THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

Ancilla-Free Preparation of Quantum Error-Correcting Codes

Miles Gorman

Under the supervision of
Dr. Terry Farrelly

A thesis submitted to the University of Queensland
in partial fulfilment of the degree of Bachelor of Adv. Science (Honours)
School of Mathematics and Physics
September 2023

Typeset in LaTeX 2_ε.

The work presented in this Thesis is, to the best of my knowledge and belief original, except as acknowledged in the text, and has not been submitted either in whole or in part, for a degree at this or any other university.

Miles Gorman

Abstract

Noise stands in the way of realizing useful large-scale quantum computers. Consequently, it is likely that any such quantum devices will require quantum error correction, which builds tolerance against noise by encoding quantum information across redundant qubits.

In this thesis, we aimed to lower the overhead of encoding arbitrary stabilizer codes. To address this, we designed an algorithm that we hypothesized would return ancilla-free encoding circuits with reduced two-qubit gate counts and reduced depths - as compared to the literature's most prominent method: the Cleve-Gottesman algorithm [1]. We tested our algorithm by running numerical simulations, comparing the overhead of the encoding circuits we produced to those found through the Cleve-Gottesman method. For appraising the algorithms, we used a random selection of codes that each contained a number of physical qubits within the range of what can be expected in near term quantum devices (up to $n \sim 100$ physical qubits). We found that, on average, our algorithm reduced these codes' encoding circuits' two-qubit gate counts by $\sim 41\%$, and their depths by $\sim 87\%$. We also saw that the reductions in two-qubit gate count and depth tended to grow with the size of the code used. Additionally, our algorithm provided these benefits at no additional asymptotic runtime cost as compared to the Cleve-Gottesman algorithm, because our method too, generally, has a time complexity of $\mathcal{O}(m^2 n)$ where m is the number of stabilizers in the code.

Acknowledgements

I would like to thank my supervisor, Dr. Terry Farrelly, for working with me on various research projects over the last 2 years. I would like to extend further gratitude to Terry for introducing me to quantum error correction and helping me work towards a career in research.

I would also like to thank my parents, James Gorman and Danielle Piat, and my grandparents, Michel and Judith Piat, for - among everything else they have done for me - supporting my education, and encouraging me to pursue my goals.

Contents

List of Figures

List of Tables

1 Introduction

By exploiting quantum phenomena, quantum computers promise particular algorithms with significantly lower time complexity than their best-known classical counterparts [2]. However, because quantum computers are necessarily coupled to a relatively noisy environment and are built from imperfect parts, even with passive hardware improvements, logical errors - that render the computation useless - are inevitable. As such, it is important to build robustness against noise by leveraging *quantum error correction*: active methods for reducing logical errors.

Quantum error correction has three core components: *encoding*, *syndrome extraction*, and *decoding*. Encoding adds redundancy to some given quantum information, so that - through syndrome extraction - the encoded state can be checked for errors, and - through decoding - any such errors can be identified for correction. In this thesis, we aim to reduce the overhead of encoding.

The thesis will be decomposed into several chapters. The first, Chapter 2, presents a review directed towards introducing quantum error correction. This review is then extended by Chapter 3 which completes the necessary background on our project. Most importantly, it presents the literature on ancilla-free, low overhead encoding. Then, in Chapter 4, we present our work. In particular, we detail our research's motivations, elaborate on our aim, propose our method for reducing the overhead of encoding circuits, present our hypotheses, and show and discuss our results. Finally, in Chapter 5, we provide concluding remarks and ideas for future work. Also note that in Appendix A we include information to supplement our introductory review of quantum error correction, in Appendix B we provide the pseudocode for our work, and lastly in Appendix C we summarize the quantum gates and circuit elements relevant to this thesis.

2

Literature Review of Quantum Error Correction

In this chapter, we will present a review of quantum error correction. Before proceeding with the review though, it is advised that a reader new to quantum information first look through the introduction to classical error correction provided in Appendix A.1, and the introduction to quantum noise provided in Appendix A.2 - as hopefully they provide a more intuitive foundation from which quantum error correction can be explained.

2.1 Introduction

Quantum error correction must be a novel technique. As is detailed in Appendix A.2, quantum information can suffer two types of errors: *bit-flip* and *phase-flip* errors. This is in contrast to classical information, which is only prone to bit-flips - because there is no classical phase. So, with this addition of phase-flip errors, classical error correction - an already well studied field - is rendered insufficient for quantum systems. Moreover, phase-flip errors are not the only additional issue that needs to be addressed in quantum error correction. Quantum error correction also needs to consider the *no-cloning theorem* which states that there is no unitary that allows the operation $|\psi\rangle \otimes |0\rangle \rightarrow |\psi\rangle \otimes |\psi\rangle$ for an unknown arbitrary state, $|\psi\rangle$. That is, the no-cloning theorem disallows unknown quantum states from being copied, which further prevents the implementation of classical error correction for quantum systems. For an example of this, see in Appendix A.1 how the classical repetition code effectively requires copies of the original information. Moreover, additional difficulties stem from the fact that measurements collapse quantum states - destroying the information. That is, retrieving information on the error/s is potentially problematic [2]. On the other hand, what quantum error correction has working for it is that the continuum of possible errors can be decomposed into just the discrete set of Pauli errors. And so, to correct any error,

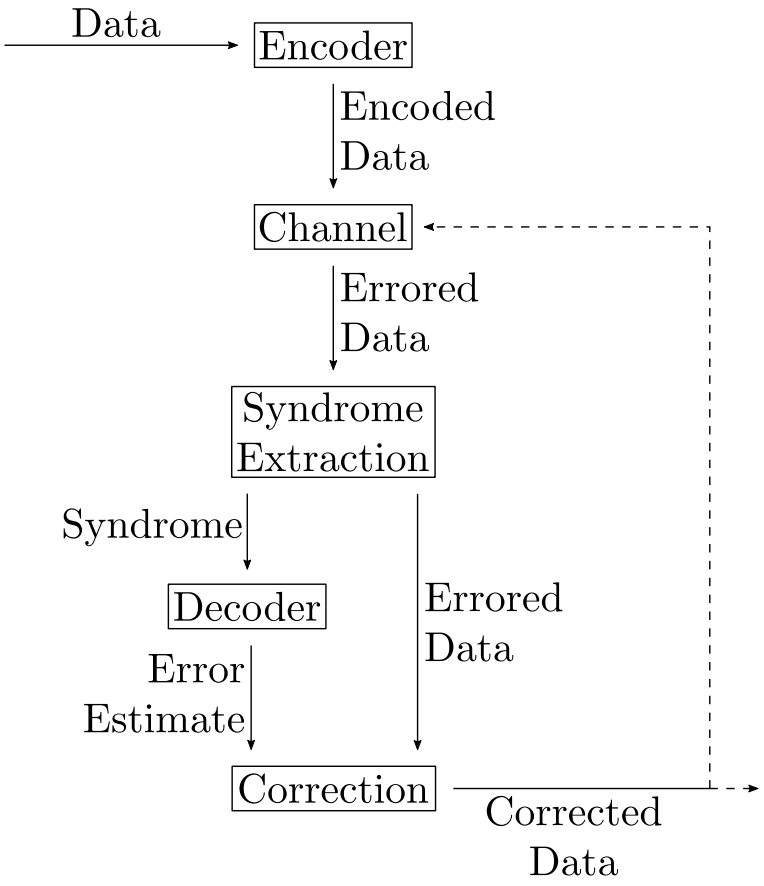


FIGURE 2.1: Flowchart summarizing the general steps of a quantum error correction procedure. The arrows represent the flow of information in time, and the boxes represent 'cogs' in the quantum error correction 'machine'. The two dashed arrows symbolise a choice between two paths for the data at the end of an error correction cycle, as explained in the main text.

quantum error correction needs only to independently correct Pauli-X and Pauli-Z errors [2, 12]. This is an important result in quantum error correction called the *digitization of errors*, which is expounded upon in Appendix A.2.

2.2 The General Quantum Error Correction Process

We present the components of quantum error correction in Fig. 2.1. As is the case with classical error correction, the core mechanism behind quantum error correction is to add redundancy, the idea being that then localized errors would not necessarily corrupt the logical information. With reference to the flowchart in Fig. 2.1, this process of spreading information across additional qubits is the first step in quantum error correction, and is done by the *encoder*. The encoded information is then communicated through a noisy channel, where it can accumulate errors. The now errored information undergoes syndrome extraction, which, in the ideal case, leaves the errored information unchanged and provides

the *syndrome*: classical information on the error/s that occurred. The syndrome is then passed to the *decoder*: a classical algorithm that returns an estimate of the Pauli error/s likely to have occurred for a given syndrome - detailed in Appendix A.9. Next, the decoder returns its guess for the Pauli error/s as classical information, which is then fed to the *correction* step, which attempts to undo the error/s on the quantum data. It is worth noting that Pauli operators square to the identity, so to undo a Pauli error, the qubit with an error must just again have the same Pauli applied to it. After corrections, the data is then either outputted to the quantum computer's user, or returned to the channel to again sit in memory or have further computations preformed. That is, as per Section A.2, because coherence times are far shorter than computation times, error correction must be performed in regular cycles - to reduce the probability of accumulating more errors than are correctable [18].

To summarize the error correction process using a medical analogy, if an error is a disease, the encoder builds a person's immunity, the channel represents the possibility of contracting a disease, the syndrome bits of an error are the disease's symptoms, the decoder takes the place of a medical doctor searching for a likely diagnosis, and the correction made is the medication prescribed.

2.3 Stabilizer Codes

A formalism for quantum error correction is given by *stabilizer codes*. Breaking this term up, we have *stabilizer checks* - which are a set of operations that extract information on errors without corrupting the encoded state, and *quantum codes* - which define how k logical qubits of information are encoded across n physical qubits. In this section, we will discuss stabilizer codes more generally, and, in the following section, we will show an example of a simple quantum error-correcting code.

In the stabilizer formalism, an error-correcting code is defined by its generating set of stabilizer operators. These stabilizer operators have three main properties [2, 12, 28, 29]:

1. They must be made from tensor products of Pauli matrices. That is, $S_j \in G^{\otimes n}$ where S_j is a stabilizer for an n-qubit code, and $G = \{\pm I, \pm X, \pm Z, \pm Y\}$ is a subset of the single-qubit Pauli group, where I is the identity, and X, Z, and Y are the Pauli operators - which are described in Tab. A.2. This constraint is in place because performing stabilizer checks made from Pauli group elements is what collapses noise onto probabilistic Pauli errors and allows for the correction of a continuum of errors [19].

2. Stabilizers must *stabilize* all of a code's logical states. Meaning that the code's logical states are $+1$ eigenvectors of all the code's stabilizers. That is, $S_i|\psi\rangle_L = (+1)|\psi\rangle_L = |\psi\rangle_L$ for all logical states of a code, $|\psi\rangle_L$, and all stabilizers of a code, S_i. This constraint is important because it allows for the retrieval of information on errors without changing the logical state; the encoded information. Note also that this constraint implies that a code's stabilizers define its logical states - and vice versa.

3. Finally, all stabilizers must commute with one another; $[S_i, S_j] = 0$ for any choice of a code's stabilizers S_i and S_j. For this reason, stabilizer checks can be performed independent of order.

An important corollary that follows from properly 2 is that corrections need only be right up to a stabilizer, as, say we have an arbitrary errored state $E_j|\psi\rangle_L$ where E_j is a Pauli error, and we make the correction $E'_j \equiv S_i E_j$, then the corrected state will be

$$E'_j E_j|\psi\rangle_L = S_i E_j E_j|\psi\rangle_L = S_i|\psi\rangle_L = |\psi\rangle_L. \tag{2.1}$$

Note that if $E'_j \equiv E_j S_i$, then the same argument applies, potentially up to a different global phase - from seeing how Paulis commute. Another important observation from these properties is that products of stabilizers are also stabilizers. This follows from the fact that if S_i and S_j are stabilizers, then a new operator $S_{ij} \equiv S_i S_j$ also satisfies the three core properties of stabilizers: (1) products of Paulis and Paulis, and products of Paulis and identities are still in the Pauli group; (2) S_{ij} satisfies

$$S_{ij}|\psi\rangle_L = S_i S_j|\psi\rangle_L = (+1)S_i|\psi\rangle_L = (+1)|\psi\rangle_L; \tag{2.2}$$

and (3) another stabilizer of the code S_k must commute with S_i and S_j and so

$$S_{ij}S_k = S_i S_j S_k = S_i S_k S_j = S_k S_i S_j = S_k S_{ij} \tag{2.3}$$

implying $[S_{ij}, S_k] = 0$. So, we can see that stabilizers form an abelian group as: each stabilizer is its own inverse because Pauli operators square to the identity; an identity of appropriate dimension clearly stabilizers any state; matrix multiplication is associative; and stabilizers must commute. In keeping with this and the previous property that products of stabilizers are also stabilizers, a code's stabilizers are often presented as a minimum generating set of a stabilizer group. This is done for brevity and because additional stabilizer checks would only return redundant information on the error - while also adding noise and time to the syndrome extraction and decoding procedures.

Stabilizers are useful in quantum error correction because each stabilizer of a code can be designed to *anti-commute* with different Pauli errors. Equivalently stated, errored states are -1 eigenvectors of particular stabilizers; $S_i E_j|\psi\rangle_L = (-1)E_j S_i|\psi\rangle_L = (-1)E_j|\psi\rangle_L$ where an errored state is defined as a logical state with some error E_j, and where S_i is one of potentially multiple stabilizers that anti-commute with the Pauli error E_j. Measuring whether an errored state is a $+1$ or -1 eigenstate of its stabilizers reveals information on the errors.

2.4 Example: The Quantum Repetition Code

To put the quantum error correction procedure all together, we will present the simplest example of a quantum code, the 3-qubit repetition code, which encodes one qubit of information, called a *logical qubit*, across three qubits, called *physical qubits*. A quantum circuit for this code is provided in Fig. 2.2. *Quantum circuits* represent quantum computations, and are composed of operations called quantum gates - all the relevant gates are provided in Appendix C. Quantum circuits are read from left to right. That is, on the far left are

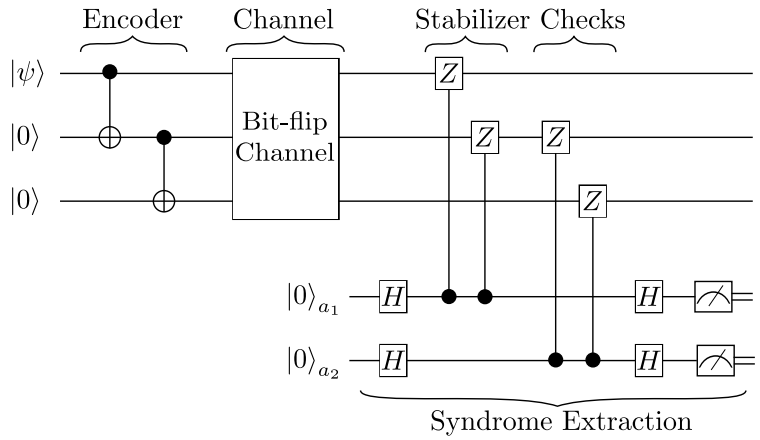


Figure 2.2: The 3-qubit repetition code's encoding and syndrome extraction circuit. All the gates are defined in Appendix C and the circuit is explained in the text from Eq. 2.4 to Eq. 2.12.

the initial states of each qubit in the system, often called the *register*, and next to them are horizontal lines representing the path of that qubit in the circuit. Note that qubits are generally initialized into the $|0\rangle$ state because it makes sense to set a convention so that algorithms have a consistent starting point, and because consistently preparing a 'classical state' is oftentimes simpler - one does not need to worry about relative phases or decoherence as a quantum state can be measured in the z-basis to collapse it onto either $|0\rangle$ or $|1\rangle$ and then the qubit can be discarded or flipped if it was measured to be in the $|1\rangle$ state [31, 32]. In this example, the initial state is

$$|\psi\rangle \otimes |00\rangle = \alpha|000\rangle + \beta|100\rangle \tag{2.4}$$

where $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ is the qubit of logical information we wish to encode. Recalling Fig. 2.2, the first two gates are $CNOT$s which act as follows: if the *controlled* qubit, represented by the black dot, is in the state $|0\rangle$ nothing is done, but if instead it is in the state $|1\rangle$, a Pauli-X is applied to the *target* qubit, represented with the white circle. That is $CNOT_{i,j} = |0\rangle_i\langle 0|_i \otimes I_j + |1\rangle_i\langle 1|_i \otimes X_j$ where we have used the notation that O_i represents the operator O acting on qubit i - with implicit identities on the other qubits in the space. From this, after applying the gates $CNOT_{1,2}$ and then $CNOT_{2,3}$, the initial state is encoded into

$$|\psi\rangle_L \equiv \alpha|000\rangle + \beta|111\rangle \tag{2.5}$$

which is the 3-qubit repetition code's general logical state, also often called a *codestate*. Here we can also define the code's *codewords*: encoded 'classical' states. In this code we have one qubit of information, so two codewords. Specifically the codewords are $|0\rangle \to |0\rangle_L \equiv |000\rangle$ and $|1\rangle \to |1\rangle_L \equiv |111\rangle$ where the subscript L denotes a logical state.

A code that encodes k logical qubits of information across n physical qubits has $m = n-k$ independent elements in its minimum generating set of stabilizers. This comes from the fact that the vector space of a code's codewords is of dimension 2^k [2]. Accordingly, we expect the 3-qubit repetition code to have $m = 3 - 1 = 2$ stabilizer generators. To find them, see that

the codestate, Eq. 2.5, is stabilized by the operators $\{I_1I_2I_3, Z_1Z_2, Z_2Z_3, Z_1Z_3\}$. Considering that $(Z_1Z_2)(Z_2Z_3) = Z_1Z_3$ and $(Z_2Z_3)^2 = I_1I_2I_3$, it is evident that $\langle Z_1Z_2, Z_2Z_3\rangle$ is a choice for the stabilizers' generating group.

Returning to the circuit in Fig. 2.2, consider the channel - which represents some noisy process. For simplicity, in this example we are considering just the bit-flip channel, meaning it only causes bit-flip errors. Say the state experiences an X_3 error - which we would not know at this stage in practice. Then the system's state would be

$$X_3(\alpha|000\rangle + \beta|111\rangle) = \alpha|001\rangle + \beta|110\rangle. \tag{2.6}$$

The next step in the quantum error correction procedure is to retrieve information on the error, which is done using stabilizer checks, also often referred to as *parity checks* - a term borrowed from an analogous process in classical error correction. A general syndrome extraction scheme requires that, for each stabilizer generator, we initialize an *ancilla qubit*; a qubit auxiliary to the main computation - that is usually prepared in the $|0\rangle$ state, has some interaction with the main encoded block of qubits, and is then measured and discarded or reset. These ancilla qubits have a Hadamard applied to them so that they are each mapped to the state $|+\rangle \equiv \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. After this, each of the code's stabilizer generators, each S, is controlled on a separate ancilla, a, implementing gates of the form

$$CS_{a,d} = |0\rangle_a\langle 0|_a \otimes I_d + |1\rangle_a\langle 1|_a \otimes S_d \tag{2.7}$$

where d represents the data qubits. The result of each of these stabilizer checks is stored on the relative phase of the respective ancilla's two computational basis states. Lastly, another Hadamard is applied to each of the ancillas to find these relative phases, and a measurement is performed on the ancillas to retrieve the syndrome.

Following Fig. 2.2, we can see this syndrome extraction process in action. After initializing the ancillas, the system's state is $(\alpha|001\rangle + \beta|110\rangle) \otimes |00\rangle$ and so after the Hadamard gates H_4 and H_5, which, again, individually map $|0\rangle \leftrightarrow |+\rangle \equiv \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle \leftrightarrow |-\rangle \equiv \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, the system's state is

$$\begin{aligned}\frac{1}{2}(&|00100\rangle + |00101\rangle + |00110\rangle + |00111\rangle \\ &+ |11000\rangle + |11001\rangle + |11010\rangle + |11011\rangle).\end{aligned} \tag{2.8}$$

Then we can check one stabilizer by performing $CZ_{4,1}$ and $CZ_{4,2}$ which leaves the system's state unchanged, and the other by applying $CZ_{5,2}$ and $CZ_{5,3}$, which changes the system's state to

$$\begin{aligned}\frac{1}{2}((&|00100\rangle - |00101\rangle + |00110\rangle - |00111\rangle \\ &+ |11000\rangle - |11001\rangle + |11010\rangle - |11011\rangle))\end{aligned} \tag{2.9}$$

which can be rewritten as

Syndrome	More Likely Error	Less Likely Error
'00'	$I_1I_2I_3$	$X_1X_2X_3$
'10'	$X_1I_2I_3$	$I_1X_2X_3$
'11'	$I_1X_2I_3$	$X_1I_2X_3$
'01'	$I_1I_2X_3$	$X_1X_2I_3$

Table 2.1: Look-up-table decoder for the 3-qubit repetition code. This table groups every possible error with its corresponding syndrome. Because syndromes are degenerate, each syndrome here has two possible corresponding errors. The 'More Likely Error', which the decoder returns, is the minimum weight error - which will clearly be the best estimate for a correction when the error rate is below 50%. This table was found by looking at how each of the code's stabilizers commutes and anti-commutes with each possible X-type error. Recall that this (toy) example bars the possibility of Z-type errors, as Fig. 2.2 only has a bit-flip channel.

$$\frac{1}{2}(\alpha|001\rangle + \beta|110\rangle)(|00\rangle - |01\rangle + |10\rangle - |11\rangle). \tag{2.10}$$

So after the final Hadamards H_4 and H_5 are applied, the state is

$$(\alpha|001\rangle + \beta|110\rangle)|01\rangle \tag{2.11}$$

implying that the measurements at the end of the circuit will return a syndrome of '01'. The syndrome is then fed to a decoder. We have provided a look-up-table decoder in Tab. 2.1, which can be used to estimate the error to be X_3. So by then applying the correction X_3 to the state, we are left with

$$\alpha|000\rangle + \beta|111\rangle \tag{2.12}$$

which is back in the codespace, and is in fact the original codestate from Eq. 2.5, meaning the correction worked and the original information was recovered. Note that in the above equation the ancillas were omitted, because they are irrelevant after being measured. Also note that because applications of quantum error correction will generally involve encoding superposition states, one cannot confirm if the correction was indeed correct without collapsing the state and losing the information.

An important point to make is that, as per the look-up-table decoder provided in Tab. 2.1, an error with this same syndrome would be X_1X_2 which is a non-equivalent error because it does not equal X_3 up to a stabilizer - as talked about in the properties of stabilizers above. That is, if the error X_1X_2 had occurred instead of the error X_3, and we did not know this was the case, the decoder would still be given the syndrome '01' and, not being able to distinguish the errors based on this information, it would recommend the correction X_3, because for an error rate $p_X < 0.5$ a single error is more likely than two errors. But this would make the error worse as the error is now of weight 3 instead of weight 2, where the

weight is the count of how many Paulis are in the operator. In fact this weight 3 error would map the original state as follows

$$\alpha|000\rangle + \beta|111\rangle \rightarrow \alpha|111\rangle + \beta|000\rangle \tag{2.13}$$

which is equivalent to applying an X to the data before encoding. And this is no coincidence. Error correction always acts to return an errored state back to the codespace. In these cases, where the error correction procedure fails to return the original information, we have a *logical error*. As such, the rate at which an error correction protocol fails is its *logical error rate*. To be clear, this is in contrast to the term *physical error* rate (or often just error rate), which refers to the probability of an error of a particular type acting on each physical qubit. As an aside, a specific type of logical error rate, called the threshold - discussed in Appendix A.3, is often used to benchmark an error correction procedure's performance.

Finally, for a discussion of why error correction cannot correct all errors with certainty, and to see examples of codes that can correct arbitrary single-qubit errors, see Appendix A.4 and Appendix A.6.

2.5 Logical Operators and Distance

An important point to consider in error correction is that when performing computations on an encoded state, one must not de-encode, apply an operation, and then re-encode because this procedure would directly expose the information to logical errors - as opposed to just physical errors. To overcome this difficulty, we apply *(encoded) logical operations*, which act on the encoded state as if it were not encoded. More precisely, an operator O applied to an un-encoded state is mapped as follows after encoding:

$$O|\psi\rangle \rightarrow O_L|\psi\rangle_L \tag{2.14}$$

where $|\psi\rangle_L$ is the logical state and the logical operator is denoted by O_L, or often also as $\bar{O}$. For example, in the 3-qubit repetition code, a choice for the logical X would be $X_L = X_1X_2X_3$ because it maps the codeword $|0\rangle_L \equiv |000\rangle$ to the other codeword $|1\rangle_L \equiv |111\rangle$ and vice versa, acting as though an X was applied to the logical information. Note that each logical qubit in a code will have its own set of logical operations.

As a corollary, the X and Z logical operators have a close relationship to a code's *distance*, and give insight into the origin of the equation for how many errors a code can correct, Eq. A.9. Firstly, the precise definition of distance is the minimum weight of an error that maps between codewords. Which, because errors can be digitized, is equivalent to the weight of the minimum weight logical Pauli operator - X_L, Z_L, or $Y_L \propto X_LZ_L$. To vary the weight of the logical operators they are multiplied by stabilizers. This follows from the fact that, if O_L is a logical operator for a logical state $|\psi\rangle_L$, then from the properties of stabilizers given earlier, $O_L|\psi\rangle_L = O_LS_i|\psi\rangle_L = O'_L|\psi\rangle_L$ where S_i is a stabilizer and $O'_L \equiv O_LS_i$ is a new logical operator that must be equivalent to O_L. That is, the definition of distance is

$$d \equiv \min_{S\in\mathcal{S},L\in\mathcal{L}} \mathrm{wt}(LS). \tag{2.15}$$

where the wt( ) function returns the weight of an operator, $\mathcal{S}$ is the complete set of a code's stabilizers, and $\mathcal{L}$ is the set of a choice for the code's logical Pauli operators: X_L, Z_L, and $Y_L \propto X_L Z_L$. This definition of distance makes apparent the motivation behind the relationship between distance and the number of errors a code can correct, which is given in Eq. A.9. That is, because error correction ideally returns errored states to their closest logical state, where closest is defined to be the minimum number of Pauli operations apart, one can only hope to correct up to just under half the minimum number of operations between logical states. Or equivalently, one can only hope to correct up to just under half of the distance, which corroborates Eq. A.9.

As an example, recall that a choice for the 3-qubit repetition code's logical X would be $X_L = X_1 X_2 X_3$, whose action is shown in Eq. 2.13. Also, with reference to the encoded state, given in Eq. 2.5, observe that a choice for its logical Z would be $Z_L = Z_1$. Noting that these are the minimum weight logical operators of the code, we can use the definition of distance in Eq. 2.15 and how distance relates to the number of correctable errors in Eq. A.9 to again see that the 3-qubit repetition code can correct up to one arbitrary X error, and no Z errors. We can also confirm that the code has distance 1, corroborating that it is unable to correct an arbitrary error.

Continuing on the topic of the properties of logical operator's, it is worth noting that logical Xs and logical Zs commute with all the code's stabilizers, because they map one logical state to another - and stabilizers stabilize all logical states. Another important point is that a logical X_j and logical Z_j (acting on the same logical qubit, j) anti-commute with one another, because they anti-commute before encoding - and so they must anti-commute after encoding, as per Eq. 2.14.

2.6 Parity Check Matrix

With many avenues of quantum error correction research relying on classical computers, it is often important to represent a quantum code's stabilizers digitally. While writing out stabilizers in their Pauli-operator representation, $X_1 Z_2 Z_3 X_4$ for instance, is easy to read, directly representing them in this way on a computer is unnatural - as compared to say a binary representation. This motivates the use of the *parity check matrix formalism*; a representation of stabilizers using a matrix consisting of binary elements - which was borrowed from classical error correction. A parity check matrix, denoted by H, is an $m \times 2n$ matrix, where m is the number of operators in the code's minimum generating set of stabilizers, and n is the number of physical qubits in the code. It is partitioned down the middle into two equal sized halves. The left side and right side represent the X- and Z-type part of the stabilizers respectively. Each row corresponds to a stabilizer in the code, and each column corresponds to a physical qubit in the code, where the $n + i$-th column - after the partition - represents the i-th qubit again, where i is an integer between 1 and n. In the matrix, 0s denote an identity on that qubit in the stabilizer, and 1s denote either an X or a Z, depending on which side of the partition the 1 is on. On qubits where an X and Z overlap, we have a Y operator.

