MASTER'S THESIS

Quasilinear Time Decoding Algorithm for Topological Codes with High Error Threshold

Author
S. HU

Supervisor
D. ELKOUSS

April 28, 2020

Contents

Chapter 1

Introduction

Bell states

There are four maximally entangled two-qubit Bell states. Together, they form a maximally entangled basis known as the Bell basis.

$$|\phi^+\rangle = \frac{1}{2}\big(|00\rangle_{AB} + |11\rangle_{AB}\big) \tag{1.1}$$

$$|\phi^-\rangle = \frac{1}{2}\big(|00\rangle_{AB} - |11\rangle_{AB}\big) \tag{1.2}$$

$$|\psi^+\rangle = \frac{1}{2}\big(|01\rangle_{AB} + |10\rangle_{AB}\big) \tag{1.3}$$

$$|\psi^-\rangle = \frac{1}{2}\big(|01\rangle_{AB} - |10\rangle_{AB}\big) \tag{1.4}$$

The Bell state that we want to prepare, $|\phi^+\rangle$, can be created by applying the Hadamard and the CNOT gate on two qubits.
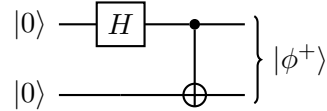
However, due to the fact that the system is imperfect, a Werner state will be created instead with fidelity F, which is given by the following density matrix,

$$\rho_W = F|\phi\rangle\langle\phi| + \frac{1-F}{3}\sum_{i=1,2,3}|\psi_i\rangle\langle\psi_i| \tag{1.5}$$

where $|\phi\rangle$ now represents the Bell pair that we want to prepare, and $|\psi_i\rangle$ are the other three Bell states. The production of this noisy Bell pair will be illustrated by
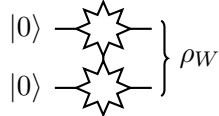
Single selection

Luckily, the fidelity of a noisy Bell pair can be enhanced through entanglement purification. In the process of *single selection*, a single noisy Bell pair is used (utility pair) in order to enhance the fidelity of another noisy bell pair (target pair). Here CZ gates are applied between the first and second qubits of the two noisy pairs. A successful parity check on the utility pair yields an enhanced target pair.
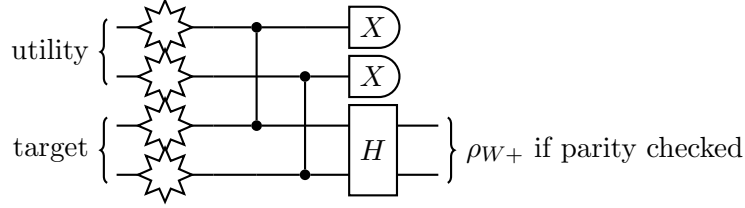
We should separate the qubits based on their location. Alice and Bob both holds one of the two qubits of both the utility and target pair. We put the qubits held by Alice on top, and Bob on the bottom.
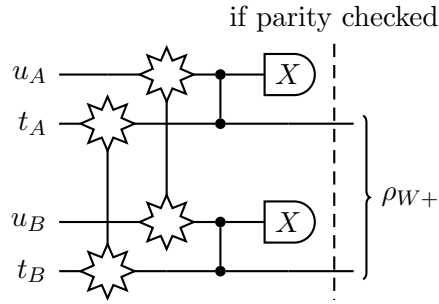
The enhanced Bell pair $\rho_{W,1}$ can be iteratively enhanced using the same scheme using newly created noisy Bell pairs ρ_W. After repeated iterations the fidelity will converge to some maximally attainable value, dependant on the fidelity of the noisy Bell pair. Two iterations of single selection is displayed below.
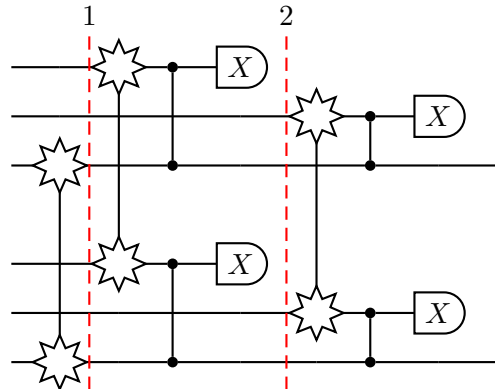
Entanglement pumping

Within each iteration, we can add additional *nested levels* to further improve the purification. In each nested level, we perform extra rounds of single selection, but taking two enhanced Bell pairs from the previous level as input. For example, in the first nested level, a Bell pair ρ_{W+} is used to purify another Bell pair with the same fidelity to ρ_{W++}, which will be the input state for the second nested level.

Double selection

Another approach which yields a higher enhanced fidelity utilizes two noisy Bell pairs ($u1$ and $u2$) to enhance a third pair (t). They perform a CZ gate locally between $u1$ and t, followed by another between $u2$ and $u1$. Now, they perform parity checks on $u1$ and $u2$. If both parity checks are passed, the protocol succeeds and the target qubit fidelity is enhanced.
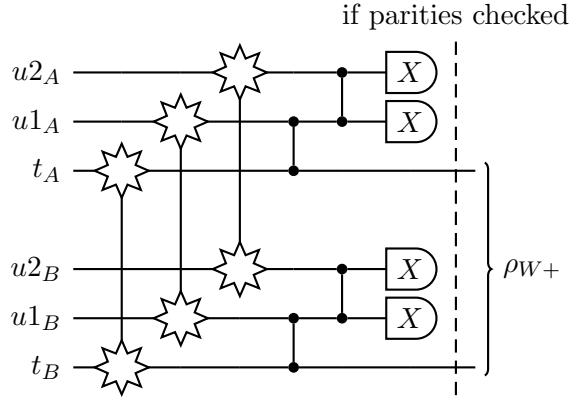
Chapter 2

Quantum error correction

To build a real world quantum computer, or a quantum communications device, one has to deal with the presence of noise, which will inevitably alter the quantum state of a qubit stored or passed through a communications channel. Recent developments have raised the fidelity of single qubit operations to up to one single failure in 10^6 operations [1]. But even this fidelity is not enough, as a full quantum computation may require millions of qubits, and the generation of entangled states over a large number of qubits. With imperfect quantum gates, anything we do in order to perform a computation will add to the error.

The theory of *quantum error correction* has been developed to counteract this noise, by using a larger number of redundant *physical qubits* to encode for a smaller number of *logical qubits*. By adding extra redundant qubits in our *error correcting code*, we can carefully encode the quantum state which we wish to protect, as long as the rate of errors on the physical errors is low enough [2–4].

In this chapter, we will introduce the principles of quantum error correction by the example of the *three-bit repetition code*. In section 2.1, we will first cover the classical variant, and the quantum variant in section 2.2. A more practical language to describe these codes is the *stabilizer formalism*, in section 2.3. The set of tools and principles explained in these sections form the basis for higher level quantum codes that we will come later in this thesis.

2.1   Classical three-bit repetition code

To introduce some of the terms that we are going to use later, let us first start with a classical example the three-bit repetition code. This code encodes bits (a single bit in the example) by repeating them. Let the *codewords* of the single logical bit be:

$$0_L = 000 \qquad\qquad 1_L = 111 \tag{2.1}$$

In order to do computations, a NOT gate may be applied to the codewords to flip the logical value:

$$000 \leftrightarrow 111 \tag{2.2}$$

A *bit-flip* error can occur on any of the three bits in the code, which flips the single bit-value from 0 to 1 and 1 to 0. An error can be *detected* by measuring the bits and comparing whether the bits have equal value. An detected error can be *corrected* by computing the majority-function of the bitstring. Thus the three-bit repetition code will be correctly corrected if less than half of the bits were flipped.

$$0_L \xrightarrow{E_2} 010 \xrightarrow{correction} 000 \qquad\qquad 0_L \xrightarrow{E_2,E_3} 011 \xrightarrow{correction} 111 \qquad (2.3)$$

The *distance d* of a classical code is the minimum amount of bit flips to transfer one codeword to another. In the case of the three-bit repetition code, the code distance is 3. The number of bits in de code n, the number of encoded bits k and the distance of a code can be used to fully describe a code in the $[n, k, d]$ notation. The three-bit repetition code is a $[3, 1, 3]$ code.

2.2 Quantum three-bit repetition code

From here, we can describe how to do computations on a quantum system. We start by considering an example of the *quantum three-bit repetition code*, where the classical bits are now replaced by qubits that can be in the superposition of the two classical 0 or 1 states. The basis states of the encoded qubit is the tensor product of the single qubit states:

$$|0\rangle_L = |0\rangle \otimes |0\rangle \otimes |0\rangle = |000\rangle \qquad |1\rangle_L = |1\rangle \otimes |1\rangle \otimes |1\rangle = |111\rangle \qquad (2.4)$$

A pure qubit state can also be a superposition of the bases states and is encoded as:

$$|\psi\rangle_L = \alpha |0\rangle_L + \beta |1\rangle_L \qquad (2.5)$$

2.2.1 Pauli operators

The Pauli operations are unitary operations on single qubits, and will be applied very often throughout this thesis. Including the identity operator, the Pauli group on a single qubit, $\mathcal{P}_1$, consists of:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad (2.6)$$

The Pauli operators represent errors that can occur on a single qubit. The Pauli X operator is analogous to the classical *bit-flip* error and acts on the qubit computational basis states:

$$X |0\rangle = |1\rangle \qquad\qquad X |1\rangle = |0\rangle \qquad (2.7)$$

Additionally, the Pauli Z operator introduces phase errors on a quantum bit:

$$Z |0\rangle = - |0\rangle \qquad\qquad Z |1\rangle = - |1\rangle \qquad (2.8)$$

8

The elements of the Pauli group on n qubits, $\mathcal{P}_n$, consists of tensor products of single qubit Pauli operators, such that $\mathcal{P}_n = \mathcal{P}_1^{\otimes n}$. We use the index of a Pauli operator to indicated on which qubit it has operated on, while other qubits are acted on by the identify. In cases without indices, the order of the operators indicate the qubit it acts on. For example, element $P = X_1 \otimes X_2 = XX$ on the three-bit repetition code means that qubit 1 and 2 have been acted on by the Pauli X operator, while qubit 3 is acted on by the identity. The tensor product symbol is often omitted for clarity, such that the above operation can be also written as $P = X_1 X_2$. The *weight* of an operator is the number of qubits on which it does not act non-trivially. On the pure three-qubit encoded state, a bit-flip error on the second qubit is applied as:

$$X_2 \ket{\psi}_L = (I \otimes X \otimes I) \ket{\psi}_L = \alpha \ket{010} + \beta \ket{101} \tag{2.9}$$

2.2.2 Logical operations

In order to do computations on the encoded qubit of our three-bit repetition code, we wish to find the Pauli operators in $\mathcal{P}_3$ which flips any basis of the basis states of the encoded qubit to the other. We find that $X \otimes X \otimes X$ transforms $\ket{0}_L$ to $\ket{1}_L$, which is known as the logical bit-flip.

Furthermore, we now have the logical Z operator which must map $\ket{0}_L$ to $-\ket{0}_L$ and $\ket{1}_L$ to $-\ket{1}_L$. We see that for example $Z \otimes I \otimes I$ achieves this, but also any other $\mathcal{P}_3$ operator with two identities and one Pauli Z operator. Thus there are multiple operators that achieves the same. We formalize the logical operators as

$$\bar{X} = XXX \qquad\qquad\qquad \bar{Z} = ZII \tag{2.10}$$

The distance of a quantum code is the minimal weight of any logical operators on the code. In the above case, the weight of the encoded X operator $\bar{X}$ is 3, hence the code can detect up to 2 X errors, analogous to the classical case. However, the weight of the encoded Z operator $\bar{Z}$ is only 1, which means that Z errors cannot be detected at all.

2.2.3 Error detection

To detect errors in our repetition code, we now cannot measure the states directly, as any measurement would collapse the encoded state, and therefore destroy the encoded information. Instead, we can now detect errors by measuring the *parity* of two or more qubits rather than single qubits. For example, for two qubits, we can measure the parity by adding an *ancillary* or *ancilla* qubit prepared in $\ket{0}$ and measure it in the computational basis after connecting our quantum circuit as:

Note that measuring the ancilla qubit in the computational basis will be equivalent to measuring $Z \otimes Z$ on the first two qubits, as

$$\begin{aligned}
(ZZ)\ket{00} &= \ket{00} & (ZZ)\ket{01} &= -\ket{01} \\
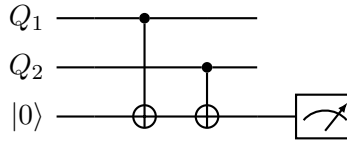(ZZ)\ket{10} &= -\ket{10} & (ZZ)\ket{11} &= \ket{11}
\end{aligned} \tag{2.11}$$

Figure 2.1: The quantum circuit for a parity measurement on two qubits, Q_1 and Q_2, which is measured on the ancilla qubit prepared in $|0\rangle$.

Now for our quantum three-bit repetition code, we need to setup ancilla qubits between each of the 3 qubits, such that the parity between any two qubits can be measured. For the state in equation 2.5, any parity measurement ZZI, ZIZ or IZZ will return even parity. If one of the qubits has encountered a bit-flip error such as in the second qubit in equation 2.9, two of the parity measurements will return a -1 eigenvalue, in this case ZZI and IZZ.

Furthermore, we see that no configuration of ancilla qubits could be set up for the three-bit repetition code to detect for phase errors, which was already set by the weight of the logical Z operator.

2.2.4   Error correction

As the error has been identified, we can apply correct Pauli operator from $\mathcal{P}_3$ to correct the error. In the above case, we wish to apply IXI to clip the second qubit to correct the code.

In the quantum three-bit repetition code, we can very simply deduct which qubit has encountered an error from the combination of uneven parity measurements. This is called *decoding* the error and is the main function of a *decoder*. More complex error correcting codes involves decoding algorithms which are far more complex than in the three-bit repetition code, which we will see more of later.

2.3   Stabilizer Formalism

As we have seen above, we can use the Pauli group $\mathcal{P}_n$ to easily describe a quantum error correcting code of n qubits, without explicitly looking at the *state* of the qubit. This powerful technique is called the *stabilizer formalism* [5], and is the most widely formalism used to describe topological codes.

A quantum error correcting code that can be described using the stabilizer formalism is called a *stabilizer code*. A stabilizer code is defined by two sets of operators, a set of *stabilizer generators* which form the *stabilizer group* $\mathcal{S}$, which is an Abelian subgroup of the Pauli group, and a set of encoded *logical operators*. A stabilizer group is the set of Pauli operators which leave all states $|\psi\rangle_i$ from the *codespace*, a subspace of the Hilbert space of n qubits spanned by the codeword basis states, invariant, such that:

$$S |\psi\rangle_i = |\psi\rangle_i, \qquad\qquad \forall S \in \mathcal{S} \qquad\qquad (2.12)$$

10

These elements of the of the stabilizer group are most simply referred to as *stabilizers*. The elements of this Abelian group can be written by a set of independent generators S_j of size N_S, where any stabilizer S can be written as

$$\mathcal{S} = \prod_j S_j^{a_j}, \qquad\qquad a_{\_}j \in 0, 1 \qquad\qquad (2.13)$$

The set of generators S_j is *independent* if no generator can be written as a product of other generators. This implies that any stabilizer can be written in terms of the bitstring $a_1, a_2, ...a_{N_S}$ and that the stabilizer group takes up n_S degrees of freedom of the Hilbert space of N qubits. The remaining $N - N_S = N_L$ degrees of freedom which are not specified by the stabilizers make up the *codespace*, the subspace spanned by the logical basis states, or the number of encoded logical qubits. Thus, if N_L logical qubits are to be encoded by n qubits, we require a total of $N_S = N - N_L$ independent stabilizer generators.

2.3.1 Encoded logical operators

Next to the set of stabilizers, we can construct the set of logical operators that will act on the encoded qubits from a set commutation rules. First of all, all logical operators must commute with all elements of the stabilizers, as a logical operator is made up from Pauli operators, and any Pauli operator which anticommutes with a stabilizer cannot leave the codespace invariant. Note that his means that a logical operator is not unique, as it can be multiplied with an element of the stabilizer. Secondly, we can impose commutation rules for the logical operators themselves based on the Pauli operators that they are representing. For example, logical $\bar{X}$ and $\bar{Z}$ operators must anticommute.

The minimum weight of the logical operator determines the distance d of the error correcting code. This is thus the minimal amount of errors that can cause a logical failure. Together with the total number of qubits n and number of logical qubits k, they provide a rough measure of the error correcting capabilities of the code.

2.3.2 Error detection procedure

As the stabilizers are a set of Pauli operators, they correspond to blip-flip or phase-flip errors that may have happened on any of our n qubits. Furthermore, as any stabilizer leaves all states $|\psi\rangle_i$ invariant, measuring the stabilizers does not disrupt the encoded information. If no error has occurred, all stabilizer measurements will return a '+1' eigenvalue, while any '-1' outcome points to the presence of errors, which we will call a *stabilizer violation.*

This outcome is dependent on whether an error caused by the error operator E, must either commute or anticommute with the stabilizer generators, since all operators are members of the same Pauli group $\mathcal{P}_n$. If E and generator S_j commute then,

$$S_j E |\psi\rangle = E S_j |\psi\rangle = E |\psi\rangle \qquad\qquad (2.14)$$

which means that the post-error state is a +1 eigenstate of S_j. If E and generator S_j anticommute then,

$$S_j E |\psi\rangle = -E S_j |\psi\rangle = -E |\psi\rangle \qquad\qquad (2.15)$$

and the post-error state is a -1 eigenstate of S_j. Errors on stabilizers codes are therefore detected by measuring the stabilizers, which returns a series of eigenvalue outcomes that is called the *syndrome*. However, it is not necessary to measure all operators in the stabilizer group $\mathcal{S}$. Measuring the set of independent stabilizer generators S_j suffices as any other stabilizer is just a combination of already measured states.

2.3.3   Error models

Any qubit can be subject to a combination of errors, each can be caused by a difference factor in our quantum system. To generalize these errors, we define certain *error models* that constricts the errors that take place. Here, we list a few models that we will encounter in this thesis.

Independent noise model   We were already introduced in the *bit-flip* and *phase-flip* errors in section 2.2.1. But let us know generalized them in the form of the density matrix ρ. Let Φ be a quantum channel that maps ρ to $\Phi(\rho)$, and let the chance of a bit-flip error be p_X, the resulting state should be

$$\Phi_X(\rho) = (1 - p_X)\rho + p_X(X\rho X). \tag{2.16}$$

Analogously, let the chance of a phase-flip error be p_Z, the resulting state should be

$$\Phi_Z(\rho) = (1 - p_Z)\rho + p_Z(Z\rho Z). \tag{2.17}$$

The bit-flip and phase-flip errors can be considered together as the *independent noise model* or the *uncorrelated noise model*. As the two types of errors are independent, they can be studied separately from each other.

The depolarizing noise model   In the *depolarizing* noise model, the afflicted quantum state is replaced by a complete mixed state with probability p_D. Let the completely mixed state be written as

$$\frac{1}{2}I = \frac{1}{2}(\rho + X\rho X + Y\rho Y + Z\rho Z), \tag{2.18}$$

then the depolarizing channel is described as

$$\Phi_D(\rho) = (1 - \frac{3}{4}p_D)\rho + \frac{p_D}{4}(X\rho X + Y\rho Y + Z\rho Z) \tag{2.19}$$

$$= (1 - p_D^*)\rho + \frac{p_D^*}{3}(X\rho X + Y\rho Y + Z\rho Z) \tag{2.20}$$

which can be interpreted as the state ρ is left untouched with probability $p_D^* = \frac{3}{4}p_D$, and each Pauli gate is applied to it with probability $1 - p_D^*$. Differently from the *independent noise model*, to optimally decode, we need to take into account correlations between X and Z errors.

The erasure noise model In the *erasure* noise model, a qubit is completely erased or lost from the system. Such a loss can be detected and the missing qubit is then replaced by a qubit in the totally mixed state (equation 2.18). The erasure channel can therefore be seen as the depolarizing channel with the extra property that it can be detected which qubits suffer the error.

2.3.4 Error correction or decoding

As the Pauli operators are self-inverse, any error E can be corrected by applying it again. From the measurement outcomes of a stabilizer measurement, we can deduce which error E must have caused the syndrome. However, this relationship is not always one-to-one, as an error E and its multiplication with a stabilizer ES will lead to an identical syndrome. This is called the *code degeneracy*. The choice of the most appropriate error to correct is not a trivial task, and algorithms that are tasked to automate this process are called *decoders*.

2.3.5 Stabilizer codes

The three-qubit repetition code we already covered can now be described in the stabilizer formalism. We had already found that the logical operators are encoded $\bar{X} = XXX$ and $\bar{Z} = ZII$. Other logical operators also exist up to a stabilizer. The stabilizer generators are needed to complete its description. We had found that two parity measurements, for example ZZI and IZZ, will identify the error, as they will either commute or anticommute with the error, as such they are a set of independent stabilizer generators.

The resulting measurement of '+1' or '-1' eigenvalues make up the syndrome. De decoder algorithm here is quite simple, but fails in the case if there is more than 1 bit-flip error. The code has $n = 3$ and $n_S = 2$ which results in the expected $n_L = 1$ encoded bits. The distance d of the is 1, which conforms that there are certain errors, in this case phase-flip errors, which this code cannot detect.

The smallest code which can correctly solve for both single bit-flip and phase-flip errors is the *5-qubit repetition code* [6]. This code has the following stabilizer generators:

$$XZZXI \qquad IXZZX \qquad XIXZZ \qquad ZXIXZ \qquad (2.21)$$

with the logical operators up to a stabilizer:

$$\bar{X} = XXXXX \qquad\qquad \bar{Z} = ZIIII \qquad (2.22)$$

This codes now has $n = 5$ bits with $n_S = 4$ stabilizers, which means it still encodes for a single bit.

With the principles of encoding and decoding in the quantum error correction in mind, we are now ready to move on to a more complicated variant of stabilizer codes, the *surface code*.

Chapter 3

The surface code
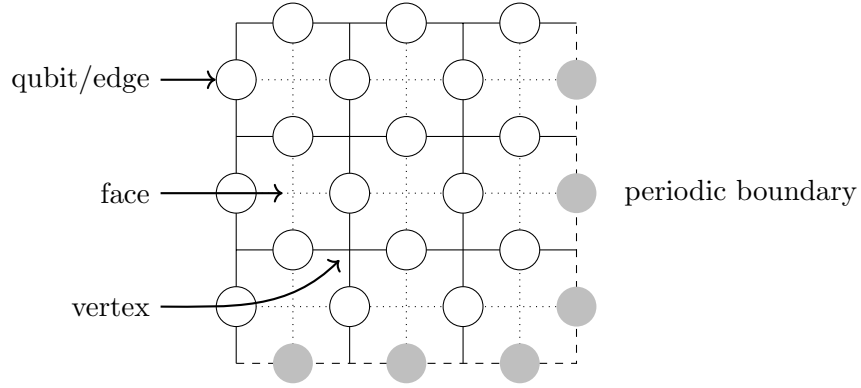
Figure 3.1: The toric code is defined as a $L \times L$ lattice (here $L = 3$) with periodic boundary conditions. The edges on the lattice, which represents the qubits, make up faces and vertices.

The variant of the stabilizer codes that we are going to explore in this thesis is Kitaev's *surface code* [7], which is of the category of *topological codes*. Among this category, the surface code is preferable as it offers the highest error tolerance under realise noise channels and requires only local stabilizer measurements of physically neighboring qubits. Two variants of the surface code will be considered here, the *toric code* in section 3.1 and the *planar code* in section 3.2, and various decoders are detailed in 3.3.

