The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Christopher J. Wright

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

2018/19

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

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 04/04/2019

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 $O(logN)$ [1] and searching unsorted databases in $O(\sqrt{N})$ [2]. There are even quantum games being created, such as Quantum Battleships [3].

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 [4]. 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 [5], 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 [6], and created the first recognizable theoretical framework for a quantum computer.

This resulted in David Deutsch's creation of the first universal quantum computer [7]. 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 [8], Shor's algorithm [1] and Grover's algorithm [2].

1.2 QIO

QIO is a Haskell package built by Alexander S. Green [4], 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

Note: a lot of this knowledge has been learnt from the G53/4NSC module lectures [9].
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 [10]. 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 by matrices, 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.

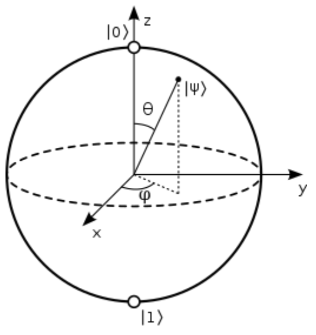


Figure 1: A Qubit represented geometrically on the Bloch Sphere

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 [11], 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.

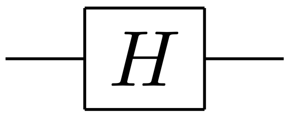


Figure 2: The Hadamard Gate

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.

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

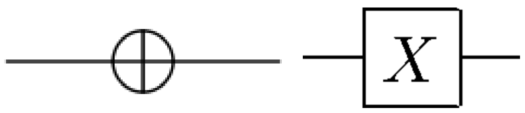


Figure 3: The 2 representations of the Pauli-X (NOT) Gate

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                          uX :: Qbit -> U
rX (x,y) = if x==y then 0 else 1        uX q = rot q rX

rY :: Rotation                          uY :: Qbit -> U
rY (False, True) = 0 :+ 1               uY q = rot q rY
rY (True, False) = 0 :+ (-1)
rY (_, _) = 0

rZ :: Rotation                          uZ :: Qbit -> U
rZ (False, False) = 1                   uZ q = rot q rZ
rZ (True, True) = -1
rZ (_, _) = 0
```

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)
```

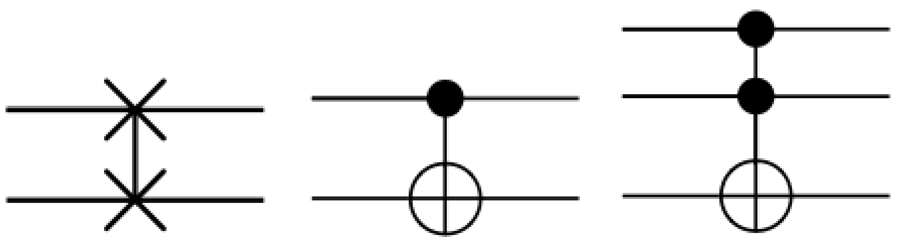


Figure 4: The SWAP gate (left), the Controlled-X (cNOT) gate (middle), and the Toffoli gate (right)

