Quantum Computing 104: The Deutsch-Jozsa Algorithm

Kshipra Wadikar

March 24, 2025

Before diving into quantum algorithms, we first explored quantum gates, entanglement, and teleportation. Now, we transition into quantum algorithms by introducing the Deutsch-Jozsa algorithm, one of the first to demonstrate quantum advantage.

1 The Deutsch Algorithm

The Deutsch algorithm serves as a foundational proof-of-concept rather than a practical tool, providing a key step in quantum algorithm development. It serves as an important stepping stone in the development of quantum algorithms and provides a clear illustration of fundamental quantum principles.

1.1 Mathematical Formulation

Consider a Boolean function $f : \{0,1\} \to \{0,1\}$. The Deutsch problem asks whether f is constant ($f(0) = f(1)$) or balanced ($f(0) \neq f(1)$).
The Deutsch problem is solved using the Deutsch algorithm, which was later generalized to the Deutsch-Jozsa algorithm.

1.2 Quantum Oracle

The function $f(x)$ is implemented as a quantum oracle, a unitary operator U_f that acts on quantum states. The oracle's action is defined as:

$$U_f \left|x\right\rangle \left|y\right\rangle = \left|x\right\rangle \left|y \oplus f(x)\right\rangle$$

where $\oplus$ represents addition modulo 2 (XOR operation).

1.3 Superposition

The algorithm uses superposition to evaluate both $f(0)$ and $f(1)$ simultaneously.

1.4 Hadamard Gates

Hadamard gates generate and manipulate superposition states, enabling quantum parallelism. They are represented by the matrix:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

And they act on a single qubit as:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

$$H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

1.5 Phase Kickback

The oracle's action introduces a phase factor that encodes the function's property (constant or balanced). Specifically, when the ancilla qubit is initialized to $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, the oracle acts as:

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

This phase factor $(-1)^{f(x)}$ is crucial for the algorithm to distinguish between constant and balanced functions.

1.6 Algorithm Steps

1. Initialize two qubits: $|0\rangle |1\rangle$.
2. Apply Hadamard gates: $H|0\rangle H|1\rangle$.
3. Apply the quantum oracle U_f.
4. Apply another Hadamard gate to the first qubit.
5. Measure the first qubit.

1.7 Qiskit Implementation for the Deutsch algorithm

The following libraries are required.

Listing 1: required libraries

```python
from qiskit import QuantumCircuit, transpile
from qiskit_aer import Aer
from qiskit_aer import AerSimulator
```

Listing 2: Qiskit Implementation for the Deutsch algorithm

```python
def deutsch_algorithm(oracle_function):
""" Implements the Deutsch Algorithm for a 1-bit function. """
    circuit = QuantumCircuit(2, 1)  # 1 input qubit + 1 ancilla qubit

    # Step 1: Initialize ancilla to $\ket{1}$ and apply Hadamard
    circuit.x(1)
    circuit.h([0, 1])

    # Step 2: Apply the oracle
    oracle_function(circuit)

    # Step 3: Apply Hadamard again
```

```python
    circuit.h(0)

    # Step 4: Measure the first qubit
    circuit.measure(0, 0)

    return circuit

# Define Oracle Functions
def constant_zero_oracle(circuit):
    """ Oracle where f(x) = 0 for all inputs (does nothing). """
    pass  # No operation needed

def constant_one_oracle(circuit):
    """ Oracle where f(x) = 1 for all inputs (flips the ancilla). """
    circuit.x(1)

def balanced_x_oracle(circuit):
    """ Oracle where f(x) = x (CNOT gate applies conditional flipping). """
    circuit.cx(0, 1)  # Apply CNOT from input to ancilla

def run_deutsch_algorithm(oracle_function):
    """ Runs the Deutsch Algorithm and prints the result. """
    circuit = deutsch_algorithm(oracle_function)
    simulator = AerSimulator()
    compiled_circuit = transpile(circuit, simulator)
    job = simulator.run(compiled_circuit, shots=1000)
    result = job.result().get_counts()
    output = "Constant" if "0" in result else "Balanced"
    print(f"Oracle: {oracle_function.__name__}, Result: {result}, Function Type: {output}")

# Run the Algorithm
print("\nDeutsch Algorithm Results:\n")
run_deutsch_algorithm(constant_zero_oracle)  # Expect: Constant
run_deutsch_algorithm(constant_one_oracle)   # Expect: Constant
run_deutsch_algorithm(balanced_x_oracle)     # Expect: Balanced
```

