Evolving Quantum Circuits and Programs through Genetic Programming

Paul Massey, John A. Clark, and Susan Stepney

Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK.
{psm111,jac,susan}@cs.york.ac.uk

Abstract. Spector *et al.* have shown [1],[2],[3] that genetic programming can be used to evolve quantum circuits. In this paper, we present new results in this field, introducing probabilistic and deterministic quantum circuits that have not been previously published. We compare our techniques with those of Spector *et al*, and point out some differences in perspective between our two approaches. Finally, we show how, by using sets of functions rather than precise quantum states as fitness cases, our basic technique can be extended to evolve true quantum algorithms.

Keywords: quantum computing, genetic programming

1 Introduction

Quantum computing [4],[5] is a radical new paradigm that has the potential to bring a new class of previously intractable problems within the reach of computer science. Harnessing the phenomena of *superposition* and *entanglement*, a quantum computer can perform certain operations exponentially faster than classical (non-quantum) computers. However, devising algorithms to harness the power of a quantum computer has proved extraordinarily difficult. Eighteen years after the publication of the first quantum algorithm in 1985 [4], almost all known quantum programs are based on two fundamental techniques: Shor's Quantum Fourier Transform [6],[7] and Grover's quantum search algorithm [8].

Spector *et al.* [1],[2],[3] show how genetic programming (GP) might be used to evolve quantum programs and circuits. The key features of their approach are:

- A *second order encoding*: individuals in the GP population are LISP programs that generate quantum circuits. The LISP *alleles* available to the GP software include functions to allow iteration and arithmetic, as well as functions to generate quantum gates.
- An emphasis on finding *probabilistic* solutions: quantum programs give the correct answer with a probability of at least 50% for every fitness case tested.
- A fitness function made up of three components: *hits*, *correctness* and *efficiency.* This function is described later, in the context of our own work.

Spector *et al* evolved a quantum circuit that solves Deutsch's Problem [4] on three qubits, with an error probability of less than 0.3, and a quantum circuit that solves the database search problem on five qubits, for the special case of four marked states (effectively an implementation of a special case of Grover's algorithm [8]).

Our work also involes using GP to evolve quantum circuits, but with a different emphasis from Spector *et al.* We use the following terminology in this paper:

Quantum circuit (or 'quantum gate array'): a collection of quantum logic gates that can be applied in sequence to a specified quantum system.

Quantum program: a set of instructions that, when executed, generates one or more quantum circuits. The program may include constructs such as iteration and branching functions as well as functions to generate particular quantum gates. In the language of GP, a quantum program is the *genotype* that can be decoded to produce a quantum circuit *phenotype*.

Quantum algorithm: a *parameterisable* quantum program that, as the value of the parameter(s) are altered, generates quantum circuits to solve a large number of different problem instances, perhaps across different sizes of quantum system.

Our terminology differs from normal non-quantum usage, where, for example, a (compiled) 'program' is software and executed on real circuitry, and where an 'algorithm' is generally a machine independent recipe that would be implemented by a program. (Both algorithm and program might be capable of handling various system instances). We use a general purpose circuit generating language, abstracting over the implementation of the circuit in hardware, to express our 'programs'. Since we have at present no general purpose quantum computer to target, this seems appropriate. Also, the ability to express parametrisable programs in the same language, allowing abstraction over different system instances, is motivated by our real goal: the use of GP-related methods to discover new ways of solving problems by quantum means.

2 Evolving Quantum Circuits

2.1 The Software

Our research has been conducted using three successive iterations of a tool called *Q-PACE* (*Q*uantum *P*rograms *A*nd *C*ircuits through *E*volution). The original Q-PACE suite is now obsolete. Q-PACE II and Q-PACE III are genetic programming suites written in C++, incorporating a number of classes and functions from Wall's GALib library [11].

