Introduction to Quantum Computing
Project Report

Aakarsh Jain
Roll Number - 2021507
IIIT - Delhi
aakarsh21507@iiitd.ac.in

Raghav Sakhuja
Roll Number - 2021274
IIIT - Delhi
raghav21274@iiitd.ac.in

Siddhant Rai Viksit
Roll Number - 2021565
IIIT - Delhi
siddhant21565@iiitd.ac.in

3 April 2024

1 Introduction

Elemental Distinctness is the following problem:

Element Distinctness: Given numbers $x_1, ..., x_N \in [N]$, are they all distinct?

We define a function $f : [1, N] \longrightarrow [a, b]$, we want to find i, j s.t $f(i) = f(j)$ and $i \neq j$. A classical algorithm for this problem would be $O(n)$, however, with quantum search we can do better than this. In this report, we initially present an approach using Grover's search that runs in $O(n^{3/4})$. Further we try to implement Quantum walk and analyse the algorithm provided by [2]. The paper provides an elegant solution of Quantum walk search on Johnson Graphs to find i and j.

2 Grover Search for Elemental Distinctness

2.1 Algorithm

Classically, this problem requires $\Omega(N)$ queries. In quantum case, there are two algorithms. The algorithm uses Grover search in a clever two-level construction and solves the problem $O(N^{3/4})$ queries. Consider the following algorithm:

1. Choose $\sqrt{N}$ random numbers $i1, ..., i_{\sqrt{N}} \in 1, 2, ..., N$. Evaluate f $(i_1), ..., f(i_{\sqrt{N}})$. If two of them are equal, stop, output the two equal elements.
2. Use Grover's search to search (among remaining $N - \sqrt{N}$indices $k \in 1, 2, ..., N$) for an index k such that $f(k) = f(i_j)$ for some j.

The algorithm involves two main steps: first, it randomly selects $\sqrt{N}$ numbers and evaluates their corresponding function values. If any two values are equal, the algorithm stops and outputs those elements. Then, it employs Grover's search to find an index among the remaining set where the function values match one of the previously obtained values. Following this, amplitude amplification is applied to increase the success probability to a constant with $O(1/\sqrt{\epsilon})$ repetitions of the entire algorithm. Since $\epsilon = const/\sqrt{N}$, $O(N^{1/4})$ repetitions are sufficient. This yields a total number of queries needed being $O(N^{3/4})$.

2.1.1 Oracle for Grover Search

For a given f, we define and implement a unitary U_{copy} which gives us query access to the function.

$$U_{copy} |x\rangle |0\rangle = |x\rangle |f(x)\rangle$$

Using this Oracle, we are able to access the values mapped to an index. Further, we use another Oracle U_{comp} to compare the retrieved value to the values that are already known to us from the first step of the algorithm . We define U_{comp} as follows:

$$U_{comp} |x\rangle |f(x)\rangle = - |x\rangle |f(x)\rangle \forall x \in S$$

$$U_{comp} |x\rangle |f(x)\rangle = |x\rangle |f(x)\rangle \forall x \notin S$$

where S is the set of elements obtained from the first step of the algorithm.

To construct our oracle, we use a combination of two oracles. We show the equation on how the Grover oracle works.

$$U_{copy}{}^{-1} U_{comp} U_{copy} |x\rangle |0\rangle \to |x\rangle |f(x)\rangle U_{comp} |x\rangle |f(x)\rangle \to U_{copy}{}^{-1} - |x\rangle |f(x)\rangle \to - |x\rangle |0\rangle \tag{1}$$

Thus, we use $U_{copy}{}^{-1} U_{comp} U_{copy}$ as Oracle for our grover iterator.

