Simons's Algorithm Tutorial

Ishaan Shah

November 2020

1 The Problem

Given a function (implemented as a black box) $f : \{0,1\}^n \to \{0,1\}^n$, promised to satisfy the property that, for some non zero binary string $s \in \{0,1\}^n$

$$f(x) = f(y) \Leftrightarrow x = y \text{ or } x = y \oplus s$$

Find the secret string s in least possible queries. It is important to note that the function $f(x)$ given to us is in the form of a black box which means we cannot extract the bit string by looking at the circuit inside.

For example consider the following truth table for $n = 2$ and $s = 10$

x	$f(x)$
00	10
01	01
10	10
11	01

2 Algorithm

2.1 The classical way

The classical algorithm is pretty straightforward. We find the output for 0^n and store it. Then we find outputs of successive binary strings until we find the output we got for 0^n. The input for which this is true will be the secret bit string. We can easily see that the time complexity for this is $\mathcal{O}(2^n)$ and space complexity is $\mathcal{O}(1)$.

2.2 The quantum way

Before we begin to understand how Simon's algorithm works, we have to look at a smaller sub-algorithm namely the **Fourier Sampling**.

Fourier Sampling

Fourier sampling is the process of applying n Hadamard Gates to a system of n qubits and measuring the result.

If the input to the circuit is denoted by $|\psi\rangle$ and then the output $|\psi'\rangle$ is given by

$$|\psi'\rangle = \sum_{y \in \{0,1\}^n} \frac{(-1)^{y \cdot \psi}}{2^{n/2}} |y\rangle$$

Now that we know how Fourier sampling works lets move on to the actual algorithm. The actual algorithm consists of three steps

2.2.1 Step 1 - Superposition

In the first step, we create an equal superposition of all the possible bit strings of length n. This can easily be done by initialising n qubits to 0 and passing them through n Hadamard gates. So now the state of our n qubits will be given by

$$|\psi\rangle = \sum_{y \in \{0,1\}^n} \frac{1}{2^{n/2}} |y\rangle$$

2.2.2 Step 2 - Apply $f(x)$ and measure

After creating the superposition described below, we pass the vector $|\psi\rangle$ into the black box provided to us. Now we measure the output qubits that we get from the black box which causes the input vector $|\psi\rangle$ to collapse to the following state

$$|\psi\rangle = \frac{1}{\sqrt{2}} |z\rangle + \frac{1}{\sqrt{2}} |z \oplus s\rangle \qquad \text{...where } z \in \{0,1\}^n$$

2.2.3 Step 3 - Fourier sampling

Now we apply the Fourier sampling algorithm to the input which is in the above mentioned state. After applying n Hadamard gates to the n input bits the state of the vector $|\psi\rangle$ becomes

$$\begin{aligned} |\psi\rangle &= \sum_{y \in \{0,1\}^n} \left(\frac{(-1)^{y \cdot z} + (-1)^{y \cdot (z \oplus s)}}{2^{(n+1)/2}} \right) |y\rangle \\ &= \sum_{y \in \{0,1\}^n} \left(\frac{(-1)^{y \cdot z}[1 + (-1)^{y \cdot s}]}{2^{(n+1)/2}} \right) |y\rangle \, s \end{aligned}$$

When we measure the input register $|\psi\rangle$ after Fourier sampling we will get a vector $|y\rangle$ such that $y \cdot s = 0 \bmod 2$. This will happen because the probability all ys which have an odd dot product will become 0.

