Christopher J. Wright

ID: 42807671
Supervisor: Prof. Thorsten Altenkirch
Module: G53IDS

2018/19

G53IDS Dissertation

Quantum Computing in Haskell

Submitted April 2019, in partial fulfilment of the conditions for the award of the degree BSc Computer Science

Author: 42807671, Christopher J Wright, psycjw
Supervisor: Prof. Thorsten Altenkirch, psztxa

School of Computer Science
University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text.

Signature

Date 09/04/2019

Abstract

Quantum computing is growing daily as new quantum computers are created, and new research is done. There are already many desirable uses of quantum computing, such as factoring integers in $O(logN)$ [Sho94] and searching unsorted databases in $O(\sqrt{N})$ [Gro96]. There are even quantum games being created, such as Quantum Battleships [Woo17].

As the field expands, more people will find interest in the field and attempt to learn and pursue a career concerning quantum computers, and so suitable learning and experimental platforms need to be present to help this happen. This is the main inspiration for this project; create a stable up-to-date platform for users to run quantum computations on a classical computer via quantum simulation, while also improving my own understanding and experience of the subject for the future.

This dissertation explains the basics of quantum computing, and provides possible methods for the implementation of certain quantum algorithms on a classical computer using Haskell, with a quantum-computation simulating package called QIO [qio16]. This package already includes structures necessary for quantum computations, such as the qubit and unitary functions, however does not work on the latest Haskell update (v8.4.3) and lacks complex quantum algorithms. Therefore, the package is first updated, and then 2 quantum algorithms are implemented. Following this, a quantum circuit builder is designed and implemented in the package, allowing users to create and test quantum circuits using a intuitive GUI.

Each part of the project, namely the two algorithms implemented and the circuit builder, is completely separate and functions independantly, and so I split the project up into 3 parts. Each part is designed, implemented, tested and evaluated before moving to the next part - avoiding confusion between parts and ensuring one part is complete before attempting the next.

Contents

1 Introduction to Quantum Computing

1.1 Brief History

The idea of a quantum computer was first introduced in a talk in 1981 by Nobel prize-winning physicist Richard P. Feynman. In his talk, reproduced in the *International Journal of Theoretical Physics* in 1982 [Fey82], he proposed that it would be impossible to simulate a quantum system on a classical computer efficiently. At the same time, physicist Paul Benioff demonstrated that quantum mechanical systems can model classical Turing machines [Ben82], and created the first recognizable theoretical framework for a quantum computer.

This resulted in David Deutsch's creation of the first universal quantum computer [DP85]. This computer could do things that a universal Turing machine cannot, such as generate truly random numbers, and simulate finite physical systems.

Over time, quantum algorithms have been developed which solve problems faster and more efficiently than classical algorithms can, demonstrating the main use for quantum computers. The main examples of such algorithms would be the Deutsch-Jozsa algorithm [DJ92], Shor's algorithm [Sho94] and Grover's algorithm [Gro96].

1.2 QIO

QIO is a Haskell package built by Alexander S. Green and Prof. Thorsten Altenkirch [qio16], which allows users to simulate a quantum computer on their own classical computer. Typical quantum operations are built up of unitary (reversible) functions under a monoid `U`, and can then be run using a monad `QIO` and the function `applyU`. This `QIO` monad also has functions `mkQbit` and `measQbit`, which can be used to create a new qubit and measure an existing qubit respectively.

This leads to an intuitive platform which can be used to create quantum computations, and then run/simulate them with 2 very simple functions `run :: QIO a -> IO a` and `sim :: QIO a -> Prob a`, where `run` is used to execute a quantum computation and return the value of qubits measured based on their probabilities and a pseudorandom number generator. The `sim` function is similar, however after executing a quantum computation, it returns the probability distribution for the qubits measured, showing the possible states and their probabilities when measured.

I will use this package to help introduce and teach quantum computing, providing relevent code segments to improve clarity, and create programs to highlight certain features of quantum computations.

1.3 The Qubit

A classical computer uses a bit to store data. This bit can only be in one of two states, 0 or 1. Strings of bits are then used to represent data and instructions, forming the basis of classical computation.

However, a quantum computer uses a quantum bit, known as a qubit. This qubit can simply be in the classical states 0 or 1, or in a superposition of both, i.e., be in both states at once.

This means an operation performed on a qubit effectively acts on both 0 and 1 at the same time.

A qubit can be described by a linear combination of $|0\rangle$ and $|1\rangle$,

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \quad \alpha, \beta \in \mathbb{C}, \quad |\alpha^2| + |\beta^2| = 1$$

where $|\alpha^2|$ represents the probability of measuring the 0 state, and $|\beta^2|$ the 1 state. Hence the normalization condition $|\alpha^2| + |\beta^2| = 1$. This condition also corresponds to the fact that $|\psi\rangle$ is a unit vector in the complex vector space.

Measuring a qubit will collapse its wave function due to Born's rule [Bor]. This means that, when measured, a qubit will collapse to be either $|0\rangle$ or $|1\rangle$. For example, if we take the qubit

$$|+\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle$$

and measure it, the probability of getting $|0\rangle$ or $|1\rangle$ is $\frac{1}{2}$. We can therefore say that the qubit $|+\rangle$ is in an equal superposition.

In QIO, a qubit is referred to as a `Qbit`. One can be created simply using the `mkQbit :: Bool -> QIO Qbit` function mentioned above, and is created based on the boolean value passed in. The function `measQbit :: Qbit -> QIO Bool` can then later be used to measure the state of the qubit passed in.

```
make2qubits :: QIO (Bool, Bool)
make2qubits = do
    q0 <- mkQbit False -- This will create a qubit, q0, in the state |0>
    q1 <- mkQbit True -- This will create a qubit, q1, in the state |1>
    b0 <- measQbit q0 -- This will measure the qubit q0, returning False
    b1 <- measQbit q1 -- This will measure the qubit q1, returning True
    return (b0, b1) -- Returns (False, True)
```

Qubit base states can also be represented as vectors, with $|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$ and $|1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$. The qubit $|\psi\rangle$ above can be rewritten, introducing a multiplier known as the global phase, making the coefficient for $|0\rangle$ (i.e., α) real and non-negative. For example, the state $\frac{-i}{\sqrt{2}}\,|0\rangle + \frac{i}{\sqrt{2}}\,|1\rangle$ is the same as $\frac{1}{\sqrt{2}}\,|0\rangle + \frac{-1}{\sqrt{2}}\,|1\rangle$ with a global phase of $-i$. This global phase does not affect the amplitudes of the states, and so doesn't change the qubit's value when measured.

This qubit state $|\psi\rangle = \alpha'\,|0\rangle + \beta'\,|1\rangle$ can now be written in the form

$$|\psi\rangle = \cos(\frac{\theta}{2})\,|0\rangle + e^{i\phi}\sin(\frac{\theta}{2})\,|1\rangle\,, \qquad \theta = 2\cos^{-1}(\alpha'), \quad \phi = \Im(\ln(\frac{\beta'}{\sin(\frac{\theta}{2})}),$$

$$0 \leq \theta \leq \pi, \quad 0 \leq \phi \leq 2\pi$$

From this, it is easy to see that a qubit can be represented geometrically using the angles θ and ϕ, on what is known as the Bloch sphere.

1.4 The Bloch Sphere

The Bloch sphere, named after physicist Felix Bloch, is a geometrical representation of a two-state quantum mechanical system (e.g. a qubit). As shown above, a qubit's coefficients can be expressed in terms of 2 angles, θ and ϕ. This makes it extremely easy to see what happens when operations are performed on qubits, and it is clear that a qubit pointing to a point above the origin has a higher $|0\rangle$ amplitude than $|1\rangle$, and vice versa.

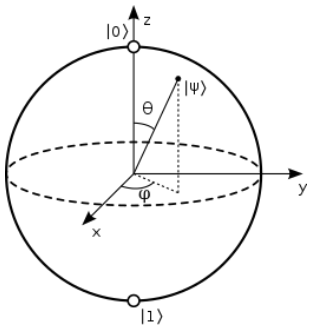


Figure 1: A Qubit represented geometrically on the Bloch Sphere

We can calculate the angle θ using QIO's underlying structure and the `sim` function. The angle ϕ cannot be calculated in this way, as we use the probabilities for each state for the calculation.

The `sim` function returns a value of type `Prob a`, where `a` is given by the quantum computation it is applied to (i.e. applying `sim` to a function of type `QIO Bool`, will return a value of type `Prob Bool`). This `Prob` type is simply a wrapper around vectors given by real probabilities, defined in the `Qio.hs` file:

```
data Prob a = Prob {unProb :: Vec RR a}
```

where `RR` is simply a `Double` value, and `Vec` is a wrapper around a list of pairs, defined in the `Vec.hs` file:

```
newtype Vec x a = Vec {unVec :: [(a,x)]} deriving Show
```

From this, it is easy to see that

```
Prob Bool = Prob {unProb :: Vec RR Bool}
          = Prob {unProb :: Vec {unVec :: [(Bool, Double)]}}
```

Now, we can extract the probabilities for qubits, and calculate θ.

```
quantum :: QIO Bool
quantum = do
    q <- mkQbit True
    measQbit q
```

```
theta :: QIO Bool -> Double
theta c = 2 * (acos alpha)
    where
        ps = unVec $ unProb $ sim c
        alpha = sqrt $ snd $ last ps
```

The function `quantum` can be changed to perform any operation on the qubit, and as long as the function's type does not change, then `theta quantum` will always return θ for the qubit measured. E.g.

```
quantum :: QIO Bool        -> theta quantum = 3.141592653589793
quantum = do
    q <- mkQbit True
    measQbit q

quantum :: QIO Bool        -> theta quantum = 0.0
quantum = do
    q <- mkQbit False
    measQbit q

quantum :: QIO Bool        -> theta quantum = 1.5707963267948966
quantum = do
    q <- mkQbit False
    applyU (uhad q)
    measQbit q
```

1.5 Decoherence

Put simply, decoherence is the loss of coherence. This basically means that the quantum state described by a system is no longer in superposition, and leads to a collapse of the wave function. If this occurs during a computation, all information stored in qubits is lost, and the computation needs to be restarted. Decoherence can occur naturally after a certain amount of time as the system will be entangled with its environment [Zur91], and is the main problem quantum computing faces.

