Delft University of Technology

Master's Thesis

Quantum Error Correction of Patch-Loss of the Surface Code

Author
S. Hu

Supervisor
D. Elkouss

November 22, 2019

Contents

1 Preliminary report **3**

2 Introduction **7**

3 The toric code **10**

3.1 Decoders 10

3.1.1 The optimal decoder 10

3.1.2 Minimum Weight Perfect Matching 10

3.1.3 The Union-Find decoder 10

3.1.4 Finding clusters 15

3.1.5 Bucket Cluster Sort 15

3.1.6 Object oriented approach 18

4 Threshold simulations **20**

Appendix A Bucket sort **22**

Chapter 1

Preliminary report

Introduction

Quantum computing has the potential to transcend the information technology as we know it. Small scale quantum systems are already possible today and the goal is to scale up these quantum architectures to build practical quantum devices. One approach to do this is to by networking many simple processor cells together through quantum links, avoiding the necessity to build a single complex structure. Processor cells that are located physically close to each other are connected by "short" links and lie in a patch. Patches that are located physically far from each other can in turn be connected by "long" links, such as remote optical connections. The total state of system, which contains the stored information, is shared across these patches, such that it can be accessed in either one of these patches.

This is somewhat analogous to the idea of a shared database. Many online services that we use today rely on servers that host the data that we want to view, store or edit. This data is often not stored on a single server, but copied to many others, in a shared database. In case one of these servers goes offline due to file corruption or an electricity outrage, the data is not lost, and can still be accessed on another server in the cluster.

In our Quantum network, information cannot be copied across different processor cells due to the no cloning theorem. In stead, it is shared across cells through entanglement. A cell can also go "offline", when a qubit or multiple qubits are lost from the system due to some interaction with the environment. This process is called decoherence, also described with *loss* or *erasure*. Luckily, if the losses are not too much, these cells can be restored through quantum error correction (QEC) such that the quantum state or encoded information can still be extracted from the system.

Quantum errors

Errors that can occur during Quantum computation can generally be classified as 1) noise, in which there is an error are within the computational basis, or as 2) a loss, in which the qubit is taken out of the computational basis. Losses are both detectable and locatable, which means that a higher rate of loss (p_{loss}) can be tolerated than noise or computational errors (p_{com}). The process of finding and correcting these errors is called decoding.

Kitaev's surfaces codes are defined by a set of stabilizers which act on a set of physical qubits that lie on the edges of a square lattice [1]. The stabilizers commute, and are generated by plaquettes (group of Z operators), or by stars (group of X operators). Logical operates corresponds to a set of stabilizer operators along a homologically nontrivial cycle. Any homologically equivalent set of operators can be used to measure the physical qubit operator. Therefore, in the case of a qubit loss, another set of operators can be used, if there is no *percolated* region of losses that span the entire lattice.

To decode for computational errors, one measures the stabilizer generators, which returns eigenvalue -1 on the edges of the error chains or syndromes, and can be corrected by finding a nontrivial closing chain, which either equals a stabilizer measurement that corrects the error, or a logical operator which equals a logical error. This problem is equivalent to the two dimensional random-bond Ising model (RBIM), where the shortest path needs to be found between matching pairs. The closing chain is found using the Edmonds' minimum weight perfect matching (MWPM) algorithm. This algorithm scales quadratic in time as the lattice size increases [2]. More recently, an almost-linear decoding approach has been described by Delfosse et al. [3].

There are also multiple methods to decode for losses on the surface code. Stace et. al [2, 4] describes the method of so-called superplaquettes and superstars, in which the lost qubit is accounted for by combining neighboring stabilizers. The resulting lattice can be than decoded using the same MWPM algorithm. Delfosse et al [5] describes a linear-time maximum likelihood method to decode for losses. Here, the errors are found in a *peeling* algorithm that iteratively peels branches away from a tree of possible error chains until the lost qubits remain.

Patch loss

In terms of the surface code, a patch is a set of neighboring qubit cells on the lattice that lie physically close to each other. The entire lattice is built by connecting multiple patches. It is possible that cells within the same patch suffer the same decoherence, due to their physical vicinity, such that entire patches may be lost. The Quantum links that connect these patches, in turn, may suffer a larger amount of noise due to the distance it must cross.

These patch losses may be investigated using the decoding algorithms described above.

There are two main question to be answered here. The first one involves the size of the patches. What are tolerable sizes of patches such that in the event of a patch loss, the encoded Quantum information can still be extracted from the system? The second question concerns the Quantum links between these patches. What effects does the patch model on the surface have on the tolerable noise thresholds, especially on the thresholds of the Quantum links that connect different patches?

