OPTIMIZING ORACLE FUNCTION IN GROVER'S ALGORITHM

Hridyesh, Mridul, Sumit

1 Abstract

Recent advancements in quantum computing hardware fuel the development of practical applications for established algorithms like Grover's search. This algorithm excels at searching unstructured databases using quantum oracles. However, manually creating these oracles is challenging. This paper addresses this issue by proposing a novel framework for automatically generating oracles from unstructured data. Our framework leverages truth tables and efficient quantum logic synthesis techniques. We use a "phase-tolerant synthesis" method that significantly reduces circuit complexity compared to existing approaches. Additionally, we present a method for scalable logic synthesis suitable for real-world hardware constraints. We evaluate our methods through benchmarks and demonstrate their effectiveness in automatic oracle generation for Grover's algorithm

2 Introduction

The field of quantum computing has witnessed a surge in interest in recent years. Grover's algorithm, a prominent example of a quantum algorithm with potential real-world applications, is particularly noteworthy. This algorithm offers efficient search capabilities for unstructured databases. However, a key challenge lies in the creation of quantum oracles, essential components of Grover's algorithm. These oracles, often masked as black boxes, typically require manual design specific to the search query and database. This impedes the broader integration of Grover's algorithm into software development tasks.

Our research addresses this challenge by introducing a fully automated framework for generating oracles for Grover's search. This framework eliminates the need for manual design and simplifies the application of Grover's algorithm. Our approach involves:

Data Encoding: Converting arbitrary data into Boolean functions represented by truth tables.

Oracle Synthesis: Generating Grover oracles from the truth table expressions.

Integration: Seamless integration of the generated oracles into the initialization and diffusion steps of Grover's algorithm.

This framework empowers developers to leverage Grover's algorithm for various tasks by providing user-friendly input procedures.

3 Problem Statement

Here's a breakdown of the key challenges addressed in this research:

Creating oracles for Grover's search is a complex and time-consuming process.

Current approaches require case-by-case design based on the specific database and search target.

The lack of user-friendly interfaces hinders the application of Grover's algorithm in practical programming tasks.

Our research offers a solution by enabling efficient and convenient input mechanisms for Grover's algorithm.

Our research aims at optimizing the oracle function in Grover's algorithm which can help in reducing the complexity of the Grover's algorithm.

4 Overview and background:-

This section provides a high-level overview of Grover's algorithm, a powerful tool for searching unstructured databases. We focus on its core functionalities without delving into the mathematical details. We then explore the concept of reversible quantum logic synthesis, which plays a crucial role in our approach to automated oracle generation for Grover's algorithm.

Grovers's Algorithm

At the heart of Grover's algorithm lies a quantum oracle, tasked with identifying specific "winner" items within the database. The specifics of oracle generation are addressed in later sections. Here, we assume the oracle possesses the following functionality:

$$O|i\rangle = (-1)^{f(i)}|i\rangle \text{ with } f(i) = \begin{cases} 0, & \text{if } i \notin \{w_j\} \\ 1, & \text{if } i \in \{w_j\}. \end{cases}$$

Note that for the oracle, we do not need to explicitly know w_j but we only need a valid function f for the search problem. Also for an explicit construction of such oracles, one generally needs an auxiliary qubit register to store intermediate results into which has to be uncomputed later on[1]. The role of this auxiliary qubit register and belonging operations is further discussed in section 3.

In order to evaluate Grover's algorithm with this oracle, we initialize the system in the fiducial state $|\Psi\rangle = |0\rangle$. Grover's algorithm starts by setting all qubits into an equal superposition state $|s\rangle$

$$H^{\otimes n}|0\rangle^n = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle = |s\rangle.$$

Here, the integer states $|i\rangle$ directly relate to corresponding binary encoded states in the computational basis.

After phase-tagging the winner states—i.e. the states representing the values we search for in the unstructured database—Grover's algorithm implements a so-called diffusion operator $U_d = 2|s\rangle\langle s|$ - I that amplifies the amplitudes for measuring the winner states. The phase-tagging and diffusion steps can geometrically be considered as two successively performed reflections, and thus as a single rotation in a

2D-plane. Each such rotation corresponds to a single Grover iteration that gradually rotates $|s\rangle$ closer to w_j . At the end of the algorithm, a measurement in the computational basis is performed and the searched and potentially found items can be identified by distinct peaks in the distribution of the measured results. It is important to remark that in literature there is a derived optimal number of Grover iterations

(i.e. amplification and oracle application) that results in $|s\rangle$ being rotated the closest to w_j . This optimal number of iterations yields the best measurement results. Thus, performing more or less rotations is expected to lead to increasingly worse search results. A more in-depth discussion about Grover iterations will be given in section 3. To summarize briefly: searching M items within an unstructured database needs at

most O(^{N}) iterations by Grover's algorithm. Hence there is a noticeable advantage over classical search algorithms, which are known to perform in O(N) for a linear search.

4.1 Implimemtation of grover's algorithm

Implementing Grover's algorithm typically involves using a quantum programming framework or language such as Qiskit for IBM Quantum computers, Cirq for Google's Quantum Computing Framework, or Quipper for a functional programming approach. Here is provide a basic implementation of Grover's algorithm using Qiskit, a popular quantum computing SDK for Python.

```
[3]: my_list = [3,2,4,5,1,7,8,9,6,10,11,12,13,45,65,0, 99,34,77,54,24,98,14,53,76,56,87]
```

```
[4]: def the_oracle(input):
         winner = 3
         if input is winner :
             response = True
         else :
             response = False
         return response
```

```
[5]: for index, trial_number in enumerate(my_list):
         if the_oracle(trial_number) is True:
             print ('Winner found at index %i'%index)
             print('%i calls to the Oracle used'%(index+1))
             break
```

```
Winner found at index 0
1 calls to the Oracle used
```

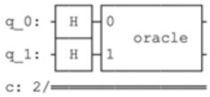
```
[6]: from qiskit import *
     from qiskit_aer import Aer
     from qiskit import QuantumCircuit
     import matplotlib.pyplot as plt
     import numpy as np
```

```
[7]: oracle = QuantumCircuit (2, name=' oracle' )
     oracle.cz (0,1)
     oracle. to_gate()
     oracle.draw()
```

```
[7]: q_0: ─■─
           │
     q_1: ─■─
```

```
[8]: backend1 = Aer.get_backend('statevector_simulator')
     grover_circ = QuantumCircuit(2,2)
     grover_circ.h( [0,1])
     grover_circ.append (oracle, [0,1])
     grover_circ.draw()
```

