CSBB 311 : QUANTUM COMPUTING

LAB PRACTICAL FILE

Submitted By:

Name: KARTIK MITTAL

Roll No: 221210056

Branch: CSE

Semester: 5th Sem

Group: 2

Submitted To: Dr. VS Pandey

Department of Computer Science and Engineering

NATIONAL INSTITUTE OF TECHNOLOGY DELHI

2024

INDEX TABLE

ASSIGNMENT - 6

Theory -

1. Introduction to Grover's Search Algorithm

- Grover's Search Algorithm is a quantum algorithm that efficiently searches an unsorted database or solves black-box search problems. It provides a quadratic speedup over classical algorithms, reducing the number of queries required to find a solution.

2. Grover’s Search with an Unknown Number of Solutions

- The central idea remains the same: Grover’s algorithm uses quantum parallelism to evaluate multiple possibilities at once, and then iteratively amplifies the amplitude of the correct solutions. The search process is repeated O(sqrt(N/M)) times, where M is the number of solutions in the database, providing an efficient way to locate all solutions.

3. Workflow of Grover's Search Algorithm

- **Quantum Circuit Initialization**:Initialize a superposition state over all possible database entries using Hadamard gates, creating an equal amplitude state.
- **Oracle Query**: Apply the oracle function, which marks the correct solutions by flipping their amplitudes. In this case, the oracle can mark multiple solutions but will flip the phase of each one.
- **Amplitude Amplification**: The diffusion operator acts by inverting the amplitude of each state about the average amplitude of all states.
- **Measurement**: After sufficient iterations, measure the state of the quantum register. The measurement will yield one of the solutions with high probability, and further measurements can be made to find additional solutions.

4. Importance of Grover’s Algorithm

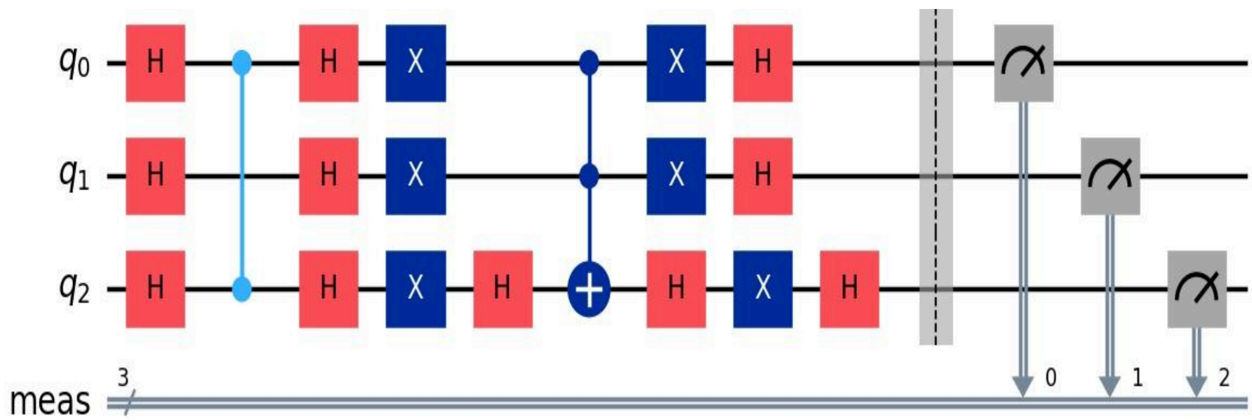
- Grover's algorithm with an unknown number of solutions is crucial for efficiently solving search problems in situations where classical methods would require a linear number of queries.

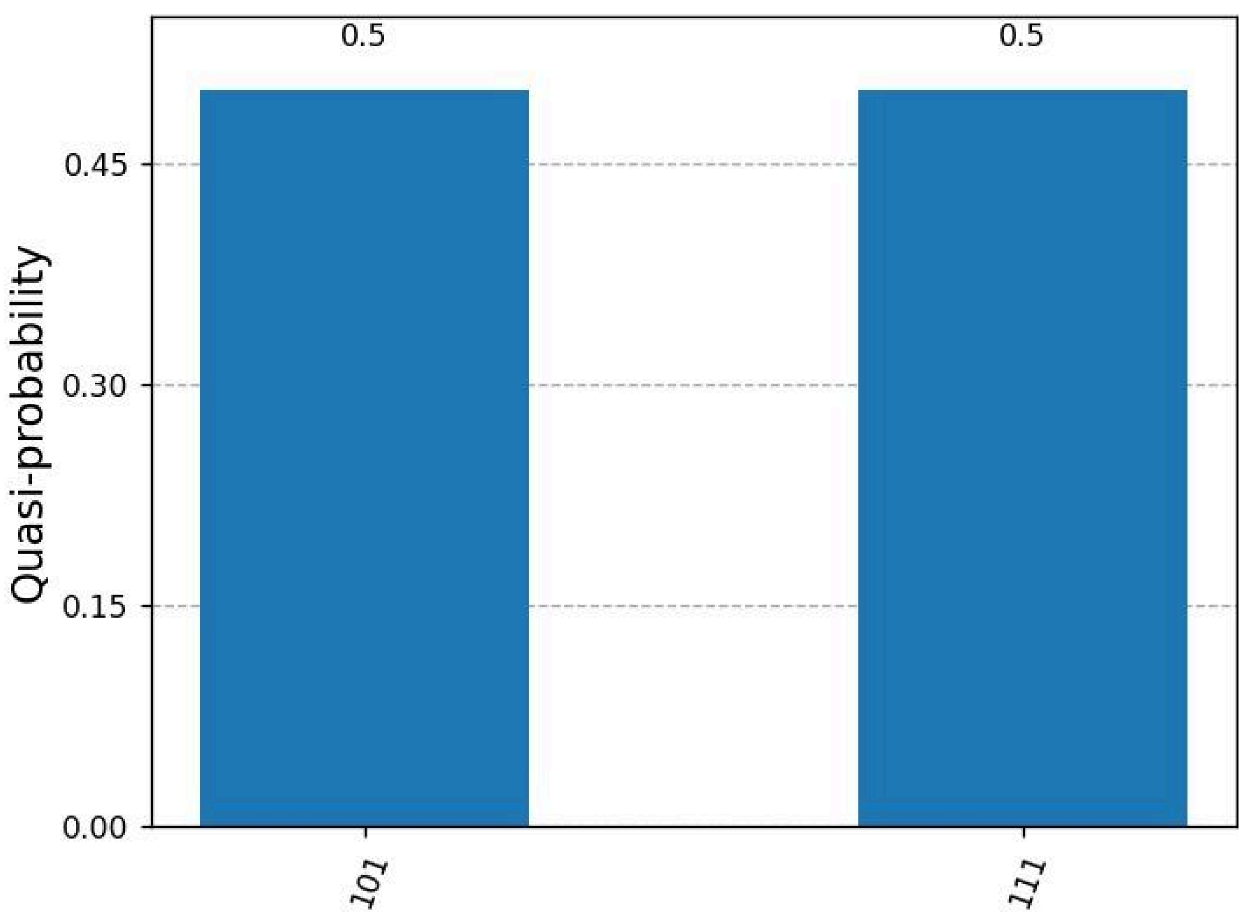
Code (Grover Search) -

```
from qiskit import QuantumCircuit , execute
from qiskit_aer  import Aer
from qiskit.visualization import plot_histogram, circuit_drawer
from qiskit.circuit.library import MCXGate
import matplotlib.pyplot as plt

# Create a 3-qubit quantum circuit for Grover's search
n = 3  # Number of qubits
qc = QuantumCircuit(n)

# Apply Hadamard gates to create a superposition
qc.h(range(n))

# Example oracle for marking the state |101>
qc.cz(0, 2)

# Apply the diffusion operator (inversion about the mean)
qc.h(range(n))
qc.x(range(n))
qc.h(n - 1)

# Add a multi-controlled Toffoli gate using MCXGate
mct_gate = MCXGate(num_ctrl_qubits=n-1)  # Create an MCX gate with (n-1) control qubits
qc.append(mct_gate, range(n))  # Append the gate to the circuit

qc.h(n - 1)
qc.x(range(n))
qc.h(range(n))
```

```
# Measure the qubits
qc.measure_all()

# Visualize the quantum circuit
circuit_diagram = circuit_drawer(qc, output='mpl')
plt.show()  # Display the circuit diagram

# Use Aer simulator to simulate and get results
simulator = Aer.get_backend('qasm_simulator')
job = execute(qc, backend=simulator, shots=1024)
result = job.result()
counts = result.get_counts()

# Plot and visualize the result
plot_histogram(counts)
plt.show()
```

