Quantum Counting Algorithm

Tomás Carneiro*
Universidade do Minho
(Dated: June 21, 2021)

The goal of the presented essay is to combine the Grover iteration with the Phase Estimation's technique using the quantum programming language *Silq* in order to implement the Quantum Counting algorithm.

I. INTRODUCTION

Silq is a relatively new language, and as such, it has been developed to be a "higher level" one. It is designed to abstract from low level implementation details of quantum algorithms. In contrast to existing quantum languages, *Silq* supports a descriptive view of quantum algorithms that expresses the high-level intent of the programmer. As a consequence, algorithms written in this language are significantly shorter and simple, less error-prone and modify the program's quantum state according to an intuitive semantics that follows the laws of quantum physics.
Silq's key technical novelty is a quantum type system that captures important aspects of quantum computation and enables safe automatic uncomputation, a fundamental challenge in existing quantum languages.
It also has a development environment which enforces static safety and also includes a simulator, which will be used in the development of this implementation.
Grover is a quantum search algorithm that attempts to find a solution to its Oracle, using the *amplitude amplification technique*, and Quantum Counting is an algorithm that tells us how many of these solutions there are. In quantum counting, we use the quantum phase estimation based on the Quantum Fourier Transform algorithm to find an eigenvalue of a Grover search iteration.

II. ALGORITHM

A. Grover's Algorithm

Grover's algorithm demonstrates the speed advantage a quantum computer has over a classical computer in searching databases. This algorithm can speed up an unstructured search problem quadratically, but its uses extend beyond that; it can serve as a general trick or subroutine to obtain quadratic run time improvements for a variety of other algorithms. It also makes use of a technique called *amplitude amplification*.

1. The Oracle

Suppose we wish to search through a search space of N elements. Instead of searching its elements directly, we'll focus on the *index* of those elements, which is in the interval $[0, N-1]$. For convenience, let's assume that $N = 2^n$, so the index can be stored in n bits, and that the problem has exactly M solutions, with $1 \leq M \leq N$. Now, let's consider the existence of a function f, which takes an input integer x, in the range $[0, N-1]$, and takes values $f(x) = 1$ if x is a solution to the search problem, and $f(x) = 0$ if it is not. We can now define a quantum *oracle* which has the ability to recognize solutions to the search problem. This recognition is signalled by making use of an *oracle qubit*, which is a unitary operator that we'll call O:

$$|x\rangle |q\rangle \xrightarrow{O} |x\rangle |q \oplus f(x)\rangle \tag{1}$$

where $|x\rangle$ is the index register, $\oplus$ denotes addition modulo 2, and the oracle qubit $|q\rangle$ is a single qubit which is flipped if $f(x) = 1$, and is unchanged otherwise. We can now define the oracle O to act on any of the simple, standard basis states $|x\rangle$ by $|x\rangle \xrightarrow{O} (-1)^{f(x)} |x\rangle$. We then say that the oracle *marks* the solutions to the search problem, by shifting the phase of the solution. [1]

2. The procedure

The algorithm begins with the computer in the equal superposition state,

$$H |0\rangle^{\oplus n} = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle \ . \tag{2}$$

The search algorithm then consists of repeated application of a quantum subroutine, known as the *Grover iteration*, which will be denoted G. Its procedure can be divided into four steps [1]:

* Correspondence email address: tomasini20@gmail.com

1. Apply the oracle O;
2. Apply the Hadamard transform $H^{\oplus n}$;
3. Perform the conditional phase shift

$$|x\rangle \rightarrow -(-1)^{\delta_{x0}} |x\rangle$$

4. Apply the Hadamard transform $H^{\oplus n}$;

The circuit for the *Grover iteration* is given by [1]:

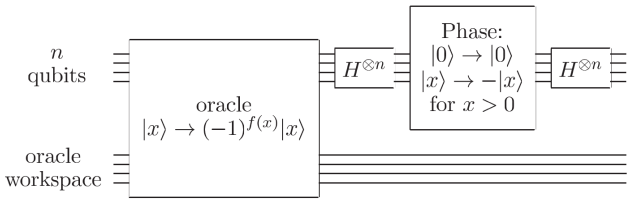

Figure 1. Grover's iterator circuit

Grover's procedure is illustrated below [3]:

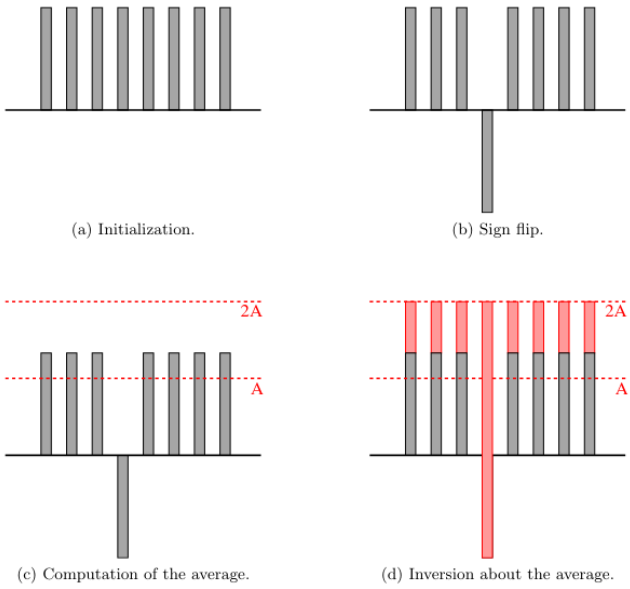

Figure 2. Grover's algorithm representation

B. Quantum Phase Estimation algorithm

1. The QFT

The *discrete Fourier transform*, in the usual mathematic notation, takes as input a vector of complex numbers, $x_0, ..., x_{N-1}$, where N corresponds to its length, and outputs a vector of complex numbers $y_0, ..., y_{N-1}$ such that:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi i j k / N}. \tag{3}$$

The *quantum Fourier transform* is the exact same transformation, only with a different conventional notation: the QFT on an orthonormal basis $|0\rangle, ..., |N-1\rangle$ is defined to be a linear operator with the following action on the basis states,

$$|j\rangle \rightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i j k / N} |k\rangle . \tag{4}$$

The action on an arbitrary state may be written as:

$$\sum_{j=0}^{N-1} x_j |j\rangle \rightarrow \sum_{k=0}^{N-1} y_k |k\rangle \tag{5}$$

where the amplitudes y_k are the discrete Fourier transform of the amplitudes x_j [1].

Using 4's result, if we consider $N = 2^n$ where n is the number of qubits, and if we let $\omega = e^{2\pi i/N}$ and $i = \sqrt{-1}$, we get the complete QFT's formula:

$$QFT(|\psi\rangle) = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} x_k \omega^{jk} |j\rangle . \tag{6}$$

The inverse QFT, or QFT^{-1} is simply:

$$QFT^{-1}(|\psi\rangle) = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} x_k \omega^{-jk} |j\rangle . \tag{7}$$

The inverse QFT operation is completely valid, as it is the computation of the QFT but in a reversible way. Since quantum operations are unitary in nature, the inverse QFT operation holds without any problems.

QFT's circuit can be described by:

2. Quantum phase estimation

This algorithm utilizes the QFT as a subroutine block. Quantum phase estimation is used to estimate the phase (or eigenvalue) of an eigenvector of a unitary operator. This unitary operator can be any quantum transformation or any quantum gate as well. In a more mathematical sense, if we have the $|\psi\rangle$ quantum state as the eigenvector of a unitary operator U with an

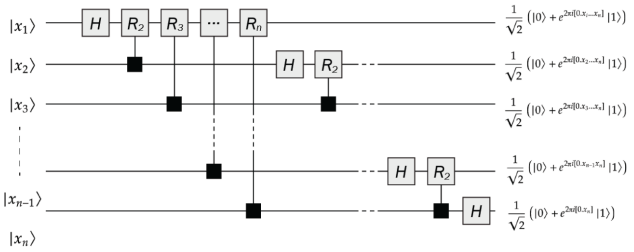

Figure 3. QFT's circuit representation

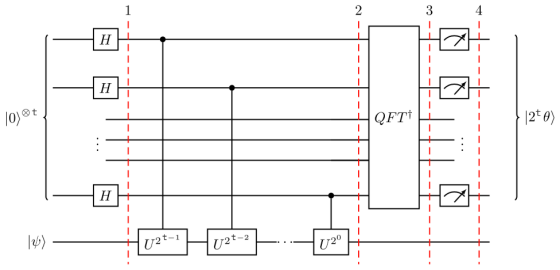

Figure 4. Quantum phase estimation circuit representation

eigenvalue of $e^{2\pi i\theta}$, then the phase estimation is used to estimate the value of θ with a high probability. The $U|\psi\rangle \geq e^{2\pi i\theta}|\psi\rangle$ eigenvalue equation holds true for the phase estimation technique. If we increase the number of qubits, then it is easier to estimate the value of θ with a high probability and a finite level of precision. To estimate the phase, we take the controlled version of the U operator and apply it to all our qubits. [1]