Q-PACE II uses a *first order encoding*: each individual is a quantum circuit, not a quantum program (as per the definitions earlier). In practice, each individual is a tree of quantum gates; the tree is traversed to generate the sequence the quantum gates.

Q-PACE III uses a *second order encoding* similar to that of Spector *et al*: each individual is a quantum program that generates one or more actual quantum circuit(s). In practice, each individual is a tree of statements, with each statement being either a function (including functions to allow iteration and functions to generate quantum gates) or a terminal symbol (a number, representing the qubit to be operated on, the number of iterations a loop should run for, etc.)

Both Q-PACE II and Q-PACE III use tournament selection and a "subtree swap" crossover operator. They both incorporate various mutation operators, including subtree insertion, subtree deletion, and subtree replacement. In Q-PACE III, constraints are imposed on the mutation operators to ensure that only syntactically valid quantum programs are produced as GP individuals.

2.2 Fitness Functions

By default, both Q-PACE II and Q-PACE III evaluate the fitness of candidate quantum programs and circuits using the following method:

- Initialisation:
  - Create a set V_I of input state vectors that span the space of all possible inputs. Each member of V_I acts as a *fitness case* for the problem under test.
  - Create a set V_T of *target vectors*, the desired results for each fitness case.
- Evaluation:
  - Apply the candidate individual to each fitness case, to produce a set of *result vectors* V_R.
  - Compare each member of V_R with the corresponding member of V_T. The chosen means of comparison defines the specific *fitness function* for the particular problem under test.

When evolving *deterministic* quantum programs or circuits (those that give the correct answer with probability 1, 100% of the time), we use the sum of the magnitudes of the differences of the probability amplitudes:

$$f = \sum_i ||V_{T_i} - V_{R_i}|| \qquad (1)$$

When trying to evolve *probabilistic* quantum programs or circuits, we use:

$$f = \mathit{hits} + \mathit{correctness} + \mathit{efficiency} \qquad (2)$$

This follows the lead of Spector *et al.* [1],[2],[3]. The *hits* component is the total number of fitness cases used minus the number of fitness cases where the program produces the correct answer with a probability of more than 0.52 (following Spector, chosen to be far enough away from 0.5 to be sure it is not due to rounding errors). The *correctness* component is defined as:

$$\mathit{correctness} = \frac{\sum_{i=1}^{n} \max(0, \mathit{error}_i - 0.48)}{\max(\mathit{hits}, 1)} \qquad (3)$$

Because it is desirable for the fitness function to focus on attaining probabilistically correct answers to all fitness cases, rather than simply improving the probability of success in those fitness cases where it is already good enough (e.g. from a 55% success rate to a 60% success rate), errors smaller than 0.48 are ignored. Also, it is desirable that reasonably fit programs are compared primarily with respect to the number of fitness cases they produce a (probabilistically) correct answer for, and only secondarily with respect to the magnitudes of the errors of the incorrect cases, the 'pure' *correctness* term is divided by *hits* (unless $hits < 1$) before being used in the fitness function.

The *efficiency* is the number of quantum gates in the final solution, divided by a large constant. Therefore, efficiency has a very small effect on the overall fitness of the solution, until programs are evolved that solve all fitness cases, at which point the other two terms become zero and the efficiency dominates. The overall effect is that the search initially concentrates on finding probabilistic solutions to the problem, and then tries to make those solutions more efficient, in terms of the number of quantum gates used. No effort is wasted on trying to make the solutions more accurate (*i.e.* increase the probability of them correctly giving the answer).

2.3 The Evolution of Deterministic Quantum Circuits

Q-PACE II evolved a deterministic full adder circuit using simple and controlled versions of the $N$and H gates (see Appendix A), and the non-unitary zeroing gate Z. Q-PACE II found 2 distinct solutions to the problem; the more efficient of which is identical to that of Gossett [15]. We also applied Q-PACE II to a number of more challenging problems, including multiplication modulo n and exponentiation modulo n. We also attempted to evolve simple circuits for quantum arithmetic using only very basic quantum transformations as alleles. However, despite many variations of problem specification and fitness function, large population sizes, and runs of thousands of generations, we were unable to evolve exact solutions to these problems.