Output -

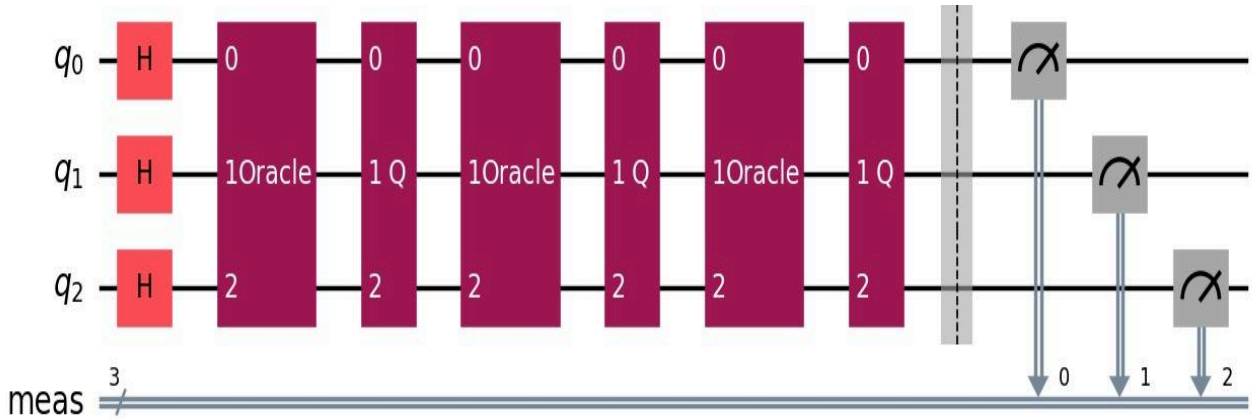

q_0
q_1
q_2
meas
3
H
X
0
1
2


Quasi-probability
0.5
0.5
0.45
0.30
0.15
0.00
101
111

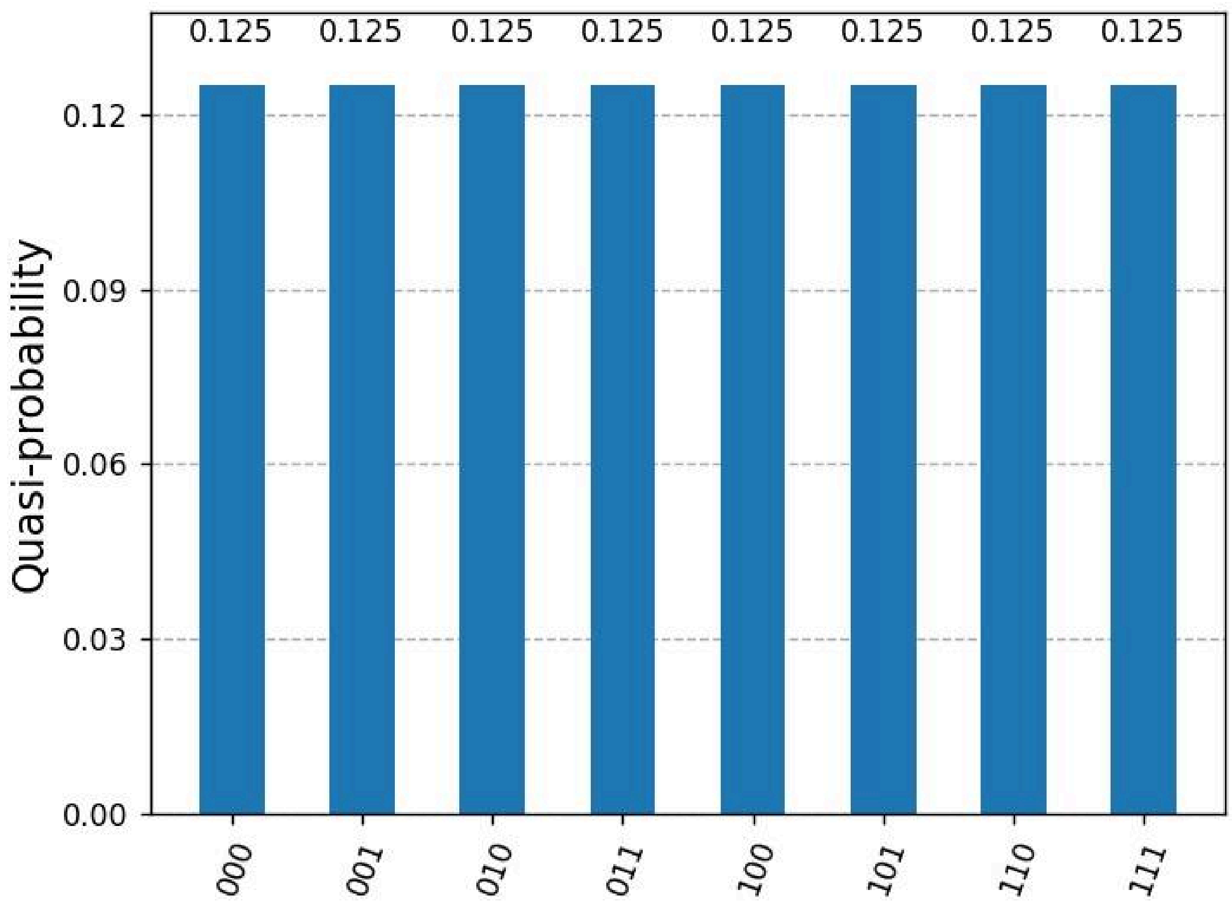
Code -

```python
from qiskit import QuantumCircuit, transpile
from qiskit_aer import Aer
from qiskit.primitives import Sampler
from qiskit.circuit.library import GroverOperator
from qiskit.visualization import plot_histogram, circuit_drawer
import matplotlib.pyplot as plt

# Define a custom oracle for multiple solutions
def custom_oracle(n):
    oracle = QuantumCircuit(n)
    # Mark states |001> and |110> as solutions
    oracle.cz(0, 2)
    oracle.cz(1, 2)
    oracle.name = "Oracle"
    return oracle

# Set up the circuit for 3 qubits
n = 3
oracle = custom_oracle(n)

# Create the Grover diffusion operator
grover_operator = GroverOperator(oracle)

# Initialize Grover's search circuit
iterations = 3  # Number of iterations for Grover's algorithm
qc = QuantumCircuit(n)
qc.h(range(n))  # Initial Hadamard gates for superposition

# Apply Grover iterations
for _ in range(iterations):
    qc.append(oracle, range(n))
    qc.append(grover_operator, range(n))

# Measure all qubits
qc.measure_all()

# Visualize the quantum circuit
circuit_diagram = circuit_drawer(qc, output='mpl')
plt.show()

# Use Sampler to simulate and get results
sampler = Sampler()
backend = Aer.get_backend('aer_simulator')
transpiled_qc = transpile(qc, backend)
result = sampler.run(transpiled_qc).result()
counts = result.quasi_dists[0].binary_probabilities()

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

Output -



q_0
q_1
q_2
meas
H
H
H
0
1Oracle
2
0
1 Q
2
3
0
1
2



0.125
0.125
0.125
0.125
0.125
0.125
0.125
0.125
Quasi-probability
0.12
0.09
0.06
0.03
0.00
000
001
010
011
100
101
110
111

Conclusion -

- **Efficiency-Scalability Trade-off**: Grover's Search Algorithm offers a trade-off between efficiency and scalability when applied to problems with an unknown number of solutions. As the number of solutions increases, the number of required iterations grows, which impacts the quantum resources needed for execution.
- **Algorithmic Significance**:Grover’s Search Algorithm is a cornerstone in quantum computing, showcasing the power of quantum parallelism for solving search problems in unsorted databases.

ASSIGNMENT - 7

Theory -

1. Introduction to Quantum Error Correction

- Quantum error correction is a critical field in quantum computing aimed at protecting quantum information from errors caused by noise, decoherence, and imperfections in quantum hardware.