2.2 Analysis

Since our algorithm's time complexity is $O(n^{3/4})$, we would expect, on average, for around $n^{3/4}$ queries to be made to the oracle. Our algorithm succeeds with probability $\epsilon = \frac{const}{\sqrt{n}}$[1], and we can get it to a constant probability within $O(\frac{1}{\sqrt{\epsilon}})$, which, since Grover's search is $O(\sqrt{n})$, means a total time complexity of $O(n^{1/2} n^{1/4}) = O(n^{3/4})$

While implementing the problem, we tried to implement the oracle in two methods. Though theoretically they should be exactly same, the results achieved through them were not identical. The better of the two implementations was the one where we first classically solve the question, and then use it's values to construct the oracle, by comparing the indices. As this method is not how we are supposed to solve a question, we also implemented the oracle using the above mentioned U_{comp} and U_{copy}. But surprisingly, the results obtained by this oracle were not as good.

2.3 Result

The python implementation for the above can be found here. We ran 1000 tests on array of size $N = 100$, tracking the number of calls made to the oracle.

To run the experiment locally, clone the above repository, and run the following commands -

```
python3 -m venv ./.venv
source ./.venv/bin/activate
pip install -r requirements.txt
python3 elemental_distinctiveness.py
```

The number of test iterations and size of the array can be modified inside the code if required.

When we set the array to only contain elements $1 \ldots N$(i.e.,$f : [1, N] \to [1, N]$).

Simulated Oracle: Fraction of incorrect classifications: 0.22 Average number of calls made to the oracle: 51.95

Solved Oracle: Fraction of incorrect classifications: 0.03 Average number of calls made to the oracle: 50.01

The number of calls for both are around 50, which is around the same bound that the authors propose of $O(N^{\frac{3}{4}})$ (upto a constant) .

3 Quantum Walk for elemental Distinctness

3.1 Quantum Walk

In classical random walks, the algorithm progresses from one vertex to another in a graph based on certain probabilities, with each step representing a transition between vertices. However, in the quantum version, the algorithm operates on superposition states, allowing it to explore multiple paths simultaneously.

The quantum walk algorithm defines basis states that include both the current and previous vertices, akin to representing edges of the graph. This expanded basis state enables quantum superposition, where the algorithm explores different paths concurrently.

Quantum walk algorithm for search is quite analogous to Grover's algorithm. The basis state $|x\rangle\,|y\rangle$ is "good" if x is a marked vertex, and "bad" otherwise. Define $|px\rangle = \Sigma_y \sqrt{Pxy}\,|y\rangle$ to be the uniform superposition over the neighbors of x. As for Grover, define "good" and "bad" states as the superpositions over good and bad basis states:

$$|G\rangle = \frac{1}{\sqrt{|M|}} \sum_{x \in M} |x\rangle|p_x\rangle$$

$$|B\rangle = \frac{1}{\sqrt{N - |M|}} \sum_{x \notin M} |x\rangle|p_x\rangle$$

where M denotes the set of marked vertices. Note that $|G\rangle$ is just the uniform superposition over all edges (x, y) where the first coordinate is marked, and $|B\rangle$ is just the uniform superposition over all edges (x, y) where the first coordinate is not marked.

If $\epsilon = \frac{|M|}{N}$ and $\theta := \arcsin(\sqrt{\epsilon})$, then the uniform state over all edges can be written as:

$$|U\rangle = \frac{1}{\sqrt{N}} \sum_{x} |x\rangle|p_x\rangle = \sin(\theta)|G\rangle + \cos(\theta)|B\rangle$$

Further, we do amplitude amplification similarly to grover, first reflecting through $|B\rangle$ and then through $|U\rangle$.

3.2 Algorithm

To approach this problem, we construct a graph based on the sets of indices $R \subseteq 0, \ldots, n_1$ of size r, denoted as $J(n, r)$, known as the Johnson graph. The vertices of this graph represent these sets R, and there's an edge between two vertices if the corresponding sets differ in exactly two elements, indicating that one element was replaced by another. The graph is $r(n-r)$ - regular, meaning each vertex has $r(n-r)$ neighbors. Additionally, we keep track of the sequence of values $x_R = (x_i)_{i \in R}$for each set R.

A vertex in $J(n, r)$ is marked if it contains a collision, meaning there exist distinct indices i and j in the corresponding set R such that $x_i = x_j$

The probability that i and j are both in a random r-set R is $\epsilon = \frac{r(r-1)}{n(n-1)}$. Therefore, the fraction of marked vertices is at least ϵ, which is approximately $(r/n)^2$.