GP is intrinsically suited to finding good *approximate* solutions, as opposed to *exact* deterministic solutions. We modified Q-PACE II to use a two stage search strategy: GP to evolve candidate solutions with a very good, but not perfect, fitness score, then hill-climbing on these good solutions to look for a nearby exact answer. However, deterministic solutions to these harder problems continued to be elusive. We believe that this is due to the discontinuity of the search space: exact solutions lie some distance from good approximate solutions, so changing a single quantum gate in a good solution is of no use in converging to an exact solution. Therefore, we changed the form of the problem to be solved to one we believe more suited to GP.

2.4 Probabilistic Quantum Circuits

Using the probabilistic fitness function(3), Q-PACE II can find probabilistic solutions to problems for which it was unable to find a deterministic solution. It

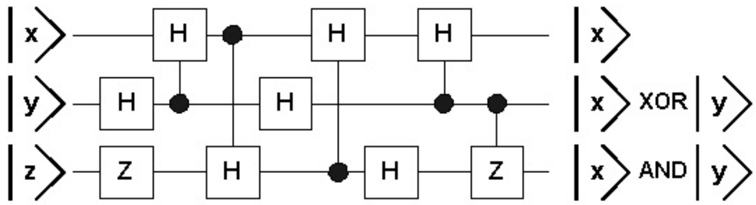

Fig. 1. A probabilistic half-adder

found a probabilistic half-adder on 3 qubits using only the H gate and the non-unitary zeroing gate Z (together with their controlled equivalents). The problem is defined as $|x, y, z\rangle \rightarrow |x, x\,\mathrm{XOR}\,y, x\,\mathrm{AND}\,y\rangle$, where $|x\,\mathrm{XOR}\,y\rangle$ is the sum bit and $|x\,\mathrm{AND}\,y\rangle$ the carry bit.

Q-PACE II evolved the circuit shown in figure 1. It has the following probabilistic solution to the problem:

initial		correct	prob of ending in state							
state		answer	$\|000\rangle$	$\|001\rangle$	$\|010\rangle$	$\|011\rangle$	$\|100\rangle$	$\|101\rangle$	$\|110\rangle$	$\|111\rangle$
$\|000\rangle$	→	$\|000\rangle$	**0.53**	0.22	0	0	0	0.09	0.14	0
$\|001\rangle$	→	$\|000\rangle$	**0.53**	0.22	0	0	0	0.09	0.14	0
$\|010\rangle$	→	$\|010\rangle$	0	0.05	**0.61**	0	0	0.09	0.24	0
$\|011\rangle$	→	$\|010\rangle$	0	0.05	**0.61**	0	0	0.09	0.24	0
$\|100\rangle$	→	$\|110\rangle$	0.09	0.04	0.03	0	0	0.02	**0.82**	0
$\|101\rangle$	→	$\|110\rangle$	0.09	0.04	0.03	0	0	0.02	**0.82**	0
$\|110\rangle$	→	$\|101\rangle$	0	0.30	0.10	0	0.02	**0.53**	0.04	0
$\|111\rangle$	→	$\|101\rangle$	0	0.30	0.10	0	0.02	**0.53**	0.04	0

The most appropriate use of probabilistic circuits remains to be determined. It is possible to exploit probabilistic biases by repeated application of a program to a particular problem instance. We can also imagine their use in more 'soft' systems, where a good deal of noise in results is expected as the computation evolves. This is an open research issue.

3 Evolving Q-MAX Algorithms using 'Functional Fitness Cases'

To evolve more advanced quantum programs (and ultimately quantum algorithms), it has proved necessary to replace the basic technique of calculating the fitness, as described above, with a different approach.

