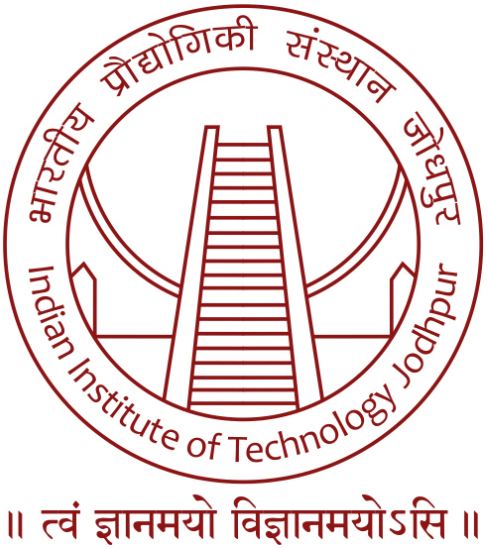
QCL6020 : Quantum Computing

Report on Quantum Algorithm's



Name: **Thirumalai M**

Roll Number: **M23IQT008**

Program: **M.Tech- Quantum Technology**

Chapter 1

Deutsch-Jozsa Algorithm

1.1 Fundamentals of Deutsch-Jozsa Algorithm

The Deutsch-Jozsa technique enables the determination of whether a given Boolean function is constant or balanced. Let us consider a binary function that accepts input values of 0 and 1, and produces output values of 0 or 1. A function is deemed to be constant if and only if all of its outputs are either 0 or 1. A function is considered balanced when exactly half of its inputs are zeroes and exactly half of its inputs are ones.

1.2 Working Principles

The Deutsch-Jozsa algorithm encompasses several distinct implementations, although with a common 5-step procedure. The initial stage involves the preparation of quantum registers and input states. During the second stage, the qubits are arranged in accordance with the Hadamard Basis. This implies that every qubit possesses an equal probability of being in the state 0 or 1, each with a probability of 50%. The third stage involves the utilisation of an oracle, which encompasses the encoding of a function that establishes the nature of the outputs, whether they are constant or balanced, through the utilisation of quantum entanglement. The fourth phase involves the reversion of the qubits to the measurement basis for the purpose of determining the solution. In the final stage, the qubits undergo measurement in order to get the answer.

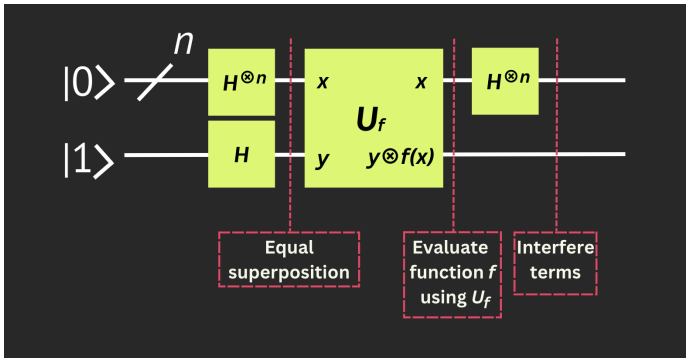


Figure 1.1: circuit diagram, credits classiq

There is minimal variation in the implementation methodologies between the initial and subsequent stages, with the primary distinction lying in the execution of the oracle. One often employed approach in implementing the oracle involves utilising a CX gate to initialise a secondary quantum register, which consists of a solitary ancilla qubit commonly known as a ”work qubit.” The standard initialization for qubits is typically set to the state 0. However, in this particular scenario, the ancilla

qubit has been initialised to the state 1. Subsequently, the oracle employs a technique known as phase kickback to evaluate the nature of the function, determining whether it is constant or balanced. In the event that the function is balanced, the phenomenon of phase kickback will introduce a negative phase to precisely half of the quantum states.

