Implementing Grover's Algorithm Using Linear Transformations in Haskell

A. Murray Gross and Justin Stallard

Brooklyn College, The City University of New York

Abstract

In contrast to the usual approach to simulating quantum computing algorithms as a series of operations to be performed on a set of "qubits," we have used the Haskell programming language to implement Grover's fast search algorithm as a composition of functional transformations applied, as a final step, to a set of qubits (a "quantum register"). In this approach, which has been (at least implicitly) suggested in the literature, but, as far as we have been able to determine, not been realized in a working program, the composition is constructed by means of common matrix manipulations and takes advantage of the associativity of matrix operations to eliminate complicated computations usually associated with simulating quantum computations. We present the crucial code sections, along with actual program results.

We discuss the implications of our approach in the areas of quantum program simulation, algorithm analysis, algorithm construction, and construction of languages for quantum programming.

1 INTRODUCTION

1.1 Quantum Computing

While it is beyond the scope of this paper to provide a detailed introduction to quantum computing,[1] we begin by reviewing the specific quantum properties that are crucial to understanding Grover's fast search algorithm [6]. We then present the algorithm itself, along with central portions of the code, and conclude with a discussion of some of the implications of the approach we have taken to simulate a quantum computation. A detailed example of a search with Grover's algorithm is given in the appendix.

Consider, then, as a starting point, a flip-flop representing a single binary digit in the classical world. Given the value of the bit, we can predict with certainty how the bit will change when a pulse is applied to the flip-flop. In effect, provided the flip-flop is not broken, we can always know precisely what the value of the represented bit is without examining it. In the quantum world, however, things are fuzzy: A so-called quantum flip-flop can be put in a state in which the only way to determine its state is to examine its output. The output of a quantum flip-flop will be zero with some probability x, and one with probability $1-x$. This measurement then actually changes the state of the quantum flip-flop to match the result of the measurement. Thus, any further measurement will yield the same result as the first.

[1] The interested reader is referred to [9], [7], and [1]

One way to deal with the behavior of our so-called quantum flip-flop is to adopt the counter-intuitive idea that it is in both the zero and one states at the same time. Another approach, which is favored by the authors, is simply to consider the state of the quantum flip-flop to be undetermined until it is measured, at which point the value it represents will be determined by some probability distribution. In any case, regardless of the viewpoint adopted, if a measurable object can present any of two or more different states when measured, we say it is in a superposition of states; an entity that can be placed into a superposition of only two possible states is known as a quantum bit, or qubit.

Now, a second quantum property that we need is the fact that we can operate on all of a set of states simultaneously by applying a linear transformation[2] to the superposition of all states in the set. This transformation can be performed by matrix multiplication. It turns out, however, that we also need a way to combine several states into one[3]; this is achieved by using a tensor product. It is the exponential increase in the dimensions of the resulting vector spaces that provides quantum computations with their famous parallelism.

Finally, the last property that we need is that matrix products are associative.[4] Indeed, the difference between our simulation of Grover's algorithm and other simulations and descriptions we have been able to locate in the literature [11, 12, 5, 8, 10] is that we associate to the left, while conventional simulations associate to the right. This radically simplifies the code and the approach. For example, we perform a series of operations $\mathbf{A}, \mathbf{B}, \mathbf{C}$ on a state $\vec{s}$ as

$$(\mathbf{CB})\,\mathbf{A})\vec{s},$$

rather than

$$(\mathbf{C}\,(\mathbf{B}\,(\mathbf{A}\vec{s}))).$$

As one example of the difficulties encountered by the conventional approach, consider the quantum simulator (in the Haskell language) written by Jan Skibinski [11, 12]. In this simulator, operations are repeatedly applied to a set of quantum bits in a superposed state. Almost every such operation generates a new, larger expression representing the new superposed state, a situation that the simulator deals with by building a tree of expressions involving the original superposed states; we found this approach difficult to apply to a real problem.

A second approach is to break down each linear transformation to be applied to a set of qubits to primitives like "quantum AND" and "quantum OR," effectively constructing a special purpose machine for the computation.[5] It strikes us that this approach is akin to requiring an engineer faced with a complicated computation to write his program in terms of NAND gates, an approach that has long been abandoned for complicated classical computations.

---

[2]Linearity is required by the physics on which quantum computation is based.

[3]For example, to combine several of our so-called quantum flip-flops into a single quantum register.

[4]This is not a quantum property, but a matrix property.

[5]This is very similar to the way analog computers were once used.

