School of Liberal Arts and Natural Sciences

University of Birmingham

Magic Square Construction and Enumeration: a Quantum Approach

Name	Lucas Van Mol
Student ID	1940383

Abstract

Magic squares are an ancient piece of mathematics. In this dissertation, a simplified version of the magic square is used to explore the use of quantum algorithms in relation to their construction and enumeration. Grover's algorithm is used for their construction, and is found to be a reasonably good alternative to similar classical alternatives. For enumeration, the quantum counting algorithm is successfully implemented for the simple square, but their use in counting solutions for higher order squares is questioned. Finally, various strategies are explored for generalizing the circuits used such that they can be applied to bigger, normal magic squares in efficient ways.

Contents

1 Introduction

1.1 Magic Squares

In my younger days, having once some leisure, (which I still think I might have employed more usefully) I had amused myself in making these kind of magic squares.

- Benjamin Franklin, in a letter to Peter Collinson [1]

1.1.1 History

Traditionally, magic squares are constructed from a set of positive integers $1, 2, ..., n^2$, such that every row, column and the two major diagonals all have the same sum, and no number is used twice. Perhaps the first recorded example appeared in the mathematical text *Shushu jiyi* ("Memoir on Some Traditions of Mathematical Art") by Xu Yue, said to have been written in 190 BCE [2].

2	9	4
7	5	3
6	1	8

Figure 1.1: Xu Yue's magic square of order 3.

Apart from their status as a mathematical curiosity, they have historically seen use in astrology, divination and the occult [2, 3]. While there are a handful of direct, real world applications of magic squares in areas such as physics, computer science, and cryptography [4–7], perhaps their most useful property is their links to groups, combinatorics and matrices [8]. For this dissertation, they will serve as a basis for exploring the applications and advantages of quantum computation.

1.1.2 Construction

Many different methods exist for constructing magic squares. Construction methods may be split into three kinds, depending on the parity of the square's size n - odd, doubly even, or singly even. For odd n, there are many simple, well-known methods, perhaps the oldest and most famous being the Siamese method [9]. Doubly even methods, for squares where n is a multiple of 4, can also be quite simple [10]. Singly even order magic square constructions are more involved, but several methods exist with tradeoffs for each [6]. Regardless, all constructions methods cited have a computational complexity $O(n^2)$, where n is the order of the magic square.

Perhaps the most significant shortcoming of these deterministic, algorithmic methods is that they are not able to generate arbitrary solutions, which can be useful in some of the applications mentioned above. Indeed, if there was a way to efficiently generate *every* solution for an n magic square, then the enumeration problem would not be so difficult to compute. However, other stochastic methods have been proposed for magic square generation [11, 12]. These use machine learning techniques such as hill climbing and evolutionary algorithms, capable of generating arbitrary solutions.

In this dissertation, the intrinsic randomness that arises from quantum superposition and measurement will be used to generate solutions to magic squares that share the same advantages as stochastic methods.

1.1.3 Enumeration

The number of unique solutions of a magic square of order n (which may be henceforth referred to as MS-n) is an unsolved problem in mathematics for $n > 5$. For $n = 1$, there is, trivially, only one solution. There are no solutions for $n = 2$. There are 8 solutions for $n = 3$, but it is common to divide the number of possible solutions by 8, due to the fact that a square's "magic" property is conserved by rotation and reflection. These isomorphisms are an important feature as they allow the search space to be reduced significantly for higher order enumerations. The full sequence up to $n = 5$ is archived by the *On-Line Encyclopedia of Integer Sequences* [13] and is stated as follows:

$$1,\ 0,\ 1,\ 880,\ 275305224. \tag{1.1}$$

The 880 solutions for $n = 4$ were first painstakingly enumerated by Frenicle de Bessy in 1693 [14], and it wasn't until 1972 that the number of solutions for $n = 5$ was considered. In the technical report *HAKMEM* [15], once described as "a bizarre and eclectic pot-pourri of technical trivia"[i], the MS-5 enumeration problem was posed as an algorithmic challenge. Schroeppel, one of the authors, subsequently computed the figure in 1973 [8]. His result marked the transition of the magic square enumeration problem from a mathematical one into a computational one.

The use of exhaustive search with computers becomes extremely difficult for this problem as n grows. Since an n by n magic square is filled with n^2 digits, the total number of ways to fill in the magic square is the number of permutations of n^2 elements, i.e. $n^2!$. Enumeration techniques used by Schroeppel and Lin *et al.* [17] use backtracking and take advantage of isomorphisms to reduce the search space as much as possible. However, the $O(n^2!)$ complexity of the problem suggests it would be extremely difficult to use these same techniques for higher n. As a rough estimate, this means that if it took only 1 second for Schroeppel's algorithm to compute the answer for $n = 5$, it would take more than $300,000,000$ years[ii] for the same algorithm to compute the answer for $n = 6$.

Despite this, estimates can be made. One estimate for the number of order 6 magic squares is $(1.7745 \pm 0.0016) \times 10^{19}$, which was found using Monte-Carlo methods [18]. This work hopes to investigate whether it's possible to improve upon this estimate using a quantum algorithm known as quantum counting, and estimates the resources needed to calculate the number of solutions to arbitrary precision.

1.2 Quantum Computing

Nature isn't classical, dammit, and if you want to make a simulation of Nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem because it doesn't look so easy.

- Richard P. Feynman, 1982 [19]

1.2.1 History

The prehistory of quantum computing can be traced back to physicist's desire to simulate quantum phenomena [19–22]. Indeed, the ability to accurately simulate physical systems has enormous practical application in any discipline that uses computational modelling. However, due to the effects of entanglement and superposition, the description of a quantum system with n particles requires keeping track of $\sim 2^n$ states. This exponential $O(2^n)$ complexity leads to the realization that a classical computer will quickly run out of memory simulating anything but the simplest quantum systems.

This desire is motivated by the discretization of the Schrödinger's equation

$$i\hbar \frac{d}{dt} |\psi\rangle = H |\psi\rangle\,, \tag{1.2}$$

which governs the dynamics of quantum systems. The solution for a time-independent Hamiltonian is

$$|\psi(t)\rangle = e^{-iHt} |\psi(0)\rangle\,. \tag{1.3}$$

The differential equation 1.3 can be discretized as in Nielsen and Chuang [23] to get

$$|\psi(t + \Delta t)\rangle \approx (I - iH\Delta t) |\psi(t)\rangle\,. \tag{1.4}$$

Hence, we can iteratively update $|\psi(t_i)\rangle$ by applying eq. (1.4) until we get to some final state $|\psi(t_f)\rangle$. The result is that time evolution of any system $|\psi\rangle$ governed by some H can be calculated. Examples could be using the Hubbard model Hamiltonian [24], describing the interaction of fermionic particles [25], or using

[i]Guy L. Steele, in the foreword to *Hacker's Delight* [16]

[ii]If the runtime is assumed to be proportional to $n^2!$: $1 \text{ sec} \times \frac{6^2!}{5^2!} \approx 2.4 \times 10^{16} \text{ sec} \approx 300 \times 10^6 \text{ yrs}$

the Ising model to identify phase transitions [26]. Solutions to these models can allow one to compute many physical properties of a material. This idea can also be expanded to using more sophisticated models, such as quantum electrodynamics and quantum chromodynamics.

The potential benefits of quantum computing outside the field of physics simulation first became apparent with the discovery of quantum algorithms that were exponentially faster than their classical counterparts, such as the Deutsch-Jozsa algorithm [27] and Peter Shor's prime factorization algorithm [28], the latter of which could be used to break some of the most popular public-key cryptography schemes such as RSA. These discoveries have propelled this blossoming field into an entirely new and interdisciplinary science of quantum computation and quantum information, consolidating major areas of physics and computer science.

1.2.2 Noisy Intermediate-Scale Quantum Era

The current state of quantum computing is often referred to as the noisy intermediate-scale quantum (NISQ) era [29]. This era is characterized by non-fault-tolerant quantum computers of ~ 100 qubits, sensitive to noise and decoherence. NISQ-specific algorithms such as the variational quantum eigensolver (VQE) or the quantum approximate optimization algorithm (QAOA) are hybrid algorithms, using a mix of classical and quantum computation, and are useful in areas like chemistry, physics, material science, cryptography and finance [29].

The characteristic properties of NISQ-era quantum computing will therefore have consequences on the experimental data collected in this dissertation. In addition, there are many different types of quantum computers, and noise profiles/error types vary significantly depending on the architecture used. Section 2.3 goes into more detail.

1.3 Grover's Algorithm & Quantum Counting

In this section, the interdisciplinary approach taken for magic square enumeration is described, notably how certain quantum algorithms can be applied to the problem. The first algorithm is Grover's algorithm, which will be capable of generating solutions to magic squares. That circuit can be expanded upon in order to implement quantum counting, which is the algorithm used to enumerate solutions.

Both of these algorithms require the use of a suitable **oracle**: a black-box function capable of discerning valid from invalid solutions. This section starts with an explanation of Grover's algorithm (used for magic square construction), including the oracle and the diffusion operator, and ends on quantum counting (used for magic square enumeration).

1.3.1 Grover's Algorithm

Grover's algorithm is also known as the quantum search algorithm. It is able to find, with high probability, an input to a black box function that produces a particular output. Classically, if N is the size of the functions' domain, we would expect to need $O(N)$ queries of the function in the worst case, as we'd need to check half the domain in order to get a 50% chance of finding a suitable input value. Grover's algorithm, however, needs just $O(\sqrt{N})$ oracle queries, presenting a modest quadratic speedup over the classical problem.

1.3.2 Oracle & Phase Kickback

We first define a function $f(x)$ that takes in a (correct or otherwise) solution x, defined such that $f(x) = 1$ if x is a valid solution to the search problem, and $f(x) = 0$ if it isn't.

The oracle is implemented as a unitary operator U_ω that is defined by its action on the computational basis [23]:

$$|x\rangle\,|q\rangle \quad \xrightarrow{U_\omega} \quad |x\rangle\,|q \oplus f(x)\rangle\,, \tag{1.5}$$

where $|x\rangle$ is the input register and $\oplus$ denotes the XOR operation. Hence the single oracle qubit $|q\rangle$ is only flipped when $f(x) = 1$, or in other words when x is a solution to the search problem. The oracle qubit $|q\rangle$ is then prepared into the $|-\rangle$ state, or in the standard computational basis:

$$|-\rangle \equiv \frac{|0\rangle - |1\rangle}{\sqrt{2}}. \tag{1.6}$$

The resulting operation of U_ω acting on $|x\rangle |-\rangle$ results in a phenomenon known as phase kickback:

$$|x\rangle |-\rangle \quad \xrightarrow{U_\omega} \quad (-1)^{f(x)} |x\rangle |-\rangle , \tag{1.7}$$

which can be trivially proven by noticing that if $f(x) = 1$, the signs of the $|0\rangle$ and $|1\rangle$ terms in $|-\rangle$ flip, resulting in an overall phase shift of -1. Hence, this phase shift is how the oracle 'marks' solutions to the search problem.

1.3.3 Grover Iteration

Grover's algorithm consists of a number of "Grover iterations", consisting of an oracle operator U_ω and a diffusion operator U_s. As illustrated in fig. 1.2, the search qubits are initially put into a superposition state $|s\rangle \equiv H^{\otimes n} |0\rangle^n$ using n Hadamard gates.

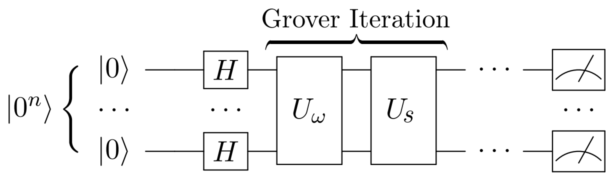


Figure 1.2: A quantum circuit diagram of Grover's algorithm. The search qubits $|0^n\rangle$ are initially put into the superposition state $|+\rangle \equiv H|0\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$. The Grover iteration itself consists of the oracle operator U_ω followed by the diffusion operator U_s. For additional information on how to read these diagrams and the gates used, refer to appendix A.

A geometric explanation of the working of Grover's algorithm is given in fig. 1.3. At the start of the algorithm in fig. 1.3a, the 2D plane spanned by the 'winner' state $|w\rangle$ and the initial superposition state caused by the Hadamard gates $|s\rangle$ is considered. The goal of Grover's algorithm is to measure the winner state $|w\rangle$. Since $|s\rangle$ is an equal superposition of all states, it contains the state $|w\rangle$ with amplitude $1/\sqrt{N}$[iii], and as such is almost, but not quite, perpendicular to $|w\rangle$. $|s'\rangle$ is defined such that it is perpendicular to $|w\rangle$, and lies on the two-dimensional plane spanned by $|w\rangle$ and $|s\rangle$.

The orthogonal vectors $|w\rangle$ and $|s'\rangle$ allow us to express $|s\rangle$ as

$$|s\rangle = \sin\theta\, |w\rangle + \cos\theta\, |s'\rangle , \tag{1.8}$$

where

$$\theta = \arcsin\langle s|w\rangle = \arcsin \frac{1}{\sqrt{N}}. \tag{1.9}$$

The U_ω and U_s operators can be thought of as reflections in this two-dimensional plane. As described in section 1.3.2, U_ω gives a negative phase to the $|w\rangle$ component of $|s\rangle$, resulting geometrically in a reflection of the state $|s\rangle$ about $|s'\rangle$ as illustrated in fig. 1.3b.

Finally, the diffusion operator U_s is defined such that it reflects about $|s\rangle$:

$$U_s = 2|s\rangle\langle s| - I. \tag{1.10}$$

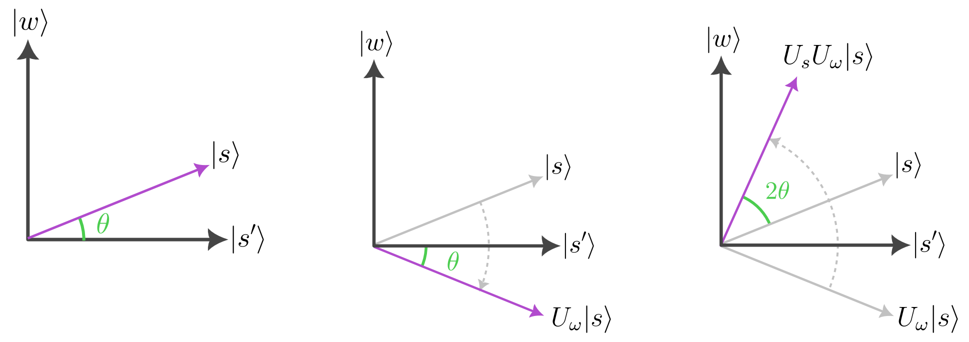
This operator is comparatively easy to construct as it's only dependent on how the initial superposition state $|s\rangle$ is set up. In the case shown in fig. 1.2, where $|s\rangle = H^{\otimes n} |0^n\rangle$, the diffuser can be implemented as

$$U_s = H^{\otimes n} X^{\otimes n} (MCZ) X^{\otimes n} H^{\otimes n}, \tag{1.11}$$

[iii]Since the probability of measuring a state is the square of its amplitude, measuring $|s\rangle$ in the computational basis would return $|w\rangle$ with probability $P = (1/\sqrt{N})^2 = 1/N$ as one would expect.

where MCZ is the multi-controlled Z gate. A derivation of this U_s is demonstrated in the online Qiskit textbook [30].