Project

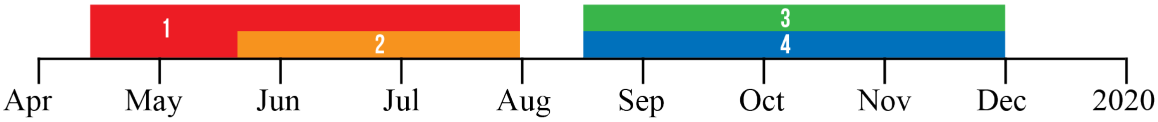
This project will focus on the two problems of the patch model described in the previous section. We will use simulations to calculate for the thresholds on patch sizes and noise. Most probably, we will try to build this simulation on top of the fault-tolerance simulations by N. Nickerson [6], whose thesis will also be used as our foundation on quantum error correction. Furthermore, the introduction on topological codes will be provided by the lecture notes of D. Browne [7]. We we use the knowledge acquired in the courses Advanced Quantum Mechanics (AP3051G), Computational Physics (AP3082D), as well as the EDX courses The Building Blocks of a Quantum Computer (DelftX - QTM2x, QTM3x) and Quantum Cryptography (CaltechDelftX - QuCryptox), which are similar to the TU Delft courses AP3421 and CS4090.

The project will be done under supervision of David Elkouss, who leads the Elkouss group, which is part of the Roadmap Quantum Internet and Networked Computing at QuTech. Weekly meetings are planned for discussions and evaluations. A presentation for the group will be held after a month. Furthermore, we also expect to be working closely together with Sebiastian de Bone, who is a Phd student at the Elkouss Group.

The project can be divided into 5 successive objectives:

1. Theory and prediction of model with patch-loss of the surface code.
2. Calculation of threshold on patch sizes.
3. Definitions of noise between patches.
4. Combine patch-loss and noise between patches.
5. Improve decoder.

The preparation for the master thesis ends on April 12, 2019. From here, the first 3 months of the project, until July 2019, we expect to be working on objective 1, expanding our theoretical knowledge of the Quantum erasure code and formulating definitions of the patch model. Ideally, we will also have some simulations results for objective 2 by the end of July. We will take a short break in August, whereafter we will continue on objectives 3 and 4. Objective 5, improving the decoder, is an optional objective. The mechanism of the decoders are so complicated, that an improvement may be a project on its own. However, due to my particular interest in these decoders, I will be alert on trying to find a point of improvement if possible. The timeline of this project will look as such.



If the simulation of the patch sizes (2) will be so complicated such that after it cannot be completed before the end of July, we will drop objectives 3 and 4 and focus on the first two objectives. The deadline for the thesis is set to the end of November, such that a defense may be held in December.

Chapter 2

Introduction

Bell states

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

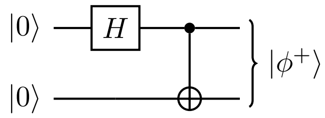
$$\left|\phi^{+}\right\rangle = \frac{1}{2}\big(|00\rangle_{AB} + |11\rangle_{AB} \big) \tag{2.1}$$

$$\left|\phi^{-}\right\rangle = \frac{1}{2}\big(|00\rangle_{AB} - |11\rangle_{AB} \big) \tag{2.2}$$

$$\left|\psi^{+}\right\rangle = \frac{1}{2}\big(|01\rangle_{AB} + |10\rangle_{AB} \big) \tag{2.3}$$

$$\left|\psi^{-}\right\rangle = \frac{1}{2}\big(|01\rangle_{AB} - |10\rangle_{AB} \big) \tag{2.4}$$

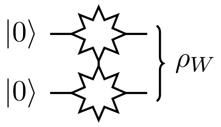
The Bell state that we want to prepare, $|\phi^{+}\rangle$, can be created by applying the Hadamard and the CNOT gate on two qubits.



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,

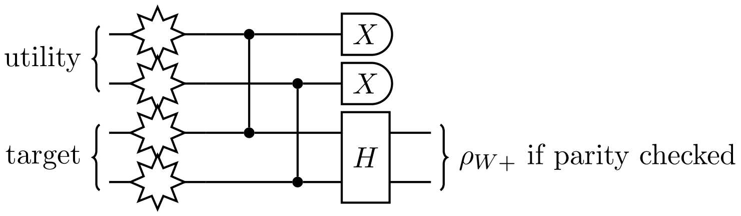
$$\rho_W = F\,|\phi\rangle\,\langle\phi| + \frac{1-F}{3}\sum_{i=1,2,3}|\psi_i\rangle\,\langle\psi_i| \tag{2.5}$$