3.1   The toric code

The *toric code* is defined by arranging qubits on the edges of a square lattice with periodic boundary conditions, as seen in Figure 3.1. The name of the toric code lends itself from the torus, or donut, shape, where any point on the surface of the torus will encounter itself after traversing the torus in either x or y directions. Hence, the top edge of the toric code meets the bottom edge, whereas the left edge meets the right. On a $L \times L$ grid there are $N = 2L^2$
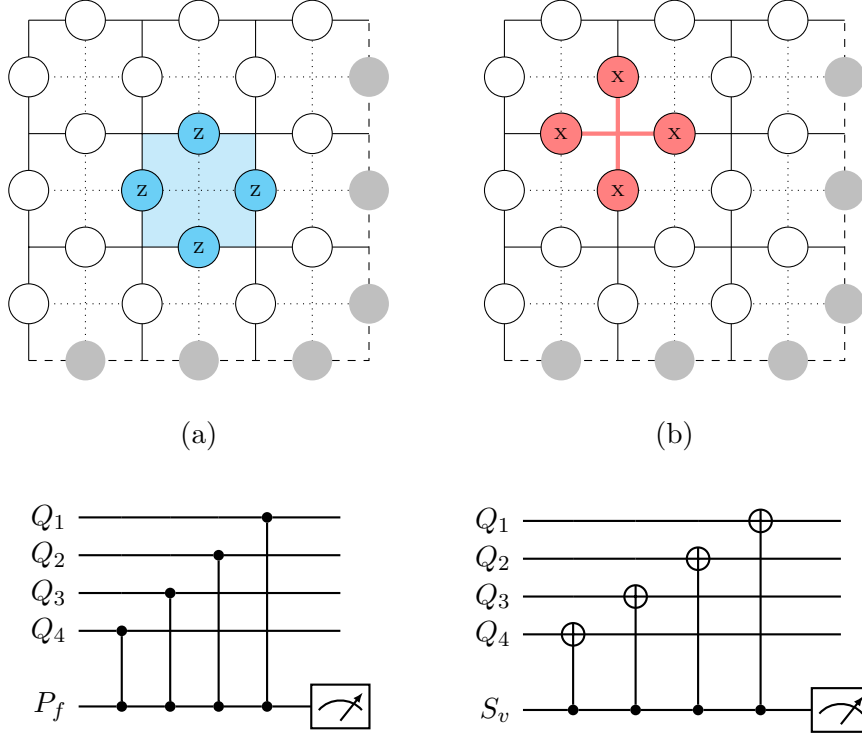
Figure 3.2: Each face (a) and vertex (b) on the lattice represents a plaquette and star operator, respectively. The non-identity single qubit operators on which they act are indicated. The set of all (but one) plaquettes and vertices make up the stabilizers of the code.

edges and the same amount of physical qubits. This topology of qubit arrangement plays an important part in encoding the logical qubits, which is stored in the non-trivial cycles on the torus. Errors, beneath a certain threshold, will only introduce local effects and does not change these cycles.

3.1.1 Stabilizer generators

To define a stabilizer code, we need to specify the m independent stabilizer generators and the encoded $\bar{X}$ and $\bar{Z}$ operators. On the toric code there are two types of stabilizer generators, *plaquette* and *star* operators, which are associated with the *faces* and *vertices* of the square lattice, respectively.

Plaquette operators For every face f on our lattice, we define a plaquette operator P_f, consisting of tensor product of Pauli Z operators on qubits on these edges (see Figure 3.1a),

$$P_f = \bigotimes_{i \in Q(f)} Z_i \tag{3.1}$$

15

where $Q(f)$ is the set of qubits touching face f. On a $L \times L$ grid there are L^2 plaquettes.

Star operators Similarly, for every vertex v on our lattice, we define a star operator S_v, consisting of tensor product of Pauli X operators on qubits neighboring the vertex (see Figure 3.1b),

$$S_v = \bigotimes_{i \in Q(v)} X_i \tag{3.2}$$

where $Q(v)$ is the set of qubits neighboring vertex v. On a $L \times L$ grid there are L^2 plaquettes.

As each plaquette and star operator needs to be measured, an ancilla qubit is needed at the physical locations of each of these operators. The structure of the full lattice is now clear, as it just a simple square arrangement of alternating data and ancilla qubits in both x and y directions.

The full stabilizer of the code $\mathcal{S}$ can be generated by multiplying elements of the generator operators. Consider two plaquette operators. These two operators will either share one boundary consisting of a qubit, or none. This means that the Pauli Z operator on the boundary qubit will add up to identity as they commute. The result is that the product of the plaquette operators will consists of the overall boundary Pauli operators of the joint plaquette (see Figure 3.3a).

However, if all plaquettes are applied to the lattice, no boundary will be left. Thus the product of all plaquettes is the identity, which means that the full set of plaquettes are not independent. The full set of plaquette generators can therefore be completed by simply removing a single plaquette from all available plaquettes. There are therefore $L^2 - 1$ independent plaquette operators.

The multiplication of star operators follow the same properties as the plaquette operators described above (see Figure 3.3b). Thus there are also $L^2 - 1$ independent star operators, which are the star generators. This sums up to $N_S = 2L^2 - 2$ independent stabilizer generators.

3.1.2   Dual lattice

Note that if we shift our lattice half a cell down, and half a cell to the right, we can create a *dual* lattice. This dual lattice has the same size and same boundary conditions as the *primal* lattice, but every plaquette in the primal lattice is a star in the dual lattice, and every star in the primal lattice is a plaquette in the dual lattice. The edges of the dual lattice are plotted with dotted lines in the figures.

This interesting property of *lattice duality* leads to the fact that plaquette and star operators are in fact the same, and we can choose from either that is best suited for the calculation. The multiplication of operators is best pictured in the plaquette picture, for example. For the square lattice in the toric code, the dual lattice is coincidentally also square. For other types of topological codes with non-square lattices, the dual lattice has a different lattice structure than the primal lattice. We will not explore these kind of lattices in this thesis.
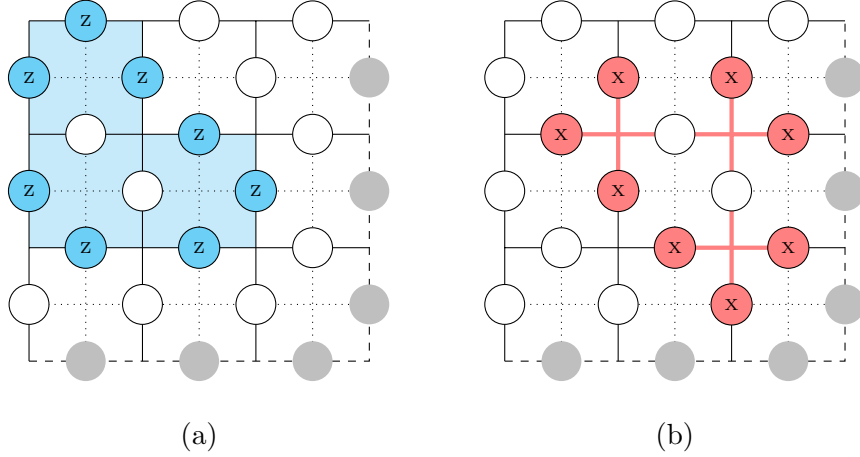
Figure 3.3: Multiplication of (a) plaquette and (b) star operators will result in a operator that consists of the Pauli operators that reside on the overall boundary of the joint plaquettes or stars.

3.1.3 Encoded qubits

Since there are $N = L^2$ qubits and $N_S = 2L^2 - 2$ independent stabilizers, we must have $N_L = N - N_S = 2$ encoded qubits and therefore 4 logical operators $\bar{X}_1, \bar{X}_2, \bar{Z}_1$ and $\bar{Z}_2$.

Recall the logical operators consists of the Pauli operators, and must commute with all stabilizer generators, but cannot be part of the stabilizer itself. We can construct the logical operators by starting with, for example, a single Pauli Z operator. It commutes with all plaquette operators trivially. In terms of the star operators, this single Pauli Z operator commutes with all but the two neighboring qubits, as all others apply to different qubits. Adding another Pauli Z operator will shift will of the anticommuting neighboring star operators. We know see that a closed loop of Z operators around the torus does not have neighboring star operators, and therefore commute with all stabilizers. As the torus has two directions we can loop over, these are the logical $\bar{Z}$ operators (see Figure 3.4a-b). Analogously, we can construct the logical $\bar{X}$ operators in the same way (Figure 3.4c-d).

Note that these logical operators are not unique. As the logical operators commute with the stabilizers, these $\bar{X}$ and $\bar{Z}$ operators can be multiplied with e.g. a plaquette or star operator, respectively, which create a diversion from its original path. But as the path still loops around the torus, this is still a valid logical operator.

The logical operators have a minimum length of L qubits, which is also the distance of the toric code. The toric code is therefore a $[L^2, 2, L]$ in the [n,k,d] notation. This implies that the toric code might be more robust against errors if the size of the lattice is increased. Later we will see that this is also very much dependent on the type of decoder that is used, and that different decoders will lead to different regimes of error for which this reasoning is true.
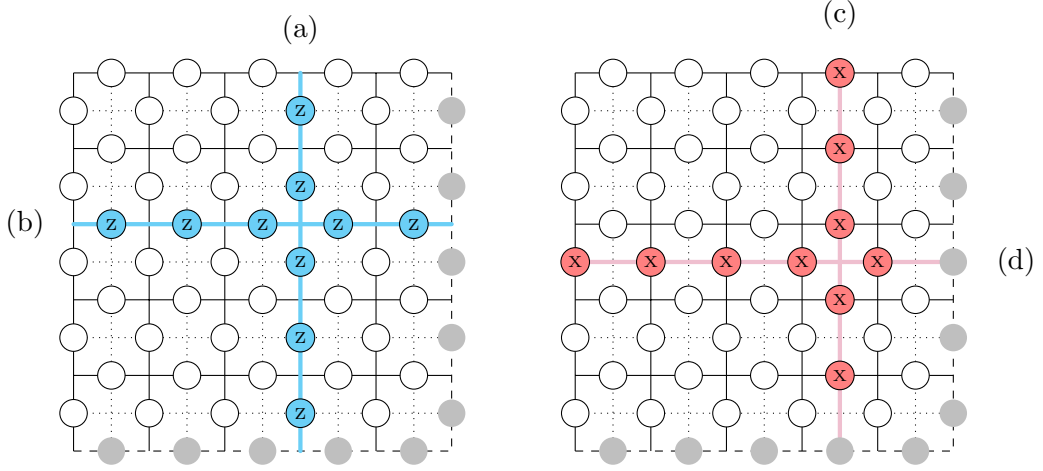
17

Figure 3.4: The logical (a) $\bar{X}_1$, (b) $\bar{X}_2$, (c) $\bar{Z}_1$ and (d) $\bar{Z}_2$ operators are the closed loop of X and Z operators, respectively, that go around the two boundaries of the torus.

3.1.4   Error detection

As discussed in the previous chapter, errors are detected by measuring the set of stabilizer generators. As we have seen in the previous section, this consists of all but one plaquette operators P_f and all but one star operators S_v. Let us first consider to measure all of them.

In the case of a single Z error (Fig 3.5a.i), the neighboring plaquette operators will commute with this error, as it consists of Pauli Z operators itself. But the neighboring star operators anticommutes with this error according to equation 2.15. Similarly, a single X error (Fig 3.5a.ii) commutes with neighboring star operators but anticommutes with neighboring plaquette operators. A Y error is a combination of X and Z operators and therefore anticommutes with all neighboring generator operators (Fig 3.5a.iii).

In the case of two Z errors (Fig 3.5a.iv), the star operators between the two errors now commute with the errors, creating a virtual path between them. This is a general property: given any string of errors, the generator operators at the end of the string will anticommute with the errors and measure -1. For Z errors, star operators at the end of strings on the primal lattice will measure -1. The detection of X errors occur in the same way, albeit now the strings of errors is defined on the *dual* lattice, and plaquette errors will measure -1 at the end of these strings.

Since Z and X errors independently affect different types of stabilizer measurements (stars and plaquettes, respectively), these two types of errors can be considered independently in two error correction processes. The two processes are analogous, up to the duality of the lattice. Therefore, for the remainder of the section, only Z errors, which leave a string of errors on the primal lattice, will be considered.

18
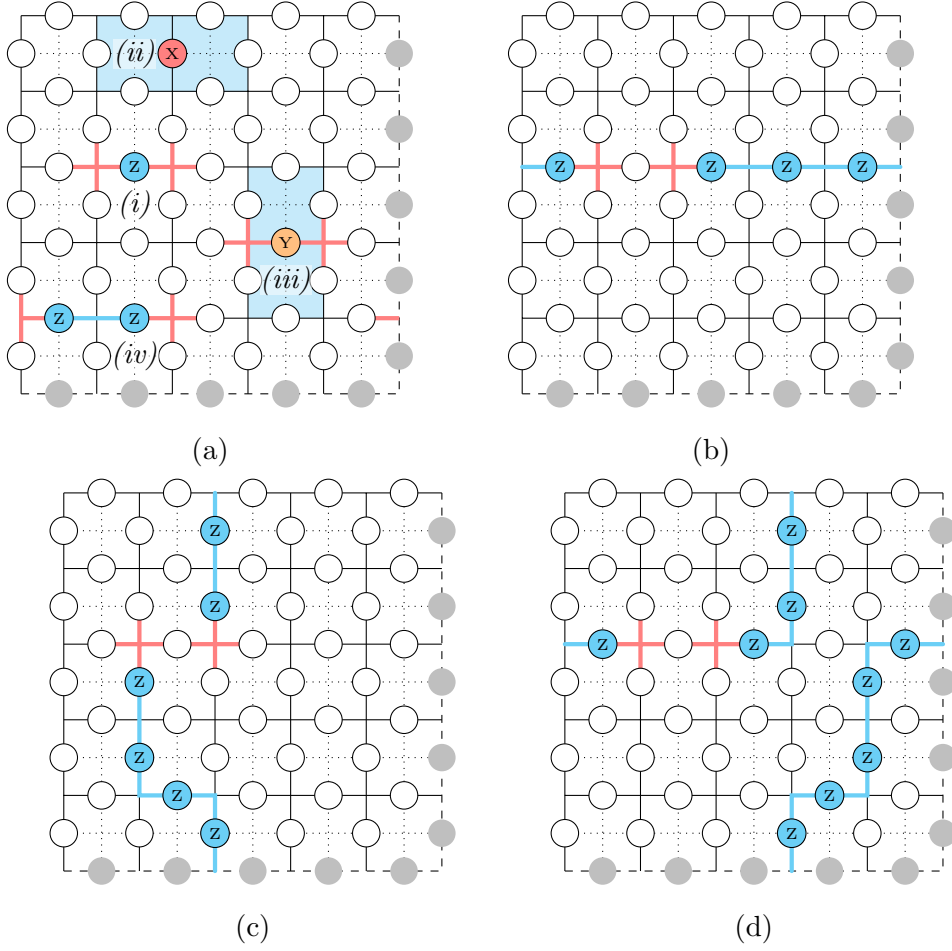
Figure 3.5: (a) Stabilizer generators that anticommute with the error will measure -1, which are (i) the neighboring star operators for a Z error, (ii) the neighboring plaquette operators for an X error, and (iii) both star and plaquette operators for a Y error. In the case of a string of errors (iv), only the stabilizer generators at the end of these strings will anticommute with the error. Due to code degeneracy, the single Z error in (a.i) E has the syndrome as (b) $E\bar{Z}_1$, (c) $E\bar{Z}_2$ and (d) $E\bar{Z}_1\bar{Z}_2$.

3.1.5   Error correction

An error E can be corrected by applying it again to the lattice. The error operator E is however unknown. We must therefore try to identify the correct operator given the measured syndrome. As mentioned in the previous chapter, this relationship between error does not always map one-to-one, which it is not in the surface code. An error E can be multiplied with some operator L that commutes with the stabilizer and they will result in the same syndrome.

If L is in the stabilizer $\mathcal{S}$, the product of the identified correction operator $C = E'$ with the real error operator E will leave the code invariant. The resulting operator $CE = L$ is a stabilizer operator. However, the encoded logical operators also commute with the stabilizer, which means that E, $E\bar{Z}_1$, $E\bar{Z}_2$, $E\bar{Z}_1\bar{Z}_2$ will all lead to the same syndrome (Fig 3.5a-d). Any identified correction operator C can therefore be categorised into four classes of operators, of which only one includes the correct logical operator. The task of choosing most appropriate correction chain is up to the decoders (section 3.3).

3.1.6   Quasiparticle picture

The processes of error detection and correction can alternatively be presented in the *quasiparticle picture*, where the anticommuting stabilizer measurements act like excitations on the lattice, which behave like the quasiparticles *anyons*. A single error creates a pair of anyons, and a chain of errors causes movement of the anyon on the lattice. A pair of anyons can also annihilate each other when two error chains merge. The correction of errors can thus be viewed of movement of the correction chains until all anyons are annihilated. The quasiparticle picture removes the distracting underlying lattice from the problem, and decoding becomes simply identifying the right pairing between anyons to minimize the chance of a logical error.

(a)   (b)   (c)
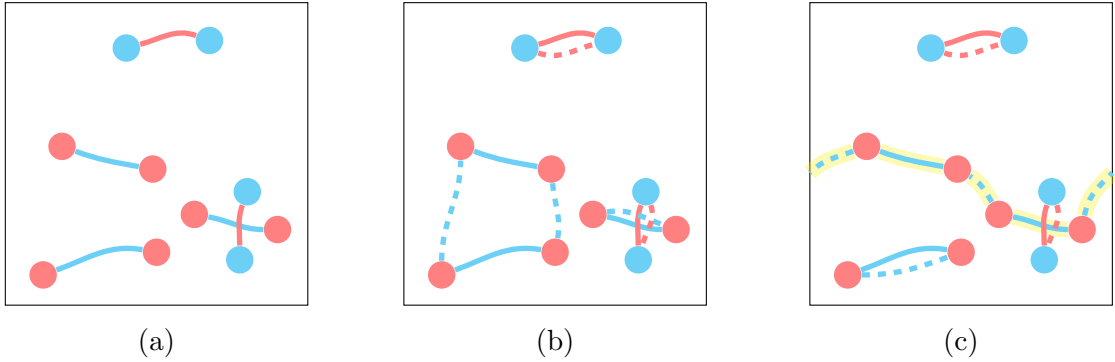
Figure 3.6: The quasiparticle picture of stabilizer measurements. Anticommuting stabilizers behave as anyons (circles), where a chain of errors (lines) creates a pair of anyons. Figure (b) shows a successful decoding of (a). Figure (c) shows a pairing that resulted in a correction operator that is in a different class as the error operator, which acquires a logical error.

Figure 3.6a shows the quasiparticle representation of the errors suffered in Figure 3.5a, which has suffered Z (blue lines) and X errors (red lines). The corresponding anyons can either be of the star type (red circle) or plaquette type (blue circle). Figure 3.6b shows a successful decoding. Note that here not all pairs are correctly identified, but the resulting loop still is in the same class of operators. In figure 3.6c the correction has failed as the resulting loop in the correction is in a difference class compared to the error. As the loop still commutes with the stabilizer, no error can be detected, but the encoded qubit has acquired a logical error.

3.1.7   Code threshold

Since the distance d of the toric code on a $L \times L$ is L, we would expect that we can improve the robustness of the code by increasing the lattice size L. However, this also increases the total number of errors in the lattice, that adds an increased level of complexity in choosing the correct correction operator.

In practice, there is a trade-off between the positive effect of a larger code distance and the negative effect of larger number of errors. When the error rate p is low, the positive effect outweighs the negative and increasing the lattice L will increase the probability of successful error correction p_C. When the error rate is large, the negative effect outweighs the positive and increasing p will decrease p_C. The point of transition in the error rate is called the *code threshold* p_{th}.

The code threshold is not the only parameter that determines the potential of a certain code for practical use. The behavior for error rates far below the threshold is also important, as is the number of physical qubits needed to achieve the sought after level of error suppression. Nevertheless, the code threshold provides us with a very easy and useful tool to benchmark different codes and different decoding algorithms, and to compare them with each other. Therefore, in this thesis we will heavily rely on the value of the code threshold. The value of the threshold is heavily dependent on the chosen error model and the physical conditions of the stabilizer measurements. To compare different decoding algorithms, we therefore will use independent and identically distributed errors (i.i.d. noise), which is the *independent noise model* from section 2.3.3.

For the toric code, when the only source of errors is i.i.d. noise under the independent noise model, and all measurements can be made perfectly, the *optimal threshold* has been proven to be 10.9% (see section 3.3.1). However, to achieve this value, one needs to consider all possible error configurations on the lattice to identify the correction operator C that is most likely to be equal to the error operator E. This is a computationally heavy task that scales exponentially with the lattice size. It is therefore an impractical approach in reality.

Luckily, there exists other decoding algorithms that can find a solution much faster, albeit at the cost of reducing the code threshold. Edmond's *Minimum Weight Perfect Matching* (MWPM) decoder scales cubic with the system, which allows for faster decoding, and achieves a code threshold of 10.3% (section 3.3.2). Including faulty measurements the threshold drops down to 2.9%. The *Union-Find* decoder is a relatively new addition to the set of decoders for the surface code. It scales *almost* linearly with the system, and

has a code threshold of 9.9% (section 3.3.3). In this thesis, we will try to combine certain properties of different decoders. In particular, we have created a heuristic for minimum weight which can be applied to the Union-Find decoder.

3.2 The planar code

Another variant of the surface code is the *planar code*, which disposes the periodic boundary conditions of the torus. This allows the qubits to be placed onto a flat 2D surface. For real systems in which the qubits physically interact with each other, this is a huge benefit. Therefore, in this thesis, we will consider both toric and planar variants of the surface code.
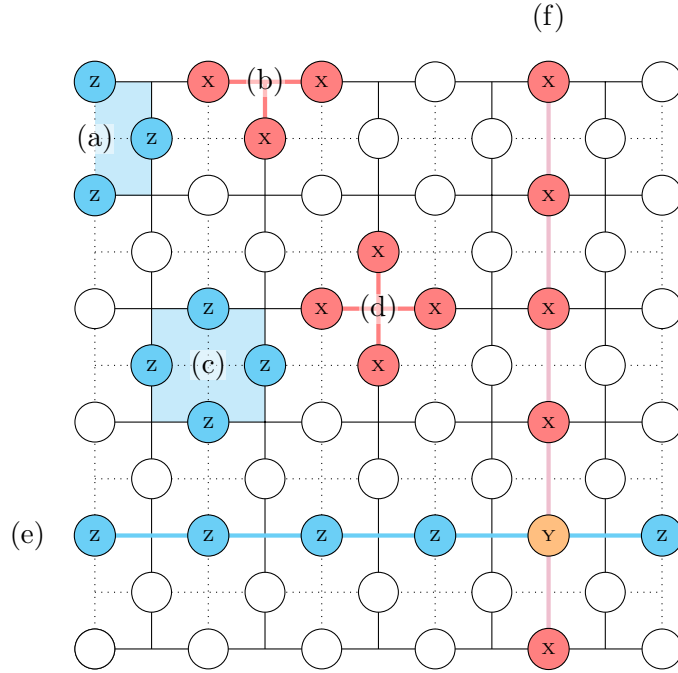
Figure 3.7: The planar code with lattice size $L = 4$, which includes $N = 2L^2 - 2L + 1$ qubits and $N_S = 2L^2 - 2L$ independent stabilizers. The boundary is defined by the (a) EDGE-plaquette and (b) EDGE-star operators, which exist next to the known (c) plaquette and (d) star operators, similar to the toric code. The planar codes encodes 1 logical qubit, which is represented by the logical (e) $\bar{Z}$ and (f) $\bar{X}$ operators.

