Fault-tolerant Error Correction of non-Abelian Anyons

Patrick Heim

03.11.2017

Abstract

In this thesis a new decoding algorithm (Weasel) was developed for the planar code and compared with other decoders (Bravyi Haah, ABCB and Expanding Diamonds). The algorithm's threshold in the qubit case is around 6%; for d=6 it is 11.5%. Error configurations that cause the decoder to fail were analyzed by applying Kruskal's algorithm to find the minimal spanning tree of errors across the 2D lattice. The scaling behaviour of these error configurations was investigated for the four decoders and the Weasel decoder was found to be competitive in that respect.

Contents

1 Introduction **3**

2 Preliminaries **4**
2.1 Surface Codes 4
2.1.1 Toric Codes 4
2.1.2 Planar Codes 5
2.2 Decoders 6
2.3 Anyon Density and Error Length Ratio 8
2.4 Kruskal's Algorithm 9

3 Implementation **10**
3.1 Error Generation 10
3.2 Clustering 10
3.2.1 Bravyi Haah and ABCB 11
3.2.2 Expanding Diamonds 12
3.2.3 Weasel 13
3.3 Kruskal's Algorithm 13

4 Results **15**
4.1 Thresholds 15
4.2 Kruskal Data 17

5 Conclusions **20**

1 Introduction

There are certain tasks that could be performed much faster on a quantum computer than on a classical one. One example of such a task is integer factorization which could be done using Shor's algorithm [1] on a quantum computer.

In this thesis we focus on topological quantum computation by anyons. Anyons are quasi-particles that live in two-dimensional space. Anyons with non-commutative exchange statistics are called non-Abelian anyons. Since these exchange statistics are only dependant on the homological properties of the paths the anyons take, they can be used for topological quantum computation as proposed in [2].

In order to do quantum computation we must first encode our qubits (or qudits) in a suitable quantum system. An introduction to surface codes is presented in detail in [3]. Quantum systems are much more susceptible to noise than classical systems. So the defects resulting from noise need to be identified and removed continuously. 'Decoding algorithms' or simply 'decoders' are used to find the necessary operations to correct such defects. Three of the decoders that are investigated in this thesis have been discussed in [4]. According to the threshold theorem [5], quantum computation can be performed for an arbitrary amount of time and with an arbitrarily high reliability, given that the error rates are below a certain threshold. This threshold is specific to each decoder.

This thesis is structured as follows: First the preliminaries are covered in section 2. This includes a brief explanation of toric and planar codes as well as an introduction to the four decoders used in this thesis. Kruskal's algorithm and the motivation for using it are also presented. A detailed explanation of our implementations is given in section 3. Finally, the results and conclusions are discussed in sections 4 and 5, respectively.

2 Preliminaries

2.1 Surface Codes

In the following sections we will take a look at the error-correcting codes introduced in [2]. Toric codes and planar codes can be generalized to a class of quantum codes called 'surface codes'. Each code of this class is then associated with a tessellation of a two-dimensional surface.

A more detailed discussion of these codes can be found in [3].

2.1.1 Toric Codes

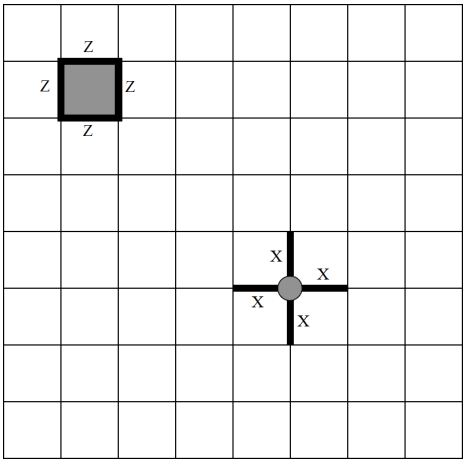


Figure 1: Check operators of the toric code.[3]

Surface codes are a type of 'stabilizer code' [6]. A quantum state is stabilized by an operator if it is an eigenstate with eigenvalue one. A binary stabilizer code is characterized by the eigenspace with eigenvalue one of a set of mutually commuting check operators, where each check operator is a Pauli operator. We use the matrices

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$

$$Y = \begin{pmatrix} 0, -i \\ i, 0 \end{pmatrix}, Z = \begin{pmatrix} 1, 0 \\ 0, -1 \end{pmatrix},$$

where I is the 2×2 identity and X, Y and Z are the Pauli matrices. The action of a Pauli operator on n qubits corresponds to one of the 2^{2n} tensor product operators

$$\{I, X, Y, Z\}^{\otimes n}.$$

For a stabilizer code of higher dimension, the appropriate generalized Pauli and identy operators are used.

An $L \times L$ lattice on a torus has $2L^2$ links which correspond to $2L^2$ qubits. We distinguish between two types of check operators: the check operators on a site that act on the four links that meet at the site, and the check operators in an elementary cell (or 'plaquette') that act on the four links contained in the cell. The former are called 'site operators', the latter are referred to as 'plaquette operators'. The action of a site operator is given by X operators acting on the four qubits that meet at the site. Plaquette operators act as Z operators on the four qubits contained in the plaquette. These check operators are mutually commuting.