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

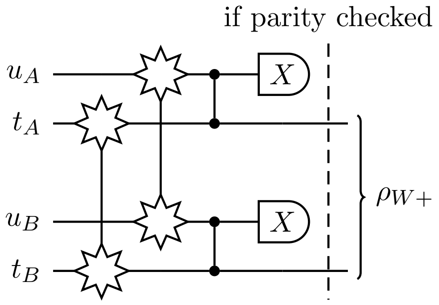


Single selection

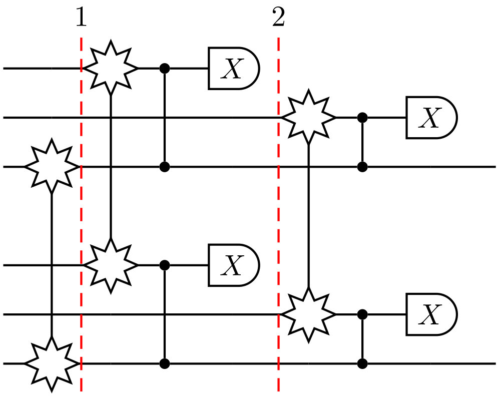
Luckily, the fidelity of a noisy Bell pair can be enhanced through entanglement purification. In the process of *single selection*, a single noisy Bell pair is used (utility pair) in order to enhance the fidelity of another noisy bell pair (target pair). Here CZ gates are applied between the first and 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.

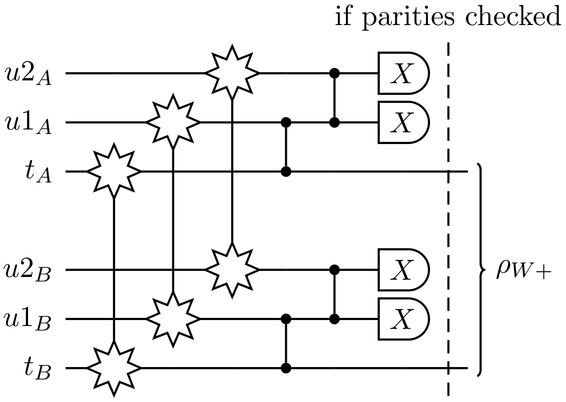


Entanglement pumping

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 3

The toric code

3.1 Decoders

3.1.1 The optimal decoder

3.1.2 Minimum Weight Perfect Matching

3.1.3 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 $\mathcal{O}(n)$ [5]. 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 [3]. 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.

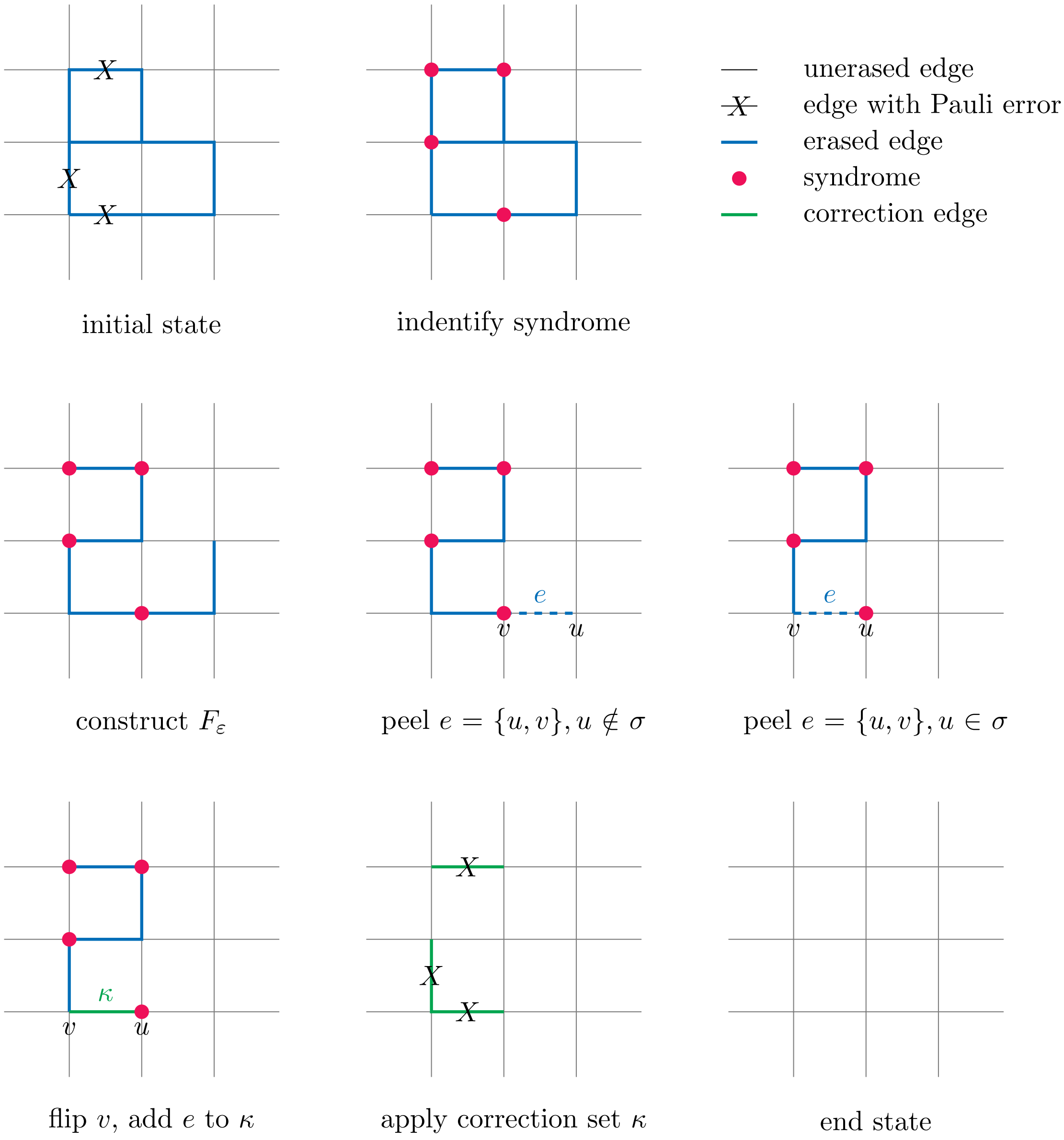
We will now describe the Peeling decoder as is presented in Algorithm 1. In step 1, we will remove all cycles present in ε. We construct a spanning forest F_ε inside erasure ε, the maximal subsect of edges of ε that contains no cycles and spans all vertices of ε. From here, we loop over all edges in 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.