As illustrated in fig. 1.3c, one Grover iteration has brought the initial state closer to $|w\rangle$, and as such the probability of measuring $|w\rangle$ has increased.



(a) Initial state **(b)** Oracle **(c)** Diffusion

Figure 1.3: Geometric interpretation of Grover's algorithm. Applying the oracle operator U_ω and U_s brings the initial superposition state $|s\rangle$ closer to the 'winner' state $|w\rangle$, increasing the probability of measuring $|w\rangle$ at the end of the circuit.

1.3.4 Quantum Counting

Quantum counting is an algorithm designed to determine the number of solutions M to an N item search problem. Such an algorithm on a classical computer would take $\Theta(N)$ oracle calls. As explained in section 1.1.3, the factorial growth of the magic square enumeration problem make this a very difficult task for classic computers.

The quantum counting algorithm combines the Grover iteration with quantum phase estimation. It estimates the eigenvalues of the Grover iteration G in the form $e^{i\theta}$ and $e^{i(2\pi-\theta)}$, and, using an inverse Fourier transform, outputs an estimate of θ or $2\pi - \theta$ onto a register of t *counting* qubits.

The algorithm is illustrated in fig. 1.4. The t counting qubits are put into an equal superposition state, and act as control qubits to an exponentially increasing amount of Grover iterations G being applied to n search qubits. The resulting phase kickback 'writes' the phase of G in the Fourier basis onto the counting qubits. This can finally be extracted using the inverse Fourier transform $\mathcal{FT}^\dagger$. The result is an estimation of the phase θ encoded as $0.\psi_1\psi_2...\psi_t$.

As demonstrated by Nielsen and Chuang [23], the phase estimation algorithm determines θ to m bits of accuracy with a probability $1-\epsilon$, where m and ϵ are parameters of the circuit. The number of counting qubits required for a given m and e is

$$t \equiv m + \lceil \log_2(2 + (2\epsilon)^{-1}) \rceil \tag{1.12}$$

The upper bound for the error is $|\Delta M|$ is

$$|\Delta M| < \left(\sqrt{2MN} + \frac{N}{2^{m+1}} \right) 2^{-m} \tag{1.13}$$

Hence, if t is large enough, then $|\Delta M| \approx 0$ so a precise value for M can theoretically be produced with high probability, given enough qubits. However, as explained in section 4.3, the exponential increase in Grover iterations G needed along with the $\Theta(t^2)$ gate complexity of the inverse quantum Fourier transform means that finding precise values of M to large search problems is still plagued by exponential complexity problems.

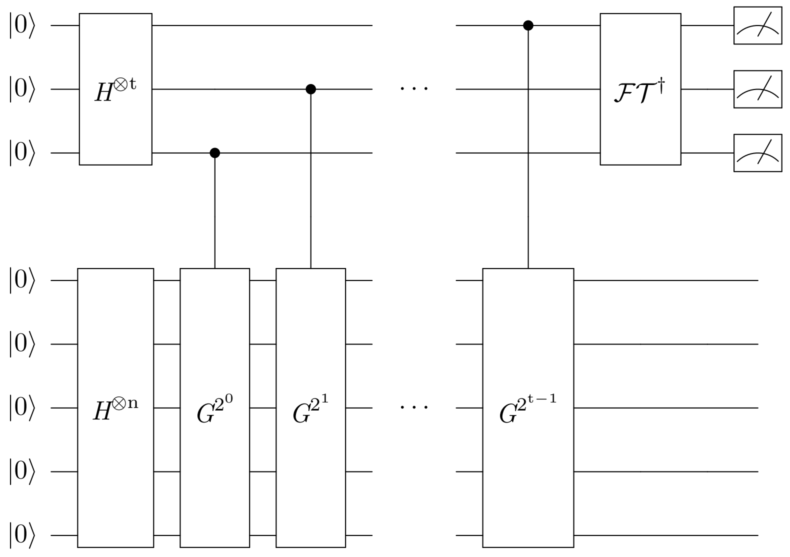


Figure 1.4: The quantum counting algorithm. In this example, there are $t = 3$ counting qubits (top register) and $n = 5$ searching qubits (bottom register).

2 Methodology

2.1 Simplified Squares

Current physical realizations of quantum computers are quite limited in their ability, especially those that are publicly available. Whilst physical quantum computers are a very active area of research, current examples of quantum advantage over classical algorithms are generally limited to specialized experiments [24].

This will drive us to investigate the magic square enumeration problem using a very simplified version of the problem. This restricted ruleset will also allow for lower use of quantum resources, and the small size allows for reasonable use on simulators.

The simplified magic square used in the experimentation section of this dissertation consists of a 3 by 3 grid, where each cell consists of a binary digit. The rules for a valid magic square is that all rows and columns must sum to the same number - this type of square is sometimes known as a ‘semi-magic square’, hence this simplified square could be referred to as a ‘binary, semi-magic square’. An example is shown in fig. 2.1.

0	1	1
1	0	1
1	1	0

Figure 2.1: A valid binary semi-magic square. In this example, rows and columns sum to 2.

This flavour of magic square means that a function that can check valid solutions needs to only make use of two simple operations. The first operation required is the ability to sum 3 binary digits, which is equivalent to a full adder circuit used in digital logic. The second operation required is one that can compare the results of two sums and check for equality. In the next section, these operations are implemented in a quantum circuit.

2.2 Subroutines

2.2.1 Full Adder

The first subroutine introduced is a full adder, shown in fig. 2.2. The circuit only needs three CNOT gates and three Toffoli gates, and the output is stored on a two qubit register *anc*, since the maximum sum of 3 bits is $1 + 1 + 1 = 11_2$.

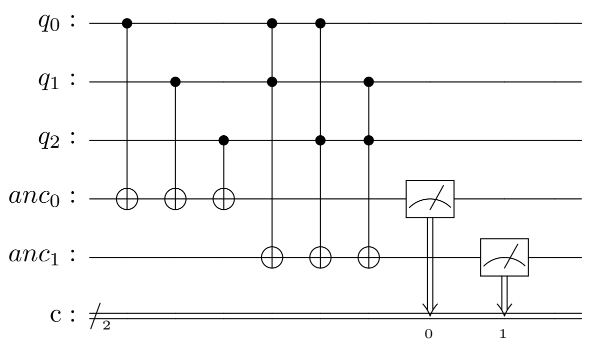


Figure 2.2: Quantum full adder. The three inputs are labelled q_0, q_1, q_2 and the two-bit output is $anc_{0/1}$. For more information on the symbols used in these diagrams refer to appendix A.

2.2.2 Equality Check

The second subroutine needed is an equality check. In order to verify the equality of two sums, two adder circuits can be used on the 6 qubits involved, and a boolean AND can be used on the two results to check for equality. This would involve more qubits and gates than necessary, however, and we can get away with using the same output qubits for both adders.

The output qubits of the adder circuit are only altered by (controlled) NOT gates. This means that if we chain two adder gates together, with different input qubits but the same output qubits, the output qubits will cancel and equal $|00\rangle$ if (and only if) both adders output the same result. If there is equality, the $|00\rangle$ result is verified using two X-gates and a Tofolli gate. Fhe final result of the equality check is output onto a separate qubit. The equality check circuit described is illustrated in fig. 2.3.

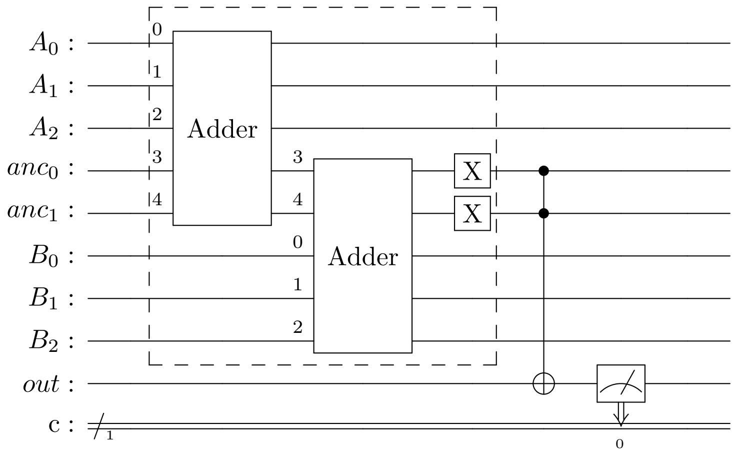


Figure 2.3: Checking if two sums are equal. The *Adder* gates' shared output qubits are named *anc*, short for *ancillary*, as they are used for intermediate calculations. The true/false result of the equality check is stored on the *out* qubit.

2.3 Quantum Processor

The qubit is the fundamental building block of quantum information processing. In physics, a qubit can be described as a two-level system (TLS), of which many examples exist in Nature. For use in a quantum computer, these systems must also be able to be manipulated and measured with relative accuracy. They must also be able to retain their quantum properties for a reasonable amount of time in order to be useful - this concept is also known as (de)coherence. There are therefore a handful of architectures that are viable for use in a quantum computer, and presented here is a non-exhaustive list.

Trapped Ion This architecture uses an ion as a TLS to represent a qubit. Charged atoms are isolated and trapped in electromagnetic wells, and subsequently cooled such that their internal state can be manipulated by incoming laser light. The most common way of representing qubits with ions is using two ground state hyperfine levels, also known as *hyperfine* qubits. Trapped ion qubits are quite long-lived: ground state hyperfine levels have been found experimentally to have lifetimes on the order of seconds to hours [31], much larger than a logic gate's operation time, which is on the order of microseconds [32]. Their stability has meant they have seen use in a handful of commercial quantum computers, such as IONQ's Aria [33] and Quantinuum's H-series [34].

Photonics In this architecture, qubits are represented as photonic quantum states, and are controlled using photonic integrated circuits (as opposed to the ubiquitous electronic integrated circuit). Photons are a particularly attractive target for storing quantum information due to their low decoherence and light-speed transmission [35].

Superconducting Superconducting qubits are constructed from solid state electronic circuits. The qubits are macroscopic, "artificial atoms" [36], and are highly configurable and scalable compared to other architectures [37]. The two logic states are the superconducting qubit's ground and excited states. There are a handful of superconducting quantum computers publicly available, including IBM's 127-qubit *ibm_ washington* [38].

For the purposes of this dissertation, we will be using IBM's Qiskit framework, and use their freely available simulator to run the quantum algorithms described. IBM's *ibmq_ QASM_ simulator* allows for simulating the noise profiles of real quantum hardware, giving a good theoretical approximation of each algorithm's performance. The noise profile chosen is that of *ibm_ washington*.

2.4 Grover's Oracle

A good implementation of Grover's oracle is a quantum circuit that is capable of discerning magic square solutions, while keeping qubits and gates used to a minimum in order to minimize the amount of noise introduced in the circuit. Firstly, a naive implementation for this problem is described, and then iteratively optimized to reduce the amount of qubits and gates used.

There are a couple of key features that will be common to any circuit used for this problem. The first is that 9 qubits will need to be used to represent the state of a binary semi-magic square, such that a single qubit represents an entry in the square, iterating through the square in row-major order. This 9 qubit encoding of the semi-magic square will be referred to as the input register. With this encoding, the example shown in fig. 2.1 would be represented as the state $|011101110\rangle$. As an aside that will be useful later, it's easy to see that there are 2^9 possible states for this problem due to the 9-bit encoding used.

Secondly, a handful of *ancilla* qubits will be required for storing intermediate calculations, such as row and column sums. Our aim will be to minimize these in order to create an efficient oracle. At least one of these will be an output qubit, that will behave like the oracle qubit $|q\rangle$ described in section 1.3.2.

Thirdly, a functioning oracle requires *uncomputation* of the ancilla qubits. This is achieved by repeating calculations after some state is written into the output qubit. This puts the ancilla qubits back into the expected state (usually $|0\rangle$) before the oracle is applied again in the next iteration. It also ensures that any phase kickback is propagated as expected back to the input register.

Finally, there are a total of 5 equalities, or clauses, that must be checked in order to verify the validity of a binary semi-magic square solution. The phrase 'all rows and columns must be equal' can be written mathematically as

$$r_1 = r_2 = r_3 = c_1 = c_2 = c_3, \tag{2.1}$$

where r_i and c_i represent the sum of row and column i, respectively. The above equation can be split into 5 distinct equality operations in a number of ways. One way of doing so, which we will come back to in section 2.4.2, is with the following 5 clauses:

$$\begin{aligned} C_1 : &\quad r_1 = r_2 \\ C_2 : &\quad r_2 = r_3 \\ C_3 : &\quad r_3 = c_1 \\ C_4 : &\quad c_1 = c_2 \\ C_5 : &\quad c_2 = c_3 \end{aligned} \tag{2.2}$$

2.4.1 Naive Oracle

A naive implementation of an oracle for this problem uses the equality circuit shown in fig. 2.3. Indeed, we can repeat the outlined part of the circuit for each of the 5 clauses, placing the adders as required. This results in a total of 10 additional ancillary qubits that will hold the results of the previous subroutine. Finally, using one more additional output qubit, a multi-controlled Toffoli gate (MCT) can then be used on the clause qubits in order to flip the output qubit if all clauses are met. In other words, all 10 clause qubits must be equal to $|1\rangle$ before the MCT in order to get a $|1\rangle$ on the output qubit.

The result is a total of $9_{input} + 10_{anc} + 1_{out} = 20$ qubits used for this circuit.

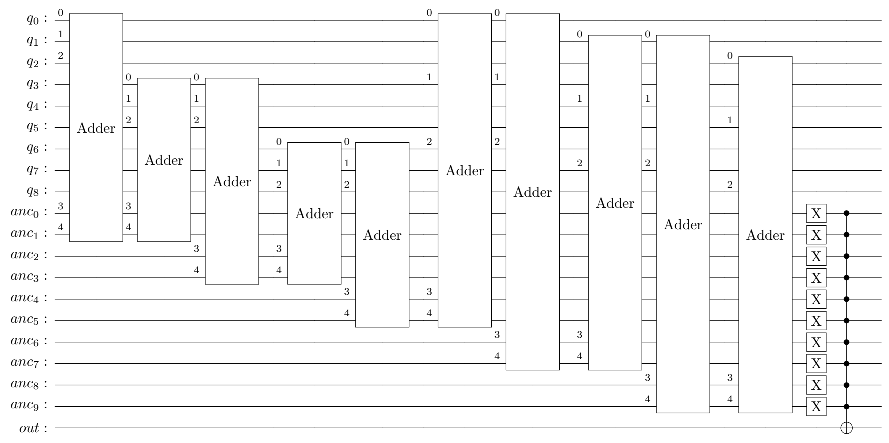


Figure 2.4: The naive oracle implementation. The equality check from fig. 2.3 is repeated 5 times, and a multi-controlled Toffoli gate is used to verify that all *anc* qubits are in the $|1\rangle$ state. Not shown is the uncomputation step, which would repeat the same X gates and Adder circuits after the MCT.

