PES UNIVERSITY

Karnataka, Bengaluru – 560080

Department of Electronics and Communications

Course Name: Quantum Entanglement and Quantum Computation

Course Code: UE22EC343AB3

ESA - REPORT

Project Title:

"Realizing the Quantum circuit to implement Durr-Hoyer algorithm".

Team Composition:

SHRINIVASA K - PES1UG22EC281

Course Instructor:

Dr. Kaustav Bhowmick ECE Dept.

Abstract:

This report examines the Durr-Hoyer algorithm, a quantum computational approach for finding the minimum element in an unsorted list. By leveraging quantum superposition and Grover's quantum search algorithm, the Durr-Hoyer algorithm introduces a remarkable enhancement in efficiency, cutting down the number of required steps or operations significantly. This paper provides an overview of the algorithm's working, theoretical foundations and implementation. The algorithm efficiently identifies the minimum value through iterative threshold updates, ensuring probabilistic convergence. It demonstrates the practical application of Grover's algorithm as a subroutine within a broader quantum framework. The theoretical complexity is backed by rigorous proofs, establishing it as near-optimal for unstructured search problems. Furthermore, its implementation highlights the feasibility of using quantum oracles and amplitude amplification in practical scenarios.

Introduction:

Finding the minimum value in an unsorted dataset is a fundamental computational problem with applications in diverse fields, including optimization, data analysis, and artificial intelligence. Classical algorithms require a linear number of probes or steps for this task, as each element must be compared. However, quantum computing provides a more efficient solution.

Quantum algorithms leverage phenomena such as superposition and interference to explore multiple possibilities simultaneously. The Durr-Hoyer algorithm builds upon Grover's quantum search algorithm, combining it with exponential search techniques to reduce the complexity of the minimum searching problem to. This reduction is near-optimal, given the quantum lower bounds for unstructured search problems.

The importance of this algorithm extends beyond its efficiency; it also demonstrates the practical implementation of quantum oracles and iterative updates in computational problems. By iteratively refining the search threshold, the algorithm converges to the minimum value with a high probability, highlighting the effective integration of quantum principles into computational algorithms.

Implementation and Working of the Durr-Hoyer Algorithm:

1. **Understand the Problem Domain:**

    - Define the unsorted array T of size N whose minimum value needs to be found.
    - Ensure T contains distinct or comparable elements.

2. **Set Up the Quantum Environment**:

    Choose a quantum computing platform, such as:

- IBM Quantum Experience with Qis kit.
- Righetti's Forest SDK

3.Algorithm Implementation:

Step 1: Initialize the Quantum Registers:

- Create a quantum register for representing indices j.
- Initialize a second quantum register to store the current threshold index y.
- Prepare a superposition of all indices |j⟩:

$$\frac{1}{\sqrt{N}}\sum_{j=0}^{N-1}|j\rangle|y\rangle$$

Step 2: Mark the Values lower than the Threshold value:

- Apply an **oracle function** to mark all indices j where T[j]<T[y]. Use a quantum comparison operation to compare values efficiently.

Step 3: Grover's Search (Quantum Exponential Search):

- Implement Grover's search algorithm:
    1. Apply the Grover diffusion operator:

$$I-2|\psi\rangle\langle\psi|$$

    2. Amplify the amplitudes of marked indices.
    3. Measure to find a new candidate index y'.

Step 4: Update Threshold:

- If T[y'] < T[y], update y to y'. Otherwise, retain the previous y.

Step 5: Repeat Until Timeout;

- Repeat steps 2–4 until a predefined number of iterations or a timeout is reached.

Step 6: Measure and Return the Result:

- Measure the state of the threshold register |y⟩ to determine the index of the minimum value.

4. Verification and Testing:

- Run the algorithm on a simulator to test correctness.
- Compare the results with classical methods for small datasets to verify accuracy.

