Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2008), *22*, 275–283. Printed in the USA.
Copyright © 2008 Cambridge University Press 0890-0604/08 $25.00
doi: 10.1017/S0890060408000188

Machine invention of quantum computing circuits by means of genetic programming

LEE SPECTOR AND JON KLEIN
School of Cognitive Science, Hampshire College, Amherst, Massachusetts, USA

(RECEIVED February 15, 2006; ACCEPTED July 19, 2007)

Abstract

We demonstrate the use of genetic programming in the automatic invention of quantum computing circuits that solve problems of potential theoretical and practical significance. We outline a developmental genetic programming scheme for such applications; in this scheme the evolved programs, when executed, build quantum circuits and the resulting quantum circuits are then tested for "fitness" using a quantum computer simulator. Using the PushGP genetic programming system and the QGAME quantum computer simulator we demonstrate the invention of a new, better than classical quantum circuit for the two-oracle AND/OR problem.

Keywords: Artificial Intelligence; Genetic Programming; Machine Invention; Quantum Computing

1. MOTIVATION

The phrase "quantum computing" describes computational processes that rely for their efficacy on specifically quantum mechanical properties of information-processing hardware. Although all computing hardware obeys the laws of quantum mechanics (as far as we know), so-called "quantum computers" make use of counterintuitive features of the quantum world to achieve strikingly novel computational effects. Many of these features, such as "entanglement," have intrigued physicists since the early days of quantum theory, but their implications for computer science have become apparent only since the 1980s. Several introductions to quantum computing are available. These include, listed roughly from least to most technical, Brown (2000), Milburn (1997), Brooks (1999), Rieffel and Polak (2000), Steane (1998), Gruska (1999), and Nielsen and Chuang (2000).

The computational effects enabled by quantum computers support several novel and potentially significant applications. For example, the "tamper resistance" of quantum data states supports new forms of ultrasecure encryption, whereas "quantum parallelism" supports dramatic (and in some cases possibly exponential) improvements in computational complexity for problems of widespread applicability such as factoring. Many researchers in the field expect these applications to transform computer science and related industries in radical ways when large-scale quantum computers become widely available.

Unfortunately, however, quantum computers are counterintuitive and difficult to program. Many of the most important results in the field can be implemented with relatively simple circuits, but either their existence was not suspected prior to their discovery or, in some cases, their specific designs proved difficult for humans to devise. Fortunately, we can use automatic programming technologies, in particular genetic programming, to help us to search for quantum circuits. These technologies allow us to leverage computer power to explore the space of quantum circuits in a mechanical way. They can be applied to the discovery of new quantum effects and also to the exploitation of known quantum effects. They have already been used in this way to produce several significant results (Spector et al., 1998, 1999*a*; Barnum et al., 2000; Spector & Bernstein, 2002; Massey et al., 2004; Spector, 2004). A subset of the results in Spector (2004) was the basis on which a gold medal was awarded in the Human Competitive Results competition at the 2004 Genetic and Evolutionary Computation Conference (the criteria for which are described in Koza et al., 2000).

In the remainder of this article we first describe the quantum computer simulator (QGAME) that we use for assessing ("fitness-testing") candidate quantum circuits during genetic programming runs. We then describe the developmental genetic programming framework within which our most recent work in this area has been conducted. This is followed by a

Reprint requests to: Lee Spector, School of Cognitive Science, Hampshire College, Amherst, MA 01002-3359, USA. E-mail: lspector@hampshire.edu detailed example in which we demonstrate the evolution of a new, previously unpublished quantum circuit for a problem of potential theoretical significance.

2. QUANTUM CIRCUIT SPECIFICATION AND SIMULATION

This work adopts the "quantum gate array" or "quantum circuit" model of quantum computing, in which computations are expressed as sequences of "quantum gates" acting on "quantum bits" (also called "qubits"). In a true quantum computer the qubits would be embodied as photons, nuclear spins, trapped atoms, or other two-state physical systems, and the quantum gates would be implemented as processes or configurations that transform those systems. Such hardware is currently exotic and expensive, so the work described here uses a simulator that runs on ordinary classical digital hardware. Such simulators are necessarily inefficient, in many cases they are *exponentially* costly in terms of time and space resources, but they allow us to determine how a quantum circuit would perform *if* it were implemented in real quantum hardware.

The details of the quantum circuit model and its simulation are beyond the scope of this article; a full presentation can be found in Spector (2004). For the purposes of this article it is only necessary that the reader understand the nature of the inputs and outputs to the quantum computer simulator that we employed, QGAME (Quantum Gate and Measurement Emulator).[1]

QGAME provides a syntax for the expression of quantum circuits and also an interpreter that simulates their performance. A QGAME circuit specification consists of a sequence of "instruction expressions," each of which is surrounded by parentheses. The most typical instruction expressions consist of the name of a gate type, followed by a combination of qubit indices (specifying to which qubit or qubits the gate is to be applied) and other parameters (such as angles or matrices). For example, an expression of the form

(QNOT q)

where q is a qubit index (a nonnegative integer), applies a quantum "not" (QNOT) gate to the specified qubit. Similarly, an expression of the form:

(CNOT q_{control} q_{target})

applies a quantum controlled NOT gate to the specified control and target qubits. Instruction expressions following the same pattern for other common quantum gates (described in detail in Spector, 2004) are as follows:

(SRN q)
(HADAMARD q)
(U-THETA q θ)
(U2 q ϕ θ ψ α)
(CPHASE q_{control} q_{target} α)
(SWAP q_{control} q_{target})

QGAME also provides a way to specify circuits that include so-called "oracle" gates with any number of inputs and one output. Each of these gates has one of two possible effects on its output qubit on any particular invocation, but unlike classical Boolean logic gates these do not act by *setting* their output qubits to 0 or 1 but rather by *flipping* or *not flipping* their output qubits to indicate outputs of 1 or 0, respectively. During the testing of a circuit that contains an oracle gate one normally tests the circuit with various instances of the oracle (implementing different Boolean functions) and collects statistics over all of the results. This is facilitated in QGAME with an instruction expression of the form:

(ORACLE Ω q_1 q_2 $\cdots$ q_n q_{out})

Ω should be the right-hand column of a Boolean truth table that specifies the action of the ORACLE gate, listed in parentheses and in binary order. The q_1 q_2 $\cdots$ q_n parameters are the indices of the input qubits, and q_{out} is the index of the output qubit. For example, the following expression:

```
(ORACLE (0 0 0 1) 2 1 0)
```

represents a gate that flips qubit 0 when (and only when) the values of qubits 2 and 1 are both 1 (the bottom line of the truth table). If Ω in an ORACLE expression is the symbol ORACLE-TT then this indicates that the interpreter should substitute a valid truth table specification in place of the symbol before execution; this is normally in the context of a call to TEST-QUANTUM-PROGRAM, which we will use in the fitness function for genetic programming runs (see below). It is sometimes useful to limit the number of times that an oracle can be used during a single simulation. For this reason QGAME also provides an instruction expression of the form:

(LIMITED-ORACLE *max* Ω q_1 q_2 $\cdots$ q_n q_{out})

This works just like ORACLE the first *max* times it is executed in a simulation; after *max* executions it has no further effect.

QGAME also provides a way to simulate the effects of single-qubit *measurements* during the execution of a quantum program, and allows for the outcomes of those measurements to influence the remainder of the simulation. In an actual run of a quantum computer such measurements would, in general, be probabilistic. However, because we generally wish, when performing our simulations, to obtain the actual probabilities for various outputs and not just particular

[1] Implementations of QGAME in C++ and Lisp are available from http://hampshire.edu/lspector/qgame.html. Lisp source code for the core components of QGAME is also provided in Spector (2004).

(probabilistically chosen) outputs, QGAME simulates *all* possible measurement outcomes. This is done by branching the entire simulation and proceeding independently for each possible outcome. In each branch the measured qubit is forced to the measured value. The probability for taking each branch is recorded, and output probabilities at the end of the simulation are calculated on the basis of all possible final states and the probabilities of reaching them.

The syntax for a QGAME measurement is as follows:

(MEASURE *q*) *branch*$_1$ (END) *branch*$_0$ (END)

This is actually a sequence of instruction expressions, beginning with the MEASURE expression that specifies the qubit to measure. Any number of instruction expressions may occur between the MEASURE expression and the first following END; all of these will be executed in the branch of the simulation corresponding to a measurement of 1. Similarly, any number of instruction expressions may occur between the first following END and a subsequent END; all of these will be executed in the branch of the simulation corresponding to a measurement of 0. Instruction expressions following the second END will be executed in both branches of the simulation, following the execution of the branch-specific instructions. If there is no END following the MEASURE expression then the entire remainder of the circuit specification is *branch*$_1$ and there is no *branch*$_0$. Similarly, if there is only one subsequent END then the entire circuit specification beyond that END is *branch*$_0$. Unmatched ENDs are ignored.

Additional instruction expressions, not used in this article, are described in Spector (2004).

The interface to the QGAME interpreter that we use in our genetic programming fitness test is TEST-QUANTUM-PROGRAM. This call takes the following inputs:

- PROGRAM: the circuit specification to be tested, in QGAME syntax.
- NUM-QUBITS: the number of qubits in the quantum computer to be simulated.
- CASES: a list of "(*oracle-truth-table output*)" pairs, where each *oracle-truth-table* is a sequence of 0s and 1s specifying the right-hand (output) column of the oracle's truth table (where the rows are listed in binary order), and where the *output* is the correct numerical answer for the given truth table; the test compares this number to the number read from the final measurement qubits at the end of the computation.
- FINAL-MEASUREMENT-QUBITS: a parenthesized list of indices specifying the qubits upon which final measurements will be performed, with the most significant qubit listed first.
- THRESHOLD: the probability of error below which a run is considered successful for the sake of the "misses" component of the return value (see below). This is typically set to something like 0.48, which is usually far enough from 0.5 to ensure that the "better than random guessing" performance of the algorithm is not because of accumulated round-off errors.

TEST-QUANTUM-PROGRAM returns a list containing the following values:

- The number of "misses"; that is, cases in which the measured value will, with probability greater than the specified threshold, fail to equal the desired output.
- The maximum probability of error for any provided case.
- The average probability of error for all provided cases.
- The maximum number of expected oracle calls across all cases.
- The number of expected oracle calls averaged across all cases.

When using genetic programming to evolve quantum circuits these outputs can be combined in various ways to suit the particular problem under study. In many of the applications that we have explored to date (and in the example presented below) the first and second values have been summed to produce a fitness value that is zero for a "perfect" algorithm, prioritizing the number of "misses" (which is always a whole number) above the maximum probability of error (which is always between zero and one).