As a toy example, consider the parity check matrix

$$H = \left(\begin{array}{cccccc|cccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) \tag{2.16}$$

which we can see represents a code that has $k = 1$ logical qubit, $n = 6$ physical qubits, and $m = 5$ stabilizers: X_1, X_2, Z_3, X_4Z_5, and Y_6.

We also note there is a common alternative way of representing the parity check matrix: using an $m \times n$ matrix constructed from adding the left side (first n columns) of the parity check matrix above to 2 times the right side (second n columns). That is, similarly to before, each row represents a stabilizer, each column represents a physical qubit, and Is are denoted by 0s. But, differently from before, Xs are denoted by 1s, Zs are denoted by 2s, and Ys are denoted by 3s. As an example, in this form, the parity check matrix from Eq. 2.16 is

$$H = \left(\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 \end{array}\right). \tag{2.17}$$

3

Preparing Quantum Error-Correcting Codes

In this chapter, we present the 'tools' that are outside the scope of a quantum error correction introductory review, but are nonetheless necessary to understand our work. Additionally, we will outline a method from the literature that is considered to be an effective way of preparing quantum error correcting codes - which we will later use to gauge the efficiency of our algorithm.

3.1 Stabilizer Propagation

As is discussed in the 'A Word on Fault-Tolerance' Section in Appendix A.8, errors can potentially propagate through a circuit - by mechanism of multi-qubit gates. While errors propagating in this way is an issue, an adaptation of this idea can be leveraged to better think about encoding. Consider that the $+1$ eigenstates of the Z and X operators, $|0\rangle$ and $|+\rangle$, are, by definition, stabilized by Z and X respectively. So, an initial state, say $|0\rangle_1 \otimes |0\rangle_2 \otimes \cdots \otimes |0\rangle_i \otimes |+\rangle_{i+1} \otimes |+\rangle_{i+2} \otimes \cdots \otimes |+\rangle_j$, is stabilized by Z_1, Z_2, ..., Z_i, X_{i+1}, X_{i+2}, ..., X_{j-1}, and X_j. Then, by considering the Heisenberg picture and seeing how these stabilizers propagate as they traverse the gates in a circuit, one can find the stabilizers of the state encoded by said circuit.

For example, with reference to how the X and Z operators propagate through Hadamards and $CNOT$s, the results of which are summarized in Fig. 3.1, it is easy to find the stabilizers of the 9-qubit Shor code given its encoding circuit. In particular, as shown in Fig. 3.2, because $|0\rangle_4$ is stabilized by Z_4, the encoded state will be stabilized by $X_1X_2X_3X_4X_5X_6$. By repeating this process for all of the qubits in the register, the entire set of stabilizers for the code can be found - which is much simpler than thinking in the Schrödinger picture and finding the arbitrary encoded state produced by a given encoding circuit, and then figuring out all of the non-equivalent operators that stabilize said state.

As an aside, a stabilizer cannot be defined for an unknown arbitrary data qubit, $|\psi\rangle$.

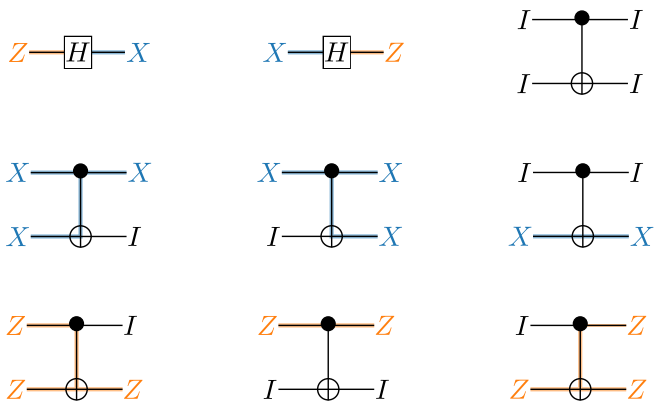


FIGURE 3.1: A summary of how Pauli-Xs and Pauli-Zs propagate through Hadamards and $CNOT$s. Note that these relationships can be derived using the Pauli matrices in Fig. A.2 and the matrix representations of these gates - which are provided in Appendix C. One such calculation is shown in Appendix A.8. Also note that these gates are their own inverses, so some of these diagrams are redundant - they can be read left to right or right to left.

However, from the definition of a logical operator, Eq. 2.14, seeing how a Pauli-X and Pauli-Z propagate from a data qubit, we can find the encoded X and Z logical operators respectively.

Also, observe that because Hadamards and $CNOT$s are self-inverse, by now adding to the end of the encoding circuit of the Shor code shown in Fig. 3.2, the gates of the encoding circuit but in reverse order, the state is de-encoded, and we are returned the initial register. For instance, in this procedure, the stabilizer $X_1X_2X_3X_4X_5X_6$ is returned to being the stabilizer Z_4 - consistent with the 4th qubit again being in the state $|0\rangle$ - as can be seen by reading Fig. 3.2 from right to left. So, given a code's stabilizers, if a sequence of self-inverse gates can be found that reduce each stabilizer down to a single Z - consistent with a $|0\rangle$ state, one reduces the encoded state down to the standard, easy to prepare register $|00\ldots0\rangle$. Then, as said before, this circuit reversed is the encoding circuit: the circuit that takes the data qubits and register to the desired logical state - which is part of the idea of our algorithm. That is, if we have the de-encoding circuit as $U_{\text{de-encode}} = U_1U_2\ldots U_T$ where $U_j \in \{H_\alpha, CNOT_{\alpha,\beta}\}$, then we can find the corresponding encoding circuit to be $U_{\text{encode}} = U_TU_{T-1}\ldots U_1$. Note that this is elaborated on in Section 4.2.

3.2 Literature on Ancilla-Free Encoding

Preparing quantum error correcting codes is important not just for the initial encoding step of error correction but also for tensor network decoding [59, 67, 78, 79], and Knill [57, 58] and Steane [56] syndrome extraction. With all of these uses benefiting from optimized encoding circuits, it is important to reduce the overhead of encoding as much as possible. There exist various algorithms for finding such circuits, including classical algorithms that produce circuits requiring no additional ancilla qubits [1], similar methods for qudit codes [95, 96], classical algorithms built for a *measurement-based quantum computing* model and hence use

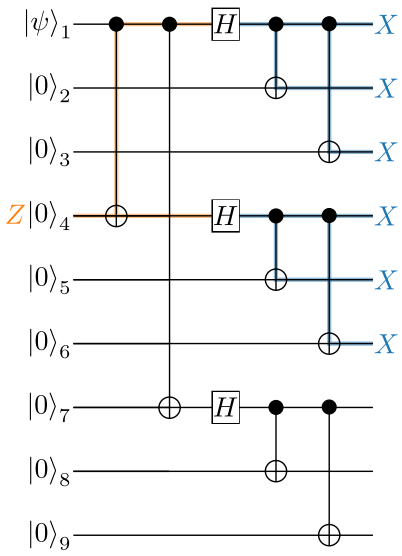


Figure 3.2: The 9-qubit Shor code's encoding circuit. By noticing that Z_4 stabilizes the initial state, and seeing how this Z_4 propagates through the encoding circuit, we can determine that the encoded state will be stabilized by $X_1X_2X_3X_4X_5X_6$.

ancillas [40, 86, 97], a method that measures stabilizers which also uses ancillas [12], and a variational quantum algorithm [94].

Because qubits on near-term quantum devices are noisy and scarce, we are most interested in studying the classical algorithms that find ancilla-free encoding circuits. So, in this section we will present the most prominent of these algorithms: the Cleve-Gottesman algorithm [1]. We will then discuss methods from the literature for reducing the produced circuits' *depths* [98] and two-qubit gate counts [99]. Note that a circuit's depth is the number of time steps it takes to execute said circuit, where each gate takes one unit of time. Hence depth quantifies a circuit's time overhead. Also note that the above optimizations only work for *Calderbank-Shor-Steane (CSS)* codes: a code where its set of stabilizer generators can be partitioned into two disjoint subsets, one of just X-type and the other of just Z-type [28, 29]. The surface code in Appendix A.7 is an example of a CSS code, as is the repetition code, and, by extension, the Shor code.

3.2.1 Cleve-Gottesman Algorithm

The Cleve-Gottesman algorithm is the literature's most prominent method for finding the ancilla-free encoding circuits of arbitrary stabilizer codes [1]. Note that this algorithm searches for the encoding circuit directly - unlike our method which will first look for the de-encoding circuit. Here we will present the steps of the Cleve-Gottesman algorithm [1, 28, 30] and then we will show a worked example of it applied to the 5-qubit code.

1. The algorithm is input with a given code's $m \times 2n$ parity check matrix, H, and its X logical operators.

2. Gaussian elimination is applied to the parity check matrix, H, to get it into row echelon form. That is, the parity check matrix is put into the form

$$H = \left(\begin{array}{cc|c} I_{r\times r} & A_{r\times(n-r)} & B_{r\times n} \\ 0_{(m-r)\times r} & 0_{(m-r)\times(n-r)} & C_{(m-r)\times n} \end{array} \right), \tag{3.1}$$

where $I_{p\times q}$ is the $p \times q$ identity; $0_{p\times q}$ is the $p \times q$ zeros matrix; and $A_{p\times q}$, $B_{p\times q}$, and $C_{p\times q}$ are some $p \times q$ binary matrices with well-defined forms, per [1, 28].

3. Use these stabilizers to row reduce the logical X operators, X_L, into the form

$$X_L = \left(\begin{array}{ccc|cc} 0_{k\times r} & D_{k\times(n-r-k)} & I_{k\times k} & E_{k\times r} & 0_{k\times(n-r)} \end{array} \right), \tag{3.2}$$

where D and E are binary matrices. As is shown in [28], this calculation of Eq. 3.2 uses the fact that stabilizers must commute, and requires the more precise form of the matrices constituting Eq. 3.1.

4. Find the initial gates needed to partially encode the pure Z-type stabilizers as follows. For all stabilizers from r on, indexed by $u \in \{r+1, r+2, \ldots, m\}$, and for all logical X operators in the code, indexed by $v \in \{1, 2, \ldots, k\}$, if the v-th logical operator has a non-identity operation on qubit u, apply a $CNOT$ targeting that physical qubit, u, and control the $CNOT$ on the data qubit $m + v$.

   For example, the 3-qubit repetition code, seen in Section 2.4 and Section 2.5, has the row reduced parity check matrix

$$H = \left(\begin{array}{ccc|ccc} 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{array} \right) \tag{3.3}$$

   and logical X operator

$$X_L = \left(\begin{array}{ccc|ccc} 1 & 1 & 1 & 0 & 0 & 0 \end{array} \right). \tag{3.4}$$

   With reference to the standard forms above, Eq. 3.1 and Eq. 3.2, see that here $r = 0$, $m = 2$, $v \in \{1\}$, and $u \in \{1, 2\}$. Hence, in this case, we need to apply the gates $CNOT_{3,2}CNOT_{3,1}$. Note that this is the complete encoding circuit in this example, as can be verified using Pauli propagation.

5. Find the remaining gates needed to encode by repeating the following for each row of the parity check matrix, H.

   5.1. If the first 1 in a given row corresponds to a Pauli-X, apply a Hadamard to that qubit/column; else if it corresponds to a Pauli-Z, skip the remaining steps; else if it corresponds to a Pauli-Y, apply SH to that qubit/column. Recall that all the relevant gates, including the S gate, are defined in Appendix C.

   5.2. Set the first 1 in the row to be a control for the following gate.

5.3. Set the remaining 1s in that row to be the targets of the gate, such that the target operation matches the operator on that physical qubit in the stabilizer.

As an example, consider the 5-qubit code, which has the stabilizers

$$\begin{aligned}\langle &XZZXI,\\ &IXZZX,\\ &XIXZZ,\\ &ZXIXZ\rangle,\end{aligned} \tag{3.5}$$

and hence has the $m \times 2n$ parity check matrix

$$H = \left(\begin{array}{ccccc|ccccc} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right). \tag{3.6}$$

Then, after applying Gaussian elimination to the parity check matrix, we get the stabilizer generators as

$$H = \left(\begin{array}{ccccc|ccccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \end{array}\right). \tag{3.7}$$

Following the procedure above, we can see that the algorithm calls for the following gates

$$\begin{aligned}&(CY_{4,5}\ \ CZ_{4,3}\ \ CZ_{4,1}\ \ S_4H_4)\\ &(CX_{3,5}\ \ CZ_{3,2}\ \ CZ_{3,1}\ \ H_3)\\ &(CX_{2,5}\ \ CZ_{2,4}\ \ CZ_{2,3}\ \ H_2)\\ &(CY_{1,5}\ \ CZ_{1,4}\ \ CZ_{1,2}\ \ S_1H_1).\end{aligned} \tag{3.8}$$

This encoding circuit produced by the Cleve-Gottesman method for the 5-qubit code is shown in Fig. 3.3. It is also important to note that the qubits with no gate controlled on them are the data qubits, hence, in this case, the 5th qubit is the data qubit. Lastly, see from Eq. 3.1 that here $r = 4$ and $m = 4$ (there are no pure Z-type stabilizers), so step 4 of the algorithm is trivial.

The Celve-Gottesman algorithm applies to all stabilizer codes, as proven in [1]. To see this, consider that the circuit encoding k logical qubits across n physical qubits will require an $m = n - k$ ancilla qubit register, $|00\ldots0\rangle$. Now, because $|0\rangle$ is the $+1$ eigenstate of the Z operator, the initial state can thought of as being stabilized by $Z_1, Z_2, \ldots, Z_m$. In the Heisenberg picture, these stabilizers propagate through the encoding circuit into the final stabilizers of the code - noting that $m = n - k$ is also the number of stabilizers in a code. And this is how we are going to show that the Cleve-Gottesman algorithm works.

In step 5 of the Cleve-Gottesman algorithm, each gate is controlled on a qubit that has either an X or Y operator acting on it in only one of the code's stabilizer generators. As such, to begin encoding said stabilizer, the corresponding initial Z stabilizer of the register

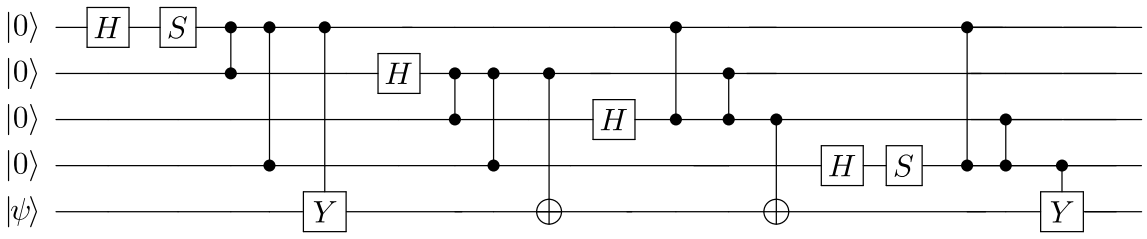


FIGURE 3.3: Encoding circuit for the 5-qubit code as produced by the Cleve-Gottesman algorithm. Here a classical single-qubit state, $|\psi\rangle$, is encoded. Observe that there are 4 Hadamards, 2 S gates, and 12 two-qubit gates: 8 CZs, 2 CYs, and 2 $CNOT$s. Also, see that the circuit has depth 10, assuming that successive single qubit gates can be done in one operation. Note that the data state, $|\psi\rangle$, must not be in a superposition - or else there is a phase issue in the encoded logical state. Lastly, we acknowledge that a similar figure is given in [30].

state (Z_1 or $Z_2 \dots$ or Z_m) must be rotated into this X or Y operator - which is done through applying a H or SH respectively. Then, because in step 5 the remaining operators of the stabilizer are controlled on this qubit, and because an X or Y 'activate' the control, the remaining operators of that stabilizer are propagated into existence, and said stabilizer is complete. For instance, in Eq. 3.8, the gates $CY_{1,5}$ $CZ_{1,4}$ $CZ_{1,2}$ S_1H_1 are used to encode the stabilizer $YZIYZ$ as these gates map

$$ZIIII \xrightarrow{S_1H_1} YIIII \xrightarrow{CZ_{1,2}} YZIII \xrightarrow{CZ_{1,4}} YZIZI \xrightarrow{CY_{1,5}} YZIZY. \tag{3.9}$$

Speaking generally again, this idea is then repeated for each of the code's stabilizer generators to complete the encoding process. Lastly, note that of the gates used, a Pauli-Z can only propagate from the target of a $CNOT$ or a CY. Hence, because the parity check matrix has been row reduced, Eq. 3.1, and the ancillas/controls correspond to the identity portion of this matrix, these gates can not target the ancilla/control qubits, and therefore the stabilizers of the register do not propagate before their Hadamard is applied. To see an example of this, consider how these register's Pauli-Z stabilizers propagate in Fig. 3.3. Also consider that the Z operators in an encoded stabilizer can propagate into an X or a Y through the single-qubit gates on the ancillas/controls. These can then go on to further propagate the already encoded stabilizers into a different form. However, from similar reasoning as before, these new stabilizers are unchanged up to a stabilizer, hence this presents no issues. We acknowledge that this may in general be difficult to see - so it is explained in the next paragraph.

Moving on, step 4 of the algorithm effectively deals with any pure Z-type stabilizers, and is the reason why pure Z-type stabilizers are not considered in step 5. Specifically, notice how step 4 applies to all stabilizers indexed by $u > r$, and, with reference to the row reduced parity check matrix, Eq. 3.1, see that pure Z-type stabilizes indeed begin from row $> r$. To see why step 4 in effect gives the pure Z-type stabilizers for free, we will follow the argument in [1, 28]. Observe that encoding a logical state $|b_1 b_2 \dots b_k\rangle$, where $b_j \in \mathbb{Z}_2$, can be done, up to normalization, by

$$X_{L,1}^{b_1} X_{L,2}^{b_2} \dots X_{L,k}^{b_k} (I + S_1)(I + S_2) \dots (I + S_m) |0\rangle^{\otimes n} \tag{3.10}$$

where $X_{L,j}$ is the logical X of the j-th logical qubit of the code, and S_j is the j-th stabilizer of the code. In particular, see here that there are two main steps: (1) a projection of the register onto the $+1$ eigenspace of the stabilizers, as $S_j(I+S_j) = (I+S_j)$ because stabilizers, S_j, are Pauli products; and (2) the application of logical operators that map the blank logical state $|0\rangle^{\otimes k}$ to the desired logical state $|b_1 b_2 \dots b_k\rangle$. Notice though that because stabilizers commute, the pure Z-type stabilizers can be commuted to the right and be trivially applied directly to the state $|0\rangle^{\otimes n}$. As such, any pure Z-type stabilizers can be neglected. Next, because logical operators commute with stabilizers, the logical operators can also be brought to the right. See that similarly, their Z components act trivially on the state and so their Z operators can also be neglected - noticing that in Eq. 3.2 the Zs in the logical X act on different qubits than the Xs, so indeed no Zs in the logical Xs will act on anything beside $|0\rangle$. As such, if one can apply the X components of each logical X operator, and then project this state onto the code space, one has prepared an encoded classical state. Recalling how Pauli's propagate, observe that the construction in step 4 encodes the X component of each logical X. Then as we have already shown, step 5 projects this state onto the code space. Hence a code state is prepared by this algorithm, as required.

As some final notes on the Cleve-Gottesman algorithm, the ancilla states' Z stabilizers will propagate through the $CNOT$s from step 4 onto the data qubits, and then will propagate from the data qubits up the subsequent gates to give the code's pure Z-type stabilizers - but this is not as clear as the equivalent argument above. Also, see that in the case of encoding logical $|00\dots0\rangle$, the gates from step 4 of the algorithm act trivially, and hence they can be skipped. Lastly, note that the partially encoded logical Xs from step 4 will not propagate into having more X operators after the later gates. This is because the row reduced parity check matrix, Eq. 3.1, implies that the controls of these two-qubit gates act on different qubits than the X operators in the logical Xs - as can be seen by comparing the identity portion of Eq. 3.1 with the corresponding qubits in Eq. 3.2.

As a final comment, the Cleve-Gottesman algorithm can run into issues regarding phase. Recall from Eq. 3.10 that the Cleve-Gottesman algorithm produces encoded classical states. Hence, it does not guarantee that these encoded classical states do not have a global phase. As such, when encoding superposition states, the Cleve-Gottesman method can leave incorrect relative phases between the encoded logical states. Fixing this phase takes additional gates, which can increase the noise and depth of the encoding circuits. However, we note that this is generally not of practical relevance because, in applications, one will generally want to prepare an encoded classical state anyway, and then perform logical operations - so that the computation is error corrected from the beginning.

3.2.2 Steane's Latin Rectangle Method

Steane's Latin Rectangle method is an add-on to the Clive-Gottesman algorithm that reduces the depth of CSS codes' encoding circuits [98, 99]. To see how it works, recall that CSS codes have just pure X-type and pure Z-type stabilizers. Thus a CSS code's parity check matrix can be separated into a pure X-type stabilizer parity check matrix, H_X, and a pure Z-type

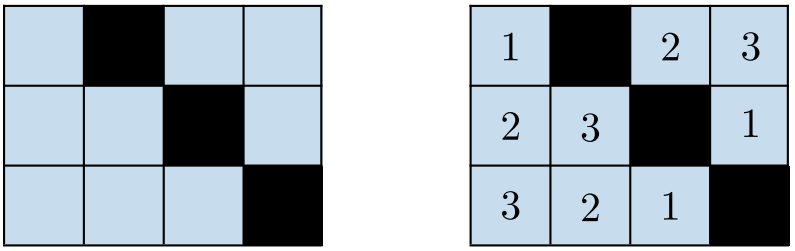


FIGURE 3.4: Steane's Latin Rectangle method applied to the Steane code. Black boxes are blocked out squares, and blue boxes are squares to be filled in such that each number in each row and column is unique. Filling in the boxes using the minimum possible set of numbers (the longest row or column), 1 through 3 here for example, gives the lowest depth scheduling. (a) is the Latin Rectangle before being filled in and (b) is a possible solution for the Latin Rectangle. Each column denotes the encoding circuit's $CNOT$s' targets - so here the 4th physical qubit to the 7th physical qubit, and row j's $CNOT$ has a control on qubit j. So, for example, the top left square corresponds to $CNOT_{1,4}$, the bottom left square corresponds to $CNOT_{3,4}$, and the top right square corresponds to $CNOT_{1,7}$.

stabilizer parity check matrix, H_Z. If one was to apply Gaussian elimination on the X-type parity check matrix of a CSS code, they would be left with

$$H_X = \left(\begin{array}{cc|c} I_{m_X \times m_X} & A_{m_X \times (n-m_X)} & 0_{m_X \times n} \end{array} \right) \tag{3.11}$$

where n is the number of physical qubits in the code, and m_X is the number of rows in H_X - the number of X-type stabilizers in the code. As such, applying step 5 of the Cleve-Gottesman algorithm to this matrix would imply that the encoding circuit has Hadamards on the first m_x qubits. Because these Hadamards act on the qubits from the identity portion of H_X, $I_{m_X \times m_X}$, they can trivially be commuted to the front of the circuit - there are no gates on these qubits before the Hadamards. Additionally, again from step 5 of the Cleve-Gottesman algorithm, because the controls of all the $CNOT$s required to encode will act on a disjoint set of qubits from the targets, the $CNOT$s can also be commuted around trivially. As such, we can schedule them arbitrarily - this is not true of non-CSS codes and is why the Steane Latin Rectangle method only holds for CSS codes. Steane's idea was to reduce the encoding circuits' depths by scheduling the $CNOT$s as follows. The matrix A in the X-type stabilizer parity check matrix, shown in Eq. 3.11, contains the information on all the $CNOT$s required: the index of each row is the control, and each 1 in the row is a target. As such, by scheduling the gates corresponding to each 1 in A to occur simultaneously with the gates of other 1s in different rows and columns, the gates would be acting on a disjoint set of qubits, and hence could then be performed in parallel. Equivalently, taking an $m_X \times (n - m_X)$ grid and setting the 1 locations in A as blank squares in the grid and 0 locations as blocked out squares, one maps this scheduling problem to that of solving a *partial Latin Rectangle*; similar to a Sudoku where each column and each row must not have a repeated number.

To see how this works, consider the following example. The Steane code has the row

reduced X-type parity check matrix

$$H_X = \left(\begin{array}{ccccccc|ccccccc} 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) \tag{3.12}$$

which, with reference to the general row reduced X parity check matrix, Eq. 3.11, has the constituting matrix

$$A = \left(\begin{array}{cccc} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{array}\right). \tag{3.13}$$

Then we can convert this matrix into the partial Latin Rectangle shown in Fig. 3.4 and solve it. That is, instead of applying

$$\begin{array}{c} (CNOT_{1,4})(CNOT_{1,6})(CNOT_{1,7}) \\ (CNOT_{2,4})(CNOT_{2,5})(CNOT_{2,7}) \\ (CNOT_{3,4})(CNOT_{3,5})(CNOT_{3,6}) \end{array} \tag{3.14}$$

to give a depth 9 circuit, as is suggested by the Cleve-Gottesman approach, one instead applies

$$\begin{array}{c} (CNOT_{1,4}CNOT_{2,7}CNOT_{3,6}) \\ (CNOT_{1,6}CNOT_{2,4}CNOT_{3,5}) \\ (CNOT_{1,7}CNOT_{2,5}CNOT_{3,4}) \end{array} \tag{3.15}$$

which is an equivalent circuit in depth 3 - the brackets show each of the parallel steps.

Lastly, while solving partial Latin Rectangles is generally NP-complete [101], the form of Steane's Latin Rectangle's allows them to be mapped to a *bipartite graph* coloring problem, and hence they can be solved in $\mathcal{O}(nN_{CNOT})$ time [102] due to [103], where n is the number of physical qubits in the code and N_{CNOT} is the number of $CNOT$s in the encoding circuit. Additionally, per [1], N_{CNOT} scales as $\mathcal{O}(mn)$ in a worst case scenario, so Steane's Latin Rectangle method has a worst-case time complexity of $\mathcal{O}(mn^2)$ where m is the number of stabilizer generators in the code.

3.2.3 Paetznick's Overlap Method

Paetznick's overlap method is a further add-on to the Cleve-Gottesman algorithm [99, 104]. It reduces the two-qubit gate count of encoding circuits for CSS codes by exploiting similarities between each X-type stabilizer; their *overlaps*. Specifically, it calls for the following steps.

1. Using the A matrix from the reduced X parity check matrix of a CSS code, Eq. 3.11, calculate the X-type stabilizers' overlap,

$$O = A^T A. \tag{3.16}$$

In words, each entry of matrix $O_{i,j}$ tells us how many of the controls of the $CNOT$s targeting qubit i overlap with the controls of the $CNOT$s targeting qubit j. As such it is symmetrical.

For example, a circuit with the gates $CNOT_{1,2}CNOT_{1,3}$ would have $O_{2,3} = O_{3,2} = 1$, as both qubits 2 and 3 share a control on only a single qubit.

2. Take the overlap matrix O and remove the diagonal entries and the entries made redundant by symmetry.

3. Select the entries > 1 remaining in O such that the chosen entries exist on a disjoint set of rows and columns and such that their sum is maximum - as each entry correspond to the number of gates that will be removed plus 1. The chosen entries must be on a disjoint set of rows and columns because each row and column represents one qubit, so removing gates will affect the other entries in the same row and column.