Algorithm 1: Peeling decoder [5]

```
   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 $F_\varepsilon$ of $\varepsilon$
2  initialize $\kappa$ by $\kappa = \emptyset$
3  while $F_\varepsilon \neq \emptyset$ do
4      pick a leaf edge $e = u, v$ with pendant vertex $u$, remove $e$ from $F_\varepsilon$
5      if $u \in \sigma$ then
6          add $e$ to $\kappa$, remove $u$ from $\sigma$ and flip $v$ in $\sigma$
7      else:
8          do nothing
9  return $\kappa$
```

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 $\sigma = \emptyset$, 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 $\mathcal{O}(n)$.



initial state

indentify syndrome

construct F_ε

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

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

flip v, add e to κ

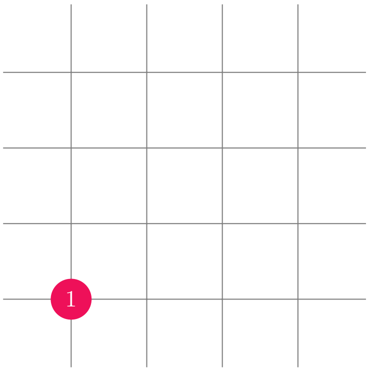
apply correction set κ

end state

Figure 3.1: Schematic representation of the Peeling decoder.

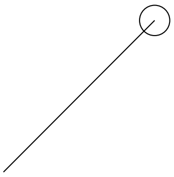
Growing erasures

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 iteratively 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 wether 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) [8].



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.

The routine The full routine of the Union-Find decoder as originally described ([3], 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 [3]

```
Data: A graph $G = (V, E)$, an erasure $\varepsilon \subset E$ and syndrome $\sigma \subset V$
Result: A grown erasure $\varepsilon'$ 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 $\mathcal{L} \neq \emptyset$ 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 $\varepsilon'$
```

Time complexity of the Union-Find decoder

3.1.4 Finding clusters

3.1.5 Bucket Cluster Sort

To further increase the error threshold for the Union-Find decoder from 9.2% to 9.9%, Nickerson implements weighted growth, where clusters are grown in increasing order based on their sizes [5]. 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 in implemented. From a review of the Union-Find decoder, an implementation of weighted growth is described using *Heapsort*, which has been incorrectly described as linear as its time complexity is also $\mathcal{O}(n\log(n))$ [9]. 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 $\mathcal{O}(\sqrt{n})$.

How to sort for weighted growth

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 of the cluster S 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 with size $S_A << S_B$, growth of cluster B will add $\sim k_B/2$ "incorrect" edges on average, whereas growth of cluster A will add $\sim k_A/2 << k_B/2$ edges as $k_A \propto S_A$ and $k_B \propto S_B$. However, if $S_A \simeq S_B$, the number of added "incorrect" edges for both clusters will also be similar, and it is the same when $S_A = S_B$. Thus clusters with the same size can be grown "simultaneously", as there is no benefit to grow one before another.

