Open Source Implementation of the Durr and Hoyer Algorithm

Mridul Sarkar

University of California-Davis

Chapter 1

Background

1.1 Quantum Computing

Quantum computing's backbone is undoubtedly quantum mechanics. Superposition, Wave Function Collapse, Entanglement, and Uncertainty are utilized by quantum computing to harness the smallest computational unit, a qubit, in order to improve computational effectiveness and efficiency. (1) The details of the quantum mechanic principles used in quantum computing will be explained in section 2.3. Through application of quantum principles we can harness a qubit and from there we can harness multiple qubits by extending quantum principles onto computational space.

1.2 Qubits and Quantum Gates

The classical computer utilizes two states, 0 and 1. Two possible states for a qubit are $|0\rangle$ and $|1\rangle$. Where

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

In order to measure either state $|0\rangle$ or $|1\rangle$ we normalize the coefficients a_1 and a_2 into:

$$|a_1|^2 + |a_2|^2 = 1$$

Measuring the qubits we get either $|0\rangle$ with probability $|a_1|^2$ or $|1\rangle$ with probability $|a_2|^2$. To better under qubits one must examine larger systems that end up being much more useful in application. There is a more generalized form of ψ. Observe $a = a_1, a_2 ..., a_n$ as representation of an arbitrary vector.

$$\sum_{i=1}^{n} a_i |x_i\rangle$$

Following the rules of linear algebra we can extend this definition to apply an arbitrary transformation matrix A which can be understood as a quantum logic

gate.
To better understand an arbitrary qubit $|x\rangle = |x_1\rangle \ldots |x_n\rangle$ we need to understand how the transformation matrix A with i columns and j rows and the arbitrary vector a interact with our n-dimensional qubit. We generalize this to:

$$\psi = \sum_{i=1}^{n}(\sum_{j=1}^{n} A_{ij}a_j)\,|x_i\rangle$$

We will now examine some useful Quantum gates. Again quantum gates are analgous to the transformation matrix mentioned above, A. Review of Classical logic gates is recommended.

It is important to remember that any Unitary matrix is a quantum gate. (3)
Not Gate:

1.3 Introduction to Quantum Mechanics and Algorithms

1.3.1 Quantum Mechanics Principles

Schrodinger equation

1-D page 15 (9) time independent (35) 2-D/3-D (135)

Uncertainty principle

page 32,122 of (9)

Identical Particles

page 191 (9)

1.3.2 Quantum Computing Principles

These postulates will help us understand how quantum mechanic principles translate into linear algebra. Allowing us to see numbers and no longer imagine quantum mechanics as some dark magic, as I did about two months ago. Additionally we are given information on how to use tools mathematicians are already familiar with in order to develop our own quantum algorithms.

Postulate 1

Associated to any isolated physical system is a complex vector space with inner product (that is, a Hilbert space) known as the state space of the system. The system is completely described by its state vector, which is a unit vector in the system's state space.

Postulate 2

The evolution of a closed quantum system is described by a unitary transformation. That is, the state—of the system at time t1 is related to the state —of the system at time t2 by a unitary operator U which depends only on the times t1 and t2

Postulate 2'

The time evolution of the state of a closed quantum system is described by the Schrodinger equation

Postulate 3

Quantum measurements are described by a collectionMm of measurement operators. These are operators acting on the state space of the system being measured. The index m refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is—immediately before the measurement then the probability that result m occurs is
" 80-84, (3)

1.3.3 Useful Definitions

Taking the information from 2.3.2 we can understand quantum mechanics and their influence on computing we to help guide us while we build an algorithm. Using these basic definitions we can keep our classically trained brain in check.

Quantum Superposition

If a system can be in state A or state B, it can also being a "mixture" of the two states. If we measure it, we see either A or B, probabilistically. (1) (9)

Quantum Entanglement

If a system can be in state A or B, it has to be a mixture of both states. When measuring the system the independent components cannot be measured unless related to each other. (1) (9)

Wave Function Collapse

When the wave function, existing in the Hermitian space in superposition as multiple eigenstates, collapses to a single eigenstate due to interaction with the external world. (1)(9)

Uncertainty principle

Pairs of measurements where greater certainty of the outcome of one measurement implies greater uncertainty of the outcome of the other measurement.(1)(9)

Chapter 2

Durr and Hoyer Algorithm

2.1 Introduction

Say we have a table with N unsorted items where you want o find the minimul value stored in this table. The Dürr–Høyer Algorithm helps us solve this problem. It was originally proposed in "A quantum algorithm for finding the minimum":[(1)], where it was called the 'Quantum Minimum Searching Algorithm'. I'll summarize the main result below in case you don't have time to read the paper in all of its glory.

1. Choose an integer (y) uniformly at random between 0...N-1, where N = flattened table length

2. Repeat the following steps until time $= 22.5 * \sqrt{(}N) + 1.4 * log^2(N)$. Time can be easily captured in python using time.clock() which returns wall clock time since program was started. The time begins once we start step 2(a), if time equals the expression with N proceed to step c.

2(a) Initialize the memory as a uniform superposition of qubits. Each qubit represents an index. After initializing the memory grab the y-th qubit and entangle the state of your register with this y-th qubit according the the Oracle given by Grover.This marks all T[j]¡T[y].

2(b) Apply the quantum exponential searching algorithm , which is a generalized Grover's search.

2(c) Measure the first register, call that outcome y' which is an index into the table. If the integer in the table at index y' is less than the integer at y, set y = y'.

3. Return y.

The steps 2(a) and 2(b) pose the biggest challenge if someone has no experience with Quantum Computing. We will first observe how to initialize the register. Then we will see how the QESA can be implemented to find a unique solution, in this case the minimum.

