Chapter 8

# EVOLVED QUANTUM PROGRAMS

This chapter presents examples of the automatic production of quantum computer programs via genetic programming. These examples demonstrate how the techniques described in previous chapters can be applied to specific problems. They also provide evidence for the claim that scientifically significant results can be produced via automatic quantum computer programming.

The examples that are presented here are solutions to two types of problems. We call problems of the first type "Boolean oracle analysis" problems because they require us to determine some property of a provided Boolean quantum gate. This gate is often called an "oracle" or a "black box" because we are given little *a priori* information about the gate's construction or behavior. All of these oracles are "Boolean" in the sense that they act by inverting a particular single output qubit when provided with specified combinations of inputs. We are allowed to use the oracle gate, but we are not told in advance which combinations of inputs will produce the inversion — that is what a solution to the problem will tell us. Sometimes we may be "promised" that the oracle is one of some subset of the possible Boolean oracles of the given size; in these cases the problem is to determine *which* member of the subset we have been given.

An example of a Boolean oracle analysis problem is Grover's database search problem, which was discussed earlier in Chapter 2. In Grover's problem the oracle represents a database containing a single "marked" item. We are promised that the oracle inverts its output for a single combination of inputs, which may be considered the address of the marked item. Our task is to determine which of the possible inputs it is for which the inversion is performed.

Other examples presented below — the Deutsch-Jozsa (XOR) problem, the Majority-ON problem, and the OR and AND/OR problems — are similar except that the "promises" that we are given about the oracles and the features of the oracles that we are asked to determine vary from problem to problem. For the Majority-ON problem we attempt not just to solve a single instance of the problem but rather to produce a scaling program that can solve instances of this problem of any size.

Several of these Boolean oracle analysis problems have practical significance because their solutions directly enable us to solve difficult real-world problems more rapidly than is possible on classical computers; for example, Grover's algorithm can be used to provide a quadratic speedup for a host of problems that involve search through unstructured databases.

The second type of problem considered here concerns the classical communication capacity of certain specific quantum gates. The problems of this type that are presented derive from recent research on the tradeoffs between classical communication and entanglement-generating powers of certain unitary transformations (Spector and Bernstein, 2003; Bennett et al., 2004). In these problems the task is to transfer information from one set of qubits to another, without any direct connection between the two sets of qubits aside from a single instance of the gate under investigation. These problems are important not because they have any direct practical application — the gates under consideration do not generally correspond to any real-world communication channels — but rather because their solutions contribute to the development of the fundamental theory of quantum communication and computation.

Sections 8.1 through 8.5 describe specific problems, specific genetic programming techniques that have been used to solve them, and interesting features of evolved solutions. Particular emphasis is given to the author's techniques described in Chapters 6 and 7 as they have been applied in specific cases. Section 8.6 discusses the general significance of the results presented in Sections 8.1 through 8.5, both with respect to the theory of quantum computation and with respect to techniques for automatic quantum computer programming.

## 1. The 1-bit Deutsch-Jozsa (XOR) Problem

In the Deutsch-Jozsa problem (Deutsch and Jozsa, 1992) we are given an oracle with some number of input qubits and one output qubit. We are told that the oracle's function is to invert its output qubit in certain situations (that is, with certain Boolean inputs), and we are promised that the oracle is either *uniform*, meaning that it either *always* or *never* inverts its output qubit, or *balanced*, meaning that it will invert and

*Table 8.1.* Push interpreter parameters for the example run of PushGP on the Deutsch-Jozsa (XOR) problem. Documentation of Push parameters and instructions is available from `http://hampshire.edu/lspector/push.html`.

| | |
|---|---|
| MAX-RANDOM-FLOAT | 1.0 |
| MIN-RANDOM-FLOAT | -1.0 |
| MAX-RANDOM-INTEGER | 10 |
| MIN-RANDOM-INTEGER | -10 |
| EVALPUSH-LIMIT | 150 |
| MAX-POINTS-IN-RANDOM-EXPRESSIONS | 50 |
| MAX-POINTS-IN-PROGRAM | 100 |
| MAX-ORACLE-CALLS | 1 |
| Types | QGATE, FLOAT, CODE, BOOLEAN, INTEGER |
| Instructions | (see Table 8.3) |

not invert *equal numbers of times* if called on all possible (Boolean) inputs. The task is to determine whether a given oracle is uniform or balanced. Classically one would have to query the oracle several times (up to one more than half the number of possible inputs) to be certain of the answer, but quantum computers can do better. Although this problem is not clearly related to any problems of practical significance, it is of historical significance because it was one of the first problems to be shown to be solvable with a better-than-classical quantum algorithm.

The use of genetic programming to re-discover the quantum program that solves the 2-bit version of this problem (which uses an oracle with 4 possible inputs) is documented in (Spector et al., 1998) and (Spector et al., 1999b).[1] Here we document the use of genetic programming to re-discover the quantum program that solves the simpler 1-bit version of this problem. In this version of the problem the oracle has only 1 input qubit and hence two possible inputs (0 and 1). The oracle is uniform, as in the general case, if it either always or never inverts its output qubit. It is balanced in all other cases, in which it inverts its output qubit for one but not the other of its 2 possible inputs. We are therefore asked to determine the truth of the logical formula $I_0 \oplus I_1$, where $I_0$ means "inverts with input 0," $I_1$ means "inverts with input 1," and $\oplus$ is the exclusive OR (XOR) function. The classical version of this problem clearly requires two oracle queries; after a query with one input it will not be known whether the result of a query with the other input will match (meaning that the oracle is uniform) or not (meaning that

[1] In these references the Deutsch-Jozsa problem is referred to as Deutsch's "early promise" problem.

the oracle is balanced). By contrast a quantum program can solve this problem with a single query.

This problem was easily solved using PushGP with the parameters shown in Tables 8.1 and 8.2 and the instruction set shown in Table 8.3, running under the OpenMCL open source Common Lisp system[2] on a 1.33 GHz Apple Macintosh laptop computer with a PowerPC G4 chip. The complete source code for this run, along with the output log, is available online.[3]

The fitness of a Push program was assessed by running it once to produce a QGAME program (which began with the empty "embryo" corresponding to the gate array shown in Figure 8.1), and by testing the QGAME program with the `TEST-QUANTUM-PROGRAM` function described in Chapter 3. The maximum permitted number of oracle calls per case (and therefore the first argument in all calls to `LIMITED-ORACLE`) was 1, so that only the first oracle call in any developed QGAME program would have any effect. The inputs provided to `TEST-QUANTUM-PROGRAM` were:

- `PROGRAM`: The developmental result of executing the chromosomal Push program.
- `NUM-QUBITS`: 2
- `CASES`: (((0 0) 0) ((0 1) 1) ((1 0) 1) ((1 1) 0))
- `FINAL-MEASUREMENT-QUBITS`: (1)
- `THRESHOLD`: 0.48

Fitness was computed as the sum of the number of misses (the first return value from `TEST-QUANTUM-PROGRAM`) and the maximum probability of error on any single case (the second return value).

The fitness of the best program in the first, random generation ("generation 0") was 3.0. Fitness improved rapidly thereafter, including a steep drop at generation 9 when the number of misses of the best program dropped from 2 to 0. At generation 18 a perfect solution was found, with a fitness value of 0 aside from a miniscule round-off error of $4.4 \times 10^{-16}$. A plot of the fitness of the best individual per generation is shown in Figure 8.2.

