Evolution of a Human-Competitive Quantum Fourier Transform Algorithm Using Genetic Programming

Paul Massey John A. Clark Susan Stepney
Department of Computer Science, University of York, Heslington, York, UK, YO10 5DD
{psm111 | jac | susan}@cs.york.ac.uk

ABSTRACT

In this paper, we show how genetic programming (GP) can be used to evolve system-size-independent quantum algorithms, and present a human-competitive Quantum Fourier Transform (QFT) algorithm evolved by GP.

Categories and Subject Descriptors

D.1.m [**Programming Techniques**]: Miscellaneous,
J.2 [**Physical Sciences and Engineering**]: Physics.

General Terms

Algorithms, Experimentation.

Keywords

Genetic Programming, Genetic Algorithms, Evolutionary Computing, Quantum Computing, Quantum Fourier Transform.

1. INTRODUCTION

Quantum Computing [4],[12] is a radical new paradigm that has the potential to bring a new class of previously intractable problems within computational reach. Harnessing the phenomena of superposition and entanglement, a quantum computer can perform certain operations more efficiently than classical (non-quantum) computers. The earliest example of a 'faster than classical' quantum algorithm was Deutsch's quantum solution to the binary promise algorithm. Here a single quantum evaluation suffices to reveal whether a binary function f is constant ($f(0) = f(1) = 0$ or $f(0) = f(1) = 1$) or balanced ($f(0) = 0$, $f(1) = 1$ or $f(0) = 1$, $f(1) = 0$). (This can be extended to n-input binary functions.) Various other faster than classical algorithms followed, but real excitement was generated in 1994 by Peter Shor with a specific application of the Quantum discrete Fourier Transform.

The Quantum Fourier Transform (QFT) is perhaps the most important building block in the quantum algorithm designer's armoury. It has a variety of applications (Chapter 5 of [10] gives a variety of specific solvable instances of the hidden subgroup problem such as Deutsch's problem, Simon's problem, period finding, order finding, hidden linear function finding), but the most important application is undoubtedly its use by Shor to provide a polynomial time quantum algorithm for factorisation of composite integers and the calculation of discrete logarithms in a finite field [14][15]. Some of the best-known and widely respected encryption algorithms in the world rely on these problems being computationally intractable. Shor had provided what is regarded by most as the 'killer application' for quantum computing. The field began to attract huge interest.

One might imagine that there would be a flood of new algorithms to harness the power of this rapidly emerging means of computation. However, this has not been the case. It is generally agreed that there are still very few distinct quantum algorithms (see [10]). This motivates our investigation of genetic programming in the quantum algorithm field. Genetic programming has discovered new artefacts in other domains. Indeed, its use has produced various patentable outputs. Can it exhibit human-competitive performance for quantum algorithm design?

In this paper we show how GP has been used to evolve a human competitive algorithm for the Quantum Fourier Transform (QFT). We show how circuits can be evolved using GP that implement the QFT for 1, 2, and 3 qubits. This is, however, the prelude to the main result of this paper: the evolution of an *algorithm* for the QFT, which when executed with specific system size (i.e. number of qubits) generates a circuit that implements the corresponding QFT. We believe this is the most significant quantum artefact yet evolved using evolutionary computing. It would appear to compete (in this instance) with the efforts of professional quantum specialists.

The power of the result comes from its generality. The drive to ever-increasing levels of abstraction goes hand in hand with increases in design sophistication in many domains (most notably software engineering). The need to handle things at a higher level is recognised by quantum specialists. It informs the evolutionary frameworks in the pioneering work of Spector and co-researchers (see below), from which we freely draw inspiration.

In Section 3 we detail the software framework we have used to evolve quantum artefacts, indicating how solutions are represented and manipulated. In Section 4 we provide details of the various fitness functions used. In Section 5 we provide details of the QFT and known implementations. In section 6, we provide some of the circuits we have evolved together with the system size independent algorithm for generating QFT circuits. Section 7 concludes. First, we review current applications of meta-heuristic search to the design and exploration of quantum artefacts.