2.4.2 Optimized Oracle

Upon inspection of the previous oracle, we can see that it can be quite wasteful to waste two bits of information to describe the validity or not of a single true/false clause. Furthering this train of thought, it could be more efficient to have only 5 clause qubits, and reuse the ancillary qubits of the equality circuit (fig. 2.3). This can be achieved as shown in fig. 2.5. By repeating the pattern shown for each of the 5 clauses, we can construct an oracle that uses a total of $9_{input} + 2_{anc} + 5_{clause} + 1_{out} = 17$ qubits.

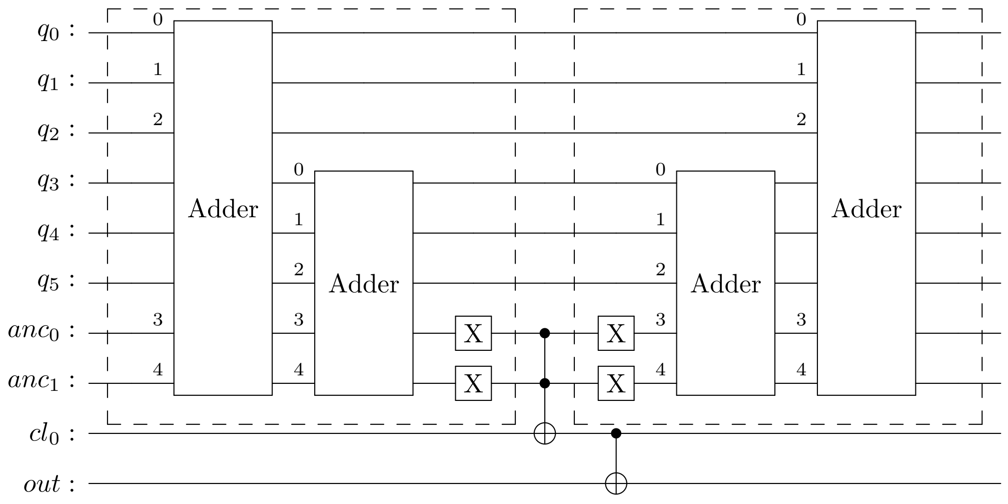


Figure 2.5: Computing the truthfulness of a clause cl_0 with uncomputation of *anc*. The two grouped elements correspond to the computation and uncomputation steps respectively.

However, we can further optimize this oracle by reducing the amount of *Adder* gates used. Clauses can essentially be 'chained' efficiently if they are put in the right order. The order shown in eq. (2.2) is a good example: we don't actually need to uncompute the r_2 sum after the first clause, since we can reuse it for the second clause. This results in significant reduction in the amounts of gates used. The resulting circuit is illustrated in fig. 2.6.

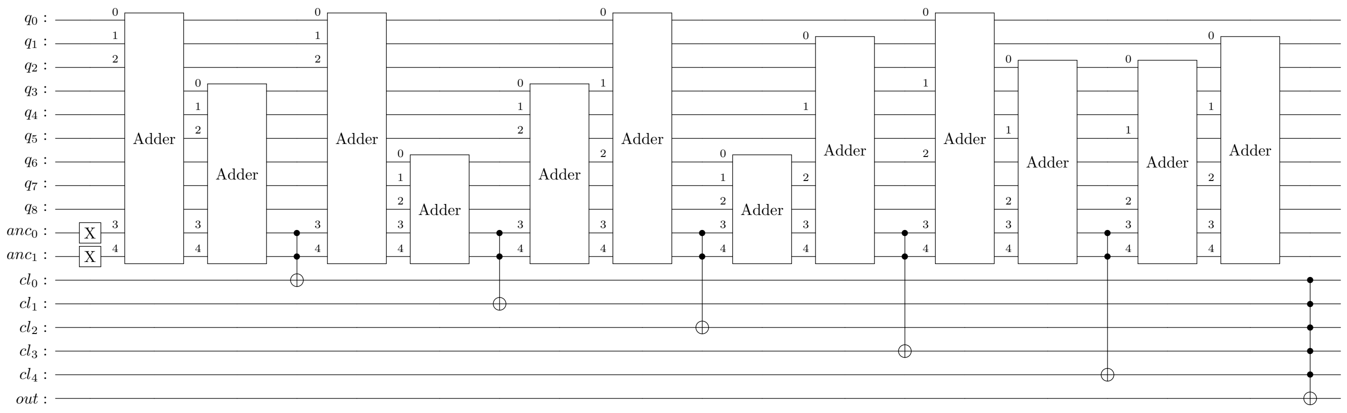


Figure 2.6: Optimized oracle. The uncomputation step after the MCT is not shown. We also introduce an additional optimization that initializes the *anc* qubits to $|1\rangle$ with an X-gate at the start of the circuit.

We thus have three different potential oracle implementations. In the next section, evidence is provided that the optimized, 17-qubit iteration show in fig. 2.6 is the most efficient, and use it in Grover's algorithm and in the quantum counting algorithm.

3 Results and Analysis

3.1 Oracle Gate Counts

In both classical and quantum computers, every processor has a distinct set of instructions that it is capable of executing. For example, it's unlikely that the 6-qubit gate MCT used in the circuit illustrated in fig. 2.6 exists as a single instruction on one of today's quantum processors. Instead, such a gate will usually be broken down into a sequence of 1- and 2-qubit gates contained in the processor's instruction set. This process is called transpilation, and as long as the gates in the instruction set form a universal gate set, the processor will be able to perform any quantum computation.

This has important consequences in the choice one makes for a particular implementation for a given algorithm. In the current NISQ era of quantum computing, it should be noted that there is a trade-off between number of gates used versus number of qubits used. More qubits means an increased susceptibility to state preparation and measurement (SPAM) errors, while more gates can be problematic if the system has high rates of gate errors. In addition, there are also practical concerns for the number of qubits used, as current quantum computers are limited in the number of available qubits. This is also true for simulators, as lower numbers of qubits can be simulated exponentially faster.

In table 1, the three oracle circuits described in section 2.4 are transpiled for a handful of processors. Clearly, the final iteration of the oracle is superior in both qubits and gates used, so that circuit will be used going forward.

Processor	**20 Qubit**	**17 Qubit**	**17 Qubit, Optimized**
IBM Eagle r1	7041	2124	1434
IBM Eagle r3	16411	4862	3232
IBM Egret r1	65243	18750	12416

Table 1: Number of gates in the transpiled circuit for each iteration of the oracle, per processor type.

3.2 Grover's Algorithm

Grover's algorithm is run with the oracle described in section 2.4.2. However, there is one important requirement that must be satisfied in order to use Grover's algorithm - the number of solutions M must be strictly less than $N/2$ [23]. If this were not the case, it would suffice to simply add a qubit to the search qubits and let the oracle enforce that it should be $|0\rangle$[iv]. Adding a qubit artificially doubles the search space - also called an augmented search space - such that the condition $M < N/2$ is always met. While it seems quite intuitive that the number of correct solutions M for an n magic square is much less than $N/2$, a derivation by Ward [39] shows that M has an upper bound of:

$$\frac{n^2!}{(2n+1)!}, \tag{3.1}$$

where n is the order of the magic square. In general, the search space N for an n magic square is the number of permutations of n^2 cells i.e. $N = n^2!$[v]. Ward's result therefore assures us that the condition

$$M \leq \frac{n^2!}{(2n+1)!} < \frac{n^2!}{2} \tag{3.2}$$

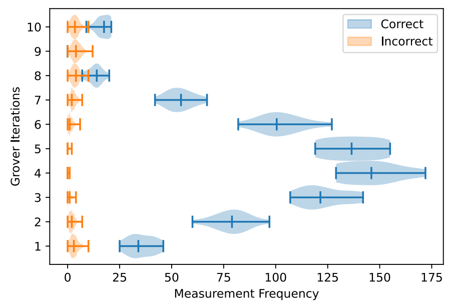
is met, for strictly positive n.

As such, no chances need to be made, and the resulting measurement frequency and accuracy is shown in fig. 3.1. The maximum accuracy achieved occurs at 4 iterations, with the algorithm measuring correct solution states 99.76% of the time.

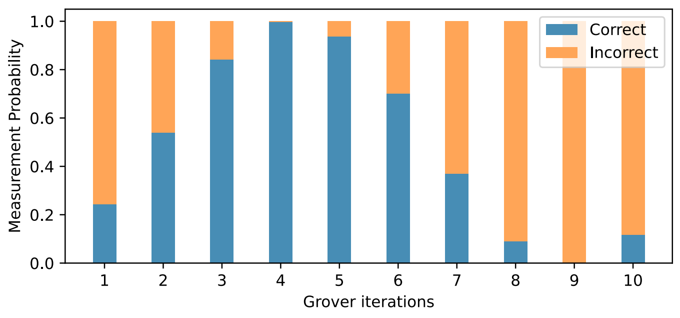
Nielsen and Chuang [23] give an upper bound for the optimal amount of Grover iterations R:

[iv] Or, equivalently, that it should be $|1\rangle$. The point is that the oracle should reject half of the solutions such that N is doubled and M stays the same.

[v] However, N could be reduced further if isomorphisms are taken into account. This is touched upon in section 4.



(a) Measurement distribution



(b) Algorithm accuracy

Figure 3.1: Results of running Grover's algorithm, varying iterations from 1 to 10, with 2048 shots each. The maximum accuracy is found to be 99.76%, at 4 iterations. In fig. 3.1a, we count how many times each possible state (from $|0\rangle^9$ to $|1\rangle^9$) is measured, and group them into correct and incorrect solutions. Correct solution states are clearly seen to be measured with higher frequency when R is close to 4. The higher variance in the measurement frequency of correct solutions is due to the fact there are far fewer correct solution states than incorrect solution states.

$$R_{opt} \leq \left\lceil \frac{\pi}{4}\sqrt{N/M} \right\rceil, \tag{3.3}$$

where N is the size of the solution space, and M is the number of correct solutions. A separate analysis by Boyer *et al.* [40] gives a tight bound on the average number of iterations in order to measure a correct solution with a probability of at least 50%:

$$R_{50\%} = \left\lfloor (\sin\frac{\pi}{8})\sqrt{\lfloor N/M \rfloor} \right\rfloor. \tag{3.4}$$

N is relatively straightforward to calculate for most search problems, and for this problem it's simply 2^9 due to the 9 (qu)bit representation of the binary semi-magic square. M is much harder to calculate - finding an easy way to calculate it in the general case would mean we have solved the magic square enumeration problem. However, due to the small size of the studied problem, it can be calculated through brute force methods, for which we find $M = 14$. Thus, applying eq. (3.3) and eq. (3.4) gives $R_{opt} \leq 5$ and $R_{50\%} = 2$ which agrees with the experimental result.

The large separation between the measurement frequency of correct vs incorrect solutions for $1 \leq R \leq 7$ in fig. 3.1a suggests it can be easy to determine the correctness of a solution if the algorithm is run for multiple shots, even for low R. Verification of a given solution can be done efficiently on a classical computer in a polynomial $O(n^2)$ steps for an n-magic square, since we simply need to go over each cell O(1) times in order to verify all the sums required. There is therefore a tradeoff between running more Grover iterations R such that we approach R_{opt} or choosing a lower R but with more shots. Finding an optimal balance will depend on the speed and accuracy of the physical quantum device.

3.3 Quantum Counting

The results of the quantum counting algorithm are shown in fig. 3.2. Two states $|\psi\rangle, |\psi'\rangle$ are measured with much higher frequency than the others: these are $|0111001_2\rangle = |57_{10}\rangle$ and $|1000111_2\rangle = |71_{10}\rangle$ and correspond to the eigenvalues $e^{i\theta}$ and $e^{i(2\pi-\theta)}$ $(0 \leq \theta < 2\pi)$. The binary representation of $|\psi\rangle \equiv |\psi_1\psi_2...\psi_t\rangle$ represents the binary decimal $0.\psi_1\psi_2...\psi_t$, where 0 corresponds to $\theta = 0$ and 1 corresponds to $\theta = 2\pi$. Hence the value of θ is calculated as follows:

$$\theta = \frac{\psi}{2^t}2\pi. \tag{3.5}$$

Nielsen and Chuang [23] also derive the equation for the estimated number of solutions M_{est} giving the equation

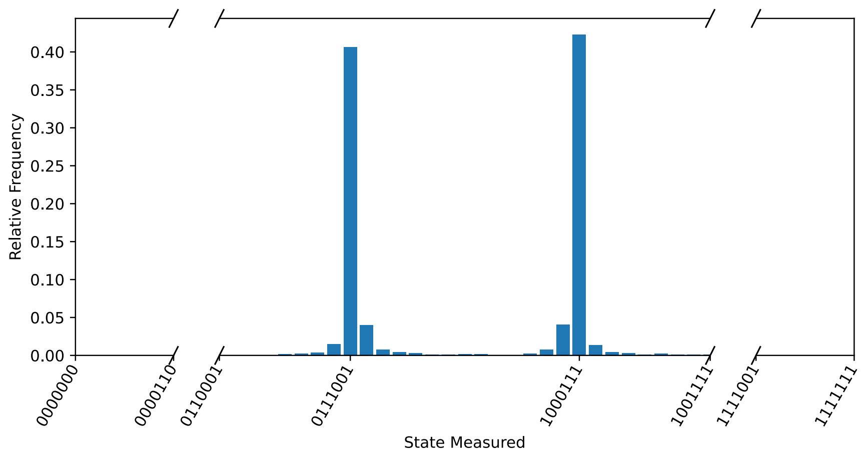


Figure 3.2: Results for quantum counting with $t = 7$ counting qubits (4000 shots, simulator). We set t directly instead of choosing a desired precision m and error rate ϵ separately as suggested by section 1.3.4, such that the maximal amount of resources are put to use.

$$M_{\text{est}} = N \sin^2(\theta/2). \tag{3.6}$$

Plugging in $|\psi\rangle$ or $|\psi'\rangle$ gives $M_{\text{est}} = 15$. This is within one to the actual count of 14. The upper bound for the error, as stated in eq. (1.13), is $|\Delta M| = 11.21$ (choosing $m = t - 1$) which is still quite significant. However, this can be improved with more counting qubits, although we start to reach the limits of what can be simulated due to the high amounts of entangled states in the algorithm.

4 Generalization to n-Magic Squares

4.1 n-Magic Square Encoding

One of the biggest advantages of the binary semi-magic square used in this dissertation is that every element of the can be represented by a single bit, and that there are no restrictions on how many 1's or 0's can be contained inside the square. This means one can instantly have a space-efficient, n^2-qubit encoding of an n by n binary semi-magic square. This isn't the case for the general magic square, however, and the choice of encoding will have consequences on how an oracle may be implemented.