To be sure, there are other quantum properties that are important for other algorithms, such as the fact that the values of the probabilities involved are actually the squares of complex numbers, which has very interesting consequences as minus signs come and go when transformations are iterated. Grover's algorithm, however does not require reference to them, though the interested reader is referred to such texts as [9] and [7] for the additional details.

1.2 Grover's Algorithm

Grover's algorithm solves a relatively simple problem: Given a function f and a value y in the range of f, find an x such that $f(x) = y$. In the absence of further information about the properties of f, a classical approach to solving this problem requires us to do a linear search through all the members of the domain, trying each one to find out if it is the one sought, i.e., the problem is one of linear search, which has computational complexity $O(N)$.

Grover, however, uses quantum effects (i.e., quantum computation) to apply the function to all of the values in its domain at the same time, and then uses a clever device to extract the answer sought from the superposition that is obtained: Through a series of iterations of a process we call diffusion (details below), he increases the probability of the value we seek at the expense of the probabilities of the remaining values, with the maximum probability being reached after $\sqrt{N}$ iterations [6]. At this point, we can determine which of the values has been selected by simply examining the probabilities, and selecting the value with the maximum probability (which we find is quite clearly distinguished from the others, at least for small register sizes).[6]

It should be clear that were this algorithm to be implemented with real quantum hardware, its computational complexity would be $O\left(\sqrt{N}\right)$, which Grover proved [6] to be optimal in its class.

2 HASKELL IMPLEMENTATION

2.1 State Vectors

A qubit can be effectively represented by a two-dimensional vector in a complex vector space. This vector is either in state 0, 1, or some superposition of the two. We are not concerned with individual qubits, however; we are concerned only with having linear representations of quantum registers. Since a quantum register can be represented by the tensor product of the qubits it is composed of, an n qubit register is represented by a 2^n-dimensional vector in a complex vector space. The qubits are abstracted out, as their simulation is not necessary. This implementation uses Haskell arrays of complex values to store such state vectors.

[6]This is not the case in the real quantum world. It only works here because we can look at the individual probabilities of the resulting vector collapsing into any particular state if it were measured.

2.2 Linear Transformations

All linear transformations in an n-dimensional vector space can be represented by $n \times n$ matrices. Since quantum operations are essentially just linear transformations, any quantum operation on a register containing n qubits can be represented by a $2^n \times 2^n$ matrix. There is no need to think about quantum gate operations or what is happening to the qubits at the hardware level. This implementation of Grover's algorithm uses Haskell arrays to represent all linear transformation matrices.

3 REQUIREMENTS FOR GROVER'S ALGORITHM

The steps required in Grover's algorithm are as follows [6]:

1. Place the quantum register in an equal superposition of all possible states.
2. Repeat $\sqrt{N}$ times:
   (a) Apply the selective phase inversion transform to the register.
   (b) Apply the inversion-about-average transformation to the register.
3. Measure the resulting state.

3.1 Walsh-Hadamard Transform

The first step in Grover's Algorithm is to initialize a state vector to an equal superposition of all possible states. This is easily achieved by using the Walsh-Hadamard transformation. The matrix representation of the Hadamard transform that operates on a single qubit is as follows:

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

For an n qubit system, the Walsh-Hadamard transform is represented by the matrix resulting from the tensor product of n Hadamard transformations. For example, for a 2 qubit system

$$\mathbf{W}_2 = \frac{1}{\sqrt{2}} \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \frac{1}{2} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix},$$

and in general

$$\begin{aligned} \mathbf{W}_1 &= \mathbf{H} \\ \mathbf{W}_n &= \mathbf{W}_{n-1} \otimes \mathbf{H}. \end{aligned}$$

The `hadamard` function takes as its input a number of qubits and returns an array containing the matrix representation of the appropriate Walsh-Hadamard transform.

```
-- ------------------------------------------------------ --
-- This function generates the hadamard matrix for n qubits  --
-- ------------------------------------------------------ --
hadamard    :: (RealFloat a) =>
               Int -> Array (Int,Int) a
hadamard 1  =  listArray ((0,0),(1,1))
                         [(1/(sqrt 2)), (1/(sqrt 2)),
                          (1/(sqrt 2)), (-1/(sqrt 2)) ]
hadamard n  =  hadamard 1 'tensor' hadamard (n-1)
```

3.2 Selective Phase Rotation

The selective phase rotation is the real meat of Grover's algorithm. Consider the classical function

$$f(x) = \begin{cases} 1 & \text{if } x = c \\ 0 & \text{otherwise} \end{cases}.$$

When we are given a quantum function $f : |x,b\rangle \rightarrow |x,b \oplus f(x)\rangle$,[7] the selective phase rotation rotates the phase of the portion of the state vector where $x = c$ by π radians. This is equivalent to inverting the phase wherever $x = c$ (which is why this operation is sometimes instead called the selective phase inversion).