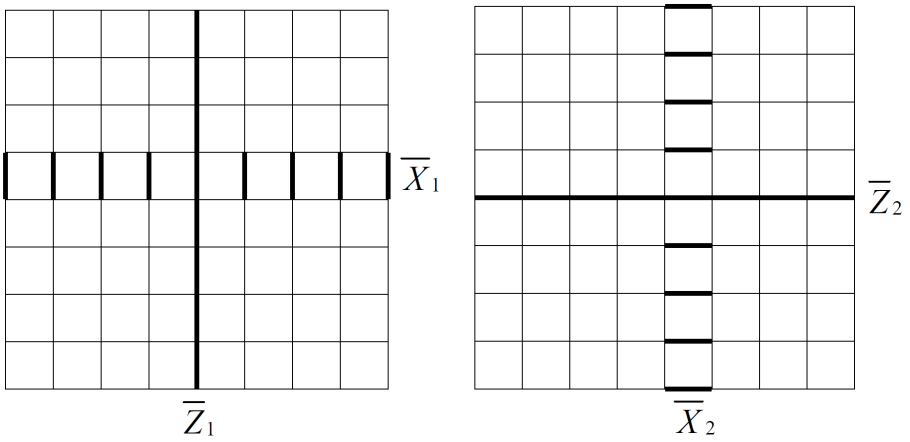


Figure 2: Basic operators acting on the two encoded qubits in the toric code.[3]

Due to the periodic boundary conditions of the toric code, each check operator can be expressed as a product of $L^2 - 1$ other such operators. There are $2 \cdot (L^2 - 1)$ independent check operators. This means that 2 qubits can be encoded using the toric code.

Operations acting on the logical qubits correspond to the homologically non-trivial cycles around the torus depicted in Fig. 2. Cycles are non-trivial if they are not contractable. Measuring all the check operators on the torus yields a 'syndrome' that can be used to identify errors. Non-trivial syndrome elements can be thought of as particles (anyons). In order to correct an error chain, the particles at the two endpoints must be brought together to annihilate. The correction will succeed if the correcting operators form a homologically trivial cycle with the error chain (i.e. they do not form a loop all the way around the torus), otherwise it will result in a logical error.

2.1.2 Planar Codes

Planar codes differ from toric codes in their boundary conditions. The lack of periodic boundary conditions in planar codes results in different check operators

at the edges of the surface. Check operators in the interior of the surface act on 4 qubits, exactly like the check operators in the toric code. Check operators at the boundary only act on 3 qubits.

We distinguish the top and bottom edges from the left and right edges. The top and bottom edges are called 'plaquette edges' (or 'rough edges') because the check operators along them are 3 qubit plaquette operators. Analogously, the left and right edges are referred to as 'site edges' (or 'smooth edges') because the corresponding check operators are 3 qubit site operators. Logical operations on the encoded qubits are cycles relative to these edges. A logical Z is caused by a cycle relative to the rough edges and a logical X results from a cycle relative to the smooth edges. For instance, a cycle relative to the smooth edges is a path from the left to the right edge of the lattice.

While defects in the toric code always appear in pairs, they can arise singly in the planar code. This happens if they are caused by an error chain that ends at an edge. Single site defects arise from error chains that end at a rough edge and single plaquette defects arise from dual error chains that end at a smooth edge.

Unlike in the toric code, only one single qubit can be encoded in this version of the planar code. The number of encoded qubits can be increased by punching holes in the surface, thus adding more possible homologically non-trivial cycles.

Both toric and planar codes can be generalized to higher dimensions with qudits instead of qubits. This results in encoded qudits and $\mathbb{Z}_d$ anyons on the lattice instead of qubits.

2.2 Decoders

In order to protect our encoded qudits from the effects of noise, we need to identify and correct defects as they occur. This process is called decoding.

There exist several different classes of decoders which have different advantages and disadvantages. The decoders covered in this thesis are all 'hard-decision renormalization group' or 'HDRG' decoders according to the definition given in [4]. These decoders function as follows: First, every non-trivial element of the syndrome forms a separate cluster. Then, the following process is repeated until the syndrome is empty:

1. Combine existing clusters to form at least one new cluster.
2. Check every new cluster for neutrality.
3. Remove neutral clusters.

Different HDRG decoders are distinguished by the methods used to do the clustering. We look at 3 decoders in particular: Bravyi-Haah [7], ABCB [8]and Expanding Diamonds decoder [9].

BH and ABCB have a similar clustering process, but they use a different metric. BH uses the Chebyshev distance (L_∞ metric) and ABCB uses the Manhattan distance (L_1 metric). The physical distance d_{jk} between non-trivial

syndrome elements j and k is defined according to the corresponding metric. The search distance $D(n)$ for the nth iteration of the algorithm is $D(n) = 2^n$ for BH and $D(n) = n + 1$ for ABCB. First, set $n = 0$ and form a graph consisting of all non-trivial syndrome elements. They correspond to the vertices of the graph. Initially, the graph consists only of vertices without any edges. Repeat the following algorithm until there are no vertices left:

1. Add an edge between each pair of vertices for which $d_{jk} \leq D(n)$.
2. Clusters are vertices connected by edges. Check all clusters for neutrality. Remove all neutral clusters.
3. If there are any vertices left, increase n by 1.

The Expanding Diamonds decoder uses the Manhattan distance for physical distances d_{jk} and $D(n) = n + 1$ for the search distance. Like for BH and ABCB, every vertex forms a separate cluster initially and $n = 0$. Repeat until there are no clusters left:

1. Number the clusters from 1 to N_0 (the number of non-trivial syndrome elements) from left to right and top to bottom. Loop through the clusters and check for every cluster j if there exists a cluster $k > j$ for which $d_{jk} < D(n)$. If so, merge the clusters and check for neutrality. If the new cluster is neutral, remove it from the syndrome immediately.
2. Label the new clusters from 1 to N_{n+1} (the number of remaining clusters after the nth iteration). Define the distance d_{jk} between clusters j and k to be the minimum distance between an anyon of one to an anyon of the other.
3. if $N_{n+1} > 0$ increase n by 1.

These decoders can fail if they incorrectly cluster anyons that were not created together. Consider the situation shown in Fig. 3. In this case, the two anyons in the middle are wrongly clustered which results in the wrong clustering of the remaining two anyons. But if we update the distance between the anyon on the left and the anyon on the right to account for the possibility that the neutral cluster we removed was part of an error chain, the logical error can be averted. Whenever a neutral cluster c is removed, we can implement 'shortcuts' by updating the distances between other clusters: $d_{jk} = min(d_{jk}, d_{jc} + d_{ck})$.