Stabilizer generators There are a few key differences between the planar and toric codes. First of all, a new type of stabilizer generators define the non-periodic boundary of the lattice, which are referred to as *EDGE operators*. These EDGE operators have only 3 neighboring qubits and are therefore the tensor product of 3 Pauli operators. The EDGE-plaquette operators lie at the east and west boundaries of the lattice (Figure 3.7a) and

the EDGE-star S_{vE} operators lie at the north and south boundaries of the lattice (Figure 3.7b). In the middle of the lattice, the bulk of the stabilizer generators still consist of 4 Pauli operators, identical to the ones in the toric code (Figure 3.7c-d). Note that the stabilizer generators are still defined by equation 3.1 and 3.2, but now the relevant faces and vertices contain three neighboring qubits.

Stabilizer violoations A second key difference is that now not all errors will cause two stabilizer violations. In the bulk of the qubits on the lattice, a single error will still cause two neighboring stabilizers to measure -1, or create two anyons. At the boundary however, it now may be the case that an error is only included in one plaquette or star operator. This will also mean the decoding in the quasiparticle picture requires a slightly different approach.

Logical qubits Furthermore, we can inspect that a planar surface of dimension L has $N = 2L^2 - 2L + 1$ physical qubits. We can also find that there are $2L^2 - 2L$ stabilizer generators. As the boundary is now non-periodic, all generators are now independent, and therefore the number independent generators is $N_S = 2L^2 - 2L$. This means that the planar code encodes $N_L = N - N_S = 1$ a single logical qubit. The logical $\bar{X}$ and $\bar{Z}$ operators are pictured in Figure 3.7e-f.

Other properties of the planar code are very similar to the toric code. The *dual lattice* also exists for the planar code, for example. But the dual lattice exists at a 90 angle compared to the primary lattice. Also, as the bulk of the lattice still consists of 4-Pauli operator stabilizers, the decoding algorithms for the planar code is very similar to the toric code. It is due to the boundary that some slight alterations are needed, as we will see in the next section.

3.3 Decoders

3.3.1 The optimal decoder

3.3.2 Minimum Weight Perfect Matching

3.3.3 The Union-Find decoder

Even the fastest MWPM algorithms still have a quadratic time complexity of $\mathcal{O}(n^2\sqrt{n})$, where n is the number of qubits. In order to realistically utilize a decoder with increasing decoding success rates using increasing lattice size, we would need to have a better time complexity. Luckily, an alternative algorithm called the Peeling Decoder has been developed which can solve errors over the erasure channel with a linear time complexity $\mathcal{O}(n)$ [8]. The Union-Find Decoder builds on top of the Peeling Decoder to solve for Pauli errors with a time complexity of $\mathcal{O}(n\alpha(n))$, where α is an inverse Ackermann function, which is smaller

than 3 for any practical input size [9]. However, these algorithms have a tradeoff in the form of a decrease in the error threshold, and has the reported value of $p_{UF} = 9.2\%$.

A topic of interest will be weighted growth function for the Union-find decoder. This function of the algorithm will increase the error threshold to $p_{UF} = 9.9\%$, but has not been fully described in its publication. In this section, we will describe the original Peeling decoder and the Union-Find decoder.

The Peeling decoder

Let $\varepsilon \subset E$ be an erasure, a set of qubits on which an erasure error occurs, and let $\sigma \subset S$ be the measured error syndrome, the subset of stabilizer generators which anticommute with the erasure errors. In the absence of Pauli errors, all errors P must lie inside the erasure. Therefore, for any pair of stabilizer generators in σ, the path of errors must also be in the erasure, which can be denoted by $P \subset \varepsilon$. Furthermore, due to the fact that errors P are randomly distributed, any coset of errors and stabilizers $P \cdot S$ that solves the error syndrome σ is the most likely coset. These features of an erasures forms the basis of the Peeling decoder. In order to find a coset of $P \cdot S$, the decoder reduces the size of the erasure by peeling edges from the erasure, while keeping the syndromes at the new boundary of the erasure. Elements of the syndrome can be moved by applying an correction on the adjacent qubit. At the end, the entire erasure is peeled or removed, and all corrections will have removed the errors up to a stabilizer.

use capitals for sets Σ, and lowercase σ for elements in the set

We will now describe the Peeling decoder as is presented in Algorithm 1. In step 1, we will remove all cycles present in ε. We construct a spanning forest F_ε inside erasure ε, the maximal subsect of edges of ε that contains no cycles and spans all vertices of ε. From here, we loop over all edges in F_ε (step 3), starting at a leaf edge $e = \{u, v\}$, removing the leaf edge from F_ε (step 4), and conditionally add the edge to the correction set κ if the pendant vertex u is in σ (step 6). If the correction is applied immediately, we can see that the pendant vertex u is removed from σ and that the value of v is flipped in σ. Edges on a branch in the forest will be added to κ until $v \in \sigma$, or a generator will be continuously moved from u to v until it encounters another generator, creating a correction path between two syndrome pairs.

Lemma 3.1 *A forest F is an undirected graph in which any two vertices are connected by at most one path. A forest F_G of some graph G is one in which some edges is removed such that there are no cycles in the graph, which satisfies the forest requirement.*

initial state

indentify syndrome

unerased edge
X edge with Pauli error
erased edge
syndrome
correction edge

construct F_ε

peel $e = \{u, v\}, u \notin \sigma$

peel $e = \{u, v\}, u \in \sigma$

flip v, add e to κ

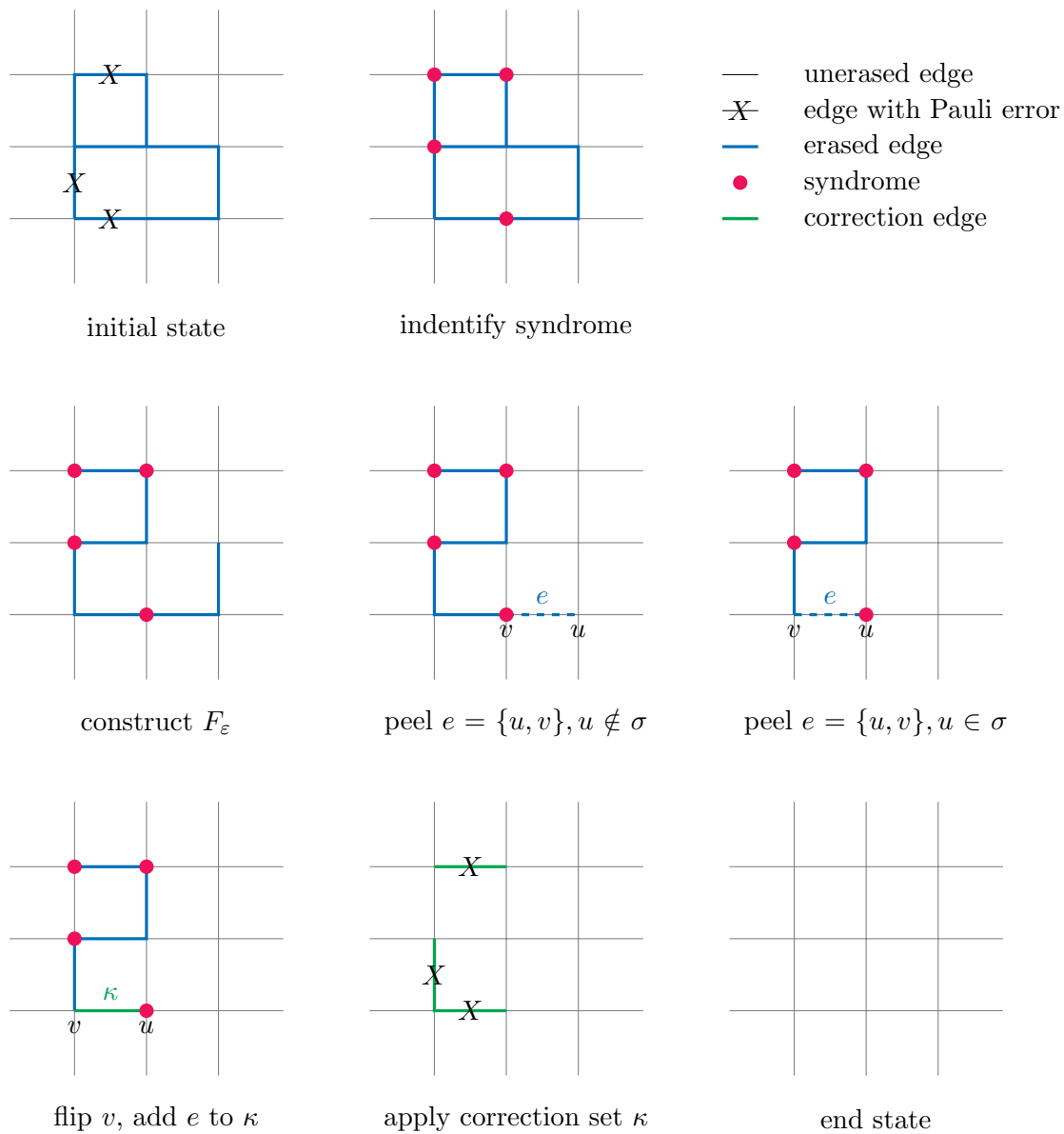apply correction set κ

end state

Figure 3.8: Schematic representation of the Peeling decoder.

Algorithm 1: Peeling decoder [8]

Data: A graph $G = (V, E)$, an erasure $\varepsilon \subset E$ and syndrome $\sigma \subset V$
Result: Correction set $\kappa \subset E$

1 construct a spanning forest F_ε of ε
2 initialize κ by $\kappa = \varnothing$
3 while $F_\varepsilon \neq \varnothing$ **do**
4   pick a leaf edge $e = u, v$ with pendant vertex u, remove e from F_ε
5   **if** $u \in \sigma$ **then**
6   add e to κ, remove u from σ and flip v in σ
7   **else:**
**8**   do nothing

9 return κ

Note that due the flipping of both u and v in σ, the parity of the number of generators in σ is always preserved. The peeling decoder can therefore always solve erasures with an even parity, as the size of σ will drop until at the end the syndrome will be empty $\sigma = \varnothing$, and call errors are corrected up to a stabilizer. This is always the case in the absence of measurement and Pauli errors, as all errors within the erasure either add or remove an even number of generators to or from σ.

Time complexity of the Peeling decoder The spanning forest F_ε can be constructed in linear time. Also, the loop over the forest can be operated in linear if the list of leaves is pre-computed and updated during the loop. Thus the Peeling decoder has a linear time complexity in the size of the erasure $\mathcal{O}(|\varepsilon|)$ and therefore also in the number of qubits $\mathcal{O}(n)$.

Growing erasures

Now in the presence of Pauli errors, errors can occur on edges that are now not part of the erasure, and odd parity clusters can occur. Clusters that consists from only a single generator also exist, which are just end-points of syndromes caused by Pauli errors. We must therefore make an erasure ε from the syndrome σ that is compatible with the peeling decoder, which contains only even parity clusters. To do this, we can attractively grow the clusters with an odd parity by an half-edge on the boundaries on the clusters. When two odd parity clusters meet, the merged cluster will have a even parity, and can now be solved by the peeling decoder.
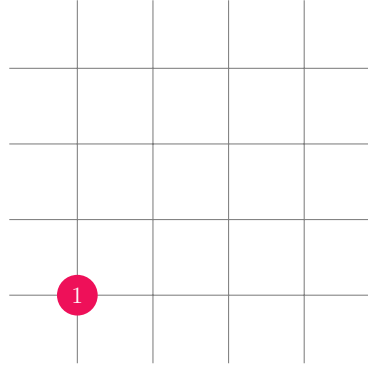
Union-Find algorithm

To keep track of the vertices of a cluster, it will be represented as a *cluster tree*, where an arbitrary vertex of the cluster will be the root, and any other vertex will be a child of the root. Whenever an edge (u, v) is fully grown, we will need to traverse the trees of the two vertices u and v, and check whether they have the same root; whether they belong to the same cluster. If not, a merge is initiated by making the root of smaller cluster a child of the bigger cluster. These functions, `find` and `union` respectively, are part of the Union-Find algorithm (not to be confused with the Union-Find decoder) [10].
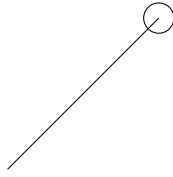
Within the Union-Find algorithm, two features ensure that the complexity of the algorithm is not quadratic. 1). With **path compression**, as we traverse a tree from child to parent until we reach the root, we make sure that each vertex encountered that we have encountered along the way is pointed directly to the root. This doubles the cost of the `find`, but speeds up any future call to any vertex on the traversed path. 2). With **weighted union**, we make sure to always make the smaller tree a child of the bigger tree. This ensures that the overall length of the path to the root stays minimal. In order to make this happen, we just need to store the size of the tree at the root.

Data structure   Now it is clear what information is exactly needed to grow the clusters using the Union-Find algorithm. We will need to store the cluster in a sort of cluster-tree. At the root of each tree we store the size and parity of that cluster in order to facilitate weighted union and to select the odd clusters. We will need to store the state of each edge (empty, half-grown, or fully grown) in a table called `support`. And we need to keep track of the boundary of each cluster in a `boundary` list.

27

The routine The full routine of the Union-Find decoder as originally described ([9], Algorithm 2) is listed in Algorithm 2. In line 1-2, we initialize the data structures, and a list of odd cluster roots $\mathcal{L}$. We will loop over this list until it is empty, or that there are no more odd clusters left.

In each growth iteration, we will need to keep track of which clusters have merged onto one, therefore the fusion list $\mathcal{F}$ is initialized in line 4. We loop over all the edges from the boundary of the clusters from $\mathcal{L}$ in line 5, and grow each edge by an half-edge in support. If an edge is fully grown, it is added to $\mathcal{F}$.

For each edge (u, v) in $\mathcal{F}$, we need to check whether the neighboring vertices belong to different clusters, and merge these clusters if they do. This is done using the Union-Find algorithm in line 6. We call find(u) and find(v) to find the cluster roots of the vertices. If they do not have the same root, we make one cluster the child of another by union(u,v). Note that this does not only merge two existing clusters, also new vertices, which have themselves as their roots, are added to the cluster this way. We also need to combine the boundary lists of the two clusters.

Finally, we need to update the elements in the cluster list $\mathcal{L}$. First, we replace each element u with its potential new cluster root find(u) in line 7. We can avoid creating duplicate elements by maintaining an extra look-up table that keeps track of the elements $\mathcal{L}$ at the beginning of each round of growth. In line 8, we update the boundary lists of all the clusters in $\mathcal{L}$, and in line 9, even clusters are removed from the list, preparing it for the next round of growth.

Algorithm 2: Union-Find decoder [9]

Data: A graph $G = (V, E)$, an erasure $\varepsilon \subset E$ and syndrome $\sigma \subset V$
Result: A grown erasure ε' such that each cluster $\gamma \subset \varepsilon$ is even

1 initialize cluster-trees, support and boundary lists for all clusters
2 initialize list of odd cluster roots $\mathcal{L}$
3 **while** $\mathcal{L} \neq \varnothing$ **do**
4   initialize fusion list $\mathcal{F}$
5   for all $u \in \mathcal{L}$, grow all edges in the boundary list of cluster C_u by a half-edge in support. If the edge is fully grow, add to fusion list $\mathcal{F}$
6   for all $e = u, v \in \mathcal{F}$, if *find(u)* $\neq$ *find(v)*, then apply *union(u,v)*, append boundary list
7   for all $u \in \mathcal{L}$, replace u with *find(u)* without creating duplicate elements
8   for all $u \in \mathcal{L}$, update the boundary list
9   remove even clusters from $\mathcal{L}$
10 run peeling decoder with grown erasure ε'

Time complexity of the Union-Find decoder

Chapter 4

Modifications to the Union-Find decoder

For the UF decoder, each cluster C_α is represented by a set of vertices $\mathcal{V}_\alpha = \{v_1, v_2, v_3...v_{C_s^\alpha}\}$, where S_α is the size of the cluster. Here, the $\mathcal{V}_\alpha$ is stored in a tree, and each tree root is a unique identifier of the cluster. When new vertices v_{new} are added during $\texttt{Grow}(C_\alpha)$, they are added to the tree as a child of the root. When an edge is fully grown, we add it to a fusion list $\mathcal{F}$, and for all edges in $\mathcal{F}$ the vertex tree for the two neighboring vertices v_x, v_y are traversed to their roots using $\texttt{Find}(v_x)$ and $\texttt{Find}(v_y)$ respectively. If $\texttt{Find}(v_x) \neq \texttt{Find}(v_y)$ the cluster are merged using $\texttt{Union}(v_x, v_y)$ by making one vertex a child of another's root. The depth of the tree $\mathcal{V}^\alpha$ is kept low due to *path compression* and *weighted union* of clusters.

The vanilla UF decoder (as described by Delfosse [9]) has an error threshold of 9.2% for a 2D toric lattice, that only suffers errors through a single Pauli channel. Delfosse has shown that the threshold can be improved by sorting the order of cluster growth, but has not provided a description of this sorting. In this chapter, we will show an implementation of this sorting routine that maintains a linear time complexity in section 4.2. In section 4.1, we will show an object oriented approach of the UF decoder that allows for a straight forward data structure that is used for our implementation. In the remaining sections, we will show some other alterations to the UF decoder, that uses the inspiration of the MLD-decoder or the MWPM decoder to improve the error threshold while retaining a low time complexity.

4.1 Object oriented approach

Others who have implemented weighted growth (wrongly) use an algorithm that has a time complexity of $\mathcal{O}(n \log n)$, which is worse than the main algorithm [11]. We will introduce a weighted growth algorithm that has a linear time complexity, and therefore preserving the inverse Ackermann time complexity of the Union-Find decoder.

4.1.1 A new data structure

4.1.2 Finding clusters

4.2 Bucket Cluster Sort (BCS)

To further increase the error threshold for the Union-Find decoder from 9.2% to 9.9%, Delfosse implements *weighted growth*, where clusters are grown in increasing order based on their sizes [9]. However, the main problem with weighted growth is that the clusters now need to be sorted, and that after each growth iteration another round of sorting is necessary, due to the fact that the clusters have changed sizes due to growth and merges, and the order of clusters may have been changed. Nickerson has not given a description of how weighted growth is implemented. As the complexity of the algorithm is now dominated by the Union-Find algorithm, we need to make sure that weighted growth does not add to this complexity. To avoid this iterative sorting, we need to make sure that the insertion of a new element in our sorted list of clusters does not depend on the values in that list.

The Bucket Cluster sorting algorithm as described in this section is evolved from a more complicated version that is described in appendix A, which has a sub-linear complexity of $\mathcal{O}(\sqrt{n})$.

4.2.1 How to sort for weighted growth using BCS

Let us now first look at what weighted growth for the Union-Find decoder exactly does. When a cluster is odd, there exists at least one path of errors connecting this cluster to a generator outside of this cluster. When the cluster grows, a number of edges k that is proportional to the size S of the cluster is added to the cluster. If $k \propto S$ new edges are added, only $1/k$ of these edges will correctly connect the cluster with the generator. Therefore, more "incorrect" edges will be added during growth of a larger cluster.

Note however, that the benefit of growing a smaller cluster is not substantial if the clusters are of similar size. Take two clusters C_α, C_β with size $S_\alpha << S_\beta$, growth of cluster C_β will add $\sim k_\beta/2$ "incorrect" edges on average, whereas growth of cluster C_α will add $\sim k_\alpha/2 << k_\beta/2$ edges as $k_\alpha \propto S_\alpha$ and $k_\beta \propto S_\beta$. However, if $S_\alpha \simeq S_\beta$, the number of added "incorrect" edges for both clusters will also be similar, and it is the same when $S_\alpha = S_\beta$.

Lemma 4.1 *For two clusters C_α, C_β with size $S_\alpha << S_\beta$ the number of vertices in the clusters, $Grow(S_\beta)$ will add a smaller amount of* incorrect *edges to the cluster, which are edges that are not part of the matching.*

The sorting method that is suited for our case is *Bucket sort*. In this algorithm, the elements are distributed into k buckets, after which each bucket is sorted individually and the buckets are concatenated to return the sorted elements. Applied to the clusters, we sort the odd-parity clusters into k buckets, which replaces the odd cluster list $\mathcal{L}$. As the sizes of the clusters can only take on integer values, each bucket can be assigned a clusters size, and sorting of each individual bucket is not necessary. Furthermore, as we are not interested in the overall order of clusters, concatenating of the buckets is not necessary.

Growing a bucket

The procedure for the Union-Find decoder using the bucket sort algorithm is now to sequentially grow the clusters from a bucket starting from bucket 0, which contain the smallest single-generator clusters of size 1. After a round of growth, in the case of no merge event, these clusters are grown half edges, but are still size 1. We would therefore need twice as many buckets to differentiate between clusters without and with half-edges. Let us call them full-edged and half-edged clusters, respectively. Starting from bucket 0, even buckets contain full-edged clusters and odd buckets contain half-edged clusters of the same size. To grow a bucket, clusters are popped from the bucket, grown on the boundary, after which the clusters is to be distributed in a bucket again in a subroutine named `Place`.

$$\texttt{Place}(C) = \begin{cases} C \to b_{2(S_C - 1)}, & \text{if } S_C \text{ even} \\ C \to b_{2(S_C - 1)+1}, & \text{otherwise} \end{cases} \tag{4.1}$$

In the case of no merge event, clusters grown from even bucket b_i must be placed in odd bucket b_{i+1}, as it does not increase in size, and clusters grown from odd bucket j must be placed in even bucket b_{j+2k+1} with $k \in \mathbb{N}_0$ the number of added vertices. Also in the case of a union event of clusters C_α and C_β, the new cluster $\texttt{union}(C_\alpha, C_\beta) = C_{\alpha\beta}$ must be placed in a bucket $b_{\alpha\beta} > b_\alpha, b_{\alpha\beta} > b_\beta$. Thus we can grow the buckets sequentially, and need not to worry about bucket that have been already "emptied". This ensures that for two clusters C_α and C_β with $S_\alpha < S_\beta$, cluster A will be grown first, adding a fewer amount of "incorrect" edges as per lemma 4.1. Clusters of the same size $S_\alpha = S_\beta$ are placed in the same bucket and their order of growth is dependent on their order of placements.

All clusters within the same bucket are grown "together"; we first grow all the boundary edges of the clusters in the bucket by half, adding all fully grown edges to the fusion list $\mathcal{F}$ and check for the union and new boundary edges for all clusters together per algorithm 2. The order of growth within the bucket is dependent on the order of cluster placement into the bucket.