To illustrate this approach, consider that we are given a set of functions on x, where x is an integer variable constrained to a certain domain and range (e.g. $[0..3] \rightarrow [0..3]$), such as the following:

$f_1(x) = 7x \bmod 3$
$f_2(x) =$ the permutation $\{0, 1, 2, 3\} \rightarrow \{2, 1, 3, 0\}$
$f_3(x) = 3 - x$
$f_4(x) = 0$ for all x (i.e. $0 \rightarrow 0, 1 \rightarrow 0, 2 \rightarrow 0, 3 \rightarrow 0$)

For any of these functions, it would be a reasonable challenge to seek a quantum program that finds the *maximum value* for that function (in other words, "which input value x gives the largest output value y"?). For the function $f_3(x)$, for example, the maximum value is 3, which occurs when $x = 0$. A more difficult but much more useful challenge would be to evolve a quantum algorithm for finding the maximum value of *any* $[0..3] \rightarrow [0..3]$ function. Such an algorithm could be given any of the functions listed above and would return the correct maximum value for that function.

3.1 The Specific Case

Before progressing to the general case, first consider how we might solve the following basic problem:

Given a permutation function f(x) which operates over the integer range [0..3]*, use a suitable encoding to evolve a quantum program U that returns the value of x that gives the maximum value of f(x).*

Before we can attempt to evolve a candidate U, we first need to create a quantum state to encode $f(x)$. As both x and $f(x)$ range from [0..3], we can encode $f(x)$ in a 4 qubit quantum state, with the first 2 qubits encoding x and the remaining 2 qubits encoding $f(x)$. For example, the state vector Y (fig 6) encodes the function $f_2(x)$ defined above.

$$Y = \begin{array}{c|c|} x\ f(x) & \\ 00\ 00 & 0 \\ 00\ 01 & 0 \\ 00\ 10 & a \\ 00\ 11 & 0 \\ 01\ 00 & 0 \\ 01\ 01 & b \\ 01\ 10 & 0 \\ 01\ 11 & 0 \\ 10\ 00 & 0 \\ 10\ 01 & 0 \\ 10\ 10 & 0 \\ 10\ 11 & c \\ 11\ 00 & d \\ 11\ 01 & 0 \\ 11\ 10 & 0 \\ 11\ 11 & 0 \end{array} \qquad R = \begin{array}{c|c|} x\ f(x) & \\ 00\ 00 & a \\ 00\ 01 & b \\ 00\ 10 & c \\ 00\ 11 & d \\ 01\ 00 & e \\ 01\ 01 & f \\ 01\ 10 & g \\ 01\ 11 & h \\ 10\ 00 & i \\ 10\ 01 & j \\ 10\ 10 & k \\ 10\ 11 & l \\ 11\ 00 & m \\ 11\ 01 & n \\ 11\ 10 & o \\ 11\ 11 & p \end{array}$$

The non-zero probability amplitudes a, b, c and d represent the desired transitions $f(0)$ =2, $f(1) = 1$, $f(2) = 3$ and $f(3) = 0$ respectively. All other transitions are disallowed in $f_2(x)$, and so are given an associated probability amplitude of zero in the state vector. To be a valid quantum state, $|a|^2 + |b|^2 + |c|^2 + |d|^2 = 1$. For simplicity, we take these amplitudes as equal: $a = b = c = d = \frac{1}{\sqrt{4}} = \frac{1}{2}$

Having created this state vector Y to act as our initial quantum state, we then apply our candidate quantum program U to it, thus creating a state vector $R = UY$. At the end of this computation, we simulate measuring the first two qubits of the quantum system to read out the 'answer', the value of x that corresponds to the maximum value of $f(x)$. In the general case, we end up with a state vector R shown in fig 6.