[8]:

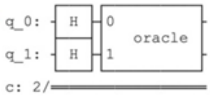



```
[12]: job = transpile(grover_circ, backend1)
      result = backend1.run(job)
```

```
[13]: grover_circ.remove_final_measurements()
      from qiskit.quantum_info import Statevector
      statevector = Statevector(grover_circ)
      print(statevector)
      np.around(statevector, 2)
      grover_circ.draw()
```

```
Statevector([ 0.5+0.j,  0.5+0.j,  0.5+0.j, -0.5+0.j],
            dims=(2, 2))
```

[13]:

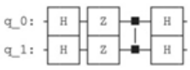



```
[14]: reflection = QuantumCircuit(2, name='reflection')
      reflection.h ([0,1])
      reflection.z([0,1])
      reflection.cz (0,1)
      reflection.h([0,1])
      reflection.to_gate()
```

```
[14]: Instruction(name='reflection', num_qubits=2, num_clbits=0, params=[])
```

```
[15]: reflection.draw()
```

[15]:

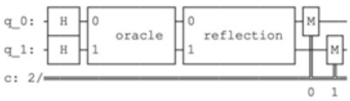



```
[16]: backend2 = Aer.get_backend('qasm_simulator')
      grover_circ = QuantumCircuit (2,2)
      grover_circ.h([0,1])
      grover_circ.append(oracle,[0,1])
      grover_circ.append(reflection,[0,1])
      grover_circ.measure([0,1],[0,1])
```

```
[16]: <qiskit.circuit.instructionset.InstructionSet at 0x12e2edfc0>
```

```
[17]: grover_circ.draw()
```

[17]:



Reversible quantum logic synthesis

1.Generating quantum oracles is intricately linked to synthesizing reversible quantum circuits from classical logic expressions.

2.The field of reversible quantum logic synthesis has seen significant advancements over the past two decades, with researchers continuously developing more efficient methods in terms of gate count and ancilla qubit usage. While established libraries like Qiskit offer oracle generation functionalities, our research focuses on a novel approach of optimizing the Oracle Function.

5 Method for automatic oracle generation:-

This section presents a method for automatically transforming arbitrary databases into functional oracles for Grover's algorithm. Approach hinges on a computationally efficient labeling function that assigns unique bitstrings of a specific length (`k`) to each database entry. This length `k` is carefully chosen based on the problem specifics

$$l : D \to \mathrm{F}^k \; e \to l(e).$$

Here, k belong to N stands for the label size, which has to be chosen according to the specifics of the problem

The selection of this labeling function is crucial for efficiency. Unlike existing methods that might require global database information, our function relies solely on local information of each entry, ensuring independence between bitstrings. In Python implementation, we leverage the native hash function to generate unique integer hash values for each entry, which are then converted to binary representations and truncated to the desired k length.

These labels essentially translate the database into a sequence of bitstrings, forming a logical truth table . This truth table then serves as the foundation for constructing a quantum circuit using a suitable logic synthesis algorithm. The resulting circuit, denoted as UD, acts as a unitary mapping:

$$\mathrm{UD}|\mathrm{i}\rangle|0\rangle = |\mathrm{i}\rangle|\mathrm{l(e(i))}\rangle$$

where `l(e(i))` represents the label of the database entry `e` with index `i`. Notably, the synthesis process only needs to be performed once per database. The generated circuit can be reused for subsequent searches within the same database and only requires updates if the database content itself changes. Efficiently updating database circuits remains an open research question that we plan to address in the future.

To query the database for a specific element `eq` with index `iq`, we combine the synthesized truth table circuit with a phase-tagging operation specific to the bitstring `l(eq)`. This phase-tagging, achieved using a multi-controlled Z gate with strategically placed X gates, can be calculated efficiently from `eq` itself. Finally, the label variable needs to be "uncomputed" again.

which is precisely the functionality we required $O(e_q)$

5.1 Hash collisions:-

As two objects can have the same hashing values.This doesn't pose a significant problem because after applying Grover's algorithm, a classical search can be conducted on the considerably reduced search space. Another important aspect is that the probability of a hash collision is effectively controlled by adjusting the hash value size (i.e., the length of the binary string).

A potentially more intricate challenge emerges when determining the optimal number of Grover iterations (R). this value depends on the quantity of elements sharing the same hash string (denoted by M) and the total number of elements within the database (denoted by N).

$$R \leqslant \left\lceil \frac{\pi}{4}\sqrt{\frac{N}{M}} \right\rceil$$

At first glance, this might seem like a minor inconvenience, potentially requiring only a few additional iterations. However, as discussed earlier, exceeding the optimal number of iterations can negatively impact the search results.

While a quantum algorithm exists to determine M precisely , it's impractical for real-world applications due to the exponential number of oracle calls it necessitates. Additionally, all these oracle calls require careful control, which translates to using Toffoli gates (essentially controlled-CNOT gates) instead of simpler CNOT gates, inflating the overall gate count by a factor of 6. These drawbacks render this approach infeasible in practice.

Therefore, we employ a heuristic approach to estimate M. This method involves calculating the expected value of another element (denoted by ei) colliding with the specific bitstring of a chosen element (denoted by e0). To achieve this, we assume a uniform distribution of bitstrings across the entire label space (Fk). This scenario sets the stage for a Bernoulli process with N-1 trials and a probability of success (p) equal to 2^{-k} (where k represents the hash value size).

The resulting model aligns with the binomial distribution, leading to a straightforward formula for the expected value :

$$\text{E}(\#\text{collisions}) = np = (N - 1)2^{-k}.$$

Since there is at least one element sharing the bitstring of e_0 (i.e. e_0 itself) we add a 1 to acquire the expected value for M:

$$\text{E}(M) = 1 + (N - 1)2^{-k}.$$

This formula captures the intuitive relationship between the hash value size (k) and the estimated number of hash collisions (M). It demonstrates that reducing the hash value size (k) increases the estimated M, while simultaneously reducing the number of Grover iterations required (as seen in Equation 6) and the number of gate operations needed. This becomes particularly interesting in future scenarios where quantum resources are readily available and cost-competitive with classical resources, but not yet capable of tackling complex problems independently. In such a hybrid setting, the quantum computer would first significantly reduce the search space by a factor of approximately 2^{-k}, followed by a classical search to pinpoint the exact element.