Theorem 4.1 *Weighted growth is achieved by growing the odd clusters sequentially starting from bucket b_0. Grown odd clusters from bucket b_c are added back to the bucket list using the `Place` subroutine, in a bucket b_g where $g > c$. Clusters C_α and C_β with $S_\alpha = S_\beta$ are placed int the same bucket b_{S_α}, and are grown together. However, their growing order is dependent on the order of placement within the bucket.*

Faulty entries

Now let us be clear: *only odd parity clusters will be placed in buckets, but each bucket does not only contain odd parity clusters.* As a merge happens between two odd parity clusters C_α and C_β during growth of C_β, cluster C_α has already been placed in a bucket, as it was still odd after its growth. But cluster C_α is now part of cluster AB and has even parity, and the entry of cluster C_α is faulty. To prevent growth of the *faulty entry*, we can check for the parity of the root cluster.
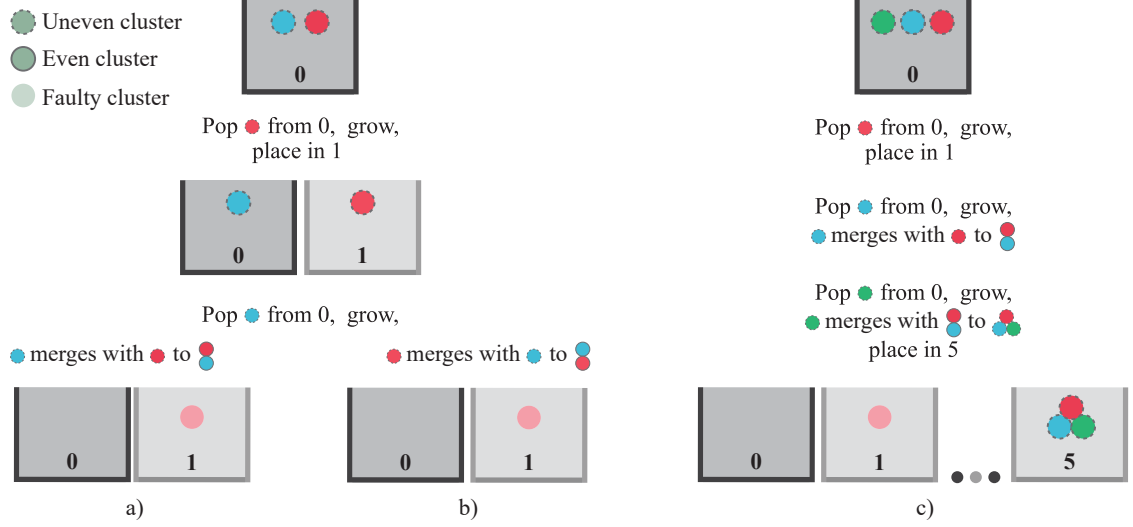
Figure 4.1: Faulty entries of clusters can occur in the buckets, a) cluster that should not be there due to a merge event. Situation a can be solved by checking the parity of the cluster. Checking the parity of the root cluster solves a) and b). Checking the bucket_number of the root cluster solves all.

Furthermore, it is possible that another cluster C_γ merges onto $C_{\alpha\beta}$, such that the cluster $C_{\alpha\beta\gamma}$ is odd again. Now, the faulty entry of cluster A passes the previous test. To solve this issue, we store an extra bucket number C_b at the root of a cluster. Whenever a cluster increases in size or merges to an odd parity cluster, we first update the C_b to the appropriate value and place it in its bucket. If the cluster merges to an even parity cluster, we update the C_b to *Null*. Now, every time a cluster is popped from bucket i, we can just check weather the current bucket corresponds to the C_b of the root cluster.

Lemma 4.2 *Each bucket b_i does not necessary contain clusters that still belong to b_i. Growth of these faulty entries are prevented by storing the bucket number j at the cluster $C_b = j$ during* `Place` *and checking for $i = j$ and odd cluster parity add the beginning of* `Grow`.

Number of buckets

How many buckets do we exactly need? On a lattice there can be n vertices, and a clusters can therefore grow to size n, spanning the entire lattice. Naturally, if a cluster spans the entire lattice, the solution given by the peeling decoder is now trivial. But we need to make sure that the decoder *can* give a solution. Consider an odd cluster C_μ of size S_α $n/2$ which covers half the lattice. There must exists another odd cluster C_β for matchings to exists, which has size $S_\beta \leq n/2$. As per lemma 4.1, C_β will grow before C_α. As the remaining

	L even	L odd
Toric	$S_\mu = L \times (\frac{L}{2} - 1) - 1$	$S_\mu = L \times (\frac{L'}{2} - 2) + (\frac{L'}{2} - 1)$
Planar	$S_\mu = L \times (\frac{L}{2} - 1)$	$S_\mu = L' \times \frac{L'}{2} - 1$

Table 4.1: The maximum cluster size S_μ for which it is not certain that another cluster will merge onto the current cluster, or the maximum cluster size for which a cluster is allowed to grow.

number of vertices is $n - S_\alpha - S_\beta$, C_β can never grow larger than C_α and will merge into C_α if no other odd cluster exists. There exists a maximum cluster size S_μ for which after $\texttt{Grow}(C_\mu)$ this is true. This cluster size S_μ is dependent on the code and the parity of lattice size L. We illustrate in figure 4.2 the clusters C_μ for the toric and planar code. Their maximum odd cluster size S_μ is listed in table 4.1, where $L' = L - 1$ for the planar code.

Lemma 4.3 *Once an odd cluster C_α has reached a size $S_\alpha > S_\mu$, it is certain that a smaller cluster C_β will grow in size before the bucket of C_α is reached, and it will merge into an even cluster $\texttt{Union}(C_\alpha, C_\beta) = C_{\alpha\beta}$.*

This maximum cluster size S_μ for growth determines the number of buckets $k + 1$ we will need.

$$k = 2(S_\mu - 1) \tag{4.2}$$

Any cluster with size $S \leq S_\mu$ will be placed into a bucket according to equation 4.1. If $S > S_\mu$, the cluster will not be placed into a bucket, and shall be assigned bucket number $C_b = Null$, as there is no bucket available.

Largest bucket occurrence

Not all buckets will be filled depending on the configuration of the lattice. It would therefore be redundant to go through all buckets just to find out that the majority of them is empty. To combat this, we can keep track of the largest filled bucket b_M. Whenever a bucket b_i has been emptied and $i = M$, we can break out of the bucket loop to skip the remainder of the buckets.

4.2.2   Complexity of BCS

Let us focus on the operations on a single cluster before it is grown an half-edge. A cluster is placed in a bucket, popped from that bucket some time after, checked for faulty entry, and if passed grown. All these operations are done linear time $\mathcal{O}(1)$. There are a maximum of $\mathcal{O}(L^2) = \mathcal{O}(N)$ buckets to go through. Thus the overall complexity of $\mathcal{O}(N\alpha(N))$ is preserved.

4.2.3   The BCS Union-Find decoder
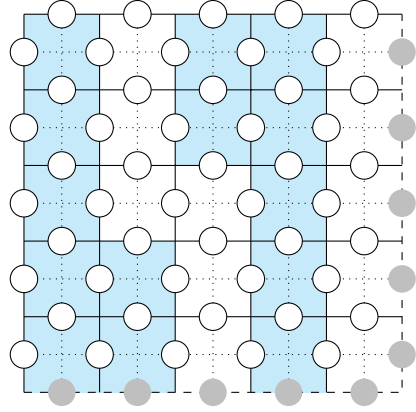
(a) Toric odd $L = 5$
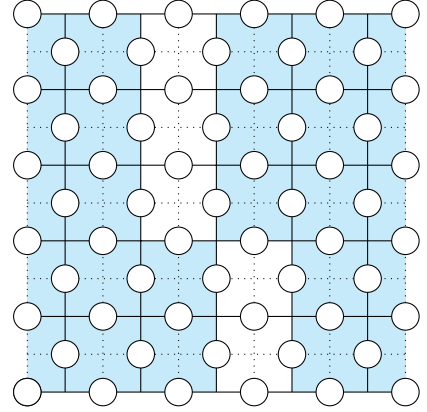
(b) Planar even $L = 6$

(c) Toric even $L = 6$

(d) Planar odd $L = 7$

Figure 4.2: The clusters C_μ with maximum cluster size S_μ that is allowed to grow is pictured for each case on the left. On the right, another cluster C_β is pictured that has a maximum size while still separated from C_μ.

4.3   Delayed Merge of boundary lists (DM)

When two clusters merge, one needs to check for the larger cluster between the two, and make the smaller cluster the child of the bigg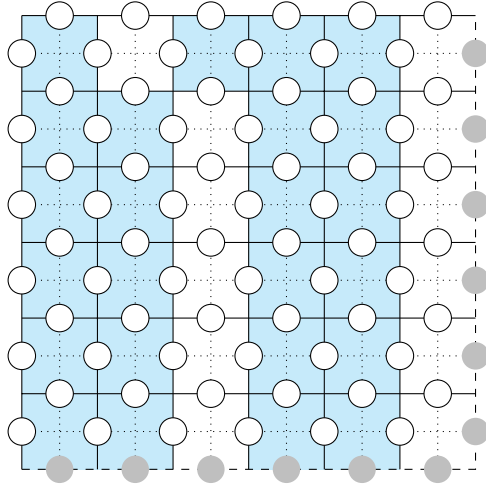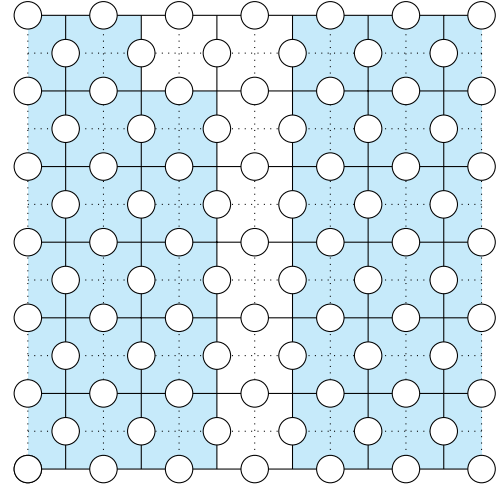er cluster, which lowers the depth of the tree and is called the *weighted union rule.* Applied to the toric lattice, the Union-Find decoder also needs to append the boundary list (which contains all the boundary edges of a cluster) of the smaller cluster onto the list of the larger cluster. This method, as explained before, requires that the new boundary list needs to be checked again.

In our application, instead of appending the entire boundary list, we just add a pointer stored at the parent cluster to the child cluster. As a parent can have many children, the pointers are appended to a list `children`. When growing a cluster, we first check if this cluster has any child clusters. If yes, these child clusters will be grown first by popping them from the list, but any new vertices will always be added to the parent cluster. Also during and after a merge, we make sure that any new vertices are always added to the parent cluster. Any child will exist in the list of a parent for one round of growth, after which its boundaries will be grown, and the child is absorbed into the parent. This method also works recursively by keeping track of the root cluster instead of just the parent cluster, and many levels of parent-child relationships can exists, but again, only for one round of growth.

TR: ● ●

A

B

boundary

a) Cluster A and B have some shared boundary.

TR: ● ●

newly grown

b) Cluster A grows its boundary.

TR: ● ●

c) Cluster A adds a new boundary.

children
TR: ●
    ●

union

d) Cluster B grows along arrow. Union(A,B).

TR: ●
    ●

e) New vertices are added to root cluster A.

TR: ●
    ●

f) New boundary is also added to root cluster A.
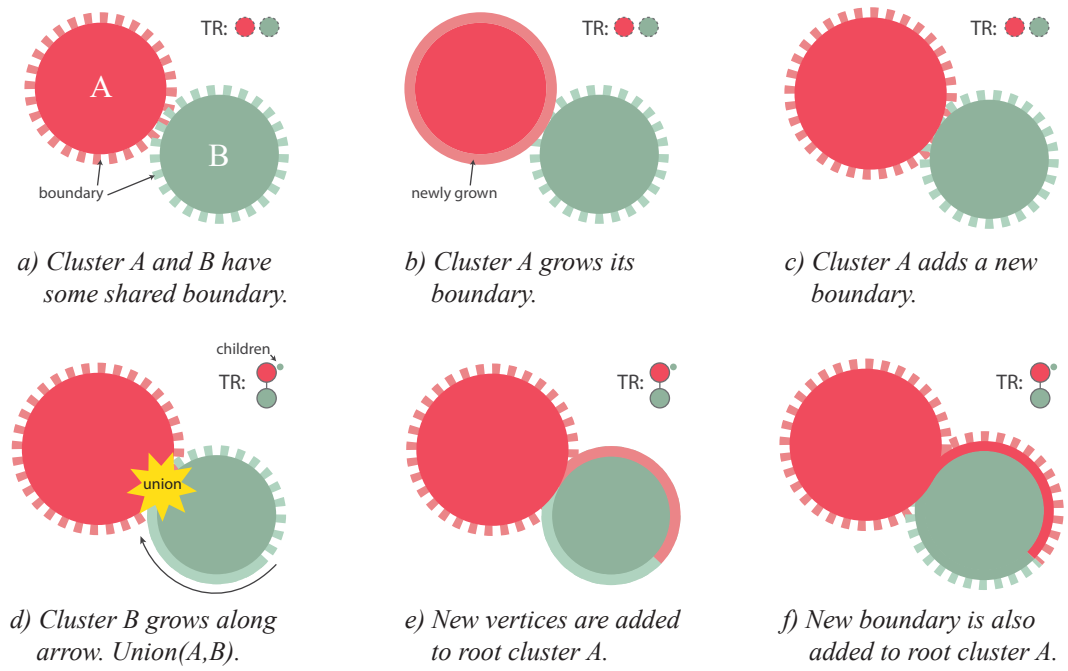
Figure 4.3: The parent-child method for merging boundary lists. By storing a list of pointers of child clusters at the parent cluster, we needn't append the full boundary list from the child to the parent cluster. The tree representation (TR) is shown on the top right.

36
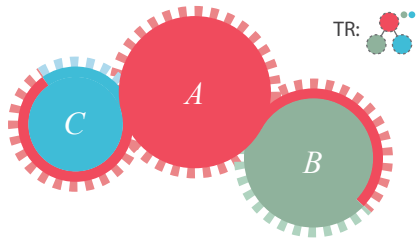
a) Cluster ABC with root cluster A is odd.
Boundary of ABC is stored at root A, but also
at children B and C.
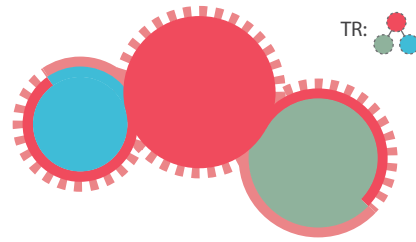
b) If cluster ABC grows, first the children are
popped from the list and grown. New vertices
are added to root A.
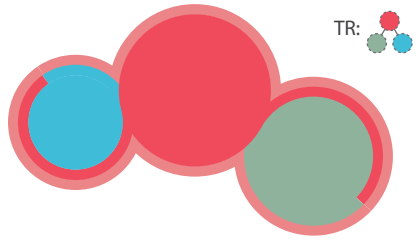
c) The boundary stored at root A is grow after.
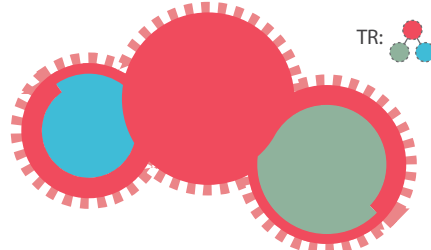
d) After growth, the children list is empty, and
new boundary is stored at root A.

Figure 4.4: Growing a merged boundary using the parent-child method. The tree representation (TR) is shown on the top right.

4.4 Growing Edge Priority based on path degeneracy (GEP)

4.4.1 Degeneracy on connecting edges between Clusters (GEP-C)

4.4.2 Degeneracy on Vertices with connecting edges (GEP-V)

4.5 Union-Find Balanced Bloom

In this section we describe a modification of the UF-decoder, dubbed the *Union-Find Balanced Bloom* (UFBB) decoder, that increases the threshold of the UF-decoder by improving its heuristic for minimum-weight matchings, while retaining a relatively low time-complexity. Within the vanilla UF-decoder, not all odd clusters are grown at the same time. Larger clusters relatively add more "incorrect edges" to themselves than compared to a smaller cluster (lemma 4.1). The UF decoder therefore applies *weighted growth* of clusters, where the order of cluster growth is sorted based on the cluster sizes. We have shown a linear time implementation in the Bucket Cluster Sort in section 4.2. With the addition of weighted growth, the error threshold of the UF decoder is increased from 9.2% to 9.9% for a 2D toric lattice [9]. This approaches but still lacks in terms of the 10.3% error threshold of the MWPM decoder.

The UF decoder is in fact a heuristic for minimum-weight matchings. A large cluster is generally the result of multiple rounds of growth of a smaller cluster. Each iteration of cluster growth buries the syndromes within that cluster with a layer of edges, of which only a small portion will be part of the matching, where each layer adds to the matching weight. With weighted growth, smaller clusters are grown first, such that this effect is less dominant. But the UF decoder is unsurprisingly less successful at minimum-weight than the MWPM decoder, which does this perfectly. The MWPM decoder considers all possible matchings by constructing a fully connected graph where the edges have the distance between syndrome as weights. The UF decoder does not look at the lattice in such a global way, but performs locally on each cluster. This should yield the same result conceptually, but in reality it does not due to a major weakness; In each round of growth, all boundary edges are grown simultaneously. The potential union of two clusters is reserved to one edge but may occur on many, is only handled after each round, where the order of the merging edges determines which edge is selected as the bridge. This leaves us with the question: Should all boundary edges of a cluster be grown simultaneously?

We suspect that the error threshold of the UF decoder can be increased by improving the heuristic for minimum-weight matchings. In this section, we will accomplish this by sorting the order of boundary edge growth within a cluster by calculation of their so-called *potential matching weight*, explained in 4.5.1. We will introduce a new data structure that we call the *node set* of a cluster in 4.5.2. Within this node set, we compute the node *parity* and *delay* in 4.5.3 and 4.5.4, which sets the order of boundary edge growth. In 4.5.5 through 4.5.7, we cover the rules for growth and join operations for the node sets, which are more complex than those of the UF algorithm. The modified decoder, the UFBB decoder, still has a relatively low worst-case quasilinaer time complexity, which is approximated in 4.5.9.
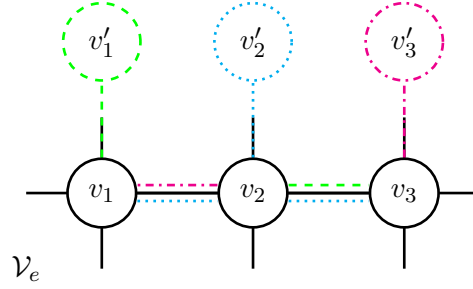
Figure 4.5: Unbalanced matching weight in cluster vertex set $\mathcal{V}$. The matching edges (dashed) correspond to the position of v'. If v' is connected to v_1 or v_3, the resulting matchings have a weight of 2. IF v' is connected to v_2, the resulting matching has a weight of 3.

4.5.1 Potential matching weight

To show that not all boundary edges within a cluster should not be grown simultaneously, we introduce the concept of *Potential Matching Weight* of a vertex. Let us first consider an example. Cluster C_e is defined by vertex set $\mathcal{V}_e = \{v_1, v_2, v_3\}$ (figure 4.5). The vertices lie on a horizontal line, distance 1 from each other, where each vertex has grown a single iteration of half-edges. Assume that each vertex in $\mathcal{V}_e$ is a syndrome, it has odd parity and is selected for growth. As UF decoder performs on the cluster locally, it has no knowledge about the state of its surrounding vertices until the cluster grows and merges with them.

Now let us investigate the weights of a matching if an additional vertex v' is connected to the cluster. If v' is connected to v_1 or to v_3, then the resulting matchings have a total weight of 2: (v', v_1) and (v_2, v_3), or (v', v_3) and (v_1, v_2). However if v' is connected to vertex v_2, then the total weight is 3: (v', v_2) and (v_1, v_3). This hypothetical weight after matching is the Potential Matching Weight (PMW) of a vertex.

Lemma 4.4 *The Potential Matching Weight (PMW) of a vertex v is the total length of matching edges within the cluster C_{grow} if the parity of the cluster C is even in an union between C_{grow} and C_{other}, where C_{other} is connected to C_{grow} on an edge touching v.*

From the above example, we can see that even for a minimal size odd cluster that is not a single vertex, the PMW is not equal for all vertices in the cluster. It would therefore not be "fair" to grow all boundary edges simultaneously. The growth of boundary edges connected to vertices with a high PMW should thus be delayed for some iterations, such that PMW's in the cluster reach the same value. However, if the PMW is to be calculated for every vertex with boundary edges in all clusters in every growth iteration, the time complexity of the algorithm would increase dramatically. Luckily, we can reduce these calculations to be performed on a set of *nodes* in each cluster, which we clarify in the next section.
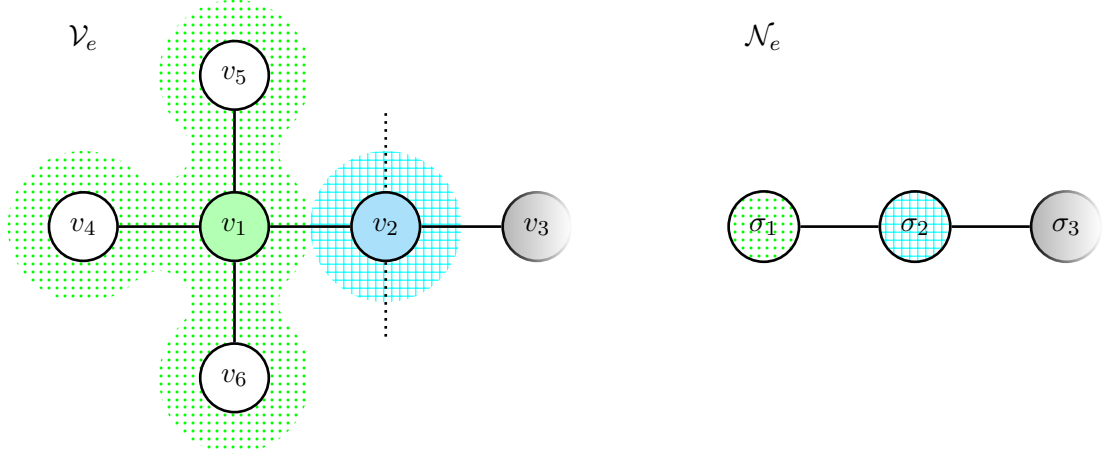
39

Figure 4.6: A node set $\mathcal{N}$ vs. a vertex set $\mathcal{V}$, both representing the same cluster. Each shaded area covers the vertices of a different node.

4.5.2 Node set representation of cluster

To efficiently calculate the PMW's in a cluster, we introduce here an additional data structure, the *node set* of a cluster. In the case of only Pauli errors, after syndrome identification, all identified clusters consist of a single vertex v_i which are non-trivial syndromes σ_i. This set of clusters is equivalent to the syndrome set $\mathcal{S}$. Within syndrome validation, these clusters are subjected to growth and merge events with other clusters. During growth, all vertices that are added to some cluster C have a closest syndrome σ within C that is in the syndrome set $\mathcal{S}$. The growth where vertices are added to a section of the cluster with a syndrome at the center can figuratively be compared with the *bloom* of a flower, where all vertices in this section are *seeded* in σ and are called the flower of σ.

Let us call these seeds the *nodes* n_i of the cluster. From our previous example, each vertex in C_e is a syndrome in $\mathcal{S}$, and is therefore a node. The node set is thus $\mathcal{N}_e = \{n_1, n_2, n_3\} = \{\sigma_1, \sigma_2, \sigma_3\}$ where $\sigma_1 \equiv v_1$, $\sigma_2 \equiv v_2$ and $\sigma_3 \equiv v_3$. The number of vertices in C_e increases in each round of growth by the bloom of the nodes in the cluster. However, the number of nodes remains the same at 3 nodes (figure 4.6). Moreover, as the bloom of a node adds vertices to the flower in all directions, all boundary vertices seeded in the same node are at equal distance to the syndrome at the center. Hence the PMW's for these boundary vertices are equal. The calculation of PMW's in the cluster thus does not require to traverse all the vertices, but just the nodes set of the cluster.