Result:

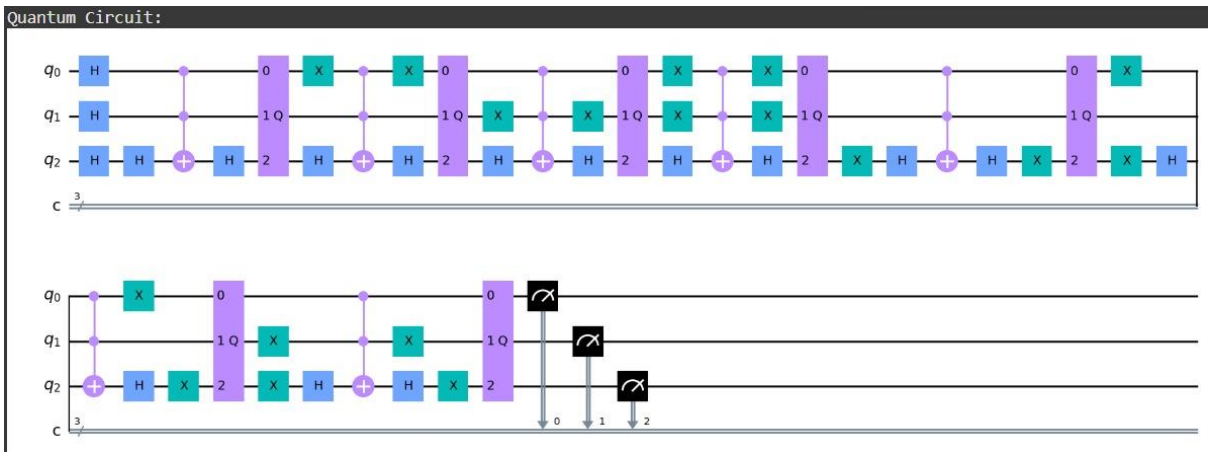

Fig 1: Quantum circuit for Durr-Hoyer algorithm

Description of the Circuit

1. **Initialization**:
   - The circuit initializes with three qubits (q0, q1, q2) and three classical bits (c).
   - All qubits are placed in a uniform superposition using Hadamard gates (H).
2. **Oracle Application**:
   - Controlled gates (purple boxes) encode the oracle function. The oracle marks the states corresponding to values below a certain threshold by flipping their phases. This corresponds to identifying elements in the list that meet a specific condition (e.g., values less than the current threshold).
3. **Grover Diffusion**:
   - After the oracle application, the Grover diffusion operator is applied:
     - Another set of Hadamard gates (H) brings the qubits back to the computational basis.
     - Inversion around the mean is performed by flipping the amplitudes about the average (visible as controlled operations).
4. **Repetition**:
   - The oracle and Grover diffusion operations are repeated multiple times (visible as multiple blocks of similar gates in the circuit). The number of repetitions depends on the desired amplification of the correct states.
5. **Measurement**:
   - After the final iteration, the qubits are measured, and the outcomes are stored in the classical bits (c).
   - The measurement collapses the superposition, providing the index of the state corresponding to the minimum value.

Interpretation of the Circuit

- **Top Circuit**: The first part shows the repeated application of the oracle and Grover diffusion operator, iteratively narrowing down the possible candidates for the minimum.
- **Bottom Circuit**: The bottom part visualizes the same operations but includes measurement gates (M), ensuring that the outcome of the quantum computation is stored in the classical register.

The graph shows the measurement outcomes of the Durr-Hoyer algorithm. The tallest bars correspond to the indices of the minimum value, indicating the algorithm's success in amplifying the probability of the correct result.

```
Index of the Minimum Value: 011
Minimum Value in the Array: 1

Measurement Counts:
 Outcome 110: 9 counts
 Outcome 100: 4 counts
 Outcome 101: 12 counts
 Outcome 001: 11 counts
 Outcome 111: 14 counts
 Outcome 011: 530 counts
 Outcome 000: 380 counts
 Outcome 010: 64 counts
```

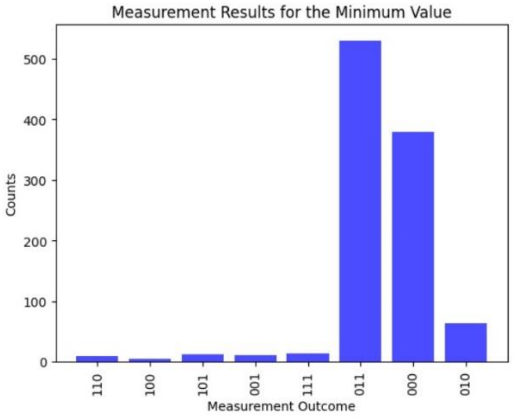

Fig 2: (a)output with min value and index of min value in binary (b) corresponding Bar graph between measurement of each value & counts