Algorithmic view on oracle generation :-

This passage discusses the algorithmic approach to generating oracles for Grover's search algorithm from a random database. It outlines two sub-algorithms and analyzes their time complexity. **Overall Procedure**

The automated oracle generation process is broken down into two main algorithms:

- **Algorithm 1: Database Encoding** - This algorithm details the steps needed to encode an arbitrary database for Grover's search. It requires the ability to iterate over the database content, regardless of order, allowing for databases containing various object types.
-
  - **Input:** Database and a labeling function (e.g., hash function) that assigns non-unique labels to each database object.
  - **Steps:**
    1. Iterate through each database object using a for-loop (complexity: O(N), N being the number of entries).
    2. Assign a label (binary string) to each object using the labeling function.
    3. Store these labels in a list for truth table preparation (complexity: O(N)).
    4. Perform quantum logic synthesis to translate the truth table into a quantum circuit (most critical step, complexity depends on chosen method).
- **Algorithm 2: Automatic Oracle Definition** - This algorithm defines the oracle based on the encoded database from Algorithm 1. It prepares a specific quantum oracle for each search value.
  - **Steps:**
    1. Create a QuantumCircuit object.
    2. Embed the quantum circuit encoding the database from Algorithm 1 (complexity dominated by embedding, related to the number of gates in the database circuit).
    3. Apply a phase tag to mark the specific value/object to search for.

4. Apply a diffuser operator.

Time Complexity Analysis

Algorithm 1: The overall time complexity of Algorithm 1 is O(N). The loop iterating through the database objects and label creation contribute O(N) complexity. While quantum logic synthesis complexity varies based on the chosen method, it's typically executed only once per database. **Algorithm 2:**The time complexity of Algorithm 2 is O(N * log(N)). Embedding the encoded database circuit, which has a complexity of O(N * log(N)) per truth table column, dominates the overall complexity

Open Research Issues

The presented approach, while not currently surpassing classical database search efficiency, opens doors for further research. Exploring new methods for quantum logic synthesis and establishing more flexible Grover oracles could ultimately lead to significant improvements in database search efficiency.

Similarity search

The method presented so far can be generalized to a technique, which also allows for searching bitstrings *similar* to the query, i.e. not the exact item but rather one that is very close according to some metric. In the case, where the oracle has been generated from labeled data, this can obviously only work if the labeling function preserves the similarities between the database elements. Another application scenario for a *similarity search* could be the case where the labels themselves constitute the data.

The general idea of encoding the similarity of two bitstrings into the quantum oracle is to replace the tagging function $T^{\text{sim}}(l(e_q))$ with a circuit that performs phase shifts based on the Hamming-Distance. One such circuit is given by the application of RZ gates on the label register. In more detail:

$$T^{\text{sim}}(l(e_q)) = \bigotimes_{j=0}^{k-1} \text{RZ}_i\left(\frac{-(-1)^{l(e_q)_j}2\pi}{k}\right).$$

For a single RZ gate acting on a qubit in a computational basis state we have:

$$RZ(-(-1)^x\phi)|y\rangle = \exp\left(\frac{i\phi}{2}(-1)^{x\oplus y}\right)|y\rangle.$$

Applying the similarity tag $T^{\text{sim}}(l(e_q))$ to the multi-qubit state $|l(e)\rangle$ therefore yields:

$$T^{\text{sim}}(l(e_q))|l(e)\rangle = \exp\left(\frac{i\pi}{k}\sum_{j=0}^{k-1}(-1)^{l_j(e_q)\oplus l_j(e)}\right)|l(e)\rangle.$$

To get an intuition of the effect of this, we now consider what happens when $l(e)= l(e_q)$. In this case we have:

$$l_i(e_q) \ \forall \ l_i(e)= 0 \ \forall \ i < k,$$

implying that the sum over j evaluates to k. Therefore the applied phase is simply π, i.e. exactly what we have in the case of a regular phase-tag. If $l(e)= l(e_q)$ for all but one j-index, the sum evaluates to k - 2. Hence, the applied phase is π(1 - 2) which is 'almost' the full phase-tag. An example application of the similarity search can be found in figure.

In the following, a more formal proof is presented regarding why the above considerations work when applied within Grover's algorithm. For this we assume that the oracle has tagged the uniform superposition

$$|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{2^n-1} \exp(i\phi(x))|x\rangle.$$

We now apply the diffusion operator $U_s = 2|s\rangle\langle s| - \mathbb{I}$ on the above state:

$$U_s|\psi\rangle = \frac{1}{\sqrt{N}}(2|s\rangle\langle s| - \mathbb{I}) \sum_{x=0}^{2^n-1} \exp(i\phi(x))|x\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{2^n-1} \exp(i\phi(x))(2|s\rangle\langle s | x\rangle - |x\rangle).$$

Using $\langle s | x\rangle = \frac{1}{\sqrt{N}}$ gives:

$$= \frac{1}{\sqrt{N}} \left(\frac{2}{\sqrt{N}} \left(\sum_{x=0}^{2^n-1} \exp(i\phi(x)) \right) |s\rangle - \sum_{x=0}^{2^n-1} \exp(i\phi(x))|x\rangle \right).$$

Next, we set

$$r_{\text{cm}} \exp(i\phi_{\text{cm}}) := \frac{1}{N} \sum_{x=0}^{2^n-1} \exp(i\phi(x)),$$

where cm stands for the center of mass. Inserting the definition of $|s\rangle$ we get:

$$= \frac{1}{\sqrt{N}} \sum_{x=0}^{2^n-1} (2r_{\text{cm}} \exp(i\phi_{\text{cm}}) - \exp(i\phi(x)))|x\rangle$$

$$= \frac{\exp(i\phi_{\text{cm}})}{\sqrt{N}} \sum_{x=0}^{2^n-1} (2r_{\text{cm}} + \exp(i(\phi(x) - \phi_{\text{cm}} + \pi)))|x\rangle.$$