3. APPLICATION OF GENETIC PROGRAMMING

A genetic programming system is a genetic algorithm (Holland, 1992) in which the chromosomes are executable computer programs (Koza, 1992). In these systems one begins with a population of computer programs, each of which is a random composition of elements from a problem-specific set of program elements. Fitness is assessed by running each program on a specified set of inputs and producing a numerical value that characterizes the quality of the program's performance. Programs that perform better are preferentially selected for reproduction and most reproducing programs are subject to small random changes ("mutations") or to recombination with one another ("crossover"). As generation follows generation one usually observes the emergence of better and better programs. The process terminates when a sufficiently good program has been found or when a predetermined number of generations has been processed. Genetic programming often requires large-scale computational resources in order to find solutions, but fortunately, it is also easily parallelized across computer clusters. A typical parallelization scheme involves the loose coupling, through occasional program "migrations," of multiple subpopulations (often called "demes") that run otherwise independently on separate processors.

The standard genetic programming technique will not be described further here; details are available in an earlier paper in this special issue (Koza, 2008) and from other sources (Koza, 1992, 1994; Banzhaf et al., 1997; Koza et al., 1999,

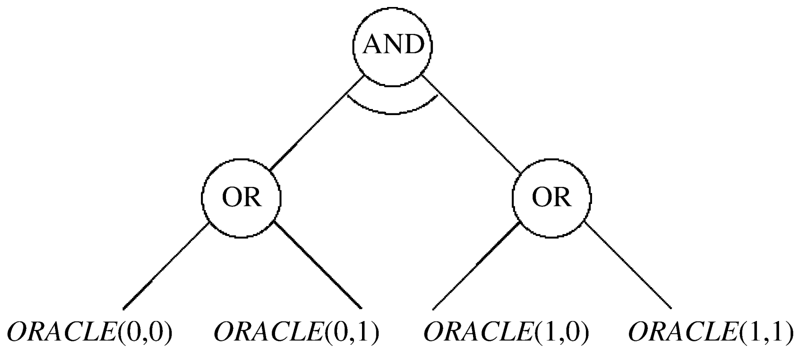

Fig. 1. An AND/OR tree describing the nature of the AND/OR oracle problem.

2003). Variations are regularly reported at several international conferences with published proceedings, most notably the *Genetic and Evolutionary Computation Conference* (GECCO), in journals such as *Genetic Programming and Evolvable Machines* and *Evolutionary Computation*, and in edited books (Kinnear, 1994; Angeline & Kinnear, 1996; Spector et al., 1999c; Riolo & Worzel, 2003; O'Reilly et al., 2004; Yu et al., 2005; Riolo et al., 2007). A searchable, online bibliography on genetic programming is also available.[2]

The genetic programming system used in the work reported here is PushGP (Spector, 2001, 2004; Spector & Robinson, 2002; Spector et al., 2005). PushGP differs from most other genetic programming systems in that it uses the Push programming language as the formalism within which evolving programs are expressed. Most other systems, by contrast, use Lisp-like expression trees, although a variety of representations (including both high-level languages and machine code) have also been used.

The use of the Push programming language has several implications for genetic programming in general, but the work described in this article is independent of those implications. To explain both the general principles of our approach and the specific example presented below we need only note that Push is a stack-based language that includes a stack for each data type and uses postfix syntax. This means that an expression like "`1 3.14 2 2.72 INTEGER.+ FLOAT.*`" adds 1 and 2 and also multiplies 3.14 and 2.72 as follows: it pushes 1 onto the integer stack, then 3.14 onto the float (floating-point number) stack, then 2 onto the integer stack, then 2.72 onto the float stack. It then executes `INTEGER.+`, which pops two integers, adds them, and pushes the result back onto the integer stack. It then executes `FLOAT.*`, which pops two floats, multiplies them, and pushes the result back onto the float stack. More details on the Push programming language, including documents that demonstrate its benefits and free source code for several implementations, is available online.[3]

[2] http://liinwww.ira.uka.de/bibliography/Ai/genetic.programming.html
[3] http://hampshire.edu/lspector/push.html

In our quantum computing applications we use genetic programming to find structures (quantum circuits) of unknown topology and parameters. As is traditional in such cases, we obtain the topology and parameters of each candidate solution by means of a developmental approach (Wilson, 1987; Kitano, 1990; Gruau, 1993; Koza et al., 1996; Spector & Stoffel, 1996; Hornby et al., 2007). In the developmental approach the execution of a program in the population builds a structure that is subsequently assessed for fitness. In this application we begin each program execution with an "embryo" that consists of a minimal topology, containing only the final measurement gates, and the execution of certain instructions causes additions to be made to that embryo. For example, the execution of the `U-THETA` instruction takes an argument from the integer stack and an argument from the float stack and adds a `U-THETA` gate expression to the QGAME circuit specification under construction.[4] As a program runs it builds a quantum circuit specification, expressed in the QGAME syntax, and this circuit specification is then submitted to QGAME for evaluation.[5]

4. ILLUSTRATION ON A NEW PROBLEM

In previous work we used genetic programming to produce significant new results for the AND/OR oracle problem (Spector et al., 1999b; Barnum et al., 2000; Spector, 2004). In this problem the task is to determine whether or not a particular given two-input, one-output oracle flips its output qubit under the conditions illustrated in Figure 1. That is, we are asked to determine if the cases for which the oracle flips its output qubit satisfy the logical formula $(I_{00} \vee I_{01}) \wedge (I_{10} \vee I_{11})$, where I_{ab}

[4] The integer argument is taken modulo the number of qubits in the system. For the runs described here no gate is added for multiqubit gate specifications if the same qubit is specified more than once, and the output qubit for oracles is chosen as the lowest numbered qubit not already specified for that gate.