4. To be clear, in this section we will often use brackets to separate two different examples - though we acknowledge that it may be easier to read by just looking at one of the examples and ignoring the brackets. As a toy demonstration of what we mean, we may say 'the cat (dog) is black (white)' to mean 'the cat is black and the dog is white'. Moving on, say the element at row j, column k in O is selected. Then the $CNOT$s that contribute to $O_{j,k} = O_{k,j}$ and target qubit j (k) must be replaced with $CNOT_{k,j}$ ($CNOT_{j,k}$). Note that this new gate needs to be placed in a round where it only propagates the X-type stabilizers from the qubits controlled on the deleted $CNOT$s to the target qubit. That is, by deleting a gate in the encoding circuit that is controlled on p and targeting j (k), the j-th (k-th) operator of the p-th stabilizer is removed. This method identifies places where an encoding circuit has a number of stabilizers, each p, that have an operator on some qubit k (j), and need an operator on qubit j (k). It then deletes the gates that propagate the stabilizers to j (k) individually, and replaces them with a single gate, $CNOT_{k,j}$ ($CNOT_{j,k}$), to propagate all the suitable stabilizers to j (k) at once. Note here we can again see why it is important to place this new gate, $CNOT_{k,j}$ ($CNOT_{j,k}$), only after each of the select stabilizers, the ps, have their X operator on qubit k (j), and before any other stabilizers have an operator on that qubit - so that the stabilizers are not propagated incorrectly.

Recall that the algorithm applies only to CSS codes. It does not apply to all stabilizer codes because it requires that gates be trivially commuted around arbitrarily, which will not generally be allowed in non-CSS codes' encoding circuits.

For an example, we will again use the Steane code - which has the A matrix given in Eq. 3.13. First, the overlap matrix, per its definition in Eq. 3.16, is

$$O = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 3 & 2 & 2 & 2 \\ 2 & 2 & 1 & 1 \\ 2 & 1 & 2 & 1 \\ 2 & 1 & 1 & 2 \end{pmatrix}. \tag{3.17}$$

This can then be written without the diagonal and symmetric entries, and with the corresponding physical qubits labeled as

	4	5	6	7
4		2	2	2
5			1	1
6				1
7				

where we know that this denotes qubits 4 through 7 from comparing Eq. 3.12 and Eq. 3.13. See that the only overlap occurs on qubit 4 and qubit 5, 6, or 7. Here we choose to exploit the overlap between qubits 4 and 6. Because the overlap is 2, we want to remove two gates and replace them with one. Specifically, we need to remove the two gates from the encoding circuit, given in Eq. 3.15, that contribute to $O_{4,6} = O_{6,4}$ and target qubit 4 (6). We then replace them with $CNOT_{6,4}$ ($CNOT_{4,6}$). See that the gates in the encoding circuit, Eq. 3.15, that target qubits 4 and 6 and have the same control are $CNOT_{1,4}$ matched with $CNOT_{1,6}$ and $CNOT_{3,4}$ matched with $CNOT_{3,6}$. That is, the two gates $CNOT_{1,4}$ ($CNOT_{1,6}$) and $CNOT_{3,4}$ ($CNOT_{3,6}$) are removed from the encoding circuit and replaced with $CNOT_{6,4}$ ($CNOT_{4,6}$). When making these changes however, one needs to be careful to place the new gate in a round where it only propagates the stabilizers from qubits 1 and 3 to qubit 4 (6) i.e. right after $CNOT_{1,6}$ ($CNOT_{1,4}$) and $CNOT_{3,6}$ ($CNOT_{3,4}$). This way, the operators in the stabilizers missing from removing these gates are replaced, and there will not be any unwanted propagation of stabilizers. Also note that this process can increase the depth of the circuit, so it may take a number of tries to find an arrangement that can preserve the depth. Nonetheless, if we were to make the change that adds the gate $CNOT_{6,4}$, the new encoding circuit would no longer have the gates in Eq. 3.15, and instead have

$$\begin{aligned}&(\cancel{CNOT_{1,4}}CNOT_{2,7}CNOT_{3,6})\\&(CNOT_{1,6}CNOT_{2,4}CNOT_{3,5})\\&(CNOT_{1,7}CNOT_{2,5}\cancel{CNOT_{3,4}}\boldsymbol{CNOT_{6,4}})\end{aligned} \tag{3.18}$$

where the removed gates have a strike through them and the new gate is in bold. See that we have 8 two-qubit gates instead of 9, and that the depth is still 3 - noting that each bracket holds one parallel step. Hence, by using the fact that later in the circuit the stabilizers propagated from qubits 1 and 3 both have an X on qubit 6 and need an X on qubit 4, we were able to remove a gate and improve the encoding circuit's efficiency. See finally that this is the only optimization that can be made for this example because the entries of O in the other rows and columns are all 1 and hence correlate to no change in the circuit, per step 3 in the method.

Note, we estimate that this algorithm generally has at least a worst-case time complexity of $\mathcal{O}((n-m_X)^3)$, where n is the number of physical qubits in the given CSS code, and m_X is the number of X-type stabilizers in the code. We base this on steps 1 and 2 of the algorithm being trivial, and step 3 boiling down to a minimum-weight perfect matching problem. That is, viewing step 3 of the algorithm in this way - as a graph, each qubit in the in the overlap matrix, O, is a node, and each non-diagonal and non-redundant value (from the matrix's symmetry) in the overlap matrix, O, is an edge. Because O is an $(n-m_X)\times(n-m_X)$ matrix - from A being an $m_X \times (n-m_X)$ matrix, we have $N_{\text{nodes}} = n - m_X$ nodes. Additionally, half of the non-diagonal elements in an $(n-m_X)\times(n-m_X)$ matrix is

$$\sum_{j=0}^{n-m_X-1} (n - m_X - 1 - j) \tag{3.19}$$

$$= \sum_{j=0}^{\alpha} (\alpha - j) \qquad \triangleright \text{where } \alpha = n - m_X - 1 \tag{3.20}$$

$$= \frac{\alpha}{2}(\alpha + 1) \tag{3.21}$$

$$= \frac{1}{2}(n - m_X - 1)(n - m_X) \tag{3.22}$$

where we have used that the sum converges. In words, the above equation takes the $n-m_X-1$ relevant elements in the matrix's first row (all the elements in the first row except the diagonal), and then adds that value to itself subtract 1, then the same subtract 2, and so on. Thus, it finds the number of edges in the graph: $N_{\text{edges}} = \frac{1}{2}(n - m_X - 1)(n - m_X)$. And so, because the lowest complexity algorithm for solving minimum-weight perfect matching problems has complexity $\mathcal{O}(N_{\text{nodes}}(N_{\text{edges}} + N_{\text{nodes}} \log(N_{\text{nodes}})))$ due to [105], the Paetznick Overlap Method will generally have a worst-case complexity of $\mathcal{O}((n - m_X)^3)$. We note that this is only a lower bound for the worst case complexity because step 4 has not been accounted for - it is non-deterministic and could take many potential rounds of rearranging the circuit before an optimization of appropriate depth is found.

4
Our Algorithm and Results

In this chapter we present our research. Specifically, we provide our work's motivation and aim, our algorithm for finding efficient encoding circuits, and we benchmark our algorithm's performance against the Cleve-Gottesman algorithm.

4.1 Motivation and Aim

Encoding is the first step in any error correction procedure. It generally involves applying many two-qubit, entangling gates - because quantum error correction is built around the idea of spreading information over additional qubits through entanglement. Minimizing the number of these multi-qubit gates and minimizing the distances over which they must act is important because two-qubit gates are among the lowest-fidelity operations - particularly when they are applied over relatively large physical distances [9, 55, 77]. Additionally, having fewer two-qubit gates in a circuit generally reduces the risk of correlated errors propagating from mid-circuit faults, and can also help passively reduce depth - lessening the likelihood of storage errors and more generally reducing the circuits' time overhead. But this is not the full story. It is also important to further reduce the risk of storage errors by actively reducing circuit depth, to reduce noise by also decreasing the single-qubit gate count, and to minimize the number of ancilla qubits required. Specifically on this last point, qubits are incredibly expensive in near term quantum devices, and so it is perhaps most important to optimize the use of ancillas.

Encoding circuits are not just useful for encoding data. For instance, encoding circuits are used in Knill [57, 58] and Steane [56] syndrome extraction, which call for encoded ancilla blocks. Encoding circuits can also be used to find various properties of the codes they represents and they are important in other applications, such as in tensor network decoding [59, 67, 78, 79]; where, because the decoder requires contracting the tensor network representation of the encoding circuit, it is vital to the decoder's latency that there be as few

loops as possible - which is generally aided by reducing the multi-qubit gate count. Note that when discussing the latency of a tensor network decoder, we are not talking on the time scales practical for decoding in a quantum computer - because tensor network decoders are likely too slow for applications in real devices. Instead, we are talking on the time scales of any other classical algorithm, because tensor network decoders, as afforded to them by their ideal accuracy, find their purpose in estimating a code's optimal performance [59, 67, 78, 79]. As a digression, see that one could in fact make encoding the only non-trivial component of an entire error correction scheme: by using a tensor network decoder with Knill or Steane syndrome extraction.

To minimize the overhead of quantum circuits, various circuit optimizers have been created [65, 80, 82–84]. However, because of the massive number of ways a circuit with many qubits and gates can be equivalently constructed, and because restrictions in classical memory prevent more than ~ 20 qubits from being represented as a density matrix, these optimizers are currently severely limited in capability. In particular, their runtime complexity is polynomial in gate count [63–65]. As such, often sub-optimal constructions must be used. This issue is exasperated in useful quantum computing, which calls for quantum codes that entangle many qubits, and will hence often present with large, far from optimal encoding circuits - too large for current optimizers. With the fact that optimizers should not be expected to improve rapidly enough to address this problem [65, 66] - as evidenced by them suffering from the issues mentioned above despite being well-studied, the challenge of finding low-overhead encodings persists.

Finding low-overhead encoding circuits has not been well addressed in the literature. In particular, as discussed in our review of such methods in Subsection 3.2, the best known current methods for optimizing ancilla-free encoding circuits' two-qubit gate counts and depths apply only to CSS codes, and in general have unfavourable time complexity. Our aim is to confront this gap in the literature by designing an algorithm that returns ancilla-free encoding circuits for all stabilizer codes at a reduced cost as compared to the most prominent method in the literature: the Cleve-Gottesman algorithm. That is, without adding to the time complexity, we want to reduce the gate count - particularly the two-qubit gate count; reduce the encoding circuit's depth; reduce the distances over which the two-qubit gates act; and use no ancilla qubits. Lastly, we note that not significantly adding to the algorithm's time complexity is important to keep our algorithm practical.

4.2 Our Algorithm

In this section we will introduce our algorithm for finding efficient encoding circuits. We will first give some context for our algorithm, and present its basic idea. Then, in separate subsections, we will explain what optimizations we have made.

The idea of our algorithm is based on reversible circuits [88–93]. As a brief aside, the synthesis of reversible circuits has been studied for over 40 years as they have various applications in electrical engineering [93]. Roughly speaking, a circuit is said to be reversible if its inputs can be recovered from its outputs. Well, observe that all measurement-free quantum circuits are reversible, because a unitary operation, U, has a reversed operation, $U^\dagger$, that satisfies $U^\dagger U = I$. That is, the outputs of a circuit implementing the unitary U fed into

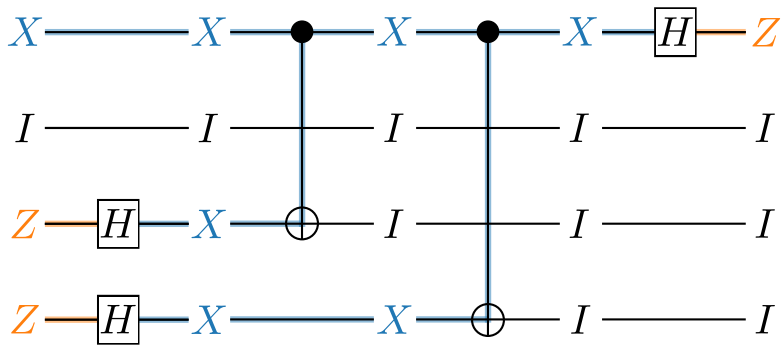


Figure 4.1: A circuit where some stabilizer is shown being 'updated' by each operation. Reading left to right, the stabilizer $XIZZ$ is de-encoded into $ZIII$. Reading from right to left, the stabilizer $ZIII$ is encoded into $XIZZ$.

the circuit implementing $U^\dagger$ gives the original inputs of U. Equivalently, an input state $|\psi\rangle$ undergoing an operation U gives the output state $U|\psi\rangle$, hence, applying the operation $U^\dagger$ to this output state gives $U^\dagger U|\psi\rangle = |\psi\rangle$ the input state, as required. However, these reversible circuit methods are generally built from an operator U, which is not ideal for our encoding problem where we have a code defined by its stabilizers and want to find its encoding circuit. As such, the idea is to think in the Heisenberg picture and set the output of some unknown encoding circuit - that implements a unitary U - to be the code's stabilizers. Additionally, set the input of this encoding circuit to be the initial state of the register - in terms of its stabilizers. Then find the de-encoding circuit implementing $U^\dagger$ - that maps the stabilizers to some initial, simple operators. This way, the encoding circuit is just $(U^\dagger)^\dagger = U$. We note that we do not explicitly find the de-encoding unitary $U^\dagger$, instead it is easier to find the de-encoding circuit in terms of some gates - in which case the encoding circuit is simply just the conjugate transpose of said gates. For example, a de-encoding circuit $CNOT_{4,3}H_4S_4^\dagger$ corresponds to the encoding circuit $S_4H_4CNOT_{4,3}$.

Before moving onto a detailed explanation of our algorithm, we will summarize its main steps. This summary will hopefully serve as a helpful reference in understanding our algorithm and will also show where each optimization fits into our procedure. As an input, our algorithm takes a code's stabilizers in the form of an $m \times n$ parity check matrix. It then executes the following.

1. Select a row of the parity check matrix (a stabilizer) that has more than one non-identity operator. These rows are chosen according to Subsection 4.2.2.

2. Apply the single-qubit gates that map each operator in the selected stabilizer/row to a Pauli X. This is elaborated on in Subsection 4.2.1.

3. Reduce the chosen stabilizer to weight 1 by applying $CNOT$s. These $CNOT$s are found using stabilizer propagation, per Subsection 4.2.1, and their scheduling is optimized in Subsection 4.2.3.

4. Apply these gates from steps 2 and 3 to all rows in the parity check matrix.

5. Repeat steps 1 through 4 for all stabilizer generators of the code. The gates used, in the order they were applied, make up the de-encoding circuit. The reverse of this circuit (the conjugate transpose) is the encoding circuit - as per Subsection 4.2.1.

6. Run a simulation of this circuit to find what Pauli eigenstate is encoded, and add single-qubit gates to produce the desired logical state, as discussed in Subsection 4.2.5.

We also note that the complete pseudocode of our algorithm is provided in Appendix B.

4.2.1 The Backbone

To reduce an arbitrary stabilizer down to having just a single non-identity operator, we need a method for rotating Z operators to X operators (Hadamard gates, H), a method for rotating Y operators to X operators (which we will call O_G gates and define shortly), and a gate to cancel out X operators on different physical qubits ($CNOT$s). That is, because stabilizers are just products of Paulis, any stabilizer can be made weight 1 by applying single-qubit gates such that every non-identity operator in the stabilizer is a Pauli-X, and then use $CNOT$s to propagate these Xs onto the other qubits with an X in that stabilizer, so that they square to the identity. For example, in Fig. 4.1 we have shown the reduction of the stabilizer $XIZZ$ into the stabilizer $ZIII$.

Before proceeding, we will define what we call *our gate*. Our gate has the matrix form

$$O_G = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & 1-i \\ 1+i & 0 \end{pmatrix} \tag{4.1}$$

which was found from imposing that O_G is Hermitian, $O_G^\dagger = O_G$, that O_G is unitary, $O_G^\dagger O_G = O_G O_G^\dagger = I$, and that $YO_G = O_G X$ where Y and X are Pauli-Y and Pauli-X respectively. Our gate is equivalent to the gate $R_Z(\frac{\pi}{2})X$ and is a *Clifford gate*: mapping $Y \leftrightarrow X$ and $Z \rightarrow -Z$. As such, it indeed fullfills our requirement of rotating Y operators to X operators. Moreover, the reason we use this gate, as opposed to a more traditional $S^\dagger$ gate, is because it is Hermitian - unlike $S^\dagger$. That is, as we have mentioned, because our algorithm finds a de-encoding circuit and then takes the conjugate transpose to find the encoding circuit, our gate perhaps yields more elegant solutions. As instead of having say $S^\dagger$ in the de-encoding circuit and thus S in the encoding circuit, we will have just O_G in both the encoding and de-encoding circuits.

So far we have established that we can de-encode a single stabilizer. However, to find the encoding circuit, all of the stabilizer generators need to be de-encoded. This may seem daunting because the gates used to reduce one stabilizer's weight must also be applied to all the other stabilizers - potentially propagating stabilizers that have already been de-encoded. For example, partially reducing a stabilizer $XIXX$ using $CNOT_{1,3}$ gives $XIIX$. But say another stabilizer of that code - that has already been de-encoded - is $XIII$. Well, the $CNOT_{1,3}$ is also applied to this stabilizer, which propagates it to $XIXI$ - undoing the previous de-encoding. However, this is actually of no concern. Recall that products of stabilizers are also stabilizers, Eq. 2.2, and that stabilizers commute. More specifically, once a stabilizer is de-encoded, to say have its single non-trivial operator on qubit k, then every other stabilizer of the code must commute with it, and so every other stabilizer must have

either an identity on qubit k or that same operator on qubit k. As such we have two cases: first, an identity on k, which does not present issues because identities do not propagate; and second, an operator on qubit k that is the same operator on qubit k of the de-encoded stabilizer, in which case we can multiply that stabilizer by the de-endcoded stabilizer - giving a new stabilizer generator with an identity on qubit k - which will not propagate later. In terms of how this looks on the parity check matrix in the algorithm, once a stabilizer is de-encoded, where that non-trivial operator is left, one can clear its column in the parity check matrix to 0 and proceed without worrying. So if we were to return to our earlier example of the stabilizers $XIXX$ and $XIII$ under $CNOT_{1,3}$, we can recognise that an equivalent set of stabilizer generators is $IIXX$ and $XIII$, and hence one would not apply $CNOT_{1,3}$, and instead use $CNOT_{3,4}$, which leaves the two de-encoded stabilizers $IIXI$ and $XIII$.

In this way one is able to de-encode each stabilizer generator of the code. That is, r stabilizer generators of the code can be reduced to r weight-1 stabilizers acting on a disjoint set of qubits. Lastly, each of these de-encoded, weight-1 stabilizers acting on a disjoint set of qubits can be rotated to a Pauli-Z through O_G gates and Hadamards. So, the de-encoding circuit can map an encoded state to a register where all the non-data qubits are in the state $|0\rangle$ - the $+1$ eignenstate of Z. Hence, as required, the encoding circuit starts from a simple to prepare, standard register: all non-data qubits, and optionally all the data qubits, in the $|0\rangle$ state.

As a simple example of this algorithm in action, where for simplicity we do not rotate the stabilizers to Pauli-Xs, consider the 3-qubit repetition code. Recall from Section 2.4 that the 3-qubit repetition code has the stabilizers $\langle Z_1Z_2, Z_2Z_3\rangle$. In $m \times n$ parity check matrix form, these stabilizers are

$$H = \begin{pmatrix} 2 & 2 & 0 \\ 0 & 2 & 2 \end{pmatrix}. \tag{4.2}$$

With reference to how Zs propagate, Fig. 3.1, see that

$$H = \begin{pmatrix} 2 & 2 & 0 \\ 0 & 2 & 2 \end{pmatrix} \xrightarrow{CNOT_{2,3}} \begin{pmatrix} 2 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} \xrightarrow{CNOT_{1,2}} \begin{pmatrix} 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}. \tag{4.3}$$

That is, applying $CNOT_{2,3}$ and then $CNOT_{1,2}$ de-encodes the state. Specifically, the state is now stabilized by Z_2 and Z_3, making it consistent with the state $|\psi\rangle \otimes |0\rangle \otimes |0\rangle$ where $|\psi\rangle$ is an arbitrary data qubit. Therefore, to encode the state $|\psi\rangle \otimes |0\rangle \otimes |0\rangle$ we can apply these gates in reverse; $CNOT_{1,2}$ and then $CNOT_{2,3}$, giving us

$$\begin{pmatrix} 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} \xrightarrow{CNOT_{1,2}} \begin{pmatrix} 2 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} \xrightarrow{CNOT_{2,3}} \begin{pmatrix} 2 & 2 & 0 \\ 0 & 2 & 2 \end{pmatrix} = H. \tag{4.4}$$

As a note, observe that this encoding is corroborated by the encoding circuit given in Fig. 2.2.

Here we can see the most important difference between our algorithm and the Cleve-Gottesman algorithm: that after Gaussian elimination, the Cleve-Gottesman finds all the encoding circuits' gates simultaneously, whereas our way has no Gaussian elimination and instead just applies the gates needed at any one time - letting the other stabilizer propagate however. Breaking the computation up like we have allows us to exploit certain optimizations

not possible in the Cleve-Gottesman approach. We will present these optimizations in the subsections to follow.

Lastly, we must make two notes. First, there are some caveats to the presentation above regarding issues with phases and what logical state is actually prepared. These are addressed in Subsection 4.2.5 and do not amount to any problems - just some additional steps in the algorithm. The second note is that we acknowledge our algorithm's similarity, as it has been presented so far, to that in [95, 96], which finds encoding circuits for qudits by similarly applying single-qubit gates and then two-qubit gates to reduce the weight of each stabilizer. However, the most important distinction to make is that their method does not introduce any of the optimizations that follow.

4.2.2 Reducing Gate Count

As alluded to earlier, we attempt to reduce the encoding circuits' gate counts through our selection of which stabilizer to de-encode in each step. In particular, we introduce a 'cost of de-encoding' function, which for each stabilizer generator is simply calculated using

$$N_{CNOT} \times C_{CNOT} + N_H \times C_H + N_{O_G} \times C_{O_G} \tag{4.5}$$

where N_A is the number of A gates required to de-encode the stabilizer, and C_A is the user inputted cost of applying a gate A. So, to calculate the cost of de-enoding a stabilizer, we must find how many of each gate is required. The number of Pauli-Z operators in the stabilizer is the number of Hadamards needed, N_H; the number of Pauli-Ys in the stabilizer is how many O_G gates are needed, N_{O_G}; and the number of $CNOT$s, N_{CNOT}, is the number of non-identity operators in the stabilizer minus 1. For instance, following the method seen in the previous subsection, the stabilizer $IXXZXIIY$ will require $N_H = 1$ Hadamard as it has a single Pauli-Z, $N_{O_G} = 1$ of our gates as it has a single Pauli Y, and $N_{CNOT} = 4$ $CNOT$s as it has 4 non-trivial operators that we need to map to the identity.

Next, we make our algorithm *greedy*; choose the local minimum cost in each step. That is, we separately calculate the cost of de-encoding each stabilizer, pick the minimum cost one (that is not already de-encoded), de-encode it, add those gates to the de-encoding circuit and apply them to all the other stabilizers, and then re-evaluate the costs and repeat until the de-encoding circuit is complete - all the stabilizer generators are weight 1. While this will generally not give a solution with a minimum number of gates globally, we expect that it will give circuits with reduced gate-counts as compared to the Cleve-Gottesman algorithm. We expect this because in each step, the gates we apply affect the other stabilizer generators and can reduce (or, in principle, increase) their weight. So, by selecting the local minimum, we take full advantage of this by effectively increasing the chances that we get some encoding/de-encoding for free - without applying extra gates.

As an example, set the cost of a $CNOT$ to $C_{CNOT} = 10$, and the cost of both the single-qubit gates to $C_{O_G} = C_H = 1$. In this case, one could potentially expect a code with the stabilizer generators and associated costs

$$XIXII \quad \triangleright \text{ with cost 10,} \tag{4.6}$$
$$ZZZZI \quad \triangleright \text{ with cost 34,} \tag{4.7}$$
$$IXXZI \quad \triangleright \text{ with cost 21,} \tag{4.8}$$
$$IIIIX \quad \triangleright \text{ with cost 0.} \tag{4.9}$$

The algorithm would then proceed by choosing to reduce the first stabilizer, $XIXII$, because it has the lowest cost of any not already de-encoded stabilizer. The algorithm could return the gate $CNOT_{1,3}$, which, from how Paulis propagate, would then update the stabilizers to

$$XIIII \quad \triangleright \text{ with cost 0,} \tag{4.10}$$
$$IZZZI \quad \triangleright \text{ with cost 23,} \tag{4.11}$$
$$IXXZI \quad \triangleright \text{ with cost 21,} \tag{4.12}$$
$$IIIIX \quad \triangleright \text{ with cost 0.} \tag{4.13}$$

Notice in the above example that applying this gate has reduced the cost of the second stabilizer, which is how we potentially save on gates in this method. Then in the next step, applying H_4 and then $CNOT_{1,3}CNOT_{1,2}$, we have

$$XIIII \quad \triangleright \text{ with cost 0,} \tag{4.14}$$
$$IIZXI \quad \triangleright \text{ with cost 11,} \tag{4.15}$$
$$IXIII \quad \triangleright \text{ with cost 0,} \tag{4.16}$$
$$IIIIX \quad \triangleright \text{ with cost 0.} \tag{4.17}$$

See again that the cost of the second stabilizer has been reduced. Then in the final step we apply H_3 and then $CNOT_{3,4}$ to get the de-encoded stabilizers

$$XIIII, \tag{4.18}$$
$$IIXII, \tag{4.19}$$
$$IXIII, \tag{4.20}$$
$$IIIIX, \tag{4.21}$$

as required. That is, our encoding circuit would have 4 $CNOT$s and 2 Hadamards. This example can also be used to show why it is important that the local minimum cost stabilizer is selected in each step, as say we were not evaluating the cost, and just selecting the stabilizers randomly. Then, instead of de-encoding the first stabilizer, $XIXII$, first, we could have randomly chosen to de-encode the second stabilizer, $ZZZZI$, first. In this case, we might apply $H_1H_2H_3H_4$ and then $CNOT_{1,2}CNOT_{1,3}CNOT_{1,4}$, giving

$$IZZII \quad \triangleright \text{ with cost 12,} \tag{4.22}$$
$$XIIII \quad \triangleright \text{ with cost 0,} \tag{4.23}$$
$$IZZXI \quad \triangleright \text{ with cost 22,} \tag{4.24}$$
$$IIIIX \quad \triangleright \text{ with cost 0.} \tag{4.25}$$

Then finishing out this example, one possible scenario requires a total of 5 $CNOT$s and 6 Hadamards. More than the greedy algorithm example. As such, it is easy to see how on larger codes, this difference in performance could grow significantly. However, we do qualify this statement by acknowledging that greedy algorithms may not always give a better solution. Nonetheless, as said before, because applying gates to de-encode one stabilizer has the potential to reduce the cost of other stabilizers, by selecting the minimum cost stabilizers in each step, we are likely to benefit from this through getting some stabilizers reduced in weight for free - reducing the number of gates required to de-encode and hence to encode. That is, we expect our locally optimal choices to translate to good global solutions. And again, more so than this, we hypothesize that our algorithm will generally outperform the Cleve-Gottesman algorithm for the same reason - as the encoding gates they find do not influence the other stabilizers, so they cannot get stabilizers reduced in weight for free like us.

Note also, in our algorithm, we break equal cost ties by selecting one of the equal cost stabilizers randomly. As such, multiple iterations of our algorithm will generally return non-equivalent circuits - of varying efficiency.

As a final point, it might be informative for one to see how a computer views our algorithm. That is, it de-encodes the $m \times n$ parity check matrix as follows

$$H = \begin{pmatrix} \boxed{\begin{matrix} i & \dots & j & k \\ \vdots & \ddots & \vdots & \vdots \\ l & \dots & m & n \\ p & \dots & q & r \end{matrix}} \end{pmatrix} \rightarrow \begin{pmatrix} \boxed{\begin{matrix} s & \dots \\ \vdots & \ddots \end{matrix}} & \begin{matrix} 0 \\ \vdots \\ 1 \end{matrix} & \boxed{\begin{matrix} t \\ \vdots \end{matrix}} \\ \begin{matrix} 0 & \dots \end{matrix} & & \begin{matrix} 0 \end{matrix} \\ \boxed{\begin{matrix} u & \dots \end{matrix}} & & \boxed{\begin{matrix} v \end{matrix}} \end{pmatrix} \rightarrow \cdots \rightarrow \begin{pmatrix} 0 & \dots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & 1 & 0 \\ 0 & \dots & 0 & 1 \end{pmatrix} \tag{4.26}$$

where $i, j, k, \dots r \in \{0, 1, 2, 3\}$ represent the stabilizers' constituting operators, per Subsection 2.6; the arrows represent each step where a stabilizer is de-encoded; and the boxes represent the elements still under consideration for gates. In particular, there are as many steps as there are rows in the matrix, because in each step a single stabilizer is de-encoded, and the algorithm does not halt until all the stabilizer generators have been de-encoded. Lastly, we say explicitly that there are as many non-zero numbers in the final matrix as there are rows in the matrix - a single non-zero number in each row i.e. a single non-identity operator in each stabilizer; 1s are not necessarily in the positions shown; and the second last row is not necessarily de-encoded first.