Lemma 3.1 *Weighted growth of cluster A with size $S_A << S_B$ ensures that a smaller amount of "incorrect" edges is added, compared with growth of cluster B.*

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. 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 the case of no merge event, clusters grown from even bucket i must be placed in odd bucket $i+1$, as it does not increase in size, and clusters grown from odd bucket j must be placed in even bucket $j+2k+1$ with $k \in \mathbb{N}_0$. Also in the case of a merge event of clusters A and B, the new cluster AB must be placed in a bucket $i_{AB} > i_A, i_{AB} > i_B$. Thus we can grow the buckets sequentially, and need not to worry about bucket that have been already "emptied".

Lemma 3.2 *Buckets can be grown sequentially after each other as new clusters will always be placed in a higher bucket than the current one.*

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 A and B during growth of B, cluster A has already been placed in a bucket, as it was still odd after its growth. But cluster A is now part of cluster AB and has even parity, and the entry of cluster A is faulty. To prevent growth of the *faulty entry*, we can check for the parity of the root cluster.

Furthermore, it is possible that another cluster C merges onto AB, such that the cluster ABC is odd again. Now, the faulty entry of cluster A passes the previous test. To solve this issue, we store an extra `bucket_number` at the root of a cluster. Whenever a cluster increases in size or merges to an odd parity cluster, we first update the `bucket_number` to the appropriate value and place it in its bucket. If the cluster merges to an even parity cluster, we update the `bucket_number` to `None`. Now, every time a cluster is popped from bucket i, we can just check weather the current bucket corresponds to the `bucket_number` of the root cluster.

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. As we are only placing odd

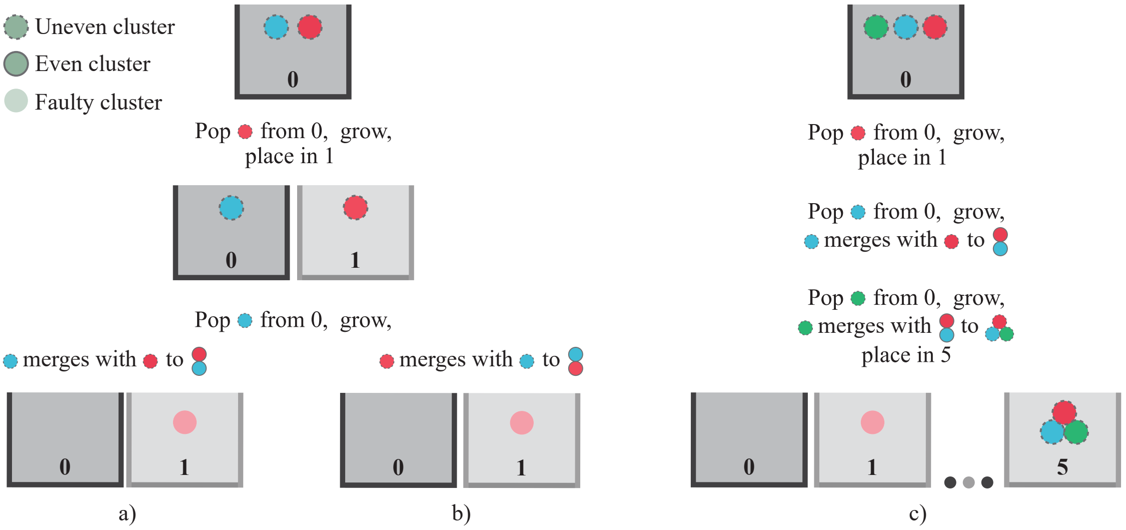

Uneven cluster
Even cluster
Faulty cluster
0
Pop ● from 0, grow, place in 1
0
1
Pop ● from 0, grow,
● merges with ● to ●
● merges with ● to ●
0
1
a)
0
1
b)
0
Pop ● from 0, grow, place in 1
Pop ● from 0, grow, ● merges with ● to ●
Pop ● from 0, grow, ● merges with ● to ● place in 5
0
1
5
c)