4.1.1 Simple $\log_2 n^2$ Qubit Encoding

We will firstly consider the simplest representation for a general magic square. An n by n magic square contains the integers in the set $\{1, 2, ..., n^2\}$, this means that each cell must be able to contain the highest number n^2 if needed. Without loss of generality, we can represent each cell with a $\lceil \log_2(n^2) \rceil$-qubit word, such that any cell can hold the binary representation of any number from 0 to $n^2 - 1$. Hence the total amount of qubits used in this representation is:

$$n^2 \lceil \log_2(n^2) \rceil. \tag{4.1}$$

With this encoding, an oracle implementation must verify additional factors. For instance, we must also check that each element in the set $\{1, 2, ..., n^2\}$ is only used once. However, it can be possible to help alleviate these additional required checks using a smarter initial superposition state. Previously, initializing the state $|s\rangle$ with Hadamard gates as illustrated in fig. 1.4 means that the quantum counting algorithm has to search over the entire Hilbert space. Using a different superposition, it's possible to reduce the search space of the problem significantly.

To illustrate why this may be useful, let us take the encoding described above for a magic square of order 3. Each cell will be described by $\lceil \log_2(3^2) \rceil = 4$ qubits. In this configuration, the states $|0000\rangle^9$ or $|1111\rangle^9$ will be contained in the initial superposition state $|s\rangle$, even though it is obvious that they are clearly invalid as they contain repeated numbers, and numbers outside the valid range ($[0000_2..1000_2]$ for $n = 3$) in the case of $|1111\rangle^9$. The ideal choice for the initial state would be an equal superposition of all possible permutations of $|1, 2, ..., n^2\rangle$:

$$|s'\rangle \equiv \frac{1}{\sqrt{n^2!}} \sum_{k=1}^{n^2!} \hat{\pi}_k \left|1, 2, ..., n^2\right\rangle, \tag{4.2}$$

where $\hat{\pi}_k$ operating on state $|1, 2, ..., n^2\rangle$ gives the k^{th} permutation of $|1, 2, ..., n^2\rangle$. This encoding would allow us to eliminate all states with repeated elements and invalid elements, reducing the oracle's domain N to $n^2!$.

However, preparing the initial state $|s'\rangle$ may be difficult, since an operation such as

$$\left|1, 2, ..., n^2\right\rangle \quad \overset{O}{\longrightarrow} \quad \frac{1}{\sqrt{n^2!}} \sum_{k=1}^{n^2!} \hat{\pi}_k \left|1, 2, ..., n^2\right\rangle \tag{4.3}$$

is not a reversible. Due to the unitarity of quantum mechanics, all quantum circuits must be reversible, and a necessary condition for reversibility is the mapping of states one-to-one such that no information is lost. Thus, additional information must be saved in $\log(n^2!)$ ancillary qubits as such:

$$\left|1, 2, ..., n^2\right\rangle |0\rangle \quad \overset{O'}{\longrightarrow} \quad \frac{1}{\sqrt{n^2!}} \sum_{k=1}^{n^2!} \hat{\pi}_k \left|1, 2, ..., n^2\right\rangle |k\rangle . \tag{4.4}$$

The $|k\rangle$ term stores the index of each permutation and hence 'stores' the previously lost information, meaning the operator O' is reversible. The auxillary register can simply be ignored upon measurement.

Another solution could be to use less restrictive superpositions that don't require additionally ancillary qubits. One candidate is the Dicke state [41, 42], written as $|D_k^t\rangle$, corresponding to all binary strings of length t with a *Hamming weight* of k (i.e. containing k ones). For example, for a magic square of order 3, any permutation of

$$\{0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000\}$$

will contain exactly $k = 13$ ones. Using eq. (4.1), the state is encoded with $t = 3^2\lceil\log_2(3^2)\rceil = 36$ qubits. The Dicke state $\left|D^{36}_{13}\right\rangle$ can therefore be used, reducing N from $36! = O(10^{41})$ to ${}_{36}C_{13} = \frac{36!}{13!(36-13)!} = O(10^9)$. Note that some clearly invalid states would still be allowed, such as

$$\{0000, 0000, 0000, 0000, 0000, 0001, 1111, 1111, 1111\}.$$

Finally, a more complex superposition encoding, perhaps inspired by the generating vectors of Lin *et al.* [17] may also be able to reduce N below $n^2!$ by eliminating certain isomorphisms.

4.1.2 Lehmer Codes

A single n magic square solution can also be considered as a unique permutation of n^2 distinct items. It may therefore be fruitful to look at permutation encodings such as Lehmer codes in order to represent magic square states. The latter makes use of the fact that there are $n^2!$ permutations of n^2 items. A Lehmer code of an n^2 sequence coincides with the factorial number system representation of its index in the list of permutations of n^2 in lexicographical order (where the list starts at index 0).

For example, say we use the set

$$S = \{0, 1, 2, 3\}.$$

The Lehmer code for the permutation $1, 3, 2, 0$ would be $1{:}2{:}1{:}0_!$, where the subscript $_!$ denotes a factoradic number. The number $1{:}2{:}1{:}0_!$ can be interpreted as follows:

- Starting with the first digit in $1{:}2{:}1{:}0_!$, select $S[1] = 1$ and remove it such that $S = \{0, 2, 3\}$
- Now for the second digit, select $S[2] = 3$ and remove it such that $S = \{0, 2\}$
- For the third digit, select $S[1] = 2$ and remove it such that $S = \{0\}$
- For the final digit, $S[0] = 0$, and $S =$
- Thus we have recovered the original permutation $1, 3, 2, 0$

The general algorithm for encoding an MS-n with this code is shown in algorithm 1. By analysis, the total amount of qubits needed for this representation is:

$$\sum_{i=1}^{n^2} \lceil\log_2(i)\rceil \approx \lceil\log_2(n^2!)\rceil \tag{4.5}$$

which can be significantly less than the previous $n^2\lceil\log_2(n^2)\rceil$ as n increases. This also means that instead of checking that each element in the set $\{1, 2, ..., n^2\}$ is only used once, as in the previous solutions, we must now check that each 'digit' in the factorial number system used by the Lehmer code is valid.

For such an encoding to be effective, a circuit is needed that's able to decode or 'select' certain digits efficiently. The classical algorithm for decoding the i^{th} digit of a Lehmer code runs in $O(i)$. However, it may be the case that the overhead required for a quantum version of it proves to be too expensive for Lehmer encoding to be worth it.

4.2 n-Qubit Addition on a Quantum Computer

For general magic squares, row and column summation will be more complex. Up until now we have been dealing with magic squares where each cell is a single binary digit - resulting in small, simple binary addition circuits to compute sums. However, to verify the solution, say, of the 3 by 3 magic square illustrated in fig. 1.1, we must be able to perform more difficult computations, such as $2 + 9 + 4$.

Algorithm 1 MS-n Lehmer Code Representation

```
Input: An array $M$ corresponding to a permutation of the set $\{1, 2, ..., n^2\}$
Output: Lehmer encoding $L = [a_{(n^2-1)}, a_{(n^2-2)}, ..., a_{(1)}]$
 1: $A \leftarrow [1, 2, ..., n^2]$
 2: $i \leftarrow 0$
 3: $L \leftarrow [\ ]$
 4: while $i < n^2 - 1$ do        ▷ The last digit of a factoradic number is always 0, so it is skipped.
 5:     $m \leftarrow M[i]$
 6:     $j \leftarrow 0$
 7:     while $a \neq m$ do
 8:         $a \leftarrow A[j]$       ▷ If this overflows, then the input was not a valid permutation
 9:         $j \leftarrow j + 1$
10:     end while
11:     Append $j$ to $L$
12:     Pop $A[j]$
13:     $i \leftarrow i + 1$
14: end while
```

One way of achieving this is to build on the analogy of the digital full adder used in this project. Indeed, a 3-bit adder can be extended into an n-bit reversible adder using additional carry operations [43].

However, such methods use $3n$ qubits to compute two n bit numbers. Arithmetic on quantum computers can be performed more efficiently with use of the quantum Fourier transform (QFT). One solution, known as the Draper adder [43], transforms one of the addends into the Fourier basis, and uses controlled phase gates to perform addition in the Fourier basis, and ends with an inverse Fourier transform so that the solution may be measured.

For example, say we want to sum two numbers a and b. We use the QFT to encode a in the relative phase of the states of a uniform superposition of all states in the computational basis. Using the computational basis $|0\rangle$, $|1\rangle$, $|2\rangle$, ... , $|d-1\rangle$, the QFT gives

$$\mathcal{QFT}\,|a\rangle\,|b\rangle = \frac{1}{\sqrt{d}}\sum_{k=0}^{d-1} e^{i\frac{2\pi ak}{d}}\,|k\rangle\,|b\rangle\,. \tag{4.6}$$

As we can see, the states $|k\rangle$ are separated by a relative phase of $e^{i\frac{2\pi a}{d}}$. In order to increment by b in the Fourier basis, we want this relative phase to be $e^{i\frac{2\pi(a+b)}{d}}$. Hence we need an operator O that achieves the following:

$$\sum_{k=0}^{d-1} e^{i\frac{2\pi ak}{d}}\,|k\rangle\,|b\rangle \quad \xrightarrow{O} \quad \sum_{k=0}^{d-1} e^{i\frac{2\pi ak}{d}} e^{i\frac{2\pi bk}{d}}\,|k\rangle\,|b\rangle = \sum_{k=0}^{d-1} e^{i\frac{2\pi(a+b)k}{d}}\,|k\rangle\,|b\rangle\,. \tag{4.7}$$

The operator O can be constructed using controlled rotation gates CR(θ), controlled by b and rotating the qubits in a. Then, we can simply use the inverse quantum Fourier transform as such:

$$\mathcal{QFT}^{\dagger}\frac{1}{\sqrt{d}}\sum_{k=0}^{d-1} e^{i\frac{2\pi(a+b)k}{d}}\,|k\rangle\,|b\rangle = |a+b\rangle\,|b\rangle \tag{4.8}$$

and measure the first register to get the results. This idea can also be expanded to use three registers, such that the Draper adder $\mathcal{D}$ does not mutate the addends:

$$\mathcal{D}\,|a\rangle\,|b\rangle\,|0\rangle = |a\rangle\,|b\rangle\,|a+b\rangle \tag{4.9}$$

Equality checking is also subtly different. If we use the definition of the magic square in the introduction, then every n magic square solution will have the same sum along their columns, rows, and diagonals; this is known as the magic constant:

$$M = n \cdot \frac{n^2+1}{2} \tag{4.10}$$

We therefore no longer need to compare two rows/columns, but rather each summation can be compared to the magic constant M, which can be hard-coded into the circuit. For example, for an MS-n, the magic sum M can be encoded into some ancillary qubits. Then, using a variation of the Draper adder, we can compute $\frac{n^3+n}{2} - \sum_{i=0}^{n-1} r_i$ for some row r composed of the elements r_i. The result can then be checked if it's equal to 0.

4.3 Quantum Counting Scaling

The quantum counting algorithm only improves the classical search algorithm from $O(N)$ to $O(\sqrt{N})$, where N is the size of the search problem. In the magic square case, $N = O(2^{n^2})$ for an n-magic square. Hence, the quantum counting algorithm will still have an exponential complexity of $O(2^{\frac{n}{2}^2})$.

In order to see the scaling issues that arise due to this complexity, we can estimate some of the resources required for computing a similar estimation as Pinn and Wieczerkowski [18]. Using Monte-Carlo methods, they estimated the number of solutions for $n = 6$ to be $(1.7745 \pm 0.0016) \times 10^{19}$.

This therefore gives us the following parameters:

$$\begin{aligned} N &= 6^2! \\ M &\approx 10^{19} \\ |\Delta M| &\approx 10^{16}. \end{aligned} \tag{4.11}$$

Solving eq. (1.13) for m gives:

$$m = \log_2\left(\frac{\sqrt{NM} + \sqrt{N(M+|\Delta M)|}}{\sqrt{2}|\Delta M|}\right). \tag{4.12}$$

Plugging in the numbers gives $m \approx 47.95$. Using eq. (1.12) and choosing $\epsilon = 0.33$ means that we'd need $t = 50$ counting qubits in order to measure M with an accuracy of $|\Delta M| < 10^{16}$ and 67% probability.

However, increasing the number of counting qubits t in the upper register comes with an increasing cost in the Grover iterations required and the complexity of the inverse quantum Fourier transform. An order of magnitude for the total number of grover iterations required for $t = 50$ is

$$\sum_{i=0}^{t-1} 2^i = O(10^{15}), \tag{4.13}$$

which is much too large even for the most optimistic estimates of the capabilities of future quantum computers.

5 Conclusion

In this dissertation, it is shown that it is possible to find solutions to magic squares with quantum unstructured search using Grover's algorithm. The algorithm used is in a higher complexity order than the $O(n^2)$ methods described in section 1.1.2; the quantum search algorithm has an exponential complexity $O(2^{\frac{n}{2}^2})$. Therefore, if the 'uniqueness' of a solution is not important, then it will be more efficient to use classical construction methods. However, these construction methods are only capable of finding a small subset of all the possible solutions, and if we wish to find solutions outside of these, then other techniques must be used. The algorithm for this dissertation is therefore more comparable to stochastic search methods such as the evolutionary algorithm proposed by Xie and Kang [11]. The results are promising for simplified magic squares. Unfortunately, the current state of quantum computing means it is too early to determine the practical advantage, or lack thereof, that a quantum alternative could have over these types of machine learning-based methods.

With the possibility of better quantum systems in the future, it would of course be interesting to look at implementing the methods described in section 4. Most notably, it could be fruitful to develop a reversible circuit implementation of the 'select' function of the Lehmer code, and compare such a permutation encoding to the simple method of encoding. This could have interesting consequences in similar search over permutations problems that are different to just magic square construction and enumeration. The study of permutations is not only an important part of combinatorics and group theory, but plays a role in other disciplines as well - such as permutations in RNA sequences [44] or particle states in quantum physics [45].

As for magic square enumeration, while section 3.3 demonstrates positive results for low qubit numbers, the scaling issues and estimates seen for bigger squares and higher precision are not promising. However, there are certain optimizations that can be explored and that may be significant. Future research that directly builds upon this dissertation may be able to construct normal magic squares for $n = 3$ or higher, comparing the encoding techniques described in section 4 - provided there are quantum computers available that are big enough.

The field of experimental quantum computing is currently in its infancy, patiently awaiting the contingent arrival of full scale quantum computers that break out of the NISQ era. The potential for these systems to be able to accurately simulate complex physical models would have enormous consequences in a wide array of fields across the sciences [46]. In terms of quantum simulation, the branches of science most affected will be those that study many-body quantum systems with many degrees of freedom - this includes areas such as condensed-matter physics, atomic physics and quantum chemistry. This idea of quantum simulation, in addition to the many quantum algorithms that have not been mentioned in this dissertation, have such wide-reaching applications that their study could yield incredible results in a post-NISQ world.