2. Quantum Error Correction Codes

- Quantum error correction codes, such as the Shor code, Steane code, and surface codes, are designed to detect and correct errors in quantum systems. These codes work by redundantly encoding quantum information in multiple qubits, allowing errors to be identified and corrected through syndrome measurements..

3. Workflow of Quantum Error Correction

- **Quantum Circuit Initialization**The quantum system is initialized into a state that encodes logical information into multiple physical qubits using a quantum error correction code.
- **Syndrome Measurement**:A series of measurements are made to detect errors. These measurements do not collapse the quantum state but instead provide information about potential errors affecting the qubits.
- **Error Detection and Correction**: After syndrome measurements, a classical post-processing step is performed to determine the errors and apply corrective operations.
- **Fault-Tolerant Computation**: Quantum error correction ensures that even if errors occur during the error-correction process, the system can still continue functioning correctly.

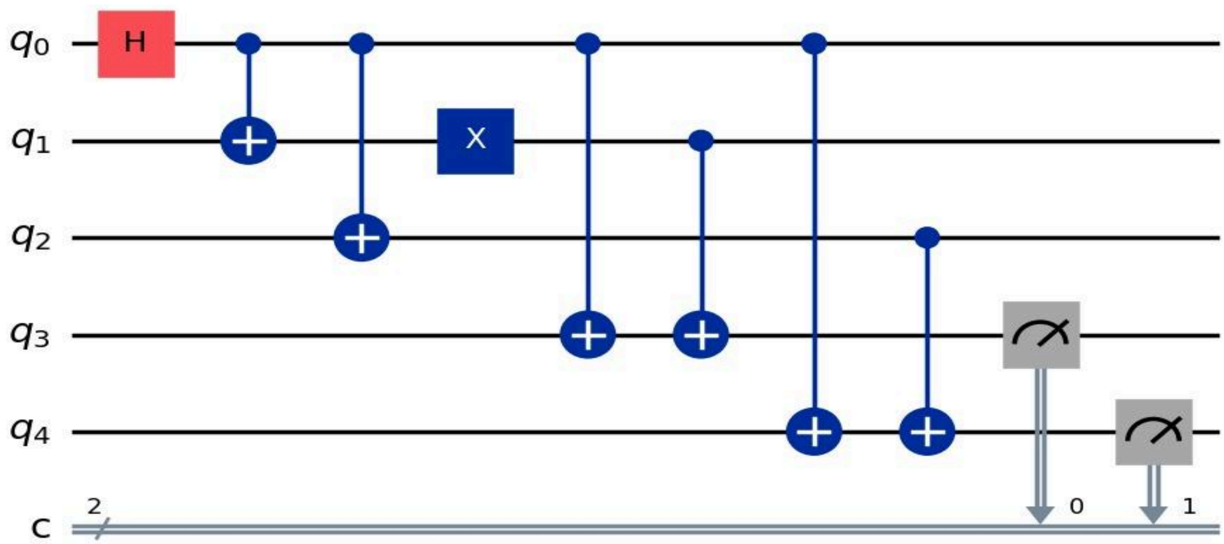
4. Importance of Quantum Error Correction

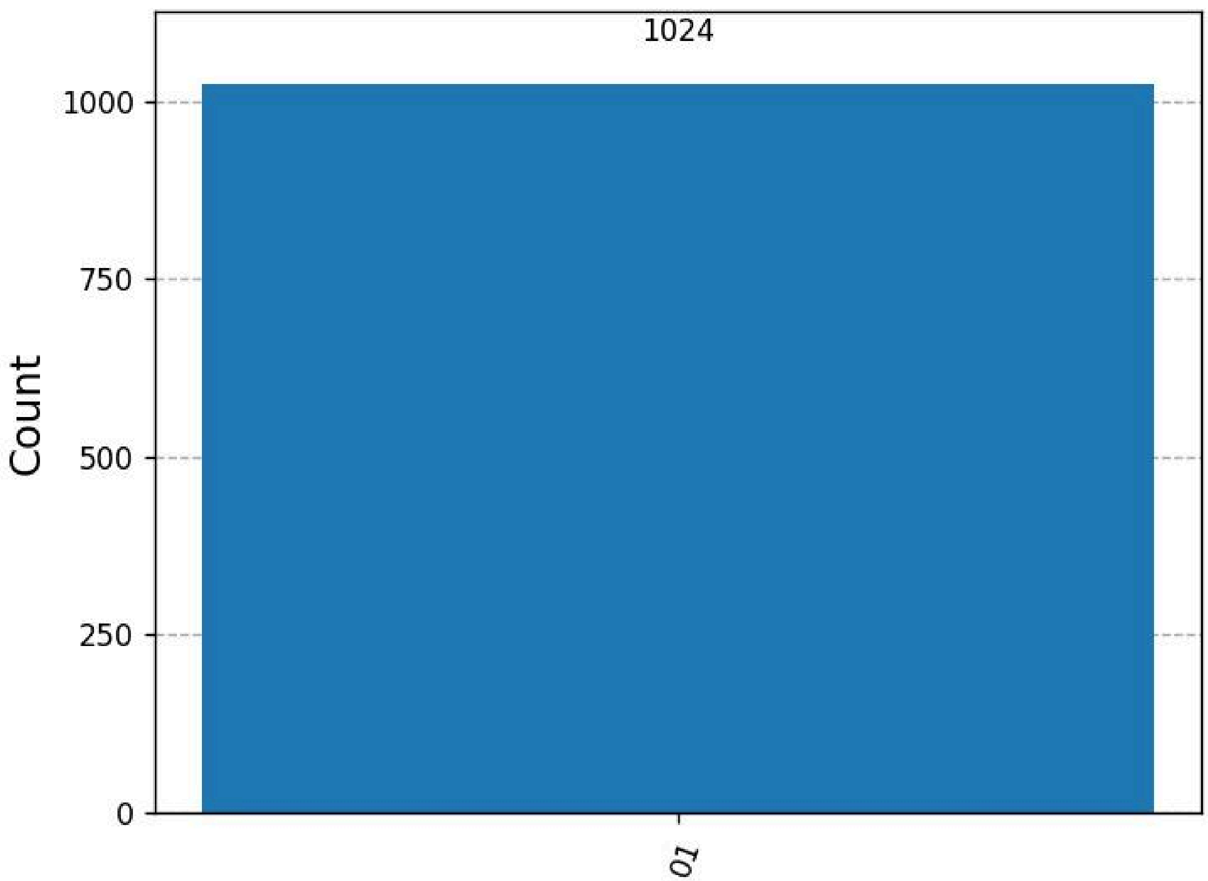
- Quantum error correction is essential for achieving practical quantum computing. It allows quantum algorithms to be executed on noisy intermediate-scale quantum (NISQ) devices by ensuring that errors do not propagate uncontrollably.

Code (Quantum Error Correction) -

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import Aer
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

# Create a circuit for the three-qubit bit-flip code
qc = QuantumCircuit(5, 2)  # 3 data qubits + 2 ancilla for syndrome measurement

# Encode |0⟩ state with redundancy: |0_L⟩ = |000⟩
qc.h(0)  # Prepare superposition state to test error correction
qc.cx(0, 1)
qc.cx(0, 2)

# Introduce an artificial bit-flip error on one qubit
qc.x(1)  # Flipping the second qubit to simulate an error

# Syndrome measurement to detect the error
qc.cx(0, 3)
qc.cx(1, 3)
qc.cx(0, 4)
qc.cx(2, 4)
qc.measure(3, 0)
qc.measure(4, 1)

# Visualize the circuit
qc.draw('mpl')
plt.show()

# Simulate and get results
backend = Aer.get_backend('aer_simulator')
transpiled_qc = transpile(qc, backend)
result = backend.run(transpiled_qc).result()
counts = result.get_counts()
plot_histogram(counts)
plt.show()
```