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
```

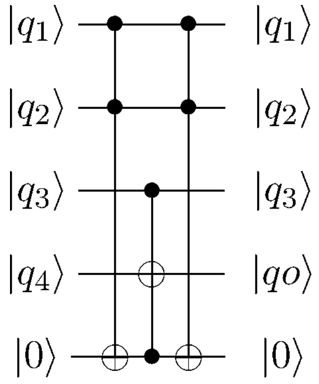


Figure 5: The cccNOT gate, made using toffoli gates and an ancilla qubit, where $|qo\rangle = |q_4 \oplus (q_1 q_2 q_3)\rangle$

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 [12]. 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 [13] - 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 [14], 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 [15], 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 perfect 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 great 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.

There are also two user guides, a Beginner's guide and Full guide, which help diminish 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. It also goes into more detail on single and multi qubit gates, alongside descriptions of quantum algorithms and some quantum error correction methods. More help is then available to users via a community forum and in-depth QISKit tutorials, 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 - 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 [2]. 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.

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 stores the result in a separate qubit (called qr), and rotates the phase of this qubit, negating the amplitude of the solution state. The reverse of the function is then run, and the qubits dispersed. This dispersion acts as an inversion around the mean amplitude, meaning the solution state is amplified due to the previous negation. When run $\log_2 n$ times, the solution will be given by measuring the qubits as long as the qubit qr is measured to be True. This will happen most of the time, however it will not always be correct due to the probabilistic nature of quantum computation.

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 $O((logN)^3)$ [16]. 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 $b^r = 1 modN$ for some integer r, and the function $f(x) = b^x modN$ 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

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

Following this, if $x + 1 \neq 0 mod N$ we can factorize the input. The probability of choosing a random b which gives a period r with these properties is ≥ 0.5.

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

$$\frac{1}{\sqrt{2^n}} \sum_{K=0}^{2^n - 1} |K\rangle_n \, |b^K mod N\rangle_m$$

$$m = \log_2 N, \quad n = 2m + 1 + \log_2(2 + \frac{1}{2\varepsilon}), \quad 0 < \varepsilon \leq 1$$

where ε is the maximum allowable error probability. 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 inverse Quantum Fourier Transformation (QFT), leaving the first register in a state approximately equal to $|s\rangle$, where $\frac{s}{2^n - 1} = \frac{s'}{r}$. Finally, we can check if r is even and if $x + 1 \neq 0 mod N$ where $x = b^r mod N$, 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 the largest prime factors of N.

5 Current QIO

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 and may produce a new state. We can then restrict unitary functions to be in a few 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 new type 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", which is the type that a quantum computation will take:

```
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, with a monadic structure.

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 → IO a` and `sim :: QIO a → 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 is a bit different to run. 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 pretty simple, increasing the usability of the project.

6 Updating QIO

6.1 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 $\geq 4.9 \&\& < 4.10$. After downloading the package, and manually changing this to allow any base package ≥ 4.9, a 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’
```

6.2 Fixing Errors

From the error messages presented, it is clear that a change to Monoids is causing errors. In base-4.11.0.0, the Haskell Semigroup class is made a superclass of Monoid [17], and so this is most likely the cause. This means that a Semigroup instance for `U` needs to be defined, along with 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. The current code is shown below:

```
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 = (<>)
```

6.3 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 previous Monoid instance, then define mempty and mappend in the Monoid definition. The current code is shown below

```
instance Monoid UnitaryC where
        mempty = U (\ fv bs -> bs)
        mappend (U f) (U g) = U (\ fv h -> g fv (f fv h))
```

Which then becomes

```
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 = (<>)
```

Finally, the Qio.hs file needs to be changed.

```
instance Monoid Unitary where
        mempty = U (\ fv h -> unEmbed $ return h)
        mappend (U f) (U g) = U (\ fv h -> unEmbed $ do h’ <- Embed $ f fv h
                                                        h’’ <- Embed $ g fv h’
                                                        return h’’)
```

Even though this definition is more complex than before, the idea for the change still remains, shown below.