The algorithm can be described by the following steps:

1. We initialize our quantum state $|\psi\rangle$ as the second register and initialize our first register (ancilla) with n qubits to state $|0\rangle$.

$$\psi_0 = |0\rangle^{\oplus n}|\psi\rangle$$

2. We create a superposition of the first register n qubits (ancilla) by applying Hadamard gates to all of the n qubits.

$$\psi_1 = \frac{1}{\sqrt{2^n}}(|0\rangle + |1\rangle)^{\oplus n}|\psi\rangle$$

3. The controlled-U gate operation is applied, where the first-register n qubits (ancilla) are the control and the second register with state $|\psi\rangle$ is the target qubit. U operator can be any quantum transformation or a quantum gate.

$$\psi_2 = \frac{1}{\sqrt{2^n}}\sum_{k=0}^{2^n-1} e^{2\pi i\theta k}|k\rangle \otimes |\psi\rangle$$

4. After the controlled-U operation, we apply the inverse QFT operation to the n qubits in the first register.

$$\psi_3 = QFT^{-1}(\psi_2) = \frac{1}{2^n}\sum_{x=0}^{N-1}\sum_{k=0}^{N-1} e^{-\frac{2\pi ik}{2^n}(x-2^n\theta)}|x\rangle \otimes |\psi\rangle$$

5. Finally, we measure the n qubits of the first register to obtain our result.

$$\psi_4 = |2^n\theta\rangle \otimes |\psi\rangle$$

Quantum phase estimation's algorithm circuit is shown in figure 4:

C. Quantum counting

By combining the Grover iteration with the phase estimation technique, it is possible, on a quantum computer, to estimate the number of solutions, M, to an N item set. This combination makes way for the Quantum counting problem.
Going back, an iteration of Grover's algorithm, G, rotates the state vector by θ in the $|\omega\rangle$, $|s'\rangle$ basis (see 5) [2]:

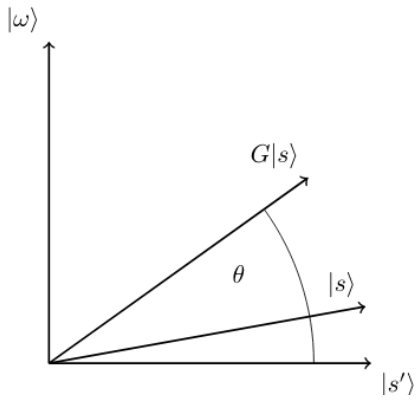

Figure 5. Geometric visualization of Grover

According to the number of solutions found, the difference between $|s\rangle$ and $|s'\rangle$ will be affected. If there are few solutions, $|s\rangle$ will be close to $|s'\rangle$ and therefore θ will be small. As is turns out, Grover iterator's eigenvalues are $e^{\pm i\theta}$, and this can be extracted using the Quantum Phase Estimation technique to estimate the number of solutions [2].

Quantum counting can be represented by the following circuit:

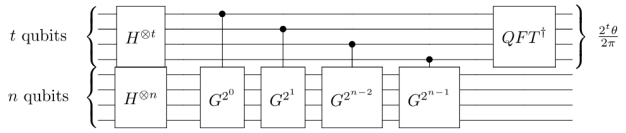

Figure 6. Quantum counting circuit representation

III. IMPLEMENTATION

A. Grover's search for one solution using Silq

We'll start by implementing Grover's diffusion operator, which applies the conditional phase shift

$$|x\rangle \to -(-1)^{\delta_{x0}} |x\rangle$$

and the Hadamard gates before and after the phase shift:

```
def groverDiff[n:!N]
    (cand:uint[n]) mfree: uint[n]{
     for k in [0..n) { cand[k] := H(cand[k]); }
     if cand!=0 { phase(pi); }
     for k in [0..n) { cand[k] := H(cand[k]); }
     return cand;
}
```

In the `n` code variable is the number of elements that we want to have in the list. Note that `n` is of type `!N`, which means that it is a *classical natural number type*, which is classically known. The `cand` variable is an unsigned array of size `n` that stores the marked element that is our candidate solution, and we want this candidate solution to have the maximum probability amplitude so that when we measure the candidate state, we get this solution easily. First, Hadamard gates are applied to all the elements of `cand` to create an equal superposition. Then, when it is checked that `cand` is not 0, we flip the phase of the candidate solution by π in `cand`, and then again apply the Hadamard gates to all the elements present in the `cand` variable to create a balanced superposition state. Finally, we return `cand`.