[2] `http://openmcl.clozure.com/`
[3] `http://hampshire.edu/lspector/aqcp/evolved-xor/`

*Table 8.2.* PushGP genetic programming system parameters for the example run of PushGP on the Deutsch-Jozsa (XOR) problem.

| | |
|---|---|
| MAX-NEW-POINTS-IN-MUTANTS | 20 |
| POPULATION-SIZE | 10,000 |
| TOURNAMENT-SIZE | 7 |
| MUTATION-PROBABILITY | 0.45 |
| CROSSOVER-PROBABILITY | 0.45 |
| MUTATION-OPERATORS | FAIR, PERTURB, ADD, REMOVE |
| CROSSOVER-OPERATORS | FAIR |
| FITNESS-FUNCTION | misses + max probability of error |

*Table 8.3.* Instructions used in the example run of PushGP on the 1-bit Deutsch-Jozsa (XOR) problem.

| | |
|---|---|
| INTEGER | INTEGER.FROMBOOLEAN, INTEGER.FROMFLOAT, INTEGER.>, INTEGER.<, INTEGER.%, INTEGER./, INTEGER.*, INTEGER.-, INTEGER.+, INTEGER.STACKDEPTH, INTEGER.SHOVE, INTEGER.YANKDUP, INTEGER.YANK, INTEGER.=, INTEGER.SWAP, INTEGER.POP, INTEGER.DUP |
| BOOLEAN | BOOLEAN.FROMFLOAT, BOOLEAN.FROMINTEGER, BOOLEAN.NOT, BOOLEAN.OR, BOOLEAN.AND, BOOLEAN.STACKDEPTH, BOOLEAN.SHOVE, BOOLEAN.YANKDUP, BOOLEAN.YANK, BOOLEAN.=, BOOLEAN.SWAP, BOOLEAN.POP, BOOLEAN.DUP |
| CODE | CODE.DISCREPANCY, CODE.DO, CODE.NTHCDR, CODE.NTH, CODE.APPEND, CODE.LIST, CODE.NOOP, CODE.IF, CODE.DO*, CODE.CONS, CODE.CDR, CODE.CAR, CODE.NULL, CODE.ATOM, CODE.QUOTE, CODE.STACKDEPTH, CODE.SHOVE, CODE.YANKDUP, CODE.YANK, CODE.=, CODE.SWAP, CODE.POP, CODE.DUP |
| FLOAT | FLOAT.FROMBOOLEAN, FLOAT.FROMINTEGER, FLOAT.TAN, FLOAT.COS, FLOAT.SIN, FLOAT.>, FLOAT.<, FLOAT.%, FLOAT./, FLOAT.*, FLOAT.-, FLOAT.+, FLOAT.STACKDEPTH, FLOAT.SHOVE, FLOAT.YANKDUP, FLOAT.YANK, FLOAT.=, FLOAT.SWAP, FLOAT.POP, FLOAT.DUP |
| QGATE | QGATE.END, QGATE.MEASURE, QGATE.U2, QGATE.CPHASE, QGATE.SWP, QGATE.CNOT, QGATE.QNOT, QGATE.SRN, QGATE.U-THETA, QGATE.HADAMARD, QGATE.LIMITED-ORACLE, QGATE.GATE, QGATE.TRANSPOSE, QGATE.COMPOSE, QGATE.STACKDEPTH, QGATE.SHOVE, QGATE.YANKDUP, QGATE.YANK, QGATE.=, QGATE.SWAP, QGATE.POP, QGATE.DUP |

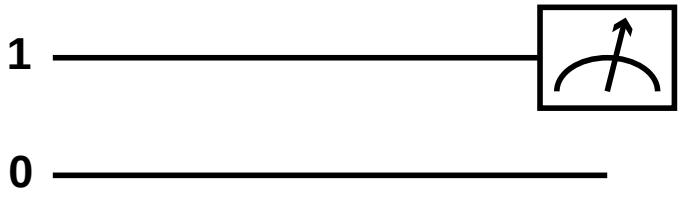


*Figure 8.1.* Gate array diagram for the empty "embryo" with which development begins for the solution to the Deutsch-Jozsa (XOR) problem. The only gate in the embryo performs a measurement of qubit 1; this need not even appear explicitly in the developed QGAME program as the call to `TEST-QUANTUM-PROGRAM` will specify that the final measurement will be performed on qubit 1. The developmental process will add gates from left to right, ending just before the measurement.

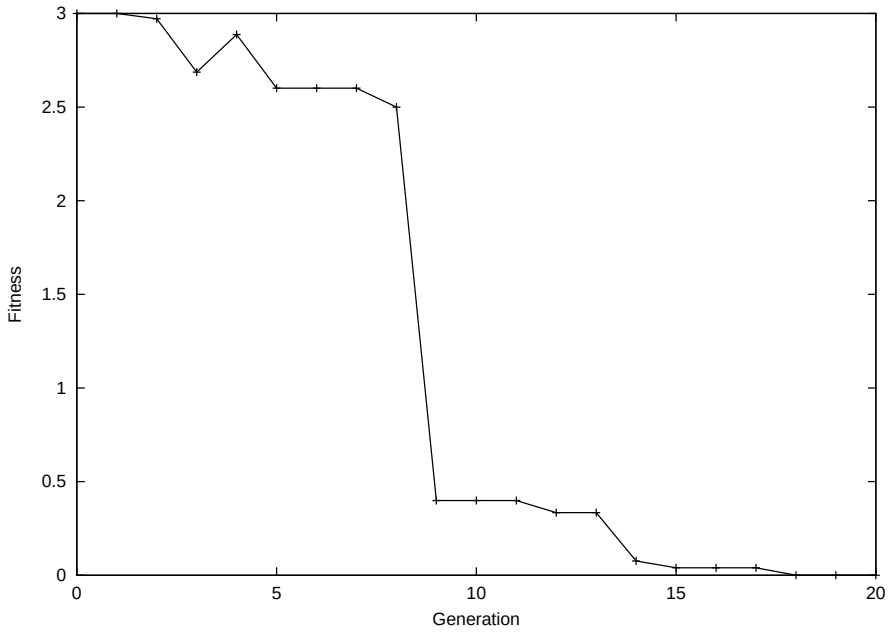


*Figure 8.2.* A plot of the fitnesses of the best individuals in each generation during a run of PushGP on the 1-bit Deutsch-Jozsa (XOR) problem.

Execution was aborted at generation 20, at which time the best reported program was as follows:

```
((BOOLEAN.= INTEGER.> CODE.DO*) ((FLOAT.TAN (FLOAT.<
(BOOLEAN.DUP (BOOLEAN.POP BOOLEAN.SHOVE INTEGER.-
QGATE.CPHASE (CODE.CAR CODE.LIST TRUE)))) (CODE.NULL
((CODE.APPEND) FLOAT.= (BOOLEAN.DUP BOOLEAN.DUP))))
CODE.CDR ((BOOLEAN.YANKDUP INTEGER.* BOOLEAN.=)
(0.16907119750976562D0) -2 (QGATE.SRN QGATE.STACKDEPTH
(QGATE.HADAMARD (QGATE.GATE CODE.STACKDEPTH)) CODE.NULL
(BOOLEAN.SWAP) (INTEGER.YANKDUP BOOLEAN.OR
(((QGATE.TRANSPOSE) CODE.NULL (QGATE.CPHASE INTEGER.>)
CODE.LIST) (QGATE.GATE ((-5 (FLOAT.STACKDEPTH)) CODE.YANK
BOOLEAN.POP))) (INTEGER.DUP)) QGATE.LIMITED-ORACLE))
(FLOAT.% QGATE.STACKDEPTH QGATE.GATE (((5 CODE.SWAP)
QGATE.LIMITED-ORACLE) FLOAT.YANK) FLOAT.SWAP FLOAT.TAN)
(TRUE)) (INTEGER.* (QGATE.SWP FLOAT.STACKDEPTH BOOLEAN.OR
CODE.CDR) BOOLEAN.STACKDEPTH))
```

Regardless of how this Push program is formatted, it is not clear from visual inspection how it works (and it has therefore been presented in the most economical format). Execution of this program produces, via development, the following QGAME program (as expressed in Lisp notation, where "#2A" indicates a 2-dimensional matrix, and with floating point numbers rounded to 4 decimal places):

```
((MATRIX-GATE #2A((0.7071 0.0 0.7071 0.0)
                  (0.0 0.7071 0.0 0.7071)
                  (0.7071 0.0 -0.7071 0.0)
                  (0.0 0.7071 0.0 -0.7071))
              ((HADAMARD 1)))
 (MATRIX-GATE #2A((0.7071 0.7071 0.0 0.0)
                  (-0.7071 0.7071 0.0 0.0)
                  (0.0 0.0 0.7071 0.7071)
                  (0.0 0.0 -0.7071 0.7071))
              (TRANSPOSED ((SRN 0))))
 (LIMITED-ORACLE 1 ORACLE-TT 1 0)
 (LIMITED-ORACLE 1 ORACLE-TT 0 1)
 (MATRIX-GATE #2A((0.7071 0.0 0.7071 0.0)
                  (0.0 0.7071 0.0 0.7071)
                  (0.7071 0.0 -0.7071 0.0)
                  (0.0 0.7071 0.0 -0.7071))
              ((HADAMARD 1))))
```

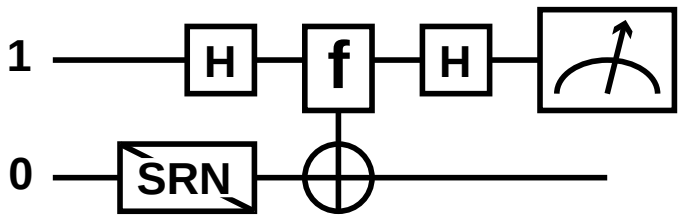


*Figure 8.3.* Gate array diagram for an evolved solution to the Deutsch-Jozsa (XOR) problem. The "f" gate is the oracle. The "SRN" gate with the diagonal line through it on qubit 0 transposed `Square Root of NOT` gate.