4.2.3 Reducing Depth

While we expect that implementing the cost function and making our algorithm greedy, as explained above, will reduce the gate count - and hence generally improve the depth too, we can make an additional change to further reduced the depth. Specifically, when scheduling the $CNOT$s to de-encode a given stabilizer, we make sure to apply these $CNOT$s on a disjoint set of qubits as much as possible - making the operation as parallel as possible. That is, we take the indices of the qubits with non-identity operators in the stabilizer to be de-encoded, $i, j, k, l, \dots$; pair them, $(i, j), (k, l) \dots$; and then schedule the $CNOT$s according to these pairs, $CNOT_{i,j}CNOT_{k,l} \dots$ and so on. This will, per stabilizer propagation in Subsection

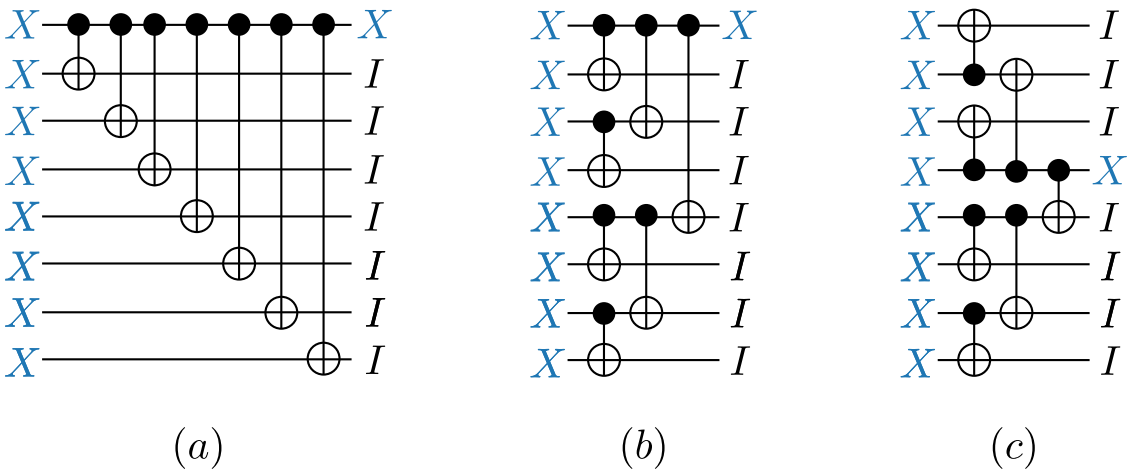


Figure 4.2: (a) The scheduling of two-qubit gates as required in the Cleve-Gottesman approach - the controls must be on the first X so as to not propagate around other stabilizers. See that the circuit's depth scales linearly with the weight of the stabilizer - here the depth is 7. (b) The kind of gates we can apply in our algorithm to de-encode - because the controls can be on any non-identity operator in the stabilizer. This circuit is not equivalent to (a) but it de-encodes the stabilizer nonetheless. See that the depth scales logarithmically with the stabilizer's weight. That is, here the depth is 3. (c) Another change that reduces the physical distance of the interactions - if we assume a linear array of qubits. It can also be simply generalised to more complicated qubit arrays and account for poor connectivity in the hardware, as is discussed in Appendix B. Note that this circuit is not equivalent to (b) or (c), it just achieves the same desired effect: de-encoding. One could easily implement this into our algorithm at no extra cost to complexity - for linear arrays, by reversing the order of specific pairings, as discussed in the main text. We note however that we did not include this in our implementation of our algorithm because it is one of the more minor points and its effect is highly dependant on the hardware. Also see that (b) sees similar improvements to the physical distance of the gates over the Cleve-Gottesman approach in (a) anyway, and that this will also be further improved if our method reduces the two-qubit gate count - as we expect.

3.1, make the operators on every second qubit of that stabilizer, $j, l, \ldots$, an identity, and leave the other qubits, $i, k, \ldots$, still with a Pauli-X operator. Therefore, we must pair these remaining qubits again, $(i, k), \ldots$, implying $CNOT_{i,k} \ldots$ and continue in this way until the stabilizer is de-encoded. See that each step of pairing the qubits like this indeed only adds at most 1 to the circuit's depth. For an example, we show how this would look for a stabilizer $X^{\otimes 8}$ in Fig. 4.2b. We also show how reversing the order of the first half of the pairings in each step, $(i, j) \to (j, i)$ implying $CNOT_{i,j} \to CNOT_{j,i}$, would look for a stabilizer $X^{\otimes 8}$ in Fig. 4.2c.

Parallelizing our $CNOT$s in this way should offer improvements over the Cleve-Gottesman approach, which has very few or no parallel operations - operations are only parallel if consecutive stabilizers after Gaussian elimination act non-trivially on a disjoint set of qubits. To see this recall that by construction, for each stabilizer, the Cleve-Gottesman algorithm requires a single-qubit to be the control on all the two-qubit gates - so as to not cause the stabilizers to propagate around in an unaccounted for way and mess up the encoding circuit

produced. For instance, consider that their approach applied to the stabilizer $X^{\otimes 8}$ would require the gates shown in Fig. 4.2a - and compare this to our equivalent handling of this problem in Fig. 4.2b and Fig. 4.2c. As such, in addition to us hypothesizing that the Cleve-Gottesman method will require more gates, we further expect that these gates will be less parallel than in our approach. Note that our expected reduction in depth is again a benefit afforded to us by breaking up the calculation of the encoding circuit for each stabilizer - because we are free to apply gates on any qubit.

To be specific, consider a stabilizer to be de-encoded. It will have weight $w \geq 2$. Both our algorithm and the Cleve-Gottesman algorithm will require $w - 1$ two-qubit gates to de-encode such a stabilizer. However, the Cleve-Gottesman approach cannot perform any of these two-qubit gates in parallel, and hence, this portion of the circuit will contribute a depth of $w - 1$. That is, the depth will scale linearly with the weight of the stabilizer. On the other hand, our algorithm matches each of these w operators with another, and applies $CNOT$s accordingly, reducing the weight by $\lfloor w/2 \rfloor$ in each step - we take the floor because odd weights w leave one operator unpaired and hence it is not reduced in that round. Then the remaining $\lceil w/2 \rceil$ operators undergo the same treatment, until the weight $w = 1$ is achieved and the stabilizer is de-encoded. That is, the depth of this portion of the circuit will scale logarithmically, $\lceil \log_2(w) \rceil$.

Recall that in both our algorithm and Cleve-Gottemsman algorithm, each stabilizer generator of the code has a set of two-qubit gates that specifically encode it. Hence, if there are m stabilizers in a code, our algorithm and Cleve-Gottemsan algorithm will have m groups of these two-qubit gates - one for each stabilizer. These two-qubit gates make up the bulk of the encoding circuits - as encoding is all about entangling qubits. As such, because the depth of these gates scales linearly with the weight of the stabilizer in the Cleve-Gottesman algorithm, and logarithmically for ours, our algorithm will generally return circuits with better depth - which should be especially true if our algorithm also uses fewer two-qubit gates as hypothesized. The main reason we qualified this statement with 'generally' is because the Cleve-Gottesman approach could see advantages when many of a code's stabilizers act on a disjoint set of qubits after Gaussian elimination. This is not something we would generally expect though because stabilizers need to overlap on qubits to better identify error locations. Hence, we hypothesize that our algorithm will usually have better circuit depth scaling than the Cleve-Gottesman algorithm, and that the difference will generally be particularly noticeable for codes with high weight stabilizers - because in this high weight regime, our log-scaling depths will deviate greatly from their linear-scaling depths.

Also note that this improvement also reduces the issues regarding long-range entangling operations. For instance, consider again the encoding gates for the $X^{\otimes 8}$ stabilizer, where the Cleve-Gottesman method would return the circuit in Fig. 4.2a, and our algorithm could return the gates shown in Fig. 4.2c. See that if we were to consider a hardware where these eight qubits were linearly spaced by a physical distance D, then the total distance traversed by all the two-qubit gates of the Cleve-Gottesman would be $28D$ and the maximum distance of any one single gate would be $7D$. On the other hand, the gates in our algorithm would total a distance of $9D$ and have the maximum distance of any $CNOT$ as $2D$. Reducing the range of the two-qubit gates in this way comes down to how the qubits with non-identity operators in the given stabilizer are paired. So this idea is easily generalized to

more complex hardwares. For instance, one could use a minimum-weight perfect matching algorithm (discussed in Appendix A.9) where the qubits with non-identity operators on them in the given stabilizer are the nodes and they form a complete graph where the edges represent the cost of performing a $CNOT$ between those two qubits. In this way, the qubits can be in any geometry and the edge costs can represent more than just a physical distance - for instance a hardware with connectivity issues could have higher costs between two physically close qubits because it could be difficult to implement a $CNOT$ across said qubits. Note that this change however could impact the complexity of our algorithm and it may be preferable to instead use a different supporting algorithm, like the union-find algorithm - which is also explained in Appendix A.9. At worst however, with minimum-weight prefect matching as the supplementing algorithm used to pair qubits, our algorithm's complexity would just be a higher order polynomial [105].

4.2.4 Complexity

We have hypothesized that our approach offers advantages due to it breaking up the calculation of the encoding circuit into more steps than the Cleve-Gottesman method. Accordingly, one may see these additional iterations and think they afflict our algorithm's complexity. However, they do not. To see this, recall that the Cleve-Gottesman method calls for Gaussian elimination to be performed on the $m \times 2n$ parity check matrix, where $m = n - k$ is the number of stabilizer generators in the code, n is the number of physical qubits in the code, and k is the number of logical qubits in the code. Well, Gaussian elimination has complexity $\mathcal{O}(m^2 n)$ [106], which is also generally the worst-case complexity of our algorithm, as per Appendix B. Therefore, our algorithm generally provides all of its hypothesized improvements at asymptotically no extra runtime cost.

We say 'generally' in the paragraph above because there are two caveats. First, as explained in Appendix B, the worst-case complexity of our algorithm becomes $\mathcal{O}(mn \log_2(n))$ when the number of logical qubits is $k > n - \log_2(n)$ or equivalently when the number of stabilizers is $m < \log_2(n)$ where n is the number of physical qubits. And second, as explained in Subsection 4.2.5, the worst case complexity of our algorithm goes to $\mathcal{O}(n^2)$ when there are $k > n - \sqrt{n}$ logical qubits, or equivalently, $m < \sqrt{n}$ stabilizers. Moreover, notice that in both these regimes where the complexity increases slightly, the quantum error-correcting codes will generally be notably bad - as there will generally be too few stabilizers to sufficiently correct an adequate number of errors. This holds particularly true for larger codes, where these regimes demand there be relatively fewer stabilizers in the code.

4.2.5 Additional Considerations

As mentioned, there are some caveats to the core mechanism of our algorithm regarding the logical state our encoding circuits produce, and the phases of the stabilizers encoded. These issues are resolved by adding a few additional steps to the algorithm, as we will discuss below.

First we will address the concern where our encoding circuits produce some unknown encoded logical Pauli eigenstate. To see an example of this issue, consider a two-qubit code

with just one stabilizer: Z_1Z_2. For this code, our algorithm might suggest the encoding circuit $H_2H_1CNOT_{1,2}H_1$ where the second qubit is the data qubit. But say that we want this code to have the codewords $|0\rangle_L = |00\rangle$ and $|1\rangle_L = |11\rangle$, or equivalently, the logical operators $X_L = X_1X_2$ and $Z_L = Z_2$. Well, observe that if the data qubit is in the state $|0\rangle$ ($|1\rangle$), instead of producing $|0\rangle_L$ ($|1\rangle_L$), our circuit produces $|+\rangle_L$ ($|-\rangle_L$). Or, equivalently, as can be seen from these states or from Pauli propagation, the encoded logical operators are swapped, $X_L \leftrightarrow Z_L$. However, from similar calculations as above, one can verify that this issue is resolved by appending a Hadamard H_2 to the beginning of the circuit.

Here we will explain a general method for ensuring that our circuits encode into the desired basis. But before that, we need to see why the issue presents in the first place. Consider that our encoding circuits are entirely composed of Clifford gates, which map Paulis to Paulis. So, if we consider our data qubits written in terms of their logical Pauli operations - which form a basis for the logical state-space, in our circuits, these operators may not propagate to their expected encoded logical operations. Note that we know a logical operator is produced (even if it is not the correct one) from the construction of quantum error correcting codes. That is, we produce the correct stabilizers, which by definition fixes the encoded state produced for each data input, and hence fixes the logical operators. As we have seen, an example of the issue is that encoding a data qubit $|0\rangle$ may produce a logical $|+\rangle_L$, and hence the logical operations will have changed; in this instance the code's logical X_L becomes the encoded state's logical Z_L. However, we want the data qubits to be encoded without a change of basis. Equivalently, if we separately placed a Pauli-Z on each data qubit, we want each of these operators to propagate into the code's corresponding logical Z_L operators. To enforce this, we first needed to either find what logical Pauli operators are generated from separately propagating a Z on each data qubit through the encoding circuit, or find what single-qubit Pauli (when multiplied by the now weight-1 stabilizer generators) each logical Z_L of the code propagates to in the de-encoding circuit. We opted for the latter approach because we already have the de-encoding circuit - from finding the encoding circuit, and the other approach will yield a logical Z_L up to a potentially complicated stabilizer, which could make it difficult to discern what logical operation is encoded - whereas, because in the approach we decided on the stabilizers are weight-1 and do not act on the data qubits, it is much easier to determine the encoded operation. So, after seeing what operator the de-encoded logical Z_L corresponds to, we need to apply a gate to that data qubit to make it a Z. For instance, if we found that a logical Z_L propagated through the de-encoding circuit into a Z, no gate would be required. But if it propagated to an X instead, we would need to apply a Hadamard. This guarantees that the encoded logical Zs indeed act like Zs on their respective data qubits, and hence from our argument above, that the correct encoded state is produced. Note that this process generally does not add to our algorithm's complexity because it is only called once and because the de-encoding circuit is a Clifford circuit - giving this process complexity $\mathcal{O}(n^2)$ due to the Gottesman-Knill theorem [26, 107], where n is the number of physical qubits in the code. We qualify this with 'generally' not affecting our typical worst-case complexity of $\mathcal{O}(m^2n)$ because it is only worse when $m^2 < n$, which implies that $k > n - \sqrt{n}$ where $m = n - k$ is the number of stabilizers for a code with k logical qubits and n physical qubits, which generally does not hold for good error correcting codes - as there will be too few stabilizers to sufficiently correct all low-weight errors.

Next, we will consider the other issue: phases. To see the problem, recall that our algorithm treats Pauli-Ys as XZs, our gate maps $Z \to -Z$, and Hadamards map $XZ \to -XZ$. Hence, without proper treatment, the stabilizers and logical operators will only be produced correctly up to a phase - which could potentially impact the syndrome measurement outcomes and the logical eigenstates encoded. To fix this issue, one needs to track the phases. That is, in our algorithm, we generate a vector with as many entries as there are stabilizer generators and logical Pauli Z_Ls in the code. Upon initializing the vector, all the Ys in the stabilizers and logical Z_Ls are converted to XZs, and a phase of i is recorded in the phase vector. Then we apply gates according to the algorithm, and if there is anything that would change a phase, we record it in this phase vector. Then after all is done, we can look at any non-trivial phases in the vector and apply single-qubit gates to the corresponding de-encoded operation at the beginning of the encoding circuit. As an example, say a stabilizer $Z_1Z_2X_4X_6$ is found to be de-encoded to $-Z_4$. This will clearly cause issues if we assume by default that the register contains the state $|0\rangle_4$, the $+1$ eigenstate of Z_4, as the stabilizer Z_4 would then be propagated into the incorrect stabilizer $-Z_1Z_2X_4X_6$. To fix this particular problem, X_4 must be applied at the beginning of the encoding circuit. That way, Z_4 anti-commutes with it, giving the required $-Z_4$ eigenstate that would then correctly propagate into the stabilizer $Z_1Z_2X_4X_6$, as required. As an aside, consider that it is important when multiplying stabilizers in our algorithm to include the phase in the calculation. Finally, we note that this phase tracking procedure does not add to our algorithm's complexity. This is because the phase vector is initialized (outside of any loops) with complexity $\mathcal{O}(n^2)$ - as we look through all the stabilizer generators and logical Zs for Y operators. Then, whenever a gate is applied that changes a phase, we have a constant time operation that records the phase of that operator accordingly - which is included in the loop that already exists for applying the de-encoding gates to all the stabilizers.

Lastly, we want to reiterate that the circuits produced by our algorithm require no ancillas and hence have no ancilla overhead - like the Cleve-Gottesman method.

4.3 Method

To gather results on the efficiency of the encoding circuits produced by our approach, we ran numerical simulations of both our algorithm and the Cleve-Gottesman method for various quantum error correcting codes. We then compared the two-qubit gate counts, Fig. 4.4, and depths, Fig. 4.6, of the codes' encoding circuits produced by each algorithm.

We tested the algorithms on 101 different codes - selected from [108, 109]. The codes used had three different logical qubit counts, $k = 6$, 16, and 36, and then for each of these logical qubit counts, we selected codes with as low as $n = 12$ physical qubits and as high as $n = 112$ physical qubits. We chose these codes because they fit within the regime expected in near-term quantum devices; where qubits are very expensive, and hence where one would be most concerned with ancilla-free encoding. Note that we did not cherry pick codes. These were all the codes tested, unless otherwise specified, and besides from satisfying the above desired property of being in a near-term regime, we picked them randomly so as to give a fair representation of our algorithm and the Cleve-Gottesman method.

For our tests, we assumed that we were to encode logical $|00\ldots0\rangle$ states. While this

has no implications for the encoding circuits found by our algorithm, it does benefit the Cleve-Gottesman algorithm. More specifically, recall that the encoding circuits produced by the Cleve-Gottesman algorithm can have relative phase issues when encoding logical superposition states, and it can only neglect the explicit encoding of pure-Z type stabilizers when the logical state is $|00\ldots0\rangle_L$. In particular, remember that the Cleve-Gottesman algorithm required additional two-qubit gates - and hence oftentimes additional depth too - to fix this latter issue. As such, the two-qubit gate count and encoding circuit depth results we present for the Cleve-Gottesman algorithm are exact for producing $|00\ldots0\rangle_L$ states and are lower bounds for producing any other logical state. On the other hand, our encoding circuits do not face any such issues and are able to encode any logical state. Hence the results presented for our encoding circuits have no such qualifier. Finally, note that we assume one wants to encode the $|00\ldots0\rangle_L$ logical state because encoding a blank register is generally of most practical relevance - it protects the computation starting from the very first (logical) operation applied.

Here we will make a final couple of comments. First, remember that our algorithm breaks equal cost ties by random selection. Therefore, for a single code, our algorithm is expected to produce various encoding circuits with differing properties - particularly for larger codes with many more equal cost ties to be broken. For this reason, in the numerical simulations, we ran five iterations of our algorithm for each code, and took the median cost encoding circuit as our data point. We did this to reduce the likelihood of our algorithm finding a particularly good or particularly bad encoding circuit - relative to the other encoding circuits it could have produced. We select five iterations because, as we will justify later using the distribution of possible encoding circuit costs that our algorithm could return for a given code, Fig. 4.8, the costs of the produced circuits are mostly concentrated around the mean. Hence, as few as five iterations should be sufficient to give a fair representation of the encoding circuit one could expect from our algorithm. Another note is that we gave $CNOT$s the only non-zero cost in our algorithm's cost function. We did this for two reasons: two-qubit gates have a significantly lower fidelity than single-qubit gates, and hence often we are most interested in reducing their count; and we would only want to include single-qubit gates in the cost function if we intended to compare our single-qubit gate counts to those of the Cleve-Gottesman algorithm. However, there is no such fair comparison because it is a hardware dependant problem. Additionally, the Cleve-Gottesman method uses $CNOT$s, CZs, and CYs, while we just use $CNOT$s; and both approaches use a variety of different single-qubit gates. So, for instance, it would not necessarily be fair to convert the CZs and CYs from the Cleve-Gottesman algorithm circuits into $CNOT$s plus their two respective single-qubit gates because choosing $CNOT$s as the base gate is arbitrary. Lastly, note again that we did not compare any parameters to quantify the physical distance of the entangling operations - because this problem is also entirely hardware dependant.

4.4 Results

Before getting to the complete data, we would like to make the encoding circuits produced by our algorithm more tangible. To do this, in Fig. 4.3, we have included an encoding circuit found for the 5-qubit code using our algorithm. Compare this circuit to the analogous circuit

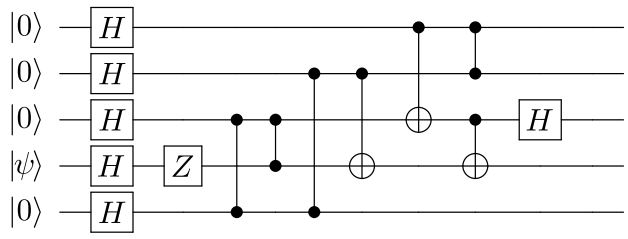


Figure 4.3: Encoding circuit for the 5-qubit code as produced by our proposed algorithm. Here an arbitrary single-qubit state, $|\psi\rangle$, is encoded. Observe that there are 6 Hadamards, a single Z gate, and 7 two-qubit gates: 4 CZs, and 3 $CNOT$s. Also, see that the circuit has depth 6 (or depth 5 if one wants to prepare $|+\rangle$ states instead of $|0\rangle$ states and if ZH can be performed in one time step through a single gate). Note that the CZs are from commuting some Hadamards around the circuit by hand. We do this here for a fairer comparison between our circuit and that produced by the Cleve-Gottesman algorithm - because there is inadvertently a comparison between the number of single-qubit gates. Note that this circuit is not only more efficient than the circuit from the Cleve-Gottesman approach, Fig. 3.3, but it is among the best known encoding circuits for this code - as far as we are aware [94, 100].

produced by the Cleve-Gottesman algorithm, shown in Fig. 3.3. In particular, notice that the circuit our algorithm produced has 7 two-qubit gates and a depth of 5, while that of the Cleve-Gottesman algorithm has 12 two-qubit gates and depth 10 (or 9 by commuting gates around). We note that this improvement by our algorithm is fairly substantial for such a small code - we hypothesized that the greatest benefits would generally be seen in larger codes.

Now, regarding the more complete data, we will consider how the encoding circuits of our algorithm compare to those of the Cleve-Gottesman approach in terms of two-qubit gate count. The data for this is presented in Fig. 4.4. See that generally our algorithm provides encoding circuits requiring fewer two-qubit gates than those of the Clive-Gottesman algorithm, as hypothesized. In particular, with reference to the left side of Fig. 4.5, see that for the random selection of codes studied, our method reduced the number of two-qubit gates by $\sim 41\%$ on average, and by up to $\sim 57\%$ - as compared to the Cleve-Gottesman approach. Additionally, observe that the reduction in two-qubit gate count (due to our algorithm) tended to grow with the number of stabilizers in the code - as per the generally increasing difference between the data points in Fig. 4.4. This was expected because codes with more stabilizers usually leave more room for improvement - there is generally a higher likelihood of a stabilizer's weight decreasing from the gates applied in the de-encoding of other stabilizers, which our algorithm exploits to get lower two-qubit gate counts. We note that this pattern of improving two-qubit gate counts for codes with more stabilizers does not always apply, but we expect the pattern to hold in general. Where it does not hold is in the many sections of data where there is a near constant two-qubit gate count for a group of successive codes. This appears to occur because the codes in each of these sections of data are equivalent up to the inclusion of some weight-1 stabilizers that act non-trivially only on the newly included physical qubits. Note that we would not use these codes in practice because these weight-1 stabilizers effectively do not add to a code's error-correction capabilities.

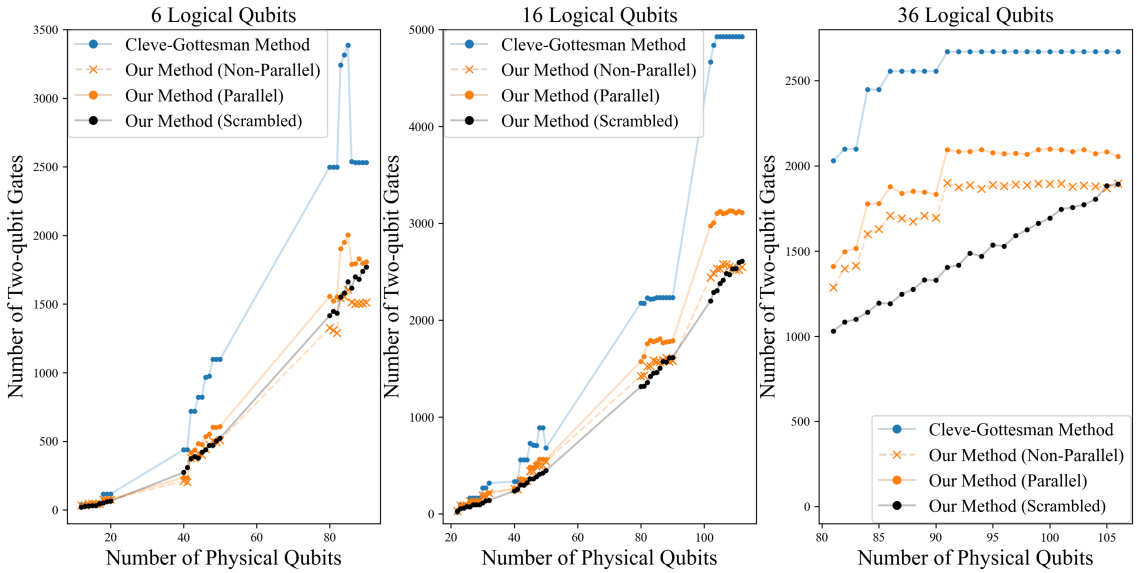


FIGURE 4.4: Two-qubit gate counts of the encoding circuits produced by our algorithm and the Cleve-Gottesman algorithm for various error-correcting codes. Each data point corresponds to a particular code from [108, 109]. Note that the line is included only to guide the eye - it does not represent any data. Also, the data for our algorithm corresponds to the median cost circuit returned after 5 iterations. Here the 'Parallel' data is for the implementation of our algorithm where the $CNOT$s de-encoding each stabilizer are applied such that they are (locally) maximally parallel. On the other hand, the 'Non-Parallel' data for our algorithm is more analogous to the Cleve-Gottesman algorithm, where each group of de-encoding $CNOT$s are (locally) minimally parallel. Our 'Scrambled' data is our complete algorithm (the 'Parallel' method) applied to the scrambled stabilizers of the code - not in row echelon form as they were for the 'Parallel' and 'Non-Parallel' simulations. Therefore, unless otherwise stated, the main text's analysis of the two-qubit gate count refers to the 'Scrambled' data because it best reflects our algorithm's performance. Lastly, as an internal consistency check, see that the two-qubit gate count generally increases with the size of the code. The exceptions to this trend are the flat sections of data, explained in the main text, and the occasional spikes - for instance, see the spike at $k = 6$ logical qubits and $n \sim 80$ physical qubits. These spikes likely result from their particular codes happening to have higher weight stabilizers than the codes around them, when in row-echelon form - thus taking more gates to encode. This is corroborated by no such spikes appearing in the 'Scrambled' data.