Now, let's see the code for Grover's algorithm, which consists of the Grover's oracle call

$$|x\rangle |q\rangle \xrightarrow{O} |x\rangle |q \oplus f(x)\rangle$$

and the Grover diffusion operator being called to find the marked element:

```
def grover[n:!N] (f:uint[n]! -> lifted B) {

    nIters := round(pi / 4 * sqrt(2^n));
    cand := 0:uint[n];
    for i in [0..n) {
        cand[i] := H(cand[i]);
    }
    for k in [0..nIters) {
        cand := groverIter[n](f, cand);
    }
    return measure(cand);
}

def groverIter[n:!N]
    (f:uint[n]! -> lifted B, cand:uint[n]) {

    if f(cand) { phase(pi); }
    cand := groverDiff(cand);
    return cand;
}
```

In the code above, the grover function takes Grover's oracle `f` as an input, which can be defined by a *lambda* function, as we will see shortly. The `const` is used because `f` preserves its argument (it is not consumed) and `uint[n]` describes the fact that `f` takes a parameter in $[0, 2^n]$. The `!-> lifted B` is used since `f` consists of classical operations (does not introduce nor destroy superposition) and returns a Boolean B.
But what defines our oracle, anyway? Since we know that the function that defines it, f, takes an input integer x, in the range $[0, N-1]$, and takes values $f(x) = 1$ if x is a solution to the search problem, and $f(x) = 0$ if it is not, it can be defined as follows:

```
def f(x:uint[n]) lifted:B{
    return x==v;
}
```

It's truly amazing how simply our oracle can be defined, which shows us how practical the language *Silq* can be.
Back to grover's function, `:!N` indicates that the return type of the function is a classical integer. The number of iterations is defined by $\frac{\pi}{4}\sqrt{2^n}$ [1].
The `cand` unsigned array is defined for `n` bits and a Hadamard operation is applied to all the `n` bits in the `cand` variable to create a superposition of all the elements present in the `cand` variable. Now, a for loop is initiated, which goes until the number of iterations means `nIterations`, and the auxiliary function `groverIter` is called, returning its result to `cand`. In this auxiliary function, `f(cand)` is checked to see whether the marked element, which is the candidate solution, is found or not. If a solution is found, meaning `f(cand)` is 1, then the phase of that marked element is flipped by π and the Grover diffusion function is called to act on the `cand` variable, which now contains the flipped version of the candidate solution. Finally, the

function returns the measurement of `cand`, which is the marked element.

B. QFT and Phase estimation using Silq

1. QFT

Starting by implementing the QFT algorithm, we will have the circuit represented in figure 3 as our model.
Using Hadamard and controlled phasing, we generate the state corresponding to the circuit with the qubits in reverse order. Thus, the reversing of the qubits is done in anticipation at the beginning of the function so that, after applying Hadamard and phasing them, the qubits are outputted in the correct order. The code is as follows:

```
def QFT [n:!N] (x: int[n]) mfree: int[n] {
    for k in [0..n div 2) {
        (x[k],x[n-k-1]) := (x[n-k-1],x[k]);
    }
    for k in [0..n) {
        x[k] := H(x[k]);
        for l in [k+1..n) {
            if x[l] && x[k] {
                phase(2*pi*2^(k-l-1));
            }
        }
    }
    return x;
}
```

In the above function, `n` qubits are taken and a quantum state of `x` is taken as the input on which the QFT is being applied. The `mfree` parameter indicates that the function **doesn't do measurements**. Lastly, `int[n]` defines the return type of the function which is a n-bit integer.
During the first *for* loop, the previous state `x` is replaced by the one with qubits in reverse order. In the second *for* loop, we start by applying Hadamard operations to our qubits. Then, in the third *for*, the next qubits are taken as control qubits and the phase operation is applied to the qubit where Hadamard was applied before. Finally, we can return our state `x`.

The reversing of the qubits is done in anticipation at the beginning of the function so that after applying the Hadamard and phasing them, the qubits are outputted in the correct order!