Finally, we use the rules of polar coordinate addition:

$$r_3 \exp(i\phi_3) = r_1 \exp(i\phi_1) + r_2 \exp(i\phi_2),$$

$$r_3 = \sqrt{r_1^2 + r_2^2 + 2r_1 r_2 \cos(\phi_1 - \phi_2)}$$

$$\phi_3 = \arctan\left(\frac{r_1 \sin(\phi_1) + r_2 \sin(\phi_2)}{r_1 \cos(\phi_1) + r_2 \cos(\phi_2)} \right),$$

to determine the absolute values of the coefficients in order to find out about the amplification factor A_x:

$$A_x = |2r_{\text{cm}} + \exp(i(\phi(x) - \phi_{\text{cm}} + \pi))| = \sqrt{1 + 4r_{\text{cm}}^2 - 4r_{\text{cm}} \cos(\phi(x) - \phi_{\text{cm}})}.$$

From this we see that A_x becomes maximal if $\phi(x)$ - $\phi_{\text{cm}} = \pm\pi$, minimal if $\phi(x)$ - $\phi_{\text{cm}} = 0$ and is monotonically developing in between, which is precisely the behavior we expected.

Even though the similarity tag allows a considerable cut in the CNOT count compared to the multi-controlled Z gate, it comes with some drawbacks. The biggest problem is that the results of this

method are very sensitive to the number of iterations. Applying the wrong amount of Grover iterations can lead to the case, where labels which are less similar get a higher measurement probability. This might be

improved through further analysis of the similarity phase-tags. Another drawback is that labels, where the bitwise NOT is similar to the query, get the same amplification as their inverted counter-part (assuming

$\phi_{\text{cm}} = 0$ in equation —more to this assumption soon). For example if we query for the label 000000, the

label 011111 would receive the same amplification as 100000. This can be explained by looking at equation (12). If $l(e_q)$ and $l(e)$ differ on every single entry, we have:

$$l(e_q)_i \oplus l(e)_i = 1 \; \forall i < k.$$

This results in the sum evaluating to -k and yielding a phase of -π, which is equivalent to a phase of π. A similar effect can be observed, when only a single bit of the inverse is mismatching. In this case, we apply the phase -π + ^{1} . Even though this phase is not equivalent to π - 1 , its absolute value is, which is the feature of relevance according to equation .

5.2 Advanced similarity tags:-

In the realm of quantum search algorithms, a typical limitation exists with similarity tags being restricted to Hamming distance. This current approach hinders the potential for capturing a wider range of similarity types. This section presents a method that overcomes this limitation by introducing a novel concept of advanced similarity tags. These tags are designed to function with a much more extensive class of similarity measures.

$$f : Q \times \mathrm{F}^k \to [0, 1]$$

Understanding Similarity Measures:

- Similarity measures are essentially functions that accept two inputs: a query object and a bitstring.
- The function then outputs a similarity score ranging from 0 (indicating no similarity) to 1 (representing identical objects).

The Power of Gray Synthesis:

- The proposed method leverages a technique known as Gray synthesis to construct these advanced tags.
- Gray synthesis empowers the creation of specialized circuits capable of manipulating quantum states in a predetermined manner.
- By incorporating Gray synthesis, the tags can interact with both the query object and a database within the search algorithm.

Benefits and Implications:

- These advanced tags significantly enhance the search algorithm's capabilities.
- Compared to the limitations of Hamming distance, they enable the algorithm to identify items based on more intricate similarity criteria.
- Notably, the computational cost associated with these advanced tags remains comparable to existing methods.

Mathematical Representation:

The document delves into mathematical formulas to illustrate the construction and interaction of these tags within the search process. These formulas encompass concepts like unitary matrices, computational basis states, and phase factors. Here's a breakdown of the key formulas:

Similarity Measure Function (f): This formula defines the function used to calculate similarity between a query object (q) and a bitstring (y) of length k.

$$\mathrm{f{:}Q{\times}F\hat{k}} \to [0,\, 1]$$

- Q represents the set of possible query objects.
- $\mathrm{F\hat{k}}$ denotes the set of all bitstrings with length k.

[0, 1] represents the range of the function, with 0 indicating no similarity and 1 signifying identical objects

Dice Coefficient Example (f_Dice): This formula showcases a specific example of a similarity measure function, the Dice coefficient.

$$\mathrm{F\_Dice{:}F\hat{k}} \times \mathrm{F\hat{k}} \to [0,\, 1]$$

This formula calculates the Dice coefficient between two bitstrings x and y by considering the ratio of the number of matching bits to the total number of bits that could potentially match

Gray Synthesis Transformation (Ugray): This formula represents the unitary matrix generated by Gray synthesis for a given tuple of real numbers (ϕ).
Ugray(ϕ) $|\mathrm{y}\rangle$ = exp(iϕ_y) $|\mathrm{y}\rangle$

- ϕ = (ϕ_0, ϕ_1, ..., ϕ_(2$\hat{\mathrm{k}}$ - 1)) is a tuple of real numbers.
- $|\mathrm{y}\rangle$ represents a computational basis state.
- The application of Ugray(ϕ) on a state $|\mathrm{y}\rangle$ introduces a phase factor of exp(iϕ_y).

Similarity Tag (Tsim): This formula defines the similarity tag for a query object (q) based on the chosen similarity measure function (f).
Tsim(q) = Ugray(ϕ_sim(q))
ϕ_sim(q) is a function that translates the query object (q) into a tuple of real numbers used by Ugray(ϕ_sim(q)) to create the phase factors.