The second oracle call is redundant and can be removed; since the oracle limit is 1 a second call to `LIMITED-ORACLE` will have no effect. The first and final gates are simply `HADAMARD` gates applied to qubit 1, while the second gate is a transposed `SRN` ("square root of NOT"; see Chapter 2) gate. The final evolved, developed and simplified quantum program is diagrammed in Figure 8.3. This program solves the 1-bit version of the Deutsch-Jozsa (XOR) problem with 100% certainty using only a single oracle call.

How does this evolved solution solve the 1-bit Deutsch-Jozsa (XOR) problem? The mathematical explanation is straightforward — one needs only to construct and multiply all of the matrices — but it is difficult to provide an intuitive explanation even for such a simple quantum algorithm. The basic idea is indeed intuitive, however: the algorithm first puts both qubits into superpositions of $|0\rangle$ and $|1\rangle$ and then calls the oracle once on this superposition, extracting information about both classical inputs in a single call. This information must then be "decoded" from the resulting superposition by means of an additional `HADAMARD` gate, which reverses the effect of the `HADAMARD` gate prior to the oracle. Note that the final measurement is made on the qubit that is nominally the *input* to the oracle call, while the nominal output is ignored. This highlights one of the ways in which quantum gate arrays differ from classical logic circuits.[4] The oracle call in this case modifies qubit 0, but in doing so it changes every amplitude in the system state. Through this action (which is sometimes called the "back action" of a quantum gate) it changes the effect of the final `HADAMARD` on qubit 1, leading to the measurement of the correct answer for both possible inputs.

[4]The potential deceptiveness of quantum gate array diagrams that results from such differences was discussed in Chapter 3.

## 2. Grover's Database Search Problem

Grover's database search problem was described above in Chapters 1 and 2, the latter of which included a detailed presentation of one solution to the 4-item version of this problem. Grover's problem is an oracle problem, much like the Deutsch-Jozsa problem, except that the "promise" we are given regarding the oracle is different and the task is not just to distinguish two classes of oracles (uniform vs. balanced) but rather to determine exactly which of the possible oracles we have been given.

More specifically, we are promised, in the instance of the problem considered here, that the oracle will invert its output for one and only one input. Our task is to determine which input it is that produces the inversion. This is described as a database problem because we may think of the oracle as a database, for which all of the possible inputs are addresses, and we may think of the output inversion as an answer of "yes" to a database query for a marked item. Under this interpretation we are promised that we have been given a database containing a marked item at one and only one address, and we are asked to determine the address of that item using as few calls to the database query function (oracle) as possible. The number of queries required for a classical program to solve this problem with an $n$-item database is $n-1$ in the worst case, but Grover's algorithm can find the marked item in approximately $\sqrt{n}$ queries. For the 4-item database considered here Grover's algorithm requires only a single database query.

Techniques similar to those described above for the Deutsch-Jozsa problem also permit evolution of a solution to the 4-item database search problem.[5] Because the oracle is in this case a 3-qubit gate (two input qubits and one output qubit), one must use a quantum computer with at least 3 qubits. One must also designate two qubits for final measurements, rather than the one qubit required for Deutsch-Jozsa, since one must be able to read a 2-bit address (0, 1, 2, or 3) from the measurement qubits at the end of the simulation. The cases on which programs are tested for fitness are:

```
(((1 0 0 0) 0)
 ((0 1 0 0) 1)
 ((0 0 1 0) 2)
 ((0 0 0 1) 3))
```

[5]The evolution of a solution to this problem using using "stackless linear genome genetic programming," as described in Chapter 7, is documented in (Spector et al., 1999b).

*Table 8.4.* Push interpreter parameters for the example run of PushGP on the 4-item database search problem. Documentation on Push parameters and instructions is available from `http://hampshire.edu/lspector/push.html`.

| | |
|---|---|
| MAX-RANDOM-FLOAT | 10.0 |
| MIN-RANDOM-FLOAT | -10.0 |
| MAX-RANDOM-INTEGER | 10 |
| MIN-RANDOM-INTEGER | -10 |
| EVALPUSH-LIMIT | 250 |
| MAX-POINTS-IN-RANDOM-EXPRESSIONS | 50 |
| MAX-POINTS-IN-PROGRAM | 100 |
| MAX-ORACLE-CALLS | 1 |
| Types | QGATE, FLOAT, CODE, INTEGER |
| Instructions | (see Table 8.6) |

*Table 8.5.* PushGP genetic programming system parameters for the example run of PushGP on the 4-item database search problem.

| | |
|---|---|
| MAX-NEW-POINTS-IN-MUTANTS | 20 |
| POPULATION-SIZE | 25,000 ($\times$ 10 demes) |
| TOURNAMENT-SIZE | 5 |
| MUTATION-PROBABILITY | 0.45 |
| CROSSOVER-PROBABILITY | 0.45 |
| IMMIGRATION-PROBABILITY | 0.005 |
| MUTATION-OPERATORS | FAIR, GAUSSIAN-PERTURB, ADD, REMOVE |
| CROSSOVER-OPERATORS | STANDARD, FAIR |
| FITNESS-FUNCTION | 10 $\times$ misses + max probability of error |

This means that the answer, to be assembled from the measured values of two qubits (we'll specify these to be qubits 1 and 2, specifying the high-order and low-order bits of the answer respectively), should be 0 if the location of the marked item is (0, 0), 1 if the location is (0, 1), 2 if the location is (1, 0), and 3 if the location is (1, 1).

This problem was solved using PushGP with the parameters shown in Tables 8.4 and 8.5 and the instruction set shown in Table 8.6, running under the CMUCL open source Common Lisp system[6] on a 10-CPU cluster of 2.1GHZ Linux workstations. The complete source code for this run, along with the output logs, is available online.[7]

[6] `http://www.cons.org/cmucl/`
[7] `http://hampshire.edu/lspector/aqcp/evolved-grover/`

*Table 8.6.* Instructions used in the example run of PushGP on the 4-item database search problem.

| | |
|---|---|
| INTEGER | INTEGER.FROMFLOAT, INTEGER./, INTEGER.*, INTEGER.-, INTEGER.+, INTEGER.SWAP, INTEGER.POP, INTEGER.DUP |
| CODE | CODE.DO*COUNT, CODE.DO*TIMES, CODE.FROMFLOAT, CODE.FROMINTEGER, CODE.DO, CODE.NTHCDR, CODE.NTH, CODE.APPEND, CODE.LIST, CODE.NOOP, CODE.IF, CODE.DO*, CODE.CONS, CODE.CDR, CODE.CAR, CODE.QUOTE, CODE.SWAP, CODE.POP, CODE.DUP |
| FLOAT | FLOAT.FROMINTEGER, FLOAT./, FLOAT.*, FLOAT.-, FLOAT.+, FLOAT.SWAP, FLOAT.POP, FLOAT.DUP |
| QGATE | QGATE.END, QGATE.MEASURE, QGATE.CPHASE, QGATE.SWP, QGATE.CNOT, QGATE.QNOT, QGATE.U-THETA, QGATE.HADAMARD, QGATE.LIMITED-ORACLE, QGATE.GATE, QGATE.TRANSPOSE, QGATE.COMPOSE, QGATE.SWAP, QGATE.POP, QGATE.DUP |

The 10-CPU cluster was utilized by means of a scheme of "demes" like that described briefly in Chapter 4. PushGP was started on each of the nodes and the 10 runs were allowed to proceed asynchronously. After the fitness-testing step of each generation a pool of emigrants, consisting of 125 individuals (0.5% of the population size of 25,000) selected via fitness tournaments (with tournament size 5), was written to a shared file system, replacing any previous pool of emigrants from the same node. Following emigration, a randomly selected file of emigrants on the shared file system (which may have come from the same node or from a different node) is read and becomes the pool of immigrants from which the `IMMIGRATION` genetic operator will randomly select individuals in the next offspring-production step. If the attempt to read a file of emigrants from the shared file system fails for any reason (for example because of network problems) then the `IMMIGRATION` operator will act as a reproduction operator, producing clones of individuals from the current population.