Now we have a way to find some vector $|y\rangle$ such that,

$$y_n s_n \oplus y_{n-1} s_{n-1} \oplus \cdots \oplus y_1 s_1 = 0$$

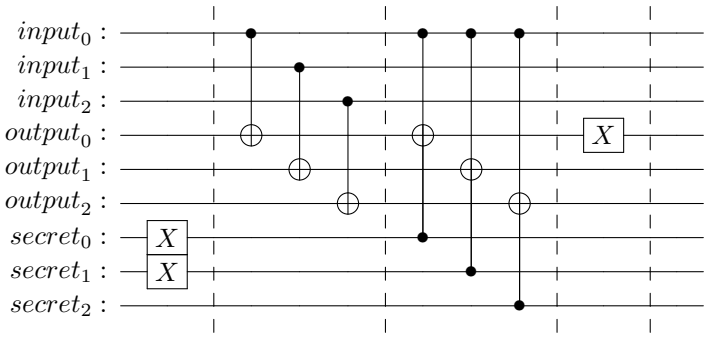
To find the secret bit string s we have to get $n-1$ independent linear equations as above. It can be shown that the probability of finding these $n-1$ equations in n tries is $\frac{1}{4}$. So if we perform the measurements $10n$ times the probability that we don't get independent equations is 0.05 which is very less. After getting the equations they can be solved to find the secret string s using Gaussian elimination, which takes $\mathcal{O}(n^3)$ time.

3 Implementation

3.1 Implementing the Oracle

There are many ways in which we can implement the oracle for Simon's Algorithm, here we look at one such implementation. Our basic goal is to create a 2-1 mapping from domain to range of $f(x)$. This can be realised by the following procedure -

- Encode the secret string using the X gate wherever there is a 1.
- Copy the input register to output register using $CNOT$ gates.
- We know that $s \neq 0^n$, so there exists at least one bit in s which has the value 1. Let us call this bit as the **MSB** bit. We now apply $CCNOT$ gate on output bit which is controlled by the corresponding secret bit and the input **MSB** bit. This creates a 2-1 mapping.
- Now we randomly flip some of the output qubits to further obfuscate the oracle. This step is optional and can be made even more complex so to make it harder to guess the secret string s.



Circuit for Simon's oracle for $s = 110$

Qiskit implementation

Following is the code for the Oracle using IBM's Qiskit library. The complete source along with the test function can be found in the Github repository.

Oracle Code

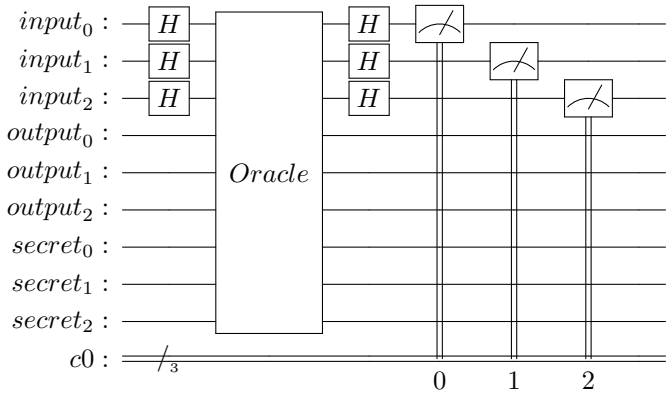
```
import itertools
import random
import time
from typing import List, Tuple

from qiskit import (Aer, ClassicalRegister,
                    QuantumCircuit, QuantumRegister,
                    execute)
from qiskit.circuit import Gate

random.seed(time.monotonic())


def simons_oracle(size: int, secret_input: List[int] = None):
    """ Returns a quantum gate which acts as
        an Oracle for Simon's Algorithm
    """
    # Register holding the secret string
    secret = QuantumRegister(size, "secret")

    # Input and Output registers
    input = QuantumRegister(size, "input")
    output = QuantumRegister(size, "output")

    # Initialize circuit
    oracle = QuantumCircuit(input, output, secret)

    # Generate random secret string
    if not secret_input:
        secret_input = [0 for i in range(size)]
        while secret_input.count(0) == size:
            secret_input = [
                random.choice([0, 1]) for i in range(size)
            ]

    # Encode it in the quantum register
    for i in range(size):
        if secret_input[i]:
            oracle.x(secret[i])
```

Oracle Code

```
    # Copy input register to output register
    for i in range(size):
        oracle.cx(input[i], output[i])

    # Find msb of secret string
    msb = secret_input.index(1)

    # Create 2-1 mapping
    for i in range(size):
        oracle.ccx(input[msb], secret[i], output[i])

    # Randomly flip qubits for further obfuscation
    for i in range(size):
        if i % 3 == 0:
            oracle.x(output[i])

    return oracle.to_gate(label="oracle"), secret_input
```

3.2 Implementing the algorithm

The implementation of actual Simon's algorithm is pretty straightforward. After we get the set of equations we need, we use Gaussian elimination to solve them and find the secret string.