[5] In some of our work we use a more complex scheme in which instructions like `QGATE.U-THETA` push unitary matrices onto a quantum gate stack where they may be manipulated and combined with other gates; a subsequent call to `QGATE.GATE` is then needed to add the resulting gate to the developing circuit (Spector, 2004). For the present work we used a simpler scheme in which gate-related instructions add gates directly to the developing circuit.

indicates whether or not the output is flipped for the input (a, b).

Note that the classical version of this problem, of determining with certainty whether a classical two-input, one-output gate has the AND/OR property, would require, in the worst case, four oracle accesses; one might have to check the oracle's behavior for all four possible inputs. Probabilistically, however, one can do reasonably well with only a single oracle access; in fact, it is possible to devise a probabilistic classical algorithm that accesses the oracle only once, and nonetheless has a maximum probability of error of only one-third. Previous application of genetic programming to this problem produced quantum circuits that perform better than any possible classical algorithm, giving a maximum probability of error of only 0.28731 (Barnum et al., 2000; Spector, 2004). This result filled a gap in the then existing knowledge of these types of problems[6] and contributed to the ongoing theoretical investigation of quantum computing.

If the quantum advantage in the AND/OR problem can be generalized to larger oracles or applied to more general oracle problems, then significant new applications might result; for example, we noted in Spector (2004) that AND/OR trees have wide applicability in artificial intelligence and that "one might therefore speculate that the quantum speedups discovered for the AND/OR problem may support some form of 'quantum logic machine.'" Unfortunately, nonclassical features of quantum circuits foil efforts to extend the results through simple concatenations of the evolved circuits. It is therefore useful to apply genetic programming to variants of the original problem in the hopes that new results will allow us to derive general principles.

In the present article we demonstrate the invention, by genetic programming, of circuits for a variant of the AND/OR problem in which two, rather than one, accesses to the oracle are permitted. The lowest error probability obtainable by a probabilistic classical algorithm on this problem is $1/6 = 0.1666...$, so any quantum circuit that achieves a lower error probability may be of interest.[7]

We conducted 23 independent single-CPU runs on our Linux cluster; although we often use this cluster for parallel deme models we allowed no migration in the runs reported here. Each run used a minimal instruction set which included none of Push's code-manipulation or even integer-manipulation instructions; the full listing of parameters is shown in Table 1. In these runs we also used a "trivial geography" tournament scheme in which the program at location L at generation G is produced only from programs in locations in the range $(L - R, L + R)$ at generation $(G - 1)$, where R is a small "neighborhood radius" and all locations are taken modulo the population size (Spector & Klein, 2005).

[6] Evidence of this is the publication of the result in the peer-reviewed physics (not computing) literature (Barnum et al., 2000).

[7] Thanks to Howard Barnum and Herbert J. Bernstein for determining the classical lower bound.

Table 1. *Parameters for the example run of genetic programming on the two-oracle AND/OR problem*

Objective	Design a quantum circuit that achieves better than classical error probability for the two-oracle AND/OR problem.
Embryo	Three-qubit quantum circuit with a final measurement on one qubit [index 2 of (0–2)].
Instructions	`FLOAT.% FLOAT.* FLOAT.+ FLOAT.- FLOAT./ FLOAT.DUP FLOAT.POP FLOAT.SWAP FLOAT.FROMINTEGER LIMITED-ORACLE HADAMARD U-THETA MEASURE SRN CNOT U2 CPHASE SWAP END`
Ephemeral random constants	INTEGER FLOAT
Fitness cases	All possible two-input, one-output Boolean oracles, specifically ($I_{00}I_{01}I_{10}I_{11}$:*answer*): 0000:0, 0000:0, 0010:0, 0010:0, 0100:0, 0100:1, 0110:1, 0110:1, 1000:0, 1000:1, 1010:1, 1010:1, 1100:0, 1100:1, 1110:1, 1110:1
Fitness function	*Misses* + *MaxError* where *Misses* is the number of cases for which the probability of error is greater than 0.48 and *MaxError* is the maximum probability of error of any case.
Population size	5000
Generation limit	800
Integer constant range	(−10, 10)
Float constant range	(−10.0, 10.0)
Initial program size limit	50
Child program size limit	250
Program evaluation limit	250
Operators and rates	Crossover 40%, mutation 40% [size fair (Crawfod-Marks & Spector, 2002) range 0.3], duplication 20%
Tournament size	7
Tournament neighborhood radius	5

