of the quantum algorithm in Figure 7.2 for the two-bit early promise problem.

oracle is balanced. The oracle was set to use qubits 0 and 1 as inputs and qubit 2 as output, although in other experiments we allowed the oracle indices to evolve.

One run of this system, using Koza's Lisp genetic programming code [Koza, 1992] and the parameters shown in Table 7.1, produced the program shown in Figure 7.1 at generation 46. (The parameters in Table 7.1 were chosen by intuition and have not been optimized.) When executed, this program produces the quantum algorithm shown in Figure 7.2. Using notation similar to that in the quantum computation literature, this can be represented diagrammatically as in Figure 7.3.

The evolved algorithm is not minimal—at least the final H can be removed, although interestingly the *NAND* cannot. This is because of the way in which qubit values are distributed across the vector of amplitudes; it turns out that quantum gates can affect their "inputs" as well as their "outputs." The quantum algorithm in Figure 7.3 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. (There are only 8 possible oracles because only 8 of the 16 2-input boolean functions are either balanced or uniform.) 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.

While this result is not new to the field of quantum computation, it demonstrates that genetic programming can automatically find better-than-classical quantum algorithms.

7.4.2 The Scaling Majority-On Problem

Consider an oracle version of the *majority-on* problem. (Genetic programming is applied to the standard non-oracle version of majority-on by Koza [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 would be "1" if it were run on all possible inputs. In addition, we seek a single program that will produce correct quantum algorithms for oracles of any size. For example, if we have an oracle that takes 5 bits of input then we'd like the evolved program, when run with `*num-input-qubits*` set to 5 and other variables set appropriately, to produce a quantum algorithm which will reliably tell if the oracle outputs "1" for a majority of the possible inputs or not. Using standard tree-based genetic programming and similar parameters to those described above we evolved a program that produces the following quantum algorithms for this problem:

For one-bit oracles:

```
Hadamard qubit:0
Oracle input-qubit:0 output-qubit:1
(read output from qubit 1)
```

For two-bit oracles:

```
Hadamard qubit:1
Hadamard qubit:0
Oracle input-qubits:0,1 output-qubit:2
(read output from qubit 2)
```

For three-bit oracles:

```
Hadamard qubit:1
Hadamard qubit:2
Hadamard qubit:0
Oracle input-qubits:0,1,2 output-qubit:3
(read output from qubit 3)
```

For four-bit oracles:

```
Hadamard qubit:1
Hadamard qubit:2
Hadamard qubit:3
Hadamard qubit:0
Oracle input-qubits:0,1,2,3 output-qubit:4
(read output from qubit 4)
```

And so on; for each problem size the program produces a quantum algorithm that applies a *Hadamard* gate to each intput qubit and then calls the oracle. The algorithms work 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. It 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.

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, and in fact it does not appear that quantum computation can do any better than classical computation on this problem [Beals et al., 1998].

7.4.3 The Database Search Problem

The problem of searching an unsorted database for an item that it is known to contain (we're looking for its specific address) can also be recast as an oracle problem. We are given an oracle that accesses the database at a particular address and returns 1 if the item we're looking for is at that address, and 0 otherwise. The problem is to determine which address will cause the oracle to return 1.

Consider a four-item database, addressed via two binary inputs. On a deterministic classical machine we would have to query the database three times, in the worst case, to be sure about the location of the item we're looking for. If we haven't found it after three queries then we know that it is in the one location we haven't looked. But after only two luckless queries there would still be a 50% chance of error for any choice we could make.

Lov Grover showed that this is a problem for which quantum computers can beat classical computers. Grover's algorithm finds an item in an unsorted list of n items in $O(\sqrt{n})$ steps, while classical algorithms require $O(n)$. We initially thought this meant that the four-item database problem could be solved using two as opposed to the three classically-required database queries, and we conducted genetic programming runs to search for such a solution. We were happily surprised when the genetic programming system found a solution that uses only *one* database call and is nearly deterministic. Further examination revealed that Grover's algorithm also finds the item in one query, and that the solution found by genetic programming is in fact almost identical to Grover's algorithm.