2. SEARCH FOR QUANTUM ARTEFACTS

The design of quantum artefacts has emerged as a promising application area for evolutionary computing and other heuristic search techniques. Here we provide an outline of work in the field. Rather than adopt a chronological account, we present results according to the approximate level of abstraction of the quantum artefacts that were sought.

2.1 Low level applications

Most published circuits or algorithms make use of some particular set of quantum gates. Examples of such gates are shown in Table 1. One widely used two–qubit gate is CNOT (or Feynmann) gate. This is often considered 'basic' by researchers, but this may be misleading. Gershenfeld and Chuang [5] show how the CNOT gate can be implemented on Nuclear Magnetic Resonance implementations of quantum computing by a series of five more primitive operations. Rethinam *et al* [12] use a basic genetic algorithm (bit string representation with single point crossover) to evolve sequences of length 3, more efficient than previously exhibited solutions. Perkowski *et al* have synthesised 'basic' gates such as Fredkin and Toffoli gates from lower level gates.

Much quantum circuit design does not take into account physical implementation constraints; for example, it may be possible to carry out CNOT operations only on qubits in very close proximity. There arise issues as to how problem variables map to physical qubits and how computational administrative costs (such as incurred when repeated qubit variable value swaps are used to bring the required values for an operation adjacent) can be minimized. Van Meter *et al* [22] outline work in progress seeking to optimize the use of quantum resources.

2.2 Specific circuits

The use of genetic programming for the evolution of quantum artefacts has been pioneered by Spector and co-researchers. Early work attempted to generate circuits to solve instances of OR, AND-OR, Deutsch Josza promise and database search problems [1][2][16][17][18]. The reader is referred to [19] for a summary and up-to-date discussion of the work. Of particular importance is working with 'second order encodings'; rather than evolve circuits directly the GP search evolves *programs* that when executed generate circuits. We too exploited this approach in [11]. The notion of second order encodings is also exploited in this paper; it is a major tool in the drive to increasing levels of abstraction in the evolution of quantum artefacts. Various researchers have built on the work of Spector and co-researchers. For example, Leier & Banzhaf have used a linear tree GP variant to evolve solutions to the 1-sat problem (Hogg's algorithm) [7] and Massey *et al* investigate the use of alternative cost functions [11].

2.3 Communications

One of the most intriguing applications of quantum phenomena is that of quantum teleportation [3]. Quantum teleportation aroused a great deal of interest and it is not surprising that various researchers have targeted the design of teleportation protocols. Williams & Gray use a genetic programming approach to evolve implementations of the subcircuits implementing each of the phases [24]. Subsequently Rubenstein has evolved teleportation sub-circuits. Yakubi & Iba criticise Williams & Gray's work stating that the approach allowed infeasible protocols to be evolved and chose to evolve a whole protocol [25] but with a structure constrained by knowing the traditional BBS protocol. Spector *et al* have also evolved a teleportation protocol with the PUSH base system (see Spector's book [19] for details).

More recently Spector & Bernstein have used genetic programming to discover the communications capabilities of quantum circuitry [20]. This has included disproving conjectures on communications capacities. It would appear that uncovering genuine insights in this field is computationally tractable by evolutionary computation and the area seems highly promising. Of further interest is that protocols and circuits uncovered by evolutionary computing were generalised by intelligent reflection. (The work could adequately be described as inspirational.) A fuller account can be found in [19].

2.4 Summary and next step

Evolutionary computation techniques have found successful application to the derivation of quantum artifacts at many levels, ranging from the evolution of implementations of 'basic' gates to the evolution of circuits for teleportation. (A fuller review can be found in [21].) The evolution of true parametrisable algorithms seems a natural goal to set ourselves. We now describe the approach that has been used to do this for one such artifact – the Quantum Fourier Transform.

3. THE GP SOFTWARE

Our research has been conducted using successive versions of a software suite called Q-PACE (**Q**uantum **P**rograms **A**nd **C**ircuits through **E**volution). The quantum program presented in Section 6.1 has been evolved using Q-PACE III, the quantum algorithms presented in Sections 6.2 and 6.3 have been evolved using Q-PACE IV. We now describe Q-PACE's key characteristics.