1.6 Basic Operations

1.6.1 1-Qubit Gates

A 1-qubit gate can be thought of as a rotation about the Bloch sphere, and as the state space of a qubit is continuous, there are an infinite amount of 1-qubit gates. Any complex unitary 2x2 matrix represents a 1-qubit gate, and there are 4 main 1-qubit gates: the Hadamard gate, and the 3 Pauli- gates. Note that all quantum gates must be unitary (i.e., is its own inverse).

As mentioned earlier in section 1.2, QIO uses a monoid `U` to form unitary functions. These functions are defined as 5 distinct forms, using recursion to create sequences of functions:

```
data U = UReturn
       | Rot Qbit Rotation U
       | Swap Qbit Qbit U
       | Cond Qbit (Bool -> U) U
       | Ulet Bool (Qbit -> U) U
```

`UReturn` is simply an empty function, and will always be the last unitary in a sequence. For this section on 1-Qubit gates, the `Rot Qbit Rotation U` form is used. QIO defines all single qubit gates as rotations, where a `Rotation :: ((Bool, Bool) -> CC)` represents the 2x2 matrix of the gate. The `rot` function is then used to create a unitary function for a given `Rotation` and qubit.

```
rot :: Qbit -> Rotation -> U
rot x r = Rot x r UReturn
```

The Hadamard Gate Named after French mathematician Jacques Hadamard, the Hadamard gate (also called the Hadamard rotation) takes the base states $|0\rangle$ and $|1\rangle$ into equal superpositions

$$|0\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$$

$$|1\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$$

and has the matrix $\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$. It is equivalent to the combination of 2 rotations: π about the Z-axis, followed by $\pi/2$ about the Y-axis.

Figure 2: The Hadamard Gate

The gate matrix (i.e. `Rotation`) is given by `rhad` in the `QioSyn.hs` file, along with the unitary function `uhad` which can be used via the `applyU` function.

```
rhad :: Rotation
rhad (x,y) = if x && y then -h else h where h = (1/sqrt 2)

uhad :: Qbit -> U
uhad x = rot x rhad

applyHadamard :: QIO Bool
applyHadamard = do
    q <- mkQbit False
```

```
applyU (uhad q)
measQbit q
```

The Pauli- Gates The Pauli- gates, created by Wolfgang Pauli, correspond to a rotation of π about one of the 3 axis, and so there are 3 - Pauli-X, Pauli-Y and Pauli-Z (often abbreviated to X, Y and Z respectively). The Pauli-X gate is most common, as it is the equivalent of a classical NOT gate.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

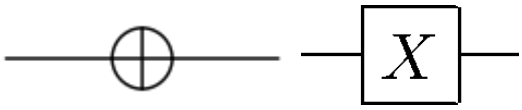


Figure 3: The 2 representations of the Pauli-X (NOT) Gate

The Pauli-X gate is defined in QIO using the `rnot Rotation` and the unitary `unot`. However, these gates can be created simply by first creating the corresponding `Rotation` matrix, and then using the `rot` function similar to how the Hadamard gate is defined.

```
rX :: Rotation
rX (x,y) = if x==y then 0 else 1

rY :: Rotation
rY (False, True) = 0 :+ 1
rY (True, False) = 0 :+ (-1)
rY (_, _) = 0

rZ :: Rotation
rZ (False, False) = 1
rZ (True, True) = -1
rZ (_, _) = 0
```

```
uX :: Qbit -> U
uX q = rot q rX

uY :: Qbit -> U
uY q = rot q rY

uZ :: Qbit -> U
uZ q = rot q rZ
```

1.6.2 n-Qubit Gates

Alongside the gates above, there are 3 other basic quantum gates; Controlled-X, SWAP and Toffoli, which are defined in QIO using the other 3 constructs for `U`.

The SWAP Gate This gate is extremely simply; it just swaps 2 qubits. In QIO, it is defined by the `swap` function.

```
swap :: Qbit -> Qbit -> U
swap x y = Swap x y UReturn
```

The Controlled-X Gate The most common 2-qubit gate, it applies the Pauli-X gate (i.e., a NOT gate) on the second qubit if and only if the first qubit is $|1\rangle$. It is often called the Controlled-NOT or cNOT gate. Note that we can do the same with the other Pauli- gates, however this is rarely used.

The gate is defined using the `Cond Qbit (Bool -> U) U` unitary construct, and the function `cond` which creates the unitary (similar to the functions `rot` and `swap` above). Essentially, when given a qubit and a unitary function which takes in a boolean, the resulting unitary will apply the boolean function to the states the qubit can be in.

This is best demonstrated with an implementation for the cNOT gate. If the qubit is measured to be `True` (i.e. in the $|1\rangle$ state), then the resulting unitary is a NOT gate. Otherwise, nothing will be done and the resulting unitary is empty.

```
cond :: Qbit -> (Bool -> U) -> U
cond x br = Cond x br UReturn

cnot :: Qbit -> Qbit -> U
cnot qc qo = cond qc (\x -> if x then (unot qo) else mempty)
```

The Toffoli Gate This gate acts on 3 qubits, and is often called the ccNOT gate. This is because it applies the Pauli-X gate on the third qubit if and only if the first two qubits are both $|1\rangle$.

```
toffoli :: Qbit -> Qbit -> Qbit -> U
toffoli q1 q2 qo = cond q1 (\x -> if x then (cnot q2 qo) else mempty)
```

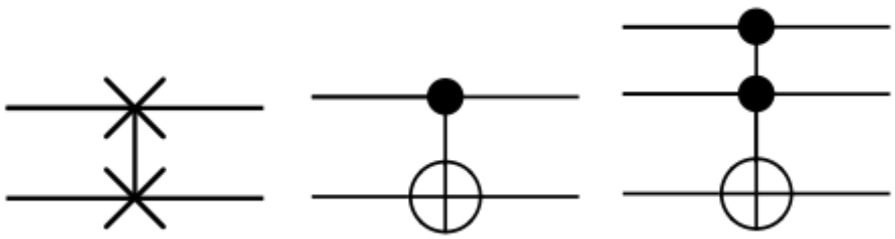


Figure 4: The SWAP gate (left), the Controlled-X (cNOT) gate (middle), and the Toffoli gate (right)

The C^n-NOT Gate The idea behind the Toffoli gate above (multiple control qubits, and one output qubit) can be extended to any arbitrary amount, creating a c^n-NOT gate. From the definition of the Toffoli gate in QIO above, it is clear that we can define a gate in QIO which acts on a list of control qubits, and one output qubit.

```
cNnot :: [Qbit] -> U
cNnot [] = mempty
cNnot (q:qs) = cond q (\x -> if x then (cNnot qs) else mempty)
```

However, this only works due to how QIO operates on qubits. On a quantum computer, a gate like this will require a sequence of toffoli gates and ancilla qubits. For example, the cccNOT gate requires an extra ancilla qubit:

```
cccNot :: Qbit -> Qbit -> Qbit -> Qbit -> Qbit -> U
cccNot q1 q2 q3 q4 qa =
    toffoli q1 q2 qa <>
    toffoli q3 qa q4 <>
    toffoli q1 q2 qa
```

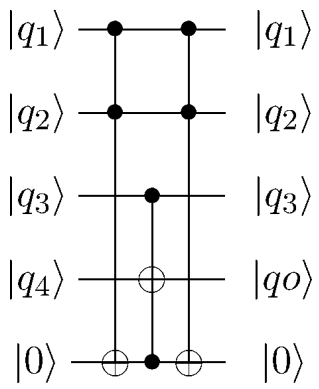


Figure 5: The cccNOT gate, made using toffoli gates and an ancilla qubit, where $|qo\rangle = |q_4 \oplus (q_1 q_2 q_3)\rangle$

When using ancilla qubits in QIO, we can use the unitary construct `Ulet` and the function `ulet`.

```
ulet :: Bool -> (Qbit -> U) -> U
ulet b ux = Ulet b ux UReturn
```

This construct will create a qubit in the state given by the boolean, for use in the given unitary. We can now execute the cccNot gate defined above. The qubit `q4` will be set to $|1\rangle$ if and only if `q1, q2` and `q3` are all in the state $|1\rangle$.

```
doCccNOT :: QIO Bool
doCccNOT = do
    q1 <- mkQbit True
    q2 <- mkQbit True
    q3 <- mkQbit True
    q4 <- mkQbit False
    applyU (ulet False (cccNot q1 q2 q3 q4))
    measQbit q4
```

1.7 Entanglement

Quantum entanglement is a phenomenon that occurs when the value of one qubit affects another, and forms a large part of many quantum algorithms. As mentioned above, when we measure

a qubit it collapses to one of its base states. If another qubit depends on this measured qubit (e.g., via a cNOT gate), then it will also collapse to one of the base states. The 2 qubits are then said to be entangled.

Consider a circuit which starts with two qubits *Alice* and *Bob*, both starting in the state $|0\rangle$. We then apply the Hadamard gate to *Alice*, converting *Alice* into an equal superpositional state of the base states $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, meaning there is a 50/50 chance of *Alice* being 0/1 when measured. Next, we apply a cNOT gate to *Alice* and *Bob*, entangling the two qubits together. Therefore, if we measure *Alice* to be $|0\rangle$, then *Bob* will collapse to the state $|0\rangle$, and the same will occur for the state $|1\rangle$. As entanglement is non-local, *Alice* and *Bob* could be separated by any distance, and this will still occur.

This can be implemented in QIO using the `uhad` and `cnot` defined earlier. The function `entangle` below will create 2 qubits, `qa` and `qb`, and initialize them in the $|0\rangle$ state. We then apply the Hadamard gate to qa, changing its state to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and a controlled-X unitary function, concatenated simply with the mappend operator `<>`. When run using the `run` function, `[True, True]` or `[False, False]` will be returned due to the entanglement. The `sim` function will return `([True, True], 0.5), ([False, False], 0.5)`.

```
entangle :: QIO [Bool]
entangle = do
    qa <- mkQbit False
    qb <- mkQbit False
    applyU (uhad qa <> cnot qa qb)
    ba <- measQbit qa
    bb <- measQbit qb
    return [ba, bb]
```

1.8 The Bell States

The two qubits shown above, *Alice* and *Bob*, together form a Bell state, named after physicist John S. Bell

$$|\Psi_{00}\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

As evident from the equation, if one of the qubits is measured to be $|0\rangle$ or $|1\rangle$, then so is the other. There are 4 Bell states, where each state has 2 entangled qubits, and the measurement of one qubit collapses the state of the other.

$$|\Psi_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \quad |\Psi_{01}\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$

$$|\Psi_{10}\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) \quad |\Psi_{11}\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$$

Each Bell state is created using the same unitary operation, namely a Hadamard gate on the first qubit, and then a cNOT to the second qubit using the first qubit as the control bit, with the resulting Bell state determined by the original state of the 2 qubits.