One of the results produced by these runs was the following Push program:

```
((FLOAT.- FLOAT.% 1 (7) (FLOAT.% (CPHASE 1 (SRN
CPHASE FLOAT.*) 2 (LIMITED-ORACLE)) SRN FLOAT./
(((-6 -10) 7) MEASURE U-THETA -0.26584211533559537
FLOAT.*) (FLOAT.FROMINTEGER U-THETA
LIMITED-ORACLE 0.667472529147 62878)
FLOAT.DUP) 6) 6 (FLOAT.FROMINTEGER (FLOAT.
FROMINTEGER -10 HADAMARD (((-10 (SRN)) SRN
(FLOAT.*) CPHASE U-THETA) 6 LIMITED-ORACLE) -
0.56456115878293323) ((HADAMARD CPHASE) -10 SRN)
MEASURE LIMITED-ORACLE (FLOAT.-) FLOAT.POP)
((SWAP ((CPHASE END) -10 LIMITED-ORACLE HADAMARD
MEASURE) (((FLOAT.POP) END LIMITED-ORACLE
((FLOAT.POP FLOAT.DUP) -9) (CNOT -
0.31790071337434944 FLOAT.* U-THETA)) -10
HADAMARD (-10 (FLOAT.SWAP)) SRN) (FLOAT.POP)
(FLOAT.DUP CPHASE) HADAMARD -0.31790071337434944)
5) (FLOAT.FROMINTEGER (((CPHASE ((FLOAT.POP) END
```

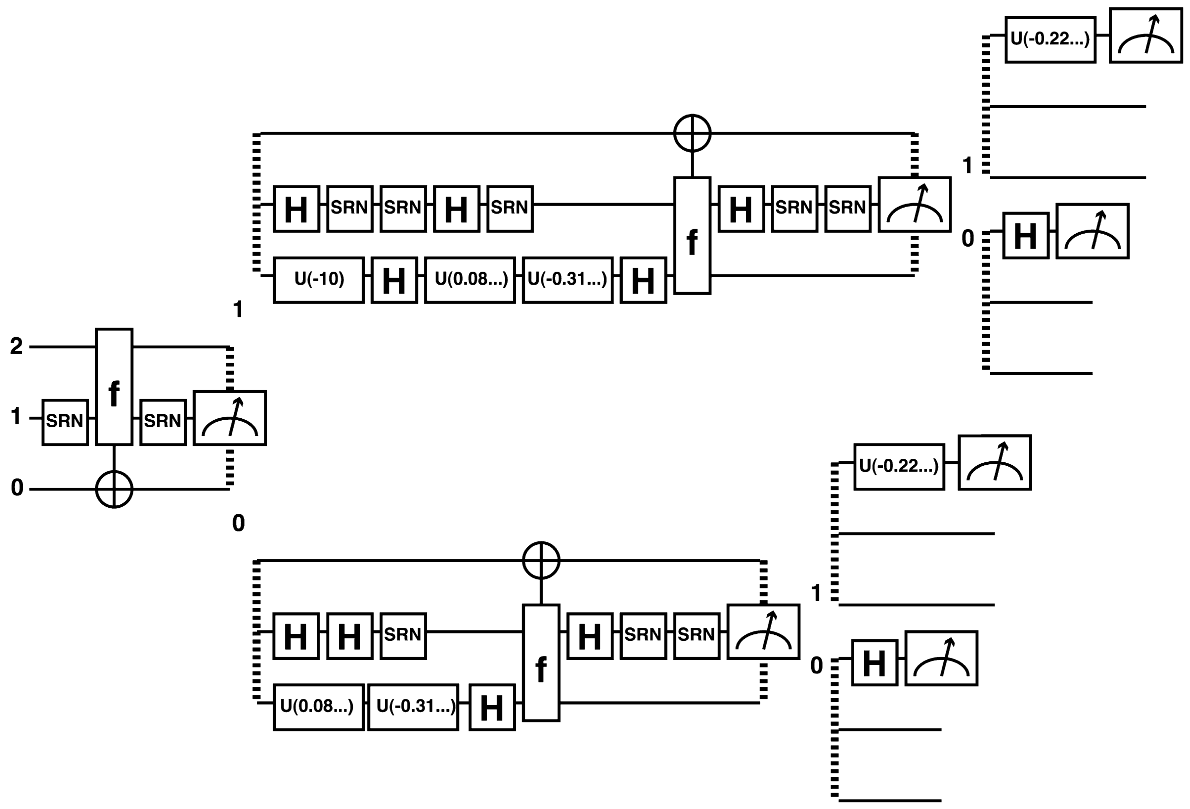

Fig. 2. Evolved quantum circuit for the two-query AND/OR problem, diagrammed according to the conventions in Spector (2004). This circuit has a maximum probability of error of 0.15177141700934016, which is better than that of any classical probabilistic algorithm. The "f" gates are the calls to the oracle, only two of which will be used on any particular execution (the one in the initial sequence followed either by the upper or the lower branch). This diagram shows the circuit specified by the pruned QGAME specification presented in the text, but further simplifications are possible.

```
LIMITED-ORACLE ((FLOAT.POP FLOAT.DUP) -9)
(CNOT -0.31790071337434944 FLOAT.* U-THETA))
((FLOAT.SWAP FLOAT.-) (MEASURE) (((-10 6) FLOAT./)
-9) FLOAT.SWAP HADAMARD FLOAT.DUP)) -10 FLOAT.
SWAP) 7 FLOAT.* -0.56456115878293323 (END)) -6
LIMITED-ORACLE (END -0.33716124234189310
HADAMARD (((-10 (SRN)) -10 ((SRN SRN) MEASURE 0
FLOAT.- (SRN)) -8 (-10 SRN)) (U-THETA -6 SRN 2)
FLOAT./) (-10 SRN)) FLOAT.POP END) HADAMARD)
```