Table 1 - Gates Recognised / Generated by Q-PACE III & IV

Name	*Symbol*	*Effect*
NOT	N(x)	$\begin{matrix}\lvert 0> \\ \lvert 1>\end{matrix}\begin{pmatrix}a\\b\end{pmatrix}\to\begin{pmatrix}b\\a\end{pmatrix}$
Hadamard	H(x)	$\begin{matrix}\lvert 0> \\ \lvert 1>\end{matrix}\begin{pmatrix}a\\b\end{pmatrix}\to\frac{1}{\sqrt{2}}\begin{pmatrix}a+b\\a-b\end{pmatrix}$
Berry Phase	P(x, θ)	$\begin{matrix}\lvert 0> \\ \lvert 1>\end{matrix}\begin{pmatrix}a\\b\end{pmatrix}\to\begin{pmatrix}a\\e^{2i\theta}b\end{pmatrix}$
SWAP	SWAP(*x*, y)	$\begin{matrix}\lvert 00> \\ \lvert 01> \\ \lvert 10> \\ \lvert 11>\end{matrix}\begin{pmatrix}a\\b\\c\\d\end{pmatrix}\longrightarrow\begin{pmatrix}a\\c\\b\\d\end{pmatrix}$
Q-PACE III & IV also recognise *controlled* versions of each of the above gates. For example, the Controlled Hadamard gate H(c,x) has the same effect as the Hadamard gate H(x) if the value of qubit c != 0, otherwise it has no effect. In the case of the NOT gate, a *double-controlled* version CCN(c1,c2,x) is also recognised (the gate N(x) is produced only if neither qubit c1 nor qubit c2 have a value of zero).		

Language and GP Engine. Q-PACE is written in C++, with GP engines based on Wall's GALib library [23].

Representation. Q-PACE uses a second order representation: individuals are not quantum circuits, but higher level constructs that need to be decoded/executed to generate quantum circuits. In Q-PACE III, individuals are programs that, when decoded, generate a single quantum circuit (appropriate to a single size of quantum system). In Q-PACE IV, individuals are pseudo-code algorithms which, when decoded/executed, produce a family of quantum circuits (one for each size of quantum system under test).

Quantum Gate Set. The individuals generated by Q-PACE III and IV, when decoded/executed, create quantum circuits built from the set shown in Table 1. A user of the software is able to constrain the GP to use any subset of these allowed gates by selecting (through an on-screen prompt at the start of a GP run) which gate-generating functions should be used in the allele set.

Allele Set. Individuals are trees of alleles, where each allele is either a *function* or a *terminal*. In Q-PACE III, functions generate quantum gates, and terminals are constants (denoting which qubit(s) should be operated on). In Q-PACE IV, functions may be *gate-generating functions*, *arithmetic functions* or *control functions*; terminals may be *constants* or *variables*. Formal definitions of all the gate-generating functions used in Q-PACE IV are presented in Table 2; formal definitions of all other alleles used in Q-PACE IV are presented in Table 3. In Q-PACE III, different functions take different numbers of parameters. In Q-PACE IV, all functions take the same number of parameters (three), to allow more general mutation operators; when a function requires only one or two parameters, the remaining parameters have no effect. The user is able to constrain the GP to use any subset of the allele set through an on-screen prompt at the start of a GP run. A linked list object is used to store the quantum circuit produced when the individual is decoded for a given system size.

GP Operators. Q-PACE uses a tournament selection operator and a subtree-swap crossover operator. A range of mutation operators are available. The default Q-PACE III mutation operator allows terminals to mutate into other terminals, and functions to mutate into other functions of the same cardinality. The default Q-PACE IV mutation operator performs one of three different types of mutation, known as "mini", "midi" and "maxi" replacement, depending on the result of a biased coin flip. In mini-replace, an allele is mutated for another allele of the same type (e.g. a constant can only mutate into another constant, a gate-producing function can only mutate into another gate-producing function, etc), and any children of the original allele are unchanged. In midi-replace, a terminal can mutate into any other terminal (e.g. a variable can become a constant, and vice versa), and a function can mutate into any other function (e.g. a gate producing function can become an arithmetic function, and vice versa), with any children of the original node left unchanged. In maxi-replace, an allele can mutate into any other node. If the original allele has children, they are destroyed and rebuilt at random (this capability allows quite extensive mutations).