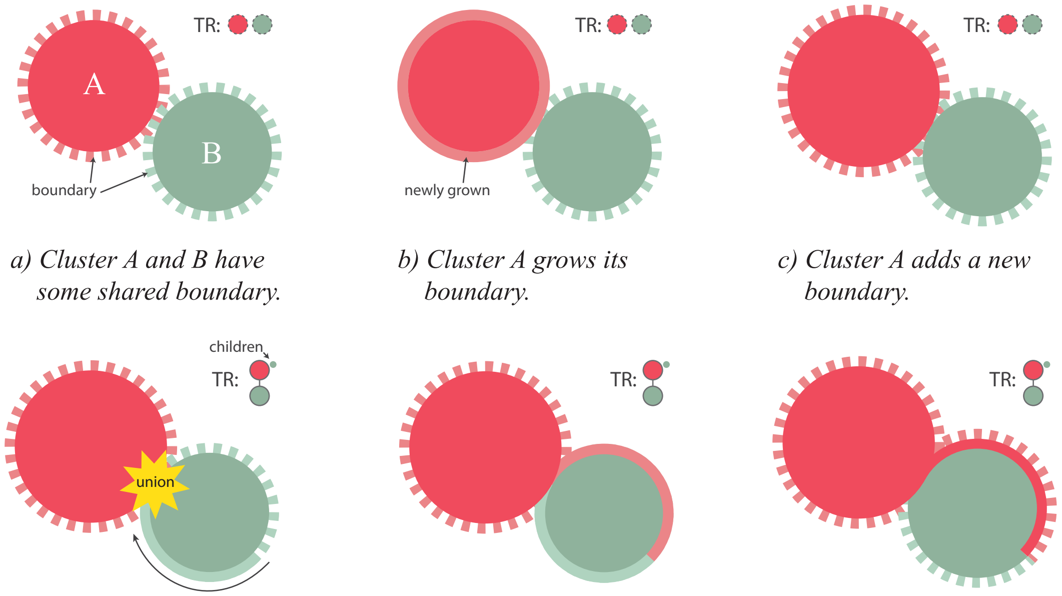
Figure 3.2: Faulty entries of clusters can occur in the buckets, a) cluster that should not be there due to a merge event. Situation a can be solved by checking the parity of the cluster. Checking the parity of the root cluster solves a) and b). Checking the bucket_number of the root cluster solves all.

parity clusters in buckets, and clusters of the same size grow ”simultaneously”, the largest odd parity cluster should only cover about half the lattice, while another odd parity cluster of the same size covers the other half. Any odd parity cluster that is larger than that should now grow as there should always be a smaller cluster available to grow instead. We can show that the largest odd parity cluster size is $S_M = L \times (\lfloor L/2 \rfloor - 1)$ with $L = \sqrt{n}$ on a square lattice. This results to $N_b = 2L \times (\lfloor L/2 \rfloor - 1)$ buckets. Any odd parity cluster with bucket number larger N_b shall be assigned `bucket_number=None` and will not be placed in a bucket, 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 bucket occurrence $i_{M,o}$. Whenever a bucket i has been emptied and $i = i_{M,o}$, we can break out of the bucket loop to skip the remainder of the buckets.

Complexity

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



a) Cluster A and B have some shared boundary.

b) Cluster A grows its boundary.

c) Cluster A adds a new boundary.

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

e) New vertices are added to root cluster A.

f) New boundary is also added to root cluster A.

Figure 3.3: 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.

preserved.

The Bucket Union-Find decoder

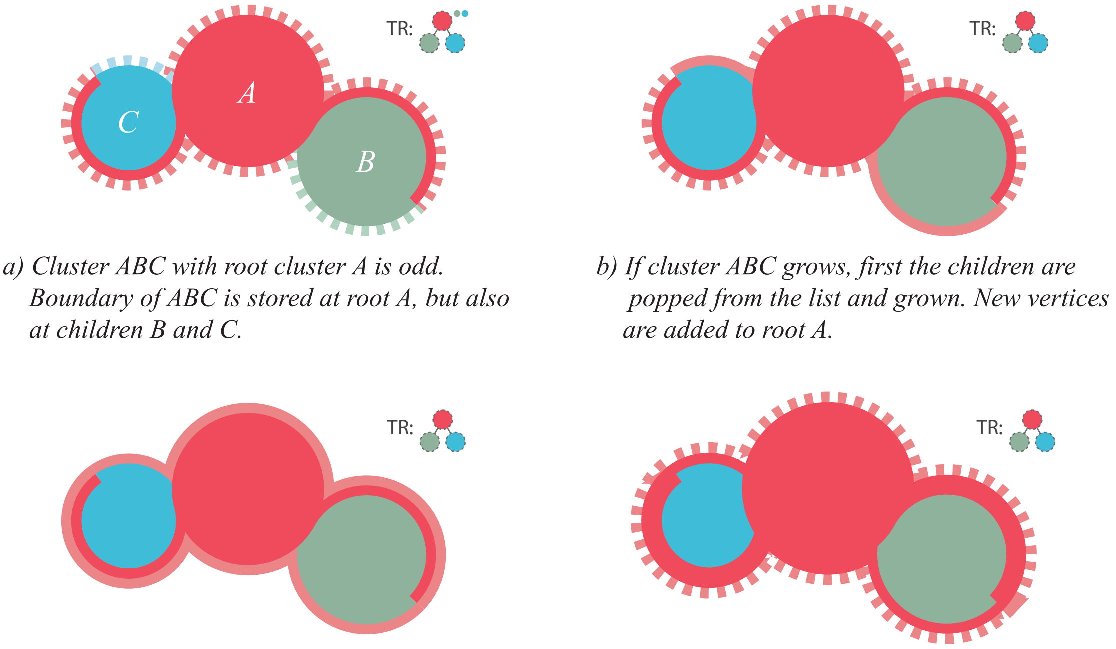
3.1.6 Object oriented approach

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 [9]. 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.