References

[1] Benjamin Franklin. *Experiments And Observations On Electricity, Made At Philadelphia in America: To which are Added, Letters and Papers On Philosophical Subjects*. London: David Henry, 1769, pp. 350–354. URL: https://www.google.co.uk/books/edition/_/-48_AAAAcAAJ.

[2] Ho Peng Yoke. "Magic Squares in China". In: *Encyclopaedia of the History of Science, Technology, and Medicine in Non-Western Cultures* (Mar. 2008), pp. 1251–1252. DOI: 10.1007/978-1-4020-4425-0_9350. URL: https://link.springer.com/referenceworkentry/10.1007/978-1-4020-4425-0_9350.

[3] Owen Davies. *Grimoires: A history of magic books*. Oxford University Press, 2010, pp. 3–27.

[4] Boris Aronov et al. "A generalization of magic squares with applications to digital halftoning". In: *Theory of Computing Systems* 42.2 (Feb. 2008), pp. 143–156. ISSN: 14324350. DOI: 10.1007/S00224-007-9005-X/METRICS. URL: https://link.springer.com/article/10.1007/s00224-007-9005-x.

[5] Peyman Fahimi and Ramin Javadi. *An Introduction to Magic Squares and Their Physical Applications*. 2015. URL: https://www.researchgate.net/publication/297731505.

[6] Zhenhua Duan et al. "Improved even order magic square construction algorithms and their applications in multi-user shared electronic accounts". In: *Theoretical Computer Science* 607 (Nov. 2015), pp. 391–410. ISSN: 0304-3975. DOI: 10.1016/J.TCS.2015.07.053.

[7] Chin-Chen Chang et al. "An image authentication scheme using magic square". In: IEEE. 2009, pp. 1–4. ISBN: 9781424445202. DOI: 10.1109/ICCSIT.2009.5234866.

[8] Martin Gardner. "Mathematical Games". In: *Scientific American* 234.4 (1976), pp. 126–130.

[9] Simon de la Loubère. *A new historical relation of the kingdom of Siam*. London: T. Horne, 1693, pp. 227–247. URL: https://digital.library.cornell.edu/catalog/sea130.

[10] Grasha Jacob and A. Murugan. *On the Construction of Doubly Even Order Magic Squares*. 2014. arXiv: 1402.3273 [math.HO].

[11] Tao Xie and Lishan Kang. "An evolutionary algorithm for magic squares". In: *The 2003 Congress on Evolutionary Computation*. Vol. 2. 2003, pp. 906–913. DOI: 10.1109/CEC.2003.1299763.

[12] Ahmed Kheiri and Ender Özcan. "Constructing Constrained-Version of Magic Squares Using Selection Hyper-heuristics". In: *The Computer Journal* 57.3 (Mar. 2014), pp. 469–479. ISSN: 0010-4620. DOI: 10.1093/COMJNL/BXT130. URL: https://academic.oup.com/comjnl/article/57/3/469/407399.

[13] *A006052 - OEIS*. URL: https://oeis.org/A006052 (Accessed Feb. 09, 2023).

[14] Frenicle de Bessy. "Des Carrez ou Tables Magiques". In: *Divers ouvrages de mathématique et de physique*. 1693, pp. 423–483.

[15] M. Beeler, R.W. Gosper, and R. Schroeppel. *HAKMEM*. Tech. rep. MIT AI Lab, Feb. 1972. URL: https://dspace.mit.edu/handle/1721.1/6086.

[16] Henry Warren. *Hacker's Delight*. 2nd ed. Addison-Wesley Professional, 2012. ISBN: 9780321842688.

[17] Ziqi Lin et al. "Generation of all magic squares of order 5 and interesting patterns finding". In: *Special Matrices* 4.1 (Jan. 2016), pp. 110–120. ISSN: 23007451. DOI: 10.1515/SPMA-2016-0011/PDF.

[18] Klaus Pinn and Christian Wieczerkowski. "Number of Magic Squares from Parallel Tempering Monte Carlo". In: *International Journal of Modern Physics C* 9.4 (1998), pp. 541–546. ISSN: 01291831. DOI: 10.1142/S0129183198000443.

[19] Richard P. Feynman. "Simulating physics with computers". In: *International Journal of Theoretical Physics* 21.6-7 (June 1982), pp. 467–488. ISSN: 00207748. DOI: 10.1007/BF02650179/METRICS. URL: https://link.springer.com/article/10.1007/BF02650179.

[20] R. P. Poplavskiĭ. "Thermodynamic Models of Information Processes". In: *Soviet Physics - Uspekhi* 18.3 (Feb. 1975), pp. 222–241. ISSN: 21695296. DOI: 10.1070/PU1975V018N03ABEH001955/XML. URL: https://iopscience.iop.org/article/10.1070/PU1975v018n03ABEH001955.

[21] Yuri I. Manin. *Computable and uncomputable (in Russian)*. Moscow, 1980.

[22] Yuri I. Manin. "Classical computing, quantum computing, and Shor's factoring algorithm". In: *Asterisque* 266 (Mar. 1999), pp. 375–404. ISSN: 03031179. DOI: 10.48550/arxiv.quant-ph/9903008. URL: https://arxiv.org/abs/quant-ph/9903008v1.

[23] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000. ISBN: 9781107002173.

[24] Andrew J Daley et al. "Practical quantum advantage in quantum simulation". In: *Nature* 607 (2022). DOI: 10.1038/s41586-022-04940-6. URL: https://doi.org/10.1038/s41586-022-04940-6.

[25] Abhinav Kandala et al. "Hardware-efficient Variational Quantum Eigensolver for Small Molecules and Quantum Magnets". In: *Nature* 549.7671 (Apr. 2017), pp. 242–246. DOI: 10.1038/nature23879. URL: http://arxiv.org/abs/1704.05018http://dx.doi.org/10.1038/nature23879.

[26] Alba Cervera-Lierta. "Exact Ising model simulation on a quantum computer". In: *Quantum* 2 (Dec. 2018), p. 114. ISSN: 2521327X. DOI: 10.22331/q-2018-12-21-114. URL: https://quantum-journal.org/papers/q-2018-12-21-114/.

[27] David Deutsch and Richard Jozsa. "Rapid solution of problems by quantum computation". In: *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439.1907 (Dec. 1992), pp. 553–558. ISSN: 0962-8444. DOI: 10.1098/rspa.1992.0167.

[28] Peter W. Shor. "Algorithms for quantum computation: Discrete logarithms and factoring". In: *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS* (1994), pp. 124–134. ISSN: 02725428. DOI: 10.1109/SFCS.1994.365700.

[29] Michael Brooks. "Beyond quantum supremacy: the hunt for useful quantum computers". In: *Nature* 574.7776 (Oct. 2019), pp. 19–21. ISSN: 14764687. DOI: 10.1038/D41586-019-02936-3.

[30] *Grover's Algorithm*. URL: https://learn.qiskit.org/course/ch-algorithms/grovers-algorithm#3qubit-implementation (Accessed Apr. 18, 2023).

[31] S. Olmschenk et al. "Manipulation and detection of a trapped Yb+ hyperfine qubit". In: *Physical Review A - Atomic, Molecular, and Optical Physics* 76.5 (Nov. 2007), p. 052314. ISSN: 10502947. DOI: 10.1103/PHYSREVA.76.052314/FIGURES/9/MEDIUM. URL: https://journals.aps.org/pra/abstract/10.1103/PhysRevA.76.052314.

[32] Reinhold Blümel et al. "Efficient, stabilized two-qubit gates on a trapped-ion quantum computer". In: *Phys. Rev. Lett.* 126 (22 June 2021), p. 220503. DOI: 10.1103/PhysRevLett.126.220503. URL: https://link.aps.org/doi/10.1103/PhysRevLett.126.220503.

[33] *IonQ Aria*. URL: https://ionq.com/quantum-systems/aria (Accessed Apr. 09, 2023).

[34] *Quantinuum | Hardware*. URL: https://www.quantinuum.com/hardware (Accessed Apr. 09, 2023).

[35] Jianwei Wang et al. "Integrated photonic quantum technologies". In: *Nature Photonics 2019 14:5* 14.5 (Oct. 2019), pp. 273–284. ISSN: 1749-4893. DOI: 10.1038/s41566-019-0532-1. URL: https://www.nature.com/articles/s41566-019-0532-1.

[36] P. Krantz et al. "A quantum engineer's guide to superconducting qubits". In: *Applied Physics Reviews* 6.2 (June 2019), p. 021318. ISSN: 19319401. DOI: 10.1063/1.5089550. URL: https://aip.scitation.org/doi/abs/10.1063/1.5089550.

[37] He-Liang Huang et al. "Superconducting quantum computing: a review". In: *Science China Information Sciences* 63 (2020), pp. 1–32. DOI: 10.1007/s11432-020-2881-9. URL: https://doi.org/10.1007/s11432-020-2881-9.

[38] *IBM Quantum*. URL: https://quantum-computing.ibm.com/ (Accessed Apr. 09, 2023).

[39] James E. Ward. "Vector Spaces of Magic Squares". In: *Mathematics Magazine* 53.2 (Mar. 1980), p. 108. ISSN: 0025570X. DOI: 10.2307/2689960.

[40] Michel Boyer et al. "Tight bounds on quantum searching". In: *Fortschritte der Physik: Progress of Physics* 46.4-5 (1998), pp. 493–505.

[41] R H Dicke. "Coherence in Spontaneous Radiation Processes". In: *Physical Review* 93.1 (Jan. 1954), pp. 99–110. ISSN: 0031899X. DOI: 10.1103/PhysRev.93.99. URL: https://journals.aps.org/pr/abstract/10.1103/PhysRev.93.99.

[42] Andreas Bärtschi and Stephan Eidenbenz. "Deterministic Preparation of Dicke States". In: *Fundamentals of Computation Theory: 22nd International Symposium*. Springer. 2019, pp. 126–139. DOI: 10.1007/978-3-030-25027-0_9. URL: https://doi.org/10.1007/978-3-030-25027-0_9.

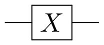
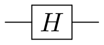
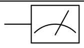
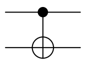
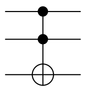
[43] Thomas G. Draper. *Addition on a Quantum Computer*. 2000. arXiv: quant-ph/0008033 [quant-ph].

[44] Iris Eckert et al. "Discovery of natural non-circular permutations in non-coding RNAs". In: *Nucleic Acids Research* 51.6 (2023), pp. 2850–2861. DOI: 10.1093/nar/gkad137. URL: https://doi.org/10.1093/nar/gkad137.

[45] Stephen French and Dean Rickles. "Understanding permutation symmetry". In: *Symmetries in physics: Philosophical reflections* (2003), pp. 212–238.

[46] J Ignacio Cirac and Peter Zoller. "Goals and opportunities in quantum simulation". In: *Nature physics* 8.4 (2012), pp. 264–266.

A Quantum Gate Notation

Diagram	**Matrix**	**Description**
	$\begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}$	The **NOT gate**, also sometimes illustrated as $\oplus$, flips the amplitudes between the $\|0\rangle$ and $\|1\rangle$ states.
	$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}$	The **Hadamard** gate, when acted on $\|0\rangle$, creates the equal superposition state $\|+\rangle \equiv (\|0\rangle + \|1\rangle)/\sqrt{2}$.
	N/A	Upon **measurement**, a state $\|s\rangle = \alpha\|0\rangle + \beta\|1\rangle$ collapses into state $\|0\rangle$ with probability α^2 and $\|1\rangle$ with probability β^2.
	$\begin{pmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{pmatrix}$	The **Controlled NOT (CNOT) gate** applies a NOT gate $\oplus$ to the bottom qubit if the control qubit $\bullet$ is $\|1\rangle$. If the control qubit is in superposition, this gate can be responsible for phenomena such as entangled states and phase kickback.
	$\begin{pmatrix}1 & 0 & 0 & 0 & 0 & 0\\0 & 1 & 0 & 0 & 0 & 0\\0 & 0 & 1 & 0 & 0 & 0\\0 & 0 & 0 & 1 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 1\\0 & 0 & 0 & 0 & 1 & 0\end{pmatrix}$	The **Toffoli** applies a NOT gate $\oplus$ to the bottom qubit if both control qubits $\bullet$ $\bullet$ are $\|1\rangle$. If either control qubits are in superposition, this gate can be responsible for phenomena such as entangled states and phase kickback.

B Code

The code used for computing the results found in this dissertation are given below. It is a Jupyter Notebook converted to LaTeX using `jupyter nbconvert`. The first code block imports required packages.

```
[1]: from qiskit import *
     from qiskit.visualization import plot_histogram
     import numpy as np
     from matplotlib import pyplot as plt
     from scipy import stats
     import matplotlib.patches as mpatches
```

B.1 Quantum Adder

A full adder is a classical gate that takes in 3 bits, and outputs their sum as a two-bit word. Its truth table is given below:

a	b	c	out
0	0	0	00
0	0	1	01
0	1	0	01
1	0	0	01
0	1	1	10
1	0	1	10
1	1	0	10
1	1	1	11

We can build a similair circuit on a quantum computer as follows:

```
[2]: def full_adder():
         qr = QuantumRegister(3, 'q')
         anc = QuantumRegister(2, 'ancilla')
         qc = QuantumCircuit(qr, anc)

         qc.cx(qr[0], anc[0])
         qc.cx(qr[1], anc[0])
         qc.cx(qr[2], anc[0])
         qc.ccx(qr[0], qr[1], anc[1])
         qc.ccx(qr[0], qr[2], anc[1])
         qc.ccx(qr[1], qr[2], anc[1])

         qc.name = "ADDER"
         return qc


     FULLADD = full_adder().to_gate()
```