Allele selection for initial population. While *current_tree_depth* < *max_tree_depth*, an allele is a randomly-selected function with some probability *function_probability*, and a randomly-selected terminal with probability (1 – *function_probability*). When *current_tree_depth* reaches *max_tree_depth*, all new alleles are randomly selected terminals.

Stopping Criteria. Evolution continues until either (a) an exact solution to the problem under test is found (in which case it is displayed), or (b) a user-defined number of generations elapse (in which case the best result so far is displayed).

Table 2 – Gate-generating functions used in Q-PACE IV

Name	*Return value*	*Side effects*
Create_N(x, -, -)	x	create N(x)

Create_CN (c, x, -)	x	if c ≠ x, create CN(c, x)
		if c = x, create N(x)

Create_CCN (c1,c2,x)	x	if c1=c2=x, create N(x)
		if c1=c2≠x, create CN(c1, x)
		if c1≠c2≠x, create CCN(c1,c2,x)
		if c1=x≠c2, create CN(c2, x)
		if c1≠c2=x, create CN(c1, x)

Create_H(x, -, -)	x	create H(x)

Create_CH (c, x, -)	x	if c ≠ x, create CH(c, x)
		if c = x, create H(x)

Create_P(x, θ, -)	x	create P(x, π/2θ)

Create_CP (c, x, θ)	x	if c ≠ x, create CP(c, x, π/2θ)
		if c = x, create P(x, π/2θ)

Create_SWAP (x, y, -)	x	if x ≠ y, create SWAP(x,y), otherwise, no effect

Table 3 – Other alleles used by Q-PACE IV

Name	*Return value*	*Comments*
PLUS(x, y, -)	x + y	
MINUS(x, y, -)	x – y	
MULTIPLY (x, y, -)	x * y	
DIVIDE(x, y, -)	x / y	return int(x / y) if y ≠ 0, 1 otherwise
ITERATE (n, BODY, -)	n	the second child of an ITERATE is always a BODY (enforced during crossover and mutation)
BODY (ch1, ch2, ch3)	ch1	
ROOT (ch1, ch2, …)	ch1	all individuals are rooted in this function; it can appear nowhere else in an individual
plus **Constants** (1.. *current_system_size*) and **Variables** (*current_system_size*, and loop counters for any ITERATE statements currently in scope)		

Notes for both Table 2 and Table 3:

1. If the value of a (decoded) parameter is < 1, it is coerced to 1; if the value is > current_system_size, it is coerced to current_system_size.
2. Parameters denoted "-" have no effect on the result. They give all functions the same arity, to allow more general mutation operators.

4. FITNESS FUNCTIONS

The state of any n qubit quantum system can be represented by a *state vector* of 2^n complex numbers. In order to determine the fitness of an evolved individual, Q-PACE compares the state vectors generated by applying that individual to a set of known initial states, with those produced by applying a known model solution to the same set of known initial states. More specifically, the technique for assessing fitness is as follows:

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, by applying a model solution to the set V_I .

Evaluation:

- Apply each 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.