Similarity Oracle (Osim): This formula depicts the effect of the similarity oracle on the superposition of database entries.
Osim(f,q,D)$|\mathrm{s}\rangle|0\rangle$=Σ_xU†_D Tsim(q)U_D$|\mathrm{x}\rangle|\mathrm{y(x)}\rangle$ = Σ_x exp(i(-1)y(x)πf(q, y(x))) $|\mathrm{x}\rangle$ $|0\rangle$

- D represents the database encoding circuit.
- $|\mathrm{s}\rangle$ and $|0\rangle$ are the initial states of the search register and the result register, respectively.
- The summation iterates over all possible database entries ($|\mathrm{x}\rangle$).
- y(x) denotes the bitstring associated with database entry $|\mathrm{x}\rangle$.
- The oracle applies phase factors based on the similarity between the query (q) and each database entry (y(x)) using the chose

$$\begin{aligned}
O^{\mathrm{sim}}(f,q,D)|s\rangle|0\rangle &= \sum_{x=0}^{2^n-1} U_D^{\dagger} T_f^{\mathrm{sim}}(q) U_D |x\rangle|0\rangle \\
&= \sum_{x=0}^{2^n-1} U^{\dagger} T_f^{\mathrm{sim}}(q)|x\rangle|y(x)\rangle \\
&= \sum_{x=0}^{2^n-1} U^{\dagger} \exp(i(-1)^{y(x)}\pi f(q,y(x)))\ |x\rangle|y(x)\rangle \\
&= \sum_{x=0}^{2^n-1} \exp(i(-1)^{y(x)}\pi f(q,y(x)))|x\rangle|0\rangle.
\end{aligned}$$

5.3 Contrast functions:-

Since we are only interested in the ordering of the values of the similarity measure, we can improve some properties without changing information by applying a monotonically increasing function with signature

$\wedge : [0,\ 1] \rightarrow [0,\ 1]$.

We call this a *contrast function.* In other words: For a given similarity measure f and a contrast function $\wedge$, instead of f we use $\sim f = \wedge \circ f$ as similarity measure (the $\circ$ denotes the composition). An example of a contrast function which improved the results in many cases can be found . Note that a large

portion of the domain gets mapped to a value close to 0. This not only ensures the assumption

$\sim f(q,\ y(x)) < 0.5$ for most $x < 2^n$ (required for $\phi_{\mathrm{cm}} \approx 0$) but also yields $r_{\mathrm{cm}} \approx 1$ as in most cases $\phi(x) \approx 0$

(compare equation). This in turn gives us a good amplification factor A_x for states that are supposed to be amplified but $A_x \approx 0$ for states that have only mediocre similarity.

5.4 Optimization of oracle function:-

Optimizing the oracle function within Grover's algorithm is essential to maximize the efficiency of quantum search operations. Minimizing gate count stands out as a crucial objective in this optimization process. Leveraging quantum XOR operations, such as those implemented through controlled-NOT (CNOT) gates, proves instrumental in reducing gate usage. By framing the problem in terms of XOR operations, unnecessary gates are sidestepped, leading to a streamlined logic that marks target states with minimal computational overhead. Quantum superposition serves as another powerful tool, allowing for simultaneous computation on multiple states and thus contributing to gate count reduction. Additionally, integrating Quantum Fourier Transform (QFT) when applicable can further streamline operations and decrease gate count. Ancilla qubits, if employed, should be optimized to minimize gate usage, potentially through strategies like recycling or reusing them across different iterations. Quantum compilation techniques, including gate synthesis and circuit optimization, offer additional avenues for refining the oracle circuit. Through careful analysis of quantum costs and selection of operations with lower costs, overall resource utilization can be minimized, thereby enhancing the efficiency of Grover's algorithm for quantum search tasks. Optimizing the oracle function in Grover's algorithm with a focus on minimizing gate count is pivotal for enhancing the efficiency of quantum search operations. One effective strategy involves utilizing quantum XOR operations, which leverage controlled-NOT (CNOT) gates to implement XOR operations efficiently. By formulating the problem in terms of XOR operations, unnecessary gates can be avoided, and the logic can be simplified to mark target states with minimal computational overhead. Moreover, the exploitation of quantum superposition enables simultaneous computation on multiple states, contributing to gate count reduction. Additionally, the integration of Quantum Fourier Transform (QFT) where applicable can streamline operations and decrease gate count. Ancilla qubits, if used, should be optimized to minimize gate usage, possibly by recycling or reusing them across different iterations. Quantum compilation techniques, including gate synthesis and circuit optimization, offer further avenues for streamlining the oracle circuit. By meticulously analyzing quantum costs and selecting operations with lower costs, the overall resource utilization can be minimized, thus enhancing the efficiency of Grover's algorithm for quantum search tasks

Implimentation of optimized pracle function:-

To optimize the oracle function within Grover's algorithm, a systematic approach is adopted, starting with a comprehensive analysis of the problem at hand. Understanding the problem's nuances, including any symmetries or structures, provides insights crucial for designing a more efficient oracle. The optimization primarily focuses on minimizing gate count and circuit depth, achieved through various techniques.

Leveraging quantum XOR operations, such as controlled-NOT (CNOT) gates, helps streamline the implementation and reduce gate usage. Furthermore, exploiting quantum parallelism allows for simultaneous operations on multiple qubits, enhancing efficiency. Ancilla qubits, if employed, are optimized to minimize resource overhead, while quantum compilation techniques ensure the circuit's efficiency. Error mitigation strategies are incorporated to enhance reliability in the presence of noise and errors. Continuous benchmarking and iteration refine the oracle function, guided by performance feedback from simulations or real-world experiments. Through these optimization efforts, the oracle function becomes more adept at marking target states, thereby enhancing the overall performance and scalability of Grover's algorithm on quantum computing platforms

Setup

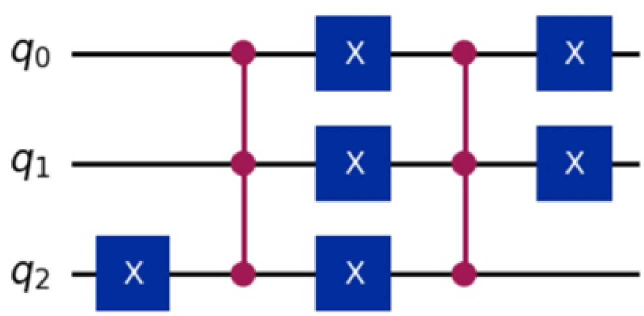
Here we import the small number of tools we need for this tutorial.