1.3 Code

```
# useful additional packages
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# importing Qiskit
import qiskit
from qiskit import BasicAer, IBMQ
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister,
execute
from qiskit.compiler import transpile
from qiskit.tools.monitor import job_monitor

# import basic plot tools
from qiskit.tools.visualization import plot_histogram


n=13


# Choose a type of oracle at random. With probability half it is constant,
# and with the same probability it is balanced
oracleType, oracleValue = np.random.randint(2), np.random.randint(2)

if oracleType == 0:
    print("The␣oracle␣returns␣a␣constant␣value␣", oracleValue)
else:
    print("The␣oracle␣returns␣a␣balanced␣function")
    a = np.random.randint(1,2**n) # this is a hidden

    parameter for balanced oracle.


# Creating registers
# n qubits for querying the oracle and one qubit for storing the answer
qr = QuantumRegister(n+1) #all qubits are initialized to zero
# for recording the measurement on the first register
cr = ClassicalRegister(n)

circuitName = "DeutschJozsa"
djCircuit = QuantumCircuit(qr, cr)
```

```
# Create the superposition of all input queries in the first register by
applying the Hadamard gate to each qubit.
for i in range(n):
    djCircuit.h(qr[i])

# Flip the second register and apply the Hadamard gate.
djCircuit.x(qr[n])
djCircuit.h(qr[n])

# Apply barrier to mark the beginning of the oracle
djCircuit.barrier()

if oracleType == 0:#If the oracleType is "0", the oracle returns
oracleValue for all input.
    if oracleValue == 1:
        djCircuit.x(qr[n])
    else:
        djCircuit.id(qr[n])
else: # Otherwise, it returns the inner product of the input with a
(non-zero bitstring)
    for i in range(n):
        if (a & (1 << i)):
            djCircuit.cx(qr[i], qr[n])
# Apply barrier to mark the end of the oracle
djCircuit.barrier()

# Apply Hadamard gates after querying the oracle
for i in range(n):
    djCircuit.h(qr[i])

# Measurement
djCircuit.barrier()
for i in range(n):
    djCircuit.measure(qr[i], cr[i])
```

The oracle returns a constant value 0

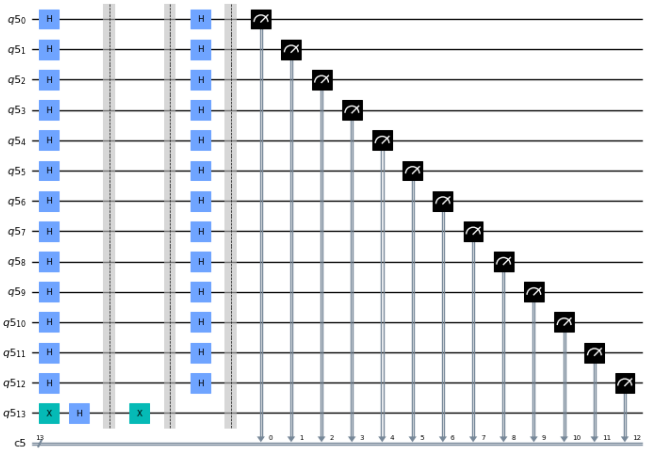


Figure 1.2: Quantum circuit diagram

```
backend = BasicAer.get_backend('qasm_simulator')
shots = 5000
job = execute(djCircuit, backend=backend, shots=shots)
results = job.result()
answer = results.get_counts()

plot_histogram(answer)
```

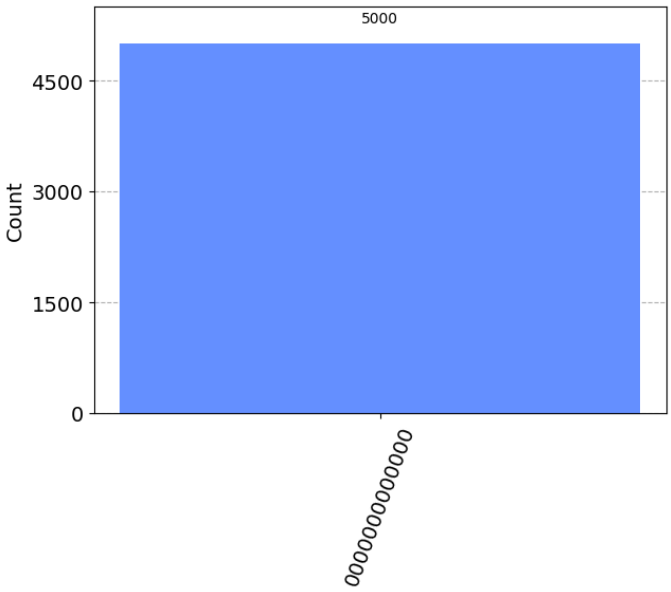


Figure 1.3: Probablity Distribution

1.4 Run-Time Complexity

1.4.1 Classical Solution

One way of measuring an algorithm's efficiency at run-time (for each use) is via 'Big-O Notation'.Let's call the size of the input domain 'n'.In the worst case, we go through half of the input domain, and get identical outputs. That's already $\frac{1}{2}$ queries.From the very next query (+1), we can determine our result — agreed output means a constant Oracle, whereas the opposite output means a balanced Oracle The time complexity in classical is $O(\frac{n}{2} + 1)$.

1.4.2 Quantum Advantage

Where the number of queries a Classical Computer must ask the Oracle for is half the input domain, a Quantum Computer can solve the Deutsch-Jozsa Problem in one call to the Oracle so the time complexity will be O(1).