Applying this function on a register that is in an equal superposition of all possible states, choosing $b = \mathbf{H} \otimes |1\rangle$,[8] yields a state where the phase of all x where $x = c$ is inverted and b is unchanged [9]. Let us call this resulting state $|\tilde{x},b\rangle$. In order to continue with this implementation of Grover's algorithm, we need to somehow obtain a representation of the state $|\tilde{x}\rangle$.

Consider what we obtain when we apply the transformation $I \otimes \mathbf{H}$ on $|\tilde{x},b\rangle$:

$$(I \otimes \mathbf{H})\,|\tilde{x},b\rangle = |\tilde{x},1\rangle = |\tilde{x}\rangle \otimes |1\rangle.$$

All we need now in order to represent $|\tilde{x}\rangle$ is a way to do what we call "tensor division" by $|1\rangle$. We have a function that does just this, and it is used in the `invertPhase` function which takes a quantum function $\mathbf{P} : |x,b\rangle \rightarrow |x,b \oplus f(x)\rangle$ and a state x and returns the state $|\tilde{x}\rangle$ as defined above.

```
-- ------------------------------------------------------ --
-- Selective phase inversion. Given a quantum function p,     --
-- apply the selective phase inversion matrix on x.           --
-- ------------------------------------------------------ --
invertphase :: (RealFloat a) =>
               Array (Int,Int) a ->  Array (Int,Int) a
                                     -> Array (Int,Int) a
invertphase x p  = ixmap ((0,0),(2^n-1,0))
                         (\(i,0) -> (2*i+1,0)) (doit)
                 where
                   doit = ((identmat n) 'tensor' (hadamard 1))
```

[7] $|s\rangle$ is the Dirac ket notation for the vector representation of the quantum state s. See [9] for more details. Also, $\oplus$ represents exclusive-OR.

[8] That is, the tensor product of the Hadamard transform and a qubit in the state 1.

```
                    'matMult' (p 'matMult' (x 'tensor'
                       ((hadamard 1) 'matMult' (one))))
    ((li,lj),(ui,uj))  =  bounds x
    n    = floor (logBase 2 (fromIntegral (ui+1)))
```

3.3 Inversion About the Average

The inversion-about-average transformation (what Grover also calls the diffusion transform) [6] does exactly what one would think: it inverts each component of a vector $\vec{v}$ about the average of all the components of $\vec{v}$. To create such a transformation, we first look at the transformation $\mathbf{A}$ such that $\mathbf{A}\vec{v} = \vec{v}'$, where each of the components of $\vec{v}'$ is the average of all the components of $\vec{v}$: $\mathbf{A}_{ij} = \frac{1}{N}$ where N is the number of components of $\vec{v}$. For example, in a 4-dimensional vector space,

$$\mathbf{A} = \begin{bmatrix} 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 \end{bmatrix}.$$

Now we want to invert about the average. We want a transformation $\mathbf{D}$ such that $\mathbf{D}\vec{v} = \vec{v}'$, where $\vec{v}'$ is the result of inverting each of the components of $\vec{v}$ about the average of all of them. Therefore, $\vec{v}'_i$ must equal $(\mathbf{A}\vec{v})_i - \vec{v}_i + (\mathbf{A}\vec{v})_i$ for all i. It now follows that $\mathbf{D} = -I + 2\mathbf{A}$, or in other words,

$$\mathbf{D}_{ij} = \begin{cases} -1 + \frac{2}{N} & \text{if } i = j \\ \frac{2}{N} & \text{if } i \neq j \end{cases}.$$

Haskell is very well suited for generating such a matrix. The `diffusion` function takes as its input a number of qubits, and returns an array containing the matrix representation of the appropriate inversion-about-average transformation.

```
-- ------------------------------------------------------ --
-- This function generates the diffusion matrix for an n  --
-- qubit register                                         --
-- ------------------------------------------------------ --
diffusion   :: (RealFloat a) =>
               Int -> Array (Int,Int) a
diffusion n = listArray ((0,0),(2^n-1,2^n-1))
                        (map ( ((+)(2/(2^n))) . negate)
                             (elems (identmat n)))
```

4 IMPLICATIONS

It is not our code, or even our implementation, to which we wish to call attention,[9] but the fact that this code provides a clear, concise and transparent description of the algorithm, unlike the commonly used melange of Dirac bra-kets, matrix manipulations, and (sometimes convoluted) diagrams and natural language.

[9] In fact, both are quite pedestrian.