For the results presented in this paper, two fitness functions are used. The first, Figure 1, gives credit only for exact matches in the various fitness functions. The second, Figure 2, sums the differences between corresponding state vector positions in the V_T and V_R vectors. Note that for the purposes of this fitness function, V_T and V_R are required to be in polar co-ordinate form, i.e. $\left|V_{T_i}\right| = (r_{T_i}, \theta_{T_i})$ and $\left|V_{R_i}\right| = (r_{R_i}, \theta_{R_i})$. This allows the fitness function to use a scaling factor α, the purpose of which is to ensure that individuals where the magnitude of the complex numbers match (but the phases do not) have a considerably better fitness than individuals where the phases match but the magnitudes do not. The fitness function is designed this way to promote a particular evolutionary strategy: to allow the GP software to first evolve solutions which are basically correct but with incorrect angles in any phase gates (*e.g.* CP(2,1,π/8) instead of CP(2,1,π/4)), before subsequently evolving the correct angles. We have found this strategy, by and large, works well for solving problems where quantum phase operations are an integral part of the solution.

$$f = \sum_i (\text{if } \left|V_{T_i}\right|^2 = \left|V_{R_i}\right|^2 \text{ then 0 else 1})$$

Figure 1 - "Match State Vector Positions" Fitness Function

$$f = \alpha \sum_i \left|r_{T_i} - r_{R_i}\right| + \sum_i \left|\theta_{T_i} - \theta_{R_i}\right|$$

Figure 2 - Polar co-ordinate difference fitness function

In addition to the functional fitness given above, the fitness function also contained an efficiency component defined by

$$efficiency = \sum_{system\_size\_tested} \left(sz - \text{target\_size}\right)/100$$

Figure 3. Efficiency component of fitness function

where *sz* is the size of the quantum circuit generated by the individual for the current system size and the *target_size* is defined as 2 for a system of size 1, 6 for a system of size 2, 10 for a system of size 3 and 16 for a system of size 4. (These values are a little greater than the most efficient sizes known.)

An additional component was introduced to penalise the absence of appropriate SWAP gates in any position for the system size under consideration. In this respect we have given the technique a small piece of system specific help. The presence or absence of a SWAP gate wildly changes the *r*-theta difference of an individual, so much so that hundreds or thousands of generations spent working towards the delicate (and very tricky) CREATE_CP loops could be undone in a single generation by a mutation that introduced the right SWAP gate (but destroyed everything else of value in the algorithm).

5. THE QUANTUM FOURIER TRANSFORM

5.1 Definition

Consider a quantum state vector (x_0, x_1, … x_{N-1} }, where N = 2^n. Applying the QFT to this state vector gives us a result state vector (y_0, y_1, … y_{N-1} } such that y_k is equal to the right hand side of the equation in Figure 4.

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \exp \frac{2\pi i\, jk}{N}$$

Figure 4

5.2 Implementation

The following pseudo-code algorithm, *QFT(n)*, implements the QFT on any size of quantum system using only the quantum gates shown in Table 1:

```
For (j = 1; j < n; j++) {
        Create_H(j);
        For (k=1; k <= (n–j); k++) {
                Create_CP(j+k, j, k+1); } }
Create_H(n);
For (i = 1; i <= (n / 2); i++) {
        Create_SWAP(i, n – i + 1); }
```

The functions `Create_H(x)`, `Create_CP(x,y,z)` and `Create_SWAP(x,y)` are defined in Table 2.

Each of these circuits implements an exact QFT for that system size. Table 4 shows the circuits produced by this algorithm for quantum systems of 1 - 4 qubits.

Table 4 - Circuits to implement the QFT for n = 1-4 qubits

n	*Circuit*
1	H(1)
2	H(1), CP(2,1,π/4), H(2), SWAP(1,2)
3	H(1), CP(2,1, π/4), CP(3,1, π/8), H(2), CP(3,2, π/4), H(3), SWAP(1,3)
4	H(1), CP(2,1, π/4), CP(3,1, π/8), CP(4,1, π/16), H(2), CP(3,2, π/4), CP(4,2, π/8), H(3), CP(4,3, π/4), H(4), SWAP(1,4), SWAP(2,3)

6. RESULTS

6.1 An evolved program to implement the QFT on a 3-qubit system

Q-PACE III is able to evolve programs which, when decoded and executed, implement an exact QFT on a 3 qubit quantum system.

