Universidade do Minho

Universidade do Minho

Licenciatura em Ciências da Computação

Interacção e Concorrência 2023-24 - Practical assignment

Miguel Ângelo Alves de Freitas (A91635)

May 31, 2024

Index

1 Introduction

This document outlines the practical assignment for the course "Interacção e Concorrência" for the academic year 2023-24. The task involves implementing the Deutsch-Jozsa algorithm and Grover's Algorithm using Qiskit to demonstrate quantum computing principles.

2 The Deutsch-Jozsa Algorithm

The Deutsch-Jozsa algorithm is a quantum algorithm for determining if a given Boolean function, which maps $\{0,1\}^n$ to $\{0,1\}$, is balanced or constant. This algorithm is significant because it illustrates an exponential speed-up compared to classical counterparts, demonstrating one of the early utilities of quantum computing.

2.1 Algorithm Overview

The algorithm utilizes a quantum circuit with $n+1$ qubits where n is the number of bits in the input to the Boolean function. The steps include:

1. **Initialization:** Prepare $n+1$ qubits where the first n qubits are initialized in the state $|0\rangle$ and the auxiliary qubit in the state $|1\rangle$.
2. **Superposition:** Apply Hadamard gates to all qubits, placing them into a superposition of all possible states.
3. **Oracle Function:** Implement an oracle that applies a phase flip if the function f outputs 1 for a given state.
4. **Interference:** Apply Hadamard gates again to the first n qubits to interfere the amplitudes produced by the oracle.
5. **Measurement:** Measure the first n qubits. A result other than $|0\rangle^n$ indicates that the function is balanced.

2.2 Task Implementation

For our task, the Boolean function appears as follows:

- Inputs '000', '001', '010', '011' produce an output of 1.
- Inputs '100', '101', '110', '111' produce an output of 0.

This pattern indicates that the function is balanced. The implementation of the Deutsch-Jozsa algorithm can determine this with a single quantum operation.

2.2.1 Code Listing

```
# Import necessary Qiskit components
from qiskit import Aer, execute
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

def deutsch_jozsa(n):
    # Create quantum and classical registers
    qr = QuantumRegister(n + 1)  # n input qubits, 1 auxiliary qubit
    cr = ClassicalRegister(n)    # n classical bits for measuring the input qubits

    # Create a quantum circuit
    qc = QuantumCircuit(qr, cr)

    # Step 1: Initialization
    # Apply Hadamard gates to n input qubits to create a superposition
    for i in range(n):
        qc.h(qr[i])
    # Prepare the auxiliary qubit in |1> then apply Hadamard to get |-> state
    qc.x(qr[n])
    qc.h(qr[n])

    # Step 2: Oracle for a balanced function
    # Example: Flip the phase for the states '000', '001', '010', '011'
    for input_state in ['000', '001', '010', '011']:
        # Apply X gates conditionally to match the input state
        for j in range(n):
            if input_state[j] == '0':
                qc.x(qr[j])
        # Apply controlled phase flip (multi-controlled Z gate)
        qc.mct(qr[:n], qr[n], None, mode='noancilla')
        # Reset the qubits to their initial state for further iteration
        for j in range(n):
            if input_state[j] == '0':
                qc.x(qr[j])

    # Step 3: Interference
    # Apply Hadamard gates to the first n qubits
    for i in range(n):
        qc.h(qr[i])

    # Step 4: Measurement
    # Measure the first n qubits
    qc.measure(qr[:n], cr[:n])

    # Simulate the circuit
    simulator = Aer.get_backend('qasm_simulator')
    result = execute(qc, simulator, shots=1).result()
```

```
    counts = result.get_counts(qc)

    # Return the histogram of results
    return counts

# Number of qubits (input size of the function)
n = 3
# Call the function and print the results
results = deutsch_jozsa(n)
print("Measurement␣Results:", results)
plot_histogram(results)
plt.show()
```

2.3 Outcome

results1.jpg

Upon executing the circuit, if all qubits are measured as $|0\rangle$, the function is constant. Otherwise, it is balanced. Given the defined Boolean function, the expected outcome is a result indicating that the function is balanced.

3 Grover's Algorithm

Grover's algorithm is a quantum algorithm that provides a quadratic speedup for searching an unstructured database. Given a database of N items, Grover's algorithm can find a target item in approximately $\sqrt{N}$ operations, which is significantly faster than the classical approach that requires N operations in the worst case.

3.1 Problem Specification

In this exercise, we consider a database with $N = 16$ elements. We aim to use Grover's algorithm to find the element indexed by the quantum state $|0010\rangle$.

3.1.1 Quantum Circuit Setup

The quantum circuit for implementing Grover's algorithm consists of several components:

- **Initialization:** All qubits are initialized to the state $|0\rangle$ and then put into a superposition using Hadamard gates.
- **Oracle:** An oracle is used to invert the sign of the amplitude corresponding to the target state $|0010\rangle$.
- **Diffuser:** A diffuser (or Grover operator) is applied to amplify the probability amplitude of the target state.

3.1.2 Oracle Implementation

The oracle for the state $|0010\rangle$ can be implemented by applying X-gates to the qubits that correspond to a '0' in the target state (qubits 0, 2, and 3 in our case), applying a multi-controlled Z gate (or equivalent operation) to flip the amplitude, and then undoing the X-gates.

```
qc.x([0, 2, 3])
qc.mcx([0, 1, 2, 3], 4)
qc.x([0, 2, 3])
```

3.1.3 Diffuser Implementation

The diffuser is designed to spread out the amplitudes of the non-target states and amplify the target state. It is generally implemented as follows:

```
    qc.h(range(n_qubits))
    qc.x(range(n_qubits))
    qc.h(n_qubits-1)
    qc.mcx(list(range(n_qubits-1)), n_qubits-1)
    qc.h(n_qubits-1)
    qc.x(range(n_qubits))
    qc.h(range(n_qubits))
```

3.1.4 Number of Iterations

The optimal number of Grover iterations k is approximately $\frac{\pi}{4}\sqrt{\frac{N}{M}}$, where M is the number of solutions. For a single solution, this reduces to $\frac{\pi}{4}\sqrt{N}$.