This run also utilized the matrix literalization scheme discussed in Chapter 7. After the fitness-testing step of each generation the Push programs were processed in order of fitness (best first) until at least 10 matrix literals were obtained. This was accomplished by re-evaluating each Push program to produce, via development, a QGAME program, and by compressing strings of matrices in the developed QGAME program to produce compressed matrix literals. These literals were then available for inclusion in mutations performed during the next offspring-production step. In addition, this run utilized a `GAUSSIAN-PERTURB`

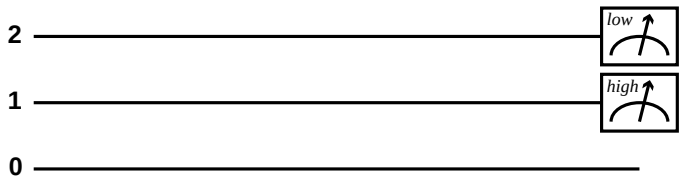


*Figure 8.4.* Gate array diagram for the empty "embryo" with which development begins for the solution to the database search problem. The only gates in the embryo perform measurement of qubits 1 (the high order bit of the answer) and 2 (the low order bit of the answer). The developmental process will add gates from left to right, ending just before the measurements.

genetic operator, the idea for which was described in Chapter 7. When this operator is chosen for a particular instance of mutation,[8] a child is produced from the parent by adding mean 0, standard deviation 0.01 Gaussian noise to each floating-point literal in the parent.

As with the Deutsch-Jozsa example in the previous section, the fitness of a Push program was assessed by running it once to produce a QGAME program (which began in this case with the empty "embryo" corresponding to the gate array shown in Figure 8.4), and by testing the QGAME program with the `TEST-QUANTUM-PROGRAM` function described in Chapter 2. The maximum permitted number of oracle calls per case was again 1, so that only the first oracle call in any developed QGAME program would have any effect. The fitness cases were those listed above and the threshold for a "miss" was again 0.48. Fitness was computed as the sum of 10 times the number of misses (the first return value from `TEST-QUANTUM-PROGRAM`) and the maximum probability of error for any one case (the second return value from `TEST-QUANTUM-PROGRAM`); this is the "lexicographic" fitness component combination scheme that was discussed in Chapter 7.

The fitnesses over the 10 demes are plotted in Figure 8.5. The elimination of "misses" is clearly visible as large drops in fitness values, which are lexicographic combinations of misses ($\times$ 10) and maximum probability of error per case. Fitness improvements within particular levels of misses are obscured by the scale, but Figure 8.6 shows the additional detail at the level of zero misses. The first deme to achieve a perfect fitness value of zero did so at generation 113, while the last deme to achieve a perfect fitness value did so at generation 152. The last of these

[8] In PushGP, a random one of the specified mutation operators is selected for each instance of mutation. Similarly for crossover: if multiple operators are specified then each instance of crossover uses a randomly selected crossover operator.

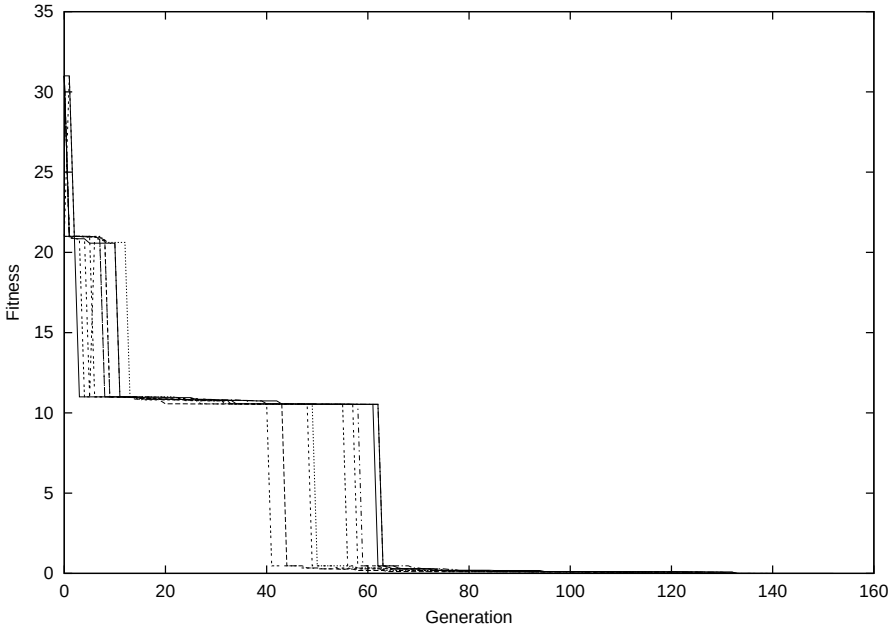


*Figure 8.5.* A plot of the fitnesses of the best individuals in each generation during a run of PushGP on the 4-item database search problem. This figure is dominated by the large drops due to the decreases in the "misses" component of the fitness function; it shows the overall structure of the evolutionary process but not the fine structure of fitness improvements at each level. Figure 8.6 shows a closer view of the improvements in fitness after all of the misses were eliminated. This run was conducted on a cluster of 10 computers that ran asynchronously, sharing individuals between generations (see text), and a line appears in the graph for each of the 10 runs. Because the individual runs ran asynchronously they reached particular generations at different times and one must be careful when inferring relations between runs from this graph; for example, an event that appears to the right of another event may actually have preceded that other event in time, and may even have influenced that other event via migration.

perfect-fitness individuals was chosen, arbitrarily, as the basis for the following analysis.

The evolved solution Push program contained 100 points, which was the maximum permitted.[9] The average number of points in the population that included this solution was 80.5, and the median fitness in this population was 0.0026. The solution Push program contained 5 unitary matrix literals, produced via the matrix literalization process described above, some of which were derived from other matrix literals earlier in

[9]Each instruction, literal, and pair of parentheses counts as one point.

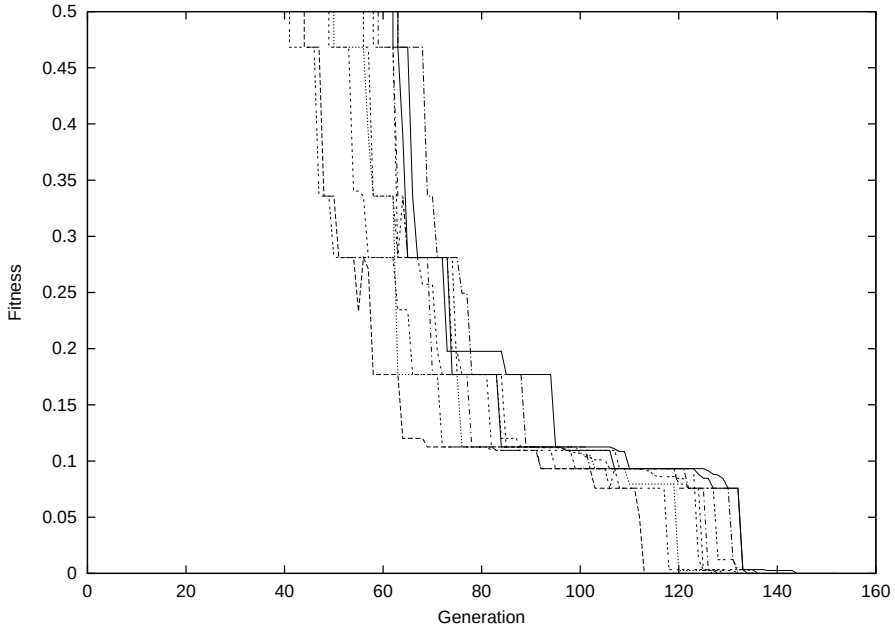


*Figure 8.6.* A plot of the fitnesses of the best individuals in each generation during a run of PushGP on the 4-item database search problem. This is a closer view of the graph in Figure 8.5, showing the improvements in fitness after all of the "misses" components of the fitness function were eliminated.