[1]:
```
import math
import warnings

warnings.filterwarnings("ignore")
from qiskit import QuantumCircuit
from qiskit.circuit.library import GroverOperator, MCMT, ZGate
from qiskit.visualization import plot_distribution
from qiskit import *
from qiskit_aer import Aer
from qiskit import QuantumCircuit
import matplotlib.pyplot as plt
import numpy as np
```

```
def grover_oracle(marked_states):
    if not isinstance(marked_states, list):
        marked_states = [marked_states]
    num_qubits = len(marked_states[0])

    qc = QuantumCircuit(num_qubits)
    for target in marked_states:
        rev_target = target[::-1]
        zero_inds = [ind for ind in range(num_qubits) if rev_target.startswith("0", ind)]
        qc.x(zero_inds)
        qc.compose(MCMT(ZGate(), num_qubits - 1, 1), inplace=True)
        qc.x(zero_inds)
    return qc
```

[8]:
```
marked_states = ["011", "100"]

oracle = grover_oracle(marked_states)
oracle.draw(output="mpl", style="iqp")
```

[8]:

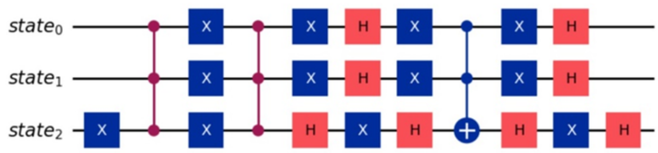


GroverOperator

The built-in Qiskit `GroverOperator` takes an oracle circuit and returns a circuit that is composed of the oracle circuit itself and a circuit that amplifies the states marked by the oracle. Here, we `decompose` the circuit to see the gates within the operator:

[9]:
```
grover_op = GroverOperator(oracle)
grover_op.decompose().draw(output="mpl", style="iqp")
```

[9]: Global Phase: π

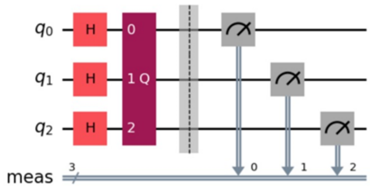


Full Grover circuit

A complete Grover experiment starts with a Hadamard gate on each qubit; creating an even superposition of all computational basis states, followed the Grover operator (`grover_op`) repeated the optimal number of times. Here we make use of the `QuantumCircuit.power(INT)` method to repeatedly apply the Grover operator.

[17]:
```
qc = QuantumCircuit(grover_op.num_qubits)
qc.h(range(grover_op.num_qubits))
qc.compose(grover_op.power(optimal_num_iterations), inplace=True)
qc.measure_all()
qc.draw(output="mpl", style="iqp")
```

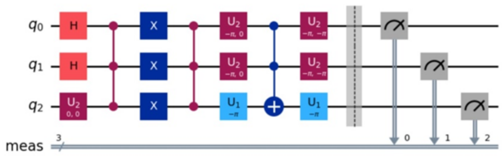
[17]:



Step 2: Optimize problem for quantum execution.

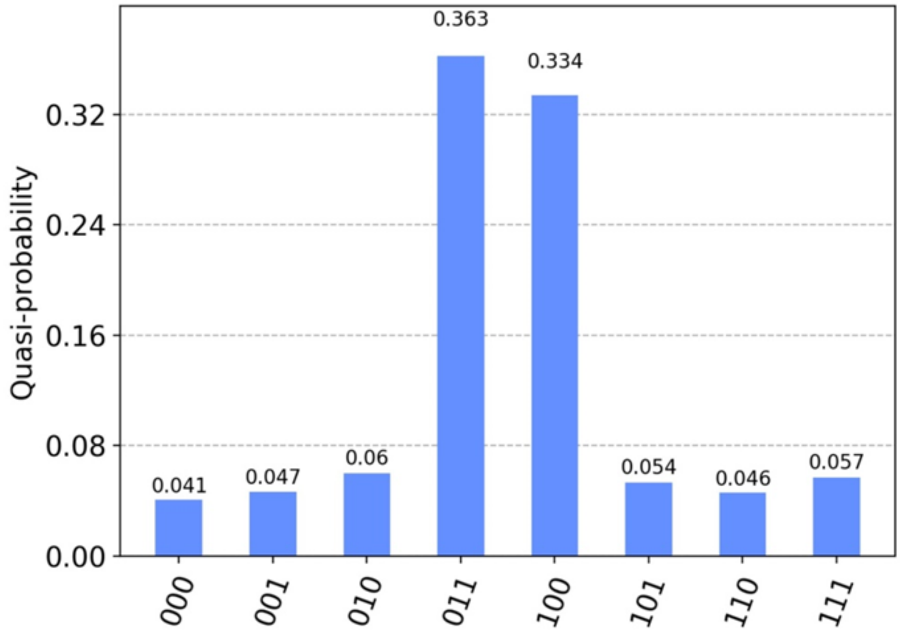
[18]:
```
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager
backend = Aer.get_backend('aer_simulator')
target = backend.target
pm = generate_preset_pass_manager(target=target, optimization_level=3)

circuit_isa = pm.run(qc)
circuit_isa.draw(output="mpl", idle_wires=False, style="iqp")
```

[18]: Global Phase: π



[13]:
```
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager
from qiskit.primitives import Sampler  # Import Sampler
from qiskit.result import postprocess
from qiskit_aer import Aer
backend = Aer.get_backend('aer_simulator')
sampler = Sampler()
sampler.options.default_shots = 10_000
result = sampler.run([circuit_isa]).result()
print(result)
```

```
SamplerResult(quasi_dists=[{3: 0.499999999999999, 4: 0.499999999999999}], metadata=[{}])
```



6 Quantum logic synthesis:-

As pointed out in the previous sections, an integral part of generating oracles is the logic synthesis. There is a multitude of approaches each having their benefits and drawbacks. In this section, we focus on the *Reed–Muller Expansion* and *Gray Synthesis*. Subsequently, a new synthesis method developed by us is introduced, which is significantly more efficient in terms of general gate count. Our approach is to relax the constraint of all outputs having the same phase. This does not interfere with the outcome of Grover's algorithm as these phases cancel out during uncomputation. Finally, we derive and introduce another synthesis method, which addresses the pitfalls and requirements for scalable implementations of the belonging quantum circuits.

6.1 Reed–Muller expansion:-

The Reed–Muller expansion is a fundamental concept in classical logic synthesis, but it also plays a role in understanding quantum logic synthesis. While not the most efficient method for quantum circuits, it provides a foundation for more advanced techniques.

Challenges of Quantum Logic Synthesis:
Quantum computers, due to their reversible nature, lack the full toolkit available in classical synthesis. Unlike classical gates, where output reveals the input state, quantum operations must be reversible. This necessitates the use of reversible building blocks to construct any quantum operation.