Indeed, we submit that development in the field of quantum computation has long been hobbled by a lack of suitable language, and, further, that a lazy functional language (and Haskell in particular) provides the field of quantum computation not only with a language that, because its laziness mimics the behavior of quantum processes, is particularly appropriate for programming, but a language that is appropriate for presentation, study, and analysis of quantum computation. In fact, we submit that Haskell (with appropriate and easily built extensions) offers to quantum computing the same service to understanding and analysis as Aho, Hopcroft, and Ullman's "Pidgin ALGOL" [2], Cormen, Leiserson, Rivest, and Stein's unnamed pseudo-code [3], and the host of other invented pseudo-code and actually implemented languages (e.g., Algol-60) that have unarguably been instrumental in communicating, teaching, and analyzing classical algorithms and computational processes.[10]

Now, as important as the availability of an appropriate language may be to furthering the study of a subject, the fact that lazy functional languages appear to provide such a language for quantum computation leads to what we surmise is a potentially significant conjecture: It would appear from the structure of such languages and what can be said in them that all quantum programs can be modeled successfully by nondeterministic probabilistic automata, a conjecture that is often implied in the literature,[11] but, as far as we have been able to determine, has never been explicitly stated.[12] If this conjecture is in fact correct, it would have a significant corollary: Rather than being a new paradigm (as was functional programming when first invented, in comparison with procedural programming), quantum computing is actually a metaphor for nondeterministic probabilistic automata, about which we know a great deal. In turn, this would explain why the great promise of quantum computing has been so poorly met. In fact, what might be doable with quantum computing has in large part actually already been done!

On the other hand, identification of quantum computing as a metaphor does not in any way reduce its value or utility. Metaphors are powerful tools for illumination and explication. Indeed, a well constructed metaphor is at once instructive, illuminative, and profound, often providing insight, motivation, and direction for further study. However, as effective and moving as metaphors might be (see, for example Donne's poetry or Shakespeare's plays), we think it wisest to adopt Prospero's knowing parental smile as he hears Miranda explode into adulthood. It is not, he knows perfectly well, a "brave new world," but it is a world that can provoke wonder and delight. And we think that quantum computing will lead us, step by step, to discoveries that will provide us with the same sort of excitement and

[10] And, we can add, if there were at present a suitable alternative to lazy functional languages for clear, concise, and transparent description of quantum algorithms, our thoughts on the matter would probably be of little significance.

[11] See, for example, repeated use of Turing machines for the purpose of analyzing the computational complexity of quantum computation [1, 4]

[12] Although we learned of some unpublished work based on this conjecture, we have no details yet [14].

satisfaction to be had from discoveries that advance the state of our art.

5 THE NEW WORLD, THROUGH A GLASS, DARKLY

At the present time, in order to test our conjectures, we plan to implement several quantum algorithms. Assuming that our conjectures hold true, and we have no reason to believe they won't, we think that the next step is to define a vocabulary for a quantum-computation description language that is adequate to the task: A vocabulary that is merely universal (and several are known, e.g. [1]) is *not* sufficient, in fact, since that would be much the equivalent of offering an engineer a programming vocabulary consisting of the single operation NAND. To be sure, NAND would provide, with enough effort at construction of more complex operations, all that is *necessary*, but it most certainly would not be particularly practical for programming any useful engineering computation. In a word, we need a set of transforms that are not merely universal, but also convenient, and in some sense complete, as opposed to simply universal.

We know, for example, that a Walsh-Hadamard operation is certainly necessary, along with time- and space-efficient matrix and tensor operations. But what else do we need for "convenience"? Shall we include the diffusion operation, or is that too specific to Grover's algorithm and its offspring? How about a quantum Fourier transform? Shall we include the elementary quantum AND and OR, or are these too elementary to be relegated to implementation of a more powerful language? And how about the "tensor division" operation that we invented for our implementation of Grover's algorithm?

Questions like those above, we think, open an extended area of inquiry that we think will ultimately be fruitful, not only for simulation of quantum computations, but for programming real quantum computers when and if they finally become available. And as long as we are limited to quantum simulations, we think extension of the Haskell language to include a quantum programming language by way of a library of extensions (functions) provides an ideal vehicle for the task we have set out to perform.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Noson S. Yanofsky, Assistant Professor of Computer Science at Brooklyn College, for his guidance and allowing us an advance copy of his soon to be published book, *Quantum Computing for Computer Scientists* [13]. We would also like to thank the symposium reviewers for incisive and helpful comments.