Simon's Algorithm circuit for $n = 3$

Qiskit implementation

Following is the code for the the actual algorithm using IBM's Qiskit library. The complete source can be found in the Github repository.

Simon's Algorithm Code

```
import sys

from qiskit import (Aer, ClassicalRegister,
                    QuantumCircuit, QuantumRegister,
                    execute)
from qiskit.visualization import plot_histogram

from simons_oracle import simons_oracle


# Get backend
backend = Aer.get_backend('qasm_simulator')

# Length of input
n = None
try:
    n = int(sys.argv[1])
except IndexError:
    n = 4
except ValueError:
    print("Invalid size")
    sys.exit(1)

# Secret input (optional)
secret_input = None
try:
    secret_input = sys.argv[2]
    secret_input = [int(bit) for bit in secret_input]
    if len(secret_input) != n:
        print("Length of secret string and size "
              "of input should be the same")
        exit(1)
except IndexError:
    pass


# Initialise input, output and secret registers
input = QuantumRegister(n, "input")
output = QuantumRegister(n, "output")
secret = QuantumRegister(n, "secret")
result = ClassicalRegister(n)
```

Simon's Algorithm Code

```
(oracle, secret_input) = simons_oracle(n, secret_input)
print(f"Secret string - "
      f"{''.join([str(bit) for bit in secret_input])}")

# Initialize circuit
circuit = QuantumCircuit(input, output, secret, result)

# Actual circuit
circuit.h(input)
circuit.append(oracle, [*input, *output, *secret])
circuit.h(input)

# Perform measurement
circuit.measure(input, result)

print(circuit.draw())

# Take the top results and create a matrix out
# of it to solve the equations
res = execute(circuit, backend).result().get_counts()
res = sorted(res, key=lambda k: res[k])[:2**(n-1)]

# Convert to proper integer matrix
mat = [[int(bit) for bit in bit_string] for bit_string in res]
for row in mat:
    row.reverse()

# Perform Gaussian Elimination
x, y = len(mat), n
cur_row, cur_col = 0, 0
while cur_row < x and cur_col < y:
    if mat[cur_row][cur_col] == 0:
        non_zero_row = -1
        for j in range(cur_row+1, x):
            if mat[j][cur_col] == 1:
                non_zero_row = j
                break
        if non_zero_row == -1:
            cur_col += 1
            continue
        else:
            (mat[cur_col], mat[non_zero_row]) =
                (mat[non_zero_row], mat[cur_row])
```

Simon's Algorithm Code

```
    for j in range(cur_row+1, x):
        if mat[j][cur_col] == 1:
            for k in range(y):
                mat[j][k] = mat[j][k] ^ mat[cur_row][k]

    cur_row += 1

mat = list(filter(lambda row: row.count(0) < y, mat))

# Solve the reduced matrix
sol = [-1 for i in range(n)]
for i in range(n-1):
    if not mat[i][i]:
        sol[i] = 1
        for j in range(i+1, n):
            sol[j] = 0
        break

if sol.count(-1) == n:
    sol[-1] = 1

start = sol.index(1)-1
for i in range(start, -1, -1):
    cnt = 0
    for j in range(n):
        if sol[j] == 1 and mat[i][j] == 1:
            cnt += 1
    sol[i] = cnt % 2

print(f'Calculated Secret String: '
      f'{"".join([str(bit) for bit in sol])}')
```

4 Conclusion

Hence we have devised an algorithm which can solve a problem in polynomial time ($\mathcal{O}(n^3)$) with the help of quantum computers for which the best known solution is exponential time ($\mathcal{O}(2^n)$) using classical computer. This disproves the **Church-Turing Hypothesis** and shows the potential power of quantum computers.