Using the fitness function shown in Figure 1, a population size of 100, a crossover probability of 0.5, and a mutation probability of 0.01, Q-PACE III is able to generate (in 1122 generations) the following individual:

```
ROOT(
 2,
 Create_CP( 3, Create_CH(1,1), Create_CP(1,2,2) ),
 2,
 Create_H(2),
 Create_CP(2,3,2),
 Create_CCN(
  Create_CP(
   Create_CCN( Create_CP(3,1,3), Create_N(1), 1 ),
   3, 2 ),
  Create_H(3), 1 ),
 Create_CN(1,3),
 2, 3 )
```

This individual, when decoded/executed, produces the quantum circuit illustrated in Figure 5. After hand-optimisation, the circuit can be simplified to that illustrated in Figure 6.

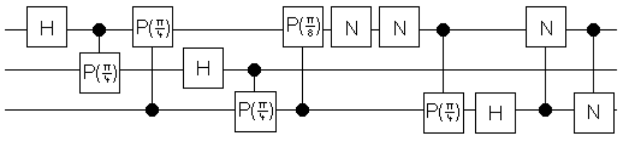

Figure 5. (Unoptimised) circuit generated by the evolved solution to QFT(3)

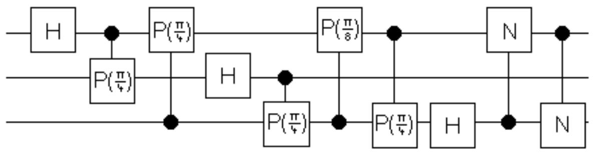

Figure 6. (Hand-Optimised) circuit generated by the evolved solution to QFT(3)

This circuit has 10 gates. Although the best known circuit to generate QFT(3) can be implemented in 7 gates (see Table 4), that circuit requires the use of a SWAP gate, which was not available as an allele to Q-PACE III in this particular GP run. The most efficient known circuit to implement QFT(3) using the alleles given to Q-PACE III in this GP run has 9 gates, just one less than the solution evolved here.

6.2 An evolved algorithm to implement the QFT on system sizes of 1 to 3 qubits

Q-PACE IV is able to evolve algorithms which, when decoded and executed, implement an exact QFT on system sizes of 1, 2 and 3 qubits. One is presented here. To evolve this algorithm, the GP used the fitness function shown in Figure 2 (together with a small efficiency component to minimise GP bloat), a population size of 2000 for the first two generations, and 50 thereafter (to ensure a "deep gene pool" at the beginning of the evolutionary process), a crossover probability of 0.75, and a mutation probability of 0.075. With these parameters, Q-PACE IV is able to generate (in 2177 generations) the following individual:

```
ROOT(
 ITERATE(
  MINUS(n, 1, n),
  BODY(
   Create_H(var1, n, n),
   ITERATE(
    MINUS(n, var1, n),
    BODY(
     Create_CP(
      PLUS(var1,var2,n),var1,PLUS(1,var2,var2)
     ),
     var1, 2),
    var1),
   n),
  n)
 Create_H(n, n, n),
 ITERATE(
  DIVIDE(n, n, n),
  BODY(Create_SWAP(var1,n,2), DIVIDE(n,n,n), n),
  n)
 )
```

This individual, when decoded/executed for different system sizes *n*:, produces the quantum circuits shown in Table 5.

Table 5. Circuits produced by the first evolved algorithm, for *n* = 1-4 qubits

n	*Circuit*
1	H(1), SWAP(1,1)
2	H(1), CP(2,1,π/4), H(2), SWAP(1,2)
3	H(1), CP(2,1, π/4), CP(3,1, π/8), H(2), CP(3,2, π/4), H(3), SWAP(1,3)
4	H(1), CP(2,1, πI/4), CP(3,1, π/8), CP(4,1, π/16), H(2), CP(3,2, π/4), CP(4,2, π/8), H(3), CP(4,3, π/4), H(4), SWAP(1,4)

These circuits are human-competitive for *n* = 1-3: there is one redundant gate in the circuit for a 1 qubit system, but the other two circuits equal the most efficient known using these quantum gates.