A SAMPLE RUN

In this example, our program is given a function that operates on an 8 qubit register. The function's value is 1 when its argument is 243, and 0 otherwise. The following is the result of running our implementation of Grover's algorithm on this function. The output is a Haskell array with the index representing a state and the value representing the square root of the probability that the register is in that state. As expected, the probability that the register is in state 243 is very high compared to all the others, and is in fact very close to 1.

```
array ((0,0),(255,0)) [
((0,0),-2.2526511019408314e-2),
((1,0),-2.2526511019408314e-2),
((2,0),-2.252651101940831e-2),
((3,0),-2.252651101940831e-2),
((4,0),-2.2526511019408293e-2),
((5,0),-2.2526511019408293e-2),
((6,0),-2.2526511019408293e-2),
((7,0),-2.2526511019408293e-2),
((8,0),-2.2526511019408293e-2),
((9,0),-2.2526511019408318e-2),
((10,0),-2.2526511019408314e-2),
((11,0),-2.2526511019408314e-2),
((12,0),-2.2526511019408314e-2),
((13,0),-2.2526511019408314e-2),
((14,0),-2.2526511019408314e-2),
((15,0),-2.2526511019408304e-2),
((16,0),-2.2526511019408307e-2),
((17,0),-2.2526511019408318e-2),
((18,0),-2.2526511019408318e-2),
((19,0),-2.2526511019408318e-2),
((20,0),-2.2526511019408318e-2),
((21,0),-2.2526511019408328e-2),
.
.
.
((233,0),-2.2526511019408377e-2),
((234,0),-2.2526511019408377e-2),
((235,0),-2.2526511019408373e-2),
((236,0),-2.252651101940838e-2),
((237,0),-2.2526511019408377e-2),
((238,0),-2.2526511019408377e-2),
((239,0),-2.2526511019408377e-2),
((240,0),-2.2526511019408384e-2),
((241,0),-2.252651101940838e-2),
((242,0),-2.2526511019408384e-2),
((243,0),0.9330604786558996),
((244,0),-2.252651101940838e-2),
((245,0),-2.2526511019408377e-2),
((246,0),-2.2526511019408373e-2),
((247,0),-2.2526511019408363e-2),
((248,0),-2.2526511019408366e-2),
((249,0),-2.2526511019408366e-2),
```

```
((250,0),-2.252651101940836e-2),
((251,0),-2.252651101940835e-2),
((252,0),-2.252651101940836e-2),
((253,0),-2.252651101940835e-2),
((254,0),-2.252651101940835e-2),
((255,0),-2.252651101940835e-2)]
```

REFERENCES

[1] Dorit Aharonov, *Quantum Computation*, 1998. e-print http://arxiv.org/abs/quant-ph/9812037

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company (Reading), 1974.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms, Second Edition*, The MIT Press (Cambridge), 2001.

[4] David Deutsch, *Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer*, Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences, Vol. 400, No. 1818 (Jul. 8, 1985), pp. 97-117.

[5] Thorsten Altenkirch and Jonathan Grattage, *A functional quantum programming language*, 20th Annual IEEE Symposium on Logic in Computer Science, 2005. e-print http://arxiv.org/abs/quant-ph/0409065

[6] Lov K. Grover, *A fast quantum mechanical algorithm for database search*, Proceedings, 28th Annual ACM Symposium on the Theory of Computing (STOC), May 1996, pp. 212-219. e-print http://arxiv.org/abs/quant-ph/9605043

[7] Michael Nielsen and Isaac Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press (Cambridge), 2000.

[8] Bernhard Ömer, *Quantum Programming in QCL*, Master thesis, Technical University of Vienna, 2001, http://tph.tuwien.ac.at/ oemer/doc/quprog.pdf

[9] Eleanor G. Rieffel and Wolfgang Polak, *An Introduction to Quantum Computing for Non-Physicists*, ACM Computing Surveys (CSUR), Volume 32, Issue 3 (September 2000), pp. 300 - 335. e-print http://arxiv.org/abs/quant-ph/9809016

[10] Amr Sabry, *Modeling Quantum Computing in Haskell*, Proceedings of the 2003 ACM SIGPLAN workshop on Haskell. http://doi.acm.org/10.1145/871895.871900

[11] Jan Skibinski, *Literate Haskell module QuantumVector.lhs*. Available at http://web.archive.org/web/20010603005905/http://www.numericquest.com/haskell/QuantumVector.html

[12] Jan Skibinski, *Haskell Simulator of Quantum Computer*. Available at http://web.archive.org/web/20010803034527/http://www.numericquest.com/haskell/QuantumComputer.html

[13] Noson S. Yanofsky and Mirco A. Mannucci, *Quantum Computing for Computer Scientists*, To be published by Cambridge University Press.

[14] Stathis Zachos, Private communications.