Lemma 4.5 *The calculation of PMW in the cluster can be limited to the nodes of a cluster, where all boundary vertices seeded in a node have the same PMW.*

Balanced Bloom

The node set $\mathcal{N} = \{n_1, n_2, ...n_{S_\mathcal{N}}\}$ is stored as a tree, an connected and acyclic graph, where the edges ϵ between the nodes are the branches in our figurative flower bush. Each node-edge ϵ can have arbitrary length and consists of one or more vertex-edges e. For any node set $\mathcal{N}$, we would prefer that the difference PMW for all nodes in the set to be minimal. The growth of a cluster with varying PMW values is thus selective in the nodes with the lowest PMW, and delays the bloom of nodes with larger PMW. As these nodes bloom and increase in size $n.s$, the number of growth iterations a node has bloomed, the cluster moves towards equal PMW. Here we use the *object.property* structure to indicate that a property variable is stored at the parent object. Once equal PMW in the cluster is reached, the growth of a node set is the *Balanced Bloom* of nodes.

Theorem 4.2 *Every vertex v that is added to a cluster is seeded in some node. All vertices with boundary edges that are seeded in the same node have the same PMW. Equal PMW in the cluster is reached by selectively blooming the nodes with the lowest PMW values, as each bloom increases the node size and its PMW.*

Furthermore, as long as the same nodes span the cluster, which is the case while no unions between clusters occur, we only need to calculate the PMW in the cluster once. The difference in PMW of a node with the minimal PMW in the cluster can be stored in memory at the node, and its bloom queued for some iterations based on its value.

Lemma 4.6 *Between union events, the PMW's of nodes in a clusters need only to be calculated once, where the bloom of a node is queued based on the difference of its PMW and the minimal PMW in the cluster.*

Junction-nodes

Syndrome-nodes σ are not the only type of nodes in the node set. Consider our example cluster C_e of 3 nodes n_1, n_2, n_3 again. Now we slightly alter this cluster to C'_e by increasing the distance between n_1, n_2 and n_2, n_3 to two edges. This means that cluster C'_e is only established after two growth iterations of the three previous separate cluster of nodes n_1, n_2, n_3, and has a total size of 13 vertices. Now consider the vertices v_{12} and v_{23} that lie between n_1, n_2 and n_2, n_3, respectively. These are *merging vertices* as they are added to the cluster during an union of two merging clusters. It is not clear in which nodes these vertices are seeded, as they lie in equal distance to two nodes. To solve this, a new type of node called *junction-nodes* j are initiated on the merging vertices, which lie on the junction of two flowers. All nodes j have the same characteristics of syndrome-nodes σ; they have their own flowers and can thus be separately delayed during growth.

Lemma 4.7 *On a merging vertex v that lie in equal distance to two syndrome-nodes from two separate clusters merging into one, a junction-node j is initiated in the joined node set $\mathcal{N}$. A junction-node has the same properties as a syndrome-node.*
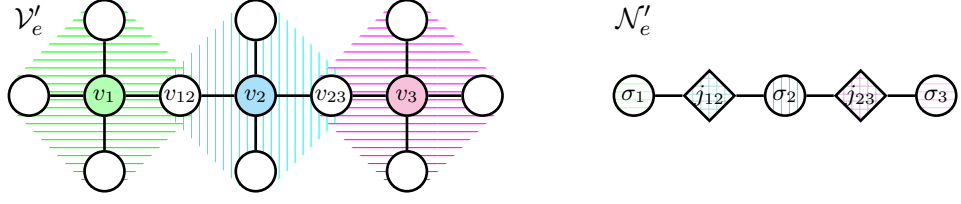
Figure 4.7: Merging vertices v_{12} and v_{23} are seeded in junction-nodes j_{12} and j_{23}, respectively, as they lie in equal distance to more than 1 syndrome-nodes.
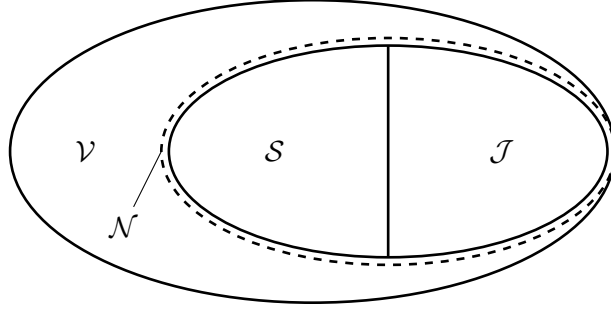

Figure 4.8: The space occupied by the sets of vertices $\mathcal{V}$ and nodes $\mathcal{N}$ (union of syndrome-node set $\mathcal{S}$ and junction-node set $\mathcal{J}$).

The union of the set of junction-nodes $\mathcal{J}$ and set of syndrome-nodes (syndromes) $\mathcal{S}$ is equal to the node set $\mathcal{N}$. A vertex can either be a node in the syndrome-node set, a node in the junction-node set, or not a node at all, but never both $\mathcal{S}$ and $\mathcal{J}$ as these sets are mutually exclusive. The node set size $S_{\mathcal{N}}$, is therefore upper-bounded by the cluster size or vertex set size $S_{\mathcal{V}}$, as all nodes are vertices, but not all vertices are nodes.

$$\mathcal{N} \subseteq \mathcal{V} \quad , \quad S_{\mathcal{N}} \leq S_{\mathcal{V}} \tag{4.3}$$
$$\mathcal{S} \;\cup\; \mathcal{J} = \mathcal{N}$$
$$\mathcal{S} \;\cap\; \mathcal{J} = \varnothing$$

To be able to bloom each node separately, we cannot store the boundary edges of a cluster in a single list $\mathcal{L}$ at the cluster. Instead, we store the boundary list for each node n_i separately in their own boundary lists $n_i.\mathcal{L}$. As we will see in the next section, the calculation of node-delays is dependant on the direction in which $\mathcal{N}$ is traversed. We store the node set by its root n_r at the cluster C.

Theorem 4.3 *The set of nodes $\mathcal{N} = \{n_1, n_2, ...n_{\mathcal{N}}\}$ of cluster C is a connected acyclic graph with root n_r, and exists next to the exists set of vertices $\mathcal{V}$. The function of $\mathcal{N}$ is to store the list of boundary edges at the nodes and to selectively bloom each node dependant on some calculated delay.*
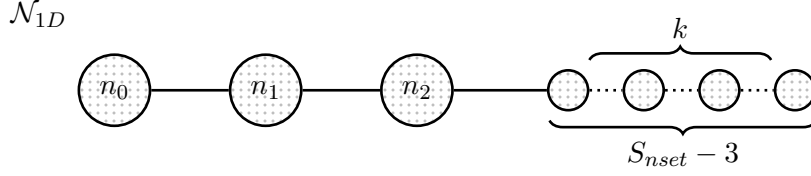
Figure 4.9: The 1D node tree $\mathcal{N}_{1D}$, consisting of only syndrome-nodes σ_i. The first 3 nodes n_0, n_1, n_3 are labelled, but the node tree continues after n_2 for $S_\mathcal{N} - 3$ nodes. The distance of the edges between n_3 and $n_{S_\mathcal{N}-1}$ is $(n_3, n_{S_\mathcal{N}-1}) = k$.

4.5.3   Node parity and delay

Even within the node set data structure of the cluster, the calculation of the PMW for each node is a heavy task. If done naively, the entire set needs to be traversed to find the PMW for every node in the set, and results to a quadratic complexity to the set size. Luckily, the node set data structure allows us to traverse the node set to compute the *differential delay* of a node to its parent, and the *root delay* after the set is fully traversed, which relates closely to the PMW. The delay computation complexity is linear to the set size.

1D node tree

To show how this calculation is performed, we first take the example of a 1D node tree $\mathcal{N}_{1D}$ of size $S_\mathcal{N}$ consisting of only syndrome-nodes σ_i, where all nodes lie on a horizontal line, but are allowed to grow in x and y directions (figure 4.9). In our example, we only look at the first 3 nodes n_0, n_1 and n_3. The node tree continues after n_2 for $S_\mathcal{N} - 3$ nodes.

Recall that the size of the node $n.s$ is equal to the iterations it has grown, one half-edge per iteration. This means that if a merge with some other cluster occurs on a boundary edge of n, the weight of the matchings edges within the flower of n is equal to $\lfloor n.s/2 \rfloor + 1$ or. For a merge on n_0, we also add edge (n_1, n_2) and some value k corresponding to the weight of matching edges in the remainder of the cluster. Let us calculate the PMW values for each of nodes n_0, n_1, n_2.

$$\begin{aligned} PMW(n_0) &= \lfloor n_0.s/2 \rfloor + 1 + (n_1, n_2) + k \\ PMW(n_1) &= \lfloor n_1.s/2 \rfloor + 1 + (n_0, n_1) + (n_1, n_2) + k \\ PMW(n_2) &= \lfloor n_2.s/2 \rfloor + 1 + (n_0, n_1) + k \end{aligned}$$

By taking the difference in PMW values of a node n_i and its parent n_{i-1}, we can compute the *differential delays* $\delta(n_i.d)$.

$$\begin{aligned} \delta(n_1.d) &= PMW(n_1.d) - PMW(n_0.d) \\ \delta(n_2.d) &= PMW(n_2.d) - PMW(n_1.d) \end{aligned}$$

The values of $n_i.d$ is thus dependant on the value of the root $n_0.d$, which can be any arbitrary value as the delay in the root is unknown when the calculation is initiated. For this reason,

we call $n_i.d$ the *root delays*, as it is the delay with respect to the root. To shorten, root delays are simply referred to as *delays*.

$$n_1.d \;=\; PMW(n_1) - PMW(n_0) + n_0.d = 2(\lfloor n_1.s/2 \rfloor - \lfloor n_0.s/2 \rfloor + (n_0, n_1)) + n_0.d$$
$$n_2.d \;=\; PMW(n_2) - PMW(n_1) + n_1.d = 2(\lfloor n_2.s/2 \rfloor - \lfloor n_1.s/2 \rfloor - (n_1, n_2)) + n_1.d$$

These delay values are not entirely correct, as $n_1.s = n_2.s = 2i$ yields the same value as $n_1.s = 2i$, $n_2.s = 2i + 1$. We introduce growth support of a node $n.g = n.s \bmod 2$ to accommodate for this degeneracy in PMW, and add this to the delay values.

$$n_1.d \;=\; 2\big(\lfloor (n_1.s + n_1.g)/2 \rfloor - \lfloor (n_0.s + n_1.g)/2 \rfloor + (n_0, n_1)\big) - (n_0.g + n_1.g) \bmod 2 + n_0.d$$
$$n_2.d \;=\; 2\big(\lfloor (n_2.s + n_2.g)/2 \rfloor - \lfloor (n_1.s + n_2.g)/2 \rfloor - (n_1, n_2)\big) - (n_2.g + n_1.g) \bmod 2 + n_1.d$$

If we were to consider some nodes $n_3, n_4...$ as well, we would find a trend in which the delay calculation is dependant on the *parity* of the node number i. The delay of odd node n_{2i+1} has the positive addition of (n_{2i}, n_{2i+1}) in its delay value, and the substraction of (n_{2i-1}, n_{2i}) for an even node n_{2i}. Thus we can generalize the delay calculation as the following:

$$n_i.d = n_{i-1}.d + 2\left(\left\lfloor \frac{(n_i.s + n_i.g)}{2} \right\rfloor - \left\lfloor \frac{(n_{i-1}.s + n_i.g)}{2} \right\rfloor + (-1)^{i+1}(n_{i-1}, n_i) \right)$$
$$- (n_i.g + n_{i-1}.g) \bmod 2 \quad | \quad n_0.d = 0. \quad (4.4)$$

Using equation 4.4, we can calculate all the relative delays in the 1D tree by traversing the node tree just once from left to right, where the initial delay is for example set to $n_0.d = 0$. The difference between the root delays and the minimal root delay value in the cluster relates to the PMW.

$$PMW(n_i) = n_i.d - \min\{n_0.d, ..., n_{S_{\mathcal{N}}-1}.d\} + K - n.w \qquad (4.5)$$

Here, the constant K is equal to the lowest PMW in the cluster. Recall from theorem 4.2 that the algorithm searches for the lowest PMW nodes in the cluster, thus the value of K is irrelevant for our algorithm. The variable $n.w$ stores the number of iterations a node has waited based on its calculated delay value, which is equivalent to the queue in lemma 4.6, and will be clarified in 4.5.5. If we store the minimal delay value in the cluster at the cluster object with

$$C.d = \min\{n_0.d, ..., n_{S_{\mathcal{N}}-1}.d\}, \qquad (4.6)$$

we can define a *Potential Normalized Weight* (PNW) that is normalized in K,

$$PNW(n_i) = n_i.d - C.d - n.w. \qquad (4.7)$$

Balanced bloom in a cluster is now achieved by blooming the nodes that has $PNW(n_i) = 0$. Additionally, we can define a normalized delay (ND) of a node that is equal to the actual number of iterations for a node to wait:

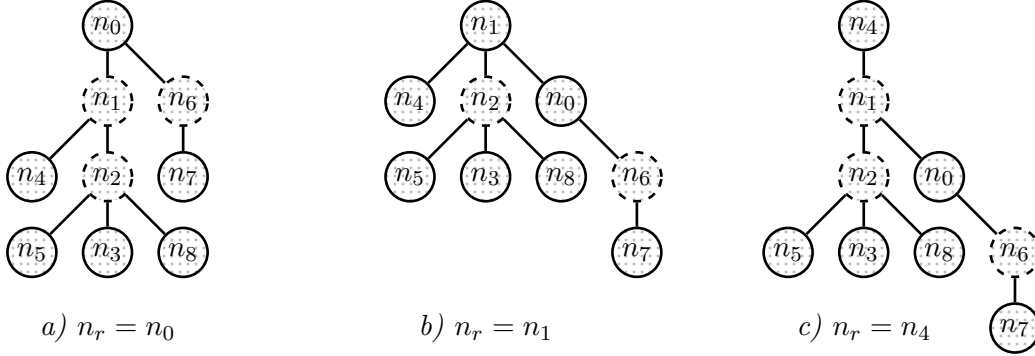$$ND(n_i) = n_i.d - C.d. \qquad (4.8)$$

Figure 4.10: The nodes in a node set can have even (solid) or odd (dashed) parities. The node parities are dependant on which node is set as root n_r. Here the same node tree $\mathcal{N}$ is illustrated with different roots.

Node tree parity

The 1D node tree does not accurately represent node trees that occupy a real lattice. On a 2D (Pauli errors) and 3D lattice (Pauli and measurement errors), the node tree is allowed to form in the same dimensions. As $\mathcal{N}$ is an acyclic graph, the 3D variant can be considered equal to the 2D variant. The difference is that now the *ancestry* in the tree, the set of parent-child relations, is not determined by some node number i, and each node can have more than 2 connections.

The delay calculation is done comparatively with the previous node, which means that there must be some directed path within $\mathcal{N}$, such that there is a clear direction to traverse the tree for the delay calculation. We can start the calculation from the root n_r. The node parity, previously determined by the node number i, is now set by the number of children nodes modulo 2. To calculate this parity for each node without traversing all children nodes, we can use the following function

$$
n_\beta.p = \begin{cases} 0, & \text{if } n_\beta \text{ has no children} \\ \left(\sum_j 1 - n_{\gamma,j}.p \right) \bmod 2 \mid \forall n_\gamma \text{ child of } n_\beta, & \text{otherwise,} \end{cases} \tag{4.9}
$$

where n_β is the node of interest, and each of the nodes $n_{\gamma,j}$ is a child of n_β. As this requires the parity of each child node to be known, the node parities of the entire set can be calculated by a depth-first search (DFS) of the node tree, and traversing back to the root recursively and applying the above equation.

Since $\mathcal{N}$ is acyclic, any node in the set can be set as the root n_r of the set, and the calculation of the parity would still be valid, although not identical. The node set $\mathcal{N}$ is therefore a *semi*-directed tree, in which the edges are undirected, but an ancestry is set by the root node n_r (see figure 4.10). If the root node changes to n_r', the ancestry within the tree changes, and the node parities within the set become unknown, or *undefined*, requiring a new calculation of a reversed DFS from n_r'.

Lemma 4.8 *Any node $n_i \in \mathcal{N}_\alpha$ is a valid root.*

Lemma 4.9 *The node parity $n_i.p$ is the number of children nodes of node n_i modulo 2, and can be calculated via a reversed DFS from root n_r. If a new node is set as root n'_r, the ancestry in a set changes, and node parities and delays in the set become undefined.*

Node tree delay

The delay equation 4.4 can be altered by replacing the node number i with some parent-child relationship between nodes, similarly to the parity calculation. To calculate the node delays within $\mathcal{N}$, we need to traverse $\mathcal{N}$ in a second DFS from root n_r with

$$
n_\beta.d = n_\alpha.d + 2\left(\left\lfloor \frac{(n_\beta.s + n_\beta.g)}{2} \right\rfloor - \left\lfloor \frac{(n_\alpha.s + n_\beta.g)}{2} \right\rfloor + (-1)^{n_\beta.p-1+1}(n_\alpha, n_\beta) \right)
$$
$$
- (n_\beta.g + n_\alpha.g) \bmod 2 \quad | \quad n_r.d = 0, \ n_\beta \text{ child of } n_\alpha, \quad (4.10)
$$

where n_β is the node of interest and n_α is an ancestor of n_β, and the sign of the edge component is now dependant on the node parity $n.p$. As the node parities are only defined while the same node is root per lemma 4.9, the delay calculation is only valid if the DFS is performed from the same root n_r as in the parity calculation.

Lemma 4.10 *The calculation of node delays is only valid while node parities within the set are defined along the same ancestry as the node delay calculation.*

An interesting aspect of the node delays is that the differential delays $\delta(n.d)$ are indifferent for which node is set as root $n_r = n$. The root delay value $n.d$ however may differ for different roots as de delay value for the root node is arbitrary. But as we subtract by the minimal delay $C.d$ to find the normalized delay, the root dependance of node PMW and node PNW is accounted for. This fact strengthens lemma 4.8.

Junction node parity and delay

Up until now, the existence of junction-nodes has been neglected in the node parity and delays calculations. But as junction-nodes have the same properties as syndrome-nodes, junction-nodes can be delayed as well, and they must be included in the parity and delay calculations. Furthermore, without junction-nodes, lemma 4.8 cannot be true for a node set $\mathcal{N}$ for all nodes $n \in \mathcal{N}$ for the same set of edges $\{\epsilon\}$. As a junction-node is also allowed to bloom, similarly to a syndrome-node, equation 4.10 still holds for junction-nodes. However, the parity of a junction-node is calculated differently. Consider an example node set $\mathcal{N}_e$ with 5 syndrome-nodes $\{\sigma_1, ..., \sigma_5\}$ lined up linearly with distance 1 between them and $n_r = \sigma_1$. Let us drop the $n.s, n.g$ components of the delay in equation 4.10 as we are now only interested in the parity component $(-1)^{n_\beta.p-1+1}(n_\alpha, n_\beta)$. The parity of σ_4 is odd, therefore

$$
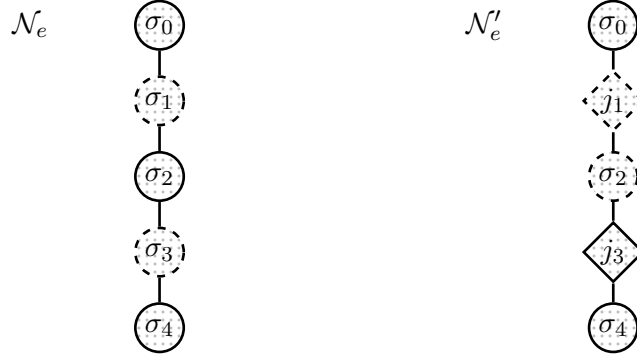\sigma_4.d = \sigma_3.d + 2(\sigma_3, \sigma_4).
$$

Figure 4.11: *Two example node sets $\mathcal{N}_e$ and $\mathcal{N}'_e$ each containing 5 nodes, where syndrome-nodes σ_i are circles and junction-nodes j_i are diamonds. Their appropriate parities are calculated where an even parity correspond to continuous lines and odd to dashed lines.*

Consider now a second example node set $\mathcal{N}'_e$ with 3 syndrome-nodes and 2 junction-nodes $\{\sigma_1, j_2, \sigma_3, j_4, \sigma_5\}$. The PMW's for σ_3 and j_4 are $(\sigma_1, j_2) + (j_4, \sigma_5)$ and $(\sigma_1, j_2) + (\sigma_3, j_4) + (j_4, \sigma_5)$, respectively, where the delay in j_4 is now

$$j_4.d = \sigma_3.d - 2(\sigma_3, j_4).$$

The edge component of the delay calculation now has an opposite sign. This flip in sign is due to a flip in node parity for junction-nodes compared to syndrome-nodes. As a result, we can generalize the parity calculation of equation 4.9 for node sets including both syndrome-nodes and junction-nodes.

$$n_\beta.p = \begin{cases} 0, & \text{if } n_\beta \text{ has no children} \\ \left(\sum_j 1 - n_{\gamma,j}.p\right) \bmod 2 \mid \forall n_\gamma \text{ child of } n_\beta, & n_\beta \equiv \sigma_\beta \\ 1 - \left(\sum_j 1 - n_{\gamma,j}.p\right) \bmod 2 \mid \forall n_\gamma \text{ child of } n_\beta, & n_\beta \equiv j_\beta \end{cases} \quad (4.11)$$

To put this into perspective of lemma 4.9, the parity of a syndrome-node is the number of children *syndrome*-nodes. The parity of a junction-node is 1 minus the number of children syndrome-nodes. From here, our definitions of parity and delay calculation stay unchanged; the parities can to be calculated by a reversed DFS of the node tree from the root with equation 4.11, and the delays by a second DFS with equation 4.10.

Lemma 4.11 *The node parity in a syndrome-node $\sigma.p$ is the number of children syndrome-nodes σ_γ modulo 2. The node parity in junction-node $j.p$ is 1 minus the above definition.*

Parity and delay calculations

With equation 4.11 and 4.13, we now finally have the tools to formulate the algorithms to calculate the node parities and delays. For a node set with root n_r, we can calculate the parities by calling the *head recursive* function `CalcParity` on n_r in algorithm 3, where we do

a reverse DFS of the node tree. The node delays are calculated by calling the *tail recursive* function `CalcDelay` in algorithm 4, where we perform a second DFS of the node tree. These Parity and Delay Calculation will from this point be referred to as the PDC. Note that in the `CalcDelay` algorithm, we have replaced the delay equation 4.10 with equation 4.13, which we will cover in the next section.

Algorithm 3: CalcParity

    Data: node
    Result: Defined parities for all children of node

  1 parity = Sum([$1-$ CalcParity(child) $\forall$ child *of* node]) %2
  2 **if** node $\equiv \sigma$ **then**
  3 | node.p = parity
  4 **elif** node $\equiv j$ **then**
  5 | node.p = $1-$ parity
  6 **return** node.p

Algorithm 4: CalcDelay

    Data: node, cluster
    Result: Defined parities for all children of node

  1 **if** node *has an ancestor* **then**
  2   calculate node.d with equation 4.13
  3   **if** node.d $<$ cluster.d **then**
  4 | cluster.d = node.d

  5 **for** child *of* node:
  6   CalcDelay(child, cluster)

Theorem 4.4 *To prepare a cluster with node set $\mathcal{N}$ and node root n_r with undefined node parities and delays, we calculate node parities in $\mathcal{N}$ by calling the head recursive function* **CalcParity**(n_r), *and sequentially calculate node delays in $\mathcal{N}$ by calling the tail recursive function* **CalcDelay**(n_r).