During the work of this thesis, we developed a decoding algorithm that uses some form of shortcuts. Instead of gradually increasing a search distance, we look for nearest neighbours for each cluster. We call it the Weasel algorithm. It works as follows: Like for the 3 decoders described above, every anyon is considered to form its own separate cluster at first. The metric used is the Manhattan distance.

1. Number the clusters from 1 to N_0 (number of non-neutral clusters) from left to right and top to bottom. Loop through them in that order. For

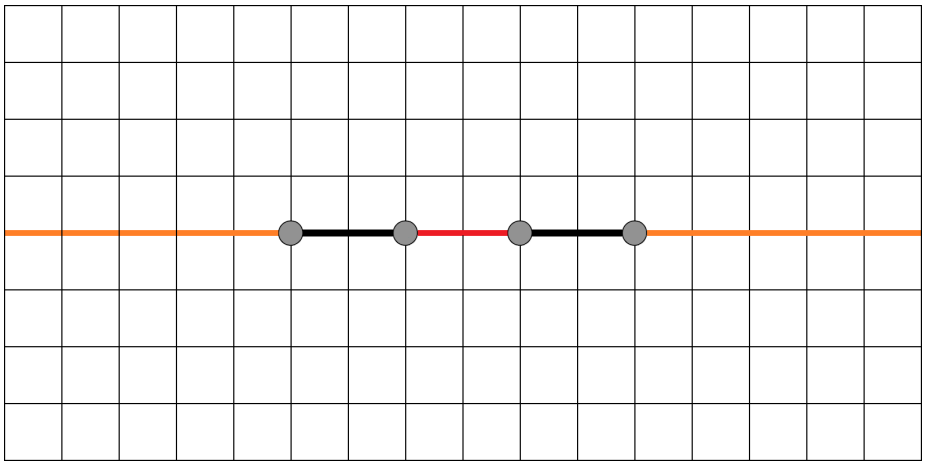


Figure 3: Two incorrectly clustered pairs of anyons. The black lines denote the actual anyon pairs, the red line highlights the incorrect clustering and the orange lines show the clustering of the remaining two anyons. The resulting logical error can be prevented if 'shortcuts' are used.

each non-neutral cluster j, loop over the other clusters and find the cluster that is j's nearest neighbour. If multiple nearest neighbours are found, choose one at random. Merge the cluster with its nearest neighbour. If an edge is closer than any nearest neighbour, merge cluster j with the edge and consider it to be neutral. Update the number of non-neutral clusters.

2. If non-neutral clusters remain, repeat 1. Otherwise remove all neutral clusters.

Note here that neutral clusters are not removed until the end. While neutral clusters do not look for nearest neighbours anymore, they can still be nearest neighbours to non-neutral clusters. In most cases where the decoding results in a logical error, we end up with a bunch of small neutral clusters and two large non-neutral clusters that then merge with different edges (which causes the logical error).

2.3 Anyon Density and Error Length Ratio

Logical errors can occur if a decoder fails to cluster defects correctly, which may result in homologically non-trivial cycles. In the case of the planar code this happens when a single defect is clustered with the wrong edge or if the two anyons of a pair in the bulk of the lattice are incorrectly clustered with opposite edges. In either case we can associate the logical error with a path from one edge of the lattice to the opposite edge. The decoding fails if it completes the path instead of removing it.

Since our goal is to compare how well different decoders perform, we need to find answers to question such as:

- What is the lowest number of errors that cause the decoder to fail?
- What does the resulting non-trivial cycle (error path) look like?
- How are anyons distributed along the error path?

In order to answer these questions, we must first find the error path for a given error configuration that causes the decoder to fail. If the probability for errors to happen is much lower than the threshold of the decoder, occurring logical errors are more likely to be caused by the lowest number of errors. But even for low error rates there will usually be some errors that do not contribute to the logical error (because the decoder clusters them correctly). We can use a tool from graph theory to filter out such errors. This tool is Kruskal's algorithm.

2.4 Kruskal's Algorithm

First we form a graph consisting of every anyon that has contributed to the logical error in some way. This can include some anyons that have not contributed to the logical error at all, as long as there are sufficiently few of them. Then we use Kruskal's algorithm to find the shortest spanning tree of this graph:

Kruskal's algorithm for a given graph G:

> "Perform the following step as many times as possible: Among the edges of G not yet chosen, choose the shortest edge which does not form any loops with those edges already chosen. Clearly the set of edges eventually chosen must form a spanning tree of G, and in fact it forms a shortest spanning tree." [10]

This shortest spanning tree connects the left edge of the lattice with the right one. After pruning off all the branches, we are left with a shortest path across the lattice that leads to a logical error.

The path gives us some interesting quantities that can be used to answer the questions in section 2.3: Total length of the path (error length), anyon density along the path, the ratio of the distances between correctly clustered anyons to the total length of the path (error length ratio), and the minimum number of errors required for the decoder to fail (minimum error number).

3 Implementation

The code for all of our implementations was written in C++. The four decoders were implemented for a version of the planar code where we only considered site defects and ignored plaquette defects. This means that only the rough edges are relevant. The way the lattice was set up in our code it is rotated from the lattice described in section 2.1.2 by 90°. So the left and the right edge are the rough edges and the top and bottom edge are the smooth edges. Since we are only looking at site defects, the smooth edges only serve as boundaries and do not give rise to single defects.

3.1 Error Generation

Throughout this thesis, errors are generated and stored in the same fashion for every decoder. The 2D square lattice of size L is stored in a 2D array with indices y and x, where y and x denote the row and column respectively. Top left corner corresponds to the coordinates $(0, 0)$, bottom right corner corresponds to $(L-1, L-1)$. The two edges are stored in a separate 1D array. All array entries are initially 0.