Output -

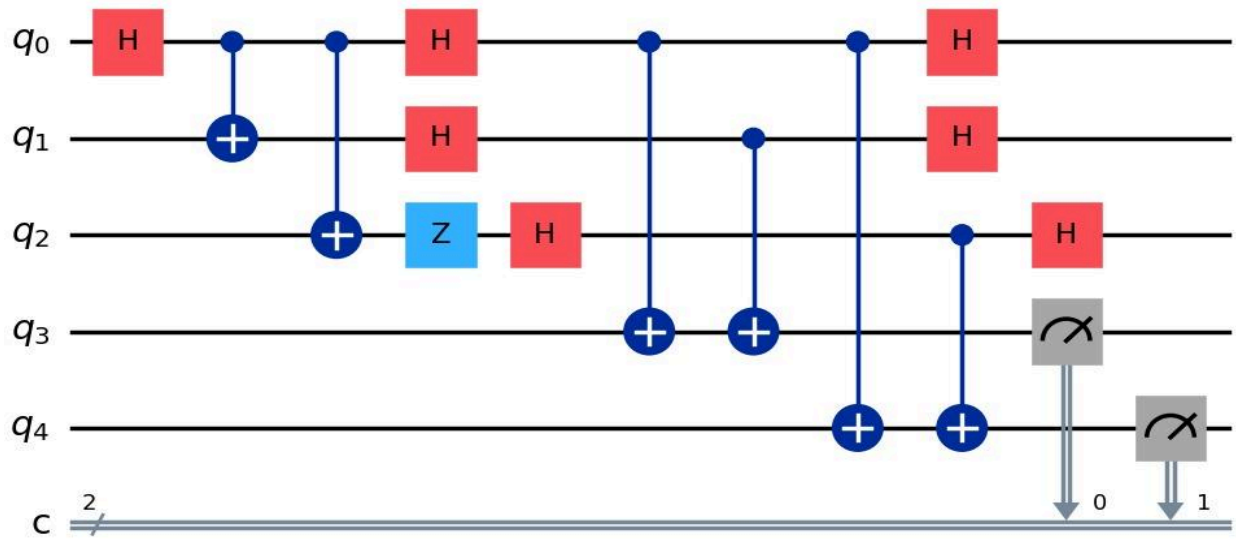


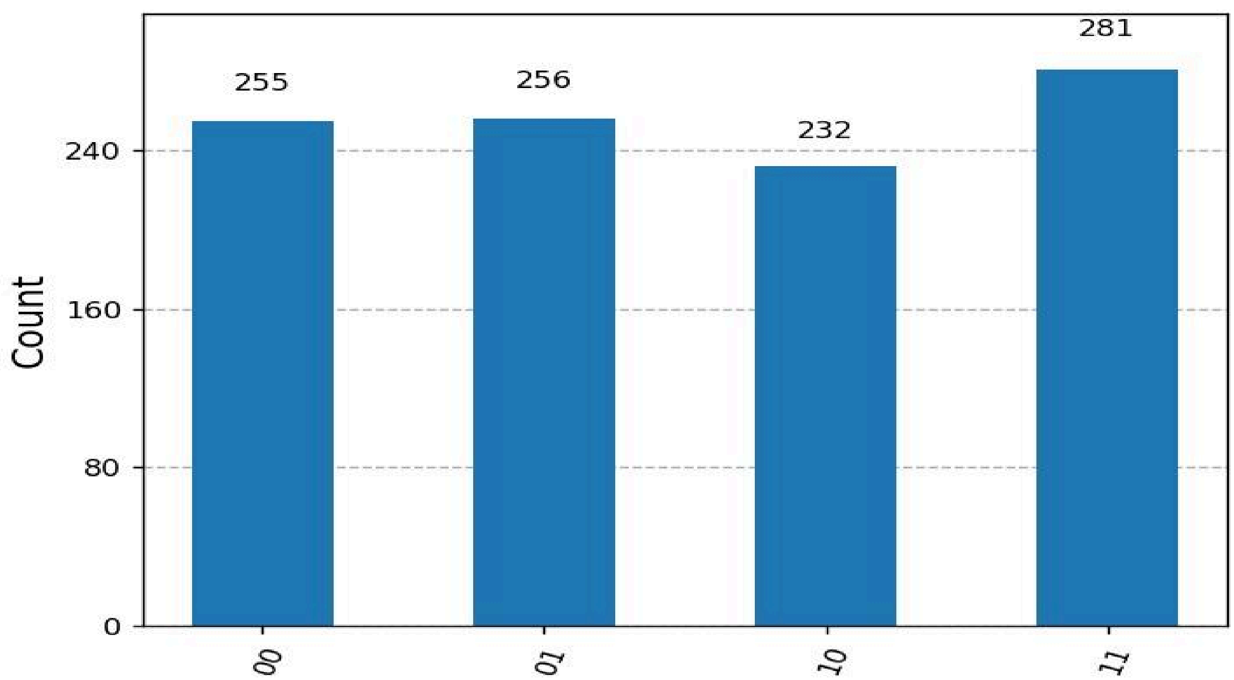


Code -

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import Aer
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

# Create a phase-flip code circuit (five qubits: 3 data qubits + 2 ancilla qubits)
qc = QuantumCircuit(5, 2)  # 3 data qubits + 2 ancilla qubits

# Encode the logical |0⟩ using a phase code: |0_L⟩ = (|000⟩ + |111⟩) / √2
qc.h(0)  # Apply Hadamard gate on qubit 0
qc.cx(0, 1)  # Apply CNOT gate between qubit 0 and qubit 1
qc.cx(0, 2)  # Apply CNOT gate between qubit 0 and qubit 2

# Introduce a phase-flip error on the third qubit (qubit 2)
qc.z(2)

# Syndrome measurement for phase-flip error detection
qc.h(0)
qc.h(1)
qc.h(2)
qc.cx(0, 3)
qc.cx(1, 3)
qc.cx(0, 4)
qc.cx(2, 4)
qc.h(0)
qc.h(1)
qc.h(2)

# Correct measurement of ancilla qubits
qc.measure(3, 0)  # Measure the ancilla qubit 3 to classical bit 0
qc.measure(4, 1)  # Measure the ancilla qubit 4 to classical bit 1

# Draw and display the circuit
qc.draw('mpl')
plt.show()

# Run simulation on the Aer simulator
backend = Aer.get_backend('aer_simulator')
transpiled_qc = transpile(qc, backend)
result = backend.run(transpiled_qc).result()
counts = result.get_counts()

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

Output -


q_0
q_1
q_2
q_3
q_4
c
2
H
Z
0
1


Count
255
256
232
281
0
80
160
240
00
01
10
11

Conclusion -

- **Efficiency-Scalability Trade-off**: Quantum error correction presents a trade-off between efficiency and scalability due to the significant overhead required for encoding and correcting errors. As the complexity of the quantum system increases, the number of qubits needed for error correction also grows, impacting the overall quantum resources required
- **Algorithmic Significance**:Quantum error correction is vital for ensuring the reliability of quantum computations, making it a foundational component of scalable quantum computing. By protecting quantum information from errors and decoherence, it enables the execution of long and complex quantum algorithms with high fidelity.

ASSIGNMENT - 8

Theory -

1. Introduction to Quantum Walk Search Algorithm

- The Quantum Walk Search Algorithm is a quantum computing paradigm inspired by the classical random walk. It enhances search efficiency in structured and unstructured datasets by leveraging quantum superposition and interference.

2. Types of Quantum Walks

- **Discrete-Time Quantum Walks**:These involve stepwise evolution determined by a unitary operator. They are particularly useful for algorithms on graphs and lattice structures.
- **Continuous-Time Quantum Walks**:These are governed by the time evolution of a quantum system under a Hamiltonian. They are better suited for certain combinatorial problems and searches.

3. Workflow of Quantum Error Correction

- **Initialization**: The algorithm initializes the quantum system in a superposition of all possible states, representing all potential solutions to the search problem.
- **Quantum Walk Evolution**: A quantum walk operator is applied iteratively, ensuring that the probability amplitude evolves across the search space while preserving quantum coherence.
- **Marking the Solution**: A phase oracle is used to mark the correct solution by modifying its amplitude, enabling its identification.
- **Amplitude Amplification**: Quantum interference is employed to amplify the probability of finding the correct solution while suppressing others.

4. Importance of Quantum Error Correction

- They outperform classical algorithms in specific scenarios, such as searching through unstructured data or solving graph traversal problems.
- By integrating with other quantum techniques, such as Grover's search or quantum annealing, quantum walks can address complex computational.