2 Motivations

2.1 Problems

A quantum computer consists of multiple two-level quantum-mechanical systems, such as multiple electrons, with each electron represents a qubit. The spin of the electron determines the qubit state, with spin up representing $|0\rangle$ and spin down representing $|1\rangle$. The main problem faced here is decoherence [SP05]. The overall system is extremely sensitive to interaction with the surroundings, such as magnetic fields affecting electrons, and must be isolated from its environment. The more qubits we have, the harder it is to maintain coherence.

Due to the difficulties of creating a quantum computer, we need to be able to simulate one on a classical computer. This simulation will not be as efficient as a quantum computer, and so its main use is the testing and creation of quantum algorithms, alongside teaching quantum computing. This means there is a high demand in quantum computing simulators, hence the need for projects like this.

2.2 Simulators

As mentioned above, creating a stable and effective quantum computer is difficult, and so while new ways of implementation are being researched, we can use quantum simulators to create, test and teach quantum computing. As quantum computing grows even larger, simulators need to be available to people in order to help them learn about quantum computations and programs. More quantum simulators across different programming languages will supply a wider range of programmers the ability to practice quantum computing on their own computer, as well as offer different methods of implementing a quantum simulator.

This project aims to add to this group of quantum simulators, offering an up-to-date and reliable Haskell package which enables users to create and run quantum programs. On top of this, they will be able to run quantum algorithms over their own data sets, increasing the teaching ability of the project.

3 Related Work

3.1 IBM Q

The IBM Q Experience is an online platform [ibmd] - found easily with a Google search "ibm q" - which gives its users access to a set of prototype quantum processors. Users can create quantum circuits or programs via a graphical or textual interface, and then either run these quantum operations on an actual quantum computer, or simulate them classically.

It was launched by IBM in 2016 [New16], and started with a 5 qubit quantum processor and matching simulator, alongside a small set of two-qubit operations. Over time, the platform has evolved into a strong stable quantum platform, with the addition of more 2-qubit operations, a simulator over a maximum of 20 qubits and limited beta access to a 16 qubit quantum processor. IBM also released a python-based quantum computing framework, QISKit [Lub17],

allowing users to simulate a quantum computer offline, or run quantum programs on the real IBM quantum processors online.

3.1.1 Evaluation

In my opinion, the IBM Q Experience platform is great for all who wish to create quantum programs and just generally experiment with quantum computing. A colourful, intuitive graphical interface can be used to build quantum circuits and then run/simulated with ease - perfect for new users. This still works well for more advanced users, however the textual interface is more suited for advanced use. Complex quantum programs can be created instead of circuits, and the use of the QISKit framework enables users the option of creating and simulating quantum computations without the online platform. However, this does mean users must learn a new programming language, potentially alongside learning about quantum computations, creating quite a large knowledge gap to bridge. Yet this is somewhat diminished by the availability of two user guides, a Beginner's guide [ibmb] and a Full guide [ibmc].

These guides successfully reduce the difficulty in learning the basics of quantum computing without confusing newcomers. The Beginner's guide briefly explains the qubit, and common single qubit gates. It then goes on to hint at some multi qubit gates, and explain the process of entanglement and how it can be used. The Full guide extends these sections, providing a greater depth of knowledge concerning the qubit, such as the Bloch sphere and decoherence, while going into more detail on single and multi qubit gates, with descriptions of quantum algorithms and some quantum error correction methods. More help is then available to users via a community forum [ibma] and in-depth QISKit tutorials [ibme], ensuring users can find solutions to any problems they are having.

3.1.2 Key Points

1. Simple and effective graphical user interface, allowing the user to drag-and-drop quantum gates onto a circuit, with bright colours used to emphasize different gates.

2. Basic textual interface, providing users with an interface they can use to build more complex quantum programs.

3. Multiple user guides to help teach newcomers without confusing them, while still providing aid to more advanced users.

4. Emphasis placed on usability.

4 Project Description

4.1 Aim

The aim of this project is to update and improve an out-of-date Quantum computing package in Haskell, QIO [qio16] - developed by Green and Altenkirch. This will result in an updated, stable package for Haskell v8.4.3, allowing users to simulate a quantum computer, and run

various quantum algorithms. It will also provide the user a graphical interface to build and run quantum circuits.

This QIO Haskell package allows a user to create a quantum program, and then run or simulate it. By running, any qubits measured via a function `measQbit :: Qbit -> QIO Bool` will be collapsed into True or False based on their probability amplitudes and a pseudorandom number generator. However, the difference between this package and others is revealed when we simulate the program instead. Then, any qubits measured will be shown as their probabilities, including multiple qubits together. From this we can easily check the program for errors.

The package is currently not working in Haskell v8.4.3, and so first needs to be updated. The update will need to maintain the quantum aspects necessary for quantum computation, such as coherence and the no-cloning theorem, and still be able to run/simulate quantum programs inputted.

Following this, two quantum algorithms (Grover's and Shor's) will be implemented in the package, allowing users to run these algorithms on their own data. There is already an implementation of Shor's algorithm present in QIO, but this needs to be updated. Each implementation will be in a separate Haskell file, and be able to perform a quantum algorithm correctly over its inputs using the rest of the QIO package.

Finally, a quantum circuit builder/interpreter will be built. It will display a graphical interface, and allow the user to drag-and-drop quantum gates onto a circuit of any amount of qubits. It will then display the resulting amplitude(s) and/or value(s) of any of the qubits. This will be similar to the IBM Q Program specified above, only shall be run on a classical computer instead of a quantum one.

4.2 Algorithms

4.2.1 Grover's algorithm

In 1996, Lov K. Grover demonstrated an algorithm which could search a quantum mechanical database quickly [Gro96]. This quantum algorithm is quadratically faster than the fastest classical variant, returning the index of the search item in $O(\sqrt{n})$, instead of the classical $O(n)$. It can also be generalized to find solutions to an unknown predicate function f in the same time. This means that any problem which involves finding solutions to a boolean function can be solved using Grover's algorithm, and so it has a large variety of uses past searching databases, such as finding the mean and median of a set of numbers, or solving the collision problem in $O(n^{1/3})$[Aar05] instead of the classical $\Theta(\sqrt{n})$

The algorithm takes a function f, and after converting it into a unitary function, runs it over a set of qubits representing the data set to search. It then negates solution states in the solution set, and disperses them. This dispersion acts as an inversion around the mean amplitude, meaning the solution state is amplified due to the previous negation. After repeating this a number of times no greater than $\pi/4\sqrt{N}$ times, where N is the amount of variables in the function, there is a high probability that the qubits, when measured, will give a solution to the function f.

This project aims to explore potential methods of implementating this algorithm in the QIO Haskell package, as well as demonstrate other ways of executing the algorithm.

4.2.2 Shor's algorithm

Before Grover's algorithm, there was Shor's algorithm for integer factorization. This quantum algorithm is used to find the factors of any integer N in $O((logN)^3)$ [BCDP96]. The algorithm specifically looks at the case of factorizing an integer $N = pq$, where p and q are both large primes. It claims that if an integer b shares no factors with N then $b^r = 1 \pmod N$ for some integer r, and the function $f(x) = b^x \pmod N$ is a periodic function of x with period r.

Finding b is simply done by taking a random number co-prime to N. If the period r is then even, we can then calculate

$$x = b^{\frac{r}{2}} \pmod N$$

Following this, if $x+1 \neq 0 \pmod N$ we can factorize the input. The probability of choosing a random b which gives a period r with these properties is $> 50\%$ [Gre09]Shor's algorithm continued, page 27.

After constructing a unitary function $U_{f(x)} = b^x \pmod N$, the algorithm applies it over an input state, leaving an equal superpositional state

$$\frac{1}{\sqrt{2^n/2}} \sum_{x=0}^{2^n-1} |x\rangle_n \, |b^x \pmod N\rangle_{n_0}$$

$$n_0 = \log_2 N, \quad n = 2n_0$$

We can then measure the second register, which will give any value for $b^x mod N$, leaving the first register in an equal superposition over the values for x. The algorithm then applies the Quantum Fourier Transformation (QFT) to the first register and measures it, giving a value for y where $y = 2^n/r$. Finally, we can check if r is even and if $x + 1 \neq 0 \pmod N$ where $x = b^{r/2} \pmod N$, and if they are both true, then N has factors $(x - 1)$ and $(x + 1)$. We can then find the greatest common denominators of both in N, and return prime factors of N.

5 The QIO Package

5.1 Underlying Structure

QIO currently uses a HeapMap to store the states of the qubits in the system, and a Qbit type which just acts as an integer reference to this map. Unitary functions take the form of a monoid `Unitary`, and represent an operation on the HeapMap which may produce a new state. We can then restrict unitary functions to be in a few distinct forms, ensuring the functions are indeed unitary - a necessity for quantum computations. These forms are defined in a datatype `U` shown below.

```
data U = UReturn
       | Rot Qbit Rotation U
       | Swap Qbit Qbit U
       | Cond Qbit (Bool -> U) U
       | Ulet Bool (Qbit -> U) U
```

Using recursion, this type dictates is what a unitary operation will be; a string of rotations, swaps, conds and lets, ending in a UReturn. From these 4 operations, we can build any unitary operation possible, and as `U` is also defined as a monoid, we can easily concatenate operations together. Following this, the package defines another datatype `QIO a` - the type of a quantum computation:

```
data QIO a = QReturn a
           | MkQbit Bool (Qbit -> QIO a)
           | ApplyU U (QIO a)
           | Meas Qbit (Bool -> QIO a)
```

This datatype enables us to create and measure Qbits, along with applying unitary operations to them.