Loop over y and x, which corresponds to looping over the lattice from left to right, top to bottom. For every position 2 errors can happen at most: a horizontal error and a vertical error. With probability p create an anyon with a random charge $0 < c < d$ at position (y,x) (if there is already an anyon there, fuse them together) and another anyon with charge $d - c$ at position (y,x+1) (again, if there is already an anyon there, fuse them). Do the same for vertical errors, the only difference being that the second anyon will be created at position (y+1,x). The last row cannot have any vertical errors. Horizontal errors in the last column add $d - c$ to the charge of the right edge. Finally, for every y with probability p increase the charge of the left edge by $0 < c < d$ and create an anyon with charge d-c at position (y,0).

In order to keep track of which anyon pairs were created together, we used a $2L^2$ by 4 array which stores the coordinates of both anyons of a pair. We defined the coordinates of the edges to be (-1,-1) for the left edge and (-1,L) for the right edge.

3.2 Clustering

After some errors have been generated, it is the decoders job to try and correct them. The decoders covered in this thesis do so by clustering non-trivial syndrome elements and eventually removing neutral clusters. The clustering process is what distinguishes these decoders.

Before anything can be clustered, we need to find all non-trivial syndrome elements (anyons). This is done by looping through our 2D lattice array and looking for non-zero entries. We use two other arrays with sizes N by 2 and N by 4 (where N is the total anyon number) to store information about anyons. In order to define the array sizes, either do an additional loop first to determine

the total anyon number or use a dynamic array and increase its size as anyons are found. Another possibility is the method we used which is to use a static array of size L^2 instead of N. This method might cause problems for large system sizes when it requires too much memory while storing a mostly empty array.

The N by 2 array stores the coordinates of the anyons and the N by 4 array stores information that will be relevant for clustering. It contains the following information:

1. Number of the first anyon in this anyon's cluster. This is initiated to be the number of the current anyon, meaning that every anyon initially forms its own cluster.

2. Number of the next anyon in this anyon's cluster. Clusters will be stored as linked anyon arrays (architecture of linked lists). -1: this is the last anyon in its cluster.

3. Total anyonic charge of this anyon's cluster. This will be set to 0 when the cluster of this anyon is removed from the syndrome.

4. Is an edge part of this anyon's cluster? 0: no, 1: left edge, 2: right edge, 3: both edges (logical error happened!).

Additionally, an N by N array was used to store the distances between all anyon pairs. The diagonal entries contain the distance from an anyon to the left edge. We named our arrays as follows:

- $Syndrome[L][L]$: 2D lattice array
- $Coords[N][2]$: anyon coordinates
- $Anyons[N][4]$: array used for clustering of anyons
- $Distance[N][N]$: distances between all anyon pairs

Now, let us first look at BH and ABCB as those are the most intuitive.

3.2.1 Bravyi Haah and ABCB

For BH the Chebyshev distance is used:

$$Distance[j][k] = max(abs(Coords[j][0] - Coords[k][0]), abs(Coords[j][1] - Coords[k][1]))$$

So we loop over all pairs of anyons, calculate these distances and store them in the $Distance$ array. The diagonal simply stores the distance of an anyon to the left edge which is given by $Coords[j][1] + 1$. Define the search distance $D(n)$ which is initially 1. Also define the cluster number C to be equal to the anyon number N. While there are non-neutral clusters left, i.e. $C > 0$, loop over all remaining anyon pairs j and k with $Anyons[j][2] \neq 0$ and $Anyons[k][2] \neq 0$ and cluster the ones whose distance is smaller than or equal to the search distance.

Only check this for pairs of anyons that are not part of the same cluster yet. This is done by comparing the first entries of the corresponding *Anyons* array which denote the first anyon in the clusters. If both anyons have the same first anyon in the cluster, then clearly they are already part of the same cluster and no distance check is necessary. Also check for every anyon if an edge is closer than $D(n)$. If so, update $Anyons[j][3]$ accordingly.

When a suitable anyon pair is found, their clusters are merged. This is done by appending the linked list of the second cluster to the linked list of the first cluster, and updating the first anyon of the second cluster (set it to be equal to the first anyon of the first cluster). After the clusters are merged, the cluster number is reduced by one.

Once the loop has completed, check all clusters for neutrality. For every anyon j with $Anyons[j][2] \neq 0$, go through the linked list of its cluster and add up the charges of its cluster's anyons modulo d. If the cluster has an edge connection, update the charge of that edge by adding the total charge of the cluster modulo d and set $Anyons[j][2] = 0$ for all anyons in the cluster (it is considered to be neutral and has been removed). If the cluster does not have an edge connection and if the total charge is 0, remove the cluster in the same manner by setting $Anyons[k][2] = 0$ for every anyon in the cluster.

This part of the code potentially does many redundant operations: e.g. checking the same non-neutral cluster for neutrality for every single anyon that is part of it. This might be a cause for long computation times and could be avoided by only looping over existing clusters rather than over all anyons. This would likely require an additional array and a method for properly numbering the clusters.

After all neutral clusters have been removed, set $D(n) = 2 \times D(n)$ and start over if there are non-neutral clusters left (if cluster number > 0).

ABCB uses the Manhattan distance:

$$Distance[j][k] = abs(Coords[j][0] - Coords[k][0]) + abs(Coords[j][1] - Coords[k][1])$$

The clustering is done in exactly the same manner, the only difference being that at the end of each iteration, the search distance $D(n)$ is only increased by 1.

Shortcuts are implemented as follows: Whenever a neutral cluster c is removed, loop over all remaining anyon pairs. For each pair of anyons j and k calculate their distance to the cluster c. This is defined as a the minimum distance from an anyon in c to j and k. If the sum of those two distances is lower than the distance between j and k, update $Distance[j][k] = d_{jc} + d_{ck}$.