A new data structure

Parent-child method for merging boundary lists

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)



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

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.

Chapter 4

Threshold simulations

To test for the threshold Pauli error value, we simulate for a large number of samples at various lattice sizes for a range of Pauli error rates around $p = 0.1$. For the threshold, only Pauli X errors are considered, as Pauli Z errors will give the same result. For each lattice size and Pauli error rate, the samples will return a probability rate of successful decodings P_{succes}. We will then fit the data to the function ([10], equation 43):

$$\begin{aligned} P_{succes} &= A + Bx + Cx^2 + \begin{cases} D_{even} \cdot L^{-1/\mu_{even}} & \text{L even} \\ D_{odd} \cdot L^{-1/\mu_{odd}} & \text{L odd} \end{cases} \quad (4.1) \\ \text{with } x &= (p - p_{thres})L^{1/\nu} \quad (4.2) \end{aligned}$$

where all but P_{succes}, p and L are fitting parameters. Note that there are distinct values for D and μ for even and odd lattices. This is due to a discrepancy in the decoder threshold caused by a nonnegligible finite-size effect for even and odd lattices. Therefore, for each fit done on any dataset, only data from even or odd lattices will be selected. The fitting of the data will be done in Python using a least squared method.

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.*

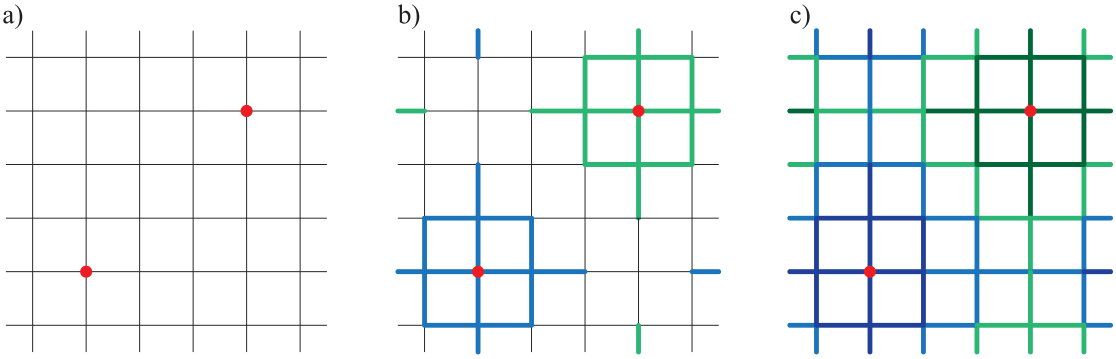


Figure A.1: If two single-generator clusters are placed a) maximally far from each other, b) a maximal number of growth iterations is needed. c) After $2\lfloor l/2 \rfloor$ iterations, the entire lattice is covered.

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.*

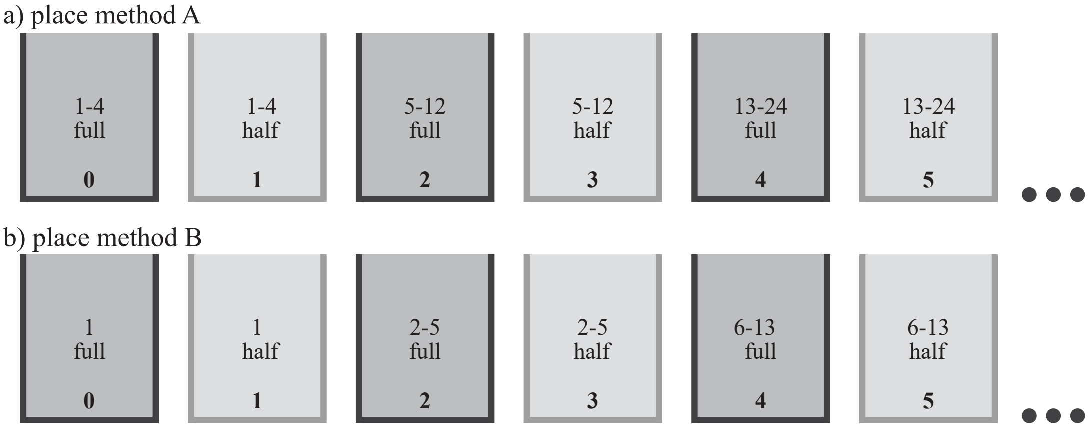


Figure A.2: The first six buckets for a) method A from definition A.1 and b) method B from definition A.2. Even buckets contains fully grown clusters, odd buckets contain half-grown clusters of the same size.