We used stack-based, linear genome genetic programming (*MidGP*) with the parameters shown in Table 7.2, attempting to solve the four-item database problem with a five qubit system. The `DB-gate` function listed in Table 7.2 is analogous to the `ORACLE-gate` function from Section 7.4.1; it adds a call to the database lookup function (oracle) to the end of the quantum algorithm. The goal was to evolve a single quantum algorithm which, given a database containing a 1 only in position k (for k in {0, 1, 2, 3}), leaves qubits 3 and 4 in states q_3 and q_4 such that $2q_4 + q_3 = 3 - k$.[5]

Figure 7.4 lists the quantum algorithm produced by the best-of-run program. Notice that only four qubits are mentioned in the algorithm. In addition, both gates using qubit 1 can be eliminated without changing the behavior of the algorithm, so it requires only three qubits. The algorithm may be further simplified by omitting the 0-angle rotation along with the first *CPHASE* and the first *CNOT* (which are controlled by qubits in state $|0\rangle$, and hence act as the identity). The final *CPHASE* can be replaced with a *CNOT* because it has $\alpha = 1$. If we also combine the successive rotations on qubit 4 and change the resulting rotation angle in the fourth decimal place (to exactly $-\frac{\pi}{4}$; this eliminates an error probability of approximately 10^{-6}) then we get the quantum algorithm diagrammed in Figure 7.5. This algorithm acts just like a single iteration of Grover's algorithm except that it gives phases of -1 to some of the computational basis states, which has no effect on the final probabilities.

[5] It would have been more standard to use $2q_4 + q_3 = k$.

Table 7.2
MidGP parameters for a run on the four-item database search problem.

max number of generations	1,001
size of population	1,000
max program length	256
reproduction fraction	0.5
crossover fraction	0.1
mutation fraction	0.4
max mutation points	127
selection method	tournament (size=5)
function/terminal set	`noop`, `+`, `-`, `*`, `%p`, `DB-gate`, `H-gate`, `U-theta-gate`, `CNOT-gate`, `CPHASE-gate`, `U2-gate`, 0, 1, 2, 3, 4, π, `ephemeral-random-constant`, `pop`

```
U2 qubit:4 phi:0 theta:3 psi:3.14159 alpha:0.25908
Controlled-phase control-qubit:3 target-qubit:4, alpha:39.54646
Controlled-not control-qubit:0 target-qubit:3
U-theta qubit:0 theta:0.02934
Hadamard qubit:3
U-theta qubit:4 theta:3.14159
Hadamard qubit:0
Controlled-not control-qubit:1 target-qubit:3
U-theta qubit:4 theta:-4.06820
U-theta qubit:0 theta:-7.82538
Database-lookup input-qubits:4,3 output-qubit:0
Hadamard qubit:4
U-theta qubit:1 theta:4
U-theta qubit:3 theta:0
Controlled-phase control-qubit:3 target-qubit:4, alpha:0
Hadamard qubit:3
(read output from qubits 3 and 4)
```

Figure 7.4
Evolved quantum algorithm for the four-item database search problem on a five-qubit system. The system is initialized to the state $|00000\rangle$ and then the algorithm is run, leaving qubits 3 and 4 in states that indicate the position k of the single "1" in the database according to the formula $2q_4 + q_3 = 3 - k$.