Observing the full state causes it to collapse to one of the 16 eigenstates. We see that the first two qubits are observed with particular values with probability equal to the sum of probabilities of the eigenstates consistent with the values. Thus, a state with $x = 00$ is observed with probability $q^2 = |a|^2+|b|^2+|c|^2+|d|^2$; a state with $x = 01$, with probability $r^2 = |e|^2 + |f|^2 + |g|^2 + |h|^2$; a state with $x = 10$, with probability $s^2 = |i|^2 + |j|^2 + |k|^2 + |l|^2$; and a state with $x = 11$, with probability $t^2 = |m|^2 + |n|^2 + |o|^2 + |p|^2$.

We then compute the probabilities $|q|^2$, $|r|^2$, $|s|^2$ and $|t|^2$, which represent the probability of measuring x to be 0, 1, 2 and 3 respectively. Suppose, for one particular candidate U, these probabilities are found to be 0.2, 0.3, 0.5 and 0. By looking at these probabilities, we can see that this candidate U gives the correct answer ($x = 2$) 50% of the time. Our goal is to find a candidate program U which somehow amplifies the probability of measuring the correct value of x, and decreases the probability of measuring an incorrect value of x.

Given a state vector R, how should we actually assess the fitness of a candidate U? The most obvious fitness measure is simply the probability of the correct answer, in this case, $|s|^2$ (in practice, using $1-|s|^2$ would be more useful, as a fitness of 0 would then correspond to an exact solution). A slightly more advanced fitness measure would allow U to be a probabilistic program (i.e. a U that gave $|s|^2 > 0.5$ would be regarded as a solution to the problem under test). Alternative fitness functions could be used that gave some credit for having "near miss" answers (if we are trying to evolve a MAX algorithm, a candidate solution which could consistently return a value of x that gives a large but not maximal value of $f(x)$ might actually be sufficient).

3.2 The General Case

So far we have shown only how we can evolve a quantum program U to solve one specific problem instance. How might we generalise this to evolve a true quantum MAX algorithm, capable of returning the value of x that gives the maximum value of $f(x)$ for any function $f(x)$ that is supplied as input?

To achieve this goal, we create a large number of functions to act as fitness cases. Each candidate U operates on all fitness cases, and its overall fitness is the sum of the fitness scores it is awarded for each f.

3.3 Evolution of a Probabilistic MAX Algorithm for Permutation Functions

Using this strategy, we have been able to evolve (using Q-PACE III) a number of probabilistic quantum programs which, when given a number of suitably encoded $[0..3] \rightarrow [0..3]$ permutation functions, returned for every one of these permutation functions (with a probability > 0.5) the value of x that gave the maximum value of $f(x)$ for that function. We refer to this problem as the "PF MAX" problem for short. Ultimately, we evolved a program that solved the problem for all 24 possible $[0..3] \rightarrow [0..3]$ permutation functions, as is shown below.

We consider only $[0..3] \rightarrow [0..3]$ permutation functions because they have the nice properties of being discrete, one-to-one, easy to generate, and capable of being encoded by a 4 qubit system.

There are $4! = 24$ different $[0..3] \rightarrow [0..3]$ permutation functions. Our default approach was to give Q-PACE III a subset of these 24 functions to act as fitness cases. If Q-PACE III evolved a MAX program that worked for all the fitness cases, we would then test it on the other functions in the set to determine the generality of the evolved solution.

3.4 The "PF MAX 1" Program

The first useful program generated by Q-PACE III (which we call PF MAX 1) was evolved using the following 8 fitness cases (expressed as permutations): {(3,1,0,2), (0,2,3,1), (3,0,1,2), (1,2,3,0), (3,2,0,1), (2,3,0,1), (2,0,1,3), (2,1,3,0)}. PF MAX 1 does the following:

fitness	correct	prob of ending in state			
case	answer	$\|00\rangle$	$\|01\rangle$	$\|10\rangle$	$\|11\rangle$
$(3, 1, 0, 2)$	$\|00\rangle$	**0.53**	0.22	0.03	0.22
$(0, 2, 3, 1)$	$\|10\rangle$	0.03	0.22	**0.53**	0.22
$(3, 0, 1, 2)$	$\|00\rangle$	**0.53**	0.22	0.03	0.22
$(1, 2, 3, 0)$	$\|10\rangle$	0.03	0.22	**0.53**	0.22
$(3, 2, 0, 1)$	$\|00\rangle$	**0.53**	0.22	0.03	0.22
$(2, 3, 0, 1)$	$\|01\rangle$	0.22	**0.53**	0.22	0.03
$(2, 0, 1, 3)$	$\|11\rangle$	0.22	0.03	0.22	**0.53**
$(2, 1, 3, 0)$	$\|10\rangle$	0.03	0.22	**0.56**	0.19

PF MAX 1 is a probabilistic solution to all 8 of the fitness cases used; what effect does PF MAX 1 have on the 16 other permutation functions in the set? For 20 out of the 24 possible permutation functions, PF MAX 1 gives the correct answer with a probability of more than 0.5; for the other 4 fitness cases, PF MAX 1 gives the correct answer with a higher probability than any given incorrect answer. Thus PF MAX 1 is a true MAX algorithm for $[0..3] \rightarrow [0..3]$ permutation functions, that "works" on all 24 of these functions. Although evolved from only 8 fitness cases, the resulting PF MAX 1 is much more general.

The circuit generated by PF MAX 1 (after removing by hand 5 gates that have no effect) is shown in figure 2.

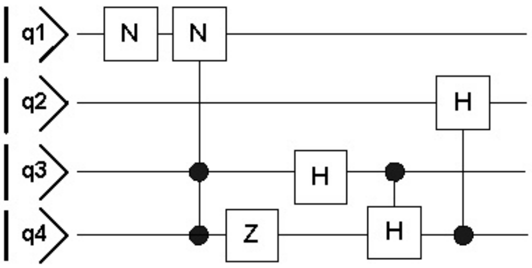

Fig. 2. PF MAX 1 circuit

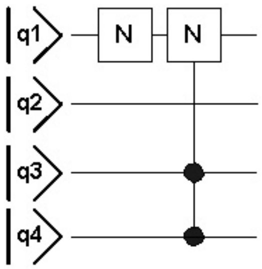

Fig. 3. PF MAX 3 circuit

3.5 Changing the Acceptance Criterion: "PF MAX 3"

The existence of PF MAX 1 suggests a quantum program might exist that would give the correct solution with a probability of > 0.5 for all 24 fitness cases. However, repeated experiments failed to evolve a program that met these parameters. Yet circuits that give the correct answer to certain fitness cases with a probability of *exactly* 0.5 were commonly produced.

So we attempted to solve the PF MAX problem for all 24 possible fitness cases with a relaxed acceptance criterion of > 0.4. When Q-PACE III was run with this relaxed acceptance criterion, it evolved a quantum program which generated a single quantum circuit that, for each of the 24 fitness cases, has a probability of 0.5 of returning the correct answer (the probabilities of returning incorrect answers are 0.25 or zero). So the quantum circuit implements a probabilistic MAX function that has twice the probability of "guessing".

The circuit generated by PF MAX 3 (after removing by hand several gates that have no effect) is shown in figure 3.