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.

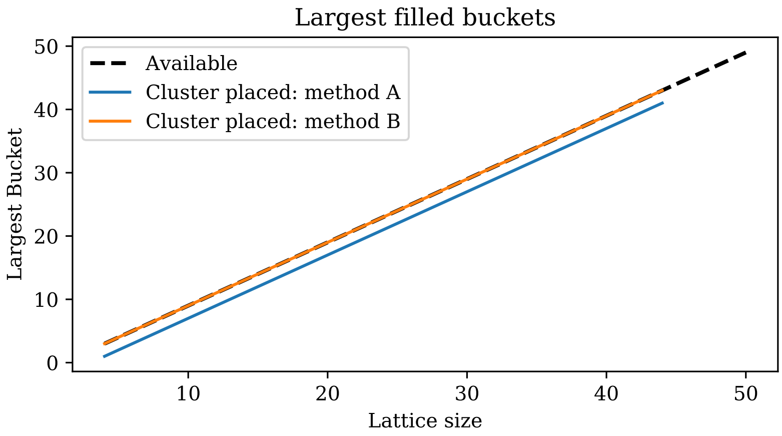


Figure A.3: The largest bucket that will be utilized for methods A and B (see fig. A.2). Method A does not utilize the largest two buckets.

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

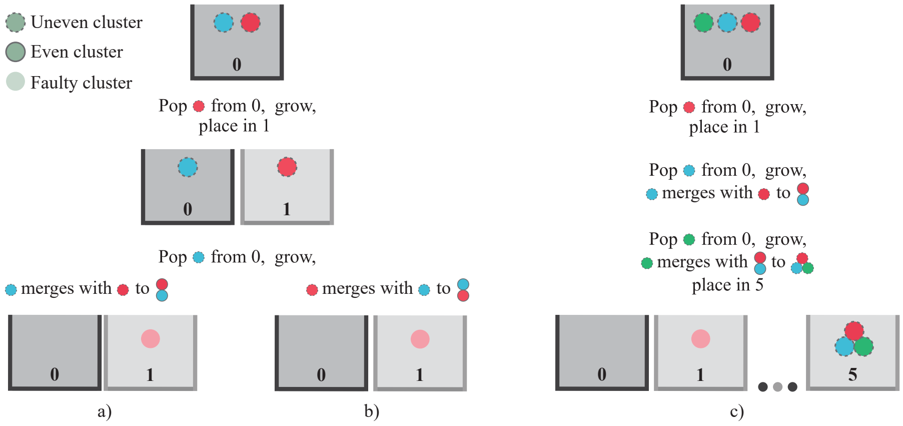

Uneven cluster
Even cluster
Faulty cluster
0
Pop ● from 0, grow, place in 1
0
1
Pop ● from 0, grow,
● merges with ● to
● merges with ● to
0
1
0
1
a)
b)
0
Pop ● from 0, grow, place in 1
Pop ● from 0, grow, ● merges with ● to
Pop ● from 0, grow, ● merges with to place in 5
0
1
5
c)

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] A. Y. Kitaev. Fault-tolerant quantum computation by anyons. *Annals of PHysics*, 303(1):2–30, January 2003.

[2] A. C. Doherty T.M. Stace, S. D. Barrett. Thresholds for topological codes in the presence of loss. *Physical Review Letters*, 102(20):200501, May 2009.

[3] N. H. Nickerson N. Delfosse. Almost-linear time decoding algorithm for topological codes. *Quantum Physics*, September 2017.

[4] S. D. Barrett T. M. Stace. Error correction and degeneracy in surface codes suffering loss. *Physical Review*, 81:022317, February 2010.

[5] G. Zemor N. Delfosse. Linear-time maximum likelihood decoding of surface codes over the quantum erasure channel. *Quantum Physics*, March 2017.

[6] N. Nickerson. *Practical fault-tolerant quantum computing.* PhD thesis, Imperial College London, December 2015.

[7] D. Browne. Lectures on topological codes and quantum computation. chapter 1-5. bit.do/topological.

[8] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery*, 22(2):215–225, April 1975.

[9] N. Leijenhorst. Quantum error correction, decoders for the toric code. Bachelor thesis, Delft University of Technology, July 2019. Applied Mathematics and Applied Physics BSc.

[10] J. Preskill C. Wang, J. Harrington. Confinement-higgs transition in a disordered gauge theory and the accuracy threshold for quantum memory. *Annals of PHysics*, (303):31–58, September 2003.