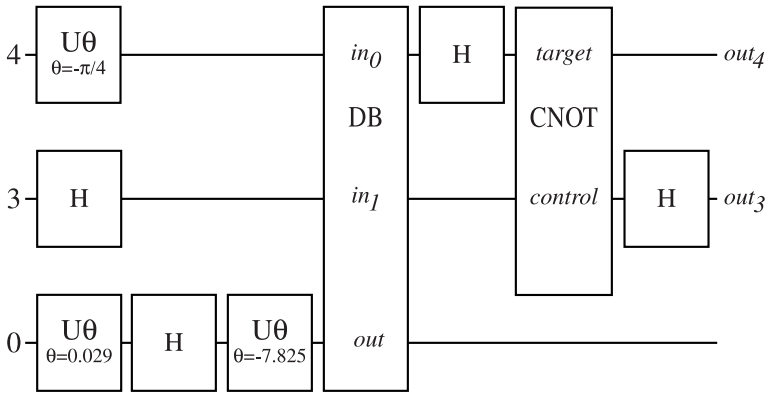

Figure 7.5
Diagram of the quantum algorithm for the four-item database search problem in Figure 7.4, reduced to use only the three essential qubits. This diagram also omits gates that have no effect, combines the rotations on qubit 4, and adjusts the combined rotation in the fourth decimal place to eliminate an error of 10^{-6}.

7.4.4 The And-Or Query Problem

The "and-or query problem" is the problem of determining whether a specific boolean function evaluates to true or false when applied to the values returned by a given oracle. The boolean function is an and-or binary tree with "AND" ($\wedge$) at the root, alternating layers of "OR" and "AND" ($\vee$) below, and the values of the oracle function, in order, at the leaves. For a one-bit oracle f, which has just the two values $f(0)$ and $f(1)$, the problem is to determine whether the expression "$f(0) \wedge f(1)$" is true or false. For a two-bit oracle f, with values $f(0)$, $f(1)$, $f(2)$, and $f(3)$, the problem is to determine whether the expression "$(f(0) \vee f(1)) \wedge (f(2) \vee f(3))$" is true or false. For a three-bit oracle the expression is "$((f(0) \wedge f(1)) \vee (f(2) \wedge f(3)) \wedge ((f(4) \wedge f(5)) \vee (f(6) \wedge f(7))$". And so on.

We chose to work on the two-bit oracle version of this problem because its quantum complexity is not yet completely understood and because we hoped that genetic programming could provide new information. Ronald de Wolf, a researcher who has worked on the quantum complexity of boolean functions [Beals et al., 1998], suggested this as an open problem and remarked that it would be "surprising" if there was a 2-sided-error solution that uses only one call to the oracle [de Wolf, personal communication]. Our genetic programming system found this "surprising" result.

We used stackless linear genome genetic programming (described above) with the parameters listed in Table 7.3. In generation 212 a program was evolved that produces the quantum algorithm in Figure 7.6. This algorithm works for all possible two-bit oracle functions, with all errors less than 0.41, using only a single call to the oracle function. We were able to analyze this algorithm and to improve and simplify it by hand, producing the al-

Table 7.3
MidGP MidGP parameters for a run on the two-bit and-or query problem.

max number of generations	1,000
size of population	100
max program length	32
reproduction fraction	0.2
crossover fraction	0.4
mutation fraction	0.4
max mutation points	8
selection method	tournament (size=7)
function/terminal set	`noop`, `ephemeral-random-quantum-gate`

gorithm in Figure 7.7. This algorithm's error is zero for the all-zero oracle function, $\frac{3}{8}$ for all other cases for which the correct answer is 0, and $\frac{1}{4}$ for the cases in which the correct answer is 1.

Notice that the quantum algorithm is better than the following classical probabilistic algorithm [Meyer, personal communication]:

1. Query the function for a random value of the input.

2. If the oracle returns 0, guess FALSE; else, guess TRUE.

Averaged over all inputs, this classical algorithm is correct $\frac{11}{16}$ of the time. Viewed in this way the quantum algorithm in Figure 7.6 is better but only slightly; it is correct $\frac{23}{32}$ of the time. On the other hand, the quantum algorithm is much better if one is considering only a single random input. In this case the classical algorithm will have an error probability of $\frac{1}{2}$ for six cases; that is, it is no better than guessing, even if run repeatedly. The quantum algorithm has a worst-case error probability of $\frac{3}{8}$, so it provides information about the correct answer that increeases with repetition.