The output of the above code is

```
Deutsch Algorithm Results:

Oracle: constant_zero_oracle, Result: {'0': 1000}, Function Type: Constant
Oracle: constant_one_oracle, Result: {'0': 1000}, Function Type: Constant
Oracle: balanced_x_oracle, Result: {'1': 1000}, Function Type: Balanced
```

2 Mathematical Formulation of the Deutsch-Jozsa Problem

Consider a Boolean function f defined as below.

$$f : \{0, 1\}^n \to 0, 1$$

where $\{0, 1\}^n$ represents the set of all n binary strings and $0, 1$ represents the set of Boolean outputs (0 or 1) and the function f is guaranteed to be either:

- **Constant**: The function f yields the same output for all possible n-bit inputs. Formally, this can be expressed as:

  $$f(x) = c, \quad \forall x \in \{0, 1\}^n, \quad \text{where} \quad c \in \{0, 1\}$$

  This means that either $f(x) = 0$ for all x, or $f(x) = 1$ for all x.

- **Balanced:** The function f produces an equal number of 0s and 1s across all possible n-bit inputs. This can be mathematically stated as:

  $$\textstyle\sum_{x \in \{0,1\}^n} f(x) = 2^{n-1}$$

  This implies that exactly 2^{n-1} inputs result in $f(x) = 0$, and 2^{n-1} inputs result in $f(x) = 1$.

The Deutsch-Jozsa problem asks: Given the function f as above, determine whether it is constant or balanced.
Example Scenario:
Consider a Boolean function f defined as below.

$$f : \{0, 1\}^3 \to 0, 1$$

where $\{0, 1\}^3$ represents the set of all 3 binary strings $\{000, 001, 010, 011, 100, 101, 110, 111\}$ and $0, 1$ represents the set of Boolean outputs (0 or 1) and the function f is guaranteed to be either:

- **Constant**: The function f yields the same output for all possible 3-bit inputs.

  $$f(x) = c, \quad \forall x \in \{0, 1\}^3, \quad \text{where} \quad c \in \{0, 1\}$$

  This means that either $f(x) = 0$ for all x, or $f(x) = 1$ for all x.

- **Balanced:** The function f produces an equal number of 0s and 1s across all possible 3-bit inputs. This can be mathematically stated as:

  $$\textstyle\sum_{x \in \{0,1\}^3} f(x) = 2^2 = 4$$

  This implies that exactly 4 inputs result in $f(x) = 0$, and 4 inputs result in $f(x) = 1$.

The Challenge: How Do We Figure Out the Type of Function?
With a classical computer, the worst case requires checking more than half the inputs up to $2^{n-1} + 1$ evaluations.

- If we only get 0s or only 1s, the function is constant.
- If we get both 0 and 1 as outputs, the function is balanced.

This means that for large values of n, checking enough inputs to be sure takes a long time.
Quantum Computation Advantage
Quantum parallelism uses superposition to evaluate a function $f(x)$ for all possible inputs at the same time.
The Deutsch-Jozsa algorithm solves this problem in a single function evaluation using quantum parallelism.
By encoding all inputs into a superposition state and applying interference, we extract the answer in one quantum measurement.

2.1 Superposition and Hadamard Transformation

The Hadamard transformation generates an equal superposition of all basis states.
The Hadamard gate H acts on a single qubit as:

$$H|x\rangle = \frac{1}{\sqrt{2}} \sum_{z=0}^{1} (-1)^{x \cdot z} |z\rangle$$

where $x, z \in \{0, 1\}$.
The Hadamard gate H acts on n-qubits as:

$$H^{\otimes n}|x\rangle = \frac{1}{\sqrt{2^n}} \sum_{z=0}^{2^n-1} (-1)^{x \cdot z} |z\rangle$$

where

- x, z are now n-bit binary numbers.
- $x \cdot z$ is the dot product (binary dot product) (bitwise AND followed by XOR sum).

The Hadamard transformation prepares our system in a uniform superposition, but on its own, it does not provide any information about the function $f(x)$. To introduce function-dependent behavior, we use a quantum oracle, which encodes $f(x)$ into the quantum state via a unitary transformation.

2.2 Quantum Oracle

Before introducing the Deutsch-Jozsa Algorithm, let's first understand quantum oracles.

1. **black-box**- In quantum computing, a black-box refers to a system where we can query a function without knowing its internal workings. We only observe its input-output behavior but not how it is implemented. For example,
   - A vending machine functions as follows: an input (button press) results in an output (drink dispensed), while the internal operational mechanisms remain unknown.
   - A password verification system processes an input password, producing an output of either access granted or access denied, without revealing its internal verification process.
2. **Quantum Oracle or Quantum Black-Box**- A quantum oracle is the quantum equivalent of a classical black-box function. Instead of directly computing $f(x)$, it encodes it into a unitary transformation U_f, allowing us to process multiple inputs in superposition simultaneously.
   Mathematically, the quantum oracle acts as:

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

where:

- $|x\rangle$ is the input register (unchanged).
- $|y\rangle$ is an auxiliary qubit that stores $f(x)$.
- $\oplus$ is modulo-2 addition (XOR), which ensures reversibility.