3.2.2 Expanding Diamonds

Like ABCB, Expanding Diamonds uses the Manhattan distance and the search distance $D(n)$ is incremented by 1 after every iteration. The main difference here is that neutral clusters are removed as they are found rather than at the end.

Our implementation differs slightly from the definition given in section 2.2. First, each anyon forms its own cluster. Then we loop through them from left to right, top to bottom, and for each cluster we check if one of the following clusters is within search distance. If such a cluster is found, we merge the two. However, instead of just merging them and then checking them for neutrality, we fuse the first cluster with the second one and then check if the resulting cluster is neutral. Due to this, no cluster will ever contain more than one anyon but the charges of the anyons will be changed. This method is directly applicable to non-abelian anyons, since they require us to fuse them in order to determine whether their clusters are neutral.

3.2.3 Weasel

Unlike the 3 decoders described above the Weasel decoder does not use an increasing search distance. Instead it looks for nearest neighbours for each cluster. Neutral clusters are not removed until the end but we keep track of how many non-neutral clusters remain. Only non-neutral clusters look for nearest neighbours and they can merge with neutral clusters. By keeping the neutral clusters around and allowing non-neutral clusters to merge with them, we have essentially implemented a form of shortcuts.

Loop over all anyons. For every anyon find the nearest neighbours of its cluster. This is done by going through the corresponding linked list in the *Anyons* array, starting with the first anyon of the cluster, and checking for each anyon the distance to all anyons that are not part of the cluster. Keep track of the anyons tied for the shortest distance. When the end of the linked list has been reached, choose one of the nearest neighbours at random and merge with its cluster. If the chosen nearest neighbour is an edge, merge the cluster with it and update the edge charge accordingly. Consider the cluster to be neutral in that case. Merging is done using the same process as described for BH and ABCB. Once this process has been completed for the final anyon, check if the number of non-neutral clusters is greater than 0. If it is, repeat the process.

3.3 Kruskal's Algorithm

The input for the Kruskal algorithm is a 2D lattice with only the anyons that contributed to a logical error. The algorithm first finds all anyons and stores their coordinates similarly to the way we stored them during the clustering process. A forest is defined where every anyon initially corresponds to a tree, similarly to how each anyon formed its own cluster before clustering. We make an array for the edges which contains the following information:

1. Anyon at one end of the edge
2. Anyon at the other end of the edge
3. Edge length
4. has this edge already been used? 0:no, 1:yes

Keep track of the number of trees and perform the following algorithm until there is only one tree left: Find the shortest edge that connects two different trees (to avoid loops) and add it to the graph (decrease tree number by 1).

In order to check if an edge connects two different trees, check if the anyons at each end belong to the same tree. The same linked list array structure that was used for clustering can be used to store the relevant information for the different trees.

Once the algorithm has found the minimal spanning tree, we need to prune of the branches since we are only interested in the path from one edge of the lattice to the other. To do this we first set the two lattice edges as fixed endpoints. Then we loop over the anyons and remove the edges of every anyon that only has one edge. We repeat this step as long as at least one edge has been removed. Eventually, all branches will have been removed and we're left with the shortest path across the lattice that caused a logical error.

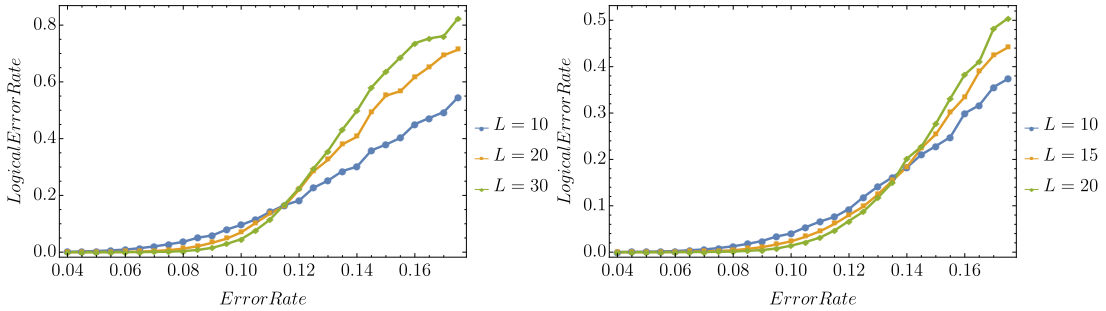
The quality of data produced by this algorithm strongly depends on the quality of the input. If we just use the original syndrome as input, the resulting shortest path across might be different from the actual path that caused the logical error. Therefore, it is important that we only use anyons that have actually contributed to the logical error. This is relatively simple for all decoders when shortcuts are not used. Simply use all anyons that were clustered incorrectly. If shortcuts are used, however, correctly clustered anyons that form neutral clusters can produce shortcuts that may cause other anyons to be clustered incorrectly. Not including the correct neutral cluster in the input would likely lead to a shortest path that has nothing to do with the logical error and any conclusions drawn from that data are bound to be flawed.

During the work of this thesis we have not yet found a solution to this problem and were unable to produce any meaningful data on the error configurations that cause decoders that use shortcuts to fail (other than the Weasel decoder).