This seems remarkably simple. What is happening here? After consideration we can see that the system is actually exploiting the initial set-up very efficiently. Suppose for example, that the maximum occurs at $x = 00$. Then $|0011\rangle$ has amplitude $\frac{1}{2}$ (corresponding to probability $\frac{1}{4}$), and $|0000\rangle$, $|0001\rangle$ and $|0010\rangle$ all have amplitude of 0. Now consider $x = 10$. We must have $f(10) = 00$, $f(10) = 01$, or $f(10) = 10$ since the maximum is already reached uniquely by $f(00) = 11$. Suppose $f(10) = 00$. Then the state $|1000\rangle$ has amplitude $\frac{1}{2}$, while $|1001\rangle$, $|1010\rangle$ and $|1011\rangle$ all have amplitudes of 0. The application of the CCNOT operation transforms $|1000\rangle$ to $|0000\rangle$ with amplitude $\frac{1}{2}$ whilst $|0011\rangle$ remains unaltered with amplitude $\frac{1}{2}$. We now have two eigenstates with $x = 00$ and amplitude $\frac{1}{2}$: $|0000\rangle$ and $|0011\rangle$. So the probability of now observing one of these eigenstates is $\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$. This is a better than classical algorithm. More generally, if $f(x) = 11$ then we can consider the states $|x\ 11\rangle$ and $|x'\ f(x')\rangle$ (where x' is obtained from x by flipping the first bit) to obtain a similar result. Furthermore, there would appear to be an obvious generalisation to n qubits: let the second negation on qubit 1 be controlled by all the qubits of $f(x)$.

4 Conclusions

The programs and circuits presented in this paper have become increasingly capable, with PF MAX 3 in particular solving quite a general problem, with its better-than-classical result generalising to n qubits. Using GP to produce probabilistic circuits, that give the right answer with high probability for all fitness cases, seems a more practical approach than requiring fully correct deterministic circuits.

In future work, we will build on our results to evolve parameterisable quantum programs, which, when given different parameters, will be able to solve problems such as the PF MAX problem for different classes of function and across different system sizes. If such things can be evolved, they will prove that genetic programming can have a part to play in discovering new, useful quantum algorithms, and help break the bottleneck that currently exists in quantum algorithm discovery.

References

Citations "*quant-ph/yymmxxx*" are available on the Internet from the Los Alamos National Laboratory pre-print server at http://www.arXiv.org

[1] L. Spector, H. Barnum, H. Bernstein, "Genetic Programming for Quantum Computers", in *Genetic Programming 1998*, Morgan Kaufmann, 1998.
[2] L. Spector, H. Barnum, H. Bernstein, N. Swamy, "Quantum Computing Applications of Genetic Programming", in *Advances in Genetic Programming 3*, MIT Press, 1999
[3] L. Spector, H. Barnum, H. Bernstein, N. Swamy, "Finding a Better-than-Classical Quantum AND/OR Algorithm using Genetic Programming", in *Congress on Evolutionary Computation*, 1999.
[4] D. Deutsch, "Quantum Theory, the Church-Turing Thesis, and the Universal Quantum Computer", *Proc. Royal Society of London*, series A, vol. 400, p97, 1985.

[5] E. Rieffel and W. Polak, “An Introduction to Quantum Computing for non-Physicists”, 1998. *quant-ph/9809016.*

[6] P. W. Shor, “Algorithms for Quantum Computation : Discrete Logarithms and Factoring”, *Proc. 35th IEEE Symposium on the Foundations of Computer Science*, p124, 1994.

[7] P. W. Shor, “Polynomial Time Algorithms for Prime-Factorisation and Discrete Logarithms on a Quantum Computer”, SIAM Journal of Computing, **26**, p1484, 1997

[8] L. Grover, “A Fast Quantum Mechanical Algorithm for Database Search”, *Proceedings of the 28th ACM STOC*, p212, 1996.

[9] J. R. Koza, *Genetic Programming*, MIT Press, 1992.

[10] J. R. Koza, *Genetic Programming II*, MIT Press, 1994.

[11] M. Wall, “GALib, a C++ Library for Genetic Algorithms”, available from http://lancet.mit.edu/ga/

[12] A. Ekert, P. Hayden & H. Inamori, “Basic Concepts in Quantum Computation”, 2000. *quant-ph/0011013.*

[13] T. Toffoli, “Reversible Computing”, in *Automata, Languages and Programming*, LNCS 84, Springer, 1980.