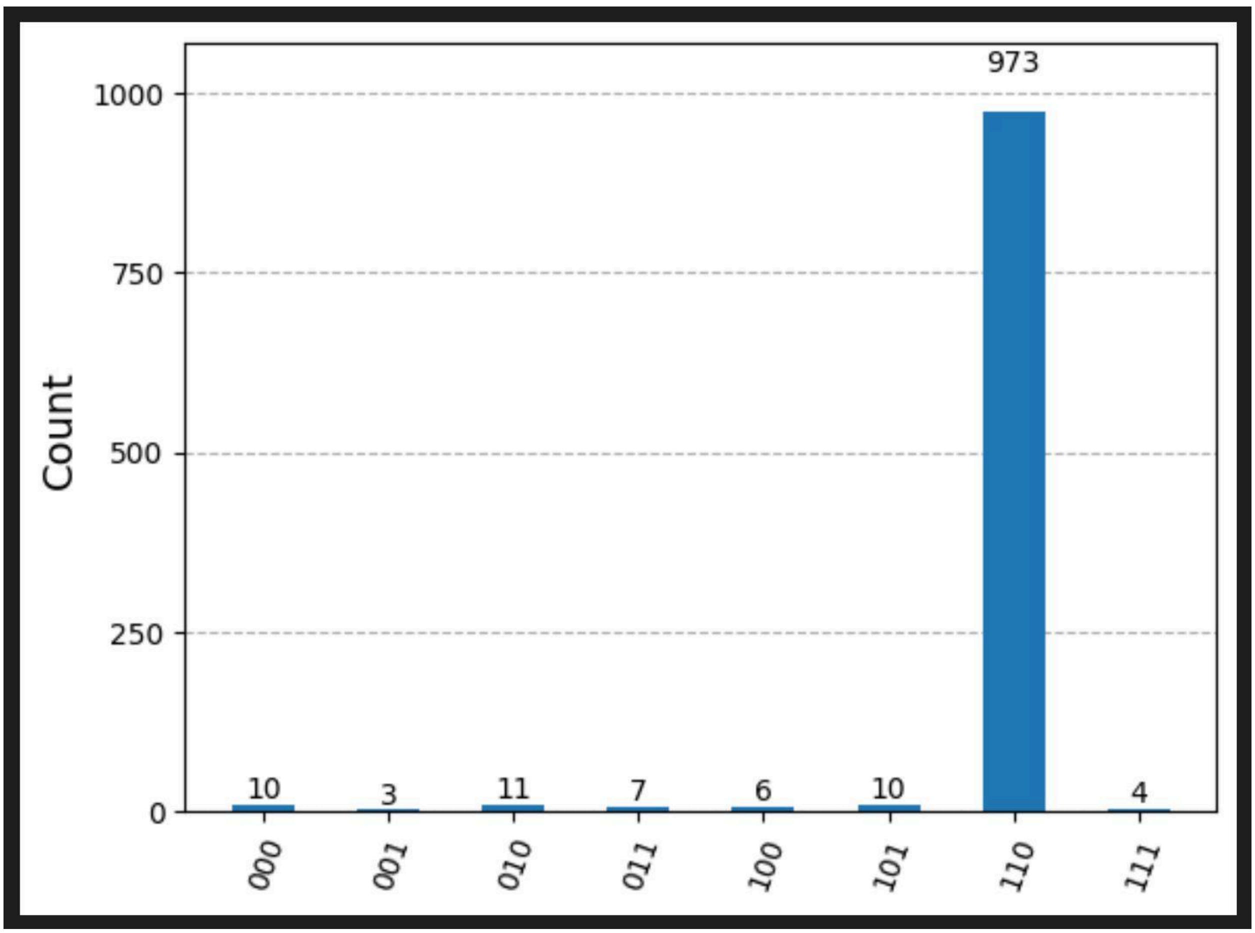
Code (Quantum Walk Search Algorithm) -

```python
from qiskit import QuantumCircuit, transpile
from qiskit_aer import Aer
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt
import numpy as np

# Function to create the Grover diffusion operator
def diffusion_operator(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)  # Multi-controlled Toffoli
    qc.h(n_qubits - 1)
    qc.x(range(n_qubits))
    qc.h(range(n_qubits))
    return qc.to_gate(label="Diffusion")

# Function to create the oracle (marks the solution state)
def oracle(n_qubits, marked_state):
    qc = QuantumCircuit(n_qubits)
    marked_state_bin = format(marked_state, f'0{n_qubits}b')
    for i, bit in enumerate(marked_state_bin):
        if bit == '0':
            qc.x(i)
    qc.h(n_qubits - 1)
    qc.mcx(list(range(n_qubits - 1)), n_qubits - 1)  # Multi-controlled Toffoli
    qc.h(n_qubits - 1)
    for i, bit in enumerate(marked_state_bin):
        if bit == '0':
```

```
            qc.x(i)
    return qc.to_gate(label="Oracle")

# Number of qubits and target state
n_qubits = 3  # Number of qubits
marked_state = 3  # Target state (e.g., |011⟩ -> decimal 3)

# Quantum Circuit for Quantum Walk Search
qc = QuantumCircuit(n_qubits)

# Initial superposition
qc.h(range(n_qubits))

# Visualize the quantum circuit after initial Hadamard gates
print("Quantum Circuit After Initial Superposition:")

# Render the quantum circuit in a matplotlib figure and display
fig = qc.draw(output='mpl')

# Display the circuit plot
plt.show()  # This will display the quantum circuit

# Number of Grover iterations
num_iterations = int(np.pi / 4 * np.sqrt(2 ** n_qubits))

for _ in range(num_iterations):
    # Apply oracle
    qc.append(oracle(n_qubits, marked_state), range(n_qubits))
    # Apply diffusion operator
    qc.append(diffusion_operator(n_qubits), range(n_qubits))

# Measurement
qc.measure_all()

# Simulate the circuit using AerSimulator
simulator = Aer.get_backend('aer_simulator')
transpiled_qc = transpile(qc, simulator)
result = simulator.run(transpiled_qc, shots=1024).result()
counts = result.get_counts()

# Plot the measurement results (final state probabilities)
print("Measurement Results:", counts)
plot_histogram(counts)
plt.show()
```

Output -

```
Quantum Circuit After Initial Superposition:
       ┌───┐
q_0: ┤ H ├
       ├───┤
q_1: ┤ H ├
       ├───┤
q_2: ┤ H ├
       └───┘
Measurement Results: {'001': 3, '110': 973, '100': 6, '010': 11, '000': 10, '101': 10, '111': 4, '011': 7}
```



Conclusion -

- **Efficiency-Scalability Trade-off**:Quantum walk search algorithms also exhibit an efficiency-scalability trade-off. While they provide significant speedups for specific search problems, their implementation demands complex quantum circuitry and precise control over qubits.
- **Algorithmic Significance**:Quantum walk search algorithms are pivotal for advancing the capabilities of quantum computing. They showcase the power of quantum mechanics in solving computational problems more efficiently than classical methods, especially in structured and combinatorial search spaces.

ASSIGNMENT - 9

Theory -

1. Introduction to Superdense Coding

- Superdense coding is a quantum communication protocol that allows the transmission of two classical bits of information using a single quantum bit (qubit) with the help of quantum entanglement. It is a fundamental demonstration of quantum information theory, showcasing the unique advantages of quantum communication over classical methods.

2. Key Elements of Superdense Coding

- **Entanglement**: The protocol relies on a shared entangled state between the sender (Alice) and the receiver (Bob), typically a Bell state.
- **Quantum Operations**: Alice performs specific quantum operations (Pauli operators) on her qubit to encode two classical bits of information.
- **Measurement**: Bob decodes the message by performing a joint measurement on both qubits to retrieve the classical information.

3. Workflow of Superdense Coding

- **Entanglement Sharing**: Alice and Bob begin by sharing a pair of entangled qubits. Bob retains one qubit, and Alice takes the other.
- **Encoding the Message**: To send two classical bits, Alice applies a quantum operation based on the message she wishes to encode.
  - `00`: No operation (Identity).
  - `01`: Apply XXX (bit-flip).
  - `10`: Apply ZZZ (phase-flip).
  - `11`: Apply X⋅ZX \cdot ZX⋅Z (bit-flip and phase-flip).
- **Transmission**: Alice sends her qubit to Bob after encoding.
- **Decoding the Message**: Upon receiving Alice’s qubit, Bob performs a joint measurement (Bell-state measurement) on both qubits to determine the original two-bit message.