the evolutionary process. For example, one of the matrix literals is the composition of two instances of another matrix literal, which in turn includes three instances of a matrix that appears to have been produced by an earlier matrix literalization process. The inclusion of the matrix literals makes the printed representation of this Push program quite large (3,458 characters, not counting spaces); it is therefore not included here, although it can be found online.[10]

Execution of the evolved Push program produces, via development, a QGAME program consisting of 18 matrix gates. Some of the matrices in these gates appeared in the Push program as matrix literals, but others were produced by the execution of the Push program either from primitive gates or from matrix literals. For example, one matrix in the developed QGAME program is a transposed version of one of the matrix literals in the Push program. Another matrix in the developed QGAME program is a transposed version of one of the matrix literals in

[10]See `http://hampshire.edu/lspector/aqcp/evolved-grover/`, at the end of the log file `pushgp-output.n01.bw01.hampshire.edu`.

the Push program that has also been augmented by an additional `QNOT` gate. Again, because the textual version of this this program is verbose it is not included here.

As in the Deutsch-Jozsa example in the previous section, some of the gates in the final QGAME program are unnecessary and can be pruned from the result. Of particular interest in the present case is the fact that two of the gates, although they include matrix literals with rather complex histories, combine the matrices from those histories to produce identity operations; components of these histories are also used elsewhere in the final QGAME program to greater effect. The final QGAME program, after hand pruning and with the matrices removed for legibility, is as follows:

```
((HADAMARD 1)
 (MATRIX-GATE <matrix1> <history1>)
 (HADAMARD 1)
 (HADAMARD 0)
 (MATRIX-GATE <matrix2> <history2>)
 (LIMITED-ORACLE 1 ORACLE-TT 2 1 0)
 (HADAMARD 2)
 (MATRIX-GATE <matrix3> <history3>)
 (MATRIX-GATE <matrix4> <history4>)
 (HADAMARD 1))
```

The matrix indicated as `<matrix1>` is just a transposed version of the matrix indicated as `<matrix2>`, which has the following history:

```
((COMPRESSED
  ((COMPRESSED ((U-THETA 2 1.233552982796235)))
   (COMPRESSED
    ((COMPRESSED ((QNOT 0))) (COMPRESSED ((CNOT 1 2))))))))
```

The matrix indicated as `<matrix3>` has the following history:

```
((COMPRESSED ((HADAMARD 1)))
 (COMPRESSED
  ((COMPRESSED
    (TRANSPOSED ((U-THETA 1 1.0642909109545906))))))
 (COMPRESSED
  ((COMPRESSED
    (TRANSPOSED ((U-THETA 1 1.0642909109545906))))))
 (COMPRESSED
  ((COMPRESSED
    (TRANSPOSED ((U-THETA 1 1.0642909109545906)))))))
```

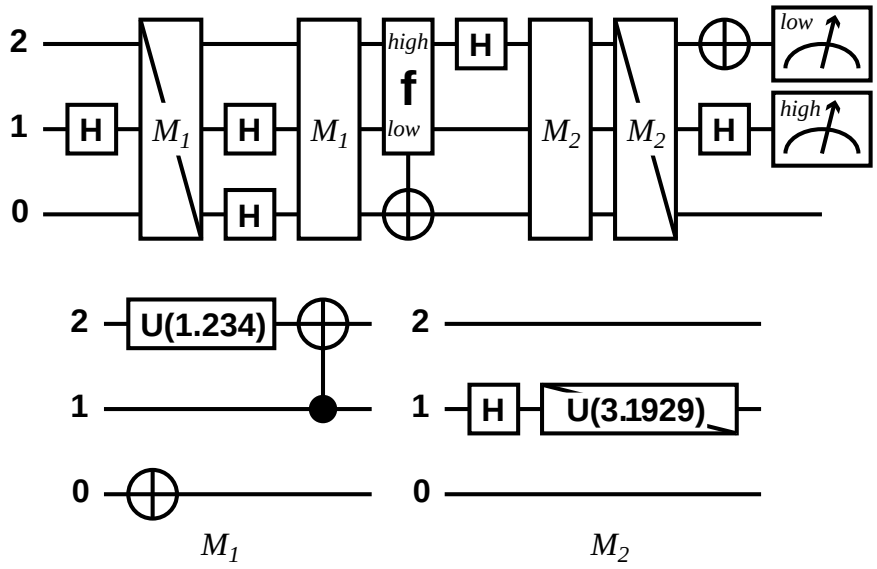


*Figure 8.7.* A gate array diagram for an evolved version of Grover's database search algorithm for a 4-item database. The full gate array is shown at the top, with $M_1$ and $M_2$ standing for the smaller gate arrays shown at the bottom. A diagonal line through a gate symbol indicates that the matrix for the gate is transposed. The "f" gate is the oracle.

The matrix indicated as `<matrix4>` is a transposed version of the matrix indicated as `<matrix3>`, to which a `QNOT` gate has also been added on qubit 2.

The resulting quantum gate array is diagrammed in Figure 8.7. $M_1$ in the figure corresponds to `<matrix2>` and $M_2$ corresponds to `<matrix3>`; the contents of each of these matrices are indicated in the smaller gate array diagrams in the bottom half of the figure. The transpositions in matrices 1 and 4 are indicated by the diagonal lines, and the additional `QNOT` gate that evolved as part of `<matrix3>` is drawn separately on the qubit 2 line in the main diagram. This gate array solves the 4-item database search problem with 100% certainty using only a single oracle call. The evolved gate array exhibits several forms of modularity, some of which were achieved via recursive matrix literalization and others of which owe to the code-manipulation and matrix-manipulation facilities of the Push instruction set used for this run.

How does this evolved solution work? At a general level of description the solution is the same as that presented in Section 3.3 above: a superposed state is fed into the call to the oracle gate and subsequent "decoding" gates extract the position of the marked item from the states

*Table 8.7.* Push interpreter parameters for the example run of PushGP on the Scaling Majority-ON problem. Documentation on Push parameters and instructions is available from `http://hampshire.edu/lspector/push.html`.

| | |
|---|---|
| MAX-RANDOM-FLOAT | 1.0 |
| MIN-RANDOM-FLOAT | -1.0 |
| MAX-RANDOM-INTEGER | 10 |
| MIN-RANDOM-INTEGER | -10 |
| EVALPUSH-LIMIT | 150 |
| MAX-POINTS-IN-RANDOM-EXPRESSIONS | 50 |
| MAX-POINTS-IN-PROGRAM | 100 |
| MAX-ORACLE-CALLS | 1 |
| Types | QGATE, FLOAT, CODE, BOOLEAN, INTEGER |
| Instructions | (see Table 8.9) |

in which the *address* qubits (as opposed to the *output* qubit) are left by the action of the oracle. The solution presented here is, however, considerably more complex than that presented in Section 3.3.[11] Part of the reason for this difference is that the result presented earlier was subjected to further human editing,[12] but part may also be due to an unfortunate evolutionary accident early in the run presented here. The oracle call in the evolved gate array uses qubit 2 as the high-order input and qubit 1 as the low-order input, while the measurements specified in the embryo use the opposite designation. If the programs that achieved limited success early in this run included the oracle call with this "backwards" configuration, then it may have been easier for evolution to find improvements that compensated for this configuration through additional gates than through the substitution of an alternative oracle configuration. Another factor contributing to the complexity of this solution may be the use of matrix literalization, which facilitates the evolution of quantum programs containing complex modules; while this probably extends the power of the automatic quantum computer programming system it may also have the unfortunate side effect of encouraging the generation of unnecessarily complex solutions.

[11] It is also considerably more complex than the solution evolved by the author previously using other techniques (Spector et al., 1999b).