One way to explain how this algorithm works is to use wave-mechanical descriptions of the quantum system. (Readers unfamiliar with wave mechanics may wish to skip the remainder of this paragraph.) To compute the OR function we use interference between the input states to the database gate. The purpose of this interference is to reinforce the amplitudes for bit values equal to "1" and to destructively interfere those for bit values equal to "0." The AND function at the root of the tree must simply effect an 'addition' of the 1 amplitudes with which it is provided. The algorithm achieves this task as follows: Remember that the database gate outputs the negation of the query result when bit 2 has initial value "1" and the result itself when that value is "0." Before querying the database the U_θ and *Hadamard* transform the state to a superposition with very unequal weight for states with bit-2 values "1" and "0." Following the database query, amplitudes for the two output values are mixed through a second rotation. Combined with the CNOT gate, which entangles the zeroth bit with the output register, this allows for interference *only* between the leaves of each of the OR nodes in the tree. The specific angle arguments of the gates ensure that the necessary amplitude pattern obtains.

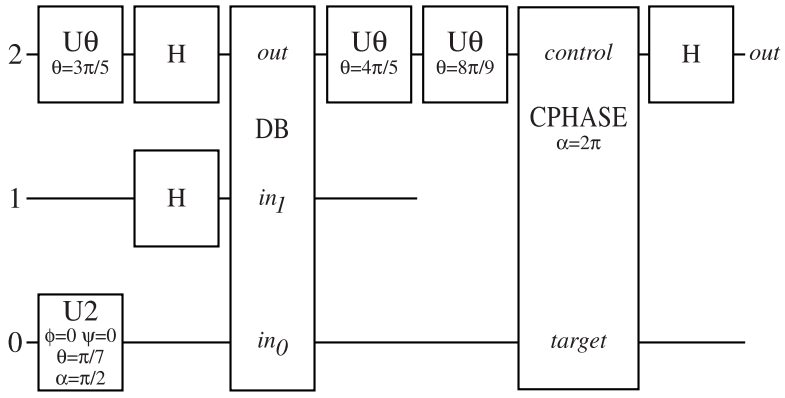

Figure 7.6
Evolved quantum algorithm for the two-bit and-or query problem. The system is initialized to the state $|000\rangle$ and then the algorithm is run, leaving qubit 2 in the "1" state with high probability if the "and-or" query is true for the provided oracle, or in the "0" state with high probability otherwise.

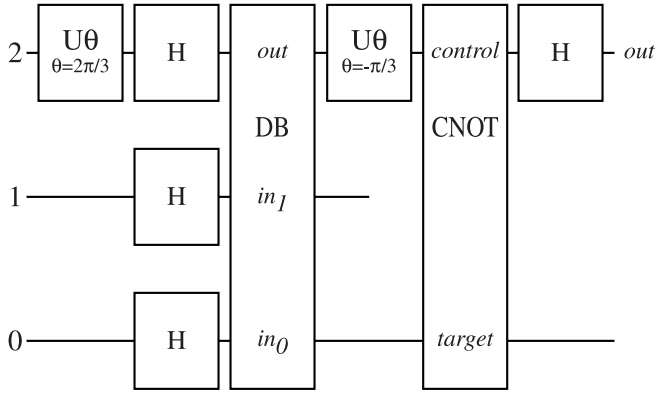

Figure 7.7
Hand-simplified and improved version of the quantum algorithm in Figure 7.6.

7.5 Conclusions

Genetic programming has been used to automatically discover new quantum algorithms, several of which are more efficient that any possible classical algorithms for the same problems, and one of which is more efficient than any previously known quantum algorithm for the same problem (Section 7.4.4). It has also been used to evolve quantum algorithms that can be scaled to work on problem instances of different sizes (Section 7.4.2).

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. Although we presented three different genetic programming approaches for quantum computation, we have not yet performed careful comparisons between these techniques or developed a theory about how genetic programming can best be applied in this area; this is a topic for future research. Other avenues for further investigation include:

- Application of the same techniques to other problems with incompletely understood quantum complexity.

- Modification of the techniques to support hybrid quantum/classical algorithms and quantum algorithms that include intermediate measurements.

- Genetic programming *on* quantum computers, using better-than-classical search algorithms that are already in the literature (such as Grover's) and other quantum computing efficiencies to speed up the genetic programming process.