2.2 Algorithm

In order to initialize the register we prepare a uniform superposition of qubits, the number of qubits is determined by the number of elements in our table. We then grab our y-th index and entangle it with our register using a Controlled Z.

```
// initialize register to number of qubits as there are indices
using ((Register) = (Qubit[TableLength]))
{
    // Create Uniform Superposition of all indices
    PrepareUniformSuperposition(TableLength, LittleEndian(Register));
    // Grab the qubit in the RandomIndex (which is initialized in our
    // python host script using numpy.uniform.random()) and set it aside
    let Marker = Register[RandomIndex];
    // Apply Oracle to flip all states that are T[j]<T[y]
    Controlled Z(Register,Marker);
}
```

Step 2(a) has been satisfied. Now we must figure out how to apply the QESA algorithm. For the following circuits assume our Random Index is 0. The QESA algorithm is a generalized Grover's search characterized by the following circuit for an even number of table elements.

![2 items in a list](/assets/images/2-item-list.gif) If we have an odd number of table elements we use this circuit, where QFT is the approximate Fourier transform as given by Kitaev [(3)]. ![3 items in a list](/assets/images/3-item-list.GIF)

Lets break down whats going on here. We utilize the register we were working with earlier and apply a H transform, this is stated to simply be the Hadmard.

```
// Apply Hadamard to register
ApplyToEach(H, Register);
```

We then apply a conditional phase shift if the qubit is 0.

```
// Reflect qubits that are 0s
ApplyConditionalPhase_0(LittleEndian(Register));
```

```
operation ApplyConditionalPhase_0(register: LittleEndian) :
Unit is Adj + Ctl
{
    using (aux = Qubit())
    {
        (ControlledOnInt(0,X))(register!,aux); // If qubit is 0 flip it!
    }
}
```

From here we apply the inverse of the Hadamard, this is Hadamard since Hadamard is unitary.

```
// Apply Hadamard to register
ApplyToEach(H, Register);
```

The last step is another conditional phase shift, though it is applied if the qubit is 1.

```
// Reflect qubits that are 1s
ApplyConditionalPhase(LittleEndian(Register));

operation ApplyConditionalPhase(register : LittleEndian) :
Unit is Adj + Ctl
{
    using (aux = Qubit())
    {
        (ControlledOnInt(1,Z))(register!,aux); // If qubit is 1 flip it!
    }
}
```

Code

Our Q script will be structured as follows:

```
namespace QESA {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Diagnostics;
    open Microsoft.Quantum.Arrays;
    open Microsoft.Quantum.Preparation;
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Arithmetic;
    open Microsoft.Quantum.Math;
    open Microsoft.Quantum.Core;
    open Microsoft.Quantum.Convert;
```

```
operation Algorithm_Even(TableLength : Int, RandomIndex : Int) : Unit
{
    // intialize register to number of qubits as there are indices
    using ((Register) = (Qubit[TableLength]))
    {
       within
           {
               // Grab the qubit in the RandomIndex and set it aside
               let Marker = Register[RandomIndex];
               // Create Uniform Superposition of all indices
               PrepareUniformSuperposition(TableLength,
               LittleEndian(Register));
               // Apply Oracle to flip all states that are T[j]<T[y]
               Controlled Z(Register,Marker);
           }
        apply
           {
               // Apply Hadamard to register
               ApplyToEach(H, Register);
               // Reflect qubits that are 0s
               ApplyConditionalPhase_0(LittleEndian(Register));
               // Apply Adjunct Hadamard to register
               ApplyToEachA(H, Register);
               // Reflect qubits that are 1s
               ApplyConditionalPhase(LittleEndian(Register));
           }
    }
}
// If qubit is 0 flip it!
operation ApplyConditionalPhase_0(register: LittleEndian) :
Unit is Adj + Ctl
{
    using (aux = Qubit())
    {
        (ControlledOnInt(0,Z))(register!,aux);
    }
}
// If qubit is 1 flip it!
operation ApplyConditionalPhase(register : LittleEndian) :
Unit is Adj + Ctl
{
    using (aux = Qubit())
    {
        (ControlledOnInt(1,Z))(register!,aux);
    }
}
```

```
}
```

For futher information on how this was derived take a look at 'Tight bounds on quantum searching' [2]. It is important to note the above algorithm only works for a table with an even number of entries.

The algorithm breaks down when applying the Hadamard gate as the Hadamard is layed across the diagnoal of a identity matrix which is equal in dimensions to the number of qubits we have. With a bit of math, if we try to lay a 2x2 matrix along an odd dimensioned identity matrix the transformation is not retained. (Include math behind this)!! To circumvent this we introduce the following implementation, utilizing QFT.

```
operation Algorithm_Odd(TableLength : Int, RandomIndex : Int) : Unit
{
    // intialize register to number of qubits as there are indices
    using ((Register) = (Qubit[TableLength]))
    {
       within
            {
                // Grab the qubit in the RandomIndex and set it aside
                let Marker = Register[RandomIndex];
                // Create Uniform Superposition of all indices
                PrepareUniformSuperposition(TableLength,
                LittleEndian(Register));
                // Apply Oracle to flip all states that are T[j]<T[y]
                Controlled Z(Register,Marker);
            }
        apply
            {
                // Implemntation for odd number of table entries
                QFTLE(LittleEndian(register));
                // Reflect qubits that are 0s
                ApplyConditionalPhase_0(LittleEndian(Register));
                // Inverse QFT by using BigEndian
                QFT(BigEndian(register));
                //Reflect qubits that are 1s
                ApplyConditionalPhase(LittleEndian(Register));
            }
    }
}
```

2.2.1 Efficiency

2.3 Applications

QESA implementation and uses

Chapter 3

Conclusion and Future Work

Chapter 4

Acknowledgments