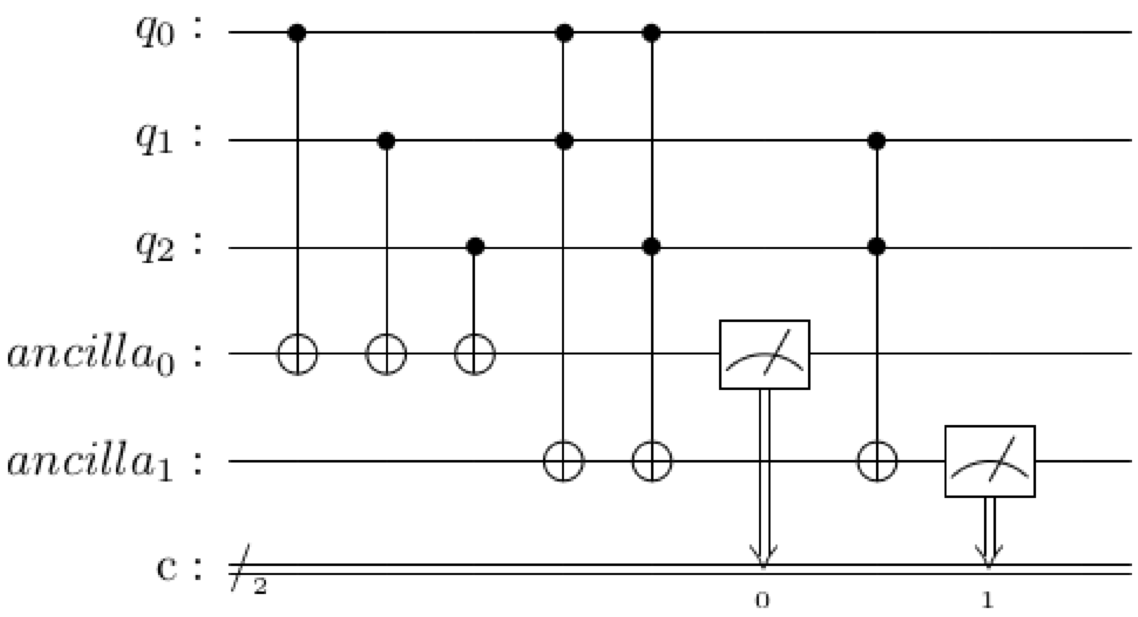
To ensure we're getting the answers we expect, we can test the gate with all possible inputs, in addition to visualising the quantum full adder circuit.

```
[35]: for inp1 in ['1', '0']:
          for inp2 in ['1', '0']:
              for inp3 in ['1', '0']:
                  # Construct the circuit
```

```
            qr = QuantumRegister(3, 'q')
            anc = QuantumRegister(2, 'ancilla')
            cr = ClassicalRegister(2, 'c')
            qc = QuantumCircuit(qr, anc, cr)

            # Qubit setup
            qc.reset(anc)
            if inp1 == '1':
                qc.x(qr[0])
            if inp2 == '1':
                qc.x(qr[1])
            if inp3 == '1':
                qc.x(qr[2])

            # Add full adder
            qc.append(full_adder().to_gate(), [*qr, *anc])

            # Measure the answer
            qc.measure(anc[0], cr[0])
            qc.measure(anc[1], cr[1])

            # Run the program on a simulator
            backend = Aer.get_backend('aer_simulator')
            qc_transpiled = transpile(qc, backend)
            job = backend.run(qc_transpiled, shots=1, memory=True)
            output = job.result().get_memory()[0]

            print('Adder with inputs', inp1, inp2, inp3, 'gives output', output)
qc_transpiled.draw(output='latex')
```

```
Adder with inputs 1 1 1 gives output 11
Adder with inputs 1 1 0 gives output 10
Adder with inputs 1 0 1 gives output 10
Adder with inputs 1 0 0 gives output 01
Adder with inputs 0 1 1 gives output 10
Adder with inputs 0 1 0 gives output 01
Adder with inputs 0 0 1 gives output 01
Adder with inputs 0 0 0 gives output 00
```

[35]:



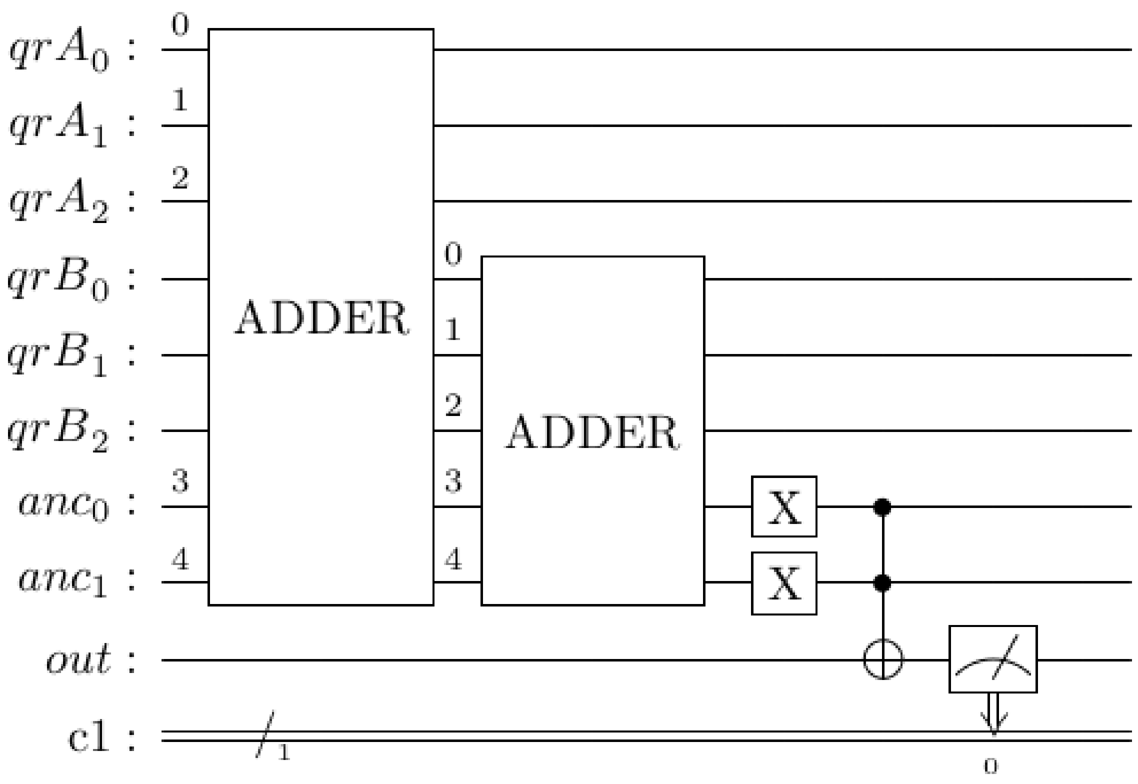
B.2 Equality Check

In order to verify the equality of two sums, we can use the `full_adder` gate on the 6 qubits involved, and use a boolean `AND` on the two results to check for equality. This would involve more qubits and gates than necessary however , and we can get away with using the same ancilla (output) qubits for both adders.

The full adder uses controlled `NOT` gates on the output qubits. This means that if we put the output of one adder into the ancilla qubits of another, the ancilla qubits will cancel and equal |00> if and only if both adders got the same result. The resulting circuit for checking equality of two 3-sums is displayed below:

[34]:
```
qrA = QuantumRegister(3, 'qrA')
qrB = QuantumRegister(3, 'qrB')
anc = QuantumRegister(2, 'anc')
out = QuantumRegister(1, 'out')
cr = ClassicalRegister(1)
qc = QuantumCircuit(qrA, qrB, anc, out, cr)

qc.append(full_adder().to_gate(), [*qrA, *anc])
qc.append(full_adder().to_gate(), [*qrB, *anc])
qc.x(anc)
qc.ccx(*anc, out)
qc.measure(out, cr)
qc.draw(output='latex')
```

[34]:



B.3 Oracles

We can use this idea to build an oracle for the problem at hand.

Indices are assigned to the 3 by 3 square as follows:

0	1	2
3	4	5
6	7	8

In order for the sum of each row and column to be equal, we need to check that:
[012] = [345] = [678] = [036] = [147] = [258]
It's easy to see that it's sufficient to check for 5 equalities:

- [012] = [345]
- [345] = [678]
- [678] = [036]
- [036] = [147]
- [147] = [258]

where [abc] corresponds to the sum of the cells at the indices a, b, c.
We can therefore build on the equality algorithm to build the full oracle

[5]:
```
def oracle():
    """
    20 qubit oracle
    """
    qr = QuantumRegister(9, 'q')
    anc = QuantumRegister(10, 'c')
    out = QuantumRegister(1, 'out')
    qc = QuantumCircuit(qr, anc, out, name='Oracle')

    # 012 = 345
    qc.append(FULLADD, [qr[0], qr[1], qr[2], anc[0], anc[1]])
    qc.append(FULLADD, [qr[3], qr[4], qr[5], anc[0], anc[1]])

    # 345 = 678
    qc.append(FULLADD, [qr[3], qr[4], qr[5], anc[2], anc[3]])
    qc.append(FULLADD, [qr[6], qr[7], qr[8], anc[2], anc[3]])

    # 678 = 036
    qc.append(FULLADD, [qr[6], qr[7], qr[8], anc[4], anc[5]])
    qc.append(FULLADD, [qr[0], qr[3], qr[6], anc[4], anc[5]])

    # 036 = 147
    qc.append(FULLADD, [qr[0], qr[3], qr[6], anc[6], anc[7]])
    qc.append(FULLADD, [qr[1], qr[4], qr[7], anc[6], anc[7]])

    # 147 = 258
    qc.append(FULLADD, [qr[1], qr[4], qr[7], anc[8], anc[9]])
    qc.append(FULLADD, [qr[2], qr[5], qr[8], anc[8], anc[9]])

    # flip output bit if all clauses are satisfied
    for a in anc:
        qc.x(a)
    qc.mct(anc, out)

    # uncomputation
    for a in anc:
        qc.x(a)

    qc.append(FULLADD, [qr[2], qr[5], qr[8], anc[8], anc[9]])
    qc.append(FULLADD, [qr[1], qr[4], qr[7], anc[8], anc[9]])
    qc.append(FULLADD, [qr[1], qr[4], qr[7], anc[6], anc[7]])
    qc.append(FULLADD, [qr[0], qr[3], qr[6], anc[6], anc[7]])
    qc.append(FULLADD, [qr[0], qr[3], qr[6], anc[4], anc[5]])
    qc.append(FULLADD, [qr[6], qr[7], qr[8], anc[4], anc[5]])
    qc.append(FULLADD, [qr[6], qr[7], qr[8], anc[2], anc[3]])
    qc.append(FULLADD, [qr[3], qr[4], qr[5], anc[2], anc[3]])
    qc.append(FULLADD, [qr[3], qr[4], qr[5], anc[0], anc[1]])
    qc.append(FULLADD, [qr[0], qr[1], qr[2], anc[0], anc[1]])

    return qc
```

```
[6]: def smaller_oracle():
         """
         17 qubit oracle
         """
         qr = QuantumRegister(9, 'q')
         anc = QuantumRegister(2, 'anc')
```

```
cl = QuantumRegister(5, 'cl')
out = QuantumRegister(1, 'out')
qc = QuantumCircuit(qr, anc, cl, out, name='Oracle')

def compute_clauses():
    # 012 = 345
    qc.append(FULLADD, [qr[0], qr[1], qr[2], anc[0], anc[1]])
    qc.append(FULLADD, [qr[3], qr[4], qr[5], anc[0], anc[1]])
    qc.x(anc)
    qc.ccx(anc[0], anc[1], cl[0])
    qc.x(anc)
    qc.append(FULLADD, [qr[3], qr[4], qr[5], anc[0], anc[1]])
    qc.append(FULLADD, [qr[0], qr[1], qr[2], anc[0], anc[1]])

    # 678 = 036
    qc.append(FULLADD, [qr[3], qr[4], qr[5], anc[0], anc[1]])
    qc.append(FULLADD, [qr[6], qr[7], qr[8], anc[0], anc[1]])
    qc.x(anc)
    qc.ccx(anc[0], anc[1], cl[1])
    qc.x(anc)
    qc.append(FULLADD, [qr[6], qr[7], qr[8], anc[0], anc[1]])
    qc.append(FULLADD, [qr[3], qr[4], qr[5], anc[0], anc[1]])

    # 147 = 258
    qc.append(FULLADD, [qr[6], qr[7], qr[8], anc[0], anc[1]])
    qc.append(FULLADD, [qr[0], qr[3], qr[6], anc[0], anc[1]])
    qc.x(anc)
    qc.ccx(anc[0], anc[1], cl[2])
    qc.x(anc)
    qc.append(FULLADD, [qr[0], qr[3], qr[6], anc[0], anc[1]])
    qc.append(FULLADD, [qr[6], qr[7], qr[8], anc[0], anc[1]])

    # 012/345 = 678/036
    qc.append(FULLADD, [qr[0], qr[3], qr[6], anc[0], anc[1]])
    qc.append(FULLADD, [qr[1], qr[4], qr[7], anc[0], anc[1]])
    qc.x(anc)
    qc.ccx(anc[0], anc[1], cl[3])
    qc.x(anc)
    qc.append(FULLADD, [qr[1], qr[4], qr[7], anc[0], anc[1]])
    qc.append(FULLADD, [qr[0], qr[3], qr[6], anc[0], anc[1]])

    # 678/036 = 147/258
    qc.append(FULLADD, [qr[1], qr[4], qr[7], anc[0], anc[1]])
    qc.append(FULLADD, [qr[2], qr[5], qr[8], anc[0], anc[1]])
    qc.x(anc)
    qc.ccx(anc[0], anc[1], cl[4])
    qc.x(anc)
    qc.append(FULLADD, [qr[2], qr[5], qr[8], anc[0], anc[1]])
    qc.append(FULLADD, [qr[1], qr[4], qr[7], anc[0], anc[1]])


compute_clauses()
qc.mct(cl, out)
```

```
    # uncomputation
    compute_clauses()

    return qc
```

[7]:
```
def smaller_oracle_opti():
    """
    17 qubit oracle, with optimized uncomputation
    """
    # maybe only the first x is required on the anc qubits
    qr = QuantumRegister(9, 'q')
    anc = QuantumRegister(2, 'anc')
    cl = QuantumRegister(5, 'cl')
    out = QuantumRegister(1, 'out')
    qc = QuantumCircuit(qr, anc, cl, out, name='Oracle')

    def compute_clauses(qc, qr, anc, cl):
        qc.x(anc)

        qc.append(FULLADD, [qr[0], qr[1], qr[2], anc[0], anc[1]])
        qc.append(FULLADD, [qr[3], qr[4], qr[5], anc[0], anc[1]])
        # 012 = 345
        qc.ccx(anc[0], anc[1], cl[0])

        # remove 012
        qc.append(FULLADD, [qr[0], qr[1], qr[2], anc[0], anc[1]])
        # add 678
        qc.append(FULLADD, [qr[6], qr[7], qr[8], anc[0], anc[1]])
        # 345 = 678
        qc.ccx(anc[0], anc[1], cl[1])

        qc.append(FULLADD, [qr[3], qr[4], qr[5], anc[0], anc[1]])
        qc.append(FULLADD, [qr[0], qr[3], qr[6], anc[0], anc[1]])

        # 678 = 036
        qc.ccx(anc[0], anc[1], cl[2])

        qc.append(FULLADD, [qr[6], qr[7], qr[8], anc[0], anc[1]])
        qc.append(FULLADD, [qr[1], qr[4], qr[7], anc[0], anc[1]])

        # 036 = 147
        qc.ccx(anc[0], anc[1], cl[3])

        qc.append(FULLADD, [qr[0], qr[3], qr[6], anc[0], anc[1]])
        qc.append(FULLADD, [qr[2], qr[5], qr[8], anc[0], anc[1]])

        # 147 = 258
        qc.ccx(anc[0], anc[1], cl[4])

        qc.append(FULLADD, [qr[2], qr[5], qr[8], anc[0], anc[1]])
        qc.append(FULLADD, [qr[1], qr[4], qr[7], anc[0], anc[1]])
        qc.x(anc)
    compute_clauses(qc, qr, anc, cl)
    qc.mct(cl, out)

    # uncomputation
    compute_clauses(qc, qr, anc, cl)

    return qc
```