3.2 Part (b): Extension for Multiple Items

If we wish to find one of multiple specific elements (e.g., $|0000\rangle, |0101\rangle, |1011\rangle, |1110\rangle$), the oracle needs to mark all these states. Additionally, the number of iterations may need adjustment since M (the number of target states) increases, affecting the amplitude amplification process.

3.2.1 Code Listing

```
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram
from qiskit.quantum_info import Statevector
import matplotlib.pyplot as plt
import math

# Number of qubits and database elements
N = 16
n_qubits = int(math.log(N, 2))

# Function to create the oracle for the state |0010>
def oracle_0010(qc):
    qc.x([0, 2, 3])  # Apply X-gates to flip the necessary qubits to match |0010>
    qc.mcx([0, 1, 2, 3], 4)  # Multi-controlled Toffoli to flip the ancilla qubit
    qc.x([0, 2, 3])  # Uncompute (reset the qubits)

# Grover's diffuser
def diffuser(n_qubits):
    qc = QuantumCircuit(n_qubits)
    qc.h(range(n_qubits))
    qc.x(range(n_qubits))
    qc.h(n_qubits-1)
    qc.mcx(list(range(n_qubits-1)), n_qubits-1)
    qc.h(n_qubits-1)
    qc.x(range(n_qubits))
    qc.h(range(n_qubits))
```

```
    return qc

# Create quantum circuit
qc = QuantumCircuit(n_qubits+1, n_qubits)  # One additional qubit for the oracle anc

# Initialize all qubits in superposition
qc.h(range(n_qubits))

# Add the oracle
oracle_0010(qc)

# Apply diffuser
qc.append(diffuser(n_qubits), range(n_qubits))

# Measurement
qc.measure(range(n_qubits), range(n_qubits))

# Execute the circuit
simulator = Aer.get_backend('qasm_simulator')
result = execute(qc, simulator, shots=1024).result()
counts = result.get_counts(qc)

# Plot the results
plot_histogram(counts)
plt.show()
```

3.3 Results

results2_2.jpg

Results from a quantum simulator should show a high probability of measuring the state $|0010\rangle$ after the correct number of iterations. The results should demonstrate the quadratic speedup in searching compared to a classical search.

3.4 Part(c) Grover's Algorithm with a Specific Oracle

3.4.1 Oracle Setup

In the provided oracle circuit, a controlled-NOT gate is applied to an ancilla qubit, with control on the DB_3 qubit. This oracle flips the ancilla qubit if DB_3 is in the state $|1\rangle$. The other qubits DB_0, DB_1, and DB_2 do not influence the operation, indicating that the oracle marks all states where the fourth bit (DB_3) is 1.

3.4.2 Implications for Grover's Algorithm

- **Database Size:** The database consists of $N = 16$ elements, requiring 4 qubits (DB_0 to DB_3).
- **Marked States:** The oracle marks the 8 states from $|1000\rangle$ to $|1111\rangle$, where $DB_3 = |1\rangle$. These states correspond to the binary representations from 1000 to 1111, or decimal numbers 8 through 15. Specifically, these are:
  - $|1000\rangle$ - Decimal 8
  - $|1001\rangle$ - Decimal 9
  - $|1010\rangle$ - Decimal 10
  - $|1011\rangle$ - Decimal 11
  - $|1100\rangle$ - Decimal 12
  - $|1101\rangle$ - Decimal 13
  - $|1110\rangle$ - Decimal 14
  - $|1111\rangle$ - Decimal 15

  These marked states are where the fourth qubit DB_3 is $|1\rangle$, identifying the elements in the database that satisfy this condition.

3.5 Number of Iterations and Experimental Expectations

- **Number of Solutions (M):** There are 8 solutions, corresponding to the marked states.
- **Optimal Iterations (k):** The number of optimal Grover iterations is approximately $\frac{\pi}{4}\sqrt{\frac{N}{M}} \approx \frac{\pi}{4}\sqrt{2} \approx 1.11$, typically rounded to the nearest whole number, suggesting about 1 iteration.

References

1. Qiskit Community Tutorials, *Deutsch-Jozsa Algorithm.* Available at: `https://github.com/qiskit-community/qiskit-community-tutorials/blob/master/algorithms/deutsch_jozsa.ipynb`
2. Qiskit Community Tutorials, *Grover's Algorithm.* Available at: `https://github.com/qiskit-community/qiskit-community-tutorials/blob/master/algorithms/grover_algorithm.ipynb`
3. Qiskit Aer Documentation, *Tutorials.* Available at: `https://qiskit.github.io/qiskit-aer/tutorials/index.html`

4. University of Minho, *Grover's Algorithm Theory from Class*. Available at: `https://lmf.di.uminho.pt/ic-2324/TP/23_24/Lecture%204%20-%20Grover/Grovers%20algorithm%20theory%20from%20class.pdf`

5. Lecture Notes, *Introduction to Quantum Computing using Qiskit*. Lecture 1 from TP/23_24. Simulated on a local Jupyter notebook. File: `lecture2.ipynb`