To elaborate on why the addition of these weight-1 stabilizers to a code would yield sections of data with a constant number of two-qubit gates, consider that these additional weight-1 stabilizers do not require any two-qubit operations to encode, as they, by definition, are not entangled with the rest of the stabilizer state. Thus, it makes sense that encoding codes equivalent up to some weight-1 stabilizers (that only act non-trivially on qubits with identities in every other stabilizer generator) would not require any additional two-qubit gates, and hence yield these flat lines in the (Cleve-Gottesman, 'Parallel' and 'Non-Parallel') data. However, we do not see this flat line trend in our 'Scrambled' data because we intentionally scramble the stabilizer generators we are encoding (by taking products of stabilizers)

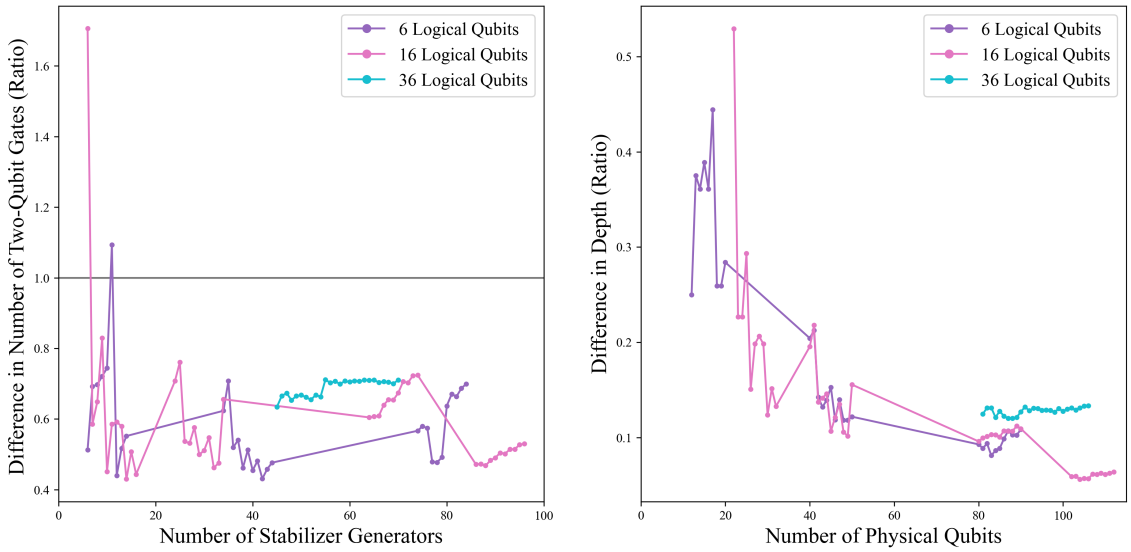


Figure 4.5: On the left we have the number of two-qubit gates in the encoding circuit produced by our algorithm ('Scrambled') divided by the number of two-qubit gates in the respective code's circuit found using the Cleve-Gottesman approach. On the right we have the analogous data for the ('Scrambled') depth. The black line at 1 is included to show when our algorithm produces worse encoding circuits. In summary, the median percentage of two-qubit gates remaining in our circuits is $\sim 59\%$, and the median proportion of depth remaining is $\sim 13\%$ - as compared to the Cleve-Gottesman approach.

to get a representation of how our algorithm would perform without row reduction. Therefore, it makes sense that the two-qubit gate count increases in these sections that are flat for the other data because these stabilizer generators that were weight-1 have been increased in weight, and hence are no longer trivial to encode. Also, note that these sections of constant two-qubit gate counts are not as flat for our algorithm's 'Parallel' and 'Non-Parallel' results - as compared to the Cleve-Gotteman's results - because our algorithm breaks even cost ties by selecting one path randomly, and hence can yield various circuits for a given code. That is, if our 'Parallel' and 'Non-Parallel' algorithm was made deterministic by removing this random selection, it would similarly have these perfectly flat, constant gate-count sections like the Cleve-Gottesman algorithm - noting that the counts from our algorithm would still be reduced though.

See in Fig. 4.4 how the two-qubit gate count data of our algorithm has a similar shape to that of the Cleve-Gottesman algorithm's data. This is likely the result of both methods having the number of two-qubit gates be linearly dependant on the weight of each stabilizer. The reason our 'Parallel' and 'Non-Parallel' data does not exactly match the Cleve-Gottemsman data though is, as explained before, because our method gets some of the stabilizers' weights reduced for free by having the de-encoding gates potentially act non-trivially on a number of the code's stabilizers. This would also explain the general trend in our algorithm's data where codes with more stabilizers require relatively fewer two-qubit gates - because getting these free reductions in stabilizers' weights is more likely if there are more stabilizers.

As another comment regarding the two-qubit gate counts, see on the left side of Fig. 4.5 that our algorithm ('Scrambled') requires more two-qubit gates than the Cleve-Gottesman algorithm for 2 of the 101 codes tested. Investigating this showed that these 2 codes had pure-Z stabilizers with a high combined weight relative to the other stabilizers in the code, and that these codes also had relatively few stabilizers in total. This makes sense because when encoding the logical state $|00\ldots0\rangle_L$, as we are, the Cleve-Gottesman approach gets pure-Z stabilizers without applying any gates, while our method does not. Hence, it is difficult in this regime - of many pure-Z stabilizers and few stabilizers in total - for our algorithm to make up the difference in the number of two-qubit gates required to encode - because in this case the Cleve-Gottesman way gets such a significant amount of the encoding for free. As an example, the worst offender of the codes where the Cleve-Gottesman approach requires fewer gates than ours is for the code in [108, 109] with $k = 16$ logical qubits, $n = 22$ physical qubits, and the stabilizers

$$\langle X_1X_2X_3X_4X_5X_6X_7X_8X_9X_{10}X_{11}X_{12}X_{13}X_{14}X_{15}X_{16}X_{17}X_{18}, \tag{4.27}$$
$$X_{19}, \tag{4.28}$$
$$X_{20}, \tag{4.29}$$
$$X_{21}, \tag{4.30}$$
$$X_{22}, \tag{4.31}$$
$$Z_1Z_2Z_3Z_4Z_5Z_6Z_7Z_8Z_9Z_{10}Z_{11}Z_{12}Z_{13}Z_{14}Z_{15}Z_{16}Z_{17}Z_{18}\rangle, \tag{4.32}$$

as can be seen in the left side of Fig. 4.5. See that this code indeed fits under the regime where we expect the Cleve-Gottesman circuit to have fewer two-qubit gates than ours - because it has few stabilizers and because its pure Z-type stabilizer accounts for much of the weight of all the stabilizer generators in the code. To give some more detail, see that here the Cleve-Gottesman encoding circuit just calls for the seventeen $CNOT$s that encode the stabilizer in Eq. 4.27. On the other hand, our algorithm might first apply $CNOT_{1,2}CNOT_{3,4}...CNOT_{17,18}$ which reduces the stabilizer in Eq. 4.27 to having Xs on odd numbered qubits, and reduces the Z-type stabilizer to having Zs on the even numbered qubit. Continuing, de-encoding the stabilizer in Eq. 4.27 would similarly require seventeen $CNOT$s in total, however, those gates only apply to the odd numbered qubits, and hence the Z-type stabilizer would not be reduced any further. Hence, we would still be required to de-encode the stabilizer

$$Z_2Z_4Z_6Z_8Z_{10}Z_{12}Z_{14}Z_{16}Z_{18} \tag{4.33}$$

which requires eight $CNOT$s: eight more $CNOT$s than the Cleve-Gottesman method. We also note, the two codes that yielded a higher two-qubit gate count both had a low distance; distance 2. We expected this because generally codes with few stabilizers correct few errors. So, to summarize, our results suggest that the Cleve-Gottesman method produces lower two-qubit gate count encoding circuits than our method in the case of encoding the logical state $|00\ldots0\rangle_L$ for codes with few stabilizers, and pure Z-type stabilizers with a relatively high combined weight as compared to the other stabilizers in the code.

Finally, consider in the two-qubit gate count results, Fig. 4.4, the differences between the 'Parallel' and 'Non-Parallel' data. In particular, notice that the 'Parallel' data, in almost

all cases, requires more two-qubit gates than the 'Non-Parallel' data. This was unexpected. Because the two-qubit gate count is reduced by the propagation of operators through a stabilizer's de-encoding gates (that happen to reduce the other stabilizers' weights), we would have hypothesized that the 'Parallel' and 'Non-Parallel' data would be similar for all codes - but not exactly the same because the circuits would be non-equivalent and still have some variation from the randomly chosen equal cost tie breaks. That said, the most likely explanation for this observation is that all the codes for the 'Parallel' and 'Non-Parallel' simulations were inputted in their row echelon form - which were included to give some results for our algorithm encoding the same exact stabilizer generators as the Cleve-Gottesman algorithm. This would perhaps help make sense of this observation because there are such large consecutive portions of identity operators in row echelon form, and so by putting the controls of the $CNOT$s wherever - as we do in the 'Parallel' approach, we are more likely to increase the weight of some stabilizers by propagating Paulis into these sections of identity operators - and hence require more gates. This is less likely in the 'Non-Parallel' approach where the controls of the $CNOT$s are concentrated on just a couple of qubits - propagating up to one Pauli into the identity matrix part of the row echelon stabilizers and decreasing its weight by at most one for each de-encoded stabilizer. As such, if we were to apply our 'Parallel' method to stabilizers not in row echelon from, we would expect its two-qubit gate counts to be closer to the 'Non-Parallel' data. And indeed, the 'Scrambled' data for our algorithm suggests this.

Here we would like to elaborate on why we included the 'Parallel', 'Non-Parallel', and 'Scrambled' two-qubit gate count data for our algorithm. The 'Parallel' data was included because it shows a reduction in the two-qubit gate count (and depth as we will see) as compared to the Cleve-Gottesman method - when encoding from the same set of stabilizer generators: their row echelon form. Hence, we can show that the reduction in the number of two-qubit gates in the actual ('Scrambled') implementation of our algorithm - on less structured stabilizer generators - is not just due to the inputted stabilizers being different (although equivalent). It was also interesting to include the 'Non-Parallel' data because it corresponds to the version of our algorithm that is most similar to the Cleve-Gottesman method and demonstrates that a reduction in two-qubit gate count was achieved just by implementing a cost function and allowing each group of de-encoding two-qubit gates to be controlled on any one qubit.

Now we will compare the depth of the encoding circuits produced by the Cleve-Gottesman algorithm and our algorithm. As per the data for the depth of the circuits found by each algorithm, Fig. 4.6, our method returned lower depth circuits for all of the 101 randomly selected codes - corroborating our hypothesis. In particular, with reference to the right side of Fig. 4.5, for all of the codes tested, our algorithm's encoding circuits' depths were reduced by at worst $\sim 47\%$ and by up to $\sim 95\%$ as compared to the circuits produced by the Cleve-Gottesman algorithm. As a specific example, the biggest difference in depth we found was for the code from [108, 109] with with $k = 16$ logical qubits and $n = 104$ physical qubits, for which the Cleve-Gottesman algorithm gave a circuit of depth 4415 and our algorithm gave a circuit of depth 249. Also notice that the code discussed earlier that had a significant increase in number of two-qubit gates in our approach, with $k = 16$ logical qubits and $n = 22$ physical qubits, still saw a reduction in depth of $\sim 47\%$ despite having

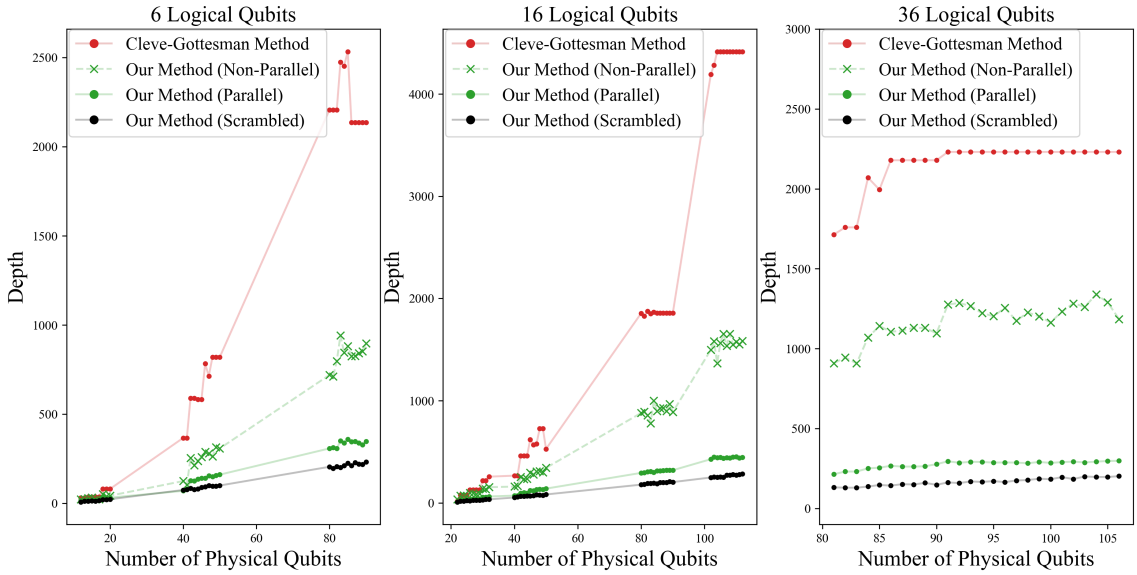


FIGURE 4.6: Depth of the encoding circuits produced by our algorithm and the Cleve-Gottesman algorithm for various error-correcting codes. Each data point corresponds to a particular code from [108, 109]. Note again that the line is included only to guide the eye - it does not represent any data. The data for our algorithm corresponds to the median cost circuit returned after 5 iterations. Here the 'Non-Parallel' data for our algorithm corresponds to a regime where the controls of the two-qubit gates encoding one stabilizer are all on the same qubit - like in the Cleve-Gottesman algorithm. Conversely, the 'Parallel' data is for a complete implementation of our algorithm - with the de-encoding $CNOT$s in each step applied such that they are maximally parallel, but also with the stabilizers in row echelon form. Our 'Scrambled' data is our complete algorithm (the 'Parallel' method), but applied to the scrambled stabilizers of the code - not in row echelon form as they were for the other simulations. Note also that we removed single-qubit gates from our calculation of depth - in accordance with our point from earlier regarding the single-qubit gates in a circuit being entirely hardware dependant. With the single-qubit gates included however, the depth data for our algorithm could increase by at most $m+1$ because single-qubit gates occur after each round of two-qubit gates that encode a stabilizer and before the first round, and the depth data of the Cleve-Gottesman method's data would vary by at most m as they have single-qubit gates only before each round. Here $m = n - k$ is the number of stabilizer generators in a given code, and n (k) is the number of physical (logical) qubits in that code. Lastly, unless otherwise stated, the main text's analysis of depth refers to the 'Scrambled' data because, again, it best reflects our algorithm's performance.

these additional gates.

Another observation from Fig. 4.5 is that the reduction in depth was generally greater for codes with more physical qubits. This was as hypothesized, because these larger codes generally had higher weight stabilizers, and hence benefited the most from our ('Parallel' and 'Scrambled') method's locally-logarithmic-in-weight depth scaling - as opposed to the locally-linear-in-weight depth scaling of the Cleve-Gottesman method and our 'Non-Parallel' method.

Also note that in the plot of encoding circuits' depths, Fig. 4.6, we include three different

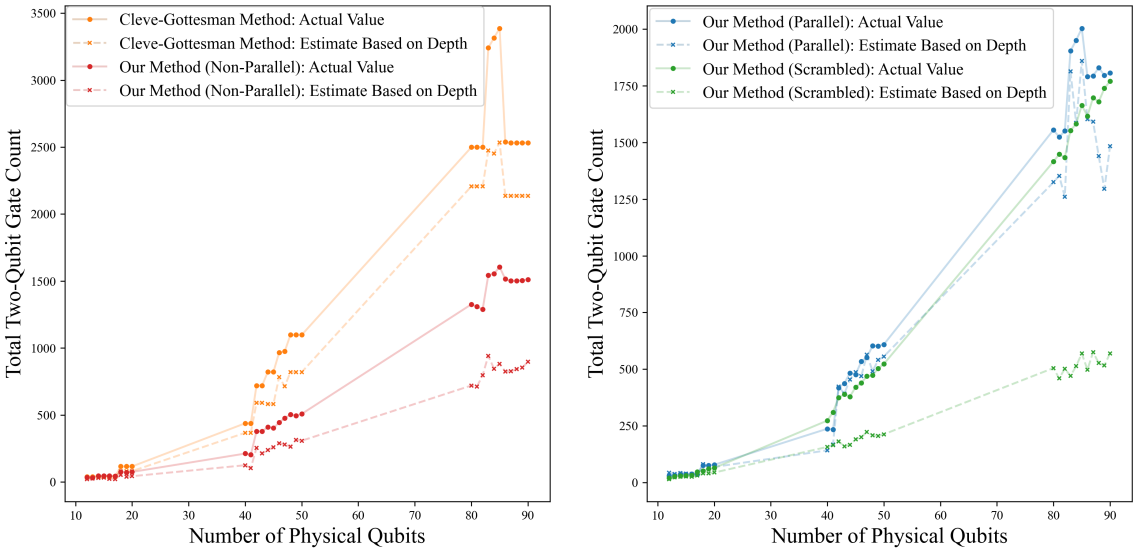


Figure 4.7: Data on the expected relationships between the two-qubit gate counts and circuit depths - as an internal consistency check. All the data here is for the codes with 6 logical qubits from [108] - but similar relations hold for the other codes too. The left plot has the data for the Cleve-Gottesman method and our algorithm ('Non-Parallel'). In this plot, the 'Actual Value' is the two-qubit gate count data from Fig. 4.4, and the 'Estimate' is the depth data from Fig. 4.6. The right graph corresponds to the 'Parallel' and 'Scrambled' data. Similarly, the 'Actual Value' is just the two-qubit gate count. However, as explained in the main text, the 'Estimate' in this plot is $m \times 2^{D_{EC}/m}$ where m is the number of stabilizer generators in the code and D_{EC} is the encoding circuit's depth.

sets of data points for our algorithm: 'Parallel', 'Non-Parallel', and 'Scrambled'. The 'Parallel' data is for our algorithm as described earlier, where, per Subsection 4.2.3, the two-qubit gates in de-encoding a stabilizer are made maximally parallel, like in Fig. 4.2b and Fig. 4.2c for instance. This is in contrast to the 'Non-Parallel' data, which instead implements these de-encoding two-qubit gates as all being controlled on the same qubit, shown for example in Fig. 4.2a. We included the 'Non-Parallel' data here because it demonstrates the reduction in depth our algorithm gains solely from requiring fewer two-qubit gates. That is, because the 'Parallel' (and 'Scrambled') data shows a substantial further reduction in depth, it explicitly suggests that actively maximising how parallel our operations are indeed significantly reduces the circuit depth, and that this reduction in depth is not entirely due to the fewer number of two-qubit gates - as expected. We also included the 'Scrambled' data because it represents how we intend our algorithm to be implemented: with no initial row reduction. Lastly, we want to point out that, when comparing the 'Parallel' and 'Scrambled' depth data, the additional reduction in depth achieved by the 'Scrambled' implementation is consistent with it using fewer two-qubit gates, per Fig. 4.4 - which generally correlates to a shorter depth circuit.

To further check that the implementation of our algorithm behaved as anticipated, we examined the hypothesized relationships between the encoding circuits' depths and their two-qubit gate counts. That is, we used the depth data and how our algorithm is expected

to perform to estimate the two-qubit gate count. We first looked at these relationships for the Cleve-Gottesman algorithm and our 'Non-parallel' method. Because in each of these approaches the two-qubit gates that encode a given stabilizer cannot act in parallel, we expected that the depth of these circuits is roughly the same as the number of two-qubit gates. This is reflected in the data, as can be seen on the left side of Fig. 4.7, or by comparing the shape of the depth data for the Cleve-Gottesman algorithm and for our 'Non-Parallel' method in Fig. 4.6, to the respective data for the number of two-qubit gates in Fig. 4.4. Moreover, see from Fig. 4.7 that this depth data ('Estimate') is strictly less than the respective number of two-qubit gates data ('Actual value'). Why this holds (and why the depth data is not closer to the two-qubit gate count data) is because one can perform, in parallel, any gates encoding consecutive stabilizers that act on some disjoint set of qubits - which reduces the depth. This explains both why the depth data is less than its respective two-qubit gate count data, and it explains why there are some deviations to the shape of the depth data as compared to the two-qubit gate count data - some circuits will manifestly be more parallel in this way. For example, one such deviation that is particularly noticeable is in the code with $k = 6$ logical qubits and $n = 47$ physical qubits, which has a drop in depth, Fig. 4.6, relative to its location to the other codes around it in the two-qubit gate count data, Fig. 4.4. Speaking generally again, because the shape of the depth and two-qubit gate count data is so similar, it suggests that usually there are not a huge number of parallel operations in these methods, as hypothesized. Also, observe that this trend seems to be a little less accurate in larger codes, and we expect that this is the result of these codes having more stabilizers, and hence having more opportunities to parallelize the encoding. Moreover, notice that the noise in our algorithm's 'Non-Parallel' data is likely from the random selection that breaks equal cost ties. See that the noise around these points seems to grow for larger codes, as one might expect because a larger code has more possible ties to be randomly broken, and hence potentially the ability to obtain a larger variety of encoding circuits with different depths. Observe also that this same noise feature is not as noticeable in the two-qubit gate count data, Fig. 4.4; probably because the cost function will actively try and keep the two-qubit gate count around the average, while the depth is only passively accounted for in the cost by virtue of having a dependence on the number of $CNOT$s in de-encoding each stabilizer.

Now consider that the shape of the 'Parallel' and 'Scrambled' depth data for our algorithm does not fit this trend of approximately matching the two-qubit gate count data. This is expected because, in the 'Parallel' and 'Scrambled' cases, the depth of de-encoding each stabilizer no longer scales linearly with the number of two-qubit gates required, instead it scales logarithmically. More specifically, the number of two-qubit gates is roughly equal to the weight of each stabilizer being de-encoded, w, and the depth of de-encoding such a stabilizer is $\log_2(w)$, as discussed in Subsection 4.2.3. We verify this by taking 2 to the power of the average ('Parallel' and 'Scrambled' data) depth per stabilizer - giving $\sim 2^{\log_2(w)} = w$, and we multiply this by the number of stabilizer, m, to give an estimate of the total weight of the stabilizers and hence an estimate of the two-qubit gate count. We show this rough correlation on the right side of Fig. 4.7. See that the 'Parallel' estimate is close to the data. This is probably because these circuits are already very parallel by construction, and hence not as many of their gates can happen to fit in parallel with others; and because the stabilizers

for this data set were inputted in row echelon form, which would likely cause the majority of the two-qubit gates to act on the same qubits - further preventing the reduction in depth from gates fitting into previous rounds. This is corroborated by the 'Parallel' estimate being far greater than the 'Scrambled' estimate - which was based on data where the stabilizers were not in row echelon form, and so their encoding gates were more likely to be applied across many more qubits and hence were more likely to fit in parallel with the other gates. Further evidence for this is that the 'Scrambled' estimate is much lower than its actual value - because this slotting of gates into previous rounds brings down the depth per stabilizer, and hence, when raised to the power of 2, will potentially give a much smaller value than 2 to the power of the actual/larger depth (of encoding each stabilizer). In particular, notice that the discrepancy between the estimate and actual value for the 'Scrambled' data grows with the size of the code - which also fits this explanation because larger codes have more operations that can happen to act on a disjoint set of qubits and thus to be performed in parallel. Lastly, we note that the 'Parallel' and 'Scrambled' estimate is not always necessarily less than the actual value, unlike in the Cleve-Gottesman and 'Non-Parallel' data. This is because the estimate here relies on an average and involves a power of 2 - which exacerbates inaccuracies in the estimate.

Because these relationships between depth and two-qubit gate count hold as expected, we have found further evidence that our algorithm is behaving correctly. We also note that these correlations could maybe be useful to get rough estimates/bounds for the circuit depth given the two-qubit gate count or the two-qubit gate count given the depth.

To understand the distribution of two-qubit gate counts in the encoding circuits produced by our algorithm, we include Fig. 4.8. It shows the number of $CNOT$s used to encode the $k = 16$ logical qubit and $n = 105$ physical qubit code in [108, 109] for 4000 iterations of our algorithm ('Non-Parallel'). Recall that iterations of our algorithm produce various encoding circuits because equal cost ties are broken by random selection. As can be seen by the distribution of these data points, shown on the right of Fig. 4.8, the two-qubit gate counts are generally concentrated near their average. That is, the median number of two-qubit gates required in this example is 2581 gates, while the maximum is 2684 gates and the minimum, which appears to be a particularly good minimum, is 2450 gates. As such, we are unlikely to get three exceedingly good or bad circuits after just five iterations. Hence, all the data we have presented above - which was run for just five iterations with the median number of two-qubit gates selected as the data point, should represent a circuit that could be expected from a typical run of our algorithm. This is corroborated by the results shown in Fig. 4.4 and Fig. 4.6 because the flat sections of the Cleve-Gottesman method correspond to fairly flat sections in the data for our algorithm (unless otherwise explained, like in the case of the 'Scrambled' data) - because the circuits chosen from our algorithm are not exceedingly good or bad. Note also that this is why we did not include any other data on the distribution of results - for all implementations of our algorithm, the gate count and depth data for each code is near its respective median. Going back to the distribution of two-qubit gate counts for a given code, Fig. 4.8, see that the maximum number of two-qubit gates found, 2684 gates, is still well below how many two-qubit gates are required for this code using the Cleve-Gottesman algorithm: 4926 gates. Noting that similar relations held for the other codes we looked at, this data affirms the conclusion that our algorithm generally provides a

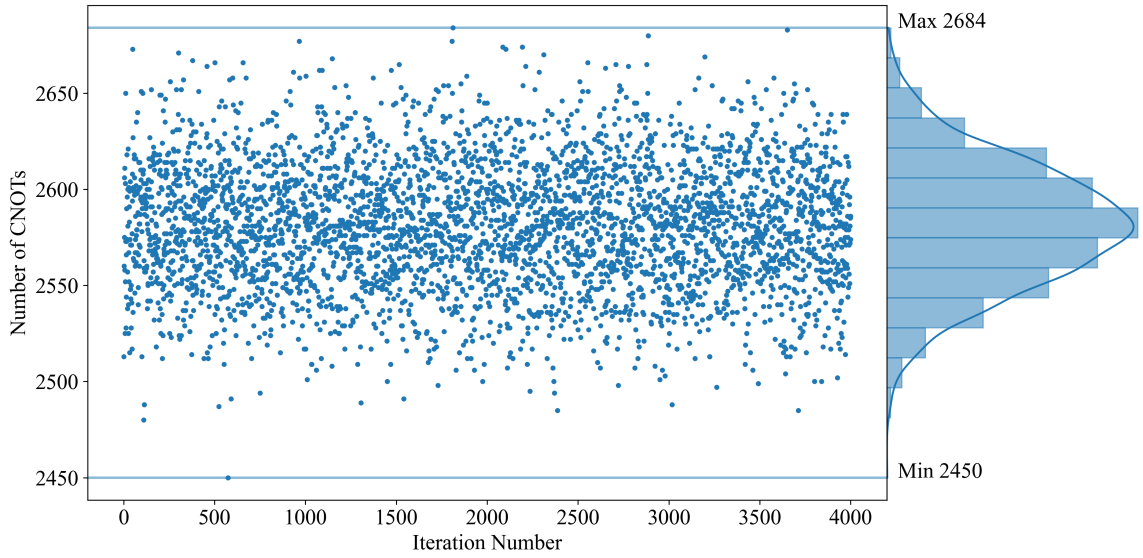


FIGURE 4.8: The two-qubit gate counts of various encoding circuits produced by our algorithm ('Non-Parallel') for the code with $n = 105$ physical qubits and $k = 16$ logical qubits in [108]. On the left, each data point corresponds to one encoding circuit's $CNOT$ count. Then on the right we have a box plot binning the data, and a probability density function - both to show how the two-qubit gate counts are distributed. Here the median $CNOT$ gate count is 2581 gates. Observe that all the data is relatively close to this median value. Also see that according to the distribution, we have found a relatively good minimum $CNOT$ count. Finally, as discussed in the main text, similar results were found for the other codes and data types tested; the depth and two-qubit gate count data for the 'Parallel', 'Non-Parallel', and 'Scrambled' methods.

significant reduction in the number of two-qubit gates - as in most cases, even the highest two-qubit gate count circuit our method produces - after many iterations - is still well below the benchmark set by the Cleve-Gottesman algorithm.