Workaround for Non-Reversibility:
Non-reversible gates can be converted into reversible ones by preserving the original inputs and storing the result in a new qubit. For example, an n-controlled X gate allows computing a multi-AND operation between n qubits on a new qubit. Followed by another (multi-)controlled X gate on the same qubit, we can achieve an XOR operation. **XAGs (XOR and AND Graphs):**

Expressions like `(x0 AND x1 AND x2) XOR (x0 AND x1)` are called XAGs. These graphs can be conveniently represented by polynomials over the Boolean algebra (F2).

Reed–Muller Expansion for Truth Tables:
Given a truth table `T` with `n` variables, its Reed–Muller expansion `RMn(x)` is a polynomial generated recursively using the following equation:

$$\text{RMn(x)} = \text{x0} * \text{RMn-1(x) XOR (x0 XOR 1)} * \text{RMn-1(x)}$$

Here, `T0` and `T1` represent the cofactors of `T`, which are truth tables obtained by considering entries of `T` where `x0` is either 0 or 1 (see Table 1 in the provided excerpt). This recursion terminates at `n = 0` with `RM0(x)` being 0 or 1 depending on the value of `T`.

Circuit Implementation from Polynomials:
The resulting polynomial can be used to generate a corresponding quantum circuit. Each "summand" in the polynomial translates to a multi-controlled X gate. However, this method is computationally expensive as multi-controlled gates require a significant number of CNOT gates.

Gray synthesis:-

At the heart of Gray Synthesis lies the ability to control the phase shift of individual computational basis states, which are essentially the input states of our quantum circuit. By manipulating these phase shifts, we can create custom logic functions by applying them to the input register and the output qubit.

To implement this, we'll be using three types of quantum gates: CNOT (controlled-NOT), Hadamard (H), and a custom gate denoted as Tm. While directly synthesizing logic functions is an option, the paper discusses limitations associated with this approach, such as the requirement for F2-linearity, which isn't always applicable.

Let's dive into a concrete example to illustrate how Gray Synthesis works:

1. Imagine we want to create a logic function where input states with a 0 in the output qubit experience a phase shift of 0, and those with a 1 in the output qubit experience a phase shift of πT(x) (where T(x) represents the desired truth table function).

2. By applying the Gray Synthesis circuit (denoted by Ugray) to the combined input register and output qubit surrounded by Hadamard gates, we achieve this transformation:

$$\begin{aligned}
H_{\text{out}} U_{\text{gray}} H_{\text{out}} |x\rangle|0\rangle &= \frac{1}{\sqrt{2}} H_{\text{out}} U_{\text{gray}} |x\rangle(|0\rangle + |1\rangle) \\
&= \frac{1}{\sqrt{2}} H_{\text{out}} |x\rangle(|0\rangle + \exp(i\pi T(x))|1\rangle) \\
&= \frac{1}{\sqrt{2}} H_{\text{out}} |x\rangle(|0\rangle + (-1)^{T(x)}|1\rangle) \\
&= |x\rangle|T(x)\rangle.
\end{aligned}$$

To fully grasp the power of Gray Synthesis, understanding phase synthesis is crucial. The paper explores the concept of parity operators, which are essentially XOR expressions involving input variables. These operators can be used to "load" specific values onto a qubit using sequences of CNOT gates.

Furthermore, the paper introduces the notion of parity networks. These networks employ RZ gates applied to parity operators to manipulate the phase of input states. The specific phase shift for a given input state is determined by the combination of parity operators traversed and their corresponding RZ gate parameters (θp).

In essence, Gray Synthesis offers a powerful tool for designing logic circuits on quantum computers. It allows us to create custom logic functions by precisely manipulating phase shifts. The detailed explanation of parity operators and networks provides valuable insights into the underlying mechanisms of this method.We can therefore control what kind of phase each input state receives by carefully deciding on how to distribute the phase shifts θ_p. As each input state x can be uniquely identified by just looking at the set of parity operators, which return 1 when applied to x, we can give each state a unique constellation of phase shifts by iterating over every possible parity operator. The next question is, how to determine the required $\theta = (\theta_x 0\ ,\ \theta_x 1\ ,\ \theta_x 0 \oplus x1\ ..\ .)$ phase shifts for a given sequence of desired overall phase shifts $\phi = (\phi_0,\ \phi_1,...,\ \phi_2 n\ )$? In this regard, by successively writing down, which state receives which phase shift one can set up a system of linear equations. The resulting matrix is denoted as the *parity matrix* D. The corresponding system of equations therefore yields:

$$\phi = \frac{1}{2} D\theta.$$

This is achieved by ordering the rows according to the natural order of the input states (i.e.000, 001, 010.) and the columns according to an algorithmic solution of the Hamming TSP, which visits all parity operators. The resulting matrix only depends on the number of input qubits. In the case of three input qubits we get:

$$D_3 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 & -1 & 1 & -1 & 1 \\ 1 & -1 & -1 & 1 & -1 & -1 & 1 & 1 \\ -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 & -1 & 1 & -1 & 1 \\ -1 & 1 & 1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 \end{pmatrix}$$

6.2 Parity operator traversal:-

The space of parity operators plays a crucial role. This space represents all possible manipulations on n-bit systems that flip an even or odd number of bits. Finding an efficient path to visit specific points within this space, known as parity operators, is essential for certain algorithms.

One challenge arises because we only care about a specific subset of parity operators – those with a non-vanishing Hadamard coefficient (a technical term related to quantum mechanics). A straightforward approach like using the Gray code, which traverses all possible bit combinations by flipping only one bit at each step, would be inefficient as it visits many unnecessary operators.

This paper proposes a heuristic solution using two "salesmen" to navigate the space of parity operators.

1. **First Salesman:** This salesman efficiently visits the required parity operators one by one, ensuring it only flips one bit at a time. This approach resembles the Gray code in terms of minimizing bit flips, but focuses on the specific operators of interest.

2. **Second Salesman:** To ensure a closed loop and return to the starting point, a second salesman is introduced. This salesman mirrors the path of the first salesman but in reverse order. When both salesmen finish their routes, they will meet, forming a complete traversal.

Key Points of the Heuristic:

- Each salesman prioritizes visiting the closest unvisited parity operator within the targeted subset.
- While penalizing large jumps between operators was explored, it did not significantly improve efficiency.