However, this algorithm does not implement the QFT perfectly for systems with more than 3 qubits. The final ITERATE loop always runs for precisely one iteration, and therefore there is always precisely one SWAP gate generated, regardless of the system size. For system sizes above 3, multiple SWAP gates are required to implement the QFT exactly (more precisely, $\lfloor n/2 \rfloor$ gates are needed, where *n* is the system size). The *n*=4 circuit shown is a reliable QFT(4) circuit apart from a missing SWAP(2,3) gate at the end. This algorithm becomes increasingly poor at implementing the QFT as the system size increases.

6.3 An evolved algorithm to implement the QFT on any size of quantum system

When set up with the same parameters as in Section 6.2, but allowed to test candidate solutions against system sizes of 1, 2, 3 and 4 qubits, Q-PACE IV is able to evolve (in 2436 generations) an algorithm that implements the QFT operation exactly on *any* size of quantum system, as follows:

```
ROOT(
 ITERATE(
  MINUS( n, 1, 4 ),
  BODY(
   Create_H( v1, n, n ),
   ITERATE(
    MINUS( n, v1, v1 ),
    BODY(
     Create_CP(PLUS(v1,v2,v1), v1, PLUS(v2,1,4)),
     1, 1),
    var1 ),
   1),
  n ),
 Create_H( n, 1, n ),
 ITERATE(
  DIVIDE( n, 2, n ),
  BODY(
   Create_SWAP(
    v1,
    PLUS( MINUS(n,v2,1), 1, 3 ),
    1 ),
   n, n ),
  3 )
 )
```

This individual, when decoded/executed for different system sizes *n*:, produces the quantum circuits shown in Table 6.

Table 6. Circuits produced by the second evolved algorithm, for *n* = 1-5 qubits

n	*Circuit*
1	H(1)
2	H(1), CP(2,1,π/4), H(2), SWAP(1,2)
3	H(1), CP(2,1, π/4), CP(3,1, π/8), H(2), CP(3,2, π/4), H(3), SWAP(1,3)
4	H(1), CP(2,1, πI/4), CP(3,1, π/8), CP(4,1, π/16), H(2), CP(3,2, π/4), CP(4,2, π/8), H(3), CP(4,3, π/4), H(4), SWAP(1,4), SWAP(2,3)
5	H(1), CP(2,1,PI/4), CP(3,1,PI/8), CP(4,1,PI/16), CP(5,1,PI/32), H(2), CP(3,2,PI/4), CP(4,2,PI/8), CP(5,2,PI/16), H(3), CP(4,3,PI/4), CP(5,3,PI/8), H(4), CP(5,4,PI/4), H(5), SWAP(1,5), SWAP(2,4)

These circuits are human-competitive for all *n*: each one equals the most efficient known circuit for that system size.

7. CONCLUSIONS AND FURTHER WORK

7.1 Summary

The QFT is arguably the most important building block in quantum algorithm construction. Its use as a crucial component of polynomial time quantum algorithms for factorisation and discrete logarithm problems alone guarantee worldwide interest in its implementation.

Spector describes circuits implementing the QFT for small numbers of qubits [19]. Here we have demonstrated correct implementations also for small numbers of qubits (up to 5). However, for the first time, genetic programming has been used to evolve a system size-*independent* algorithm capable of generating a correct circuit for any supplied *n*. The algorithm, when executed, generates efficient circuits. The leap in abstraction level is of crucial importance. Circuits are system size specific; an algorithm can generate a circuit for any supplied size. It captures an intellectual idea about a family of correct circuit structures.

7.2 Can GP compete with humans?

The algorithm evolved using GP described in this paper would appear to be one of the most significant quantum artefacts discovered using evolutionary computation. Since implementations of the QFT have been demonstrated only very recently, it would appear that a claim to be human competitive is justified. We hope its discovery by GP will inspire further interest in the field.

However, the results presented in this paper do not extend the portfolio of known quantum algorithms. Given the difficulty of devising new quantum algorithms analytically, an important open research problem remains: can GP evolve new quantum algorithms to solve open problems in computer science?