The data in Fig. 4.8 also indicates that if one wants an exceedingly good circuit relative to the other circuits produced by our algorithm, it may take many iterations. However, it should not matter if one uses just a typical encoding circuit produced by our algorithm - it will generally not be that different from the best circuit our algorithm can find and it will still generally be a substantial improvement to the analogous circuit found using the Cleve-Gottesman method.

4.5 Comments on Other Results

Here we would like to comment on some of the miscellaneous results we collected that are tangentially related to our aim. First, we compared the performance of an early version of our algorithm to the optimizations of the Cleve-Gottesman method for CSS codes that were presented in Subsection 3.2.2 and Subsection 3.2.3. In particular, we checked our algorithm against the benchmark set in [99], and found that our method performed worse. However, this was only for one code and could have been expected because, as explained earlier, these

optimizations of the Cleve-Gottesman method gain improvements by exploiting properties unique to the encoding circuits of CSS codes, and so our algorithm - which applies to all stabilizer codes and hence cannot make such assumptions - misses out on these improvements. Additionally, we again note that, in general, these optimizations have poor time complexity. Thus, even for CSS codes, our algorithm could still be useful - especially in larger CSS codes.

Also, in the interest of finding a general method for constructing encoding circuits that are efficient in tensor network decoders, we considered preparing CSS codes using entangling measurements of *cluster states*; highly entangled states. In previous work [86] we proposed a general method for preparing CSS codes in this way. However, when converting these encoding 'circuits' to a graph - which is effectively the form it will take in a tensor network decoder - and calculating the treewidth, we found that both the $[[46, 2, 9]]$ and the $[[48, 6, 8]]$ codes from [33] would each yield two instances of a graph with treewidth 19 - where we used both the minimum degree heuristic and minimum fill-in heuristic [87]. So with the time cost of contracting a tensor network generally scaling exponentially with treewidth, we have found that this is not a viable approach for tensor network decoding [78].

5
Conclusion

In this work we aimed to reduce the overhead of encoding arbitrary quantum error-correcting codes. We addressed this by designing an algorithm that returns ancilla-free encoding circuits with reduced two-qubit gate counts and reduced depths - as compared to the literature's most prominent method for producing such circuits: the Cleve-Gottesman algorithm.

Our algorithm works by trying to encode just one stabilizer at a time. In so doing, unlike the Cleve-Gottesman method, while our algorithm is encoding one stabilizer, it could happen to encode portions of other stabilizers. We hypothesized that by implementing a cost function we could better leverage these free encodings of other stabilizers and hence reduce the gate count. Moreover, while a reduced gate count generally correlates to a reduction in depth, we suspected that we could further reduce the depth by scheduling the $CNOT$s so that they were maximally parallel locally. We believed this would improve on the Cleve-Gottesman approach because their method was restricted in where it could apply two-qubit gates - and thus its circuits were severely limited in how parallel they could be.

To test these hypothesis we ran numeric simulations that took in various codes' stabilizers and returned their encoding circuits - as found through our algorithm and the Cleve-Gottesman algorithm. For each code, the overhead - two-qubit gate count and circuit depth - of its encoding circuits were compared. We assessed the algorithms for various random codes whose number of qubits were within the realm of what can be expected in near term quantum devices - where an ancilla-free regime is most important.

For the codes tested, we found that our algorithm reduced the two-qubit gate count by $\sim 41\%$ on average, and reduced the encoding circuit's depth by $\sim 87\%$ on average - relative to the encoding circuits produced by the Cleve-Gottesman algorithm. Moreover, we saw a reduction in gate count by up to $\sim 57\%$ and reductions in depth of up to $\sim 95\%$. Additionally, this improved efficiency of our algorithm's encoding circuits was generally most evident in the larger codes we tested: up to $n \sim 100$ physical qubits with $k \ll n$ logical qubits. It is also worth pointing out that our algorithm is able to account for the difficulty

of long range entangling operations - through its selection of which $CNOT$s to apply, and can reduce the single-qubit gate count by including single-qubit gates in the cost function. However, no data was generated for these points because they are both highly hardware dependant. Moving on, with such favourable reductions in the two-qubit gate counts and circuit depths, we believe our project's main aims were achieved. We had also aimed for our algorithm's time complexity to not differ greatly from that of the Cleve-Gottesman algorithm. We believe that we have also satisfied this aim too, as generally our algorithm will have the same time complexity as that of the Cleve-Gottesman algorithm; specifically, the time complexity will usually be $\mathcal{O}(m^2 n)$ where m is the number of stabilizers and n is the number of physical qubits in the code. We qualify this statement with 'generally' because in regimes when $m < \log_2(n)$ or $m < \sqrt{n}$, our algorithm's complexity will be slightly worse. However, these conditions are generally only satisfied by quantum codes with poor error-correction capabilities.

To summarize, we believe that all of our project's aims were achieved. Indeed, our algorithm generally outperforms the literature's most prominent method for finding ancilla-free encoding circuits, and usually does so in the same asymptotic time.

We would also like to note that our work shifted slightly from the original proposal. We had initially aimed to implement a tensor network decoder. Although not explicitly stated in the proposal, the challenge we were to focus on was producing an algorithm, like we have, that would supplement a tensor network decoder by finding efficient encoding circuits for a given code - with the idea that such an algorithm would, in part, address the difficulties regarding the runtime of tensor network decoders. Midway through our research, we realised that actually programming a tensor network decoder would not add anything novel to our work, and thus that our time was better spent on improving our algorithm. Moreover, this also made sense in the context of our algorithm not just potentially being useful for tensor network decoders, but also for preparing quantum error-correcting codes more generally. With this latter purpose also benefiting from an improved algorithm, it affirmed our decision to focus on further developing our algorithm - as opposed to spending our time implementing a tensor network decoder.

The remaining text will discuss suggestions for further work on the ancilla-free preparation of error-correcting codes. We will break this into three parts: the first will focus on improving our algorithm, the second will look at adapting our algorithm to specific problems, and the third will discuss other approaches for preparing codes.

While we believe that we have exhausted all the biggest improvements to our algorithm, we do have some suggestions for further work that may yield benefits. The first of which is to add CZ and CY gates to our algorithm by applying them in a similar way as the Cleve-Gottesman method. Another idea that might have a similar effect is to add a basic optimizer that commutes the single-qubit gates around our encoding circuits to see if they can cancel with other single-qubit gates in the circuit. However, while implementing either of these changes might be an improvement in a general sense that it allows for a fairer comparison of single-qubit gate counts, it is still somewhat unfounded in that for any practical implementation, one would want to know the cost of encoding in terms of operations that are native to some hardware. As such, an interesting change would be to replace the gates in our algorithm with a set of gates native to a particular architecture. That way a more accurate

- hardware specific - cost could be found. Note that for simplicity these gates should still be Clifford gates. It would also be interesting to use this hardware specific implementation of our algorithm to find the encoding overhead of various codes and compare them - with the idea that perhaps some codes are more suitable to be encoded using a particular gate set.

Another improvement would be to include a calculation of depth into the cost function. In particular, keep a running total of where gates have been applied so that when de-encoding a given stabilizer, one could know how many of these new gates would add to the depth - as some of these de-encoding gates may fit between previously applied gates. This would give a more complete cost function, which should translate to encoding circuits with properties reflecting said cost: lower depths.

Similarly, we could use the cost function to further address the difficulties regarding long range entangling operations and qubit connectivity. For instance, in a linear array of qubits, a stabilizer X_1X_{300} may otherwise seem attractive to de-encode because it only requires a single $CNOT$, but because the qubits are so far apart perhaps the cost function is not accurate, and maybe it is better to wait and see if through other operations, this stabilizer could change to not requiring such long distance gates - which a more sophisticated cost function could account for. Additionally, one could apply more $CNOT$s than necessary to close the physical distance between qubits or look at how to relabel the qubits in a code in order to reduce these distances. Also to this end, it might be worth implementing a method that varies stabilizers by other stabilizers i.e. takes products of stabilizers. This could be used to reduce the weight of stabilizers - leading to them requiring fewer gates to de-encode, and also could help reduce the formerly mentioned issues with entangling gates. It would also be interesting to test the performance of our idea from Section 4.2.3, which accounts for these difficulties regarding long range multi-qubit gates and qubit connectivity through how it applies $CNOT$s - see Fig. 4.2c.

Another idea would be to allow for each stabilizer to be de-encoded to a Pauli-Y or Pauli-Z instead of Pauli-X. This could potentially yield very different circuits - some of which could have a lower overhead. We note that it does not necessarily make more sense to immediately de-encode stabilizers to Pauli-Zs with the idea that then it would require fewer Hadamards at the end, because the circuits produced will likely be completely different and hence they will not necessarily have a lower overhead. Another recommendation for further work is to take a similar approach as the overlap method in Subsection 3.2.3, by incorporating in the cost some measure that quantifies how much de-encoding one stabilizer will reduce the weight of the code's other stabilizers. That way, these gates will de-encode/encode more of the other stabilizers for free, and hence reduce the encoding circuit's global cost.

In terms of other problems where our algorithm may be useful, one could look at adapting our cost function and parallelization technique to the similar algorithm for qudits in [95, 96]. Additionally, one might want to research how our algorithm could be adapted to de-entangle codes into smaller sub-codes. More trivial work could also look at running our algorithm for various other codes and investigating how our algorithm performs for larger codes or codes with specific properties - like low-density parity-check codes [33, 34]. Finally, towards improving our algorithm specifically for the purpose of tensor network decoding, it would make sense to represent the encoding circuit as a graph and to add a substantial cost for any loops in the graph. This way, iterating through our algorithm would yield circuits with

fewer loops, which would help address a major drawback of tensor network decoders - their runtime.

We also have some recommendations for research into preparing quantum error-correcting codes more generally. The most important suggestion would be to consider how to prepare these codes while accounting for mid-circuit noise - such that the errors don't propagate through the gates into errors too large to correct. Schemes like this are said to be fault-tolerant - see Subsection A.8 for a more precise definition - and are not well developed for encoding. Currently, there exist methods for guaranteeing fault-tolerance that work by preparing noisy copies of a code - using encoding circuits like those produced by our algorithm, and applying gates between them and making measurements to verify if one is indeed a codestate [49, 50, 99]. Moreover, in keeping with minimizing the ancilla usage, these verification techniques seem practical; they trade requiring an undefined number of iterations before a suitable state is found for using fewer qubits. Additionally, this trade off is particularly suitable for a regime with few qubits because the probability of mid-circuit errors is not as high for smaller codes and hence, generally, fewer iterations will be required before a robust state is prepared and verified. However, these techniques currently only apply to CSS codes. As such, an important direction for further research would be to extend these ideas to non-CSS codes too. To this end, one might consider noting that this verification procedure is similar to Steane syndrome extraction - which also only works for CSS codes [56]. Hence one may want to try adapting Knill syndrome extraction in a similar way [57, 58, 110]. It is important to point out that such research should keep overhead in mind. Another thing to consider is that verification requires each noisy codestate to be produced by a unique encoding circuit - to reduce the chance that these encoded states have the same correlated errors. Recall that our algorithm does this manifestly by breaking equal cost ties using a random selection.

Other research on preparing codes could include trying to achieve encoding circuit optimizations for arbitrary stabilizer codes through more analytic means. For this, one might try to generalize Steane's Latin rectangle method - Subsection 3.2.2 - or Paetznick's overlap method - Subsection 3.2.3 - to all stabilizer codes. We do note that this is likely a difficult problem - especially while trying to keep time overhead in mind, because the gates in the circuits of non-CSS codes are not as easily commuted around as they are in CSS codes. Lastly, because our algorithm saw such a large improvement over the Cleve-Gottesman algorithm, it would be interesting to compare the overhead of the ancilla-free circuits produced by our algorithm to other encoding circuit algorithms that are not necessarily ancilla-free. The idea here is that our encoding circuits could also be considered efficient in regimes where ancilla qubits are not so scarce that they dominate the cost of computing - where there are on the order of 1000s of qubits or more for instance.

A

Supplementary Material for the Quantum Error Correction Review

A.1 Introduction to Classical Error Correction

The practicality of classical information relies on a physical implementation. As such, it is important to consider the consequences of a non-ideal hardware that interacts with its environment. That is, we must entertain the possibility of *errors*; unintended operations that corrupt information, or more precisely, a mapping from the logical state that would be expected in an ideal case to a different, logically distinct state. For instance, perhaps an error maps '100001' $\rightsquigarrow$ '000001'. To formalise this idea consider two single bits, '0' and '1', and see that the only possible errors would be to map '0' $\rightsquigarrow$ '1' and '1' $\rightsquigarrow$ '0' respectively. As such, this type of error is termed a *bit-flip* and because, by definition, any information-corrupting process can be decomposed into bit-flips, it is the only type of error that can occur in classical information. So, as an example, the error '100001' $\rightsquigarrow$ '000001' corresponds to a bit-flip on the first bit. On a hardware level, because a bit is represented by the state of a classical two-level system - each level corresponding to a binary logical state - a bit-flip error corresponds to a physical transition between the two levels. The change of a 'signal' to a 'no signal', or vice versa, for instance.

In keeping with the 'signal-no-signal' representation of a bit, all sources of errors, irrespective of their origins, are defined as *noise*. To give an example that demonstrates the importance of considering noise, consider the following true story. In parts of Belgium, voting can be done electronically. The way this works is, on a computer, each voter selects the candidate they would like to vote for and then their selection is saved to the computer and on a separate magnetic memory. Well, in a 2003 election using this system, there was an inconsistency with the number of votes cast and the number of votes possible. To investigate this, authorities recounted the votes using the separate magnetic memory, and a discrepancy

of 4096 votes was discovered. After an investigation, it was found that the inconsistency was most likely caused by a single bit-flip error. One of the main pieces of evidence for this conclusion being that 4096, the number of votes in question, is a power of 2, specifically 2^{12}, which is consistent with a bit-flip error on the bit-string's 13th bit, as per how positive integers are represented in binary [6, 81]. This example illustrates how even just a single error can significantly corrupt the output of an important computation, and so it demonstrates the importance of protecting information from errors.

Because physical devices are necessarily coupled to an external environment and are built from imperfect components, hardware improvements alone do not allow for the arbitrary suppression of errors. So, while passively reducing noise through hardware improvements is helpful, an additional technique is required if information is to be protected from errors, motivating the study of *error correction*; active procedures to protect information from errors.

The core mechanism behind error correction is to delocalize information, the idea being that then localized errors are unable to corrupt the information. This process of spreading some information across additional bits is called *encoding*. To see how error correction works, first recall from before that a bit suffering from a single error appears as follows '0' $\rightsquigarrow$ '1' or '1' $\rightsquigarrow$ '0', as summarised in the second and third rows of Tab. A.1. In this case, there is no differentiating the possible received signal from the possible signals sent, so there is insufficient information to identify an error. Now, to improve this situation, we could use the simplest example of encoding: sending multiple copies of a single bit. In this protocol we could encode by repeating one bit of information across two bits: '0' $\rightarrow$ '00' and '1' $\rightarrow$ '11'. Then if there is a single error, the signal received would be either '01' or '10'. With reference to the fourth and fifth rows of Tab. A.1, we can see that the states received with a single error are easily distinguished from the ideal signals sent. As such, we can detect if a single bit-flip error has occurred. However, still referring to the fourth and fifth rows of Tab. A.1, because the signals received with a single error are degenerate for non-equivalent errors, there is not enough information to identify which bit the error occurred on, and so the original signal cannot be un-corrupted. To resolve this, consider encoding by repeating one bit of information across three bits: '0' $\rightarrow$ '000' and '1' $\rightarrow$ '111'. Then, from looking at the sixth and seventh rows of Tab. A.1, it can be seen that the all the ideal signals that were sent and all the the states received with a single error are distinguishable. Hence, not only can an error be detected, but its location can be identified; and so, by flipping the identified bit, the error can be corrected. For example, say that you were using this *3-bit repetition code* and were guaranteed that at most one error could occur, well if you were to receive the errored signal '010' you could take the majority vote of the bits to find that the un-corrupted signal sent was '000' - corresponding to the bit of information: '0'. Note that while this procedure may raise the question of what to do in the case of multiple errors, we save the discussion of this issue for Section 2.1.

While we have only introduced classical error correction, it is worth noting that the concepts we have discussed in this section are what form the basis of the technologies that allow us to use lightly scratched disks, scan damaged QR codes, and reliably use WiFi and 5G in confined spaces - to name but a few examples [72]. Also, hopefully this introduction to classical error correction serves as an intuitive reference point for the topics to come,

Signal Sent	Bit-flip on Bit	Signal Received
'0'	1	'1'
'1'	1	'0'
'00'	1	'10'
	2	'01'
'11'	1	'01'
	2	'10'
'000'	1	'100'
	2	'010'
	3	'001'
'111'	1	'011'
	2	'101'
	3	'110'

Table A.1: How single bit-flip errors act on the three smallest classical repetition codes. The rows are grouped into three blocks, each block corresponding to the 1-, 2-, and 3-bit repetition codes respectively. The first column, signal sent, represents the encoded bit of information before it potentially encounters an error. The second column describes every possible single bit-flip error, and the third column, signal received, explicitly shows how the signal is effected by each of these errors.

because, as we will discuss later, much of this Chapter is the generalization of these classical concepts to quantum systems.

A.2 Introduction to Quantum Noise

Classical devices are often inherently error-resistant. For instance, consider classical computers, where bits are often represented by the charge or absence of charge of many electrons. While noise might cause, say, some of these electrons to be lost, it is incredibly rare for enough of the electrons to be affected so as to significantly change the total charge and cause a *logical error*; corrupting the information. Specifically, in DRAM - a classical memory, it is estimated that a single bit will go on average 1.6 to 4.6 million years before it experiences a bit-flip error [7]. Note that this error rate is small relative to the run-time of, and number of bits needed in, classical computations. That is, these errors are often negligible [2]. In comparison, quantum computers are far more delicate. As an example, while thermal noise is not a significant issue in classical hardware, it is a major issue in most quantum devices, as evidenced by fridges making up the bulk of most quantum architectures. Quantum computers are so fragile because, by the very nature of needing to deal with quantum systems, they often realize a qubit as the quantum state of a single particle. And so, left unchecked, an error on that one particle will corrupt its stored information. Moreover, quantum devices are often more error-prone because difficulties in controlling qubits makes them more susceptible to non-environment noise. This greater sensitivity to noise in quantum systems can be seen experimentally. In particular, sitting idle in memory, it will take on average 9.1 seconds before an error occurs on an ion trap qubit and 10.0 milliseconds before an error

occurs on a superconducting qubit [8]. To show that qubits are less robust against noise, we could compare these qubit error rates to the millions of years that pass - on average - before an error on a bit in DRAM occurs. However, it is a naive comparison because it does not account for the time needed to perform quantum computations and how these computations can significantly increase the error rate. Additionally, note that error rates in quantum devices are related to the computation times. This is because the error rate and the time required to perform a quantum operation – which requires an external interaction, both depend on the degree to which a qubit is coupled to its environment [2]. This can be seen experimentally by looking at the time to perform *quantum gates*; operations common in quantum computing that perform a desired logical operation on a quantum state, some of which are provided in Appendix C. For instance, a $CNOT$ gate takes $\sim 120,000$ nanoseconds on an ion trap device and ~ 22 nanoseconds on a superconducting device. Moreover, we can use this information to improve our estimate of how destructive noise is in quantum computers by considering computation times - approximated using the amount of time required to perform gates and the number of gates needed in a quantum algorithm. For example, using data on the number of gates required [8] for Shor's algorithm - an example of a quantum algorithm with significantly lower time complexity than its classical counterpart - to factor a 1024 bit long integer, assuming no gates are performed in *parallel*, we can calculate that it would take approximately on the order of days in an ion device and on the order of seconds in a superconducting device. Comparing these computation times to the average times before an error, on the order of seconds and milliseconds respectively, reveals the true destructive nature of noise in quantum computing.

The above estimate of how destructive noise is in quantum computing is based on error rates for qubits stored in memory. However, in a computation - with gates - there would be significantly more errors, worsening the outlook of quantum computing without error mitigation. This is because gates do not act ideally; during the time it takes to perform a gate, environmental noise can cause errors, and also, difficulties in controlling qubits can cause experimental errors [9]. To quantify these gate error rates, one calculates the gate's *fidelity*; the probability that a qubit acted on by the non-ideal version of the gate passes as a qubit acted on by an ideal version of the gate. Fidelities achieved in current quantum devices that are considered high are $\sim 99.99\%$ for single-qubit operations, and $\sim 99.9\%$ for two-qubit operations [10, 11, 13]. With such high percentages, these fidelities may seem near ideal. However, when you consider that without error correction one error corrupts the information and that computations require performing many gates, on the order of 10^9 gates in the previous example of Shor's factoring algorithm, it is clear that useful computations cannot be made without also suppressing gate errors. As a side-note, with reference to the fidelities provided above, we can see evidence that a gate's fidelity is indeed dependant on the time required to perform the operation, because generally two-qubit gates take longer than single-qubit gates [8]. Also, it is worth noting that the difficulty of performing multi-qubit gates grows with the range of the interaction [12, 13].

To get more insight into quantum errors, consider that *incoherent noise* causes errors probabilistically, and hence, under incoherent noise, a pure state will become mixed. Particularly, the state after applying incoherent noise will be an error rate dependant probabilistic combination of the original state and errored states. Note that this is often equivalently

Name of the Operation	Matrix Representation	Action on a Qubit Basis State
Identity	$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$I\|k\rangle = \|k\rangle$
Pauli-X / Bit-flip Error	$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	$X\|k\rangle = \|k \oplus 1\rangle$
Pauli-Z / Phase-flip Error	$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$Z\|k\rangle = (-1)^k\|k\rangle$
Pauli-Y	$Y = iXZ = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	$Y\|k\rangle = i(-1)^k\|k \oplus 1\rangle$

Table A.2: Summary of the identity operator and Pauli operators in the z-basis. Here i is the imaginary unit, $k \in \mathbb{Z}_2$, and $\oplus$ is addition modulo 2.

stated as: in a process called *decoherence*, noise causes the mixing of pure states – and in this context, pure, superposition states are regularly called *coherent states*.

Before proceeding, consider that because qubits are realized in what are roughly two-level quantum systems, and because they have some relatively weak coupling to an environment that has a continuum of states, a qubit's state will decay approximately according to Fermi's golden rule. That is, a qubit's state is expected to decay exponentially in time, where the exponential can be written as $e^{-t/T}$ where t is time and T is a constant called a *coherence time* [14]. Using this, we can describe how noise is generally quantified in the literature; through coherence times or *relaxation times* denoted T_1, T_2, and T_2^*. In particular, T_1 is the exponential-decay time constant for a state $|1\rangle$ to no longer be measured to be in the state $|1\rangle$ [14]. It can be thought of as the lifetime of a classical, bit-like quantum state [2]. On the other hand, T_2 and T_2^* are called *dephasing times* and they quantify the lifetime of superposition states [2]. Specifically, T_2^* is the exponential-decay time constant for the Ramsey experiment; the decay constant for the state $|\pm\rangle = \frac{1}{\sqrt{2}}(|0\rangle \pm |1\rangle)$ to evolve into an equal probability mixture of $|+\rangle$ and $|-\rangle$ [14]. Meanwhile, T_2 is also an exponential-decay time constant but for the Hahn Echo experiment; a similar but often more complicated process. The Hahn Echo experiment is not as instructive and so its details will be omitted, but it can be read about here [15].

To see why these two types of coherence times are important in discussing quantum errors, we must go beyond just saying that errors corrupt information, and instead give them an explicit form. Experimental results reveal that small incoherent errors are accurately approximated as probabilistic *Pauli errors*; errors represented by the set of Pauli operators - where an *operator* defines a matrix in a given basis [16]. In Tab. A.2 we provide the types of Pauli errors and how they act on qubits. Observe that, in analogy to classical errors, a Pauli-X error acts like a quantum bit-flip. This is of course in someways encouraging as classical error correction provides methods for correcting bit-flip errors. However, because qubits have the unique property of a phase between their basis states, and because this relative phase is observable - defining some information, it makes sense that qubits can also experience phase errors. And indeed, unique to qubits are phase-flip errors, which leave $|0\rangle$ unchanged

and map $|1\rangle \rightsquigarrow -|1\rangle$. Because phase-flip errors do not have a classical analogue, they are not addressed by classical error correction, and so they may be cause for concern. The last type of error that needs to be considered is a Pauli-Y error, which, up to an unobservable global phase, is equivalent to a bit- and phase-flip error. Hence, any small incoherent error can be accurately approximated as independent bit- and phase-flip errors. Accordingly, to accurately correct any small incoherent errors, quantum error correction needs only to independently correct bit- and phase-flip errors. With this information, we substantiate why just the two coherence times, T_1 and T_2^*, are good parameters for quantifying errors: they describe the likelihood of bit- and phase-flip errors respectively.

However, this is not the entire picture, as more common than incoherent errors are *coherent errors*; small unitary transformations of the qubit's state [17]. While coherent errors may seem troublesome - because one might think that infinite precision would be required to correct a continuum of errors, they are in fact correctable, as addressed by a result called the *digitisation of errors*. The digitisation of errors states that because the Pauli operators along with the identity form a basis for 2×2 matrices, any single-qubit coherent error can be written as the unitary [2, 12, 18]

$$U(\alpha_I, \alpha_X, \alpha_Z, \alpha_Y) = \alpha_I I + \alpha_X X + \alpha_Z Z + \alpha_Y Y \tag{A.1}$$

where the coefficients $\alpha_{O\in\{I,X,Y,Z\}}$ are complex, and I is the identity and X, Z, and Y are the Pauli operators given in Tab. A.2. Again noting that $Y = iXZ$ and defining $\alpha_{XZ} \equiv i\alpha_Y$ any single-qubit coherent error can be written as

$$U(\alpha_I, \alpha_X, \alpha_Z, \alpha_{XZ}) = \alpha_I I + \alpha_X X + \alpha_Z Z + \alpha_{XZ} XZ. \tag{A.2}$$

So, just like incoherent errors, coherent errors can be represented as independent bit- and phase-flip errors. In fact, as a prelude to quantum error correction, measuring a set of operators - called *stabilizers* - projects errors onto this basis. Hence, quantum error correction only needs to correct bit- and phase-flip errors independently to correct any quantum error.

For quantum error correction, it is instructive to formalize the action of noise on qubits. For this, an important piece of information is that quantum error correction methods decohere coherent errors into probabilistic Pauli errors [19]. Now, recognising that incoherent errors can also be written as probabilistic Pauli errors - as explained above, we can see that when error correction is employed, the action of incoherent and coherent errors alike can be represented as mixing states. Thus, in this formalism, we want noise to be a mapping from density operator to density operator. That is,

$$\rho \rightarrow \rho' = \Lambda(\rho) \tag{A.3}$$

where ρ is the initial state, ρ' is the state after experiencing noise, and Λ represents the noisy *channel*; the means through which quantum information is communicated. To expand on this formalism, we must note that it is a property of density operators that they have trace 1; are Hermitian, meaning they are equal to their conjugate transpose; and that they are a positive operator - in any basis all their diagonal elements are greater than or equal to zero [2]. Accordingly, because a channel maps a density operator to another density operator,

it must preserve these properties after the mapping. A general solution for a channel that satisfies these constraints is

$$\Lambda(\rho) = \sum_i K_i \rho K_i^\dagger \tag{A.4}$$

where

$$\sum_i K_i^\dagger K_i = I \tag{A.5}$$

and K_i is called a Kraus operator [2]. Well, because we want to use this channel formalism to represent noise, and as mentioned before, noise can be represented as probabilistic Pauli errors, it makes sense to define a general noisy channel as a *Pauli channel*; a channel with Kraus operators $K_i = \{\sqrt{1-p_X-p_Y-p_Z}I, \sqrt{p_X}X, \sqrt{p_Z}Z, \sqrt{p_Y}Y\}$ where $p_{O\in\{X,Y,Z\}}$ are the probabilities of their respective Pauli error occurring in a given time step, and $1-p_X-p_Y-p_Z$ is the probability of no error occurring in a given time step. That is, an arbitrary single-qubit Pauli channel can be written as

$$\Lambda(\rho) = (1-p_X-p_Y-p_Z)\rho + p_X X\rho X + p_Z Z\rho Z + p_Y Y\rho Y \tag{A.6}$$

where we have used that the identity and Pauli's are Hermitian. Note that a Pauli channel with multiple qubits just requires a generalisation of these Kraus operators to higher dimensional Hilbert spaces - taking all possible tensor products of Pauli errors and identities and adjusting the probabilities accordingly. Some channels commonly seen in the discussion of quantum error correction are the depolarizing channel which sets $p_X = p_Y = p_Z = p$ where p is some error rate; the dephasing channel which has $p_X = p_Y = 0$ and $p_Z = p$; and the bit-flip channel which sets $p_Z = p_Y = 0$ and $p_X = p$. It is also worth noting that in real quantum devices, noise is dephasing biased. Particularly, it is often the case that [20]

$$10 \lesssim \frac{p_Z}{p_X + p_Y} \lesssim 1000. \tag{A.7}$$

Accounting for noise bias has recently become an area of interest in quantum computing as it was found that a quantum error correction protocol's performance could be dramatically improved by tailoring it to focus on biased noise [20–22]. Moreover, because of the strong performance of quantum error correction techniques under biased noise, recent work has presented experimental procedures for intentionally further biasing noise in hardware [23–25].