4 Results

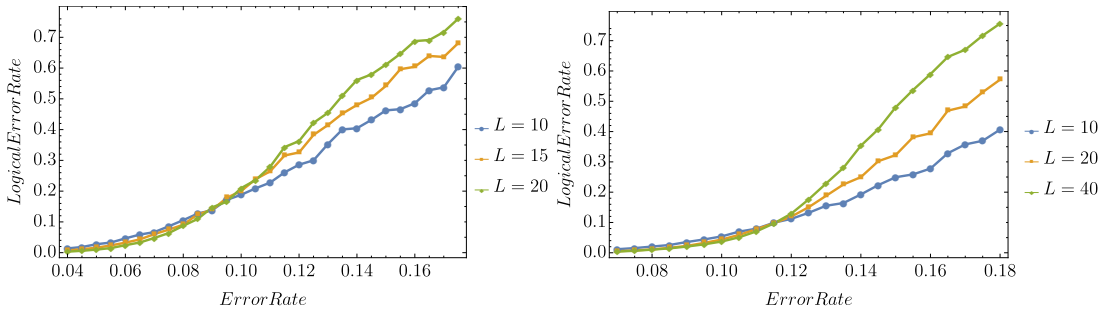
4.1 Thresholds

For $\mathbb{Z}_6$ anyons the Weasel decoder has a threshold of around 11.5%. This is roughly the same threshold we get for our implementation of the Bravyi Haah decoder when shortcuts are used. Without shortcuts the threshold for BH is around 11%. Our implementation of ABCB has a threshold of about 13.5%. Expanding Diamonds achieves a threshold of only 9.5%.



(a) Bravyi Haah decoder with shortcuts.

(b) ABCB decoder with shortcuts.



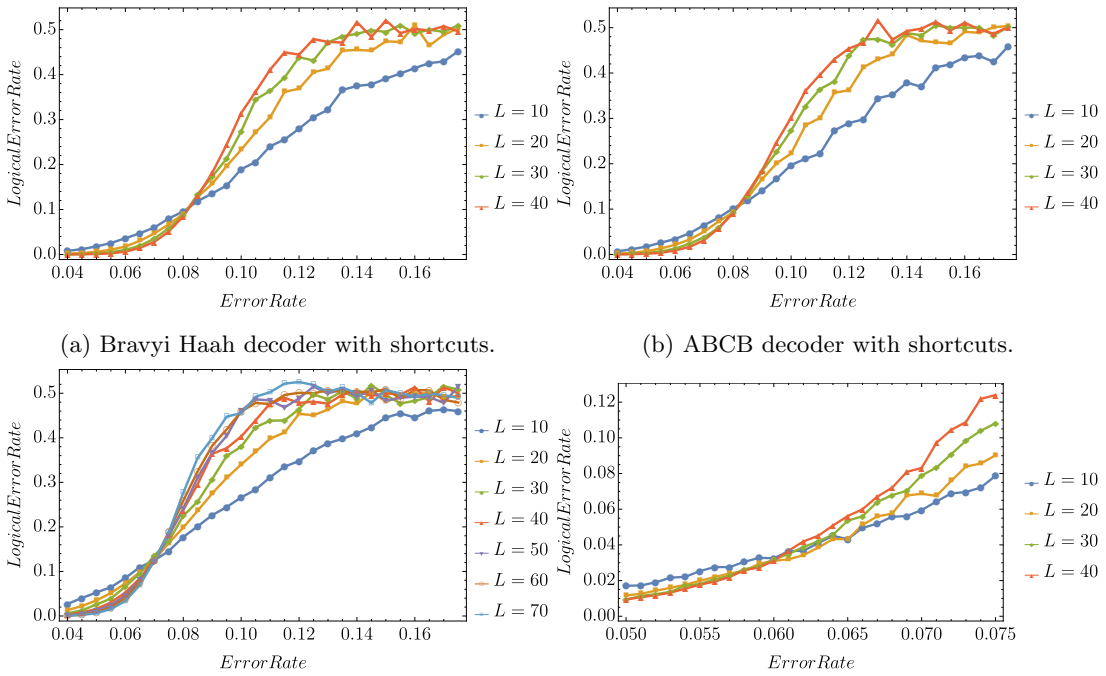
(c) Expanding Diamonds decoder with shortcuts.

(d) Weasel Decoder.

Figure 4: Threshold plots of different decoders for $\mathbb{Z}_6$ anyons.

In order to verify that we implemented the Bravyi Haah, ABCB and Expanding Diamonds decoders correctly, we gathered some threshold data for the qubit case (d=2) and compared that to already existing data. Bravyi and Haah obtained a threshold of 6.7% in [7] for their decoder. A higher threshold of 8.4% was achieved for ABCB in [8]. Expanding Diamonds' threshold lies between those two at around 7.25% - 7.5% [9]. In comparison, the Weasel decoder only reaches a threshold of around 6% in the qubit case. Weasel appears to perform better when anyons of higher dimensions are used. Our implementation of ED scores slightly lower at around 7%, which is likely due to the simplifications we made. For ABCB we obtain roughly the same threshold as in [8].

Our Implementation of BH achieves a threshold between 8% and 8.5% which



(a) Bravyi Haah decoder with shortcuts.

(b) ABCB decoder with shortcuts.

(c) Expanding Diamonds decoder with shortcuts.

(d) Weasel Decoder.

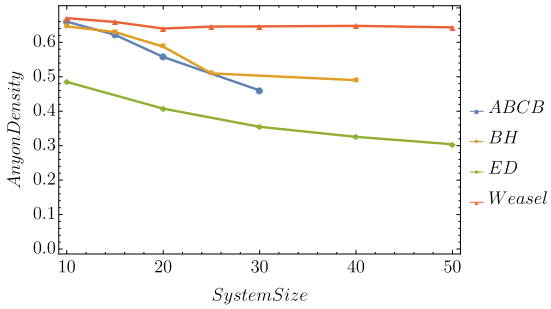
Figure 5: Threshold plots of different decoders for $\mathbb{Z}_2$ anyons.

is far higher than it should be. This might have several causes. We used the planar code, whereas Bravyi and Haah used the toric code. In the bulk the same results are expected for those two codes but they have different boundary conditions. Since we used relatively small system sizes (10 - 40), the effects of the different boundary conditions become more relevant. In the planar code the largest possible distance from any anyon to one of the two edges is $L/2$. This means that the decoding will be finished at the very latest when the search distance $D(n)$ is greater than $L/2$. For small error probabilities we expect most anyons that were created by the same error to be close to each other. So it is more likely that the decoding will be done long before $D(n) \geq L/2$. For small system sizes that only require 3 or 4 iterations, the search distance increases almost linearly, making BH very similar to ABCB. Consider the first 3 iterations: The search distances are 1, 2 and 4. In this case we can expect BH to give roughly the same results as ABCB.