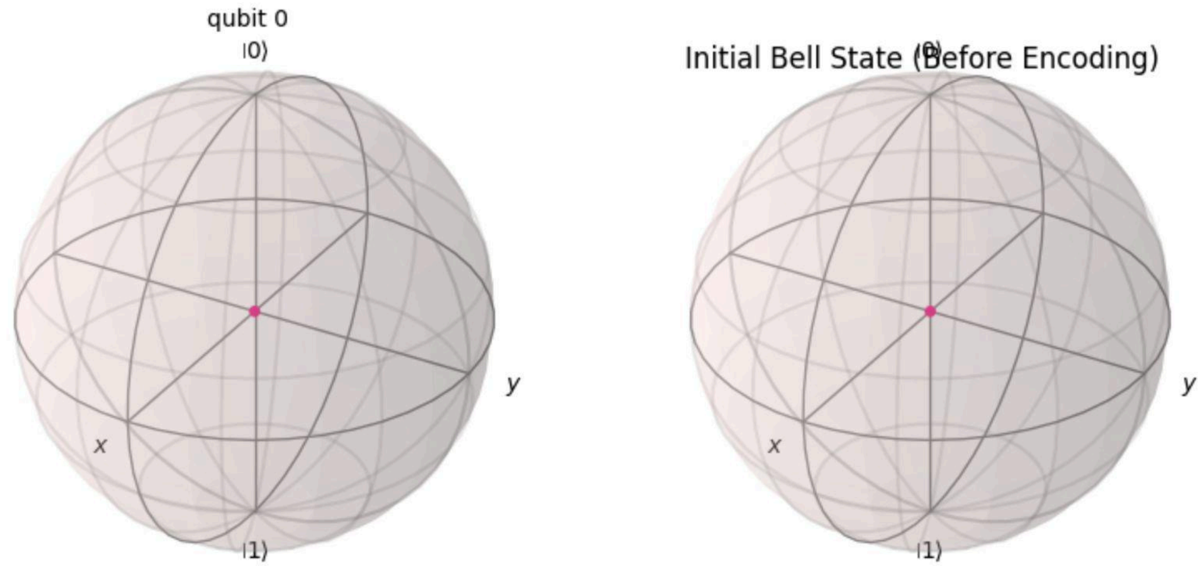
Code (SuperDense Coding) -

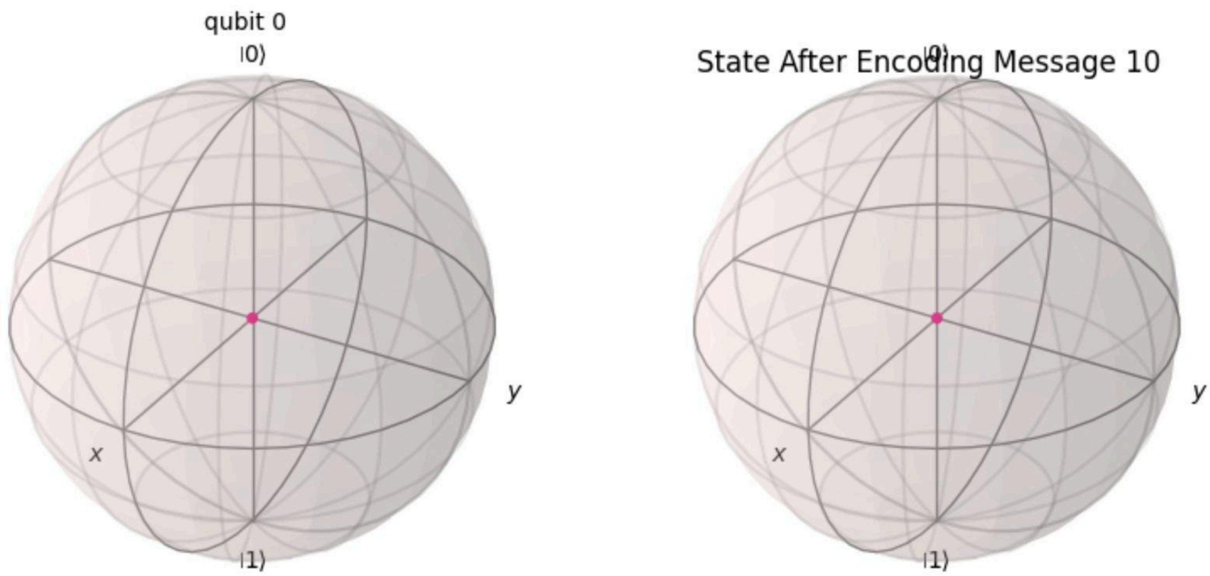
```python
from qiskit import QuantumCircuit
from qiskit_aer import Aer
from qiskit.visualization import plot_histogram, plot_bloch_multivector
import matplotlib.pyplot as plt

# Step 1: Create a Bell State (entangled state)
def create_bell_pair():
    qc = QuantumCircuit(2)
    qc.h(0) # Apply Hadamard gate to qubit 0
    qc.cx(0, 1) # Apply CNOT gate (control: qubit 0, target: qubit 1)
    return qc

# Step 2: Encode the message (classical bits) onto the Bell pair
def encode_message(qc, message):
    if message == "00":
        pass # Do nothing
    elif message == "01":
        qc.x(0) # Apply X gate
    elif message == "10":
        qc.z(0) # Apply Z gate
    elif message == "11":
        qc.x(0)
        qc.z(0)
    else:
        raise ValueError("Message must be one of '00', '01', '10', or '11'")

# Step 3: Decode the message by reversing the entanglement
def decode_message(qc):
    qc.cx(0, 1) # Apply CNOT gate (control: qubit 0, target: qubit 1)
    qc.h(0) # Apply Hadamard gate to qubit 0
```

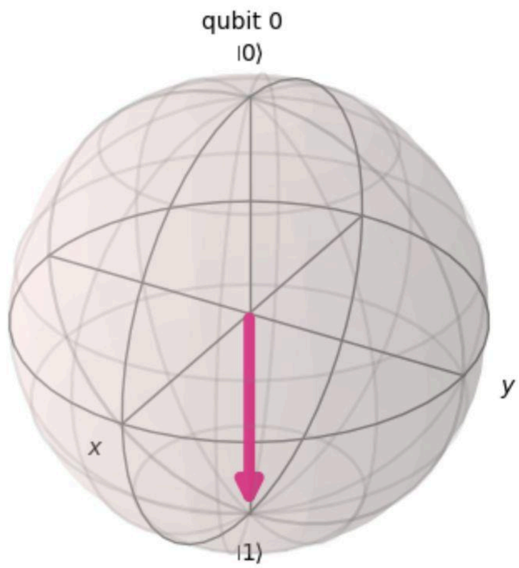
```
# Main function to demonstrate superdense coding
def superdense_coding(message):
    if message not in ["00", "01", "10", "11"]:
        raise ValueError("Message must be one of '00', '01', '10', or '11'")

    # Step 1: Create a Bell pair
    qc = create_bell_pair()

    # Visualize the initial state of the qubits in the Bell state
    print("Initial Bell state (before encoding):")
    backend = Aer.get_backend('statevector_simulator')
    job = backend.run(qc)
    result = job.result()
    statevector = result.get_statevector(qc)
    plot_bloch_multivector(statevector)
    plt.title("Initial Bell State (Before Encoding)")
    plt.show()

    # Step 2: Encode the message
    encode_message(qc, message)

    # Visualize the state after encoding the classical message
    job = backend.run(qc)
    result = job.result()
    statevector = result.get_statevector(qc)
    plot_bloch_multivector(statevector)
    plt.title(f"State After Encoding Message {message}")
    plt.show()
```

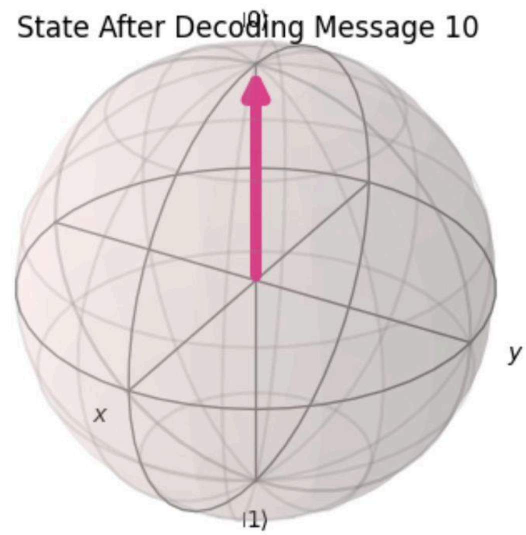
```
    # Step 3: Decode the message
    decode_message(qc)

    # Visualize the state after decoding (before measurement)
    job = backend.run(qc)
    result = job.result()
    statevector = result.get_statevector(qc)
    plot_bloch_multivector(statevector)
    plt.title(f"State After Decoding Message {message}")
    plt.show()

    # Step 4: Measure the qubits
    qc.measure_all()

    # Simulate the circuit and get the results
    simulator = Aer.get_backend('qasm_simulator')
    job = simulator.run(qc, shots=1024)
    result = job.result()
    counts = result.get_counts(qc)

    return qc, counts

# Test the superdense coding protocol
message = "10" # Replace with "00", "01", "10", or "11"
qc, counts = superdense_coding(message)

# Display the quantum circuit and the result
print(f"Message sent: {message}")
print(f"Measurement result: {counts}")
qc.draw("mpl")

# Plot histogram of results
plot_histogram(counts)
plt.show()
```