2. Phase estimation

Now that we've implemented QFT's algorithm, we can proceed to develop the quantum phase estimation, since it uses QFT as a subroutine block.
The implementation is as follows:

```
def phaseEstimation[k:!N]
    (U:int[k]!->mfree int[k],
    u:int[k], precision:!N) {

    ancilla := 0:int[precision];
    for i in [0..precision) {
        ancilla[i] := H(ancilla[i]);
    }
    for i in [0..precision) {
        if ancilla[i] {
            for l in [0..2^i) {
                u := U(u);
            }
        }
    }
    ancilla := reverse(QFT[precision])(ancilla);
    result := measure(ancilla);
    measure(u);
    return result;
}
```

The function above takes the `U` operator, `u` as the $|\psi\rangle$ state, and the `precision` value (depending on the number of qubits) of the θ estimate as the input. This means that given a unitary operator U and an eigenvector $|u\rangle$ such that $U|u\rangle = e^{2\pi i\theta}|u\rangle$, with $0 \leq \theta < 1$, the phase estimation algorithm outputs the phase θ of $|u\rangle$ with a given level of precision which is $!N$. `u` is an eigenvector of `U`. `U` and `u` are defined as `int[k]` where `k` is a generic parameter. The `!->mfree` signifies that there are classical operations with no measurements.
The ancilla register is the first register with n precision value qubits, and we apply the Hadamard gate to all the ancilla qubits, which creates a superposition. After creating the superposition, we iterate through our precision value, set the ancilla qubits as the control, and apply the `U` operator to `u` ($|\psi\rangle$ state) in the `u := U(u);` line. This is called controlled-U because we apply the `U` operator depending on the value of a control bit, in this case, one of the ancilla qubits. After the controlled-U operation is finished, we perform an inverse QFT operation in the `ancilla := reverse(QFT[precision])(ancilla);` line to all our first register n qubits.
Finally, we measure our ancilla qubits and state `u` and return the result.

C. Quantum counting

Now, putting it all together, a function `quantumCount` was created, which implements our Quantum Counting circuit (see figure 6):

```
def quantumCount[k:!N]
    (f:uint[k]! -> lifted B,
    precision:!N) {

    n := 0:uint[k];
    ancilla := 0:uint[precision];
    for i in [0..precision) {
        ancilla[i] := H(ancilla[i]);
    }
    for i in [0..k) { n[i]:= H(n[i]); }
    for i in [0..precision) {
        if ancilla[i] {
            for l in [0..2^i) {
                // Grover Operator which
                // calls the oracle f
                if f(n) { phase(pi); }
                n := groverDiff(n);
            }
        }
    }
    ancilla := reverse(QFT[precision])(ancilla);
    result := measure(ancilla);
    return result;
}
```

This function is, in fact, structurally identical to the Phase Estimation one. As input, it receives the function `f` which is our Grover oracle, the number of counting qubits (`precision`) and the remainder number of qubits (`k`). We initialize both `k` and `precision`-sized states, `n` and `ancilla`, respectively, applying Hadamard gates to both, creating both superpositions. We then iterate through our precision value, set the ancilla qubits as the control, and apply the Grover iterator to state `n`. After the for-loop, we apply the inverse-QFT to our ancilla bits, measuring its value afterwards. The value measured should correspond to the number of matches found.

IV. RESULTS AND DISCUSSION

For testing, a function `test_qcount` was created, which creates different oracles and executes `quantumCount` with each one of them. The code for the test function is as follows (note that a main function must be defined in the program's body in order for it to run our test function):

```
def main () {
    test_qcount();
}

def test_qcount () {
    n := 4:!N;

    // Create oracles
    f0 := lambda(x:uint[4])lifted:B{
        return x==8 || x==9 || x==10 || x==11; };

    f1 := lambda(x:uint[4])lifted:B{
        return x==4 || x==5 || x==6; };

    f2 := lambda(x:uint[4])lifted:B{
        return x==1 || x==2; };

    f3 := lambda(x:uint[4])lifted:B{ return x==3; };

    // Run Quantum Count function for each oracle
    g := quantumCount(f0,4);
    x := quantumCount(f1,4);
    y := quantumCount(f2,4);
    z := quantumCount(f3,4);

    //Print results
    print(g);
    print(x);
    print(y);
    print(z);

    //Confirm obtained values
    assert ( g==4 );
    assert ( x==3 );
    assert ( y==2 );
    assert ( z==1 );
}
```

The results obtained weren't, unfortunately, correct. Sometimes, one or two assertions would confirm, but never all of them.

[1] M. A. Nielsen, I. L. Chuang, *Quantum Computation and Quantum Information* (Cambridge University Press, Cambridge).

[2] Qiskit, `https://qiskit.org/textbook/ch-algorithms/quantum-counting.html`.

[3] Giacomo Nannicini, *An Introduction to Quantum Computing, Without the Physics* (IBM T.J. Watson, Yorktown Heights, NY).

[4] C. H. Bennett, Logical reversibility of computation, *IBM Journal of Research and Development*