4.5.4 BB-state optimization

In this section, we alter the delay equation 4.10 with an extra parameter f_{bloom} to optimize a trade-off in the Balanced Bloom algorithm. In the PDC, the appropriate delays in nodes are calculated such that, if the bloom in these nodes are delayed a number of iteration equal to the calculated delays, the PNW for every node in the set is zero. We will see how to grow a node set and how to join node sets in the following sections. But before we move on, there is one final problem with the delay calculation.

If some odd number of nodes $\mathcal{N}^o$ is attached to n^e of $\mathcal{N}^e$ during an union of two node sets, node parities for nodes in subset $'\mathcal{N}^e = \{n_i \in \mathcal{N}^e | n_i$ ancestor of $n^e\}$ are flipped, where odd nodes become even and even become odd, which is called *parity inversion*. Per lemma
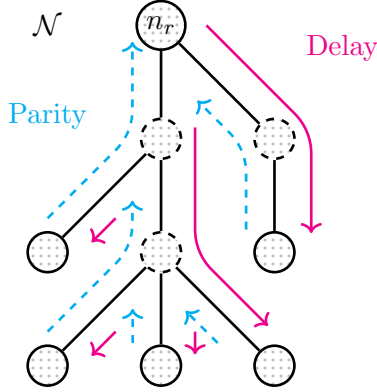
Figure 4.12: The parity and delay calculation (PDC) performed on a the root node n_r, which is equivalent to two depth-first searches on $\mathcal{N}$ to compute node parities (head recursively) and delays (tail recursively).

4.9 and 4.10, the delays in $'\mathcal{N}^e$ are now undefined and need to be recalculated. Before the union, the cluster of $'\mathcal{N}^e$ is grown according to Balanced Bloom, where the odd nodes are delayed and consequently the even nodes will have some node sizes larger than the odd node sizes $n_{even}^e.s > n_{odd}^e.s$. After the union, the parities for nodes in $'\mathcal{N}^e$ flip, and the pre-union even nodes are now odd and have some positive delay. As $n_{even}^e.s > n_{odd}^e.s$, the absolute delays (equation 4.8) of these nodes are larger than the absolute delays of the pre-union odd nodes per equation 4.10. Subsequent parity inversions further increases the absolute delays in the post-union odd nodes.

To better illustrate this problem, we introduce the concept of the *BB-state* $(M : I)$ of a cluster, where M is equal to the maximum absolute delay in the cluster, and $I \leq M$ is the number of iterations grown while zero PNW in the cluster has not been reached. For example, a cluster with $M = 4$ requires 4 growth iterations to reach zero PNW in all nodes in the cluster. The BB-state thus gives us an indication of the how near Balanced Bloom a cluster performs. Let the fraction $f = I/M$ be the *balance factor*, then $f \to 1$ is equivalent to a larger degree of balanced PNW's and thus improves the heuristic for minimum-weight.

Lemma 4.12 *The BB-state $(M : I)$ describes the degree of difference in PMW in the cluster, where the balance factor $f = I/M = 1$ is equivalent to equal PMW.*

In the context of the BB-state, the delayed bloom of odd nodes in cluster growth is equivalent to increasing the value of I. Once $f = 1$ is reached after $I = B$ iterations, all nodes have zero PNW and none are delayed during further growth. As $f \to 1$ increases the difference between even and odd node sizes, the balance factor at the moment of union f_u is proportional to the increase in the value of M after union or the fraction of the after-union M_{AC} and before-union M_{BC}:
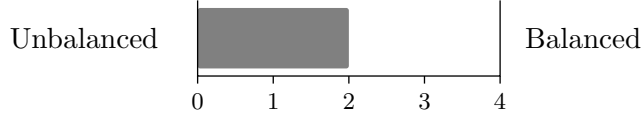
$$f_u \propto \frac{M_{AU}}{M_{BU}} \geq 1. \tag{4.12}$$

Figure 4.13: Visual representation of the BB-state (4 : 2). The size of the full x-axis is M = 4 and the length of the bar is I = 2. The left side of the spectrum is equivalent to the unbalanced BB-state, and the right the balanced state.

Subsequent parity inversions cause a gradual but certain increase in M of the BB-state, depending on f_u during the parity inversion at the union, requiring a growing number of growth iterations I to reach the balanced BB-state $(M : M)$. Furthermore, as the lattice size is increased, the total number of unions of a cluster with other clusters also increases, leading to a growing number of parity inversions. Thus increasing the lattice size has the consequence that more growth iterations I are needed to reach a balanced BB-state.

This is a trade-off in the effectiveness of Balanced Bloom. On the one hand, it is preferred that $f \to 1$ to reach a balanced BB-state, but on the other, f is also proportional to the number of iterations needed to actually reach $f = 1$ in the case of parity inversions. To this end, we introduce the balance constant f_{bloom} to equation 4.10.

$$n_\beta.d = n_\alpha.d + \left\lceil f_{bloom}\left(2\left(\left\lfloor \frac{(n_\beta.s + n_\beta.g)}{2} \right\rfloor - \left\lfloor \frac{(n_\alpha.s + n_\beta.g)}{2} \right\rfloor + (-1)^{n_\beta.p-1+1}(n_\alpha, n_\beta)\right)\right)\right.$$
$$\left. - (n_\beta.g + n_\alpha.g) \bmod 2 \right\rceil \quad | \quad n_r.d = 0, \ n_\beta \text{ child of } n_\alpha, \ f_{bloom} \in [0,1] \quad (4.13)$$

Lemma 4.13 *The balance factor f is the cause of a trade-off of Balanced Bloom, where $f \to 1$ during growth improves the heuristic for minimum-weight, but also increases the number of growth iterations needed to reach $f = 1$.*

For any $f_{bloom} < 1$, a cluster will never actually reach the balanced BB-state, but only $(M : f_{bloom}I)$. Consequently, in the case of parity inversion, the difference in node sizes between even and odd nodes is decreased, such $f \to f_{bloom}$ can be reached in a lower number of growth iterations. Alternatively, the BB-state can be redefined as $(f_{bloom}M : I)$, where $f = 1$ can now be reached in fewer growth iterations if $f_{bloom} < 1$. From intuition the balance constant should be around $f_{bloom} = 1/2$, which we will use in this thesis. For this value, the absolute delays in a node set are halved, such that both before and after the parity inversion, the lower halve of the BB-state spectrum of delays is occupied.

Theorem 4.5 *The degree of delay $f_{bloom} \in [0,1]$ determines the factor of the calculated delays between the even and odd nodes. For any $f_{bloom} < 1$, a cluster will never reach zero PNW in the cluster, but only $f = I/M = f_{bloom}$, which also decreases the number of growth iterations needed to reach $f = f_{bloom}$ again after parity inversion.*
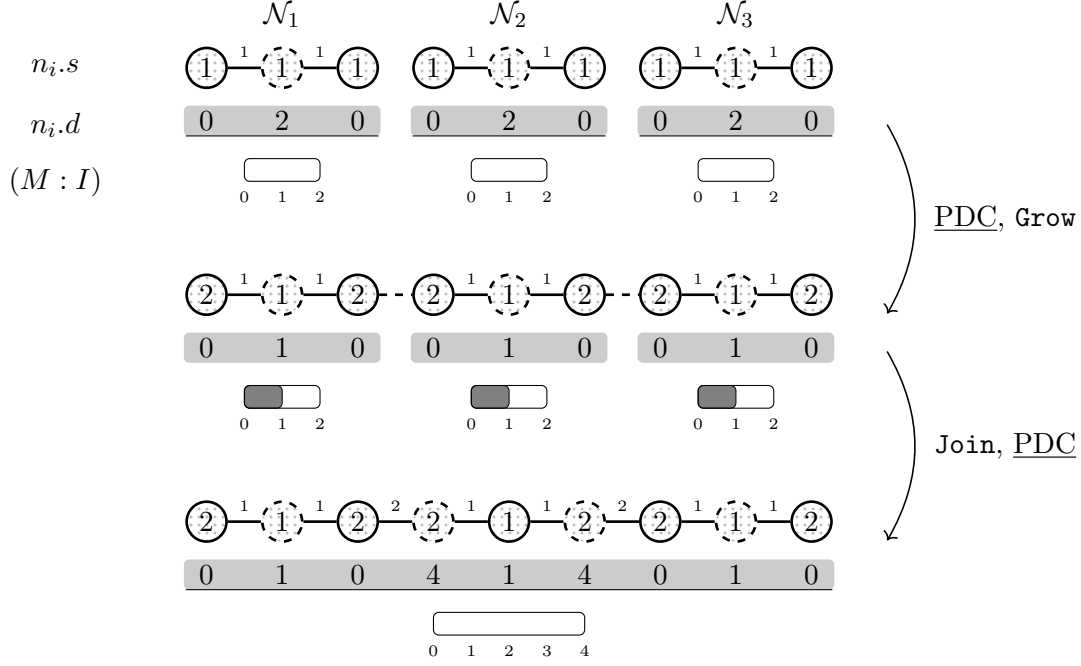
Figure 4.14: The delay values $n_i.d$ and the BB-states $(M : I)$ for 3 odd clusters $\{\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3\}$ of 3 nodes that grow and join into a size-9 cluster. Initially (top), PDC's are performed with delay equation 4.10 on each odd cluster which have BB-states $(2 : 0)$, with delay 2 in the middle node. The clusters are grown (middle), where the middle node is delayed, such that it's delay value decreases to 1, and the clusters have B-states $(2 : 1)$. The clusters join (bottom) to a single odd cluster, which is selected for growth. Hence PDC is performed again, and the BB-state is $(4 : 0)$, thus requiring 4 growth iterations before zero PNW is reached in all nodes.

Figure 4.15: The same clusters, growths and joins as in figure 4.14, but now with delay equation 4.13 for $f_{bloom} = 1/2$. With the redefined BB-state $(f_{bloom}M : I)$, $f = 1$ can be reached in fewer growth iterations; e.g. after 1 round (middle), $(1 : 1)$ is reached. Also after join to a single cluster (bottom), fewer iterations are needed (2 compared to 4 in figure 4.14).

The optimal value of f_{bloom} is dependant on the number of parity inversions, and consequently the lattice size , bucket number i,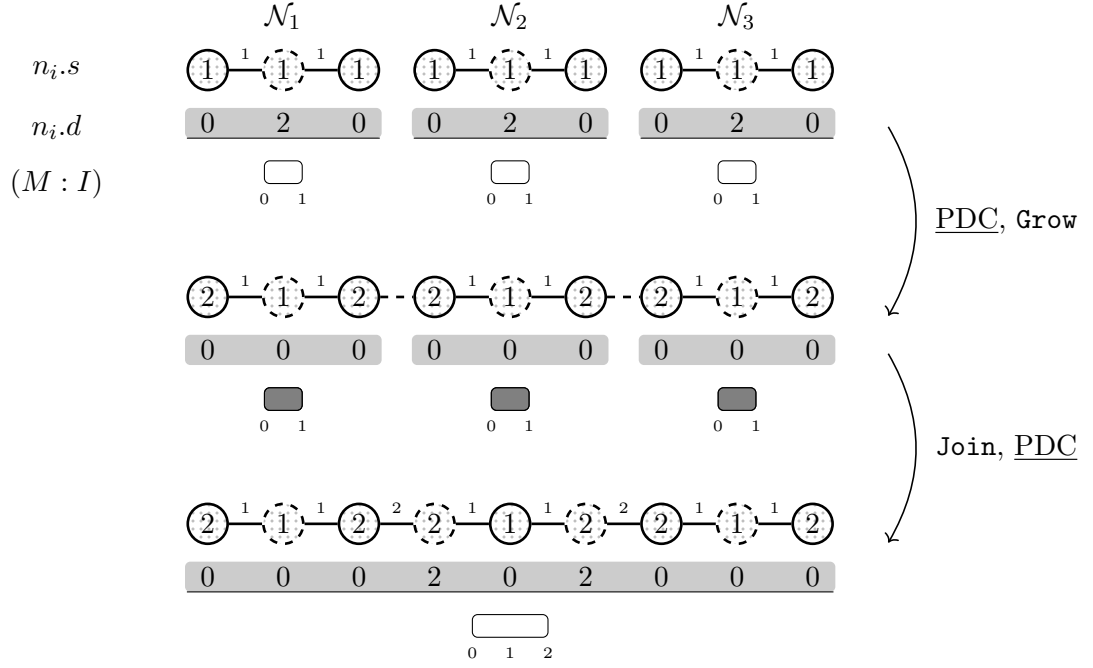 and the cluster sizes $S_{\mathcal{N}}, S_{\mathcal{V}}$, such that $f \to 1$ after the last parity inversion of a cluster is maximally occupied. We do not provide an analytical approach to estimate the optimal value of $f_{bloom}(L, i, S_{\mathcal{N}}, S_{\mathcal{V}})$ here, but note that other values than $f_{bloom} = 1/2$ should be explored.

4.5.5 Growing a cluster

With the knowledge of previous section, we now have the equations and algorithms available to describe the steps to grow a cluster in the context of Balanced Bloom. In the node data structure, the growth of a cluster is equivalent to a DFS of the node set. The boundary list for each cluster is not stored at C, but separately stored at each of the nodes n_i in $\mathcal{N}$. We traverse all $n_i \in \mathcal{N}$ from the root n_r and apply `Bloom`(n_i), which increases the support of all boundary edges in $n_i.mL$ by 1.

Recall from theorem 4.2 that with Balanced Bloom, we need to conditionally bloom the nodes with minimal PMW, or zero PNW. Also, from lemma 4.6, the delays are not recalculated after each growth iteration (in the absence of unions) but stored in memory at the nodes, and the PNW is updated via $n.w$ (equation 4.7), the number of growths a node has waited. Thus, we can define a function `Grow`(n_i) where `Bloom`(n_i) is only applied if $PNW(n_i) = n_i.d - C.d - n_i.w = 0$ is satisfied. If not, node n_i is skipped, the wait is increased $n_i.w = n_i.w + 1$ and `Bloom` is recursively applied on its children.

New vertices v_{new} grown from node n_i are added to $\mathcal{V}$, while storing the seed node at each new vertex $v_{new}.n = n_i$. New boundary edges are appended to the boundary list $n_i.\mathcal{L}$ stored each seed node n_i. The number of nodes in $\mathcal{N}$ and the shape of the flower bush tree therefore does not change while no merge between clusters has happened.

Theorem 4.6 *A cluster C is grown by calling **Grow**(n) on the root node n_r, which first checks for the wait of the current node $n.d - C.d - n.w = 0$ to grow its boundary edges with **Bloom**(n), and then recursively applies **Grow** to its children.*

Algorithm 5: Grow

   Data: node
   Result: A node that has either grown or waited one iteration.

1 **if** node.d $-$ cluster.d $-$ node.w $= 0$ **then**
2   Bloom(node), add all edges edge.support $= 2$ to $\mathcal{F}$
3 **else:**
4   node.w $+ = 1$
5 **for** child *of* node:
6   Grow(child)

4.5.6 Joining node sets

With the extension of the node set data structure, during a union of clusters C_α and C_β, we have to additionally combine the node sets $\mathcal{N}_\alpha$ and $\mathcal{N}_\beta$ that requires its own set of rules that we will explain in this section. Let us first make a clear distinction between the various routines. On the vertex set $\mathcal{V}$ we apply $\texttt{Union}(v^\alpha, v^\beta)$, on the two vertices spanning the edge connecting two clusters. On node set $\mathcal{N}$, we introduce here $\texttt{Join}(n^\alpha, n^\beta)$, which is called on the two nodes n^α, n^β that seed vertices v^α, v^β, respectively. During a merge of two clusters, these routines are both applied on their respective sets. From this point, when either one of the expressions "merge clusters C^α and C^{β}", "the union of vertex sets $\mathcal{V}_\alpha$ and $\mathcal{V}_\beta$" or the "join of node sets $\mathcal{N}_\alpha$ and $\mathcal{N}_\beta$" is mentioned, it is always implied that both routines are executed.

Within the vertex set $\mathcal{V}$, we apply *path compression* and *weighted union* to minimize the depth of the tree and therefore minimizing the calls to the $\texttt{Find}$ function. Similarly, in the node set $\mathcal{N}$, we would also like to apply a set of rules to minimize the calls to $\texttt{CalcParity}$ and $\texttt{CalcDelay}$, which are the parity and delay calculation(s) (PDC). As the structure of the tree is crucial in computing the parity of the nodes and relative delays between the nodes, these rules will be quite different than in vertex set $\mathcal{V}$ that changes the ancestry dynamically by path compression. Join rules will be dependant on the parities of the joining node sets $\mathcal{N}.p$, which is the number of syndrome-nodes in the set modulo 2. The parity of a node set $\mathcal{N}.p$ is equivalent to the parity of a cluster $C.p$, which also refer to the number of syndromes in the cluster.

Lemma 4.14 *The parity of node set $\mathcal{N}.p$ is the number of syndrome-nodes $a_i \in \mathcal{N}$ modulo 2. The parity of node set $\mathcal{N}.p$ is analogous to cluster parity $C.p$.*

Only odd clusters with odd parity node sets are grown in the UF-decoder. It may thus be tempting to conclude that a join must include at least one odd node set. This is however not true as within the same growth iteration, there may be many joins, where some odd cluster $\mathcal{N}_1^o$ first joins with odd cluster $\mathcal{N}_2^o$, but also joins with even cluster $\mathcal{N}_3^e$. The second join is effectively between even clusters. Hence there are 3 types of joins: 1) odd-odd, 2) even-odd and 3) even-even, where even-odd is equivalent to odd-even. These joins can be put into 2 *classes*, dependant on the parity of the resulting cluster. Both odd-odd and even-even joins to an even cluster and thus belongs to the even class join (E-join), whereas even-odd (and odd-even) joins to an odd cluster in the odd class join (O-join).

E-joins

For E-joins, the joined even cluster $\mathcal{N}^e$ will not be selected for growth by the UF-decoder. One could naively conclude that no PDC will be performed and no PDC minimization can be made. This is of course not true as it is entirely possible that another cluster grows, and merges with the cluster of $\mathcal{N}^e$ in a O-join. In that case, we might think about "reusing" some of the node parities and delays that were already calculated in the subsets of $\mathcal{N}^e$, such

that $\mathcal{N}^e$ doesn't have to be traversed entirely for its parities and delays. In this case, a *partial PDC* is performed over the undefined part.

To reuse prior calculated parities and delays, we need to traverse $\mathcal{N}^e$ to find which sections are still valid, and which sections are not. This is no trivial task and often requires us to traverse the entire set $\mathcal{N}^e$, especially when the clusters in the E-join are the results of joins within the same growth iteration. Checking the validity to reuse prior parities and delays then acquires the same complexity as redoing the PDC over the subset $\mathcal{N}^e$. Hence the node parities and delays in the joined set after an E-join are *undefined*.

Lemma 4.15 *Node parities and delays become undefined if multiple node sets joins into a new set $\mathcal{N}$ with even parity.*

O-joins

Consider now an O-join between an even node set $\mathcal{N}^e$ and an odd node set $\mathcal{N}^o$ in nodes n^e, n^o respectively, and assume that this join is due to the growth of odd cluster $\mathcal{N}^o$ onto an "idle" $\mathcal{N}^e$. The join of these two sets produces a new odd node set $\mathcal{N}^o_{new}$ with subsets $'\mathcal{N}^e$ and $'\mathcal{N}^o$, referring to the original node sets. We are provided with two choices, A) make n^e child of n^o, or B) make n^o child of n^e. The ancestry in the parent node set stays unchanged, but the ancestry in the child subset is changed by setting the joining node in the child set n^c as the sub-root of the child subset $'\mathcal{N}^c$. This is allowed per lemma 4.8, but removes any calculated parities or delays per lemma 4.9 and 4.10.

For option A, an even number of nodes of $'\mathcal{N}^e$ is attached to n^o, and the ancestry in $'\mathcal{N}^o$ hasn't changed. The parities and delays in $'\mathcal{N}^o$ stay valid and can be reused. From n^e, which is now the sub-root of $'\mathcal{N}^e$, a partial PDC is applied, where the relative delay of n^e is calculated with respect to its parent n^o (figure 4.16A). This is efficient as the parities and delays in $'\mathcal{N}^e$ are already undefined per lemma 4.15. For option B, we need to redo the PDC in both $'\mathcal{N}^o$ and $'\mathcal{N}^e$ (figure 4.16B), as $'\mathcal{N}^o$ has a changed ancestry and $'\mathcal{N}^e$ is even. The PDC is thus minimized if option A is always chosen.

The rules are thus very simple for the function $\texttt{Join}(n^\alpha, n^\beta)$. For O-joins between an even and an odd node set $\mathcal{N}^e, \mathcal{N}^o$ in the nodes n^e, n^o, always make the even node set a child of the even node set, where n^e is now the sub-root of the subset $'\mathcal{N}^e$. For E-joins between two even or two odd node sets, the parent and child sets can be picked at random. Finally, we note the concept of a *partial* PDC is rather redundant, as a PDC is always applied partially on the undefined part of a set for any set with $S_\mathcal{N} > 1$.

Theorem 4.7 *The union of node sets $\mathcal{N}^\alpha, \mathcal{N}^\beta$ on nodes n^α, n^β respectively is performed with $\texttt{Join}(n^\alpha, n^\beta)$. If the join is between an even and an odd node set $\mathcal{N}^e, \mathcal{N}^o$ in the nodes n^e, n^o, $\texttt{Join}(n^e, n^o)$ makes the node of the even set n^e a child of the node of the odd set n^o. If the join is between two even or two odd node sets, the choice is arbitrary.*
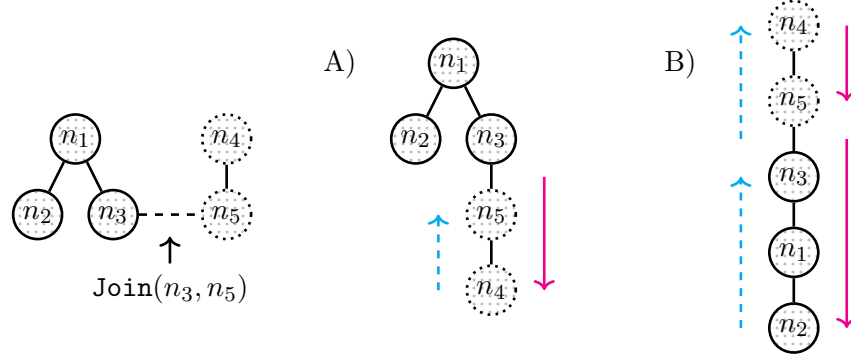
Figure 4.16: An odd cluster $\mathcal{N}^e = \{n_1, n_2, n_3\}$ with root $n_r^e = n_1$ joins with an odd cluster $\mathcal{N}^o = \{n_4, n_5\}$ with root $n_r^o = n^4$ on nodes n_3, n_5, respectively, to a new set $\mathcal{N}$ with subsets $'\mathcal{N}^e$ and $'\mathcal{N}^o$. Here we use dotted outlines on the nodes of $'\mathcal{N}^e$ to indicate that their parities and delays are undefined. If we choose to A), make n_5 a child of n_3, the parities and delays in $'\mathcal{N}^o$ can be reused, and we only have to redo PDC over $'\mathcal{N}^e$. If we choose to B), make n_3 a child of n_5, PDC's have to be redone over both $'\mathcal{N}^o$, as it has a new sub-root n_3, as well as $'\mathcal{N}^e$ as its parties and delays were undefined.