FPGA implementability: This output represents the FPGA-based (e.g., zed Board) implementation of the Durr-Hoyer algorithm, a quantum-inspired method for finding the minimum value in an unsorted dataset. The result indicates that the index of the minimum value is 3 and the corresponding value is 1, validating the algorithm's effectiveness in solving unstructured search problems. By utilizing quantum principles such as superposition, interference, and amplitude amplification, the algorithm reduces computational complexity compared to classical approaches. Implementing this on FPGA highlights its real-world applicability, demonstrating how quantum-inspired algorithms can be adapted to classical hardware systems for enhanced computational efficiency. This implementation serves as a step toward integrating quantum algorithms with hardware platforms, paving the way for advancements in optimization, data processing, and artificial intelligence applications.

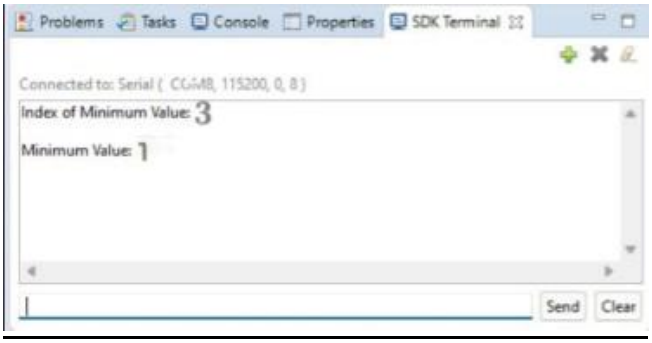

Fig 3: FPGA-Based Output of Durr-Hoyer Algorithm for Minimum Value Identification.

Thus the output from the qi skit implementation of Durr-Hoyer algorithm matches with the output of the FPGA's SDK terminal window.

Theoretical Justification:

The quantum speedup in this algorithm is based on Grover's search principle, which can locate a marked element within O(√N/t) steps, where t is the number of marked items. By iteratively refining the threshold, the algorithm ensures that the probability of selecting the minimum value increases over time.

Key features:

1. **Probability of Selecting the Minimum:** At any point, the algorithm selects an index with probability proportional to the inverse of its rank among elements less than the current threshold.

Lemma: Let p (t, r) be the probability that the index of the element of rank r will ever be chosen when infinite algorithm searches among t elements.
Then p (t, r) =1/r if r ≤ t, and p (t, r) =0 otherwise.

2. **Expected Time to Find the Minimum:** The expected number of quantum steps to locate the minimum value is bounded by: $O(\sqrt{N} + \log^2 N)$

Analysis of Algorithm Performance:

The algorithm's runtime derives from the combination of:

- The cost of Grover's search to locate a marked element.
- The logarithmic cost of refining the threshold iteratively.

Given a uniformly random initialization and iterative refinement, the algorithm probabilistically converges to the minimum value with at least 50% success probability. This probability can be amplified by repeating the algorithm multiple times.

Mathematical Foundations:

1.Quantum Superposition: The algorithm begins by preparing a quantum state:

$$|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} |j\rangle|y\rangle,$$

where:

- $|j\rangle$: Represents the index j in the unsorted list
- $|y\rangle$: Represents the current threshold index.
- $1/\sqrt{N}$: Ensures the quantum state is a normalized uniform superposition over all indices. This allows simultaneous processing of all indices.

2.Marking and Searching: A quantum oracle mark indices j where T[j]<T[y]. Grover's search amplifies these marked indices, requiring $O(\sqrt{N}/t)$ iterations, where t is the count of marked indices.