From all this, we can build a quantum program which creates qubits in an initial base state (i.e., either $|0\rangle$ or $|1\rangle$), applies any amount of unitary functions to them, and then measures qubits. Now we need to let the user execute a quantum program on a simulated quantum system. To do this, QIO has 2 functions: `run :: QIO a` $\rightarrow$ `IO a` and `sim :: QIO a` $\rightarrow$ `Prob a`.

Run takes a quantum computation in the type defined above, and executes it on a simulated quantum system. When qubits are measured and returned in the computation, the function prints this state value to the console using the IO monad. The state value to return will originally be a quantum state, and so is made up of probability amplitudes. Therefore, the actual state to return will be chosen at random based on these probability amplitudes and a psuedorandom number generator.

Sim, however, is a bit different. It still takes a quantum computation and executes it, however it returns the actual probabilities of the possible states the qubits measured may be in. This makes testing quantum programs fairly simple, drastically increasing the usability of the project.

5.2 Updating QIO

5.2.1 Current Errors

Currently, QIO cannot be installed on the latest Haskell version (v8.4.3). This is because the package depends on the Haskell base package version being ≥ 4.9 and < 4.10. After downloading the package, and manually changing this to allow any base package ≥ 4.9, two errors are shown depicting what is incorrect with the current QIO structure.

```
QIO/QioSyn.hs:39:10: error:
        * No instance for (Semigroup U)
            arising from the superclasses of an instance declaration
        * In the instance declaration for ‘Monoid U’
...
QIO/QioSynAlt.hs:118:10: error:
        * No instance for (Semigroup U)
```

```
            arising from the superclasses of an instance declaration
        * In the instance declaration for ‘Monoid U’
```

From the error messages presented, it is clear that a change to Monoids is causing errors. In the update to base-4.11.0.0, the Haskell Semigroup class was made a superclass of Monoid [bas]. This means that a Semigroup instance for `U` needs to be defined, alongside the Monoid instance.

The current definition for `U` in QioSyn.hs is shown below

```
instance Monoid U where
        mempty = UReturn
        mappend UReturn u = u
        mappend (Rot x a u) u' = Rot x a (mappend u u')
        mappend (Swap x y u) u' = Swap x y (mappend u u')
        mappend (Cond x br u') u'' = Cond x br (mappend u' u'')
        mappend (Ulet b f u) u' = Ulet b f (mappend u u')
```

All that needs to be done to fix this is to include a Semigroup definition for `U` which defines the structure for executing multiple unitary functions, and then to define mempty and mappend for the Monoid instance.

```
instance Semigroup U where
        UReturn <> u
        (Rot x a u) <> u' = Rot x a (u <> u')
        (Swap x y u) <> u' = Swap x y (u <> u')
        (Cond x br u') <> u'' = Cond x br (u' <> u'')
        (Ulet b f u) <> u' = Ulet b f (u <> u')

instance Monoid U where
        mempty = UReturn
        mappend = (<>)
```

After this, the alternate file QioSynAlt.hs needs to be updated also. The definition for `U` is very similar, however uses a non-recursive datatype in its definition.

```
instance Monoid U where
        mempty = Fx UReturn
        mappend (Fx UReturn) u = u
        mappend (Fx (Rot x a u)) u' = Fx $ Rot x a (mappend u u')
        mappend (Fx (Swap x y u)) u' = Fx $ Swap x y (mappend u u')
        mappend (Fx (Cond x br u')) u'' = Fx $ Cond x br (mappend u' u'')
        mappend (Fx (Ulet b f u)) u' = Fx $ Ulet b f (mappend u u')
```

The fix for this is very similar to before:

```
instance Semigroup U where
        (Fx UReturn) <> u = u
        (Fx (Rot x a u)) <> u' = Fx $ Rot x a (u <> u')
        (Fx (Swap x y u)) <> u' = Fx $ Swap x y (u <> u')
        (Fx (Cond x br u')) <> u'' = Fx $ Cond x br (u' <> u'')
        (Fx (Ulet b f u)) <> u' = Fx $ Ulet b f (u <> u')

instance Monoid U where
        mempty = Fx UReturn
        mappend = (<>)
```

5.2.2 New Errors

After updating the files, the package produces 2 new error messages when attempting to build, and the previous errors do not show. The same errors are occuring (lack of a Semigroup instance) but for different files, QioClass.hs and Qio.hs.

```
QIO/QioClass.hs:17:10: error:
        * No instance for (Semigroup UnitaryC)
            arising from the superclasses of an instance declaration
        * In the instance declaration for ‘Monoid UnitaryC’
...
QIO/Qio.hs:32:10: error:
        * No instance for (Semigroup Unitary)
            arising from the superclasses of an instance declaration
        * In the instance declaration for ‘Monoid Unitary’
```

Therefore, the fixes are the same as before: create a definition for a Semigroup instance based on the Monoid instance, then define mempty and mappend in the Monoid definition.

```
QioClass.hs:
instance Semigroup UnitaryC where
         (U f) <> (U g) = U (\ fv h -> g fv (f fv h))

instance Monoid UnitaryC where
        mempty = U (\ fv bs -> bs)
        mappend = (<>)

Qio.hs:
instance Semigroup Unitary where
        (U f) <> (U g) = U (\ fv h -> unEmbed $ do h' <- Embed $ f fv h
                                                  h'' <- Embed $ g fv h'
                                                  return h'')

instance Monoid Unitary where
```

```
mempty = U (\ fv h -> unEmbed $ return h)
mappend = (<>)
```

5.2.3 Testing

After updating QioClass.hs and Qio.hs, the package builds and installs successfully onto Haskell v8.4.3. *Note: The package still installs and runs correctly on the newest version of Haskell, namely v8.6.3.* Running various quantum programs using the updated package gives correct results, and so the package still works as intended after being updated, and so all that is left to do is change the base dependancy of the package to ≥ 4.11.

6 Algorithms

During the following two sections, namely those relating to Grover's algorithm and Shor's algorithm, I will attempt to fully describe how the algorithms work while simultaneously implementing them in Haskell using the QIO package, in order to improve understanding and readability.

6.1 Grover's Algorithm

As stated earlier in section 4.2.1, Grover's algorithm can be used to search a quantum database in $O(\sqrt{n})$, and can be extended to solve any blackbox boolean function. This extension reveals the numerous uses Grover's algorithm has, and so will be the main focus for implementation. Allowing the user to input a custom function will abstract them away from the algorithm's complexity, enabling simple testing over multiple use cases - hopefully improving understanding.

In order to find a solution to a blackbox function - which returns 1 for a solution input, and 0 otherwise - a classical computer must repeatedly apply the function to random integers in order to find the solution, and so is $O(N)$. However, a quantum computer can find a solution to the function after applying it no greater than $\pi/4\sqrt{N/s}$ times - where N is the amount of variables in the function, and s is the amount of solutions to the function - with a probability of success close to 1 when N is large. While the algorithm works for all functions which produce either 0 or 1 based on the input, it is often still thought as just an algorithm which searches through an unsorted database faster than any classical method.

Put simply, Grover's algorithm takes in a blackbox function, creates a superposition over all possible function inputs and then emphasizes states which are not solutions. After doing this roughly $\pi/4\sqrt{N/s}$ times, there is a high probability that a solution state is found. The algorithm can be split up into multiple explicit parts:

1. **The Oracle** - Creating a unitary function based on the blackbox function, and applying it to a set of qubits.

2. **Diffusion** - Increasing the amplitude of solutional state(s).

3. **Measurement** - Measuring specific qubits to get a solution.

6.1.1 The Oracle

The blackbox function f returns 0 or 1 based on if the inputted value (i.e. an n-bit integer) x is the solution a;

$$f(x) = 0, \quad x \neq a \qquad f(x) = 1, \quad x = a.$$

This function can be represented in the form of a unitary function U_f, which acts on an n-qubit register containing x, and a single qubit register containing the result $f(x)$.

$$U_f(|x\rangle_n |y\rangle) = |x\rangle_n |y \oplus f(x)\rangle$$

A simple example of this would be a unitary function which returns 1 when 0010 is inputted, and 0 otherwise:

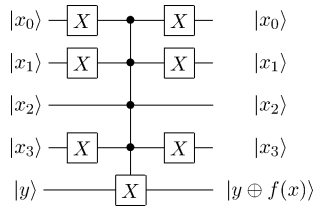


Figure 6: A unitary function, setting the last qubit to 1 when 0010 is inputted, and 0 otherwise

This can be modified so that the overall state's sign is changed if $x = a$ by first setting the output qubit to $|1\rangle$ and applying the Hadamard gate to it before the application of U_f. So, the output qubit is in the state $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$ before application, and after applying U_f the overall state becomes

$$U_f(|x\rangle \oplus |-\rangle) = (-1)^{f(x)} |x\rangle \oplus |-\rangle$$

From this equation it is obvious that applying U_f to the state is the same as doing nothing to the single output qubit, while applying a unitary transformation V to the input register, where

$$V|x\rangle = (-1)^{f(x)} |x\rangle = \begin{cases} |x\rangle, & x \neq a, \\ -|a\rangle, & x = a. \end{cases} \tag{1}$$

If we initially transform the n-qubit input register into a uniform superposition of all possible inputs, i.e.

$$|\phi\rangle = H^{\otimes n} |0\rangle_n = \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle_n$$

Then, after applying V, the component of the state along $|a\rangle$ (i.e. the state we are looking for) will have a negative phase. The unitary V can therefore be written as

$$V|\Psi\rangle = |\Psi\rangle - 2|a\rangle\langle a|\Psi\rangle \qquad \therefore V = 1 - 2|a\rangle\langle a|$$

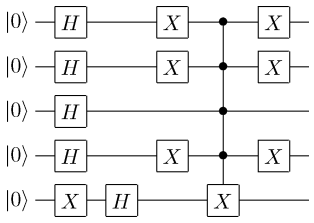


Figure 7: Applying V to an equal superposition of all possible inputs.