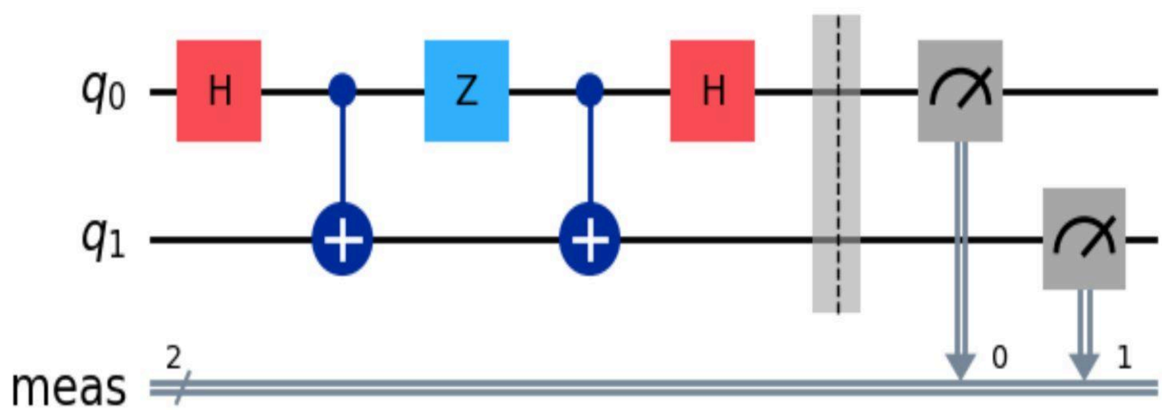
Output -

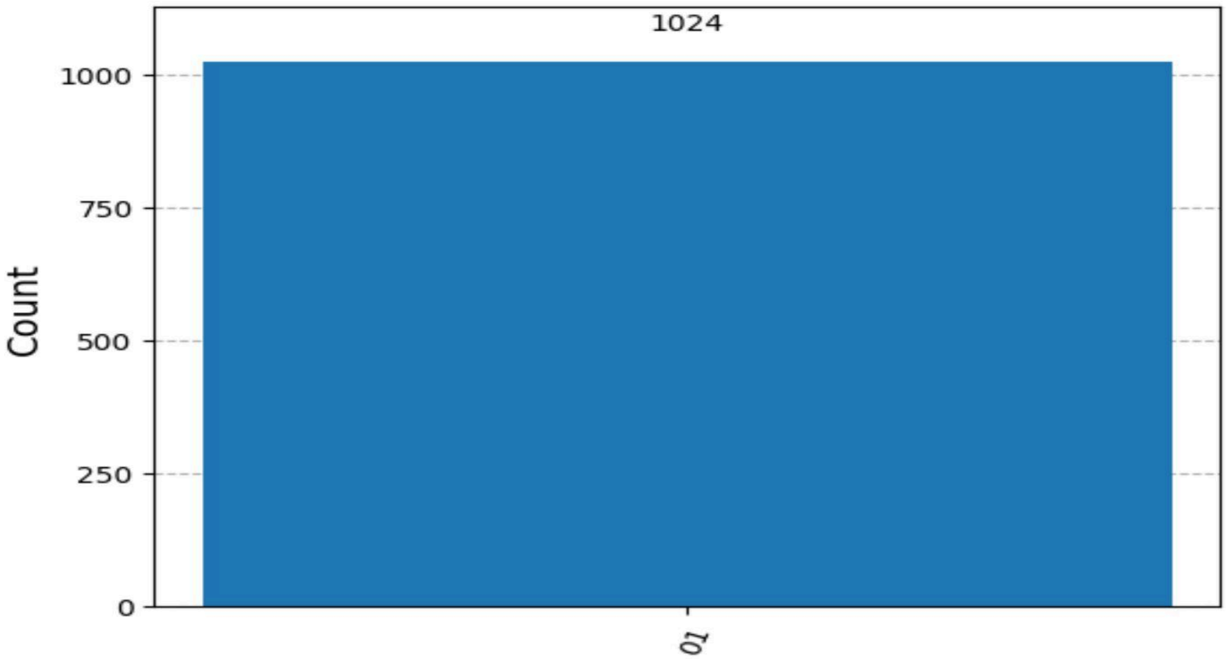

qubit 0
|0⟩
Initial Bell State (Before Encoding)
x
y
|1⟩


qubit 0
|0⟩
State After Encoding Message 10
x
y
|1⟩


qubit 0
|0⟩
x
y
|1⟩


State After Decoding Message 10
|0⟩
x
y
|1⟩


q0
q1
meas
H
Z
H
2
0
1



Conclusion -

- **Efficiency-Scalability Trade-off:** Superdense coding demonstrates an efficiency-scalability trade-off. While it enables the transmission of two classical bits using a single qubit, the protocol depends on the availability of high-quality entangled states and reliable quantum channels.
- **Algorithmic Significance**: Superdense coding is a foundational protocol in quantum communication, showcasing the power of entanglement to enhance classical information transfer. It enables efficient use of quantum resources and forms the basis for advanced communication techniques, playing a critical role in the development of secure quantum networks and distributed quantum systems.

ASSIGNMENT - 10

Theory -

1. Introduction to TSP and Quantum Phase Estimation

- The Traveling Salesman Problem (TSP) is a classical optimization problem where the goal is to find the shortest route that visits a set of cities and returns to the starting point. Solving TSP becomes computationally challenging as the number of cities increases. Quantum computing offers promising approaches, including the QPE algorithm, to address such problems.

2. Key Elements of Quantum Phase Estimation for TSP

- **Hamiltonian Representation**: TSP is encoded into a cost Hamiltonian whose ground state represents the optimal solution.
- **Unitary Operator**: A unitary operator UUU is derived to encode the eigenvalues (costs) corresponding to potential solutions.
- **QPE Algorithm**: QPE estimates the eigenvalues of UUU, allowing identification of the optimal solution.

3. Workflow of Solving TSP Using QPE

- **Encoding the Problem**: The TSP is mapped into a quantum system using the following steps:
  - **City Representation**: Cities and routes are encoded as quantum states.
  - **Cost Hamiltonian Construction**: A Hamiltonian is built to represent the total travel cost for each route.
- **Quantum State Preparation**:
  - Prepare an initial quantum state as a uniform superposition of all possible routes using Hadamard gates.
  - Implement a trial phase to approximate the ground state of the cost Hamiltonian
- **Phase Estimation**: Use ancilla qubits to store the phase (related to eigenvalues).