3.3 Implementation

For Implementing a quantum walk, we have used an simple example of a 4-d hypercube, where 1011 and 1111 is the marked vertices. We were unable to create a generalized oracle for solving elemental distinctness using Johnson graph. The implantation for the dummy quantum walk is done using a hard-coded oracle.

For the implementation, we have directly used the library implementation of QFT available in qiskit.circuit.library. Further, we referenced a standard implementation of the same avialable here.

3.4 Results

The python implementation for the above can be found here. The code simulates Quantum Random Walk search on a 4D hypercube, and returns the results in dictionary format.

To run the experiment locally, clone the above repository, and run the following commands -

```
python3 -m venv ./.venv
source ./.venv/bin/activate
pip install -r requirements.txt
python3 quantumwalk_hypercube.py
```

The results are consistent with theoretical expectations, as we can see in the following histogram -

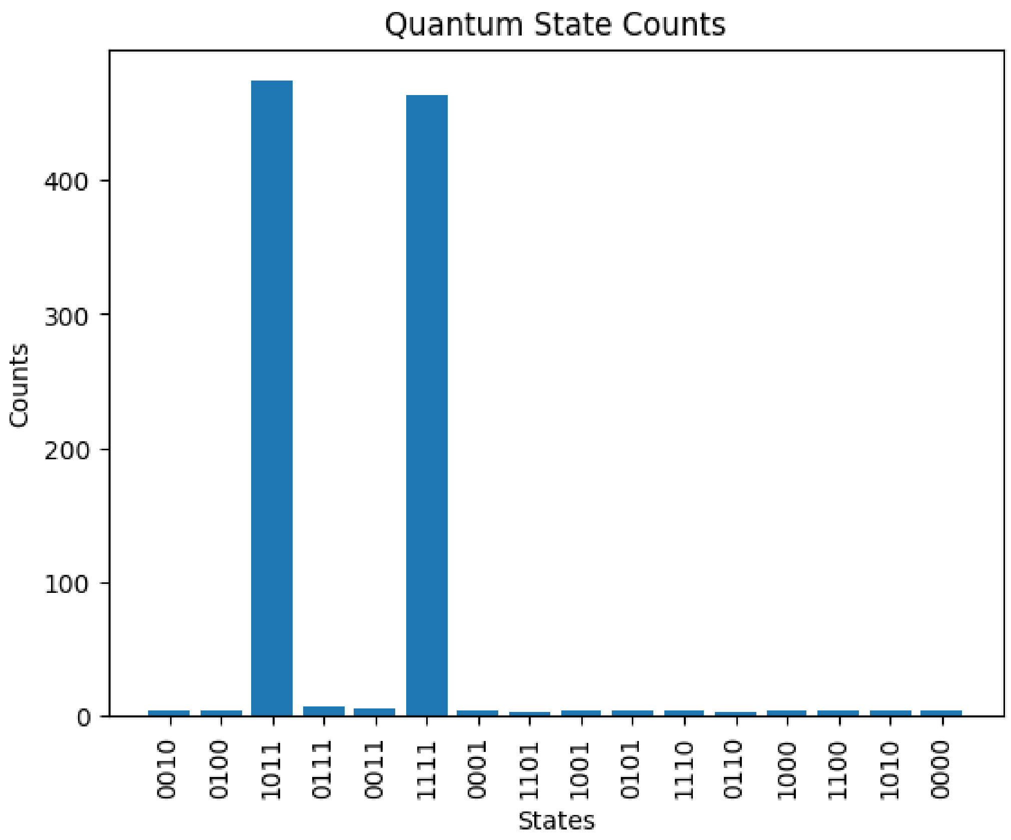

Figure 1: Histogram of results after simulation

References

[1] Andris Ambainis. *Quantum search algorithms*. 2005. arXiv: quant-ph/0504012 [quant-ph].

[2] Andris Ambainis. "Quantum walk algorithm for element distinctness". In: *SIAM Journal on Computing* 37.1 (2007), pp. 210–239.