Implementation Before actually implementing the oracle used for Grover's algorithm in Haskell, it is useful to first define a few functions to work over lists of qubits - allowing easy expansion to custom oracles later on.

```
mkQbits :: Int -> Bool -> QIO [Qbit]}
mkQbits n b = mkQbits' n b []
  where
    mkQbits' 0 _ qs = return qs
    mkQbits' n b qs = do
      q <- mkQbit b
      mkQbits' (n-1) b (qs ++ [q])
```

As you can see, `mkQbits` is a function which uses recursion to create a list of `n Qbits` in the state given by `b`. This leads on to the creation of another function, `measQbits :: [Qbit] -> QIO [Bool]` which returns a list of boolean values representing the measurements of the inputted list of `Qbits`. The only two remaining functions necessary pertain to applying unitary functions to lists of `Qbits`. More specifically, we need a function which applies a single-qubit unitary operation to all qubits in a list, and another function which applies a single-qubit unitary operation to the last qubit in a list, conditional on the state of each other qubit in said list.

```
unitaryN :: (Qbit -> U) -> [Qbit] -> U
unitaryN _ []      = mempty
unitaryN uf (q:qs) = uf q <> unitaryN uf qs

condN :: (Qbit -> U) -> [Qbit] -> U
condN uf (q:[]) = uf q
condN uf (q:qs) = cond q (\x -> if x then (condN uf qs) else mempty)
```

Now that these functions are created, initializing the qubits and applying the oracle is simple. First, we need to create i qubits for each variable in the function and put them in the state $|+\rangle$, and then create o qubits for each "line" in the function, each put in the state $|-\rangle$.

```
initialize :: Int -> Int -> QIO ([Qbit], [Qbit])
initialize i o = do
  qI <- mkQbits i False
  qO <- mkQbits o True
```

```
  applyU (unitaryN uhad (qI ++ qO))
  return (qI, qO)
```

Now we need to create a function representing V, the oracle unitary. For now, this will be for the example above, searching for the state $|110\rangle$.

```
oracle :: [Qbit] -> [Qbit] -> U
oracle qI qO =
  unot (qI!!2) <>
  condN unot (qI ++ [qO!!0]) <>
  unot (qI!!2)
```

6.1.2 Diffusion

Grover's algorithm requires one more unitary W, which does not depend on the oracle function. This unitary transformation changes the sign of the component orthogonal to $|\phi\rangle$, i.e.

$$W = 2\,|\phi\rangle\,\langle\phi| - 1$$

The unitary $-W$ works fine here, as the final state will only differ by an overall minus sign, if it differs at all. From $-W$, and the fact that the Hadamard gate is its own inverse, you can see that we need a gate which does nothing for all states apart from $|00...00\rangle$, which it multiplies by -1.

$$-W = 1 - 2\,|\phi\rangle\,\langle\phi|\,, \qquad |\phi\rangle = H^{\otimes n}\,|0\rangle_n$$

$$\therefore -W = H^{\otimes n}(1 - 2\,|00...00\rangle\,\langle 00...00|)H^{\otimes n}$$

The Pauli-Z gate is the main gate to use here. This gate, with matrix $\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$, multiplies the state $|11...11\rangle$ by -1, and does nothing on all other states. From this, it is easy to see that if we applied a controlled Pauli-Z gate to the qubits, surrounded by n Pauli-X gates, we get the unitary $-W$.

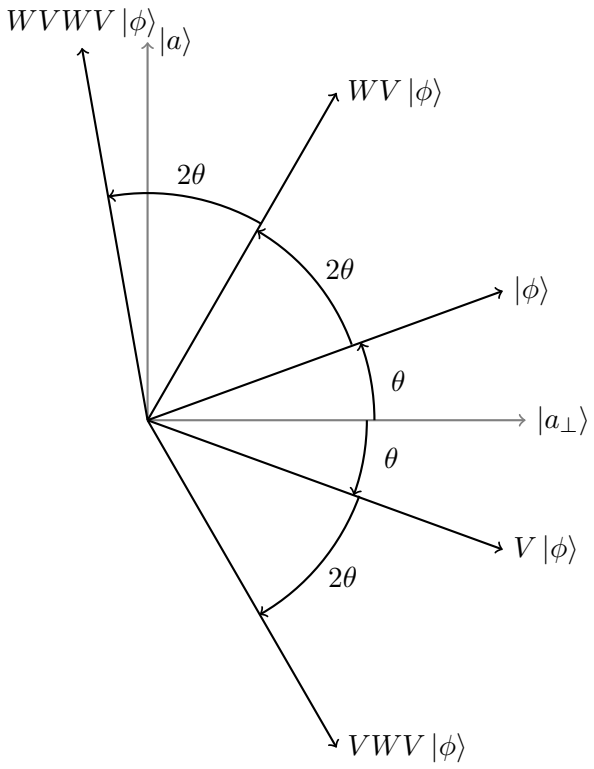
$$-W = H^{\otimes n}X^{\otimes n}(c^{n-1}Z)X^{\otimes n}H^{\otimes n}$$

Implementation Implementing this is extremely simple using the functions created earlier, as shown below

```
diffuse :: [Qbit] -> U
diffuse qI =
  unitaryN uhad qI <>
  unitaryN unot qI <>
  condN (\q -> uphase q pi) qI <>
  unitaryN unot qI <>
  unitaryN uhad qI
```

6.1.3 Application & Measurement

All that is left now is to apply WV repeatedly to the intital state $|\phi\rangle$ enough times for a solution state to be found. This is best described using simple geometry.



The intial state $|\phi\rangle$ will contain one state $|a\rangle$ with amplitude $\frac{1}{2^{n/2}} = 1/\sqrt{N}$, and so starts out at an angle θ from $|a_\perp\rangle$, where $\sin\theta = 1/\sqrt{N}$.

The unitary V negates the state $|a\rangle$ in $|\phi\rangle$, and so on the graph, it reflects $|\phi\rangle$ about the axis $|a_\perp\rangle$. Next, the unitary W is applied which negates any state orthogonal to $|\phi\rangle$ (i.e. any negative state in $|\phi\rangle$, notably the state $|a\rangle$). Graphically, this means the unitary W reflects about $|\phi\rangle$, and so when applied to $V\,|\phi\rangle$, it results in a state at an angle of 3θ above $|a_\perp\rangle$. Therefore, each iteration (application of the unitary V followed by W) rotates the qubits' state by 2θ anticlockwise, giving us a simple method to find out the amount of iterations, i, required. Note: when N is large, $\sin\theta \approx \theta$.

$$\theta + 2i\theta \approx \pi/2 \qquad \sin\theta = 1/\sqrt{N} \qquad \therefore \theta \approx 1/\sqrt{N}$$

$$\therefore i \approx \frac{\pi - 2\theta}{4\theta} \approx \frac{\pi}{4}\sqrt{N}$$

The amplitude of the state $|a\rangle$ over the qubits is given by the sine of the angle between the qubits and $|a_\perp\rangle$, and so the probability of measuring the solution from the qubits is equal to the square of this.

Example Say we want to search for $|110\rangle$ across a space of 8 (i.e. 3 qubits). The iterations required will therefore be $\approx \frac{\pi}{4}\sqrt{8} \approx 2$. The oracle function will perform a controlled-X gate on its output qubit if its inputs are in the state $|110\rangle$. Then, we need to diffuse the qubits using the unitary W described above. After initializing the qubits, we must apply V, then W, and repeat this again to get a solution.

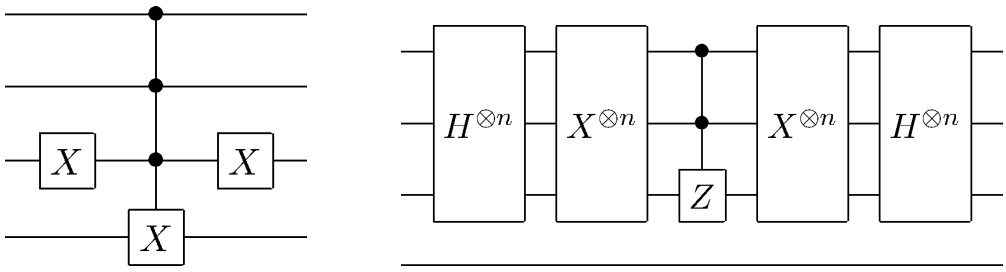


Figure 8: The 2 unitaries V (left) and W (right) used to search for $|110\rangle$.

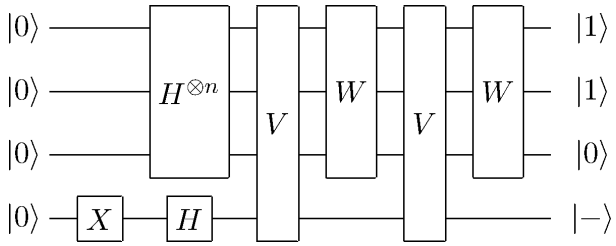


Figure 9: The circuit used to perform Grover's algorithm over 3 input qubits, with solution $|110\rangle$.

Implementation All that remains in the implementation is to combine the functions described above and apply them the correct amount of times (i.e. twice when searching for the state $|110\rangle$).

```
grover :: QIO [Bool]
grover = do
  (qI, q0) <- initialize 3 1
  let groverStep = oracle qI q0 <> diffuse qI
  applyU (groverStep)
  applyU (groverStep)
  measQbits qI
```

Testing Testing this is relatively simple: simply `run` the `grover` function, check the result and then `sim` the function, checking the probability of expected results.

```
run grover -> [True, True, False]
sim grover -> [..., ([True, True, False], 0.9453125), ...]
```

with all other states having probability $7.812...e^{-3}$

6.1.4 Expansion to multiple solutions

The algorithm above can also search for multiple solutions, the only thing that changes is the amount of iterations necessary. If there are s solutions, then the amount of iterations required

becomes $i \approx \frac{\pi}{4}\sqrt{\frac{N}{s}}$.

6.1.5 Implementing a Custom Oracle

In the example above, the oracle is searching for `True, True, False`, i.e. $x1 \wedge x2 \wedge \neg x3$. So, if the program takes in a function as a boolean predicate, it can create a unitary function representing the oracle using a list of the input variables in the predicate, and applying 2 NOT gates to negated variables.