We conclude this section with some additional notes. First, by virtue of using Pauli channels to represent noise, it is relatively easy to simulate errors using a classical computer - a consequence of the *Gottesman-Knill theorem* [26, 27]. This offers the advantage of being able to test quantum error correction schemes without actually implementing them on real quantum devices, which will be discussed in more detail later. And secondly, as a final note on quantum noise, one can visualise, on a Bloch sphere, the state of a single-qubit evolving in time under Pauli noise. This can be done done using *Lindblad master equation*; a differential equation that approximates the time evolution of a quantum state in an open system. Also,

note that the Lindblad master equation is fairly accurate, especially in the limit of a weak interaction with the environment [2].

A.3 Threshold Theorem

While distance gives a good indication of a code's performance, it is not necessarily an accurate representation of how good a code is. A better measure would be an error correction procedure's logical error rate. To get this more complete picture, one needs to first construct an entire error correction protocol: by selecting a code, decoder, and an error model - a choice of the ratios between the Pauli error rates. Then on a classical computer, by choosing a physical error rate and only keeping track of the errors - independently considering each 'qubit' and applying an error with some probability, the syndrome can be found - by checking how these errors commute with the stabilizers of the code. By then applying the remainder of the error correction procedure and checking if the estimated error was equivalent to the initial error up to a stabilizer, over many iterations, the protocol's logical failure rate can be estimated. This can be done efficiently on classical computers because, as mentioned earlier, it is easy to classically simulate Pauli channels, and because decoding is designed to be performed on classical devices. Note that, in these Monte Carlo simulations of the logical error rate, the number of iterations required for a specific level of precision can be calculated using the standard error. That is, one could expect that running N iterations would give an uncertainty of $\sim \frac{1}{\sqrt{N}}$. These iterations could then be performed again for various physical error rates to generate a graph that looks like any one of the curves in Fig. A.1. If this procedure was repeated for different distance codes in the same *family*, we could populate our graph with additional curves, similarly to whats shown in Fig. A.1. Note that a code's family consists of all generalisations of some code to a different number of physical qubits. For example the repetition code family consists of all codes with the logical state $\alpha|00\ldots0\rangle+\beta|11\ldots1\rangle$. By adding these additional curves to the logical error rate vs physical error rate graph, we can observe the consequences of one of the most important results in quantum error correction, the *threshold theorem*.

The threshold theorem states that for an error correction procedure, if the physical error rate is below some threshold, the logical error rate of that procedure can be arbitrarily suppressed by scaling up the distance of that code [2, 12]. As such, the threshold is perhaps the most important single-figure in determine an error correction protocol's performance. As, by its definition, if a quantum error correction procedure achieves a threshold higher than the physical error rate of a real quantum device, the logical errors can be arbitrarily suppressed. That is, the logical qubits can be made arbitrarily close to ideal, unlocking reliable, large-scale, useful quantum computations. For a more visual definition, the threshold, as depicted in Fig. A.1, is the physical error rate for which all the codes in a family have the same logical error rate. Intuitively, the threshold theorem is a consequence of a higher distance code in a family requiring more physical qubits, and so, because it has more physical qubits it is more likely to experience errors. So, there comes a physical error rate of diminishing returns, where adding more physical qubits does not lower the logical error rate - as more errors are occurring than can be corrected. That is, past this threshold, the disadvantages of encoding begin to outweigh the advantages, as seen in the form of higher logical error rates. This

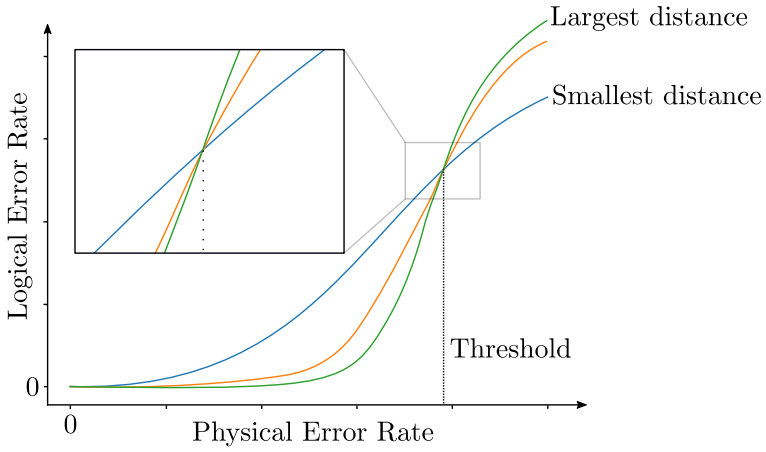


Figure A.1: Sketch of the graph expected from classically simulating the performance of a code. Each colored curve on the graph represents a different distance code in the same family. The threshold is the physical error rate at which point all codes of a family have the same logical error rate - as denoted on the graph by the dotted black line. Adapted from [44].

is why, in Fig. A.1, we see the highest distance code in the family giving the lowest logical error rate up until the threshold, after which point, it gives the highest logical error rate. As a final note, while many code families have a threshold, not all do.

A.4 Syndrome Degeneracy and Full Quantum Error-Correcting Codes

From the above section, we can recognise two issues with the 3-qubit repetition code. Its syndrome degeneracy prevents it from correcting more than a single X error, and its stabilizers commute with Pauli-Zs and so it does not correct any Z-type errors. These issues will be discussed separately. Firstly, recall that the syndrome is a string of classical bits that give information on the errors afflicting the encoded state, and that if we retrieve all the information on a qubit, its state will collapse and be analogous to a classical bit. Also, see that because of how the code's Z-like stabilizers commute, we know that they only give information on X-type errors, of which there are eight possible errors on three physical qubits, as per Tab. 2.1. As such, we can identify that because the two syndrome bits give two bits of information on these errors the probability space of errors is reduced by $2^{-2} = 1/4$ as from Shannon information

$$P_S = 2^{-I_S} \tag{A.8}$$

where P_S is the fraction by which the state space is reduced, given I_S bits of information on the system. Indeed this is why we see that each syndrome is doubly degenerate in this case; the probability space has eight errors and the syndrome reduces this state space by a quarter, leaving two errors per syndrome. Shannon information, Eq. A.8, implies that for

the syndrome to be non-degenerate - allowing us to differentiate between the eight possible X errors, three bits of information are required. The difficulty here being that obtaining another bit of information would collapse system's state to be analogous to a classical system, losing the advantage afforded by quantum phenomena. We can see this as to obtain another syndrome bit on X errors we require another Z-type measurement, however all the possible Z-type measurements either return no information, redundant information, or completely collapse the state. As such, these two syndrome bits are the limit of what can be known about the errors on this system without losing information - and this degeneracy of syndromes is a property of all stabilizer codes. Consequently, when codes are presented, they are accompanied with a parameter that can be used to quantify the minimum number of errors it can correct for certain, called the *distance* which is denoted d. Specifically, the distance relates to the number of correctable errors, t, according to the equation

$$t = \left\lfloor \frac{d-1}{2} \right\rfloor . \tag{A.9}$$

Although we save the precise definition of code's distance for later in this section, see that for a code to at least correct any single-qubit Pauli, $t \geq 1$, and be classed as a *full error correction code*, the distance must be $d \geq 3$. And this leads to the other issue that needed to be addressed for the repetition code; it does not correct Z-type errors. Before proceeding see that if Hadamards were applied to the three physical qubits after the encoding part of Fig. 2.2, the state would be stabilized by $\langle X_1X_2, X_2X_3\rangle$ and we would have the reverse problem where we can correct a single Pauli-Z error, but not any X-type errors. However, by combining these codes through encoding into one of the codes, and then encoding these already encoded qubits into the other code - shown in in Fig 3.2, in a process called *concatenation*, we are able to correct for a single-qubit X error and a single qubit Z error. In fact, this concatenated code was the first full error correcting code discovered, and is called the *9-qubit Shor code*, named after its discoverer, Peter Shor [12]. Note that the Shor code has nine physical qubits and one logical qubit because it first encodes one qubit of information across three physical qubits, and then encodes each of these three qubits across three additional physical qubits. So, to summarize the properties of the Shor code, it encodes $k = 1$ qubit of information across $n = 9$ physical qubits and has distance $d = 3$. These properties are often written more concisely in the form $[[n, k, d]]$, where, for instance, the Shor code is a $[[9, 1, 3]]$ code.

Finally, note that some codes can correct specific errors of wight greater than their respective distances. In fact, one can check a given code's ability to correct any specified set of errors using the *quantum error correction condition* - which, more generally, provides a necessary and sufficient condition for quantum error correction, as presented in Appendix A.5.

A.5 The Quantum Error Correction Condition

An important result regarding quantum codes is the *quantum error correction condition* which offers a necessary and sufficient condition on the ability of a code to correct some set of errors. Specifically, it states that for every pair of errors in the set of errors being

considered $E_i, E_j \in \mathcal{E}$, it is a necessary and sufficient that to be able to correct all the errors in $\mathcal{E}$, the condition

$$PE_i^\dagger E_j P = \alpha_{ij} P \tag{A.10}$$

must be satisfied, where α is a complex valued Hermitian matrix, and $P = \sum_k |k\rangle_L \langle k|_L$ is a projection onto the codewords $|k\rangle_L$ [2]. While examples using the quantum error correction condition quickly grow too large to present here, it is an important result not just for proving the ability of a code to correct some set of errors, but also in the discovery of quantum codes [2].

A.6 Code Rate and Perfect Error Correcting Codes

A property of a code is its *rate*; the ratio of logical qubits to physical qubits; k/n. That is, the Shor code has a rate of 1/9. While there are many important factors to consider when evaluating the usefulness of a quantum code, because qubits in quantum hardware are currently scarce [73–76], it is beneficial to have higher rate codes - of some fixed distance chosen to fit the level of error robustness needed, and some constant number of logical qubits necessary for the computation. This way fewer qubits are utilized in encoding the same information at the same distance. And indeed, it is possible to encode $k = 1$ logical qubit at distance $d = 3$ using $n = 7$ physical qubits with the Steane code, and utilizing $n = 5$ physical qubits with the 5-qubit code. Moreover, there is a bound, called the *Hamming bound*, which proves that $n = 5$ is the minimum number of physical qubits that can possibly encode $k = 1$ logical qubit at distance $d = 3$. Specifically, the Hamming bound says that if a code with $m = n - k$ independent stabilizers - which gives $m = n - k$ bits of information on an error - is to correct N_E errors, it must satisfy

$$2^{n-k} \geq N_E \tag{A.11}$$

as per the definition of Shannon information in Eq. A.8. As such, because distance $d = 3$ corresponds to being able to correct an X, Y, or Z error on any single-qubit, the number of errors it can correct for is $N_E = 3n + 1$, where n is the number of physical qubits, and the $+1$ term is included to account for the identity (no error). Hence, because in our example we are encoding $k = 1$ logical qubit, we must satisfy the condition that $2^{n-1} \geq 3n + 1$. Which holds for $n \geq 5$, and is why the 5-qubit code is the smallest full error correcting code. In fact, it is said to be *perfect* because it saturates the Hamming bound, $2^{5-1} = 3 \times 5 + 1$.

Because finding the state of a code is a cumbersome process, for brevity we will just present the stabilizers of the 5-qubit code as $\langle X_1 Z_2 Z_3 X_4, X_2 Z_3 Z_4 X_5, X_1 X_3 Z_4 Z_5, Z_1 X_2 X_4 Z_5 \rangle$. Then by observing how all potential single-qubit Pauli errors commute and anti-commute with these stabilizers, the look-up-table decoder presented in Tab. A.3 can be constructed. Observe that the syndromes are non-degenerate for all the possible single-qubit errors, so indeed, the 5-qubit code is a full error correcting code.

Error	Syndrome
I	'0000'
X_1	'0001'
X_2	'1000'
X_3	'1100'
X_4	'0110'
X_5	'0011'
Z_1	'1010'
Z_2	'0101'
Z_3	'0010'
Z_4	'1001'
Z_5	'0100'
Y_1	'1011'
Y_2	'1101'
Y_3	'1110'
Y_4	'1111'
Y_5	'0111'

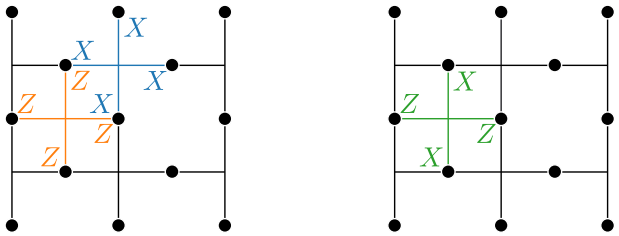
TABLE A.3: Look-up-table decoder for the 5-qubit code. This table was found by looking at how the error operators commuted and anti-commuted with the code's stabilizers. Observe that the syndromes corresponding to Y_i errors can also be found this way, or by taking sum of the syndromes of X_i and Z_i modulo 2 - which is a consequence of $Y \propto XZ$. Also see that because we have 16 unique syndromes across 4 bits, because $2^4 = 16$, any other non-equivalent error operators would give degenerate syndromes - which is consistent with the 5-qubit code being the smallest code capable of correcting single-qubit errors.

A.7 The Surface Code

The *surface code* is considered to be one of the most promising quantum codes for practical applications. As such, it has been the focus of many theoretical and experimental error correction efforts.

A common class of codes are called *low-density parity-check (LDPC)* codes. Their defining property is that all members of an LDPC code's family must have a constant upper bound on the number of physical qubits that each of the code's stabilizers act on, and another constant upper bound on the number of stabilizers acting on each physical qubit [33, 34]. The surface code is LDPC - and CSS. As a short digressions, WiFi and 5G employ classical LDPC codes for error correction - which have an analogous definition [34].

Its prominence is owed to its high *threshold* under depolarizing noise, its ability to match the topology of hardware - by allowing for embedding onto a planar geometry, and its ability to passively reduce errors by only having localized stabilizers checks - reducing the long range interactions across qubits in the syndrome extraction procedure, which addresses the difficulties of performing long range multi-qubit gates with high fidelity [10–13, 34–37, 77]. Note that a threshold is just a characterization of a quantum error correction procedure's performance, as Appendix A.3 describes in further detail.



(a) (b)

Figure A.2: The surface code and the $XZZX$ surface code. In these figures the black dots represent the locations of physical qubits. The black edges joining up to four qubits is called a *star*, and the blank squares that make up the negative space between up to four qubits is called a *plaquette*. The stars and plaquettes tile the latice and each represent a stabilizer of the code. (a) In the surface code, each star represents an $XXXX$ stabilizer, and each plaquette represents a $ZZZZ$ stabilizer. (b) In the $XZZX$ surface code, the stars and palquettes both represent $XZZX$ stabilizers.

An illustration of the surface code and its stabilizers are shown in Fig. A.2a. See that the code has only $XXXX$ and $ZZZZ$ type stabilizers tiling its surface, and so it is indeed CSS. Note that the family members of the surface code are constructed by taking one of the 4-qubit squares of the code as a basis, and tiling it around the edges of the code to grow its surface. Scaling the code in this way shows that it is indeed LDPC; the stabilizers will never exceed weight four, and each qubit will never be acted on by more than four stabilizers. Not shown on the figure are the code's logical operators. A choice for the minimum weight logical operators would be a horizontal line of Xs along the entire top edge of the plane and a vertical line of Zs along the entire left edge of the plane. Accordingly, a planar surface code's distance scales according to $d = \frac{1}{2}(\sqrt{2n-1}+1)$. Equivalently a planar surface codes of distance d is a $[[d^2+(d-1)^2, 1, d]]$ code. That is, it encodes $k = 1$ qubit of information across $n = d^2 + (d-1)^2$ physical qubits.

The popularity of the surface code has encouraged the discovery of many surface code variants [20, 38–43]. One such recent variation that has gained much attention is the XZZX surface code, presented in Fig. A.2b [20, 21, 23, 25]. It is obtained by simply applying a Hadamard to every second qubit on the surface code. This way all the stabilizers are now of the from $XZZX$ - making it no longer a CSS code. By doing this, the code better captures the structure of depolarizing biased noise, offering improved thresholds in these more realistic regimes - while retaining all the features that make the original surface code attractive [20, 44]. Also, this idea has since been generalized to other LDPC codes [21].

However, the surface code has three main drawbacks. First, it is low rate; encoding just $k = 1$ qubit across $d^2 + (d-1)^2$ qubits in the case of a d distance planar (surface) code - and so it potentially overuses physical qubits [45]. In fact, the number of encoded logical qubits of a code is bounded by the locality of its stabilizers [45], and so while logical qubits can be added to the surface code [43], no simple change to it can significantly increase its rate [45]. Second, its distance d scales slowly relative to the number of physical qubits n, as $d = \frac{1}{2}(\sqrt{2n-1}+1)$ [34, 45]. And third, it does not perform well relative to the best

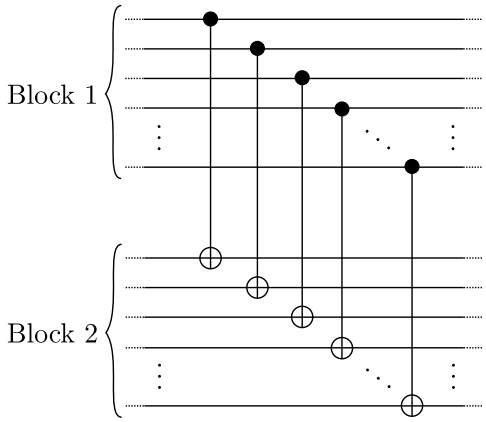


Figure A.3: A transversal $CNOT$ gate.

known classical codes [43, 45]. Which is in contrast to quantum codes that in fact match the properties of their classical counterparts [43, 45–47]. The issue with these codes though is that as they are scaled up, their stabilizer checks act on a rapidly growing number of physical qubits per logical qubit, which increases the difficulty of reliably performing stabilizer checks [34], and could demand that syndrome extraction and decoding take substantially more time - extending computation times, and increasing the risk of a logical error. But this problem is not exclusive to quantum codes, in fact, it was an issue for classical codes too. An issue solved by classical LDPC codes - which have a definition analogous to that of quantum LDPC codes [34]. Accordingly, a promising direction for quantum error correction is the investigation of quantum LDPC codes with less locality, better distance scaling, and higher rates. As a final note, the existence of quantum LDPC codes with performance comparable to their classical analogues remains an open question [34].

A.8 A Word on Fault-Tolerance

With reference to Fig. 2.1 and Fig. 2.2, observe that the discussion of error correction thus far has made the simplifying assumption that errors only occur in one location - conveniently wedged between encoding and syndrome extraction. However, because encoding and syndrome extraction require time and gates, in reality, these areas of the circuit are not exempt from errors. In fact, quite the opposite, often being composed of the lowest-fidelity operations, they are very prone to errors [12]. So, while there are techniques for tolerating errors in all areas of the circuit, the addition of these errors does dramatically reduce an error correction procedure's threshold [12, 48, 49].

If a non-ideal component of error correction can be performed reliably, it is said to be *fault-tolerant*. Specifically, for a procedure to be considered fault-tolerant, it must prevent any single error - at any location in the circuit - from propagating through the same block of qubits, and cause no more than one other single-qubit error in each other qubit block [2, 12]. To see what it means for an error to propagate through a circuit, consider that $(X \otimes I)CNOT = CNOT(X \otimes X)$. That is, in the Heisenberg picture, an X error passing

through a $CNOT$ will, in effect, multiply - and similar relations hold for other multi-qubit gates and other Pauli errors. So, in error correction procedures, which - by their very nature of requiring entanglement - need many multi-qubit gates, it is easy to imagine that, without fault-tolerance, one error could runaway and uncontrollably spread to many more qubits. Cascading into an error too large to correct.

As such, it is important to employ fault-tolerance. However, doing so requires a quantum circuit to be edited in a way that will always increase its *overhead*; computational resources required [12]. For example, a resource heavy operation commonly found in fault-tolerant procedures is the transversal gate; a large operation built from separately applying a gate to all the system's qubits. A diagram of a transvesal $CNOT$ gate is provided in Fig. A.3. See that while it is resource heavy, any single-qubit error in the transversal $CNOT$ only has the potential to propagate to one qubit on the other block, hence it is fault-tolerant.

Note that extensive work has been done on fault-tolerance, which can be read about in [2, 12, 28, 29, 53].

A.9 Decoders

A decoder is a classical algorithm that returns an estimate of the error most likely to have occurred for a given syndrome. That is,

$$D(s) = \hat{E} \tag{A.12}$$

where D is the decoder, s is the syndrome, and $\hat{E}$ is the decoder's guess at the error that occurred. Note that decoders are designed to run on classical computers because they need to be intrinsically error tolerant; otherwise the decoder would need its own decoder, continuing *ad infinitum*. While a decoder needing to be classical is not inherently an issue, as the only input, the syndrome, is classical information, it does bring with it some considerations. That is, a decoder effective for practical applications must satisfy the following three properties [59–62]:

1. It must be low latency. In the error correction procedure, after syndrome extraction, the errored state waits in memory for the decoder to return a correction - all the while accumulating errors, which increases the probability of a logical error. A low latency decoder reduces this wait time and so decreases the chance of a logical error. Additionally, because coherence times are far shorter than computation times, error correction needs to be performed regularly through quantum computations, and so, a faster decoder will not significantly lengthen the time of a quantum computation.

2. It must be accurate. Meaning that the decoder must correctly identify the most likely error to have occurred for a given syndrome with high accuracy. This consideration exists because there is often a trade-off between speed and accuracy; a faster estimate of the error is generally a worse estimate.

3. It must be scalable. With useful quantum algorithms and practical quantum error correction requiring many physical qubits, it is important that a decoder can be scaled

to support these systems. For this reason, a decoder with good scalability will have low space and time complexity.

As an example, consider these properties for the simplest kind of decoder, the *look-up-table decoder* - which was presented earlier in Tab. 2.1 and Tab. A.3. The look-up-table (LUT) decoder maps each possible syndrome to a predetermined most likely error. Hence, a LUT decoder is very accurate and has low latency, but this comes at the cost of it having poor scalability [59]. The issue being that the size of the table will grow at a rate of 2^m where $m = n - k$ is the number of stabilzier checks, n is the number of physical qubits and k is the number of logical qubits. That is, because a useful quantum computer would require many stabilizer checks - for its many qubits, a practical implementations of a LUT would be too large to store in memory; with the limit set at $m \sim 40$ stabilizer checks [63, 64].

The LUT decoder's failures motivate the search for other, more successful decoding algorithms. While there are many different decoders - each with varying properties [44, 59, 62, 67, 68], here we will present two prominent ones: *minimum-weight perfect matching (MWPM)* and *union-find (UF)*. For a more intuitive, visual representation, we will explain these decoders using the surface code. Note however that they have both been generalized to other codes [44, 62, 68].

First consider the UF decoder. Following the example provided in Fig. A.9, this decoder starts by taking the *non-trivial syndrome bits* and then, in steps, grows their edges by half the distance to the nearest other syndrome bits. To be precise, for a syndrome bit to be non-trivial, its value must suggest that its corresponding stabilizer check anti-commuted with the error. Each non-trivial syndrome bit's *cluster* of edges is grown in this way until it joins another cluster/s such that the combined cluster is of even-parity. Observe that, by construction, X-type errors and Z-type errors occur on separate sub-lattices and hence they are corrected independently - as is required to correct Y-type errors. Once all the clusters are of even parity, the excess edges are trimmed and the decoder moves to its next step: correcting. This consists of moving a non-trivial syndrome bit in a cluster along the path of the cluster's remaining edges, and making the appropriate correction on each qubit it crosses - according to the sub-lattice being considered. Then, once two non-trivial syndrome bits in a cluster meet, they are deleted. The algorithm repeats this for all the clusters, until none remain. And this defines the error estimate output by the decoder [59, 59, 60, 69]. The UF decoder has a high accuracy, low latency, and high scalability [59, 60]. In fact, its scalability is its most attractive feature, with near-linear time complexity relative to the number of physical qubits [60].

Consider now the MWPM decoder. Continuing to follow the example provided in Fig. A.9, the first step of MWPM is to label all the *non-trivial syndrome bits* and then make them the vertices of a graph. The algorithm then joins these vertices with edges, assigning each edge a value representing the distance between the two non-trivial syndrome bits on the surface code's lattice. This continues until it has generated a complete graph. Note that similarly to UF, MWPM treats X- and Z-type errors independently - so it can also correct Y-type errors. MWPM then decomposes the complete graph into a perfect matching that has the sum of the weights of the remaining edges minimized. It can then proceed to correct these minimum weight paths in a way similar to the UF decoder. In terms of its performance, MWPM generally offers a slightly higher accuracy than the UF decoder, but at

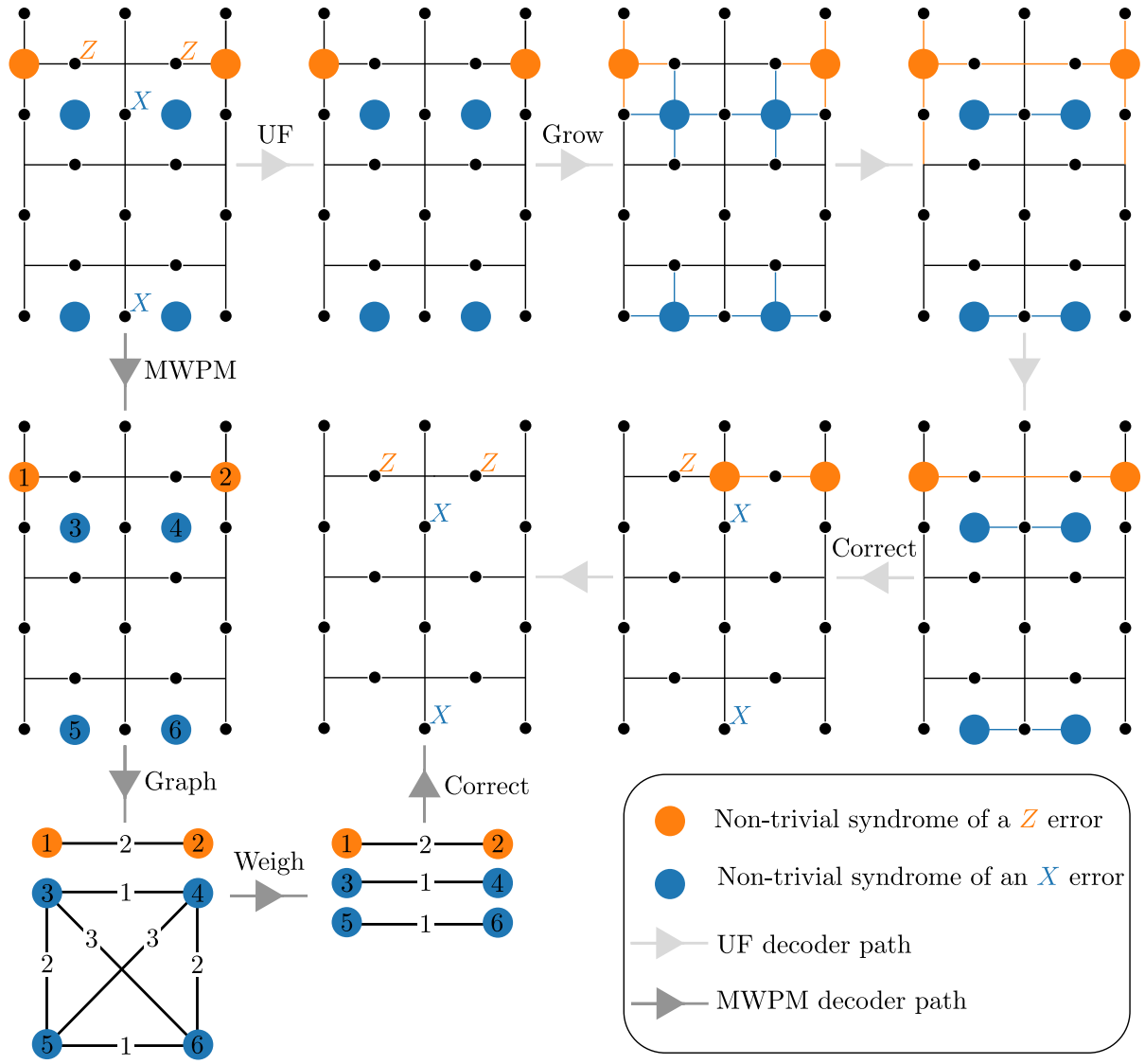


Figure A.4: An example of the minimum-weight perfect matching and union-find algorithms decoding an error on the surface code. The starting point for this figure is the example error provided in the top left corner. From there, the two algorithms go through their respective steps. We can then see that in the final step of this example, both algorithms return the same correction - and that this correction is right because it is equivalent to the initial error. Note that a detailed explanation of these algorithms is provided in the text.

the cost of having moderate latency and moderate scalability [60]. That said, there are many variants of the MWPM algorithm, each with a different accuracy, latency, and complexity. Specifically, depending on the implementation, MWPM has a complexity between $\mathcal{O}(n^3)$ and $\mathcal{O}(n^7)$ where n is the number of physical qubits [60, 71].