When the quantum oracle U_f acts on a superposition, it effectively applies the function $f(x)$ to all inputs x in the superposition simultaneously.
This is the Quantum parallelism.
Unlike a classical function call, a quantum oracle can evaluate multiple inputs at once due to superposition, enabling quantum parallelism.

2.3 Step-by-Step Explanation of the Deutsch-Jozsa Algorithm

The Deutsch-Jozsa algorithm is a quantum algorithm that determines whether a given function $f(x)$ is constant or balanced in just one function evaluation.

1. **Initialize the Quantum Register**
   We start with $n+1$- qubit system.

   - The first n qubits are initialized in the $|0\rangle$ state. The first n qubits are used for input x.
   - The last qubit (ancilla- this is auxiliary) is initialized to $|1\rangle$. The last qubit is used for function evaluation.

   At this stage our system is $|\psi_{system}\rangle = |0\rangle^n \otimes |1\rangle$.

2. **Superposition and Hadamard transformation**
   We apply a Hadamard gate $H^{\otimes(n+1)}$ to create a superposition of all possible inputs:

   - $H^{\otimes(n)}|0\rangle^{\otimes(n)} = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle$.
   - $H|1\rangle = \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle)$.

   So total state after applying Hadamard gate is,

$$H(|\psi_{system}\rangle) = \left(\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle \right) \otimes \left(\frac{1}{\sqrt{2}} (|0\rangle - |1\rangle) . \right) = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle (|0\rangle - |1\rangle)$$

3. **Apply the Function $f(x)$ using an Oracle**
   The (Boolean) function $f(x)$ is implemented using a quantum oracle U_f which acts as

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

   where:

   - $|x\rangle$ is the input register (unchanged).
   - $|y\rangle = |0\rangle - |1\rangle$ is an auxiliary qubit that stores $f(x)$.

- $\oplus$ is modulo-2 addition (XOR), ensuring reversibility.

[Note: This is equivalent to the more general oracle action when the ancilla qubit is initialized to $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$:

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

where:

- $|x\rangle$ is the input register (unchanged).
- $f(x)$ is the Boolean output of the function.

]
If $f(x) = 0$ then $|y \oplus f(x)\rangle = |y\rangle$ and if $f(x) = 1$ then $|y \oplus f(x)\rangle = -|y\rangle$ and so

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

After applying a quantum oracle U_f to $H(|\psi_{system}\rangle) = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle (|0\rangle - |1\rangle)$, we get $\frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} |x\rangle (|0\rangle - |1\rangle)$.
Since the second qubit $(|0\rangle - |1\rangle)$ is now independent, we can ignore it and focus on the first register:

$$\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} |x\rangle$$

This **adds a phase factor** $(-1)^{f(x)}$ to each basis state $|x\rangle$.

4. **Apply Hadamard Transformation Again**
Now, we apply the Hadamard gate to the first n qubits again:

$$\frac{1}{\sqrt{2^n}} \frac{1}{\sqrt{2^n}} \sum_{z=0}^{2^n-1} \sum_{x=0}^{2^n-1} (-1)^{x \cdot z + f(x)} |z\rangle$$

where

- x, z are now n-bit binary numbers.
- $x \cdot z$ is the dot product (bitwise AND followed by XOR sum).

5. **Analysis of the result**
If $f(x)$ is a constant:

- The phase factor $(-1)^{f(x)}$ is the same for all x.
- The final state is $|0\rangle^{\otimes n}$.

If $f(x)$ is a balanced:

- The phase factor $(-1)^{f(x)}$ varies, leading to interference.
- The final measurement will never yield $|0\rangle^{\otimes n}$ because of destructive interference.

6. **Measure the First n Qubits**

- If we measure all 0s (000...0), the function is constant.
- If we measure any state other than $|0\rangle^{\otimes n}$, the function is balanced.

Thus, the Deutsch-Jozsa algorithm determines whether $f(x)$ is constant or balanced in just one function call!

3 Qiskit Implementation of the Deutsch-Jozsa algorithm

We implement the above algorithm in Qiskit.

3.1 Initialize the Quantum Register, Hadamard Gates and Oracle functions

For n-qubit system, we need first n-qubits initialized to $|0\rangle$ state and $n+1$-th qubit is and ancilla qubit which we need to be initialized to state $|1\rangle$.