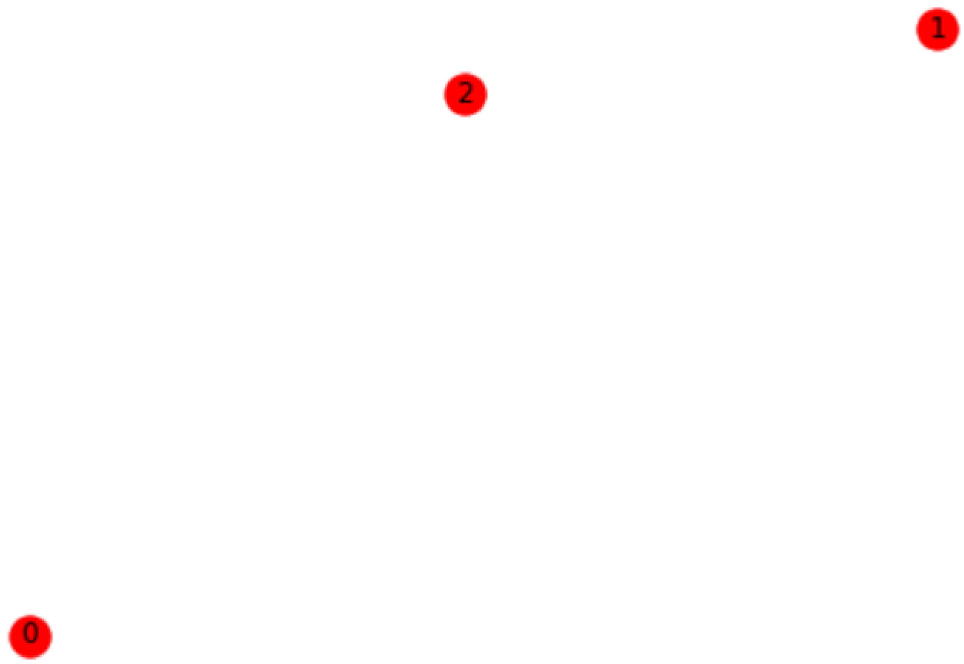
Code -

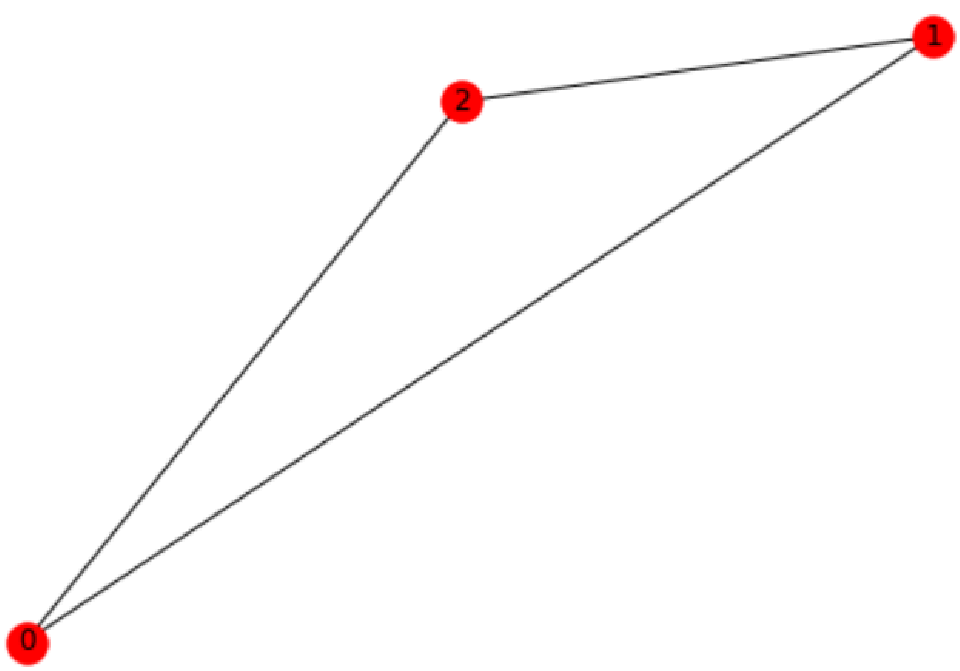
```
import matplotlib.pyplot as ply
%matplotlib inline

import networkx as nx
import numpy as np

from qiskit_aqua.translators.ising import tsp
from qiskit_aqua.input import EnergyInput
from qiskit_aqua import run_algorithm
from qiskit_qcgpu_provider import QCGPUProvider
```

```
locations = 3

problem = tsp.random_tsp(locations)
positions = {k: v for k, v in enumerate(problem.coord)}

G = nx.Graph()
G.add_nodes_from(np.arange(0, locations, 1))
nx.draw(G, with_labels=True, pos=positions)
```

```
best_distance, best_order = brute_force(problem.w, problem.dim)
draw(G, best_order, positions)
```

Output -





Code -

```
operator, offset = tsp.get_tsp_qubitops(problem)
algorithm_input = EnergyInput(operator)

algorithm_parameters = {
    'problem': { 'name': 'ising', 'random_seed': 23 },
    'algorithm': { 'name': 'VQE', 'operator_mode': 'matrix' },
    'optimizer': { 'name': 'SPSA', 'max_trials':100 },
    'variational_form': {'name': 'RY', 'depth': 5, 'entanglement': 'linear'}
}
```

```
backend = QCGPUProvider().get_backend('statevector_simulator')
%time result_qiskit = run_algorithm(algorithm_parameters, algorithm_input)
%time result = run_algorithm(algorithm_parameters, algorithm_input, backend=backend)
```

```
#print('tsp objective:', result['energy'] + offset)
x = tsp.sample_most_likely(result['eigvecs'][0])
print('feasible:', tsp.tsp_feasible(x))
z = tsp.get_tsp_solution(x)
print('solution:', z)
print('solution objective:', tsp.tsp_value(z, problem.w))
draw(G, z, positions)
```

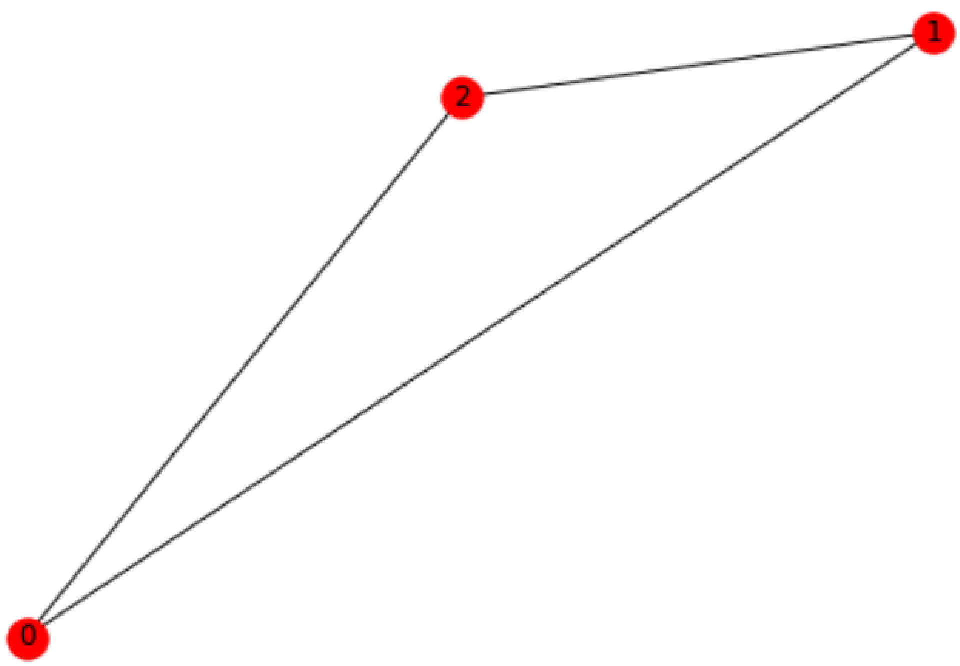
```python
# Utitlity Functions
def draw(G, order, positions):
    G2 = G.copy()
    n = len(order)
    for i in range(n):
        j = (i + 1) % n
        G2.add_edge(order[i], order[j])
    nx.draw(G2, pos=positions, with_labels=True)

# Classically solve the problem using a brute-force method
from itertools import permutations

def brute_force(weights, N):
    a = list(permutations(range(1, N)))
    best_distance = None
    for i in a:
        distance = 0
        pre_j = 0
        for j in i:
            distance += weights[j, pre_j]
            pre_j = j
        distance += weights[pre_j, 0]
        order = (0,) + i
        if best_distance is None or distance < best_distance:
            best_order = order
            best_distance = distance

    return best_distance, best_order
```

```python
import warnings
warnings.filterwarnings('ignore')
```

Output -

```
CPU times: user 32.1 s, sys: 127 ms, total: 32.3 s
Wall time: 36.8 s
CPU times: user 22.9 s, sys: 241 ms, total: 23.1 s
Wall time: 23.1 s
```

```
feasible: True
solution: [2, 1, 0]
solution objective: 203.0
```



Conclusion -

- **Efficiency-Scalability Trade-off:** Quantum Phase Estimation (QPE) offers significant efficiency advantages in solving optimization problems like TSP by leveraging quantum parallelism and precise eigenvalue computation.
- **Algorithmic Significance**: QPE demonstrates the potential of quantum algorithms to address combinatorial optimization problems effectively. By exploiting quantum properties such as superposition and entanglement, QPE enables the exploration of exponentially large solution spaces