When run, this Push program produces a QGAME quantum circuit specification that, after the pruning of expressions that have no effect, is as follows:

```
(SRN 1) (LIMITED-ORACLE 2 ORACLE-TT 2 1 0) (SRN 1)
(MEASURE 1) (U-THETA 0 -10) (HADAMARD 1) (SRN 1)
(HADAMARD 0) (SRN 1) (END) (HADAMARD 1) (END)
(U-THETA 0 0.084511) (HADAMARD 1) (SRN 1) (U-THETA
0 -0.317901) (HADAMARD 0) (LIMITED-ORACLE 2
ORACLE-TT 0 1 2) (HADAMARD 1) (SRN 1) (SRN 1)
(MEASURE 1) (U-THETA 2 -0.2274) (END) (HADAMARD 2)
```

This circuit, which is diagrammed in Figure 2, achieves zero "misses" (i.e., it has a probability of error of less than 0.48 for each fitness case) and has a maximum probability of error of 0.15177141700934016. Because the best classical probabilistic algorithm has a maximum probability of error of 1/6 = 0.1666... , this result is better than classical. To our knowledge no better than classical solution to this problem has previously been published.

To further explore the evolved circuit's potential we conducted a second round of 23 runs with the same parameters (as listed in Table 1) except that we "seeded" each population with one instance of the Push program produced in the previous round. One of the runs in the second round produced the following program:

```
((END (3.7480855299504672 U-THETA SRN) (-10)
FLOAT./ (CPHASE (U2 -10 (SRN (-10) FLOAT./) 2
(LIMITED-ORACLE)) SRN FLOAT.POP (((U2 -10) 7)
MEASURE FLOAT.+ LIMITED-ORACLE FLOAT./)
(-0.31790071337434944 FLOAT.+ FLOAT.% FLOAT./)
FLOAT.DUP) 6) 6 (6 (-9 -10 (SRN) (((-10 (SRN)) SRN
(2.5552216434281370 CPHASE) CPHASE FLOAT.*) 0
LIMITED-ORACLE) FLOAT.%) ((FLOAT./
-0.31790071337434944 LIMITED-ORACLE U-THETA) -10
6.96903464 77946816) HADAMARD U- THETA MEASURE
FLOAT.SWAP) ((MEASURE ((FLOAT.% END) -10 SRN
HADAMARD FLOAT.%) (END -10 HADAMARD (-10 CPHASE)
SRN) FLOAT.FROMINTEGER U-THETA FLOAT.- -8) CNOT)
```

