Deutsch–Jozsa Algorithm

(Importance of the Problem and Power of Quantum Computing)

This is one of the earliest algorithms to show power of quantum computing: solving a problem in *polynomial time* which requires *exponential time* in classical computing. Although, the problem is practically not so important, it shows various important constructs which are useful for solving practical problems using quantum computing. We start with the problem statement:

Problem statement: Given a function $f: \{0,1\}^n \rightarrow \{0,1\}$ and the fact that the function *f* is either balanced or constant, we need to determine whether the function *f* is balanced or constant?

Let us understand the problem statement with the help of example functions having 3 input bits. Figure 1 shows three truth tables corresponding to three example functions. In Figure 1(a), all the output values are 1 irrespective of input bits, hence, the corresponding function is a *constant function*. In contrast, function in Figure 1 (c) has half the output values as 0 and the other half as 1, hence, it's a *balanced function*. Function in Figure 1(b) is neither constant nor balanced as number of 0's in output is neither of 0, 4, or 8. For the problem statement of Deutsch-Jozsa algorithm, we are given that the function is either *constant* or *balanced*, hence, we know that function is not the type of Figure 1(b).

Bit3	Bit2	Bit1	output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Bit3	Bit2	Bit1	output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Bit3	Bit2	Bit1	output
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 1: Three example functions (a) Constant (b) Neither constant nor balanced (c) Balanced

Classical solution

We measure the complexity of the solution based on the number of times we need to execute the function to determine whether the function is balanced or constant. If we have *n*-bits input, trivially, we can run the function $2^{(n-1)}+1$ times to know whether the function is constant or balanced. But there is a randomized algorithm available which can be used if we are ready to tolerate some error. Specifically, take any k values for input bits as shown in Figure 1 and if all the outputs are same, declare the function as constant, otherwise, declare it as balanced. This algorithm has an error probability of $2^{-(k-1)}$.

Black box operator

Before giving the quantum solution, we need to understand the black box to be used to find whether the function *f* is constant or balanced. Let us discuss the black box function with the help of an example with 2 inputs. Further let this function be a constant function with output value 1. Figure 2 shows the truth table for the example function

Bit2	Bit1	output
0	0	1
0	1	1
1	0	1
1	1	1

Figure 2: Example function with 2 inputs

Any quantum function can be represented by its operator matrix which can be used to get output for any given input. In comparison to truth table, for using operator matrix, both inputs and outputs are represented using amplitudes of superposition (i.e., for 2 qubits, we represent input as 4 complex amplitudes of $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$). Thus, operator matrix for the function shown in Figure 2 is:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

i.e., irrespective of input, output has amplitude of $|0\rangle$ as 0 and amplitude $|1\rangle$of as 1. There are two issues with the operator matrix shown above:

1. The matrix is not a square matrix.
2. It is not a unitary matrix.

We won't get the operator output as a valid quantum state if these issues are not resolved. To resolve these issues, instead of using function *f* as black-box function, we use a modified function. This modified function B_f takes (*n*+1) inputs (corresponding to *n* inputs and 1 output for *f*) and produces the same number of outputs. Specifically, for *n*-qubits $|x\rangle$ and 1-qubit $|y\rangle$, B_f is given as:

$$B_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle$$

Thus, function B_f produces output $y \oplus f(x)$, where $\oplus$ denotes CNOT operator which will output *f*(*x*) if *y* is 0 and it will produce complimentary (X gate) of *f*(*x*) if *y* is 1. For the function shown in Figure 2, since *f*(*x*) is always 1, output of B_f will be X(*y*). Figure 3 shows Q# code for implementing B_f. This operator takes two parameters where *inqubits* array represent *x* and *outqubit* represents *y*. The operator reverses the *outqubit* as desired. Similarly, for a function given by Figure 1(c), operator can be implemented as shown in Figure 4. We can implement any of these functions using Q# operators.

```
operation Bf_C1(inqubits:
Qubit[],outqubit: Qubit): ()
{
        body
        {
            X(outqubit);
        }
}
```

Figure 3: Q# implementation of B_{f_2} (Fig. 2)

```
operation Bf_B1(inqubits: Qubit[],
outqubit: Qubit): ()
{
        body
        {
           CNOT(inqubits[1], outqubit);
        }
}
```

Figure 4: Q# implementation of B_{f_1c} (Fig 1c)

Derivation of Deutsch–Jozsa algorithm

We will be using basic gates to design Q# program implementing Deutsch-Jozsa algorithm. Using quantum computing, the problem is solved using a *single execution* of the operator B_f—which shows power of quantum computing. We will first show how quantum computing gates can be used to know whether the function is *constant* or *balanced* before giving Q# implementation for the same.