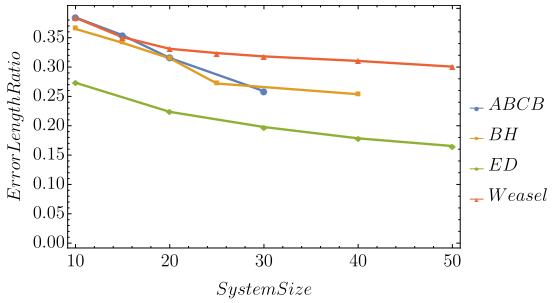
An alternative and possibly more likely explanation for the high BH threshold would be that there is a bug in our code. However, this would then beg the question why our implementation of ABCB gets good results, since the only differences between BH and ABCB are the definition of $D(n)$ and the metric, both of which were seemingly implemented correctly.

4.2 Kruskal Data

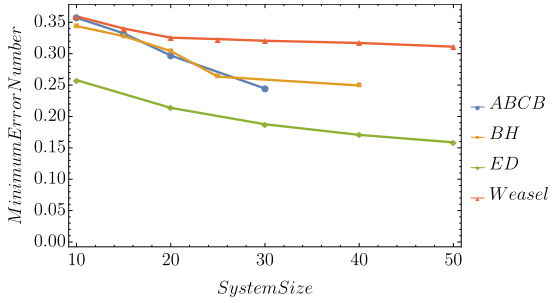
We are interested in the lowest number of errors that still result in a logical error. So Ideally we would use almost infinitesimally small error rates, since then logical errors are almost guaranteed to be caused by a minimal number of errors. In practise, however, this would lead to abysmally long computation times. Instead we chose some error rate that is lower than the corresponding decoder's threshold but still produces logical errors at a reasonable rate. We chose the error rate p=5% for Bravyi Haah, ABCB and the Weasel decoder and p=4% for Expanding Diamonds. The shortest possible path across the lattice



(a) Average anyon density.



(b) Average error length ratio.

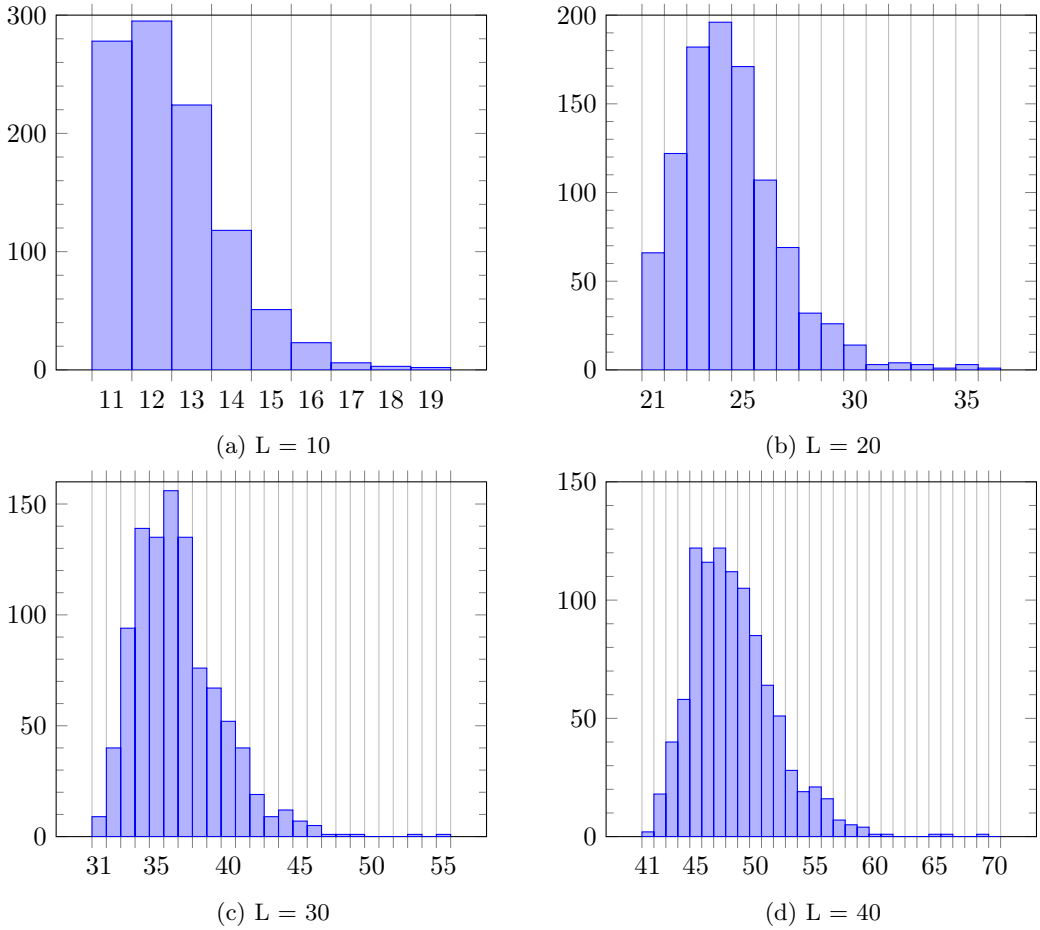


(c) Minimum number of errors required for a logical error.