4.5.7 Prevention of redundant PDC's

The final rule for joins between clusters introduces a data structure to store undefined parts of a cluster, such that multiple PDC's over a subset is prevented. If there are many O-joins (and E-joins) within the same growth iteration i, that at the end of i results to one single cluster $\mathcal{N}$, every O-join will require the PDC over the even subset. There may be subsets where multiple PDC's are performed before the final cluster $\mathcal{N}$ is constructed. If the PDC is done directly after an O-join, the calculated values may thus be redundant.

Consider an example with 5 odd clusters $\mathcal{N}_1, ..., \mathcal{N}_5$ (figure 4.17). The join of $\mathcal{N}_1$ and $\mathcal{N}_2$ to $\mathcal{N}_{12}$ is an E-join and requires no PDC. The join of $\mathcal{N}_{12}$ and $\mathcal{N}_3$ is an O-join, and we apply PDC in $\mathcal{N}_{12}$. The join of $\mathcal{N}_{123}$ and $\mathcal{N}_4$ is an E-join and the join of $\mathcal{N}_{1234}$ and $\mathcal{N}_5$ is an O-join, with PDC executed in $\mathcal{N}_{1234}$. The earlier computation in $\mathcal{N}_{12}$ was therefore unnecessary and possibly invalid.

Note that some odd node set $\mathcal{N}^o$ must always consist of some odd part $'\mathcal{N}^o$ and an even part $'\mathcal{N}^e$. The even part $'\mathcal{N}^e$ may be subdivided into a number of odd and even sub-subsets $''\mathcal{N}$, as long as the sum of their parities is even. Let us call the *final O-join* between $'\mathcal{N}^e$ and $'\mathcal{N}^o$ in a series of joins between clusters within the same growth iteration a *FO-join*, and all others O-joins that play a role in constructing $'\mathcal{N}^e$ *intermediate O-joins* or *IO-joins*. The PDC needs only to be executed on the even subset in the FO-join $'\mathcal{N}^e$.

Lemma 4.16 *An odd node set $\mathcal{N}$ that is the result of some joins must consist of an odd subset $'\mathcal{N}^o$ and an even subset $'\mathcal{N}^e$, where the even subset $'\mathcal{N}^e$ may consist of smaller sub-subsets $''\mathcal{N}$.*

To circumvent the PDC multiplicity, the calculation is suspended as much as possible. The
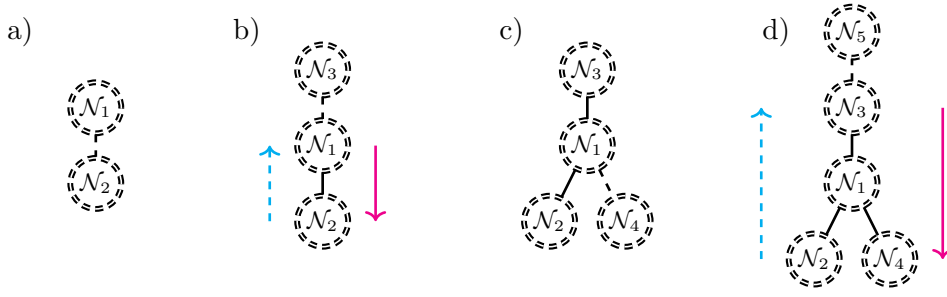
Figure 4.17: If the PDC is directly performed on the even subset in an O-join, there may be redundant PDC's in a series of O-joins and E-joins within the same growth iteration. Here we picture a series of join events between odd node sets (double lined circles, dashed for odd parity), where the O-join in b) initiates a redundant PDC.

parities and delays are required for the growth of a cluster. Thus the PDC is to be executed just before a cluster is grown, not when some O-join has occurred.

Lemma 4.17 *Parity and delay calculations are only performed on the undefined part of a node set when a cluster is grown, not directly after a join.*

The only task now is to store where the even subset $'\mathcal{N}^e$ of the FO-join starts in the ancestry of subset $\mathcal{N}^o$, as the sub-root of $'\mathcal{N}^e$ is the starting point of the DFS's of the PDC. For each join between odd node set $'\mathcal{N}^o$ and even node set $'\mathcal{N}^e$ on nodes $'n^o, 'n^e$, we additionally store the sub-root $'n_r^e$ of subset $'\mathcal{N}^e$ at the root node of the resulting set $\mathcal{N}^o$ as $n_r^o.u$, the *undefined* sub-root. If $\mathcal{N}^o$ is selected for growth as per theorem 4.1, we apply `CalcParity`$(n_r^o.u)$ and `CalcDelay`$(n_r^o.u)$ calculate parities and delays in undefined parts of the set, if it exists. The recursiveness of these function will make sure that the PDC are performed on all children nodes of (and including) $n_r^o.u$. We then call `Bloom`(n_r) per theorem 4.6.

This data structure dynamically saves the root of the undefined part of a cluster to the root node. For any IO-join, we don't know yet whether another O-join will occur, thus each IO-join to cluster $''\mathcal{N}^o$ is treated as a FO-join. For a IO-join, we thus also store the undefined sub-root u_1 at the root $r_1 = '' n_r^o$. If $''\mathcal{N}^o$ joins with other clusters in subsequent E-join to cluster $'\mathcal{N}^e$ and lastly the "real final" FO-join with $'\mathcal{N}^o$ to $\mathcal{N}^o$, we again store the undefined sub-root $u_2 = ' n_r^e$ at the new root of $r_2 = ' n_r^o$. Due to theorem 4.7, it is certain that u_2 is an ancestor of u_1, and the PDC will traverse over all undefined regions of the set.

Theorem 4.8 *Undefined region of an odd cluster $\mathcal{N}^o$ is defined as the sub-root u for which all children nodes including u have undefined parities and delays, and is stored at root node n_r^o. PDC is performed for $n_r^o.u$ and its children before cluster $\mathcal{N}^o$ is grown.*

4.5.8 Pseudocode

Now we have the full description of the *Balanced Bloom* alteration of the UF decoder, which we dub the *Union-Find Balanced Bloom* (UFBB) decoder. We present its pseudocode

in algorithm 6. The recursive `Grow` function of algorithm 5 has been added fully to the pseudocode in lines 7-12, as it is a crucial part of the decoder. Note that the structure of the code is mostly identical to the BCS UF decoder, where we sort the clusters growth in buckets, and apply the merge, in this case the combination of `Union` and `Join`, after each bucket iteration.

Algorithm 6: Union-Find Balanced Bloom (UFBB)

Data: buckets
Result: Set of even clusters grown according to Balanced Bloom

1 **for** bucket *in* buckets:
2   **for** cluster *in* bucket:
3   check if cluster belongs is current bucket
4   **for** node *in* cluster.$\mathcal{C}$:
5   `CalcParity(node)`
6   `CalcDelay(node, cluster)`
7   **if** node.w = node.d − cluster.d **then**
8   `Bloom(node)`, add all edges edge.support = 2 to $\mathcal{F}$
9   **else**:
10   node.w $+=1$
11   **for** child *of* node:
12   repeat lines 7-12 on child
13   **for** edge *in* $\mathcal{F}$:
14   `Union`(v_1, v_2) for edge $= (v_1, v_2)$
15   `Join`(n_1, n_2) for v_1, v_2 seeded in nodes n_1, n_2
16   `Place(cluster)` $\forall$ odd clusters

4.5.9 Complexity of Balanced Bloom

The contribution to the time complexity of the UF-EG decoder compared to the UF decoder can be divided into two parts. First is the contribution by `CalcParity` and `CalcDelay`, the parity and delay calculations (PDC), which is called *PDC complexity*. The second contribution will be caused by `Grow` of algorithm 5, as now we have to additionally traverse the node set tree's of each cluster to access its boundary edges and grow them with `Bloom` as compared to a single boundary list per cluster. We call this second contribution the *bloom complexity*.

PDC complexity

As per lemma 4.10 and 4.15, the total cost of the PDC is increased when the ancestries within subtrees change due to join operations, and node parities and delays have to be recalculated by traversal of the subtree. Per theorem 4.7 and 4.8, these calculations can be limited to the even subtrees in FO-joins. The size of the even subtrees in FO-joins, multiplied by the number of join operations thus estimates the count of these calculations.
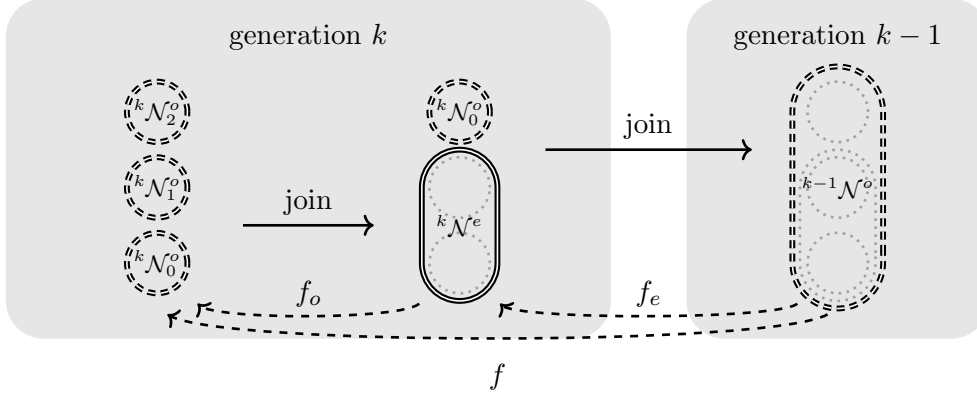
Figure 4.18: In the fragmentation of cluster $^{k+1}\mathcal{N}^o$ that belongs to ancestral generation $k-1$, we find the clusters $^k\mathcal{N}_j$ of which the joins constructed cluster $^{k+1}\mathcal{N}^o$. Any odd cluster can be fragmented into an odd (double dashed) and even (double continuous) cluster.

We will take a top-down approach to find these estimates, where we retrace the ancestor node sets in their join operations in what we call the *fragmentation* of $\mathcal{N}$.

Fragmentation of a node set For each odd node set $\mathcal{N}^o$ that is grown, it may have constructed by many joins of smaller ancestral node sets in some previous growth iteration. Before $\mathcal{N}^o$ is grown, PDC is performed on the even subset $'\mathcal{N}^e$ of the FO-join of its ancestral node sets, where its size is proportional to the *PDC count* N_{PDC}, the number of calculations performed during PDC. Subset $'\mathcal{N}^e$ may itself be the result of many IO-joins and E-joins in some previous growth iteration i. But as these joins do not add towards the PDC count, it is not crucial to know which joins have occurred. What matters to the PDC count is to know the entire set of odd subsets $''\mathcal{N}^o$ that spans $'\mathcal{N}^e$, as each of $''\mathcal{N}^o$ is subjected to PDC the first time it is grown.

We introduce a function that is called the *fragmentation* of a node set $^{k-1}\mathcal{N}^o$, that splits $^{k-1}\mathcal{N}^o$ into its ancestral node sets $\{^k\mathcal{N}_j\}$, and resembles the inverse of a join operation. Here the prefix k indicates the *ancestral generation*, where a larger k is equivalent to a more distant ancestor set of smaller subsets. As the size of the even node set in the FO-join is crucial for the PDC count, we make the distinction of *partial fragmentations* f_e and f_o. Partial fragmentation f_e is equivalent to the inverse of the FO-join to $^{k-1}\mathcal{N}^o$, where

$$f_e(^{k-1}\mathcal{N}^o) = \mathcal{F}_k^e = \{^k\mathcal{N}_0^o, {}^k\mathcal{N}^e\}. \tag{4.14}$$

Partial fragmentation f_o is equivalent to the combination of all IO-joins and E-joins that spans $^k\mathcal{N}^e$, with

$$f_e(^k\mathcal{N}^e) = \mathcal{F}_k^o = \{^k\mathcal{N}_1^o, ..., {}^k\mathcal{N}_{N_f}^o\}, \tag{4.15}$$

with the *partial fragmentation number* N_f indicating the total number of odd ancestral sets of $^k\mathcal{N}^e$. Let us call the 2 fragmentations f_e, f_o of $^{k-1}\mathcal{N}^o$ into a set of node sets

$\mathcal{F}_k = \{{}^k\mathcal{N}_0^o, ..., {}^k\mathcal{N}_{N_f}^o\}$ a *fragmentation step* f. Note that a node set $\mathcal{N}^o$ can only be fragmented if $S_{\mathcal{N}^o} \geq 3$, in which case the resulting subsets have size 1.

$$f({}^{k-1}\mathcal{N}^o) = \mathcal{F}_k = f_o(f_e({}^{k-1}\mathcal{N}^o)) = \{{}^k\mathcal{N}_0^o, ..., {}^k\mathcal{N}_{N_f}^o\} \quad | \quad S_{{}^k\mathcal{N}_j^o} \geq 3 \qquad (4.16)$$

Lemma 4.18 *Let the separation of an odd node set ${}^{k-1}\mathcal{N}^o$ into subsets $\mathcal{F}_k^o = \{{}^k\mathcal{N}_0^o, {}^k\mathcal{N}^e\}$ be the partial fragmentation f_e and subsequently ${}^k\mathcal{N}^e$ into $\mathcal{F}_k^o = \{{}^k\mathcal{N}_1^o, ..., {}^k\mathcal{N}_{N_f}^o\}$ be f_o. The combination of the two is a fragmentation step f.*

If partial fragmentation function f_e is called on a set of node sets $f_e(\{\mathcal{N}^o, \mathcal{N}^e, ...\})$, it fragments all odd node sets in the set, and f_o fragments all even node sets. Accordingly, the entire set of odd node sets $\mathcal{F}_k$ can undergo the same fragmentation step into odd subsets, resulting in a second set of node subsets $\mathcal{F}_{k+1}$. We can do this some p times on ${}^0\mathcal{N}^o$, where we have set $k - 1 = 0$, until our resulting set of node sets $\mathcal{F}_p$ consists only of smallest possible node subsets ${}^p\mathcal{N}^o$ where $S_{{}^p\mathcal{N}^o} = 1$. Let us call the series of all p fragmentation steps on ${}^0\mathcal{N}^o$ the *full fragmentation F*, with

$$F({}^0\mathcal{N}^o) = \underbrace{f(f(...f({}^0\mathcal{N}^o)))}_{\text{p times}} = \{{}^p\mathcal{N}_1^o, {}^p\mathcal{N}_2^o, ..., {}^p\mathcal{N}_{N_\sigma}^o\} \quad | \quad S_{{}^p\mathcal{N}_i^o} = 1. \qquad (4.17)$$

To find the worst case complexity, we want to maximize N_{PDC} during the construction of all the final clusters on the lattice. Let us assume that the final clusters are a single odd cluster ${}^0\mathcal{N}^o$ of size $N/2 - 1$, which is the largest odd cluster that can be grown. As the PDC is only executed on the even subsets, the sequence of join operations that maximizes the sum of even node sets sizes $S_{{}^k\mathcal{N}^e}$, in the partial fragmentations $\mathcal{F}_k^e$ in all fragmentation steps $k = [1, ..., p]$ in $F({}^0\mathcal{N}^o)$, maximizes N_{PDC}.

$$N_{PDC} = \sum_{k=1}^{p} \sum_j \{S_{{}^k\mathcal{N}_j^e} | {}^k\mathcal{N}_j^e \in \mathcal{F}_k^e\}. \qquad (4.18)$$

Proposition 4.1 *The worst-case delay complexity is computed by maximizing N_{PDC} of the full fragmentation of ${}^0\mathcal{N}^o$ with $S_{{}^0\mathcal{N}^o} = N/2 - 1$.*

Partial fragmentation number For the partial fragmentation f_o, an even set ${}^k\mathcal{N}^e$ can be separated in $2n_o$ odd and n_e even subsets, where $n_o \geq 1$ and $n_e \geq 0$. The fragmentation number is thus $N_f = 2n_o + n_e$. But any even subset will be subjected to the same partial fragmentation f_o in the full fragmentation. Thus we can set $n_e = 0$. To find n_o, let us consider two cases where $n_o = 1$ or $n_o = 2$. If an even node set ${}^{k-1}\mathcal{N}^e$ is fragmented with $N_f = 2$, a fragmentation step $f({}^{k-1}\mathcal{N}^e) = f_e(f_o({}^{k-1}\mathcal{N}^e))$ produces the following partial fragmentation sets:

$$\begin{aligned} \mathcal{F}_{k-1}^o &= \{{}^{k-1}\mathcal{N}_1^o, {}^{k-1}\mathcal{N}_2^o\}, \\ \mathcal{F}_k^e &= \{{}^k\mathcal{N}_{1,0}^{o,o}, {}^k\mathcal{N}_1^{o,e}, {}^k\mathcal{N}_{2,0}^{o,o}, {}^k\mathcal{N}_2^{o,e}\}. \end{aligned}$$
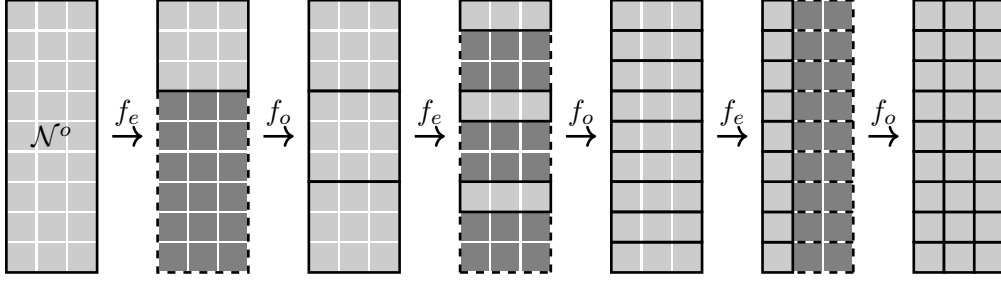
Figure 4.19: The full fragmentation of $\mathcal{N}^o$ per equation 4.17. Each odd node set in the fragmentation is a rectangle with continuous lines, and even node set has dashed lines. Each square is equivalent to a node, where the sum of all dark shaded squares is N_{PDC}. Here, N_{PDC} is maximized as $N_{f_e} = N_f = 2$ and $R_j = \frac{1}{2}$.
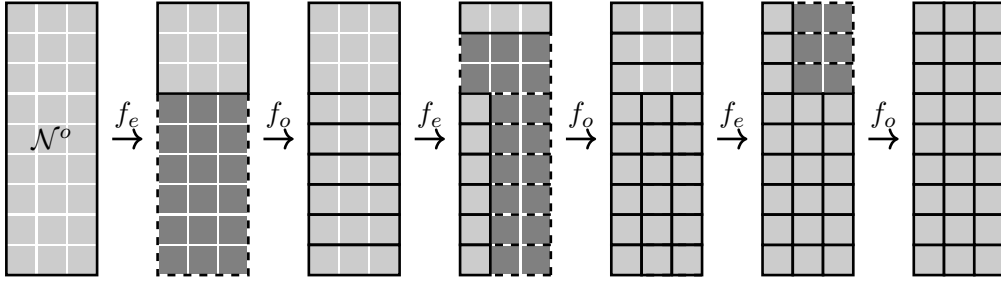

Figure 4.20: A full fragmentation of $\mathcal{N}^o$ where in the first f_0, the fragmentation number is increased to $N_{f_o} = 6$. The number of dark shaded squares or N_{PDC} has decreased from the fragmentation with optimal settings (figure 4.19).

For $N_f = 4$, the partial fragmentation sets are

$$
\begin{aligned}
\mathcal{F}_{k-1}^e &= \{{}^{k-1}\mathcal{N}_1^o, {}^{k-1}\mathcal{N}_2^o, {}^{k-1}\mathcal{N}_3^o, {}^{k-1}\mathcal{N}_4^o\}, \\
\mathcal{F}_k'^e &= \{{}^{k}\mathcal{N}_{1,0}^{o,o}, {}^{k}\mathcal{N}_1^{o,e}, {}^{k}\mathcal{N}_{2,0}^{o,o}, {}^{k}\mathcal{N}_2^{o,e}, {}^{k}\mathcal{N}_{3,0}^{o,o}, {}^{k}\mathcal{N}_3^{o,e}, {}^{k}\mathcal{N}_{4,0}^{o,o}, {}^{k}\mathcal{N}_4^{o,e}\}.
\end{aligned}
$$

If the size of $S_{\mathcal{N}^e}$ is large enough, and we fragment in the same ratio (see next paragraph), the sum of even node set sizes in these two kinds of fragmentations will be the same. However, the number of subsets in each fragmentation step has increased by a factor of 2, which means that the average size of subsets have decreased by 2. Consequently, the node set size decreases faster towards the minimum size of 3 nodes as more fragmentation steps are applied. As the sum of even node set sizes in each fragmentation step is the same, increasing n_o will decrease the number of fragmentation steps and consequently the PDC count N_{PDC} per equation 4.18. Hence N_{PDC} is maximized for minimal $n_o = 1$, and we find that $N_f = 2$.

Partial fragmentation ratio To complete the fragmentation formalism, we will need to find the *fragmentation ratios* R_0, R_1, R_2 of a fragmentation step. The fragmentation ratios determine the node set sizes of the subsets in $\mathcal{F}_k$ with respect to the size of $^{k-1}\mathcal{N}^o$, where $R_i S_{k-1}\mathcal{N}^o$ is the size of subset $^k\mathcal{N}_i^o$. Note that R_0 corresponds to the odd subset from f_e, and R_1, R_2 to the odd subsets in f_o.

Lemma 4.19 *Let the fragmentation ratios R_0, R_2, R_2 be the relative set sizes of the odd subsets in the fragmentation set $\mathcal{F}_k = \{^k\mathcal{N}_0^o, {}^k\mathcal{N}_1^o, {}^k\mathcal{N}_2^o\}$ with respect to set $^{k-1}\mathcal{N}^o$, where*

$$R_i = \frac{S_{k\mathcal{N}_i^o}}{S_{k-1}\mathcal{N}^o} \tag{4.19}$$

Recall lemma 4.17 that the delay calculations are only done before a cluster is grown. During this grow process, some n_v vertices are added to the cluster, and some join operations can occur. If no join operations occur, the node set stays unchanged, and the cluster is allowed to continue growth without delay calculations per lemma 4.6. We want to minimize n_v, as each added vertex here is not a node that can possibly count towards N_{PDC}. Thus in each growth iteration of a cluster, some join operation must occur for the maximization of N_{PDC}.