Acknowledgements

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. Some work reported here was performed at the Institute's 1998 Research Conference on Quantum Computation, supported by ISI and the ELSAG-Bailey corporation. Ronald de Wolf provided valuable information on the and-or query problem and its complexity, and David Meyer and Bill Langdon provided essential reviewer's comments. Special thanks to Rebecca S. Neimark for assistance with the figures.

Bibliography

Barenco, A., Bennett, C. H., Cleve, R., DiVincenzo, D. P., Margolus, N., Shor, P., Sleator, T., Smolin, J. A., and Weinfurter, H. (1995), "Elementary gates for quantum computation," *Physical Review A*, 52:3457–3467.

Beals, R., Buhrman, H., Cleve, R., Mosca, M., and de Wolf, R. (1998), "Tight quantum bounds by polynomials," in *Proceedings of the Thirty-ninth Annual Symposium on Foundations of Computer Science (FOCS)*, To appear. Preliminary version available from `http://xxx.lanl.gov/abs/quant-ph/9802049`.

Beckman, D., Chari, A. N., Devabhaktuni, S., and Preskill, J. (1996), "Efficient networks for quantum factoring," Technical Report CALT-68-2021, California Institute of Technology, 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*, Fandel and Gal (Eds.), pp 1–11, Springer-Verlag.

Braunstein, S. L. (1995), "Quantum computation: a tutorial," Available only electronically, on-line at URL http://chemphys.weizmann.ac.il/~schmuel/comp/comp.html.

Chester, M. (1987), *Primer of Quantum Mechanics*, John Wiley & Sons, Inc.

Costantini, G. and Smeraldi, F. (1997), "A generalization of Deutsch's example," Los Alamos National Laboratory Quantum Physics E-print Archive, http://xxx.lanl.gov/abs/quant-ph/9702020.

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

Deutsch, D. and Jozsa, R. (1992), "Rapid solution of problems by quantum computation," in *Proceedings of the Royal Society of London A 439*, pp 553–558.

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

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

Jozsa, R. (1997), "Entanglement and quantum computation," in *Geometric Issues in the Foundations of Science*, S. Huggett, L. Mason, K. P. Tod, S. T. Tsou, and N. M. J. Woodhouse (Eds.), Oxford University Press, http://xxx.lanl.gov/abs/quant-ph/9707034.

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

Koza, J. R. and Bennett, III, F. H. (1999), "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.

Langdon, W. B., Soule, T., Poli, R., and Foster, J. A. (1999), "The evolution of size and shape," in *Advances in Genetic Programming 3*, L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline (Eds.), MIT Press.

Löwdin, P. (1998), *Linear Algebra for Quantum Theory*, John Wiley and Sons, Inc.

Milburn, G. J. (1997), *Schrödinger's Machines: The Quantum Technology Reshaping Everyday Life*, W. H. Freeman & Co.

Perkis, T. (1994), "Stack-based genetic programming," in *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pp 148–153, IEEE Press.

Preskill, J. (1997), "Quantum computing: Pro and con," Technical Report CALT-68-2113, California Institute of Technology, 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*, S. Goldwasser (Ed.), IEEE Computer Society Press.

Shor, P. W. (1998), "Quantum computing," *Documenta Mathematica*, Extra Volume ICM:467–486, http://east.camel.math.ca/EMIS/journals/DMJDMV/xvol-icm/00/Shor.MAN.ps.gz.

Spector, L. (1997), "MidGP, a Common Lisp stack-based genetic programming engine similar to HiGP," http://hampshire.edu/lspector/midgp1.5.lisp.

Spector, L., Barnum, H., and Bernstein, H. J. (1998), "Genetic programming for quantum computers," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 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 (Eds.), pp 365–374, Morgan Kaufmann.

Steane, A. (1998), "Quantum computing," *Reports on Progress in Physics*, 61:117–173, http://xxx.lanl.gov/abs/quant-ph/9708022.

Stoffel, K. and Spector, L. (1996), "High-performance, parallel, stack-based genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), pp 224–229, MIT Press.