[14] E. Fredkin & T. Toffoli, “Conservative Logic”, *Intl. J. Theoretical Phys*, **21**, p219, 1982.

[15] P. Gossett, “Quantum Carry-Save Arithmetic”, 1998. *quant-ph/9808061.*

A Quantum Gates used

In this paper, we use only three basic types of quantum gate:

symbol	name	specification
$N(x)$	NOT gate	$\|0\rangle \to \|1\rangle, \|1\rangle \to \|0\rangle$
$H(x)$	Hadamard gate	$\|0\rangle \to \frac{1}{\sqrt{2}}(\|0\rangle + \|1\rangle)$ $\|1\rangle \to \frac{1}{\sqrt{2}}(\|0\rangle - \|1\rangle)$
$Z(x)$	Non-unitary Zeroing gate	$\|0\rangle \to \sqrt(\|\|0\rangle\|^2 + \|\|1\rangle\|^2)$ $\|1\rangle \to 0$

These are all single-qubit gates, and take a single parameter x representing the target qubit (e.g. $N(2)$ would represent a NOT gate being applied to the second qubit in a system). However, all three gates can have *controlled* equivalents. Thus, the controlled not applies N to a target qubit only if the control qubit is set. Thus, with the first qubit being the control and the second qubit being the target $|00\rangle$ and $|01\rangle$ remain unchanged but $|10\rangle$ and $|11\rangle$ are transformed to $|11\rangle$ and $|10\rangle$ respectively. This is typically written C_{NOT} (or C_N). C_H and C_Z gates are similar: for example, the gate $C_H(2,4)$ would be a controlled Hadamard gate with the second qubit in a quantum system acting as the control qubit and the fourth qubit acting as the target qubit.

Sometimes, it is useful to use gates that have more than one control bit. These gates only perform their action if all control bits are in state $|1\rangle$, otherwise they

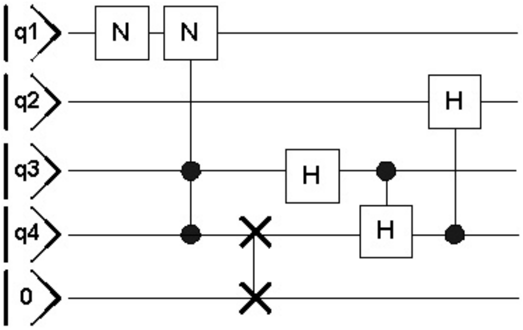

Fig. 4. A circuit equivalent to that of figure 2, with the zeroing gate replaced by an ancilla qubit and with x—x representing a "swap" gate

have no effect. For example, the "controlled controlled NOT" gate $CCN(1, 2, 3)$ negates its target qubit (qubit 3) if both control qubits (1 and 2) are in state $|1\rangle$. Controlled controlled NOT gates are usually called *Toffoli gates*, after their inventor [13],[14].

B Comments on the use of the Zeroing Gate

Here we comment on the use of the non-unitary zeroing gate Z in the various circuits in this paper. The GP suite was given the Z gate to allow qubits to be initialised to the state $|0\rangle$, as occurs in many papers in the literature. However, the GP suite generalised beyond this and made use of the Z gate in unexpected places, such as in mid-circuit (figure 2).

What does it *mean* to force a qubit to zero in the middle (or at the end) of a circuit? The Z gate acts rather like a standard measurement gate; the key difference is that a measurement gate collapses the quantum state vector into one of its component quantum states at random, whereas the Z gate forces the qubit into the state $|0\rangle$. However, both types of gates have the same side-effect: reducing the number of possible positions in the state vector with a non-zero probability amplitude.

How to implement Z? One way would be to perform the measurement, and proceed only if it were zero. This would reduce the probabilities of correct answers unacceptably. An alternative way is to regard it as a swap operation with an ancilla zeroed qubit. So the circuit illustrated in figure 2 is equivalent to the circuit shown in figure 4.