Compared to a naive approach visiting all operators, this method reduces the number of visited operators by roughly 10%. However, it requires more classical resources (resources not specific to quantum computation) for implementation.

6.3 Phase tolerant synthesis:-

Phase tolerant synthesis emerges as a revolutionary approach in optimizing quantum circuit design. It tackles the resource inefficiency associated with traditional synthesis methods by leveraging the inherent tolerance of certain quantum operations to phase errors. This section delves deeper into the concept, incorporating relevant formulas to illustrate its power.

The Flaw in Conventional Methods:

Existing synthesis methods, like Gray and PPRM, aim to achieve a specific state within the circuit. This state features a label register, representing potential solutions, in a uniform superposition − all possibilities existing with equal probability (represented by the summation symbol Σ). This state is described by the formula:

$$|s\rangle = \sqrt{(1/2\hat{n})} * \Sigma(|x\rangle) \ (1)$$

where:

- |s⟩ represents the overall state of the circuit
- n is the number of qubits in the label register
- |x⟩ represents each possible state of the register (combination of 0s and 1s)

In order to understand how to synthesize with garbage-phases, we note that in this method of synthesis, the logical information is only captured in the phase *differences* of the 0 and 1 state of the output qubit. Following the principles applied , we arrive at:

$$\begin{aligned} H(\exp(i\phi_0)|0\rangle + \exp(i\phi_1)|1\rangle) &= \exp(i\phi_0)H(|0\rangle + \exp(i(\phi_1 - \phi_0))|1\rangle) \\ &= \begin{cases} \exp(i\phi_0)|0\rangle \text{ if } \phi_1 - \phi_0 = 0 \\ \exp(i\phi_0)|1\rangle \text{ if } \phi_1 - \phi_0 = \pi. \end{cases} \end{aligned}$$

To make this notion accessible from a more formal point of view, consider the case that we are applying Gray synthesis to the state of uniform superposition $|s\rangle$:

$$|s\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n} |x\rangle.$$

We factor out the output qubit:

$$|s\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^{n-1}} (|0\rangle + |1\rangle)|x\rangle.$$

The 'first step' in this view of Gray synthesis is synthesizing phases $\phi_{(0,x)}$ and $\phi_{(1,x)}$ on the output qubit:

$$U_{\text{gray}}^{\text{Step 1}}|s\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^{n-1}} \left(\exp(i\phi_{(0,x)})|0\rangle + \exp(i\phi_{(1,x)})|1\rangle\right)|x\rangle =: |\psi\rangle.$$

The second step would now be to synthesize a phase on the remaining qubits which could be seen as 'correcting' the garbage phase X_x. This however does not change the relative phase of the $|0\rangle$ and $|1\rangle$ state of the output qubit:

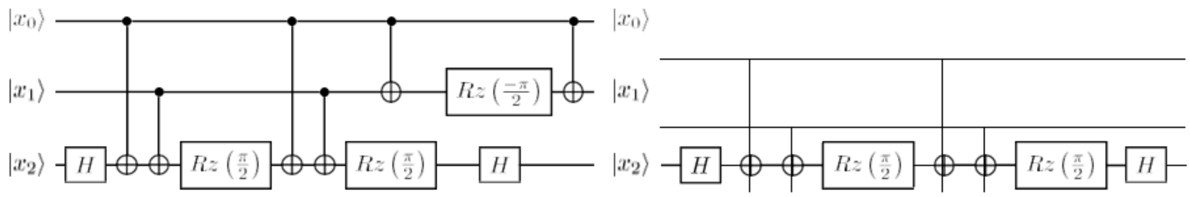
$$U_{\text{gray}}^{\text{Step 2}}|\psi\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^{n-1}} \exp(-i\chi_x)\left(\exp(i\phi_{(0,x)})|0\rangle + \exp(i\phi_{(1,x)})|1\rangle\right)|x\rangle.$$

The important point of phase tolerant synthesis is that the *difference* $\phi_{(0,x)}$ - $\phi_{(1,x)}$ determines the result of the logic output state:

$$\phi_{(0,x)} - \phi_{(1,x)} = \begin{cases} 0 & \text{if } T(x) = 0 \\ \pi & \text{if } T(x) = 1. \end{cases}$$

7 CSE synthesis:-

In this section, we present further progress beyond state-of-the-art from our research activities—the implementation of quantum logic synthesis under the consideration of restrictions in a real world setup. Phase tolerant Gray synthesis may be very resource friendly to both classical and quantum resources, however scaling on real devices is still challenging: Taking a look at equation we see that, because the only values of the entries of ϕ that can appear in the scenario of logic synthesis are $\pm\ \pi$. Therefore the values of the entries of ϕ are integer multiples of $\pm\ \pi$. This implies that if we want to encode an array with N entries,



CSE synthesis is a technique used to efficiently implement the oracle gate, which is a crucial component of the algorithm. The oracle gate marks the target state(s) in the search space, allowing Grover's algorithm to amplify the probability amplitudes of these states.

The oracle gate is implemented using CSE synthesis to perform a conditional phase shift on the target state(s). The CSE synthesis technique involves several steps:

1. Initialization: Initialize a quantum register with qubits representing the search space. These qubits are typically put into a superposition of all possible states.

2. Oracle Construction: Construct an oracle gate that flips the phase of the target state(s) while leaving all other states unchanged. This oracle gate can be represented as a diagonal matrix with phases flipped for the target state(s).

3. Conditional Shifted Expansion (CSE): Apply the oracle gate conditionally based on the current state of the qubits. This involves using controlled operations to ensure that the oracle gate is applied only when the qubits are in the target state(s). The CSE technique efficiently expands the oracle gate to act on the entire search space while avoiding unnecessary operations on non-target states.

4. Grover Iterations: Repeat the application of the oracle gate followed by the inversion about the mean (diffusion operator) for a certain number of iterations. This process amplifies the probability amplitudes of the target state(s) while suppressing the amplitudes of other states.

5. Measurement: Finally, measure the qubits to collapse the superposition and obtain the solution(s) with high probability.

CSE synthesis plays a crucial role in optimizing the efficiency of Grover's algorithm by minimizing the computational resources required to implement the oracle gate. This helps in achieving the quadratic speedup provided by Grover's algorithm for unstructured search problems.