8. REFERENCES

[1] H. Barnum, H. J. Bernstein, L. Spector. Quantum circuits for OR and AND of ORs. *J. Physics A: Mathematical and General*, **33**(45):8047-8057, November 2000

[2] H. Barnum, H. J. Bernstein, L, Spector. A quantum circuit for OR. quant-ph/990756

[3] G. Brassard. Teleportation as Quantum Computation. In *Proc. 4th Workshop on Physics and Computation,* New England Complex Systems Institute 1996. Also as quant-ph/9605035, 1996

[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] N. A. Gershenfeld, I. L. Chuang. Bulk Spin-Resonance Quantum Computing. *Science* **275**, 350–356, January 1997

[6] L. K. Grover. A fast quantum mechanical algorithm for database search. *Proc. 28th Ann. ACM Symp. on the Theory of Computing (STOC)*, 212–219, 1996

[7] A. Leier, W. Banzhaf. Evolving Hogg's Quantum Algorithm Using Linear-Tree GP. *GECCO 2003*, 390–400. LNCS 2723, Springer, 2003

[8] A. Leier, W. Banzhaf. Exploring the search space of quantum programs. *CEC 2003*, 170–177. IEEE Press, 2003

[9] A. Leier, W. Banzhaf. Comparison of Selection Strategies for Evolutionary Quantum Circuit Design. *GECCO 2004*, 557–568. LNCS 3103, Springer, 2004

[10] M. A. Nielsen, I. L. Chuang. *Quantum Computation and Quantum Information*. CUP, 2000

[11] P. Massey, J. A. Clark, S. Stepney. Evolving Quantum Programs and Circuits through Genetic Programming, *GECCO 2004*, 569-580. LNCS 3103, Springer, 2004

[12] M. J. Rethinam, A. K. Javali, E.C. Behrman, J.E. Steck, S. R. Skinner. A genetic algorithm for finding pulse sequences for NMR quantum computing. Available as quant-ph/0404170, April 2004

[13] E. Rieffel, W. Polak. An Introduction to Quantum Computing for non-Physicists. Available as quant-ph/9809016, 1998

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

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

[16] L. Spector, H. Barnum, H. J. Bernstein, N. Swamy. Genetic Programming for Quantum Computers. In *Genetic Programming 1998*, 365-374. Morgan Kauffman, 1998

[17] L. Spector, H. Barnum, H. J. Bernstein, N. Swamy. Finding a Better-than-Classical Quantum AND/OR Algorithm using Genetic Programming. *CEC 1999*, 2239–2246. IEEE, 1999

[18] L. Spector, H. Barnum, H. J Bernstein, N. Swamy. Quantum Computing Applications of Genetic Programming. In L. Spector, W. B. Langdon, U.-M. O'Reilly, P. J. Angeleine, eds, *Advances in Genetic Programming 3*, chapter 7, pp 135–160. MIT Press, 1999

[19] L. Spector. *Automatic Quantum Computer Programming: a genetic programming approach*. Kluwer, 2004

[20] L. Spector, H. J. Bernstein. Communication Capacities of some Quantum Gates, discovered in part through Genetic Programming. In *Proc. 6th Int. Conf. Quantum Communication, Measurement, and Computing (QCMC)*, pp. 500–503, 2003

[21] S. Stepney, J. A. Clark. Evolving Quantum Algorithms and Protocols. In M. Rieth, W. Schommers, eds. *Handbook of Theoretical and Computational Nanotechnology*, American Scientific Publishers, 2005 (in press)

[22] R. Van Meter, K. Binkley. Compiling Quantum programs Using Genetic Algorithms. In *The Wild and Crazy Idea Session IV*, abstracts, part of *11th Intl. Conf. Architectural Support for Programming Languages and Operating Systems*, October 2004

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

[24] C. P. Williams, A. G. Gray. Automated Design of Quantum Circuits. In *Quantum Computing and Communications: First NASA Conference, QCQC'98*, 113–125. LNCS 1509, Springer, 1999

[25] T. Yabuki, H. Iba. Genetic algorithms for quantum circuit design – evolving a simpler teleportation circuit. In *Late Breaking Papers at GECCO 2000*, pp. 425–430, 2000