The oracle can also search for multiple lines of predicates separated by OR operations; simply have a different output qubit for each line, so that solutions to every line will be negated, allowing easy usage for multiple solutions. For example, if the user wants to apply the algorithm over a function f, where

$$f(x_n) = (x1 \wedge x2 \wedge x3) \vee (\neg x1 \wedge x2 \wedge \neg x3)$$

then they can input

```
1 2 3
-1 2 -3
```

So the oracle function needs to be changed to take in a list of a list of `Int`s, i.e. `[[Int]]`, where each inner list represents a line of the function. From this, it will recursively apply the NOT gates over qubits referenced by negative integers, and apply controlled-X gates over the qubits referenced in each line to the output qubits, where each inner list is applied to its own unique output qubit.

```
oracle :: Int -> [[Int]] -> [Qbit] -> [Qbit] -> U
oracle _ [] _ _ = mempty
oracle c (p:ps) qI q0 =
  unitaryN unot (unots p qI) <>
  condN unot (conds p qI ++ [q0!!c]) <>
  unitaryN unot (unots p qI) <>
  oracle (c+1) ps qI q0
  where
    unots p qI = map (\x -> qI!!((abs x)-1)) $ filter (< 0) p
    conds p qI = map (\x -> qI!!((abs x)-1)) p
```

The program also needs to know how many solutions there are in order to iterate the application of V and W the correct amount of times, roughly equal to $\frac{\pi}{4}\sqrt{\frac{N}{s}}$. The application can then be done using the `foldr` function, folding with the `mappend` operator `<>`, starting from `mempty`.

The entry point to the algorithm will be using the `main` function in the code. This function, of type `IO ()`, will take in a custom function inputted in the form described above, and then run the algorithm over it. This can also be timed effectively using the `TimeIt` package [Aug09].

6.1.6 Testing

In order to test this, the program needs to be changed slightly to `sim` the algorithm, and print out the probabilities. The above predicate $x1 \wedge x2 \wedge \neg x3$, inputted as `1 2 -3` returns the expected result:

`[...,([True, True, False]), 0.9453125),...]`
with all other states having probability $7.812...e^{-3}$

A more complicated predicate shown below also returns the correct result, indicating the implementation works correctly.

$$(x1 \wedge \neg x2 \wedge x3 \wedge x4) \vee (x2 \wedge \neg x3 \wedge x4) \vee (x1 \wedge x2 \wedge x3)$$

Solutions: $0101, 1011, 1101, 1110, 1111$
Output:

```
[([True, True, True, True], 0.1914...),
([True, True, True, False], 0.1914...),
([True, True, False, True], 0.1914...),
                ...,
([True, False, True, True], 0.1914...),
                ...,
([False, True, False, True], 0.1914...),
...]
```

with all other states having probability $3.906...e^{-3}$

6.1.7 An Improvement

This algorithm can be altered to actually not use any ancilla qubits in QIO (due to the way QIO executes multi-qubit gates) and use less gates, improving run time. The current implementation uses ancilla qubits to negate solution states in the input qubits. However, this can simply be done using a controlled Pauli-Z gate over the inputs, when the input qubits are initially in the state $|-\rangle$. After this, the diffusion is also simplified to become an inversion about the mean [Gid13]; decrease the amplitude of non-solutional states about the mean amplitude while increasing the amplitude of solutional states. This can be achieved by $n-1$ Hadamard gates surrounding a C^n-NOT gate.

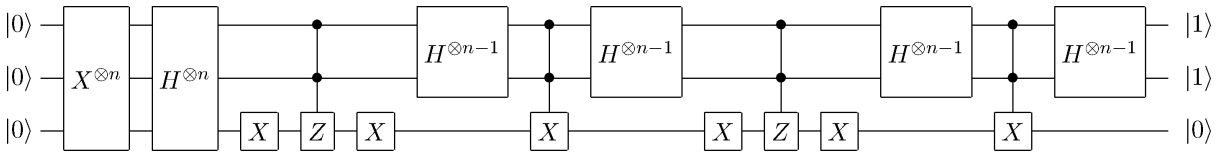


Figure 10: Improved circuit for Grover's algorithm, searching for the state $|110\rangle$

In order to change the code to implement this, the `initialize`, `oracle` and `diffuse` functions need to be altered.

6.1.8 Testing Solutions

The current algorithm works well when solving a function with a known amount of solutions, and returns correct solutions with a high probability as shown. However, the implementation can be changed to measure whether or not the input qubits contain a solution at the end, This will enable the program to return definite, distinct solutions and find solutions to functions with an unknown amount of solutions.

By simply using ancilla qubits to test the state of input qubits with C^n-NOT gates, a final ancilla qubit will be in the state $|1\rangle$ if the input qubit's contain a solution.

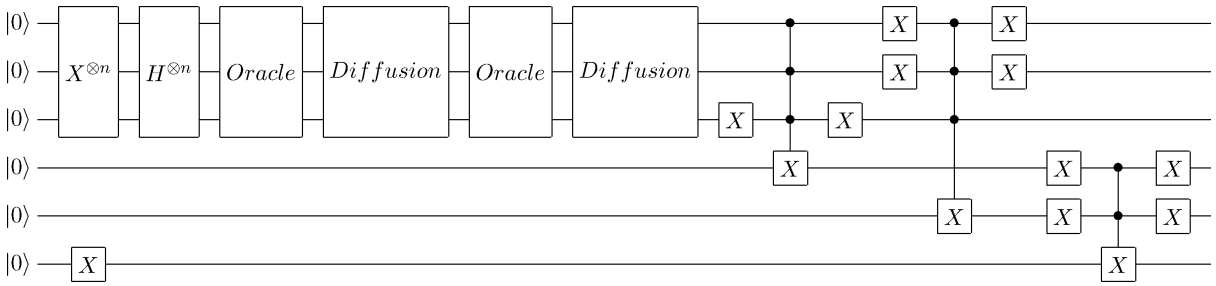


Figure 11: Improved circuit for Grover's algorithm, searching for the state $|110\rangle$ and $|001\rangle$, while also checking n qubits for the solution.

The results of each line of the function (i.e. $x0 \wedge x1 \wedge \neg x2$ and $\neg x0 \wedge \neg x1 \wedge x2$) are saved into two separate qubits. Following this, a third ancilla qubit is set to $|0\rangle$ if both the two ancilla qubits are in the state $|0\rangle$, which would only occur when the input qubits do not contain a solution.

Implementation In order to implement this, the `initialize` function will need create the necessary ancilla qubits, all of which will be in the state $|0\rangle$ except for the last, which will be in the state $|1\rangle$. A new function needs to be created to apply the necessary C^n-NOT gates over the input qubits on distinct ancilla qubits, which will be applied after the algorithm has been executed. Finally, the program will measure the last ancilla qubit in the list, and return this with the measurements of the input qubits. The ancilla measurement can then be checked to determine if a solution was found, allowing easy looping to get multiple, unique and definite solutions.

This will also enable the algorithm to be used for functions with an unknown amount of solutions, as this requires multiple executions of the algorithm with differing amounts of iterations until a solution is found. In this case, the amount of iterations starts as $\frac{\pi}{4}\sqrt{N}$. If this does not result in a solution, the algorithm is executed again with $\frac{\pi}{4}\sqrt{\frac{N}{2}}$ iterations, and then $\frac{\pi}{4}\sqrt{\frac{N}{4}}$ iterations, and so on until a solution is found. The total amount of iterations required is still $O(\sqrt{N})$ as shown below.

$$\frac{\pi}{4}\sqrt{N}\left(1+\frac{1}{\sqrt{2}}+\frac{1}{2}+\frac{1}{2\sqrt{2}}+...\right)=\frac{\pi}{4}\sqrt{N}\sum_{i=0}^{\infty}\frac{1}{2^i}=\frac{\pi}{4}\sqrt{N}(2+\sqrt{2})$$

6.1.9 Probability

We can also print out the probability of finding the solution by running the `sim` function over a function which only measures the last ancilla qubit (which will only be measured to be True if the input qubits contain a solution).

```
probability :: Int -> Int -> [[Int]] -> IO ()
probability n i ps = putStrLn $ show $ sim $ probOfSolution
  where
    probOfSolution = do
      (qI, qA) <- initialize n ((length ps)+1)
      let groverStep = oracle ps qI <> diffuse qI
      applyU (foldr (<>) mempty (replicate i groverStep))
      applyU (testForSolution 0 ps qI qA)
      measQbit $ last qA
```

6.2 Shor's Algorithm

As mentioned in section 4.1, there is already an implementation of Shor's algorithm in QIO, in the file `Shor.hs`. Therefore, this section will be dedicated to explaining the current implementation, and describing any potential improvements.

In section 4.2.2, it was stated that Shor's algorithm is for integer factorization. This is not exactly correct, as the algorithm actually finds the period of a function, which can then be expanded to factorize integers. Let f be a function, periodic on addition, over integers, i.e.

$$f(x) \equiv f(y) \iff x = y + cr, \qquad c \in \mathbb{Z}$$

Classical algorithms for finding the period of such a function take a time which grows faster than any power of the number of bits of r. However, Shor's algorithm does this in a time which scales slightly faster than n^3 [Mer07]Page 63.

Say we have a periodic function of x with period r, where N is the product of 2 primes, p and q.

$$b^r = 1 \pmod N, \qquad r \in \mathbb{Z}, \qquad N = pq$$

$$\therefore f(x) = b^x \pmod N \implies f(x+s) = f(x) \Leftrightarrow s = cr, \qquad c \in \mathbb{Z}$$

In order to find a value for b, we all the numbers coprime with N, done simply using Euclid's algorithm. Next, we pick a random value for b from the numbers found, and check which gives the required properties for the period r, with a probability of $> 50\%$[Gre09]Shor's algorithm continued, page 27. The necessary properties are:

1. r needs to be even

   If r is even, then we can calculate

   $$x = b^{\frac{r}{2}} \pmod N$$

2. $x - 1 \neq 0 \pmod{N}$ as r must be the smallest integer for which $b^r = 1 \pmod{N}$

From this,

$$x^2 = (b^{\frac{r}{2}})^2 \pmod{N} = b^r \pmod{N} = 1 \pmod{N}$$

$$\therefore (x-1)(x+1) = x^2 - 1 = 0 \pmod{N}$$

So now, $x - 1$ and $x + 1$ aren't divisible by N, but $(x - 1)(x + 1)$ is, and as N is the product of 2 primes, p and q, it follows that $x - 1$ is divisble by p, and $x + 1$ is divisble by q.

Therefore, $p = gcd(x - 1, N)$ and $q = gcd(x + 1, N)$, and N has been factorised.