3.Probability of Success: The probability of selecting the global minimum improves with each iteration of the algorithm. If the algorithm is repeated ccc times independently, the success probability of finding the global minimum is:

$$P(\text{success}) = 1 - (1/2^c).$$

This formula reflects the cumulative probability improvement due to multiple independent runs. Here:

- c: Number of repetitions of the algorithm.
- 1/2: Represents the base success probability of a single run of the algorithm.

By increasing c, the probability of finding the correct minimum approaches 1.

Conclusion:

The Durr-Hoyer algorithm concludes with the observation that its probability of success can be significantly improved by running the algorithm multiple times. After c repetitions, the probability of finding the minimum increases to $1 - (1/2^c)$ ensuring a high likelihood of success. Additionally, the algorithm remains effective even when the table contains non-distinct values, with minor adjustments ensuring that the success probability still adheres to the established bounds. This flexibility and efficiency highlight the robustness of the quantum approach to solving the minimum searching problem.

References:

[1] C. Durr and P. Hoyer, "A quantum algorithm for finding the minimum," *arXiv preprint arXiv: quant-ph./9607014v2*, Jan. 1999.

[2] C. H. Bennett, E. Bernstein, G. Brassard, and U. Vazirani, "Strengths and weaknesses of quantum computing," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1510–1523, 1997.

[3] M. Boyer, G. Brassard, P. Høyer, and A. Tapp, "Tight bounds on quantum searching," *Fortschritte der Physik*, 1998.

[4] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. 28th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 212–219, 1996.

APPENDIX

Implementation using Qiskit :

```
import matplotlib.pyplot as plt
import numpy as np
from qiskit import Aer, ClassicalRegister, QuantumCircuit, QuantumRegister,
execute
from qiskit.circuit.library import GroverOperator

def create_oracle(threshold, num_qubits, index_register):
    """Creates a quantum oracle that marks indices below the threshold."""
    oracle = QuantumCircuit(index_register)
    binary_threshold = f"{threshold:0{num_qubits}b}"
    for i, bit in enumerate(reversed(binary_threshold)):
        if bit == "0":
            oracle.x(index_register[i])

    # Apply logic to build oracle (multi-controlled Z gate)
    oracle.h(index_register[-1])
    oracle.mcx(list(range(num_qubits - 1)), index_register[-1])
    oracle.h(index_register[-1])
    oracle.name = f"Oracle (Threshold={threshold})"
    return oracle

def durr_hoyer(num_items, example_array):
    """Implements the Dürr-Høyer algorithm and visualizes results."""
    num_qubits = int(np.ceil(np.log2(num_items)))

    # Initialize registers
    index_register = QuantumRegister(num_qubits, "q")
    classical_register = ClassicalRegister(num_qubits, "c")
    combined_circuit = QuantumCircuit(index_register, classical_register)

    # Initialization step
    combined_circuit.h(index_register)

    # Start with maximum value as initial guess for minimum
    min_value = float("inf")
    min_index = None
    final_counts = None  # To store the counts of the final iteration

    for threshold in range(num_items - 1, -1, -1):  # Iterate through all
possible thresholds
        oracle = create_oracle(threshold, num_qubits, index_register)
        grover_op = GroverOperator(oracle=oracle)

        # Apply the Grover operator
        combined_circuit.append(grover_op, index_register)

        # Add measurements to test results for this threshold (not part of the
combined circuit)
        temp_circuit = combined_circuit.copy()
        temp_circuit.measure(index_register, classical_register)

        # Simulation
        simulator = Aer.get_backend("qasm_simulator")
        result = execute(temp_circuit, simulator, shots=1024).result()
        counts = result.get_counts()

        # Store the counts for visualization if this is the best threshold so far
        if counts:
            most_frequent_index = max(counts, key=counts.get)
            current_value = example_array[int(most_frequent_index, 2)]
            if current_value < min_value:
```

```
                min_value = current_value
                min_index = most_frequent_index
                final_counts = counts  # Update with the best iteration's counts

    if min_index is None:
        print("No results found satisfying the oracle's condition.")
        return

    # Display the combined quantum circuit
    #print("\nFinal Combined Quantum Circuit:")
    combined_circuit.measure(index_register, classical_register)  # Add final
measurement
    combined_circuit.draw("mpl", scale=0.8)  # Display the circuit
    plt.show()

    # Display the results
    print("\nIndex of the Minimum Value:", min_index)
    print("Minimum Value in the Array:", min_value)

    # Display counts
    print("\nMeasurement Counts:")
    for outcome, count in final_counts.items():
        print(f" Outcome {outcome}: {count} counts")

    # Plot histogram of the final counts
    plt.bar(final_counts.keys(), final_counts.values(), color='blue', alpha=0.7)
    plt.xlabel('Measurement Outcome')
    plt.ylabel('Counts')
    plt.title('Measurement Results for the Minimum Value')
    plt.xticks(rotation=90)
    plt.show()

# Example usage
num_items = 8  # Number of items in the list (or range of values)
example_array = [5, 2, 8, 1, 7, 3, 6, 4]  # Example array
durr_hoyer(num_items, example_array)
# Visualize the circuit
print("\nQuantum Circuit:")
#print(circuit.draw(output='text'))

# Display the circuit as a diagram
circuit.draw('mpl')
```