[12] The editing performed here was limited to the removal of gates that had no effect on the result; further analysis may produce additional simplifications by substituting single gates for groups of gates, etc.

*Table 8.10.* Push interpreter parameters for the example runs of PushGP on the OR and AND/OR problems. Documentation on Push parameters and instructions is available from `http://hampshire.edu/lspector/push.html`.

| | |
|---|---|
| MAX-RANDOM-FLOAT | 1.0 |
| MIN-RANDOM-FLOAT | -1.0 |
| MAX-RANDOM-INTEGER | 9 |
| MIN-RANDOM-INTEGER | -10 |
| EVALPUSH-LIMIT | 150 |
| MAX-POINTS-IN-RANDOM-EXPRESSIONS | 50 |
| MAX-POINTS-IN-PROGRAM | 100 |
| MAX-ORACLE-CALLS | 1 |
| Types | QGATE, FLOAT, CODE, INTEGER |
| Instructions | (see Table 8.12) |

## 4. The OR and AND/OR Problems

The OR and AND/OR problems are oracle problems similar to the XOR problem described above, but they ask us to determine a different property of the oracles. The OR problem is identical to the XOR problem except that we are asked to determine the truth of the logical formula $I_0 \vee I_1$, where $I_0$ means "inverts with input 0," $I_1$ means "inverts with input 1," and $\vee$ is the (inclusive) OR function. In the notation used for QGAME's `TEST-QUANTUM-PROGRAM` function, the cases that we use to assess fitness are:

```
(((0 0) 0)
 ((0 1) 1)
 ((1 0) 1)
 ((1 1) 1))
```

In other words, we are asked to determine whether the oracle we have been given *ever* inverts its output qubit, whether for a 0 input, or for a 1 input, or for both. This turns out to be a harder question to answer than the XOR question (which omits the "or both"), and it is known that there is no error-free single query solution.

But a quantum program can nonetheless do better than a classical program on this problem, and genetic programming was used to discover a quantum algorithm that performed better than any that had previously been published. The evolved quantum program has a maximum probability of error of $\frac{1}{10}$. This is better than can be achieved using even a probabilistic classical program, which must necessarily have a max-

*Table 8.11.* PushGP genetic programming system parameters for the example runs of PushGP on the OR and AND/OR problems.

| | |
|---|---|
| MAX-NEW-POINTS-IN-MUTANTS | 10 |
| POPULATION-SIZE | 50,000 ($\times$ 13 demes) |
| TOURNAMENT-SIZE | 7 |
| MUTATION-PROBABILITY | 0.48 |
| CROSSOVER-PROBABILITY | 0.48 |
| IMMIGRATION-PROBABILITY | 0.005 |
| MUTATION-OPERATORS | PERTURB, ADD, REMOVE |
| CROSSOVER-OPERATORS | FAIR |
| SIZE-PRESSURE | 2, IDEAL-SIZE= 50 |
| FITNESS-FUNCTION | if misses $= 0$ then:<br>$0.1 \times p_{max}$<br>otherwise:<br>$(0.1 \times p_{max}) + \left\lfloor 10^6 \times \frac{\sum_{i=1}^{n} \frac{1}{1+e^{-\psi(p_i - 0.48)}}}{n} \right\rfloor$<br>where:<br>$n$ = number of fitness cases,<br>$p_i$ = probability of error for case $i$,<br>$p_{max}$ = maximum probability of error,<br>and $\psi = e^{(e+1)}$ |

imum probability of error of at least $\frac{1}{6}$. The evolved program, which was originally produced using the LGP genetic programming system[15] and a precursor to QGAME, is presented along with an analysis of the problem's classical and quantum complexity in (Spector et al., 1999a) and (Barnum et al., 2000).

In this section we describe the more recent evolution of an equivalent quantum algorithm using PushGP and QGAME. For this run an alternative, stackless implementation of the `QGATE` data type was used. There was no `QGATE.GATE` Push instruction and the execution of Push instructions corresponding to primitive quantum gates (such as `QGATE.HADAMARD`) sent QGAME instructions directly to the developing embryo. This decreased the amount of Push code required to build simple QGAME programs, but it did not allow the Push program to manipulate and store novel unitary matrices during development.

The implementation of `QGATE.MEASURE` in this run was also unusual. The implementation used in the previous examples simply added an instruction expression, "`(measure` $q$`)`," to the developing embryo, with $q$ taken from the `INTEGER` stack (modulo the number of qubits in the sys-

[15] Available from `http://helios.hampshire.edu/lspector/code.html`.

*Table 8.12.* Instructions used in the example runs of PushGP on the OR and AND/OR problems. These runs used alternative implementations of the QGATE instructions (see text).

| | |
|---|---|
| INTEGER | INTEGER.MAX, INTEGER.MIN, INTEGER.%, INTEGER./, INTEGER.*, INTEGER.-, INTEGER.+, INTEGER.STACKDEPTH, INTEGER.SHOVE, INTEGER.YANKDUP, INTEGER.YANK, INTEGER.SWAP, INTEGER.POP, INTEGER.DUP |
| CODE | CODE.QUOTE, CODE.SWAP, CODE.POP, CODE.DUP |
| FLOAT | FLOAT.TAN, FLOAT.COS, FLOAT.SIN, FLOAT.MAX, FLOAT.MIN, FLOAT.%, FLOAT./, FLOAT.*, FLOAT.-, FLOAT.+, FLOAT.STACKDEPTH, FLOAT.SHOVE, FLOAT.YANKDUP, FLOAT.YANK, FLOAT.SWAP, FLOAT.POP, FLOAT.DUP |
| QGATE | QGATE.MEASURE, QGATE.HALT, QGATE.U2, QGATE.CPHASE, QGATE.SWP, QGATE.CNOT, QGATE.QNOT, QGATE.SRN, QGATE.U-THETA, QGATE.HADAMARD, QGATE.LIMITED-ORACLE |

tem). Subsequent calls to QGATE.END were required to complete the branches of the computation for the two possible measurement outcomes (0 and 1).[16] For the present run an alternative implementation of QGATE.MEASURE was used that ensures, assuming that the Push program that contains it runs to completion, that all measurements are followed by complete branches for both possible outcomes. QGATE.MEASURE does this by taking two arguments from the CODE stack in addition to the index of the qubit to be measured (which is taken from the INTEGER stack). It then does the following:

- Adds the MEASURE expression to the developing QGAME program.
- Recursively executes one of the popped pieces of code (the one that was deeper in the stack), possibly adding additional elements to the developing QGAME program in the process.
- Adds an (END) to the developing QGAME program.
- Recursively executes the other popped piece of code, possibly adding additional elements to the developing QGAME program.
- Adds another (END) to the developing QGAME program.

The other parameters for this run are shown in Tables 8.10, 8.11, and 8.12. The SIZE-PRESSURE parameter referred to in Table 8.10 relates to an experimental feature of PushGP that is intended to help control

[16] See page 26 for the syntax of measurement constructions in QGAME.

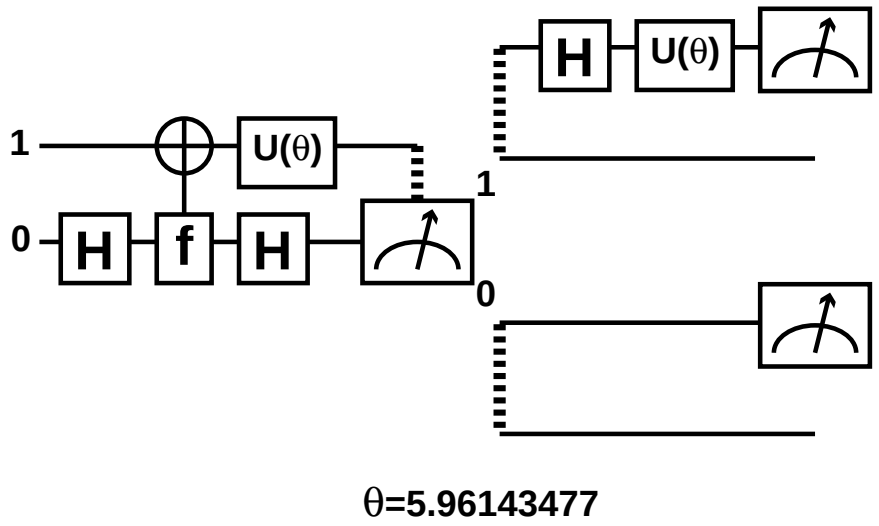


*Figure 8.9.* A gate array diagram for an evolved solution to the OR oracle problem. The gate marked "f" is the oracle. The two sub-diagrams on the right represent the two possible execution paths following the intermediate measurement. In the bottom sub-diagram the result of the intermediate measurement is 0 and the result of the overall computation is read immediately from the other qubit. In the top sub-diagram the result of the intermediate measurement is 1 and additional gates are applied to the other qubit prior to the final measurement.

program bloat; when this feature is enabled each attempt to use a genetic operator causes the operator to be called the indicated number of times (2 in this case), producing that number of potential offspring. The single offspring closest in size to the specified `IDEAL-SIZE` is chosen from these, and the others are discarded.