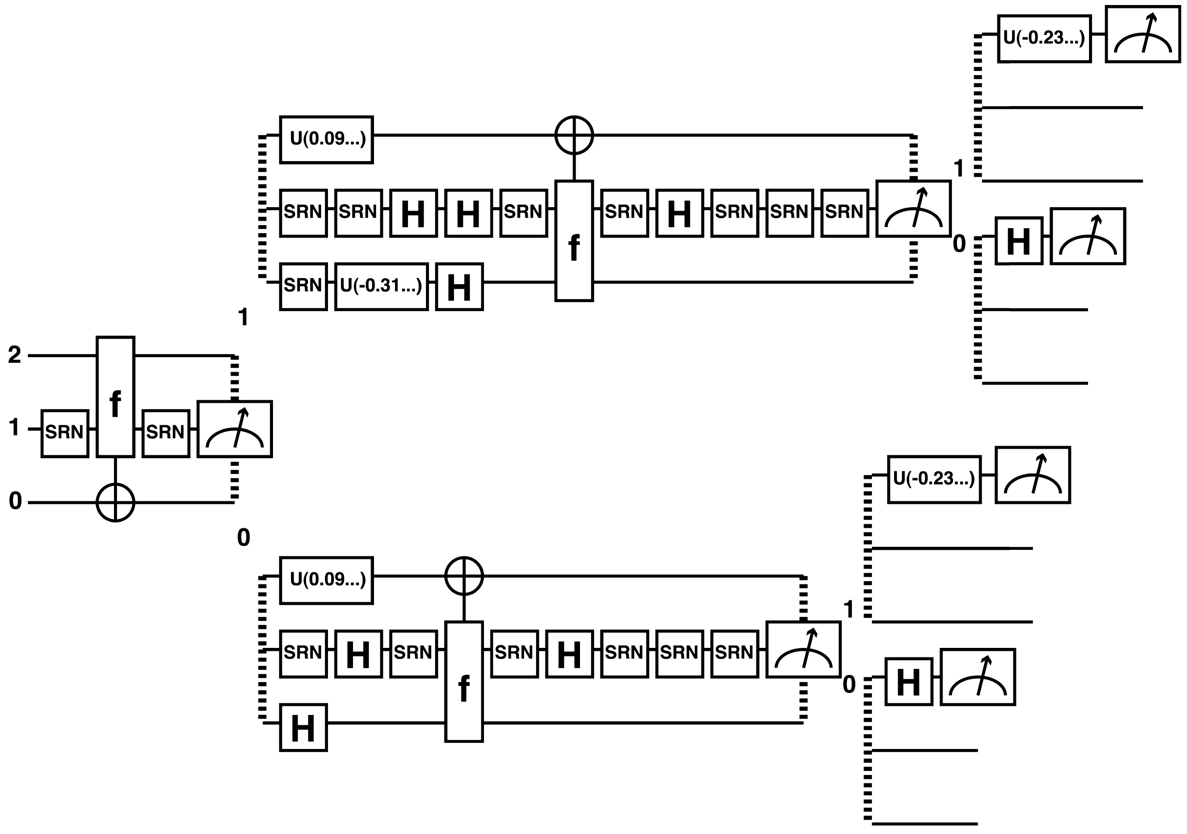

Fig. 3. Evolved quantum circuit for the two-query AND/OR problem, produced in a run that was seeded with the solution diagramed in Figure 2. This circuit has a maximum probability of error of 0.1089010279090783, which is better than that of any classical probabilistic algorithm. The "f" gates are the calls to the oracle, only two of which will be used on any particular execution (the one in the initial sequence and then either the one in the upper or the lower branch). This diagram shows the circuit specified by the pruned QGAME specification presented in the text, but further simplifications are possible.

```
(U-THETA (((SWAP ((-8 (-10) (-10 CPHASE)) 7)
(HADAMARD (U-THETA -10) -9 FLOAT.DUP HADAMARD
HADAMARD)) (U-THETA -10) U-THETA) U-THETA FLOAT.*
FLOAT.* CPHASE) -6 LIMITED-ORACLE (((SRN (-10)
(-10 -10)) 7) -9.7762992418781618 HADAMARD (((((SRN
(-10) LIMITED-ORACLE) 7) END) -10 ((SRN SRN) MEASURE
-8 -10 -0.31790071337434944) -8 (-10 HADAMARD))
(FLOAT.% U-THETA SRN) FLOAT./) END) (HADAMARD
(U-THETA -10) -9 END HADAMARD HADAMARD) -10) HADAMARD)
```

When run, this Push program produces a QGAME quantum circuit specification which, after the pruning of expressions that have no effect, is as follows:

```
(SRN 1) (LIMITED-ORACLE 2 ORACLE-TT 2 1 0) (SRN 1)
(MEASURE 1) (SRN 1) (SRN 1) (SRN 0) (U-THETA
0 -0.317901) (HADAMARD 1) (END) (SRN 1) (END)
(HADAMARD 1) (SRN 1) (U-THETA 2 0.096896)
(HADAMARD 0) (LIMITED-ORACLE 2 ORACLE-TT 0 1 2)
(SRN 1) (HADAMARD 1) (SRN 1) (SRN 1) (SRN 1)
(MEASURE 1) (U-THETA 2 -0.239278) (END) (HADAMARD 2)
```

This circuit, which is diagrammed in Figure 3, has a maximum probability of error of 0.1089010279090783, which we suspect may be nearly minimal; the best we have obtained in any run to date has a maximum error of 0.108720808 49223331. Notice that most of the structure and many of the parameters of the solution from the first round of runs are preserved in this refined solution.

5. CONCLUSIONS AND PROSPECTS

Quantum computing holds great promise, but the design of quantum circuits is a black art that requires substantial expertise. Even the most skilled theorists can fail to see that certain quantum efficiencies are possible. This is true even when the desired quantum efficiencies can be obtained from relatively simple circuits.

Fortunately, genetic programming systems can automatically invent quantum circuits that solve some of these problems. In this article we showed how relatively straightforward genetic programming techniques, based on a well-established developmental approach that has also been applied to a range of other applications, can effectively search the space of quantum circuits to find solutions to previously unsolved problems. In particular, we demonstrated the evolution of a new, better than classical quantum circuit for the two-query AND/OR problem. This problem has potential theoretical and practical significance, and the result presented here is, as far as we know, the best that has been published.

The full significance of this new result on the two-oracle AND/OR problem is not yet known. Our next steps on this front will be to further simplify and analyze the evolved circuits and to try to understand and generalize the mechanisms by which they achieve their performance.

What is clear at this point, however, is that genetic programming is capable of inventing quantum circuits that can help to move the study of quantum computing forward.

ACKNOWLEDGMENTS

Manuel Nickschas wrote the version of QGAME and Maarten Keijzer wrote the version of Push that was used for this work. This material is based upon work supported by the National Science Foundation under Grants 0308540 and 0216344. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Angeline, P.J., & Kinnear, Jr., K.E. (1996). *Advances in Genetic Programming 2.* Cambridge, MA: MIT Press.

Banzhaf, W., Nordin, P., Keller, R.E., & Francone, F.D. (1997). *Genetic Programming—An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* San Mateo, CA: Morgan Kaufmann.

Barnum, H., Bernstein, H.J., & Spector, L. (2000). Quantum circuits for OR and AND of ORs. *Journal of Physics A: Mathematical and General 33(45)*, 8047–8057.

Brooks, M. (1999). *Quantum Computing and Communications.* London: Springer–Verlag.

Brown, J. (2000). *Minds, Machines and the Multiverse: The Quest for the Quantum Computer.* New York: Simon & Schuster.

Crawford-Marks, R., & Spector, L. (2002). Size control via size fair genetic operators in the PushGP genetic programming system. *Proc. Genetic and Evolutionary Computation Conf.*, pp. 733–739.

Gruau, F. (1993). Genetic synthesis of modular neural networks. *Proc. 5th Int. Conf. Genetic Algorithms*, pp. 318–325.

Gruska, J. (1999). *Quantum Computing.* New York: McGraw–Hill.

Holland, J.H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* Cambridge, MA: MIT Press.

Hornby, G.S., Kumar, S., & Jacob, C. (2007). Editorial introduction to the special issue on developmental systems. *Genetic Programming and Evolvable Machines 8(2)*, 111–113.

Kinnear, Jr. K.E. (1994). *Advances in Genetic Programming.* Cambridge, MA: MIT Press.

Kitano, H. (1990). Designing neural networks using genetic algorithms with graph generation system. *Complex Systems 4(4)*, 461–476.