```
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 = (<>)
```

6.4 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 it 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.

7 Algorithms

7.1 Grover's Algorithm

Note: a lot of this section is learnt using the book Quantum Computer Science: An Introduction[18], specifically Chapter 4, pages 88-98.

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 $\pi/4\sqrt{N}$ times, 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 thought of as just an algorithm which searches through an unsorted database faster than any classical method - however this is just one of its uses.

Put simply, Grover's algorithm takes in a blackbox function, creates a superposition over all possible function inputs and then removes states that are not solutions. After doing this a certain amount of 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.

The Oracle The blackbox function f will simply return 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:

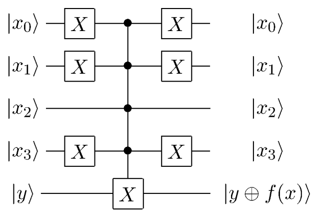


Figure 6: A unitary function, setting the last qubit to 1 when 0010 is inputted, and 0 otherwise

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

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

Therefore, applying U_f is the same as doing nothing to the single output qubit, and applying a unitary transformation V to the input register:

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

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

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

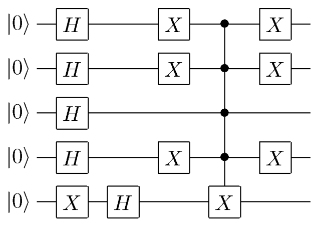


Figure 7: Applying V to an equal superposition of all possible inputs.

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 $|\phi\rangle$:

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

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.

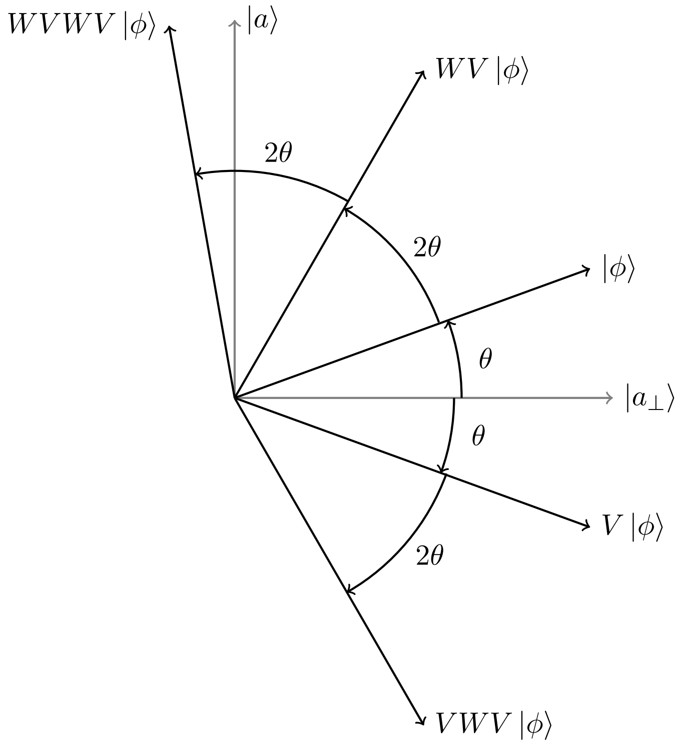
$$-W = 1 - 2\,|\phi\rangle\,\langle\phi|\,, \qquad |\phi\rangle = H^{\otimes n}\,|0\rangle_n$$

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

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 $X^{\otimes n}$, we get the unitary $-W$.

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

Application & Measurement All that is left is to apply WV repeatedly to the intital state $|\phi\rangle$, and the result of this is shown clearly 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 $\sin\theta = 1/\sqrt{N}$.

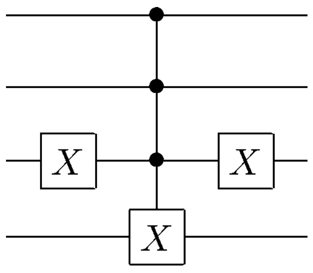
The unitary V negates the state $|a\rangle$ in $|\phi\rangle$, and so on the graph, it reflects $|\phi\rangle$ about the axis $|a_\perp\rangle$. Next, the unitary W is applied which negates any state orthogonal to $|\phi\rangle$ (i.e. any negative state in $|\phi\rangle$, notably the state $|a\rangle$). 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 from $|a_\perp\rangle$. Therefore, each iteration (application of the unitary V followed by W) rotates the qubits' state by 2θ, 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.



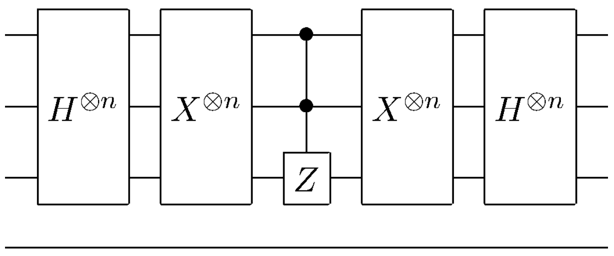


Figure 8: The 2 unitaries V (left) and W (right) used to search for $|110\rangle$.

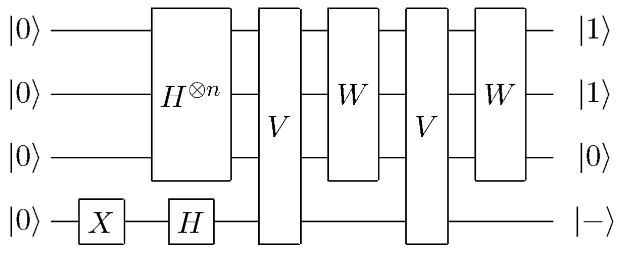


Figure 9: The circuit used to perform Grover's algorithm over 3 input qubits, with solution $|110\rangle$.

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

If we do not know how many solutions there are, you can simply perform the algorithm multiple times, with $\frac{\pi}{4}\sqrt{N}, \frac{\pi}{4}\sqrt{\frac{N}{2}}, \frac{\pi}{4}\sqrt{\frac{N}{4}}, \frac{\pi}{4}\sqrt{\frac{N}{8}}$... iterations, leading to a total number of

$$\frac{\pi}{4}\sqrt{N}(1 + \frac{1}{\sqrt{2}} + \frac{1}{2} + \frac{1}{2\sqrt{2}} + ...) = \frac{\pi}{4}\sqrt{N}(2 + \sqrt{2})$$

iterations, which is still $O(\sqrt{N})$.

7.1.1 Implementation

Implementing the algorithm in QIO from the circuit is fairly simple. We need to initialize the input qubits and ancilla qubits, and then apply $\frac{\pi}{4}\sqrt{\frac{2^n}{s}}$ Grover iterations, where n is the amount of qubits, and s is the amount of solutions to the inputted function.

It's easiest to implement this for a known function, specifically the example above searching for $|110\rangle$, and then expand this to allow custom functions to be input. In order to allow easy expansion, functions first need to be created to allow the creation and measurement of qubit lists, a helper function to make any unitary function conditional and act over a list of n qubits, where the first $n-1$ qubits are control qubits, and a helper function to apply any unitary function over a list of qubits.