Take the first fragmentation sets $\mathcal{F}_k^e = \{\mathcal{N}_0^o, \mathcal{N}^e\}$ and $\mathcal{F}_k = \{\mathcal{N}_0^o, \mathcal{N}_1^o, \mathcal{N}_2^o\}$ of cluster $^{k-1}\mathcal{N}^o$. These partial fragmentations correspond to 2 join operations, the E-join between two odd clusters $\mathcal{N}_1^o, \mathcal{N}_2^o$ in f_o, and the FO-join between odd and even clusters $\mathcal{N}_0^o, \mathcal{N}^e$ in f_e. If we want to minimize n_v in f_o, these odd clusters must grow within the same bucket b_i, which means that $S_{\mathcal{V}_1} = S_{\mathcal{V}_2}$. Note that these are the cluster sizes and not node set sizes. For f_e, the merge event is caused by growth of $\mathcal{N}_0^o$ in either some larger or equal bucket $b_j \geq b_i$ where $S_{\mathcal{V}_0} \geq S_{\mathcal{V}_1}$. This leaves us with $S_{\mathcal{V}_0} \geq S_{\mathcal{V}_1} = S_{\mathcal{V}_2}$. To maximize N_{PDC}, we want to maximize $S_{\mathcal{N}^e} = S_{\mathcal{N}_1^o} + S_{\mathcal{N}_2^o}$ in f_e. Recall from equation 4.3 that $S_{\mathcal{N}} \leq S_{\mathcal{V}}$. We assume the largest possible node set size $S_{\mathcal{N}} = S_{\mathcal{V}}$ to find that $S_{\mathcal{N}^e}$ is largest if $S_{\mathcal{V}_0} = S_{\mathcal{V}_1}$. We can therefore conclude that $S_{\mathcal{N}_0^o} = S_{\mathcal{N}_1^o} = S_{\mathcal{N}_2^o}$ and $R_0 = R_1 = R_2 = \frac{1}{3}$

Lemma 4.20 *A fragmentation step of $^{k-1}\mathcal{N}^o$ is maximized in $S_{k\mathcal{N}^e}$ if the fragmentation ratios take the value $R = \frac{1}{3}$.*

Time complexity The last unknown parameter for the delay calculation is p, the number of fragmentation steps. If we assume that in each growth step not a single non-node vertex is added $n_v = 0$, the full fragmentation of some node set $\mathcal{N}^o$ is just the continuous division of the set in 3 parts per lemma 4.20, which can be calculated easily.

$$p = \log_3(S_{\mathcal{N}^o}) \tag{4.20}$$

In each fragmentation step $\mathcal{F}_k^e$, f_e is equivalent to the join operation of odd node sets with even node sets where the sum of odd sets sizes is

$$\sum_j \{S_{k\mathcal{N}_j^o} | {}^k\mathcal{N}_j^o \in \mathcal{F}_k^e\} = \frac{1}{3} S_{\mathcal{N}^o}, \tag{4.21}$$
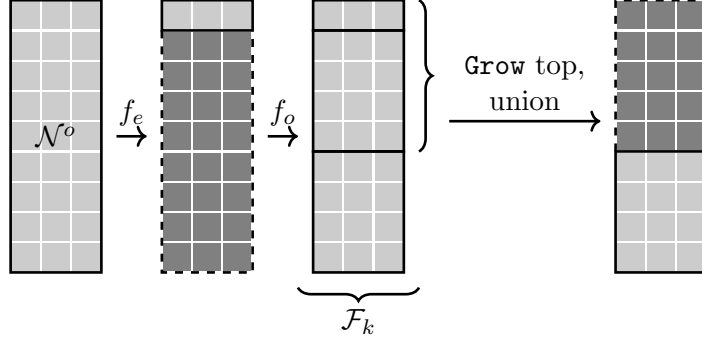
Figure 4.21: A fragmentation step f of $\mathcal{N}^o$, where the fragmentation ratios are not optimal $R_i \neq \frac{1}{3}$. This fragmentation is not possible, as the clusters in $\mathcal{F}_k$ will grow and join in a different path according to the rules of weighted growth.

and the sum of even node set of sizes is

$$\sum_j \{S_{k\mathcal{N}_j^e} |^k \mathcal{N}_j^e \in \mathcal{F}_k^e\} = \frac{2}{3} S_{\mathcal{N}^o}. \tag{4.22}$$

This approximation is true as we have taken $S_{\mathcal{N}} = S_{\mathcal{V}}$ and $n_v = 0$. Filling in equation 4.20 and 4.22 in 4.18, we find that

$$\begin{aligned}
N_{PDC} &\leq \sum_{k=1}^{p} \sum_j \{S_{k\mathcal{N}_j^e} |^k \mathcal{N}_j^e \in \mathcal{F}_k^e\}. \\
&= \sum_{k=1}^{\log_3(S_{\mathcal{N}^o})} \frac{2}{3} S_{\mathcal{N}^o} \\
&= \frac{2}{3} S_{\mathcal{N}^o} \log_3(S_{\mathcal{N}^o}) \tag{4.23}
\end{aligned}$$

The node set size of set is bounded by the lattice size $\mathcal{N}^o \leq N$. The worst case time complexity of the delay computation is thus bounded by $\mathcal{O}(N \log_3(N))$. The average-case complexity is even lower as it is quite certain that not all vertices are nodes such that $S_{\mathcal{N}} < S_{\mathcal{V}}$ and $n_v > 0$.

Bloom complexity

To grow a cluster represented by a node set $\mathcal{N}$, we have to traverse the entire set from root to stem to iterate over each boundary list that are stored at the nodes. Let's call the total number of times any node is traversed by `Bloom` N_{bloom}.

Similar to the previous section we make the assumption of a maximum number of nodes on the lattice where in each cluster $S_{\mathcal{N}} = S_{\mathcal{V}}$ and $n_v = 0$. Recall that every odd node

set $^kN_j^o$ in each fragmentation set $\mathcal{F}_k$ is subjected to growth in each partial fragmentation, and that we start with a maximum number of smallest cluster of size $S_{p\mathcal{N}} = S_{p\mathcal{V}} = 1$. Thus we are certain that with this assumption we have the upper bound in N_{bloom}.

$$N_{bloom} \leq \sum_{k=1}^{p} \sum_{j} \{S_{k\mathcal{N}_j} |^k\mathcal{N}_j \in \mathcal{F}_k\} \tag{4.24}$$

For a full fragmentation of $\mathcal{N}$ of size $S_\mathcal{N}$, the sum of all set sizes in each fragmentation set $\mathcal{F}$ is

$$\sum_{j} \{S_{k\mathcal{N}_j} |^k\mathcal{N}_j \in \mathcal{F}_k\} = S_\mathcal{N}. \tag{4.25}$$

By filling in p we find that

$$
\begin{aligned}
N_{bloom} &\leq \sum_{k=1}^{p} \sum_{j} \{S_{k\mathcal{N}_j} |^k\mathcal{N}_j \in \mathcal{F}_k\} \\
&= \sum_{k=1}^{\log_3(S_{\mathcal{N}^o})} S_\mathcal{N} \\
&= S_{\mathcal{N}^o} \log_3(S_{\mathcal{N}^o}),
\end{aligned}
\tag{4.26}
$$

which again corresponds to a worst case time complexity that is bounded by $\mathcal{O}(N \log_3(N))$.

4.5.10 Boundaries

For the UF decoder on surfaces with boundaries, we introduced the concept of *boundary vertices* that in contrast to normal vertices are not equivalent to stabilizers generators, measurements or ancillary qubits. During formation of the spanning forest F_C of a cluster, we must make sure that F_C does not contain more than 1 element of the set of boundary vertices $\delta\mathcal{V}$, as multiple elements of $\delta\mathcal{V}$ is equivalent to a cycle.

The addition of boundaries requires a new type of node element, the *boundary node* β, that is exclusive to boundary vertices of $\delta\mathcal{V}$, and are initiated on a boundary vertex if a cluster grows into the boundary. For a cluster, it is already defined in the vanilla UF decoder that there can be only 1 boundary vertex in $\mathcal{V}$, and therefore only one boundary node in $\mathcal{N}$. As a result, a boundary node will always be a trailing node in $\mathcal{N}$ with no children, and will never be the root node. However, the always-trailing boundary node always has parity 1, as a matching with the boundary is equally valid as a matching with another syndrome. The addition of boundary nodes just requires a small alteration to algorithm 3.

Algorithm 7: CalcParity for surfaces with boundaries

Data: node

Result: Defined parities for all children of node

1 parity = Sum([1− CalcParity(child) ∀ child *of* node]) %2

2 **if** node ≡ σ **then**

3 | node.p = parity

4 **elif** node ≡ j **then**

5 | node.p = 1− parity

6 **elif** node ≡ β **then**

7 | node.p = 1

8 **return** node.p

For a surface containing N qubits, the number of boundary elements scales with $\sqrt{N}$. The number of node elements is thus bounded by $N + \sqrt{N}$. The added complexity due to the boundary elements will therefore not exceed some linear factor and remains the same as previously computed.

4.5.11   Erasure noise

The inspiration for the UF decoder is the Peeling decoder [8], that only accounted for *erasure* errors. As the UFBB decoder is a descendant of the original Peeling decoder, we naturally needs to make sure that it can also solve erasure errors. The UF decoder solves for Pauli errors by considering each non-trivial syndrome as an single vertex odd cluster, and growing odd cluster in size until only even clusters remain. Each even cluster can than be considered as an pseudo-erasure to be solved by the Peeling decoder. Real erasures undergo the same growth, but have larger initial sizes.

To account for these erasures, we must construct the node sets for these initial erasure clusters. We can easily check that for an erasure-cluster, the PMW for each neighboring vertex is different. Each vertex in the cluster is therefore a node in $\mathcal{N}$, where each syndrome vertex is a syndrome-node σ, and every other vertex is a junction-node j. Note that if the erasure is connected to the boundary, we need to make sure that only a single edge is connected to the boundary, where the single boundary vertex in the cluster naturally is a boundary node β. After constructing these initial clusters and node sets, we can proceed to the UFBB algorithm.

Appendices

Appendix A

Bucket sort

Maximum number of growth iterations To categorize the clusters into buckets, we need to know how many buckets we would need. To find out, let us look at two generators that are placed maximally far from each other. On a even toric lattice with length l, they are placed $l/2$ horizontally and $l/2$ vertically from each other. If clusters from each bucket grow simultaneously by a half-edge, these two single-generator clusters would meet each other after l iterations, or including odd lattices, after $2\lfloor l/2 \rfloor$ iterations. As illustrated in figure A.1, we see that now the entire lattice is covered, thus this is exactly the number of iterations needed.

Lemma A.1 *Given a toric lattice with size $\{l, l\}$, a maximum number of $N_B = 2\lfloor l/2 \rfloor$ buckets is required for all generators to be connected, where clusters from each bucket is grown simultaneously.*
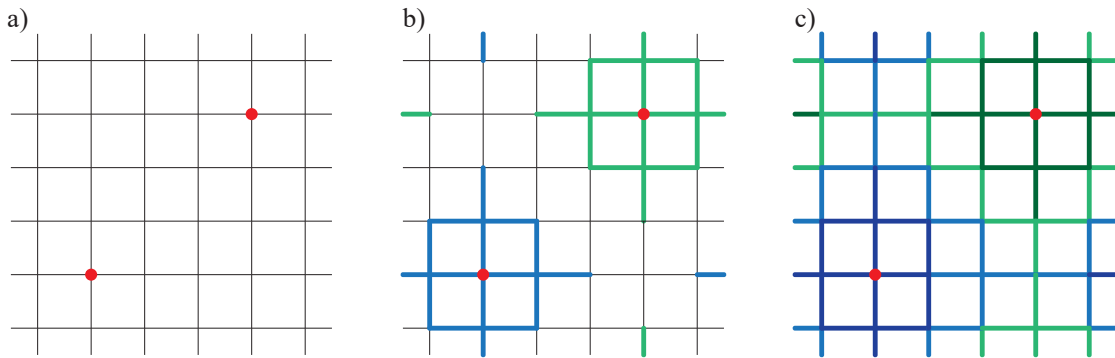

Figure A.1: If two single-generator clusters are placed a) maximally far from each other, b) a maximal number of growth iterations is needed. c) After $2\lfloor l/2 \rfloor$ iterations, the entire lattice is covered.

Analytical cluster categorization To sort or place the clusters into the buckets, we can again look at the case of the two generators. We presume that each growth iteration of the cluster is started from a different bucket. Knowing this, we can use the sequence of the size of the single-generator cluster as it grows.

$$S_{cluster,i} = 1, 1, 5, 5, 13, 13, 25, 25, 41, 41, 61, 61, ... \tag{A.1}$$

Note that a cluster does not increase in size in alternating iterations, as the growth of half-edges does not add new vertices to the cluster. We ignore these duplicate numbers for now. We find that this series of numbers can be written as a finite sum.

$$S'_{cluster}(i) = 1 + \sum_{0}^{x=i} 4x = 2i(i+1) + 1 \tag{A.2}$$

We now have a very simple analytical function to place clusters into buckets, where the number of elements per bucket increases linearly with $4i+8$. For a cluster of size S, its bucket number i can be calculated by rewriting and flooring equation A.2 $i = \lfloor(\sqrt{2S-1}-1)/2\rfloor$. Now earlier we disregarded the duplicate entries is $S_{cluster,i}$, so we need to take an extra step to check whether a cluster is in its half-grown state. These buckets are illustrated in figure A.2a.

Definition A.1 (Bucket place method 1) *A cluster with size S is placed in bucket number i of l buckets with $i = 2\lfloor(\sqrt{2S-1}-1)/2\rfloor$ if cluster is fully grown, or $i = 2\lfloor(\sqrt{2S-1}-1)/2\rfloor + 1$ otherwise.*

a) place method A

1-4 full	1-4 half	5-12 full	5-12 half	13-24 full	13-24 half
0	1	2	3	4	5

● ● ●

b) place method B

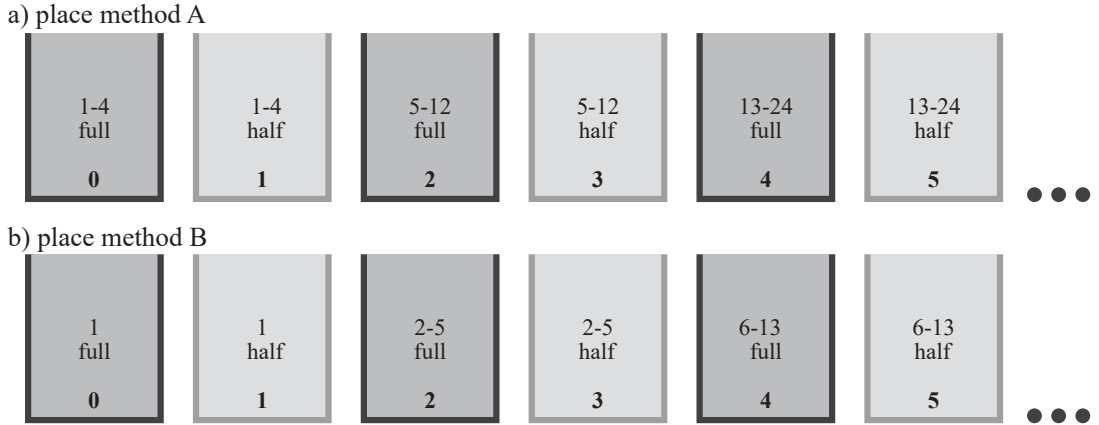|					
1 full	1 half	2-5 full	2-5 half	6-13 full	6-13 half
0	1	2	3	4	5

● ● ●

Figure A.2: The first six buckets for a) method A from definition A.1 and b) method B from definition A.2. Even buckets contains fully grown clusters, odd buckets contain half-grown clusters of the same size.

Single-generator clusters, or clusters with size 1, is never caused by an only an erasure error. It might therefore by useful to give these clusters their own bucket, such that these single-generator clusters are always grown first, such as in figure A.2b.

Definition A.2 (Bucket place method 2) *For separate single-generator buckets, check whether the cluster has size 1, in which case the clusters belongs in bucket 0 or 1 depending on its growth state. Otherwise, a cluster with size S belongs in bucket number i of l buckets with $i = 2\lfloor(\sqrt{2S-3}-1)/2\rfloor + 1$ if cluster is fully grown, or $i = 2\lfloor(\sqrt{2S-3}-1)/2\rfloor + 2$ otherwise.*

To compare method 1 of definition A.1 and method 2 of definition A.2, let us find the size of the largest odd possible on a lattice. Given that any two odd parity clusters grow simultaneously, this size is $S_{max} = (\lfloor l/2\rfloor - 1)l$. We calculate the bucket number (for the half-grown state) for both methods and plot them versus the lattice size in figure A.3. The bucket number of the largest odd parity cluster for method 2 is exactly the largest available bucket. Therefore method 2 is most probably superior.
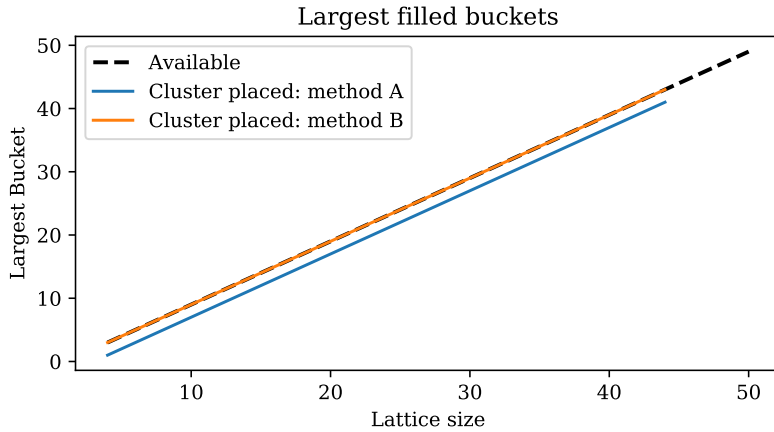

Figure A.3: The largest bucket that will be utilized for methods A and B (see fig. A.2). Method A does not utilize the largest two buckets.

The big benefit of this type of bucket sort is that we do not need to look at existing entries in the buckets. Only thing that is necessary is to initialize the buckets, which can be done in linear time.

Growing a bucket The procedure for the Union-Find decoder using the bucket sort algorithm is now to sequentially grow the buckets starting from bucket 0, which contains all the single-generator clusters. From each bucket, which can be a simple list, clusters are popped from the bucket, grown, and placed into a new bucket, a process which we will call *growing a bucket*. In the case that no merge happens, clusters from even bucket i are placed in odd bucket $i + 1$. However, what happens to clusters grown from odd bucket j?

It would be problematic if a cluster can be placed in even bucket $j - 1$, which has the same size range, as that bucket comes first in the sequence, and thus will not be grown again. To proof that this will never happen, let's state the following:

Lemma A.2 *A cluster with size $S_i = 2(i + 1)(i + 2) + 1$ grown from a single-generator cluster has the least number of neighbor vertices of all clusters of size S_i.*

This is equivalent to saying that single-generator seeded cluster with size S_i will have the least amount of new vertices added to itself for all clusters with the same size. This is fairly easy to proof, as there can be no more compactly organized cluster than a cluster grown from a single generator. We know odd parity clusters from this most compactly organized set must go into the next bucket, as the size of this set if clusters is what defines the sequence (eq. A.1). Thus less compactly organized odd parity clusters must at least grow into the next even bucket, and potentially grow into a higher even bucket.

Lemma A.3 *In the case of no merging event, clusters from even bucket i are grown into bucket $i + 1$, and cluster from odd bucket j are grown into bucket $j + 2k + 1$ with $k \in \mathbb{N}_0$.*

We now know that our buckets can be grown sequentially, where we need not to worry about buckets that already been emptied.

Merging   Up until now we have avoided talking about the merging of clusters, so let us finally address this. In the original paper of the Union-Find algorithm, when two clusters merge, one needs to check for the larger cluster between the two, and make the smaller cluster the child of the bigger cluster, which lowers the depth of the tree and is called the *weighted union rule.* Applied to the toric lattice, the Union-Find decoder also needs to append the boundary list (which contains all the boundary edges of a cluster) of the smaller cluster onto the list of the larger cluster.

In our application, instead of appending the entire boundary list, we just add a pointer at the parent cluster to the child cluster. As a parent can have many children, the pointers are appended to a *children list.* When growing a cluster, we first check if this cluster has any child clusters. If yes, these child clusters will be grown first by popping them from the list, but any new vertices will always be added to the parent cluster. Also during and after a merge, we make sure that any new vertices are always added to the parent cluster. Any child will exist in the list of a parent for one iteration, after which its boundaries will be grown, and the child is absorbed into the parent. This method also works recursively by keeping track of the root cluster instead of the parent cluster, and many levels of parent-child relationships can exists.

Faulty entries   Now let us first be clear: *only uneven parity clusters will be placed in buckets, but each bucket does not only contain uneven parity clusters.* As a merge happens between two uneven parity clusters A and B during growth of B, cluster A has already been placed in a bucket. But clusters A is now part of cluster AB and is even. So, to prevent
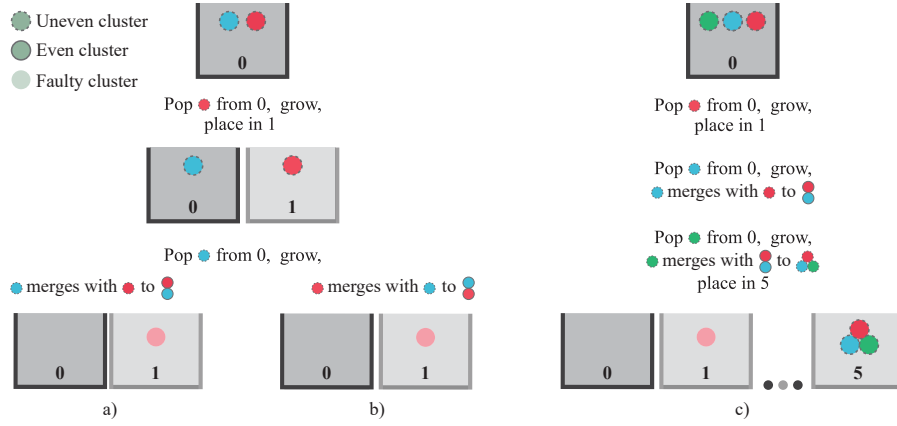
70

Figure A.4: bla

growth of the *faulty entry*, we just need to have an extra check of the parity of the root cluster.

Wastebasket   If the lattice consists of more than just the two single-generator clusters, it is entirely possible for odd parity clusters to grow larger than $S_{max} = (l/2 - 1)l$. The bucketing formulas also does not prevent itself from outputting a larger bucket number than the available number of buckets. The elegance of this method is that whenever a cluster has a bucket number $i_1 > l$, there will always be another *smaller* cluster with bucket number $i_2 \leq l$ that will merge into the larger cluster, which evens the parity. In another words, if a clusters bucket number i is computed to be larger than l, we can figuratively put the cluster into the *wastebasket*.

Bibliography

[1]   CJ Ballance et al. "High-fidelity quantum logic gates using trapped-ion hyperfine qubits". In: *Physical review letters* 117.6 (2016), p. 060504.

[2]   A Robert Calderbank and Peter W Shor. "Good quantum error-correcting codes exist". In: *Physical Review A* 54.2 (1996), p. 1098.

[3]   Andrew Steane. "Multiple-particle interference and quantum error correction". In: *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 452.1954 (1996), pp. 2551–2577.

[4]   John Preskill. "Reliable quantum computers". In: *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 454.1969 (1998), pp. 385–410.

[5]   Daniel Gottesman. *Stabilizer Codes and Quantum Error Correction*. 1997. arXiv: `quant-ph/9705052 [quant-ph]`.

[6]   Raymond Laflamme et al. "Perfect Quantum Error Correcting Code". In: *Phys. Rev. Lett.* 77 (1 July 1996), pp. 198–201. DOI: `10.1103/PhysRevLett.77.198`. URL: `https://link.aps.org/doi/10.1103/PhysRevLett.77.198`.

[7]   A Yu Kitaev. "Fault-tolerant quantum computation by anyons". In: *Annals of Physics* 303.1 (2003), pp. 2–30.

[8]   Nicolas Delfosse and Gilles Zémor. "Linear-time maximum likelihood decoding of surface codes over the quantum erasure channel". In: *arXiv preprint arXiv:1703.01517* (2017).

[9]   Nicolas Delfosse and Naomi H Nickerson. "Almost-linear time decoding algorithm for topological codes". In: *arXiv preprint arXiv:1709.06218* (2017).

[10] Robert Endre Tarjan. "Efficiency of a good but not linear set union algorithm". In: *Journal of the ACM (JACM)* 22.2 (1975), pp. 215–225.

[11] N. Leijenhorst. "Quantum Error Correction, Decoders for the Toric Code". Applied Mathematics and Applied Physics BSc. Bachelor Thesis. Delft University of Technology, July 2019.