The fitness function for programs that achieve zero misses is the maximum probability of error on any single fitness case times 0.1. For programs with misses, however, the fitness function is a lexicographic combination of a sigmoid function (based on the differences between each probability of error and the "miss threshold") and the maximum probability of error. As discussed in Chapter 7, this sigmoid function provides a smoother fitness landscape while still prioritizing the elimination of misses, although the effectiveness of this measure has not been empirically tested.

The gate array in Figure 8.9 shows one result of this run, obtained at generation 302 and simplified by hand. This result exhibits elements of modularity even though it used only a minimal subset of Push's code-manipulation instructions and only one instruction — the modi-

fied `QGATE.MEASURE` instruction — that triggers recursive execution of code on the `CODE` stack. For example, the same angle appears twice as an argument to `U-THETA`, even though there are no duplicate floating point literals in the evolved Push program, and the final QGAME program includes three `HADAMARD` gates even though the evolved Push program contains only two instances of `QGATE.HADAMARD`.

This algorithm calls the oracle on a qubit in a superposition of $|0\rangle$ and $|1\rangle$ and then, after an additional Hadamard transformation of the qubit used as the input (and which was affected by the "back action" of the oracle), performs an intermediate measurement of the input qubit. Regardless of the result of this intermediate measurement, the final measurement is made on qubit 1 (as was specified in the embryo), but in one case qubit 1 is transformed, using copies of gates that appeared earlier in the algorithm, prior to the final measurement.

The maximum probability of error for this algorithm is $\frac{1}{10}$, while classical algorithms necessarily have a probability of error of at least $\frac{1}{6}$. The existence of quantum algorithms with a maximum probability of error of $\frac{1}{10}$ was first discovered by genetic programming.

The AND/OR problem extends the OR problem to a larger oracle and to a more complex logical property. In this problem we are asked to determine if the cases for which the 2-qubit oracle flips its output qubit satisfy the logical formula $(I_{00} \vee I_{01}) \wedge (I_{10} \vee I_{11})$, where $\wedge$ is the AND function. This formula is illustrated as an "and/or tree" in Figure 8.10. In the notation used for QGAME's `TEST-QUANTUM-PROGRAM` function, the cases that we use to assess fitness are:

```
(((0 0 0 0) 0)
 ((0 0 0 1) 0)
 ((0 0 1 0) 0)
 ((0 0 1 1) 0)
 ((0 1 0 0) 0)
 ((0 1 0 1) 1)
 ((0 1 1 0) 1)
 ((0 1 1 1) 1)
 ((1 0 0 0) 0)
 ((1 0 0 1) 1)
 ((1 0 1 0) 1)
 ((1 0 1 1) 1)
 ((1 1 0 0) 0)
 ((1 1 0 1) 1)
 ((1 1 1 0) 1)
 ((1 1 1 1) 1))
```

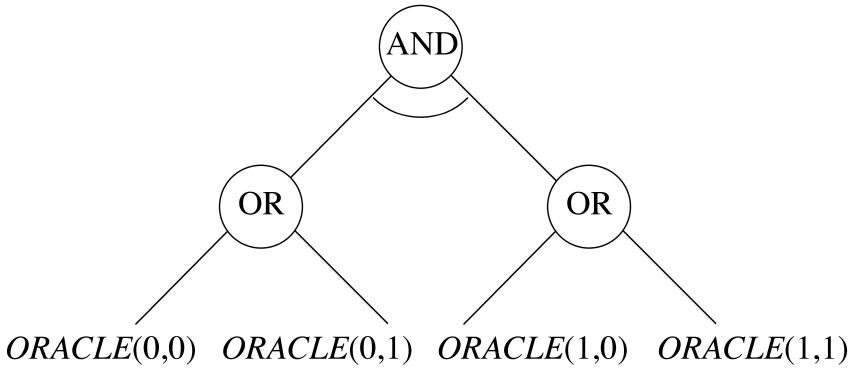


*Figure 8.10.* An AND/OR tree describing the nature of the AND/OR oracle problem.

The existence of better-than-classical quantum algorithms for the AND/OR problem was first discovered by genetic programming. The first evolved programs for this problem (which were also evolved using LGP and a predecessor to QGAME) are presented, along with a complexity analysis, in (Spector et al., 1999a) and (Barnum et al., 2000). Here we present a program equivalent to the best of these that was evolved more recently using PushGP and QGAME, with the same parameters as those used for the run on the OR problem above (Tables 8.10, 8.11, and 8.12); only the fitness cases and the size of the embryo were changed.

The evolved quantum program, a hand-simplified version of which is shown in Figure 8.11, has a maximum probability of error of 0.28731. By contrast the best that can be achieved by a probabilistic classical program is an error probability of $\frac{1}{3}$. Like the solution to the OR problem above, this algorithm works by calling the oracle on inputs in superposition and by subsequently performing intermediate measurements on the input qubits, which will have been affected by the back action of the oracle call. The final measurement is again made on the oracle's output qubit, but only after additional transformations to the output qubit that are conditional on the intermediate measurements.

It is also noteworthy that the Push program that produced this solution contained only one instance of `QGATE.MEASURE`, meaning that the multiple-measurement solution resulted from the use of the use of Push's code-manipulation instructions, only a minimal subset of which were included in this run.

It is natural to ask how these algorithms, both for the OR problem and for the AND/OR problem, can be scaled up to larger problem in-

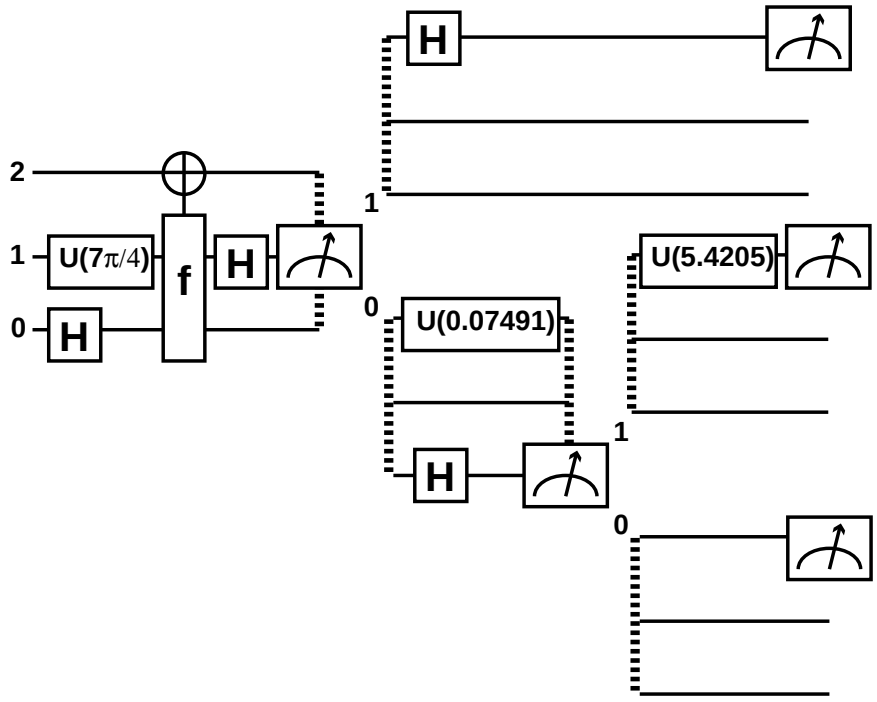


*Figure 8.11.* A gate array diagram for an evolved solution to the AND/OR oracle problem. The gate marked "f" is the oracle. The sub-diagrams on the right represent the possible execution paths following the intermediate measurements.

stances. Unfortunately, simple concatenations of the evolved algorithms do not suffice for this purpose. It is possible, however, that solutions to larger problem instances may be discovered through future genetic programming runs, and that the principles by which these algorithms can be scaled up can subsequently be inferred.