```
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])
```

```
measQbits :: [Qbit] -> QIO [Bool]
measQbits qs = measQbits' qs []
  where
    measQbits' [] bs = return bs
    measQbits' (q:qs)  bs = do
      b <- measQbit q
      measQbits' qs (bs ++ [b])
```

```
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)
```

Initializing the qubits Using the functions above, initializing the qubits is straightforward. We need a function taking in the amount of input qubits, each put in the state $|+\rangle$, and the amount of output qubits, each 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)
```

Oracle Next, a unitary function V representing the blackbox function needs to be created. This function will be the same as in 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)
```

The function takes in the input qubits, `qI`, and the ancilla (output) qubit, `qO`. The unitary function `unot` is applied to the third input qubit, and then a controlled-X gate is applied across the input qubits and the output qubit. Finally, the `unot` is applied again, making the function reversible.

Diffusion The unitary function W also needs to be created, and again is the same as in the example above.

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

Application & Measurement The complete implementation is then just a combination of the functions. The `oracle` and `diffuse` functions are applied one after the other using the `applyU` function, and then once again for 2 iterations. The input qubits are then measured and returned.

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

7.1.2 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 $7.812...e^{-3}$

7.1.3 Expanding to Custom Oracle

In the example above, the oracle is searching for `True, True, False`, i.e. $x1 \land x2 \land \neg x3$. 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.

```
oracle :: Int -> [[Int]] -> [Qbit] -> [Qbit] -> U
oracle _ [] _ _ = mempty
oracle c (p:ps) qI qO =
  unitaryN unot (unots p qI) <>
  condN unot (conds p qI ++ [qO!!c]) <>
  unitaryN unot (unots p qI) <>
  oracle (c+1) ps qI qO
  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 to the function to iterate the application of V and W the correct amount of times. The time taken for the solution to be found can be timed using Haskell's TimeIt[19] module.

```
iterations :: Int -> Int -> Int
iterations n s = round $ (pi/4) * sqrt((fromIntegral (2^n))/(fromIntegral s))

grover :: Int -> Int -> [[Int]] -> QIO [Bool]
grover n i ps = do
  (qI, qO) <- initialize n (length ps)
  let groverStep = oracle 0 ps qI qO <> diffuse qI
  applyU (foldr (<>) mempty (replicate i (groverStep)))
  measQbits qI

main :: IO ()
main = do
  putStrLn "How many variables?"
  inV <- getLine
  let n = read inV :: Int
  putStrLn "Input each predicate with lines of AND operations, where each line
            will be OR'd, with an empty line after the last."
  putStrLn "For example, (x1 & -x2) | x1 | x2 would be inputted as\n1 -2\n1\n2\n
            \n-----"
  inP <- getLines
  let ps =  filter (not . null) $ map (filter (\p -> abs p <= n || p /= 0)) inP
  putStrLn "How many solutions does this formula have? This is used to calculate
            how many Grover iterations are needed."
  inS <- getLine
  let s = read inS :: Int
  solution <- timeIt $ run $ grover n (iterations n s) ps
  print solution
  where
    getLines = do
      l <- getLine
      if l == "" then return []
        else do
          let x = map read $ words l :: [Int]
          xs <- getLines
          return (x:xs)
```