Listing 3: Initialize the Quantum Register first Hadamard Gates Oracle functions

```
def create_deutsch_jozsa_circuit(oracle_function, n):
""" Creates Deutsch-Jozsa circuit with n input qubits """
    qr = list(range(n + 1))  # Create a list of qubit indices
    circuit = QuantumCircuit(n+1, n)  # n input qubits + 1
       ancilla

    # Initialize the ancilla qubit in $\ket{1}$ state
    circuit.x(qr[n])

    # Apply Hadamard to all qubits
    for i in range(n + 1):
        circuit.h(qr[i])

    # Apply the oracle
    oracle_function(circuit, qr, n)

    # Apply Hadamard again to the input qubits
    for i in range(n):
        circuit.h(qr[i])

    # Measure all input qubits
    circuit.measure(qr[:n], range(n))

return circuit
```

So here we have covered steps 1 to 4 of the algorithm.

3.2 Oracle functions

In Deutsch-Jozsa Algorithm, we need to determine whether the function is constant (always 0 or 1) or balanced (equal number of 0s and 1s).

1. constant_zero_oracle $f(x) = 0$ for all x
   This oracle function returns 0 for all inputs. The oracle does nothing to the circuit.

Listing 4: constant_zero_oracle

```
def constant_zero_oracle(circuit, qr, n):
""" Oracle for f(x) = 0 (constant function returning
    always 0) """
     pass  # Do nothing
```

2. constant_one_oracle $f(x) = 1$ for all x
   Note that Quantum circuits do not "return" values like classical functions. In Deutsch-Jozsa setup, the last qubit or ancilla qubit is initialized to $|1\rangle$. X-gate on ancilla qubit flips $|0\rangle$ to $|1\rangle$.

Listing 5: constant_one_oracle

```
def constant_one_oracle(circuit, qr, n):
""" Oracle for f(x) = 1 (constant function returning
    always 1) """
```

3. balanced_xor_oracle (balanced_identity_oracle)
   This oracle function

   - Computes XOR of all input bits and stores the result in the last qubit (ancilla).
   - Uses CNOT gates from each input qubit to the ancilla.

Listing 6: balanced_xor_oracle

```
def balanced_identity_oracle(circuit, qr, n):
""" Balanced oracle where f(x) = x1 XOR x2 XOR ...
    XOR xn """
    for i in range(n):
        circuit.cx(qr[i], qr[n])  # Apply CNOT from
            input qubits to ancilla
```

4. balanced_random_oracle (random_balanced_oracle)
   This oracle function

   - Randomly flips half of the possible inputs.
   - Uses X gates on half of the input qubits before and after applying CNOT gates.

Listing 7: balanced_random_oracle

```
def balanced_random_oracle(circuit, qr, n):
""" Balanced oracle that flips half of the possible
    inputs """
    for i in range(n // 2):  # Flip half of the
       qubits randomly
        circuit.x(qr[i])
    for i in range(n):
        circuit.cx(qr[i], qr[n])
    for i in range(n // 2):
        circuit.x(qr[i])
```

3.3 Simulation

The following code demonstrates the required simulation for algorithm.

Listing 8: Simulation

```
def run_deutsch_jozsa_algorithm(oracle_function, n, shots
    =3000):
""" Runs the Deutsch-Jozsa algorithm for an n-qubit system ""
    "
```

```
        circuit = create_deutsch_jozsa_circuit(oracle_function, n
           )
        simulator = AerSimulator()
        compiled_circuit = transpile(circuit, simulator)
        job = simulator.run(compiled_circuit, shots=shots)
        result = job.result()
        counts = result.get_counts(circuit)
        return counts
```

3.4 test the algorithm

The following test the algorithm.

Listing 9: Testing

```
    def test_deutsch_jozsa(n=3):
    """ Test Deutsch-Jozsa algorithm for n input qubits """

        print("\nTesting Deutsch-Jozsa Algorithm with", n, "input
            qubits...\n")

        # Constant Function f(x) = 0
        result = run_deutsch_jozsa_algorithm(constant_zero_oracle
           , n)
        print("Constant Zero Oracle:", result)

        # Constant Function f(x) = 1
        result = run_deutsch_jozsa_algorithm(constant_one_oracle,
            n)
        print("Constant One Oracle:", result)

        # Balanced Function f(x) = x1 XOR x2 XOR ... XOR xn
        result = run_deutsch_jozsa_algorithm(
           balanced_identity_oracle, n)
        print("Balanced Identity Oracle:", result)

        # Balanced Function (Random Flip of Half Inputs)
        result = run_deutsch_jozsa_algorithm(
           balanced_random_oracle, n)
        print("Balanced Random Oracle:", result)

# Run test for n = 5 qubits
test_deutsch_jozsa(n=5)
```

The complete code for Deutsch-Jozsa Algorithm is as below.