6.2.1 On a quantum computer

Let n_0 be the number of bits in $N = pq$. If N is a 500-digit number, then n_0 will be ~ 1700. In order to deal with values of x and $f(x) = b^x \pmod{N}$ between 0 and N, the input and output regisiters need at least n_0 qubits. However, to efficiently find the period r, the input register needs to actually have $n = 2n_0$ qubits. This ensures that the range of values of x for which $f(x)$ is calculated has at least N full periods of f [Mer07][Page 69].

Calculating f(x) Firstly, we need a unitary U_f which calculates $f(x) = b^x \pmod{N}$, i.e.

$$U_f |x\rangle \otimes |y\rangle = |x\rangle \otimes |y \oplus f(x)\rangle$$

Doing this is actually fairly simple: square $b \pmod{N}$, then square the result $\pmod{N}$, then square that etc. calculating the number of powers $b^{2^j} \pmod{N}$ where $j < n$. The binary expansion of $x = x_{n-1}x_{n-2}...x_1x_0$ dictates which of these must be multiplied together to get

$$b^x = \prod_j (b^{2^j})^{x_j}$$

So, if initially x is in the input register, 1 (i.e. 00..01) is in the output register, and b is in an additional work register, then the process is simple:

1. Multiply the output register by the work register using modular multiplication if $x_0 = 1$,
2. Replace the contents of the work register by its modulo-N square,
3. Repeat step 1, now conditional on $x_1 = 1$,
4. Repeat step 2,
5. Repeat step 1, now conditional on $x_2 = 1$, etc.

At the end of this, x will still be in the input register, and $b^x \pmod{N}$ will be in the output register.

Applying $\mathbf{U_f}$ Applying the above unitary to the state $|\phi\rangle_n \otimes |0\rangle_{n_0}$, where

$$|\phi\rangle_n = H^{\otimes n} |0\rangle = \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle$$

results in the state

$$\frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle_n |f(x)\rangle_{n_0}$$

If we then measure the output register to be f_0, the state of the input register will collapse to be

$$|\Psi\rangle_n = \frac{1}{\sqrt{m}} \sum_{k=0}^{m-1} |x_0 + kr\rangle_n , \qquad 0 \leq x_0 < r, \qquad f(x_0) = f_0$$

where m is the smallest integer where $mr + x_0 \geq 2^n$, and so

$$m = \left[\frac{2^n}{r}\right] \text{ or } m = \left[\frac{2^n}{r}\right] + 1, \text{ where } \left[\frac{2^n}{r}\right] \text{ is the largest integer less than or equal to } \frac{2^n}{r}$$

Measuring this would give us a value for $x_0 + kr$, where x_0 is unknown and random, and so we cannot calculate the value of r yet. We first need to remove the x_0 from the state using the quantum Fourier transform.

Quantum Fourier Transform Periodic functions can be decomposed into the sum of simple sine and cosine functions, and this decomposition is calculated by the Fourier transform. The discrete Fourier transform from the vector $x_0, x_1, ..., x_{2^n-1}$ to the vector $y_0, y_1, ..., y_{2^n-1}$ is defined by

$$y_a = \frac{1}{\sqrt{2^n}} \sum_{b=0}^{2^n-1} e^{2\pi iab/2^n} x_b$$

Here, the quantum Fourier transform is used to turn the x_0 in the state into a harmless overall phase factor, allowing us to find the value for the period r.The equation for the quantum Fourier transform unitary U_{FT} is shown below

$$U_{FT} |x\rangle_n = \frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} e^{2\pi ixy/2^n} |y\rangle_n$$

When applied to a superposition of states $|x\rangle$ with complex amplitudes $\gamma(x)$, U_{FT} produces another superposition with amplitudes related to $\gamma(x)$ by the appropriate discrete Fourier transform

$$U_{FT} \left(\sum_{x=0}^{2^n-1} \gamma(x) |x\rangle \right) = \sum_{x=0}^{2^n-1} \tilde{\gamma}(x), |x\rangle \qquad \text{where } \tilde{\gamma}(x) = \frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} e^{2\pi ixy/2^n} \gamma(y)$$

When constructing a quantum Fourier transform circuit, it is best to define a unitary U_Z, which is simply n Pauli-Z gates over n qubits, where

$$U_Z \left|y\right\rangle_n = e^{2\pi i y/2^n} \left|y\right\rangle_n$$

From this, it is clear that

$$U_Z^x H^{\otimes x} \left|0\right\rangle_n = U_Z^x \left(\frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} \left|y\right\rangle_n \right) = \frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} e^{2\pi i x y/2^n} \left|y\right\rangle_n = U_{FT} \left|x\right\rangle_n$$

From here, it is easiest to look at a specific case, say $n = 4$, and then generalize for any value of n. So, $n = 4$ means

$$U_{FT} \left|x_3\right\rangle \left|x_2\right\rangle \left|x_1\right\rangle \left|x_0\right\rangle = U_Z^x H_3 H_2 H_1 H_0 \left|0\right\rangle \left|0\right\rangle \left|0\right\rangle \left|0\right\rangle$$

If $\left|y\right\rangle_4 = \left|y_3\right\rangle \left|y_2\right\rangle \left|y_1\right\rangle \left|y_0\right\rangle$, then $y = 8y_3 + 4y_2 + 2y_1 + y_0$. From this, we can construct U_Z using its definition above.

$$U_Z y = e^{\pi i/8}(8y_3 + 4y_2 + 2y_1 + y_0)$$

$$\therefore U_Z = \exp\left(\frac{\pi i}{8}(8\mathrm{n}_3 + 4\mathrm{n}_2 + 2\mathrm{n}_1 + \mathrm{n}_0)\right)$$

$$\therefore U_Z^x = \exp\left(\frac{\pi i}{8}(8x_3 + 4x_2 + 2x_1 + x_0)(8\mathrm{n}_3 + 4\mathrm{n}_2 + 2\mathrm{n}_1 + \mathrm{n}_0)\right)$$

Now, the unitary $\exp(2\pi i \mathrm{n})$ is the same as a two Pauli-Z gates, and so doesn't affect 1-qubit states, i.e. is simply an identity operator. This means any $\exp(2^p \pi i \mathrm{n})$ can be removed, allowing us to simplify U_Z^x

$$\begin{aligned} U_Z^x = \exp(\pi i(&\mathrm{n}_3(8x_3 + 4x_2 + 2x_1 + x_0) + \\ &\frac{1}{2}\mathrm{n}_2(8x_3 + 4x_2 + 2x_1 + x_0) + \\ &\frac{1}{4}\mathrm{n}_1(8x_3 + 4x_2 + 2x_1 + x_0) + \\ &\frac{1}{8}\mathrm{n}_0(8x_3 + 4x_2 + 2x_1 + x_0)) \end{aligned}$$

$$\therefore U_Z^x = \exp\left(\pi i(x_0\mathrm{n}_3 + (x_1 + \frac{1}{2}x_0)\mathrm{n}_2 + (x_2 + \frac{1}{2}x_1 + \frac{1}{4}x_0)\mathrm{n}_1 + (x_3 + \frac{1}{2}x_2 + \frac{1}{4}x_1 + \frac{1}{8}x_0)\mathrm{n}_0\right)$$

Next, the operators $\exp(\pi i x \mathrm{n})$ and $H\left|0\right\rangle$ obey the following relation over single qubits.

$$\exp(\pi i x \mathrm{n}) H \left|0\right\rangle = H \left|x\right\rangle$$

This lets us simplify the above definition for $U_Z^x H^{\otimes x} \left|0\right\rangle_n$. For now, lets ignore the terms with fractional coefficients.

$$\begin{aligned} &\exp(\pi i(x_0\mathrm{n}_3 + x_1\mathrm{n}_2 + x_2\mathrm{n}_1 + x_3\mathrm{n}_0) H_3 H_2 H_1 H_0 \left|0\right\rangle \left|0\right\rangle \left|0\right\rangle \left|0\right\rangle \\ = \quad &[\exp(\pi i x_0 \mathrm{n}_3) H_3 \left|0\right\rangle] \quad [\exp(\pi i x_1 \mathrm{n}_2) H_2 \left|0\right\rangle] \quad [\exp(\pi i x_2 \mathrm{n}_1) H_1 \left|0\right\rangle] \quad [\exp(\pi i x_3 \mathrm{n}_0) H_0 \left|0\right\rangle] \\ = \quad &[H_3 \left|x_0\right\rangle] \quad [H_2 \left|x_1\right\rangle] \quad [H_1 \left|x_2\right\rangle] \quad [H_0 \left|x_3\right\rangle] \\ = \quad &H_3 H_2 H_1 H_0 \left|x_0\right\rangle \left|x_1\right\rangle \left|x_2\right\rangle \left|x_3\right\rangle \end{aligned}$$

When we add in and group the remaining terms we are left with the following

$$\begin{aligned} U_{FT}\left|x_3\right\rangle\left|x_2\right\rangle\left|x_1\right\rangle\left|x_0\right\rangle =& H_3 \exp\left[\pi i \mathrm{n}_2 \frac{1}{2} x_0\right] H_2 \exp\left[\pi i \mathrm{n}_1 \left(\frac{1}{2}x_1 + \frac{1}{4}x_0\right)\right] H_1 \\ &\times \exp\left[\pi i \mathrm{n}_0 \left(\frac{1}{2}x_2 + \frac{1}{4}x_1 + \frac{1}{8}x_0\right)\right] H_0 \\ &\times \left|x_0\right\rangle\left|x_1\right\rangle\left|x_2\right\rangle\left|x_3\right\rangle \end{aligned}$$

The state $\left|x_0\right\rangle\left|x_1\right\rangle\left|x_2\right\rangle\left|x_3\right\rangle$ is an eigenstate of the number operators n_3, n_2, n_1, n_0 with eigenvalues x_0, x_1, x_2, x_3, allowing us to simplify further.

$$\begin{aligned} U_{FT}\left|x_3\right\rangle\left|x_2\right\rangle\left|x_1\right\rangle\left|x_0\right\rangle =& H_3 \exp\left[\pi i \frac{1}{2} \mathrm{n}_2 \mathrm{n}_3\right] H_2 \exp\left[\pi i \mathrm{n}_1 \left(\frac{1}{2}\mathrm{n}_2 + \frac{1}{4}\mathrm{n}_3\right)\right] H_1 \\ &\times \exp\left[\pi i \mathrm{n}_0 \left(\frac{1}{2}\mathrm{n}_1 + \frac{1}{4}\mathrm{n}_2 + \frac{1}{8}\mathrm{n}_3\right)\right] H_0 \\ &\times \left|x_0\right\rangle\left|x_1\right\rangle\left|x_2\right\rangle\left|x_3\right\rangle \end{aligned}$$

After defining a new unitary operator for the 2-qubit operators, this becomes easier to read.

$$V_{ij} = \exp(\pi i \mathrm{n}_i \mathrm{n}_j / 2^{|i-j|})$$

$$\begin{aligned} &\therefore U_{FT}\left|x_3\right\rangle\left|x_2\right\rangle\left|x_1\right\rangle\left|x_0\right\rangle \\ &\quad = H_3(V_{32}H_2)(V_{31}V_{21}H_1)(V_{30}V_{20}V_{10}H_0)\left|x_0\right\rangle\left|x_1\right\rangle\left|x_2\right\rangle\left|x_3\right\rangle \end{aligned}$$

From here, it is clear that V_{ij} simply rotates qubit x_j by $e^{\pi i/2^{|i-j|}}$ when the qubit x_i is in the state $\left|1\right\rangle$, allowing us to define a simple rotation operator.