B.4 Diffuser and Grover circuit

[8]:
```
# Qiskit implementation of diffuser circuit from
# https://learn.qiskit.org/course/ch-algorithms/grovers-algorithm#3qubit-implementation
# (Accessed 3 Feb 2023)

def diffuser(nqubits):
    qc = QuantumCircuit(nqubits)
    # Apply transformation |s> -> |00..0> (H-gates)
    for qubit in range(nqubits):
        qc.h(qubit)

    # Refleciton around |0>
    for qubit in range(nqubits):
        qc.z(qubit)

    # Apply transformation |00..0> -> |11..1> (X-gates)
    for qubit in range(nqubits):
        qc.x(qubit)
    # Do multi-controlled-Z gate
    qc.h(nqubits-1)
    qc.mct(list(range(nqubits-1)), nqubits-1)  # multi-controlled-toffoli
    qc.h(nqubits-1)
    # Apply transformation |11..1> -> |00..0>
    for qubit in range(nqubits):
        qc.x(qubit)

    # Apply transformation |00..0> -> |s>
    for qubit in range(nqubits):
        qc.h(qubit)

    U_s = qc.to_gate()
    U_s.name = "U$_s$"
    return U_s
```

[9]:
```
def grovers_algorithm(oracle, n_qubits, iters):
    """
    Return a Grover circuit given:
        - an oracle function,
        - the number of qubits it requires
        - the number of oracle+diffusion iterations to do
    """
    qr = QuantumRegister(n_qubits)
    cr = ClassicalRegister(9, 'c')
    qc = QuantumCircuit(qr, cr)

    # Initialize |s>
    qc.h(qr)

    # Initialize oracle qubit in state |-> for phase kickback
    qc.initialize([1, -1]/np.sqrt(2), qr[-1])
    qc.barrier()

    for _ in range(iters):
        qc.append(oracle(), qr)
        qc.append(diffuser(9), qr[:9])

    qc.measure(qr[:9], cr)
    return qc
```

Before running the algorithms, we introduce a helper function that can take in the result of an experiment and classically verify the measured state for correctness.

[10]:
```
import itertools


def split_correct_counts(counts, normalize=False, hex_keys=False):
    """
    Given a `result` object of a Grover's algorithm run, split the
    measured states into correct and incorrect states
    """
    inputs = list(itertools.product([0, 1], repeat=9))
    inputs.reverse()
    correct_counts = {}
    incorrect_counts = {}

    total_count = sum(counts.values())

    for input in inputs:
        key = ''.join([str(x) for x in input])
        if hex_keys:
            key = hex(int(key, 2))

        try:
            count = counts[key]
        except KeyError:
            count = 0

        if normalize:
            count /= total_count

        # classical verification
        if sum(input[0:3]) == sum(input[3:6]) == sum(input[6:9]) == sum(input[::3]) ==␣
↪sum(input[1::3]) == sum(input[2::3]):
            correct_counts[key] = count
        else:
            incorrect_counts[key] = count
    c = list(correct_counts.values())
    i = list(incorrect_counts.values())
    return c, i
```

Transpile the three qubits for the availble basis gates on a handful of IBM's systems.

[11]:
```
first = oracle()
second = smaller_oracle()
third = smaller_oracle_opti()

print("| Quantum Processor | 20 qubit | 17 qubit | 17 qubit, opti |")
print("|--------------------|----------|----------|----------------|")

# https://quantum-computing.ibm.com/services/resources
ibm_washington = ('IBM Eagle r1', ['cx', 'id', 'rz', 'sx', 'x'])
ibm_sherbrook = ('IBM Eagle r3', ['ecr', 'id', 'rz', 'sx', 'x'])
ibm_prague = ('IBM Egret r1', ['cz', 'id', 'rz', 'sx', 'x'])
for name, basis_gates in [ibm_washington, ibm_sherbrook, ibm_prague]:
    first = transpile(first, basis_gates=basis_gates)
    second = transpile(second, basis_gates=basis_gates)
    third = transpile(third, basis_gates=basis_gates)

    print(f'| {name:17} | {first.size():>8} | {second.size():>8} | {third.size():>14}␣
↪|')
```

```
| Quantum Processor | 20 qubit | 17 qubit | 17 qubit, opti |
|--------------------|----------|----------|----------------|
| IBM Eagle r1      |     7041 |     2124 |           1434 |
| IBM Eagle r3      |    16411 |     4862 |           3232 |
| IBM Egret r1      |    65243 |    18750 |          12416 |
```

B.5 Grover's algorithm

We decide to use the final iteration of the above oracle for the rest of this analysis.

[12]:
```
def grovers_algorithm_opti(n):
    qr = QuantumRegister(9, 'q')
    anc = QuantumRegister(2, 'anc')
    cl = QuantumRegister(5, 'cl')
    out = QuantumRegister(1, 'out')
    cr = ClassicalRegister(9, 'c')
    qc = QuantumCircuit(qr, anc, cl, out, cr)

    # Initialize |s>
    qc.h(qr)

    # Initialize 'out' in state |->
    qc.initialize([1, -1]/np.sqrt(2), out)
    qc.barrier()

    for _ in range(n):
        qc.append(smaller_oracle_opti(), [*qr, *anc, *cl, out])
        qc.append(diffuser(9), qr)
    qc.measure(qr, cr)
    return qc
```

We now run experiments on IBM's cloud simulators. There are a few backend options to choose from:

- *simulator_ statevector* simulates a quantum circuit by computing the wavefunction of the qubit's statevector as gates and instructions are applied. Supports general noise modeling.
- *simulator_ mps* can simulate a higher number of qubits and is generally faster especially if there is only weak entanglement, but can't do noise modell 1ing
- **_ stabilizer* types can do a lot more qubits, but are quite limited in the gates they can use and will not work with our circuit
- *ibmq_ qasm_ simulator* is a general-purpose simulator and selects the simulation method based on input circuits and parameters.

[15]:
```
# API token required on first call to `IBMQ.load_account()`
# IBMQ.save_account("YOUR-API-TOKEN")
IBMQ.load_account()
IBMQ.providers()
provider = IBMQ.get_provider(hub='ibm-q')

# list available backends
provider.backends(simulator=True)
```

[15]:
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_statevector') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_mps') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_extended_stabilizer') from IBMQ(hub='ibm-q',
group='open', project='main')>,
 <IBMQSimulator('simulator_stabilizer') from IBMQ(hub='ibm-q', group='open',
project='main')>]
```

[19]:
```
backend = provider.get_backend('ibmq_qasm_simulator')
backend.status()
```

[19]:
```
<qiskit.providers.models.backendstatus.BackendStatus at 0x286d5fb77c0>

name: ibmq_qasm_simulator
version: 0.1.547, pending jobs: 0
status: active
```

In this first experiment we run Grover's algorithm for 1 to 10 iterations and plot the results.

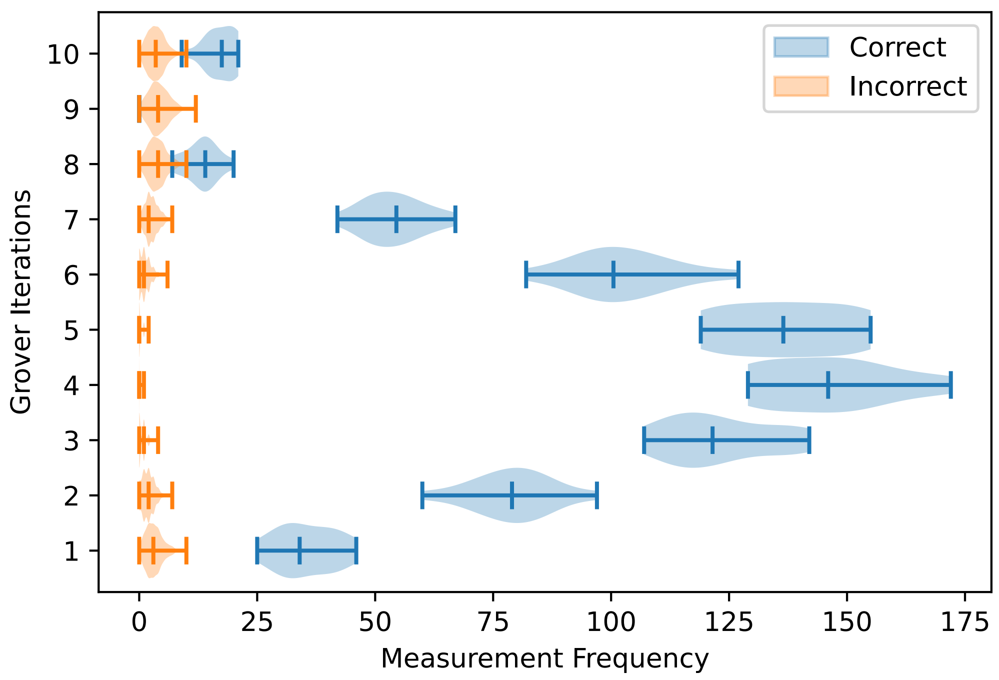
[20]:
```
from qiskit.providers.fake_provider import FakeWashington
from qiskit_aer import AerSimulator

# Simulate IBM's 127-qubits sytem, ibm_washington
fake_backend = FakeWashington()

# For local simulation
#backend = AerSimulator.from_backend(fake_backend)
```

```
experiments = []
for i in range(1, 11):
    qc = grovers_algorithm_opti(i)
    qct = transpile(qc, fake_backend)
    experiments.append(qct)

# Uncomment line to below to execute job on cloud
#job = execute(experiments, backend, shots=2048)
```

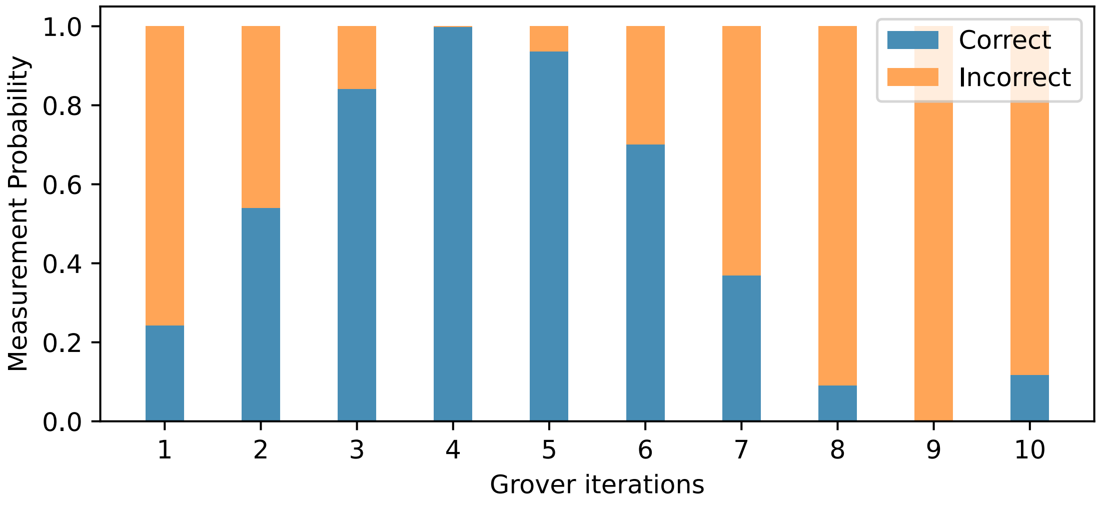
[22]:
```
# Alternatively, retrieve previously run jobs
job = backend.retrieve_job('643036e26fb1e57b32c623f0')
```

Plotting Results

[23]:
```
results = job.result().results
correct = []
incorrect = []
for result in results:
    counts = result.data.counts

    # Split counts into correct/incorrect solutions by verifying them classically
    c, i = split_correct_counts(counts, normalize=False, hex_keys=True)
    correct.append(c)
    incorrect.append(i)

# Draw violin plots
violinlabels = []
def add_label(violin, label):
    color = violin["bodies"][0].get_facecolor().flatten()
    violinlabels.append((mpatches.Patch(color=color), label))

f = plt.figure()
f.set_figwidth(6)
v1 = plt.violinplot(correct, vert=False, showmedians=True, widths=1)
v2 = plt.violinplot(incorrect, vert=False, showmedians=True, widths=1)

add_label(v1, "Correct")
add_label(v2, "Incorrect")

ylabels = [str(x + 1) for x in range(len(results))]
plt.yticks(np.arange(1, len(ylabels) + 1), labels=ylabels)
plt.ylim(0.25, len(ylabels) + 0.75)
plt.xlabel('Measurement Frequency')
plt.ylabel('Grover Iterations')
plt.legend(*zip(*violinlabels), loc=1)
plt.show()
```



[63]:
```
xticks = list(range(1, 11))
x_axis = np.arange(len(xticks))
total_c = []
total_i = []
for c in correct:
    total_c.append(sum(c))
for i in incorrect:
    total_i.append(sum(i))

shots = total_c[0] + total_i[0]
yticks = [f"{x:.1f}" for x in np.linspace(0, 1, 6)]
y_axis = np.linspace(0, shots, 6)

plt.figure(figsize=(7, 3))
plt.bar(x_axis, total_c, 0.4, label='Correct', color='#478DB5')
plt.bar(x_axis, total_i, 0.4, label='Incorrect', color='#FFA557', bottom=total_c)
plt.xticks(x_axis, xticks)
plt.yticks(y_axis, yticks)
plt.xlabel("Grover iterations")
plt.ylabel("Measurement Probability")
plt.legend()
plt.show()


accuracy = [sum(correct[i]) / (sum(incorrect[i]) + sum(correct[i]))
            for i in range(len(results))]
print(
    f"Max accuracy occurs at {accuracy.index(max(accuracy)) + 1} iterations␣
↪({max(accuracy)*100:.4f}%)")
```