Figure 6: The plots show three interesting quantities obtained by application of Kruskal's algorithm. Here, $\mathbb{Z}_6$ anyons were used. The error rate was fixed at 5% for Bravyi-Haah, ABCB and Weasel and 4% for Expanding Diamonds.

is a straight line from the left edge to the right edge with a length of $L + 1$. For a fixed error rate this path occurs most frequently for lower system sizes. For larger systems the path tends to be curved as excitations become less likely to lie on a straight line. The average error length increases faster than linearly with the system size for all 4 decoders. This can probably be reduced to linear scaling by using lower error rates p. The occurrences of the straight path across the lattice for ED can be seen in Fig. 7. For sufficiently low error rates the

peak of the histograms is expected to be at the minimum length even for larger systems. This would, however, result in a substantial increase in computation time as logical errors become much less frequent.



(a) L = 10

(b) L = 20

(c) L = 30

(d) L = 40

Figure 7: Histograms showing the distribution of the total length of the error paths for the Expanding Diamonds decoder for an error rate p=0.04. The peak for L=10 is almost at the minimum length which corresponds to a straight line across the lattice. For larger systems the peak starts wandering to the right.

Our data implies that the average anyon density for the Weasel decoder is constant at 0.65 regardless of system size. For Bravyi Haah the decrease in anyon density slows down drastically at around 0.5 whereas for ABCB it shows no signs of slowing down at a system size of 30. More data for larger systems might be necessary to determine the scaling behaviour for ABCB. The decrease in anyon density for Expanding Diamonds begins to slow down by system size 50.

The average error length ratio seems to converge at 0.3 for the Weasel decoder. Combining the average error length ratio and the anyon density, we get a lower bound for the minimum number of errors required to cause a logical error. The anyon density tells us how many anyons there are on the error path. The error length ratio is the ratio of the distances between anyons in the same cluster to the total length of the error path.

Consider two anyons that belong to the same cluster and are a distance 3 apart. At least 3 errors are necessary to create this configuration. One single error creates 2 anyons, but they might not both be on the error path. Since we are interested in the lower bound, we assume that all anyons created by errors are on the path. So the minimum number of errors required to create a given number of anyons, is the number of anyons / 2. The error length ratio gives us information about the distances between the anyons which are the result of more errors. We add up the error length ratio and the anyon density / 2 and divide this term by 2 in order to account for double counting of errors. This gives us a lower bound estimate for the minimum error number. To be precise: it gives us the minimum number of errors over the system size.

For the Weasel decoder the minimum error number appears to scale linearly with the system size with $0.3 \times L$. If it retains this scaling behaviour for larger systems, this could imply that it becomes more resistant to noise than other decoders. The minimum error numbers of the other decoders are expected to decrease further as the system size increases.

5 Conclusions

The Weasel decoder, while not achieving the highest threshold, can definitely compete with other decoders, especially for anyons of higher dimension. It should be easily applicable to non-abelian anyons as well.

Finding and analyzing the minimal spanning error trees of decoders can provide useful information about their advantages and disadvantages. We used Kruskal's algorithm to find the minimal spanning trees, but other such algorithms (e.g. Prim algorithm) could be used as well. It might be useful to compare these algorithms and to determine which one provides the most accurate error path. The quality of the resulting spanning tree depends strongly on the input. A way to filter out anyons that don't contribute to the logical error needs to be found in order to improve our code. Furthermore, a reliable method to include neutral clusters that created shortcuts causing a logical error is still needed to make the algorithm usable for decoders with shortcuts.

An improvement can be made in the part of our code that evaluates the shortest error path. We only keep track of the error length ratio and the anyon density. This does not allow us to determine the actual minimum number of errors, so we make an estimate where we have to account for double counting. The minimum number of errors could be found when the edge lengths are added up. It is simply the total length of all edges within clusters plus the remaining anyons that aren't part of any clusters on the path.

References

[1] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*. IEEE Comput. Soc. Press, doi: 10.1109/sfcs.1994.365700.

[2] A. Yu. Kitaev, "Fault tolerant quantum computation by anyons," *Annals Phys.*, vol. 303, pp. 2–30, 2003, arXiv:quant-ph/9707021.

[3] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, "Topological quantum memory," *J. Math. Phys.*, vol. 43, pp. 4452–4505, 2002, arXiv:quant-ph/0110143.

[4] A. Hutter, D. Loss, and J. R. Wootton, "Improved hdrg decoders for qudit and non-abelian quantum error correction," *New Journal of Physics*, vol. 17, no. 3, p. 035017, 2015, arXiv:1410.4478.

[5] E. Knill, R. Laflamme, and W. H. Zurek, "Resilient quantum computation: Error models and threshold," *Proc. Roy. Soc. Lond.*, vol. A454, pp. 365–384, 1998, arXiv:quant-ph/9702058.

[6] D. Gottesman, "Stabilizer codes and quantum error correction," *Caltech Ph.D. Thesis*, 1997, arXiv:quant-ph/9705052.

[7] S. Bravyi and J. Haah, "Quantum self-correction in the 3d cubic code model," *Phys. Rev. Lett.*, vol. 111, p. 200501, Nov 2013, doi: 10.1103/PhysRevLett.111.200501.

[8] H. Anwar, B. J. Brown, E. T. Campbell, and D. E. Browne, "Fast decoders for qudit topological codes," *New Journal of Physics*, vol. 16, no. 6, p. 063038, jun 2014, doi: 10.1088/1367-2630/16/6/063038.

[9] J. R. Wootton, "A simple decoder for topological codes," *Entropy*, vol. 17, no. 4, pp. 1946–1957, 2015, arXiv:1310.2393.

[10] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–48, jan 1956, doi: 10.1090/S0002-9939-1956-0078686-7.