Listing 10: Deutsch-Jozsa Algorithm

```


    def constant_zero_oracle(circuit, qr, n):
    """ Oracle for f(x) = 0 (constant function returning always
      0) """
        pass  # Do nothing

    def constant_one_oracle(circuit, qr, n):
```

```
""" Oracle for f(x) = 1 (constant function returning always 1) """
    circuit.x(qr[n])  # Flip the last qubit (ancilla)

def balanced_identity_oracle(circuit, qr, n):
""" Balanced oracle where f(x) = x1 XOR x2 XOR ... XOR xn """
    for i in range(n):
        circuit.cx(qr[i], qr[n])  # Apply CNOT from input qubits to ancilla

def balanced_random_oracle(circuit, qr, n):
""" Balanced oracle that flips half of the possible inputs """
    for i in range(n // 2):  # Flip half of the qubits randomly
        circuit.x(qr[i])
    for i in range(n):
        circuit.cx(qr[i], qr[n])
    for i in range(n // 2):
        circuit.x(qr[i])

def create_deutsch_jozsa_circuit(oracle_function, n):
""" Creates Deutsch-Jozsa circuit with n input qubits """
    qr = list(range(n + 1))  # Create a list of qubit indices
    circuit = QuantumCircuit(n+1, n)  # n input qubits + 1 ancilla

# Initialize the ancilla qubit in $\ket{1}$ state
circuit.x(qr[n])

# Apply Hadamard to all qubits
for i in range(n + 1):
    circuit.h(qr[i])

# Apply the oracle
oracle_function(circuit, qr, n)

# Apply Hadamard again to the input qubits
for i in range(n):
    circuit.h(qr[i])

# Measure all input qubits
circuit.measure(qr[:n], range(n))

return circuit

def run_deutsch_jozsa_algorithm(oracle_function, n, shots=3000):
""" Runs the Deutsch-Jozsa algorithm for an n-qubit system """
    circuit = create_deutsch_jozsa_circuit(oracle_function, n)
    simulator = AerSimulator()
    compiled_circuit = transpile(circuit, simulator)
```

```
        job = simulator.run(compiled_circuit, shots=shots)
        result = job.result()
        counts = result.get_counts(circuit)
        return counts

    def test_deutsch_jozsa(n=3):
    """ Test Deutsch-Jozsa algorithm for n input qubits """

    print("\nTesting Deutsch-Jozsa Algorithm with", n, "input
        qubits...\n")

    # Constant Function f(x) = 0
    result = run_deutsch_jozsa_algorithm(constant_zero_oracle, n)
    print("Constant Zero Oracle:", result)

    # Constant Function f(x) = 1
    result = run_deutsch_jozsa_algorithm(constant_one_oracle, n)
    print("Constant One Oracle:", result)

    # Balanced Function f(x) = x1 XOR x2 XOR ... XOR xn
    result = run_deutsch_jozsa_algorithm(balanced_identity_oracle
        , n)
    print("Balanced Identity Oracle:", result)

    # Balanced Function (Random Flip of Half Inputs)
    result = run_deutsch_jozsa_algorithm(balanced_random_oracle,
        n)
    print("Balanced Random Oracle:", result)

    # Run test for n = 5 qubits
    test_deutsch_jozsa(n=5)
```

The output of teh above code is

```
Testing Deutsch-Jozsa Algorithm with 5 input qubits...

Constant Zero Oracle: {'00000': 3000}
Constant One Oracle: {'00000': 3000}
Balanced Identity Oracle: {'11111': 3000}
Balanced Random Oracle: {'11111': 3000}
```

4 Next Steps: Bernstein-Vazirani Algorithm

The Deutsch-Jozsa algorithm demonstrated how quantum computation can distinguish between constant and balanced functions in a single query. However, what if instead of determining a function's type, we needed to extract hidden information encoded within the function? This leads us to the Bernstein-Vazirani Algorithm, which extends Deutsch-Jozsa by efficiently solving a hidden binary string problem.

In the next article, we will explore how Bernstein-Vazirani builds on superposition and Hadamard gates to discover a hidden bitstring in a single quantum query, while classical algorithms require multiple evaluations.

References

[1] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, 2010. See Sections 1.4.3, 1.4.4, and 2.2.3 for a detailed discussion of the Deutsch and Deutsch-Jozsa algorithms.