## 5. Gate Communication Problems

This section describes several problems that emerged from explorations of the relations between the communication and entanglement-generation capacities of certain quantum gates (Spector and Bernstein, 2003; Bennett et al., 2004). These explorations involved several iterative cycles of problem formulation, genetic programming, and human analysis. All of the genetic programming runs used PushGP, QGAME, and techniques similar to those described above. Due to space limitations the details of the many individual runs will not be presented here, except for the few novel features introduced specifically for these problems.

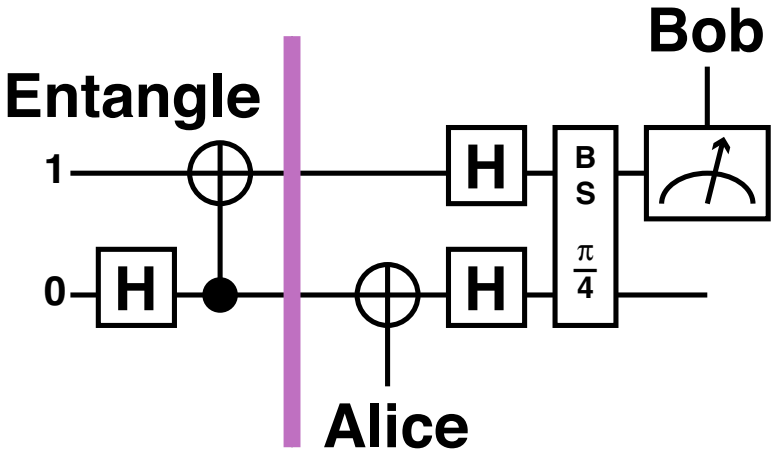


*Figure 8.14.* A gate array diagram for an evolved protocol for communicating one classical bit through a $BS(\frac{\pi}{4})$ gate in the context of prior entanglement. The entanglement-generating gates, to the left of the vertical bar, were included in the embryo to which the developmental process was applied.

## 6. Significance of These Results

Most of the results presented in this chapter demonstrate the human competitive nature of genetic and evolutionary computing technologies. A few also demonstrate the production, via genetic programming, of genuinely new knowledge with respect to the nature and power of quantum computing.

What is meant by "human competitive" in this context? John Koza and his colleagues have developed a list of eight criteria for the assertion of human competitiveness of results produced by intelligent technologies (Koza et al., 2003). These criteria are expressed relative to measures that are commonly employed to assess *human* contributions to scientific and technological research and development, such as patents and publications in reputable, peer-reviewed scientific journals. The criteria all focus on properties of the results themselves, not on their automatic production by computer systems.

Several of Koza's criteria apply to the results presented in this chapter. Two that are particularly helpful in assessing the significance of these results are the following:

- B: The result is equal to or better than a result that was accepted as a new scientific result at the time when it was published in a peer-reviewed scientific journal.

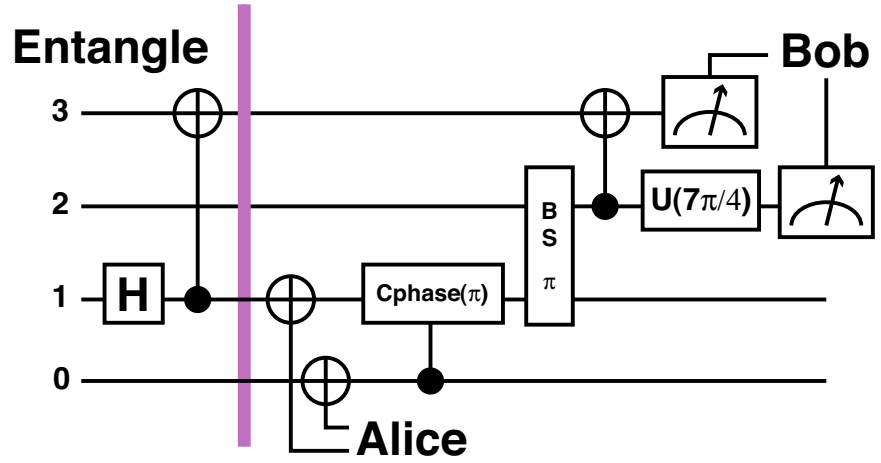


*Figure 8.15.* A gate array diagram for an evolved protocol for communicating two classical bits through one application of a $BS(\pi)$ gate in the context of prior entanglement. This is a form of quantum superdense coding re-discovered by genetic programming. The entanglement-generating gates, to the left of the vertical bar, were included in the embryo to which the developmental process was applied.

- D: The result is publishable in its own right as a new scientific result—independent of the fact that the result was mechanically created.

All of the results in this chapter, with the exception of the result for the scaling Majority-ON problem, meet criterion B. The results for the OR, AND/OR, and gate communication problems also meet criterion D, as established by publications in physics venues (Barnum et al., 2000, Spector and Bernstein, 2003).

The solution to the 1-bit Deutsch-Jozsa (XOR) problem appears simple in retrospect, but one must remember that this surprising and powerful effect went unnoticed for the first 60 years following the development of the underlying quantum mechanics. And even now it is counterintuitive to most people. It is true that much of the intelligence behind this result lies in the human discovery that the problem was worth posing in the first place, but the steps from the problem statement to a solution are nonetheless non-trivial. The fact that genetic programming can proceed automatically to a solution when provided only with the problem statement and a generic set of quantum gates is therefore significant.

Similar comments apply to the result for Grover's database search problem. Although a human being (Lov Grover) was responsible for the insight that quantum computers could outperform classical computers on this problem, the production of a better-than-classical quantum algo-

rithm for the problem is nonetheless difficult and represents a significant achievement for an automatic programming system. It is also noteworthy that the first time this result was produced by genetic programming it exceeded the expectations of the person performing the experiment (the author of this book), who had naively assumed that the $\sqrt{n}$ improvement would allow only for a two-oracle-call solution. Although the zero-error, single-call solution added nothing to the state of the art in quantum computing, its possibility was news to the designer and user of the automatic quantum computer programming system (who was at that time new to the field of quantum computing). This is important because it demonstrates that the system can produce knowledge beyond that possessed by the system designers or users.

The results on the OR and AND/OR problems were published in *Journal of Physics A: Mathematical and General* on the strength of their contributions to the theory of quantum computing, not on the basis of their production by mechanical means. Although the article does briefly describe the genetic programming methodology that produced the results, neither the article's title nor its abstract mention how the results were produced. The novel methodology by which these results were produced would probably not, by itself, warrant publication in this particular journal, which routinely publishes articles on quantum complexity theory but not on the design of automatic programming systems. The fact that these results were published in a high-quality, peer-reviewed *physics* journal demonstrates that the approach to automatic quantum computer programming described in this book can produce new scientific results that are on par with those produced by human scientists.

The result on the Scaling Majority-ON problem is of more limited significance; it serves only to demonstrate how genetic programming can be employed to find scalable solutions to problems that have instances of various sizes. But the result itself is not better than classical, and it is also fairly obvious. It is significant only insofar as it points the way to more ambitious applications of genetic programming to other problems in the future.

Several of the results on classical communication via particular quantum gates are new scientific contributions, significant independent of the means by which they were produced. Evidence for this is their publication in the *Proceedings of the Sixth International Conference on Quantum Communication, Measurement, and Computing*. It is also noteworthy that in this case the genetic programming system was employed in a role similar to that of a scientific colleague. The system was used first to investigate a particular question ("Can classical information be trans-

mitted via a `SMOLIN` gate?") but its result ("Yes") was not the end of the story; the details of the result inspired a round of human analysis and the production of new questions for the system. Results of the runs on these secondary questions have led to further analysis and insights. This work is ongoing and additional publications in the physics literature are expected in the future (Bennett et al., 2004).