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

Koza, J.R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press.

Koza, J.R. (2008). From computer aided design to human-competitive machine invention by means of genetic programming. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 22(3)*, 185–193.

Koza, J.R., Andre, D., Bennet III, F.H., & Kean, M. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving.* San Mateo, CA: Morgan Kaufman.

Koza, J.R., Bennett III, F.H., Andre, D., & Keane, M.A. (1996). Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In *Artificial Intelligence in Design '96* (Gero, J.S. & Sudweeks, F., Eds.), pp. 151–170. Dordrecht: Kluwer Academic.

Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J., & Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Boston: Kluwer Academic.

Koza, J.R., Keane, M.A., Yu, J., Bennett III, F.H., & Mydlowec, W. (2000). Automatic creation of human-competitive programs and controllers by means of genetic programming. *Genetic Programming and Evolvable Machines 1(1/2)*, 121–164.

Massey, P., Clark, J., & Stepney, S. (2004). Human-competitive evolution of quantum computing artefacts by genetic programming. *Evolutionary Computation 14(1)*, 21–40.

Milburn, G.J. (1997). *Schrödinger's Machines: The Quantum Technology Reshaping Everyday Life.* New York: Freeman.

Nielsen, M.A., & Chuang, I.L. (2000). *Quantum Computation and Quantum Information.* Cambridge: Cambridge University Press.

O'Reilly, U.-M., Yu, T., Riolo, R.L., & Worzel, B. (2004). *Genetic Programming Theory and Practice II.* New York: Springer.

Rieffel, E., & Polak, W. (2000). *An Introduction to Quantum Computing for Non-Physicists*. Accessed at http://arxiv.org/quant-ph/9809016

Riolo, R.L., Soule, T., & Worzel, B. (2007). *Genetic Programming Theory and Practice IV.* New York: Springer.

Riolo, R.L., & Worzel, B. (2003). *Genetic Programming Theory and Practice.* Boston: Kluwer.

Spector, L. (2001). Autoconstructive evolution: Push, Pushgp, and Pushpop. *Proc. Genetic and Evolutionary Computation Conf.*, pp. 137–146.

Spector, L. (2004). *Automatic Quantum Computer Programming: A Genetic Programming Approach.* Boston: Kluwer Academic.

Spector, L., Barnum, H., & Bernstein, H.J. (1998). Genetic programming for quantum computers. *Genetic Programming 1998: Proc. 3rd Annual Conf.*, pp. 365–373.

Spector, L., Barnum, H., Bernstein, H.J., & Swamy, N. (1999*a*). Quantum computing applications of genetic programming. In *Advances in Genetic Programming 3* (Spector, L., Langdon, W.B., O'Reilly, U.-M. & Angeline, P.J., Eds.), pp. 135–160. Cambridge, MA: MIT Press.

Spector, L., Barnum, H., Bernstein, H.J., & Swamy, N. (1999*b*). Finding a better-than-classical quantum AND/OR algorithm using genetic programming. *Proc. Congr. Evolutionary Computation*, pp. 2239–2246.

Spector, L., & Bernstein, H.J. (2002). Communication capacities of some quantum gates, discovered in part through genetic programming. *Proc. Sixth Int. Conf. Quantum Communication, Measurement, and Computing*, pp. 500–503.

Spector, L., & Klein, J. (2005). Trivial geography in genetic programming. In *Genetic Programming Theory and Practice III* (Yu, T., Riolo, R.L., & Worzel, B., Eds.), pp. 109–123. New York: Springer.

Spector, L., Klein, J., & Keijzer, M. (2005). The Push3 execution stack and the evolution of control. *Proc. Genetic and Evolutionary Computation Conf.*, pp. 1689–1696.

Spector, L., Langdon, W.B., O'Reilly, U.-M., & Angeline, P.J. (1999c). *Advances in Genetic Programming 3.* Cambridge, MA: MIT Press.

Spector, L., & Robinson, A. (2002). Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines 3(1)*, 7–40.

Spector, L., & Stoffel, K. (1996). Ontogenetic programming. *Genetic Programming 1996: Proc. 1st Annual Conf.*, pp. 394–399.

Steane, A. (1998). Quantum computing. *Reports on Progress in Physics 61*, 117–173.

Yu, T., Riolo, R.L., & Worzel, B. (2005). *Genetic Programming Theory and Practice III.* New York: Springer.

Wilson, S.W. (1987). The genetic algorithm and biological development. *Proc. 2nd Int. Conf. Genetic Algorithms*, pp. 247–251.

Lee Spector is a Professor of computer science at Hampshire College in Amherst, MA. He received a BA in philosophy from Oberlin College and a PhD from the Department of Computer Science at the University of Maryland. His areas of interest include evolutionary computation, quantum computation, and the intersections between computer science, cognitive science, evolutionary biology, and the arts. He is a member of the ACM Special Interest Group on Evolutionary Computation (SIGEVO) Executive Committee and is a Fellow of the

International Society for Genetic and Evolutionary Computation. He serves on the editorial boards of *Genetic Programming and Evolvable Machines* and *Evolutionary Computation.*

Jon Klein is a Senior Research Fellow at Hampshire College, where he received his BA. He holds a Master's degree in complex adaptive systems from Chalmers University in Gothenburg, Sweden. He developed and maintains the open-source breve simulation environment for multiagent systems and artificial life research (http://www.spiderland.org/breve).