Testing In order to test this, the program needs to be changed slightly to `sim` the algorithm, and print out the probabilities;

```
main :: IO ()
main = do
   ...
  let s = read inS :: Int
  print $ show $ sim $ grover n (iterations n s) ps
  ...
```

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 $3.906...e^{-3}$

7.1.4 An Improvement

This algorithm can be altered to actually not use any ancilla qubits (in QIO at least) 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 [20]; decrease the amplitude of non-solutional states about the mean amplitude, and increase the amplitude of solutional states. This can be achieved by $n-1$ Hadamard gates surrounding a C^n-NOT gate.

This new implementation is shown below.

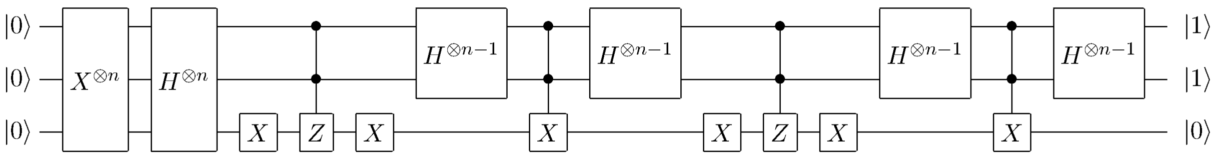


Figure 10: Improved circuit for Grover's algorithm, searching for the state $|110\rangle$

```
initialize :: Int -> QIO [Qbit]
initialize i = do
  qI <- mkQbits i True
  applyU (unitaryN uhad qI)
  return qI
```

```
diffuse :: [Qbit] -> U
diffuse qI =
  unitaryN uhad (init qI) <>
  condN unot qI <>
  unitaryN uhad (init qI)
```

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

grover :: Int -> Int -> [[Int]] -> QIO [Bool]
grover n i ps = do
  qI <- initialize n
  let groverStep = oracle ps qI <> diffuse qI
  applyU (foldr (<>) mempty (replicate i groverStep))
  measQbits qI
```

7.1.5 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. However, the implementation can be changed to also measure whether or not the input qubits contain a solution - and so we can get an amount of 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.

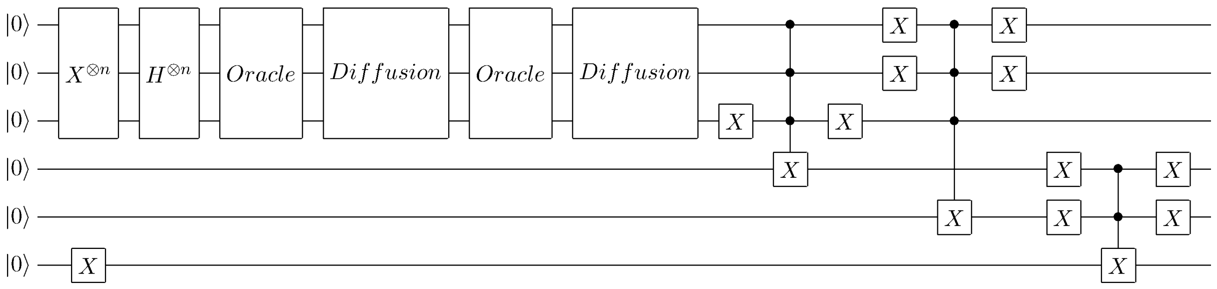


Figure 11: Improved circuit for Grover's algorithm, searching for the state $|110\rangle$ and $|001\rangle$, while also checking n qubits for the solution.

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. From this, the program can get any amount of definite solutions, as well as work for functions with an unknown amount of solutions.