```
Max accuracy occurs at 4 iterations (99.7559%)
```

B.6 Quantum Counting

First create a controlled version of the grover iteration

[26]:
```
def grover_iteration():
    """
    A single grover iteration (oracle + diffuser)
    """
    qr = QuantumRegister(9, 'q')
    anc = QuantumRegister(2, 'anc')
    cl = QuantumRegister(5, 'cl')
    out = QuantumRegister(1, 'out')
    qc = QuantumCircuit(qr, anc, cl, out)

    qc.append(smaller_oracle_opti().to_gate(), [*qr, *anc, *cl, out])
    qc.append(diffuser(9), qr)

    return qc.to_gate()


cgrit = grover_iteration()
cgrit.label = "Grover"
cgrit = cgrit.control()
```

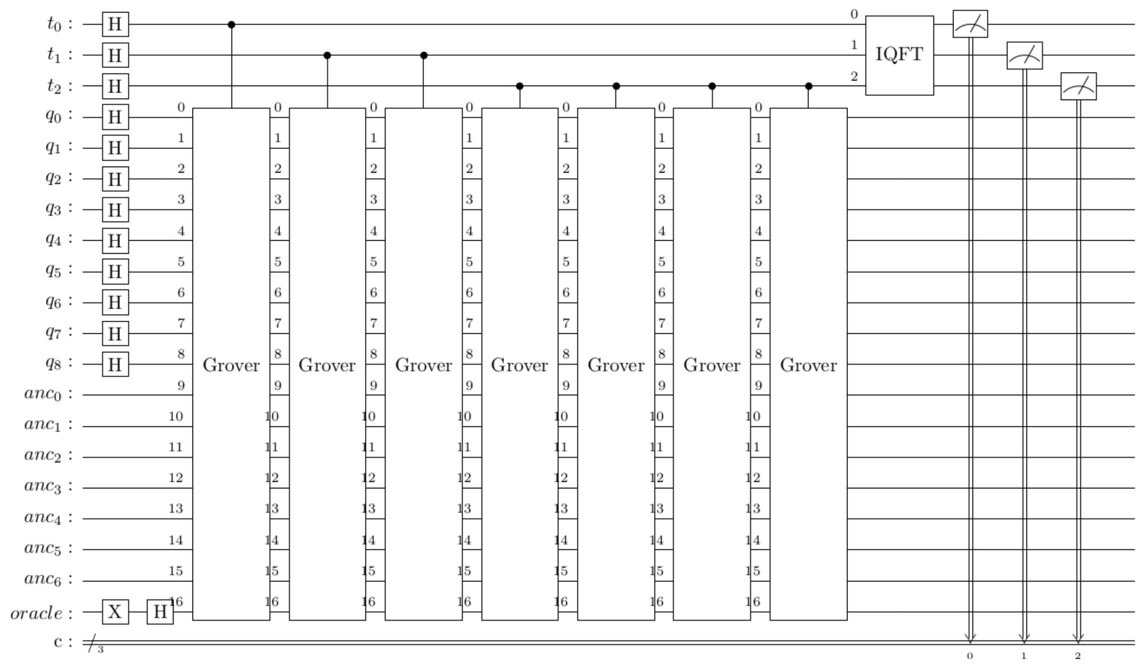
Now we can implement quantum counting, making use of qiskit's built in QFT operator.

[31]:
```
from qiskit.circuit.library import QFT

def quantum_counting(t):
    """
    Returns quantum counting circuit for t counting qubits
    """
    counting_reg = QuantumRegister(t, 't')
    searching_reg = QuantumRegister(9, 'q')
    ancilla_reg = QuantumRegister(7, 'anc')
    oracle_reg = QuantumRegister(1, 'oracle')
    classical_reg = ClassicalRegister(t, 'c')

    n = 17  # no. of oracle searching qubits
    qc = QuantumCircuit(counting_reg, searching_reg, ancilla_reg, oracle_reg,␣
↪classical_reg)

    inv_qft = QFT(num_qubits=t, inverse=True)
    inv_qft.label = "QFT†"

    # Init counting qubits and searching qubits to |+>
    qc.h(counting_reg)
    qc.h(searching_reg)

    # Init oracle qubit to |->
    qc.x(oracle_reg)
    qc.h(oracle_reg)


    # Begin controlled grover iterations
    iterations = 1
    for qubit in range(t):
        for i in range(iterations):
            qc.append(cgrit, [qubit, *searching_reg, *ancilla_reg, *oracle_reg])
        iterations *= 2

    # Do inverse QFT on counting qubits
    qc.append(inv_qft, counting_reg)

    # Measure
    qc.measure(counting_reg, classical_reg)
    return qc
```

As an example we show the circuit for t=3

[33]:
```
qc = quantum_counting(3)
qc.draw(output='latex')
```

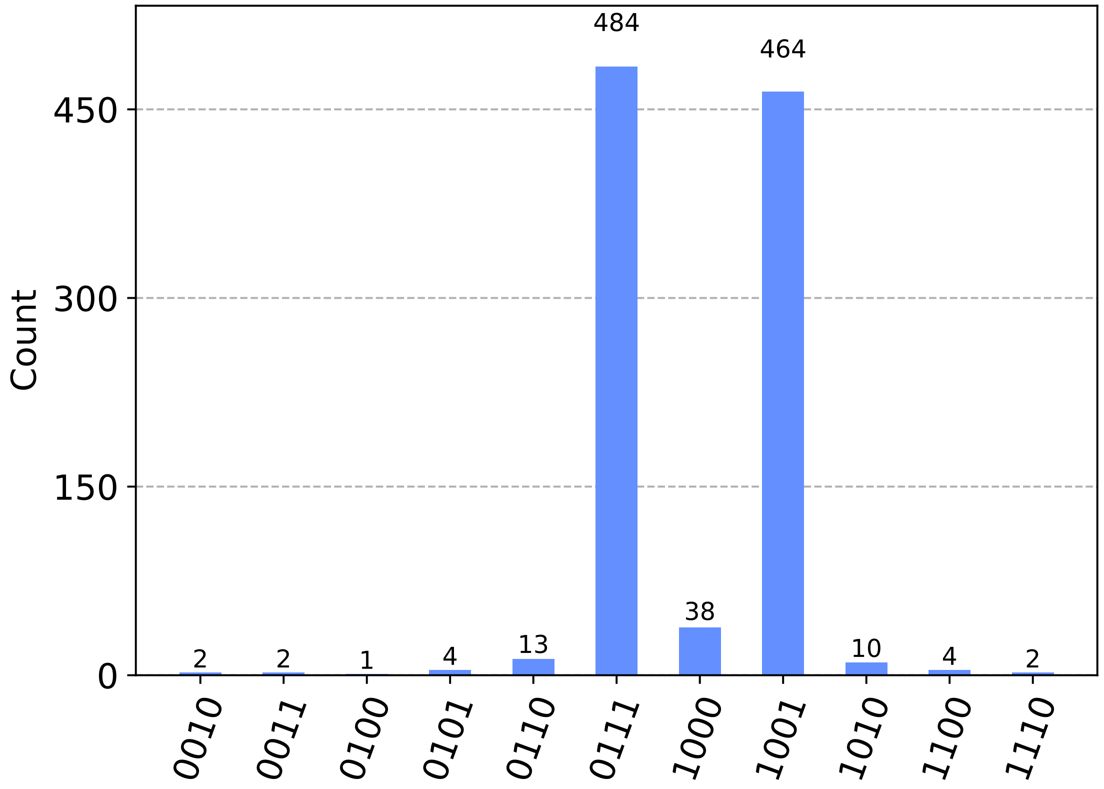
[33]:



For small t we can simulate this locally

[64]:
```
# Construct circuit
qc = quantum_counting(4)

# Run on local simulator
aer_sim = Aer.get_backend('aer_simulator')
transpiled_qc = transpile(qc, aer_sim)
qobj = assemble(transpiled_qc)
job = aer_sim.run(qobj)

# Plot results
hist = job.result().get_counts()
plot_histogram(hist)
```

[64]:



[ ]:
```
from qiskit.providers.fake_provider import FakeWashington

fake_backend = FakeWashington()
backend = provider.get_backend('ibmq_qasm_simulator')

qc = quantum_counting(7)
qct = transpile(qc, fake_backend)

# Uncomment line to below to execute job on cloud
# job = execute(qct, backend, backend_properties=fake_backend.properties())
```

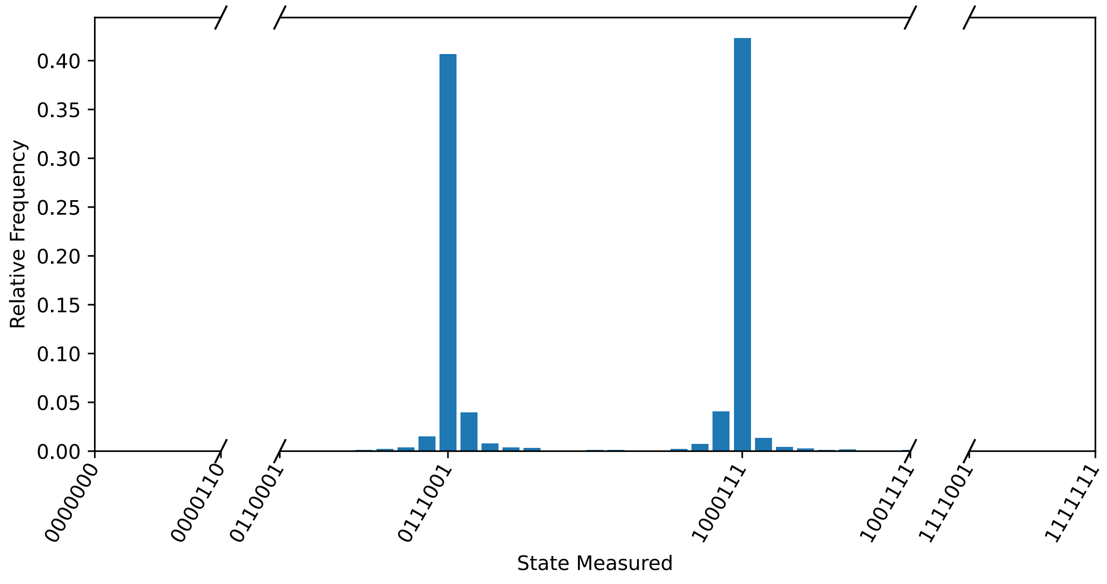
[37]:
```
# Quantum counting result for t = 7, w/ noise model
backend = provider.get_backend('ibmq_qasm_simulator')
job = backend.retrieve_job('63f77a72747fb9e2dbca428f')
hist = job.result().get_counts()
```

Plotting results for t = 7

[48]:
```
for i in range(2**7):
    b = f"{i:>07b}"
    if b not in hist.keys():
        hist[b] = 0

measured = sorted(list(hist.keys()), key=lambda x: int(x, 2))
total = sum(hist.values())
```

```
counts = [hist[i]/total for i in measured]
states = list(range(2**7))
mid = states[len(states)//2]

# adjust middle window
cutoff = 6
midwidth = 30

# decimal values of bar peaks
peak1 = counts.index(max(counts))
temp = counts[:peak1] + counts[peak1+1:]
peak2 = counts.index(max(temp))
peaks = (peak2, peak1)

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=True, gridspec_kw={'width_ratios':␣
 ↪[cutoff, midwidth, cutoff]}, figsize=(9, 4))
fig.subplots_adjust(hspace=0.05)  # adjust space between axes

ax1.bar(states, counts)
ax2.bar(states, counts)
ax3.bar(states, counts)
ax1.set_xticks(
    [0, cutoff],
    [measured[0], measured[cutoff]],
    rotation=60, rotation_mode='anchor', ha='right', va='center')
ax2.set_xticks(
    [mid - midwidth//2, peaks[0], peaks[1], mid + midwidth//2],
    [measured[mid - midwidth//2], measured[peaks[0]], measured[peaks[1]], measured[mid␣
 ↪+ midwidth//2]],
    rotation=60, rotation_mode ='anchor', ha='right', va='center')
ax3.set_xticks(
    [2**7-1-cutoff, 2**7-1],
    [measured[2**7-1-cutoff], measured[2**7-1]],
    rotation=60, rotation_mode='anchor', ha='right', va='center')

# hide spines between ax and ax2
ax1.spines.right.set_visible(False)
ax2.spines.left.set_visible(False)
ax2.spines.right.set_visible(False)
ax3.spines.left.set_visible(False)
ax2.yaxis.set_visible(False)
ax3.yaxis.set_visible(False)

ax1.set_xlim(0, cutoff)
ax2.set_xlim(mid - midwidth//2, mid + midwidth//2)
ax3.set_xlim(2**7 - cutoff - 1, 2**7 - 1)

# cut-out slanted lines
d = .5  # proportion of vertical to horizontal extent of the slanted line
kwargs = dict(marker=[(-d, -1), (d, 1)], markersize=12,
              linestyle="none", color='k', mec='k', mew=1, clip_on=False)
ax1.plot([1, 1], [0, 1], transform=ax1.transAxes, **kwargs)
ax2.plot([0, 0, 1, 1], [0, 1, 0, 1], transform=ax2.transAxes, **kwargs)
```

```
ax3.plot([0, 0], [0, 1], transform=ax3.transAxes, **kwargs)

ax1.set_ylabel('Relative Frequency')
ax2.set_xlabel('State Measured')
plt.show()
```



B.7 Calculating M from θ

A common way to create the diffusion operator U_s is to actually implement $-U_s$, as we have done. In Grover search this is a global phase that can be ignored, but in quantum counting this will actually give tell us how many states are *not* solutions. To fix this, we simply calculate NM.

[49]:
```
import math


# Some code from qiskit.org
# https://learn.qiskit.org/course/ch-algorithms/quantum-counting#finding_m
# (Accessed 25 Feb 2023)
# The mathematical justification is derived in Nielsen and Chuang
t = 7
measured = int(max(hist, key=hist.get), 2)
print(f"Registered Output = {measured}")
theta = (measured/(2**t))*math.pi*2
print(f"Theta = {theta:.5f}")
N = 2**9 # there are 2**9 permutations of 9 elements so that's how many solutions␣
 ↪there are
M = N * (math.sin(theta/2)**2)
print(f"Estimated No. of Solutions = {(N-M):.1f}")
m = t - 1 # Upper bound: Will be less than this
err = (math.sqrt(2*M*N) + N/(2**(m+1)))*(2**(-m))
print(f"Error < {err:.2f}")
print(f"Num iterations suggested: {math.ceil(np.pi / 4 * np.sqrt(N / (N-M)))}")
```

```
Registered Output = 71
Theta = 3.48520
Estimated No. of Solutions = 15.0
Error < 11.21
Num iterations suggested: 5
```

B.8 Finding M through brute force

[66]:
```
M = 0
for i in range(2**9):
    b = [int(x) for x in f"{i:09b}"]

    # classic verification
    if sum(b[:3]) == sum(b[3:6]) == sum(b[6:]) == sum(b[::3]) == sum(b[1::3]) ==␣
 ↪sum(b[2::3]):
        M+=1
print("M =", M)
```

```
M = 14
```

[67]:
```
import qiskit.tools.jupyter
%qiskit_version_table
```

```
Version Information
Qiskit Software                                        Version
qiskit-terra                                            0.21.2
qiskit-aer                                              0.11.0
qiskit-ibmq-provider                                    0.19.2
qiskit                                                  0.38.0

System information
Python version                                           3.9.7
Python compiler                        MSC v.1929 64 bit (AMD64)
Python build                                tags/v3.9.7:1016ef3
OS                                                     Windows
CPUs                                                         6
Memory (Gb)                                              15.36
```