In terms of UF and MWPM decoders' specific performance, in a non-fault-tolerant setting, under pure dephasing noise, and using a surface code projected onto a Torus, called the

toric code, the UF and MWPM decoders achieve thresholds of 9.9% and 10.3% respectively [60]. More on their performance, it would be remiss to not note that a target of many experimental efforts is to, through hardware improvements, reduce the physical error rate - number of physical errors divided by the amount of time in a round of error correction - to be below the fault-tolerant MWPM threshold on the surface code; $p_{th} \lesssim 1\%$ [70]. Also note that while here we have presented the UF and MWPM decoders in their simplest form, much work has gone into improving them and adapting them for various applications [20, 44, 59, 61, 68, 71]. As a final point, consider that the problem of latency in decoding could perhaps in part be addressed by using architectures that naturally have a weaker coupling to the environment - because they have longer decoherence times, giving the decoder relatively more time to run.

B

Algorithm Pseudocode

In this section, we present our algorithm's pseudocode. In particular, Alg. 1 is our algorithm, which calls the functions in Alg. 2 and Alg. 3.

We would like to state that step 5 in Alg. 3 contains the add-on to our algorithm that maximally parallelizes the two-qubit operations de-encoding each stabilizer - explained in Subsection 4.2.3. Recall that the idea here is to pair non-identity operators of the stabilizer being de-encoded, then remove half of these operators paired via $CNOT$s, and repeat this process until only one non-identity operator remains on the given stabilizer. Well then we can see that the complexity scales as $\lceil \log_2(l) \rceil$ where $l \leq n$ is the weight of the stabilizer being de-encoded, and generally $l \ll n$ - because in one round a weight l stabilizer is reduced to a weight $\lceil l/2 \rceil$ stabilizer, so we want to know how many times we have to divide by two to get a weight-1 stabilizer. Moreover, we note that step 5 in Alg. 3 pairs each qubit that has a non-trivial operation on it in the given stabilizer with the next qubit in that stabilizer that also has a non-trivial operation. As explained in the main text, Subsection 4.2.3, this is perhaps effective for a linear array of qubits where the qubits are indexed by their order in the array. However, as elaborated on in Subsection 4.2.3, to generalize this idea to more general stack geometries, and to minimize the consequences of having to apply two-qubit gates across qubits with connectivity issues, one might want to replace step 5 in Alg. 3 with a more complicated algorithm to pair qubits, like union-find or minimum-weight perfect matching algorithms - which are explained in Appendix A.9. We note that, depending on the supporting algorithm used, this may increase our method's complexity.

In a worst case scenario - with stabilizers of weight equal to the number of physical qubits, $l = n$, we find the worst-case complexity in step 5 of Alg. 3 to be $\mathcal{O}(n \log_2(n))$, which dominates what is generally the highest complexity in this step, $\mathcal{O}(mn)$, when $m < \log_2(n)$. Therefore, see that in this regime of generally poor error-correcting codes ($m < \log_2(n)$ generally leaves too few stabilizers to correct all low-weight errors), our algorithm has the slightly higher worst-case complexity of $\mathcal{O}(mn \log_2(n))$ - as step 5 of Alg. 3 is called m times

in Alg. 1. Also, for completeness, recall the additional change in worst-case complexity from Subsection 4.2.5, where the complexity is $\mathcal{O}(n^2)$ when $m < \sqrt{n}$ - which similarly only holds for poor error-correcting codes.

Lastly, note that our algorithm, as it is presented here, does not include the phase tracking from Subsection 4.2.5 nor does it include the Gottesman-Knill simulation used to determine the encoded state. Because in Subsection 4.2.5 these two concepts have already been explained and their complexity considered, we will just note that the phase tracking takes place in steps 6 and 7 of Alg. 3, and that the simulation of the encoded stabilizer state is done after the algorithm presented below is complete, and the encoding circuit is returned.

Algorithm 1 Proposed algorithm for efficient encodings ($\mathcal{O}(m^2n)$). **Complexity**

input the $m \times n$ parity check matrix of the code and the costs of CNOTs, Hadmards, and O_G gates.
output the encoding circuit.

1: **for** stabilizer in range(0, m) **do** $\mathcal{O}(m)$
2: REDUCE(parityCheckMatrix,FINDSAFEROW(parityCheckMatrix,... ...costCNOT, costH, costO)) $\mathcal{O}(mn)$
3: **end for**

Algorithm 2 Evaluates cost ($\mathcal{O}(mn)$). **Complexity**

1: **function** FINDSAFEROW(parityCheckMatrix, costCNOT, costH, costO) $\boldsymbol{\mathcal{O}(mn)}$

input the $m \times n$ parity check matrix of the code and the costs of CNOTs, Hadmards, and our gates.
output the cheapest row to reduce.

2: rowSums $\leftarrow$ a list containing the sum of each parity check matrix row. $\mathcal{O}(mn)$
3: operatorsZ $\leftarrow$ a list containing the number of Z operators in each row. $\mathcal{O}(mn)$
4: operatorsY $\leftarrow$ a list containing the number of Y operators in each row. $\mathcal{O}(mn)$
5: costRows $\leftarrow$ (rowsSums $-$ 1) $\times$ costCNOT + operatorsZ $\times$ costH + operatorsY $\times$ costO $\mathcal{O}(m)$
6: **return** the row number of the minimum value in costRows, breaking ties by selecting from the degenerate minimums randomly. $\mathcal{O}(m)$
7: **end function**

Algorithm 3 Reduces the parity check matrix ($\mathcal{O}(mn)$). **Complexity**

1: **function** REDUCE(parityCheckMatrix, safeRow) $\boldsymbol{\mathcal{O}(mn)}$

input the $m \times n$ parity check matrix, and the the cheapest row's index.
output an updated parity check matrix with the given row reduced and the quantum gates needed to de-encode that row.

2: operatorsZ ← a list containing the column numbers of the Z operators in the minimum cost row. Say it has $l_Z \leq n$ elements. $\mathcal{O}(n)$

3: operatorsY ← a list containing the column numbers of the Y operators in the minimum cost row. Say it has $l_Y \leq n$ elements. $\mathcal{O}(n)$

4: operatorsNonTrivial ← a list contatining the column numbers of all non-identity operators on the minimum cost row. Say there are $l \leq n$ such operators. $\mathcal{O}(n)$

5: CNOTs ← pair consecutive items in the operations NonTrivial list and record them to this list. The first number in each pair is the control of the CNOT, and the second number is the target. Now similarly pair the controls from this list, continuing until just one element remains. For example, we might have the list [[(1,2),(3,4)],[(1,3)]]. $\mathcal{O}(n \log_2(l))$

6: in the parity check matrix, apply Hadamards to the columns listed in operatorsZ, and O_G gates to the columns listed in operatorsY. $\mathcal{O}(m(l_Z + l_Y))$

7: apply each CNOT in CNOTs to every row of the parity check matrix. $\mathcal{O}(mn)$

8: remove all operators in the same column as the remaining stabilizer in the newly reduced row - which is equivalent to multiplying by the de-encoded stabilizer. $\mathcal{O}(m)$

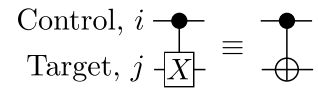
9: **return** the reduced parity check matrix, CNOTs, operatorsZ, and operatorsY. $\mathcal{O}(1)$

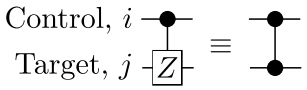
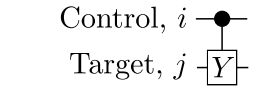
10: **end function**

C

Relevant Quantum Circuit Elements

The various elements in a quantum circuit that are referred to throughout this paper.

Circuit Element Name	Circuit Symbol	Matrix Representation
Measurement		-
Pauli-X or NOT (X)		$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
Pauli-Z (Z)		$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
Pauli-Y (Y)		$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$
Hadamard (H)		$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
Phase gate (S)		$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$
z-rotation by θ ($R_Z(\theta)$)		$\begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$
Controlled-NOT ($CNOT_{i,j}$ or $CX_{i,j}$)		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

Circuit Element Name	Circuit Symbol	Matrix Representation
Controlled-Z ($CZ_{i,j}$)		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$
Controlled-Y ($CY_{i,j}$)		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{pmatrix}$

References

[1] R. Cleve, D. Gottesman, *Efficient Computations of Encodings for Quantum Error Correction*. Phys. Rev. A **56**, 76 (1997).

[2] M. Nielsen, I. Chuang, *Quantum Computation and Quantum Information: 10th anniversary edition*. Cambridge University Press (2010).

[3] J. Bell, *On the Einstein Podolsky Rosen paradox*. Physics Physique Fizika **1**, p.195–200 (1964).

[4] B. Hensen, H. Bernien, A. Dréau, *et al*, *Loophole-free Bell inequality violation using electron spins separated by 1.3 kilometres*. Nature **526**, p.682–686 (2015).

[5] C. Shannon, *A Symbolic Analysis of Relay and Switching Circuits*. Transactions of the American Institute of Electrical Engineers **57**, p.713-723 (1938).

[6] I. Johnson, *Cosmic particles can change elections and cause planes to fall through the sky, scientists warn*. The Independent (2017).

[7] B. Schroeder , E. Pinheiro , W. Weber, *DRAM Errors in the Wild: A Large-Scale Field Study*. Association for Computing Machinery **54**, p.100-107 (2011).

[8] M. Suchara, J. Kubiatowicz, A. Faruque, *et al*, *QuRE: The Quantum Resource Estimator toolbox*. IEEE 31st International Conference on Computer Design, p.419-426 (2013).

[9] Google Quantum Ai, *Exponential suppression of bit or phase flip errors with repetitive error correction*. Nature **595**, p.383-387 (2021).

[10] C.Ballance, T. Harty, N. Linke, M. Sepiol, D. Lucas, *High-Fidelity Quantum Logic Gates Using Trapped-Ion Hyperfine Qubits*. Phys. Rev. Lett. **117**, 060504 (2016).

[11] J. Muhonen, A. Laucht, S. Simmons, *et al*, *Quantifying the quantum gate fidelity of singleatom spin qubits in silicon by randomized benchmarking*. J. Phys.: Condens. Matter **27**, 154205 (2015).

[12] J. Roffe, *Quantum Error Correction: An Introductory Guide*. Contemporary physics **60**, p.226-245 (2019).

[13] A. Noiri, K. Takeda, T. Nakajima, *et al*, *Fast universal quantum gate above the fault-tolerance threshold in silicon*. Nature **601**, p.338-342 (2022).

[14] I. Chuang, F. Zhao, *Lecture 19: How to Build Your Own Quantum Computer*. MIT Department of Mathematics (2003).

[15] B. Hensen, W. Huang, C. Yang, *et al*, *A silicon quantum-dot-coupled nuclear spin qubit*. Nature **15**, p.13-17 (2020).

[16] M. Gutiérrez, C. Smith, L. Lulushi, *et al*, *Errors and pseudo-thresholds for incoherent and coherent noise*. Phys. Rev. A **94**, 042338 (2016).

[17] Z. Cai, X. Xu, S. Benjamin *Mitigating coherent noise using Pauli conjugation*. npj Quantum Information **6**, 17 (2020).

[18] N. Delfosse, B. Reichardt, K. Svore, *Beyond single-shot fault-tolerant quantum error correction*. IEEE Transactions on Information Theory **68**, p.287-301 (2022).

[19] S. Beale, J. Wallman, M. Gutiérrez, *et al*, *Coherence in quantum error-correcting codes*. arXiv:1805.08802 (2018).

[20] J. P. B. Ataides, D. K. Tuckett, S. D. Bartlett, *et al.*, *The XZZX surface code*. Nat. Commun. **12**, 2172 (2021).

[21] J. Roffe, L. Cohen, A. Quintivalle, *et al.*, *Bias-tailored quantum LDPC codes*. arXiv:2202.01702 (2022).

[22] D. Tuckett, S. Bartlett, S. Flammia, B. Brown, *Fault-tolerant thresholds for the surface code in excess of 5% under biased noise*. Phys. Rev. Lett. **124**, 130501 (2020).

[23] A. Darmawan, B. Brown, A. Grimsmo, *et al.*, *Practical quantum error correction with the XZZX code and Kerr-cat qubits*. PRX Quantum **2**, 030345 (2021).

[24] S. Singh, A. Darmawan, B. Brown, S. Puri, *High-Fidelity Magic-State Preparation with a Biased-Noise Architecture*. arXiv:2109.02677 (2021).

[25] J. Claes, J. Bourassa, S. Puri, *Tailored cluster states with high threshold under biased noise*. arXiv:2201.10566 (2022).

[26] S. Aaronson, D. Gottesman, *Improved Simulation of Stabilizer Circuits*. Phys. Rev. A **70**, 052328 (2004).

[27] S. Beale, J. Wallman, M. Gutiérrez, *et al*, *Quantum Error Correction Decoheres Noise*. Phys. Rev. Lett. **121**, 190501 (2018).

[28] D. Gottesman, *Stabilizer Codes and Quantum Error Correction*. arXiv:quant-ph/9705052 (1997).

[29] D. Gottesman, *An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation*. arXiv:0904.2557 (2009).

[30] S. Chandak, J. Mardia, M. Tolunay, *Implementation and analysis of stabilizer codes in pyQuil*. Stanford (2019).

[31] G. Long, Y. Sun, *Efficient Scheme for Initializing a Quantum Register with an Arbitrary Superposed State*. Phys. Rev. A **64**, 014303 (2001).

[32] S. Kak *The Initialization Problem in Quantum Computing*. Foundations of Physics **29**, p.267-279 (1999).

[33] P. Panteleev, G. Kalachev, *Degenerate Quantum LDPC Codes With Good Finite Length Performance*. Quantum **5**, 585 (2021)

[34] N. Breuckmann, J. Eberhardt, *Quantum Low-Density Parity-Check Codes*. PRX Quantum **2**, 040101 (2021).

[35] S. Bravyi, A. Kitaev, *Quantum codes on a lattice with boundary*. arXiv:quant-ph/9811052 (1998).

[36] A. Kitaev, *Fault-tolerant quantum computation by anyons*. Annals of Physics **303**, p.2-30 (2003).

[37] M. Freedman, D. Meyer, *Projective Plane and Planar Quantum Codes*. arXiv:quant-ph/9810055 (1998).

[38] A. Bolt, G. Duclos-Cianci, D. Poulin, T. Stace, *Foliated Quantum Codes*. Phys. Rev. Lett. **117**, 070501 (2016).

[39] A. Bolt, D. Poulin, T. Stace, *Decoding schemes for foliated sparse quantum error-correcting codes*. Phys. Rev. A **98**, 062302 (2018).

[40] S. Bartolucci, P. Birchall, H. Bombin, *et al.*, *Fusion-based quantum computation*. arXiv:2101.09310v1 (2021).

[41] D. Tuckett, A. Darmawan, C. Chubb, *et al.*, *Tailoring Surface Codes for Highly Biased Noise*. Phys. Rev. X **9**, 041031 (2019).

[42] D. Tuckett, S. Bartlett, S. Flammia, *Ultrahigh Error Threshold for Surface Codes with Biased Noise*. Phys. Rev. Lett. **120**, 050505 (2018).

[43] J. Auger, H. Anwar, M. Gimeno-Segovia, *Fault-tolerance thresholds for the surface code with fabrication errors*. Phys. Rev. A **96**, 042316 (2017).

[44] M. Gorman, T. Farrelly, *Union-find Decoding on the XZZX Surface Code*. The University of Queensland (2021).

[45] S. Bravyi, D. Poulin, B. Terhal, *Tradeoffs for reliable quantum information storage in 2D systems*. Phys. Rev. Lett. **104**, 050503 (2010).

[46] A. Calderbank, P. Shor, *Good Quantum Error-Correcting Codes Exist*. Phys. Rev. A **54**, 1098 (1996).

[47] A. Ashikhmin, S. Litsyn, M. Tsfasman, *Asymptotically good quantum codes*. Phys. Rev. A **63**, 032311 (2001).

[48] D. Wang, A. Fowler, A. Stephens, L. Hollenberg, *Threshold error rates for the toric and surface codes*. arXiv:0905.0531 (2009).

[49] C. Lai, Y. Zheng, T. Brun, *Fault-tolerant Preparation of Stabilizer States for Quantum CSS Codes by Classical Error-Correcting Codes*. Phys. Rev. A **95**, 032339 (2017).

[50] Y. Zheng, C. Lai, T. Brun, *Efficient preparation of large-block-code ancilla states for fault-tolerant quantum computation*. Phys. Rev. A **97**, 032331 (2018).

[51] E. Dennis, A. Kitaev, A. Landahl, J. Preskill, *Topological quantum memory*. J. Math. Phys. **43**, 4452 (2002).

[52] M. Plenio, V. Vedral, P. Knight, *Conditional generation of error syndromes in fault-tolerant error correction*. Phys. Rev. A **55**, 4593 (1997).

[53] J. Preskill, *Fault-Tolerant Quantum Computation*. arXiv:quant-ph/9712048 (1997).

[54] D. DiVincenzo, P. Shor, *Fault-Tolerant Error Correction with Efficient Quantum Codes*. Phys. Rev. Lett. **77**, 3260 (1996).

[55] S. Huang, K. Brown, *Between Shor and Steane: A unifying construction for measuring error syndromes*. Phys. Rev. Lett. **127**, 090505 (2021).

[56] A. Steane, *Active Stabilization, Quantum Computation and Quantum State Synthesis*. Phys. Rev. Lett. **78**, 2252 (1997).

[57] E. Knill, *Quantum computing with realistically noisy devices*. Nature **434**, p.39–44 (2005).

[58] E. Knill, *Scalable quantum computing in the presence of large detected-error rates*. Phys. Rev. A **71**, 042322 (2005).

[59] P. Das, C. A. Pattison, S. Manne, *et al*, *A Scalable Decoder Micro-architecture for Fault-Tolerant Quantum Computing*. arXiv:2001.06598v1 (2020).

[60] N. Delfosse, N. Nickerson, *Almost-linear time decoding algorithm for topological codes*. Quantum **5**, 595 (2021).

[61] A. Fowler, *Minimum weight perfect matching of fault-tolerant topological quantum error correction in average O(1) parallel time*. arXiv:1307.1740v3 (2014).

[62] V. Kolmogorov, *Blossom V: a new implementation of a minimum cost perfect matching algorithm*. Mathematical Programming Computation **1**, p.43–67 (2009).

[63] T. Jones, A. Brown, I. Bush, S. Benjamin, *QuEST and High Performance Simulation of Quantum Computers*. Scientific Reports **9**, 10736 (2019).

[64] E. Pednault, J. Gunnels, G. Nannicini, *et al.*, *Pareto-Efficient Quantum Circuit Simulation Using Tensor Contraction Deferral*. arXiv:1710.05867 (2017).

[65] Y. Nam, N. Ross, Y. Su, A, *et al.*, *Automated optimization of large quantum circuits with continuous parameters*. npj Quantum Information **4**, 23 (2018).

[66] A. Botea, A. Kishimoto, R. Marinescu, *On the Complexity of Quantum Circuit Compilation*. The Eleventh International Symposium on Combinatorial Search **9**, p.138-142 (2018).

[67] A. Ferris, D. Poulin, *Tensor Networks and Quantum Error Correction*. Phys. Rev. Lett. **113**, 030501 (2014).

[68] N. Delfosse, M. Hastings, *Union-Find Decoders For Homological Product Codes*. Quantum **5**, 406 (2021).

[69] S. Hu, D. Elkouss, *Quasilinear Time Decoding Algorithm for Topological Codes with High Error Threshold*. Delft University of Technology (2020).

[70] A. Fowler, A. Stephens, P. Groszkowski, *High threshold universal quantum computation on the surface code*. Phys. Rev. A **80**, 052312 (2009).

[71] S. Huang, M. Newman, K. Brown, *Fault-Tolerant Weighted Union-Find Decoding on the Toric Code*. Phys. Rev. A **102**, 012419 (2020).

[72] R. Affeldt, J. Garrigue, T. Saikawa, *A Library for Formalization of Linear Error-Correcting Codes*. Journal of Automated Reasoning **64**, p.1123–1164 (2020).

[73] J. Preskill, *Quantum Computing in the NISQ era and beyond*. Quantum **2**, 79 (2018).

[74] Google Quantum AI, *Hartree-Fock on a superconducting qubit quantum computer*. Science **369**, 6507 (2020).

[75] W. Huang, W. Chien, C. Cho, *et al.*, *Mermin's Inequalities of Multiple qubits with Orthogonal Measurements on IBM Q 53-qubit system*. Quantum Engineering **2**, e45 (2020).

[76] B. Villalonga, S. Boixo, B. Nelson, *et al.*, *A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware*. npj Quantum Information **5**, 86 (2019).

[77] W. Huang, C. Yang, K. Chan, *et al.*, *Fidelity benchmarks for two-qubit gates in silicon*. Nature volume **569**, p.532–536 (2019).

[78] A. Ferris, D. Poulin, *Branching MERA codes: a natural extension of classical and quantum polar codes*. arXiv:1312.4575 (2013).

[79] T. Farrelly, R. Harris, N. McMahon, T. Stace, *Parallel decoding of multiple logical qubits in tensor-network codes*. arXiv:2012.07317 (2020).

[80] IBM Quantum, *Qiskit: An Open-source Framework for Quantum Computing*. IBM (2021).

[81] B. Ferreira, *How Space Weather Can Influence Elections on Earth*. Vice (2017).

[82] A. Kissinger, J. Wetering, *PyZX: Large Scale Automated Diagrammatic Reasoning*. EPTCS 318, p.229-241 (2020).

[83] T. Jones, S. Benjamin, *Robust quantum compilation and circuit optimisation via energy minimisation*. Quantum **6**, 628 (2022).

[84] M. Davis, E. Smith, A. Tudor, *et al.*, *Towards Optimal Topology Aware Quantum Circuit Synthesis*. IEEE International Conference on Quantum Computing and Engineering, p.223-234 (2020).

[85] A. Vardy, *Algorithmic Complexity in Coding Theory and the Minimum Distance Problem*. Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, P.92–109 (1997).

[86] M. Gorman, T. Farrelly, *Fusion-based Foliated Quantum Codes*. The University of Queensland (2021).

[87] A. Hagberg, D. Schult, P. Swart, *Exploring network structure, dynamics, and function using NetworkX*. Proceedings of the 7th Python in Science Conference, p.11–15 (2008).

[88] K. Patel, R. Markov, J. Hayes, *Efficient Synthesis of Linear Reversible Circuits*. arXiv.quant-ph/0302002 (2003).

[89] T. Brugière, M. Baboulin, B. Valiron, S. Martiel, C. Allouche, *Gaussian elimination versus greedy methods for the synthesis of linear reversible circuits*. ACM Transactions on Quantum Computing **2**, 11 (2021).

[90] T. Brugière, M. Baboulin, B. Valiron, S. Martiel, C. Allouche, *Quantum CNOT Circuits Synthesis for NISQ Architectures Using the Syndrome Decoding Problem*. Lecture Notes in Computer Science **12227** (2020).

[91] M. Almazrooie, A. Samsudin, R. Abdullah, K. Mutter, *Quantum reversible circuit of AES-128*. Quantum Inf Process **17**, 112 (2018).

[92] M. Abubakar, L. Jung, N. Zakaria, A. Younes, A. Abdel-Aty, *Reversible circuit synthesis by genetic programming using dynamic gate libraries*. Quantum Inf Process **16**, 160 (2017).

[93] M. Saeedi, I. Markov, *Synthesis and Optimization of Reversible Circuits—A Survey*. ACM Computing Surveys **45**, 21 (2013).

[94] X. Xu, S. Benjamin, X. Yuan, *Variational Circuit Compiler for Quantum Error Correction*. Phys. Rev. Applied **15**, 034068 (2021).

[95] M. Grassl, M. Roetteler, T. Beth, *Efficient Quantum Circuits for Non-Qubit Quantum Error-Correcting Codes*. International Journal of Foundations of Computer Science **14**, p. 757-775 (2003).

[96] P. Nadkarni, S. Garani, *Encoding of Quantum Stabilizer Codes Over Qudits with $d = p^k$*. IEEE Globecom Workshops, p. 1-6 (2018).

[97] W. Huang, Z. Wei, *Journal of Physics A: Mathematical and Theoretical Efficient one-way quantum computations for quantum error correction*. J. Phys. A: Math. Theor. **42**, 295301 (2009).

[98] A. Steane, *Fast fault-tolerant filtering of quantum codewords*. arXiv.quant-ph/0202036 (2002).

[99] A. Paetznick, *Resource optimization for fault-tolerant quantum computing*. arXiv.1410.5124 (2014).

[100] M. Gong, X. Yuan, S. Wang, *et al*, *Experimental exploration of five-qubit quantum error correcting code with superconducting qubits*. National Science Review, Volume **9** (2022).

[101] C. Colbourn, *The complexity of completing partial Latin squares*. Discrete Applied Mathematics **8**, p. 25-30 (1984).

[102] A. Cross, D. Divincenzo, B. Terhal, *A comparative code study for quantum fault tolerance*. Quantum Information Computation **9**, p. 541–572 (2009).

[103] A. Kapoor, R. Rizzi, *Edge-Coloring Bipartite Graphs*. Journal of Algorithms **34**, p. 390-396 (2000).

[104] A. Paetznick, B. Reichardt, *Fault-tolerant ancilla preparation and noise threshold lower bounds for the 23-qubit Golay code*. arXiv:1106.2190 (2013).

[105] H. Gabow, *Data structures for weighted matching and nearest common ancestors with linking*. Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms, p.323-443 (1990).

[106] S. Boyd, L. Vandenberghe, *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. Cambridge University Press (2018).

[107] D. Gottesman, *The Heisenberg Representation of Quantum Computers*. arXiv:quant-ph/9807006 (1998).

[108] M. Grassl, *Bounds on the minimum distance of linear codes and quantum codes*. Online available at http://www.codetables.de (Accessed May 8th 2022).

[109] M. Grassl, *Searching for linear codes with large minimum distance*. Algorithms and Computation in Mathematics **19** (2006).

[110] R. Takagi, T. Yoder, I. Chuang, *Error rates and resource overheads of encoded three-qubit gates*. Phys. Rev. A **96**, 042302 (2017).