C – CODE for implementing on FPGA:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <complex.h>
#include <limits.h>

typedef double complex qampl;

void hadamard(qampl *state, int qubit, int num_qubits) {
   int num_states = 1 << num_qubits;
   for (int i = 0; i < num_states; i++) {
      if ((i >> qubit) & 1) {
         qampl temp = state[i];
         state[i] = (temp + state[i ^ (1 << qubit)]) / sqrt(2);
         state[i ^ (1 << qubit)] = (temp - state[i ^ (1 << qubit)]) / sqrt(2);
      }
   }
}
```

```
void cnot(qampl *state, int control, int target, int num_qubits) {
   int num_states = 1 << num_qubits;
   for (int i = 0; i < num_states; i++) {
      if (((i >> control) & 1) && !((i >> target) & 1)) {
         qampl temp = state[i];
         state[i] = state[i ^ (1 << target)];
         state[i ^ (1 << target)] = temp;
      }
   }
}

void phase_oracle(qampl *state, int threshold, int num_qubits) {
   int num_states = 1 << num_qubits;
   for (int i = 0; i < num_states; i++) {
      if (i < threshold) {
         state[i] *= -1;
      }
   }
}

void grover_diffusion(qampl *state, int num_qubits) {
   for (int i = 0; i < num_qubits; i++) {
      hadamard(state, i, num_qubits);
   }
   phase_oracle(state, 0, num_qubits);
   for (int i = 0; i < num_qubits; i++) {
      hadamard(state, i, num_qubits);
   }
}

int measure(qampl *state, int qubit, int num_qubits) {
   double probability_0 = 0.0;
   int num_states = 1 << num_qubits;

   for (int i = 0; i < num_states; i++) {
      if (!((i >> qubit) & 1)) {
probability_0 += (creal(state[i]) * creal(state[i])) + (cimag(state[i]) * cimag(state[i]));
      }
   }

   if ((double)rand() / RAND_MAX < probability_0) {
      return 0;
   } else {
      return 1;
   }
}

int main() {
   int num_qubits = 3;
   int num_states = 1 << num_qubits;
   qampl *state = (qampl *)malloc(num_states * sizeof(qampl));

   for (int i = 0; i < num_states; i++) {
      state[i] = (i == 0) ? 1.0 : 0.0;
   }

   for (int i = 0; i < num_qubits; i++) {
      hadamard(state, i, num_qubits);
   }

   int array[] = {5, 2, 8, 1, 7, 3, 6, 4};
   int min_value = INT_MAX;
   int min_index = -1;
```

```
    for (int threshold = num_states - 1; threshold >= 0; threshold--) {
        phase_oracle(state, threshold, num_qubits);
        grover_diffusion(state, num_qubits);

        int measured_index = 0;
        for (int i = 0; i < num_qubits; i++) {
            measured_index |= (measure(state, i, num_qubits) << i);
        }

        if (array[measured_index] < min_value) {
            min_value = array[measured_index];
            min_index = measured_index;
        }
    }

    printf("Index of Minimum Value: %d\n", min_index);
    printf("Minimum Value: %d\n", min_value);

    free(state);
    return 0;
}
```