```
initialize :: Int -> Int -> QIO ([Qbit], [Qbit])
initialize i a = do
  qI <- mkQbits i True
  qA <- mkQbits a False
  applyU (unitaryN uhad qI)
  return (qI, qA)

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

grover :: Int -> Int -> [[Int]] -> QIO [Bool]
grover n i ps = 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)
  measQbits ([last qA] ++ qI)

getSolutions :: Int -> Int -> [[Int]] -> [[Bool]] -> Int -> IO ()
getSolutions n s ps bs k = do
  if (k > 0)
    then do
      putStrLn "\nSearch time:"
      out <- timeIt $ run $ grover n (iterations n s) ps
      if head out
        then if (tail out) ‘notElem‘ bs
            then do
              putStrLn "Found new solution:"
              putStrLn $ show $ tail out
              getSolutions n s ps (bs ++ [tail out]) (k-1)
            else do
              putStrLn "Found old solution:"
              putStrLn $ show $ tail out
              getSolutions n s ps bs k
        else do
          putStrLn "Found non-solution:"
          putStrLn $ show $ tail out
          getSolutions n s ps bs k
    else do
      putStrLn "\nSolution(s):"
      mapM_ print bs
      putStrLn "\nTotal time:"

solve :: Int -> Int -> [[Int]] -> IO ()
solve n s ps =
  if s == 1
    then do
      putStrLn "Search time:"
      out <- timeIt $ run $ grover n (iterations n 1) ps
      if head out
        then do
          putStrLn "Found solution:"
          putStrLn $ show $ tail out
          putStrLn "\nTotal time:"
        else do
          putStrLn "Found non-solution:"
          putStrLn $ show $ tail out
          solve n 1 ps
    else do
      putStrLn $ "How many solutions do you want out of " ++ (show s) ++ "?"
      inK <- getLine
      let k = read inK :: Int
      if (k < 1 || k > s)
        then getSolutions n s ps [] s
        else getSolutions n s ps [] k

main :: IO ()
main = do
  ...
  timeIt $ solve n s ps
  ...
```

We can also print out the probability of finding the solution by running the `sim` function over a function which returns the last 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

main :: IO ()
main = do
  ...
  let s = read inS :: Int
  putStrLn $ "\nProbability of finding a solution after " ++
              (show $ iterations n s) ++ " iteration(s) is: "
  probability n (iterations n s) ps
  timeIt $ solve n s ps
  ...
```

The program can be changed to be able to handle the case where the user does not know how many solutions there are to the inputted function. In order to do this, it will need to perform Grover's algorithm with differing amounts of iterations until a solution is found (indicated by the last qubit).

```
getUnknownSolution :: Int -> Int -> [[Int]] -> IO ()
getUnknownSolution n s ps = do
  if s >= n || i >= round ((pi * sqrt(2^(fromIntegral n))/4) * (2 + sqrt(2)))
    then putStrLn "No solutions found."
    else do
      putStrLn "\nSearch time:"
      out <- timeIt $ run $ grover n i ps
      if head out
        then do
          putStrLn "Found solution:"
          putStrLn $ show $ tail out
          putStrLn "\nTotal time:"
        else do
          putStrLn "Found non-solution:"
          putStrLn $ show $ tail out
          getUnknownSolution n (s+1) ps
  where
    i = iterations n (2^s)

main :: IO ()
main = do
  ...
  putStrLn "How many solutions does this formula have? This is used to calculate
          how many Grover iterations are needed. Input '?' if the amount of
          solutions is not known."
  inS <- getLine
  if inS == "?"
```

```
    then timeIt $ getUnknownSolution n 0 ps
    else do
      let s = read inS :: Int
      putStrLn $ "\nProbability of finding a solution after " ++
                  (show $ iterations n s) ++ " iteration(s) is: "
      probability n (iterations n s) ps
      timeIt $ solve n s ps
  where
    ...
```

7.2 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.

Put simply, Shor's algorithm finds the period, r, of a function, f, on integers which is periodic under addition [18]$^{Page\ 63}$ i.e.

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

This can then be used to factorise large prime numbers. dd

8 Quantum Circuit Builder

The

8.1 Design

8.2 Implementation

8.3 Testing

9 Reflections

10 References

[1] 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.

[2] 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.

[3] 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.

[4] The qio haskell package built by alexander s. green with supervision from thorsten altenkirch., 2016. https://github.com/alexandersgreen/qio-haskell.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Alexander S. Green. G53nsc lectures given by alexander s. green over the academic years 2009/2010., 2009. http://www.drinkupthyzider.co.uk/asg/NSC/.

[10] Max Born. Zur quantenmechanik der stoßvorgänge [on the quantum mechanics of collisions]. *Zeitschrift für Physik*, 37(12):863–867, Dec 1926.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

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

[19] 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.

[20] Craig Gidney. Grover's quantum search algorithm, 2013. Last accessed 16/1/2019. http://twistedoakstudios.com/blog/.