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{\pi i/2^k} \end{bmatrix}, \text{ where } k = |i-j|$$

All that is left is to define a unitary S, where $S\left|x_3\right\rangle\left|x_2\right\rangle\left|x_1\right\rangle\left|x_0\right\rangle = \left|x_0\right\rangle\left|x_1\right\rangle\left|x_2\right\rangle\left|x_3\right\rangle$ (made out of multiple SWAP operations), and we have a definition for U_{FT} using 1 and 2 qubit gates.

$$U_{FT} = H_3(R_1H_2)(R_2R_1H_1)(R_3R_2R_1H_0)S$$

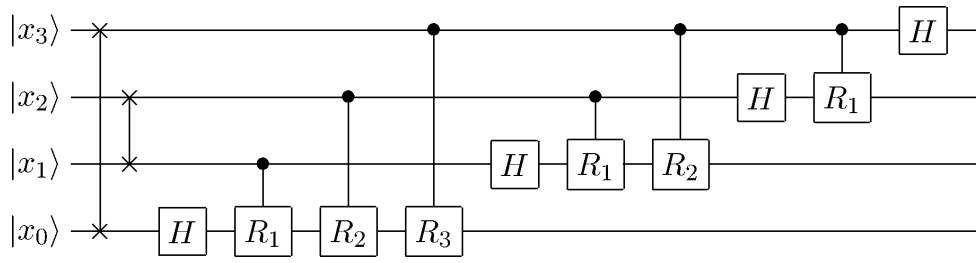


Figure 12: The circuit diagram for the quantum Fourier transform operation over 4 qubits

Finding the period So, after applying the unitary U_f to the state $|\phi\rangle_n \otimes |0\rangle_{n_0}$ we are left with the state

$$\frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle_n |f(x)\rangle_{n_0}$$

After measuring the output register, the input register collapses to the state

$$|\Psi\rangle_n = \frac{1}{\sqrt{m}} \sum_{k=0}^{m-1} |x_0 + kr\rangle_n , \qquad 0 \leq x_0 < r, \qquad f(x_0) = f_0$$

To extract the period r, we can apply the quantum Fourier transformation as shown below.

$$\begin{aligned} U_{FT} |\Psi\rangle &= \frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} \frac{1}{\sqrt{m}} \sum_{k=0}^{m-1} e^{2\pi i(x_0+kr)y/2^n} |y\rangle \\ &= \sum_{y=0}^{2^n-1} e^{2\pi i x_0 y/2^n} \frac{1}{\sqrt{2^n m}} \left(\sum_{k=0}^{m-1} e^{2\pi i k r y/2^n} \right) |y\rangle \end{aligned}$$

The probability of getting the result y is therefore given by

$$\frac{1}{2^n m} \left| \sum_{k=0}^{m-1} e^{2\pi i k r y/2^n} \right|^2$$

which is highest when y is close to integral multiples of $2^n/r$. So, the period is given by the equation $r = 2^n/y$.

Factorizing N Now that we have a value for r, we need to check that it is even, and then calculate $x = b^{r/2} \pmod N$. After this, we need to check that $(x-1)$ is not divisible by N. If both these cases are true, we can factorize N, and if not we must find a new value for r using a different value for b for the application of U_f.

All that is left to do is find the greatest common divisor of N and $x-1$, and N and $x+1$, which will be the values of p and q in the equation $N = pq$.

6.2.2 An alternative

The quantum Fourier transformation has another use; it's inverse can be used in a procedure known as phase estimation. If we have a superposition of states

$$\frac{1}{2^{t/2}} \sum_{j=0}^{2^t-1} e^{2\pi i \varphi j} |j\rangle_n |u\rangle_m$$

we can apply the inverse quantum Fourier transform to get the state

$$|\hat{\varphi}\rangle_n |u\rangle_m \text{ where } \hat{\varphi} \text{ is an approximation of } \varphi \text{ up to } n \text{ bits.}$$

We can use phase estimation to execute Shor's algorithm with a higher accuracy. So, to recap, after applying U_f, we are left with the state

$$\frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle \, |b^x \pmod N\rangle$$

This state is approximately equal to

$$\frac{1}{\sqrt{r2^n}} \sum_{s=0}^{r-1} \sum_{x=0}^{2^n-1} e^{2\pi i x \frac{s}{r}} \, |x\rangle \, |u_s\rangle \text{ where } |u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{\frac{-2\pi i s k}{r}} \, |b^k \pmod N\rangle$$

Applying the inverse quantum Fourier transform to this state gives

$$\frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |(\frac{\hat{s}}{r})\rangle \, |u_s\rangle$$

and so measuring the first register will give us an integer result for $\varphi = \left(\frac{\hat{s}}{r}\right)$ up to n bits. We can then use the continued fractions algorithm for $\frac{\varphi}{2^n}$ to calculate r.

7 Quantum Circuit Builder

The

7.1 Design

7.2 Implementation

7.3 Testing

8 Reflections

9 References

[Aar05] Scott Aaronson. Limits on efficient computation in the physical world. 2005. https://arxiv.org/abs/quant-ph/0412143v2.

[Aug09] Lennart Augustsson. Haskell package timeit-1.0.0.0: Time a computation, 2009. Last accessed 4/2/2019. http://hackage.haskell.org/package/timeit-1.0.0.0.

[bas] Changelog for base-4.11.0.0 — Hackage. http://hackage.haskell.org/package/base-4.11.0.0/changelog.

[BCDP96] David Beckman, Amalavoyal N. Chari, Srikrishna Devabhaktuni, and John Preskill. Efficient networks for quantum factoring. *Phys. Rev. A*, 54:1034–1063, Aug 1996. https://link.aps.org/doi/10.1103/PhysRevA.54.1034.

[Ben82] Paul Benioff. Quantum mechanical models of turing machines that dissipate no energy. *Phys. Rev. Lett.*, 48:1581–1585, Jun 1982. https://link.aps.org/doi/10.1103/PhysRevLett.48.1581.

[Bor] Max Born. Zur quantenmechanik der stoßvorgänge [on the quantum mechanics of collisions]. *Zeitschrift für Physik*.

[DJ92] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992. https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1992.0167.

[DP85] David Deutsch and Roger Penrose. Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 400(1818):97–117, 1985. https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1985.0070.

[Fey82] Richard P. Feynman. Simulating physics with computers. *Internation Journal of Theoretical Physics*, 21(6/7), 1982. https://people.eecs.berkeley.edu/christos/classics/Feynman.pdf.

[Gid13] Craig Gidney. Grover's Quantum Search Algorithm, 2013. Last accessed 16/1/2019. http://twistedoakstudios.com/blog/Post2644_grovers-quantum-search-algorithm.

[Gre09] Alexander S. Green. G53/G54 NSC Lectures given by Alexander S. Green over the academic years 2009/2010., 2009. http://www.drinkupthyzider.co.uk/asg/NSC/.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM. http://doi.acm.org/10.1145/237814.237866.

[ibma] IBM Q Experience Community. Last accessed 06/12/2018. https://quantumexperience.ng.bluemix.net/qx/community.

[ibmb] IBM Q Experience Documentation Beginners Guide. Last accessed 06/12/2018. https://quantumexperience.ng.bluemix.net/qx/tutorial?sectionId=beginners-guide&page=introduction.

[ibmc] IBM Q Experience Documentation Full Guide. Last accessed 06/12/2018. https://quantumexperience.ng.bluemix.net/qx/tutorial?sectionId=full-user-guide&page=introduction.

[ibmd] IBM Q Experience, the online platform created by IBM allowing users access to Quantum Computers. Last accessed 06/12/2018. https://quantumexperience.ng.bluemix.net/qx/experience.

[ibme] Jupyter Notebook Viewer - Qiskit Tutorials. Last accessed 06/12/2018. https://nbviewer.jupyter.org/github/Qiskit/qiskit-tutorial/blob/master/index.ipynb.

[Lub17] David Lubensky. Quantum Computing gets an API and SDK, March 2017. https://developer.ibm.com/dwblog/2017/quantum-computing-api-sdk-david-lubensky/.

[Mer07] N. David Mermin. *Quantum Computer Science: An Introduction.* Cambridge University Press, 2007.

[New16] IBM Newsroom. Ibm makes quantum computing available on ibm cloud to accelerate innovation. May 2016. https://www-03.ibm.com/press/us/en/pressrelease/49661.wss.

[qio16] The QIO Haskell package built by Alexander S. Green and Prof. Thorsten Altenkirch., 2016. https://github.com/alexandersgreen/qio-haskell.

[Sho94] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Nov 1994.

[SP05] D. Solenov and V. Privman. Evaluation of decoherence for quantum computing architectures: Qubit system subject to time-dependent control. 2005. https://arxiv.org/abs/cond-mat/0506286.

[Woo17] Dr. James Wootton. Quantum Battleships: The first multiplayer game for a quantum computer, 2017. Last accessed 14/12/2018. https://medium.com/@decodoku/quantum-battleships-the-first-multiplayer-game-for-a-quantum-computer-e4d600ccb3f3.

[Zur91] W.H. Zurek. Decoherence and the transition from quantum to classical - revisited. *Physics Today*, 44:36–44, 10 1991.