1. For a given B_f, we start with (n+1) qubits such that first n are 0 and (n+1)th qubit is 1.
2. We pass each of these inputs through H gates. Thus, we get superposition of all the 2^n combinations of n qubits along with one ancilla. State of the (n+1) qubits can represented as:

$$|\psi_0\rangle = \frac{1}{\sqrt{2^n}}\left(\sum_{s\in\{0,1\}^n}|s\rangle\right)|-\rangle$$

   where $|-\rangle$ is used to indicate ancilla $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.
3. Applying the function B_f with these inputs as $|x\rangle$ and $|y\rangle$ in $(B_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle)$ we get:

$$|\psi_1\rangle = \frac{1}{\sqrt{2^n}}\left(\sum_{s\in\{0,1\}^n}|s\rangle\,|-\oplus f(s)\rangle\right)$$

4. If the function f is constant, $f(s)$ will be 0 or 1 for all the values of s. If it is 0, $|-\oplus f(s)\rangle$ will be $|-\rangle$ and if it is 1, $|-\oplus f(s)\rangle$ will be (-1) $|-\rangle$. Thus, $|-\oplus f(s)\rangle$ can, in general, be written as:

$$(\text{-}1)^{f(s)}\,|-\rangle$$

5. Thus, for a constant f, output of B_f will be:

$$|\psi_2\rangle = (-1)^{f(s)}\,\frac{1}{\sqrt{2^n}}\Big(\sum_{s\in\{0,1\}^n}|s\rangle\Big)\,|-\rangle$$

6. Applying the H gate to this output, we will get all 0 string (just reverse operation of what we did in step 1). Thus, if the output of the measurement is all-0 string, the function is constant, otherwise the function is balanced.

Figure 5 shows the circuit for the same. As described above, if the output of measurement is all 0 string, the function is constant.

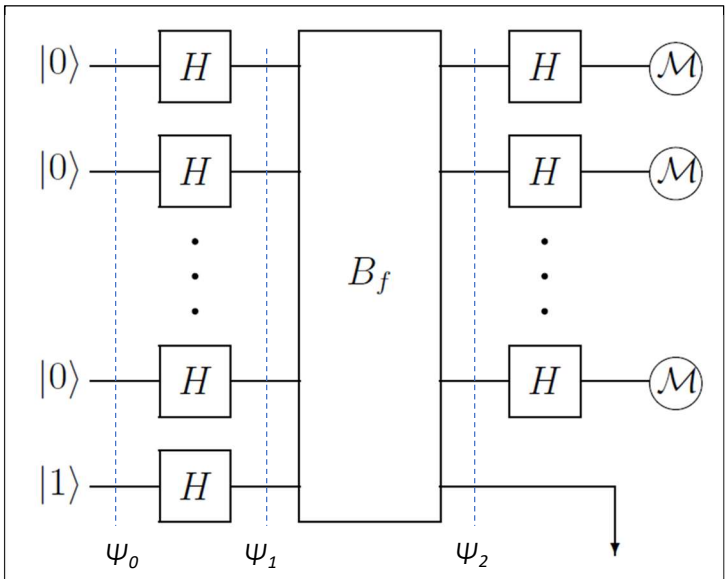

Figure 5: Circuit for Deutsch Jozsa algorithm

Implementing Deutsch–Jozsa operator

Figure 6 shows implementation of Deutsch Jozsa algorithm. We start with declaring input qubits and output qubit. In the yellow background, steps 1 and 2 are performed to initialize the qubits and applying H gate to them. The function call *blackBox* is corresponding to step 3 where we apply the function B_f to outputs of H gates. We apply H gates again to the output of B_f as depicted in green background code. We check these outputs to set value of mutable variable *m* as shown in the code with cyan background. This mutable variable is returned to the calling function. If the mutable variable is 1, the *blackBox* function is constant otherwise it is balanced.

In this implementation we assume the existence of two helper functions *Set* and *ResetAll*. The first function sets value of a qubit to a specific constant (Zero or One) whereas the second function resets values of all the input qubits to 0.

Implementing Driver

Driver program is used to call the Q# operators. In this we first create a quantum simulator and used that to run Deutsch Josza operator. Figure 7 show code fragment for the same.

Author: Rajeev Gupta
(rajeev.gupta@microsoft.com)
Microsoft AI & R, Hyderabad, India.

```
using(var sim = new
QuantumSimulator())
{
 var res =  Deutsch_Jozsa_Algorithm.
Run(sim).Result;
if (res == 1)
{
 System.Console.WriteLine("Constant");
}
else if (res == 0)
{
 System.Console.WriteLine("Balanced");
}
}
```

Figure 7: Driver implementation

```
operation Deutsch_Jozsa_Algorithm():(Int)
{
  body{
    mutable m = 0;
    mutable c = 1;
    using(inqubits = Qubit[6]){
    using(outqubit = Qubit[1]){
        ApplyToEach(H, inqubits);
        X(outqubit[0]);
        H(outqubit[0]);
        blackBox(inqubits, outqubit);
        ApplyToEach(H, inqubits);
        for(i in 1..6){
          if(c > 0){
            if(M(qubits[i-1] != Zero){
              set c = 0;
          }
        }
        }
        if(c > 0)
        {
          set m = 1;
        }
        ResetAll(inqubits);
        ResetAll(outqubit);
      }
      }
      return m;
  }
}
```

Figure 6: Algorithm implementation

Acknowledgements: Thanks to Pavan and Mariia to review this blog and giving important suggestions.

References:

1. The Q# Programming Language https://docs.microsoft.com/en-us/quantum/language/?view=qsharp-preview
2. Stephan Gulde, Mark Riebe, Gavin P. T. Lancaster, Christoph Becher, Jürgen Eschner, Hartmut Häffner, Ferdinand Schmidt-Kaler, Isaac L. Chuang & Rainer Blatt. Implementation of the Deutsch–Jozsa algorithm on an ion-trap quantum computer. Nature, Jan 2003.
3. John Watrous's Lecture Notes, https://cs.uwaterloo.ca/~watrous/LectureNotes.html
4. Q# algorithms implementation https://github.com/Microsoft/QuantumKatas/tree/master/DeutschJozsaAlgorithm