Delft University of Technology

Master's Thesis

Quasilinear Time Decoding Algorithm for Topological Codes with High Error Threshold

Author
S. Hu

Supervisor
D. Elkouss

April 15, 2020

Contents

Chapter 1

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 [1]) 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 1.2. In section 1.1, 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.

1.1 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 [2]. 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.

1.1.1 A new data structure

1.1.2 Finding clusters

1.2 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 [1]. 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 **??**, which has a sub-linear complexity of $\mathcal{O}(\sqrt{n})$.

1.2.1 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 1.1 *For two clusters C_α, C_β with size $S_\alpha << S_\beta$ the number of vertices in the clusters, Grow(S_β) 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`.

$$\texttt{Place}(C) = \begin{cases} C \to b_{2(S_C-1)}, & \text{if } S_C \text{ even} \\ C \to b_{2(S_C-1)+1}, & \text{otherwise} \end{cases} \tag{1.1}$$

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 1.1. 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 **??**. The order of growth within the bucket is dependent on the order of cluster placement into the bucket.

Theorem 1.1 *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.

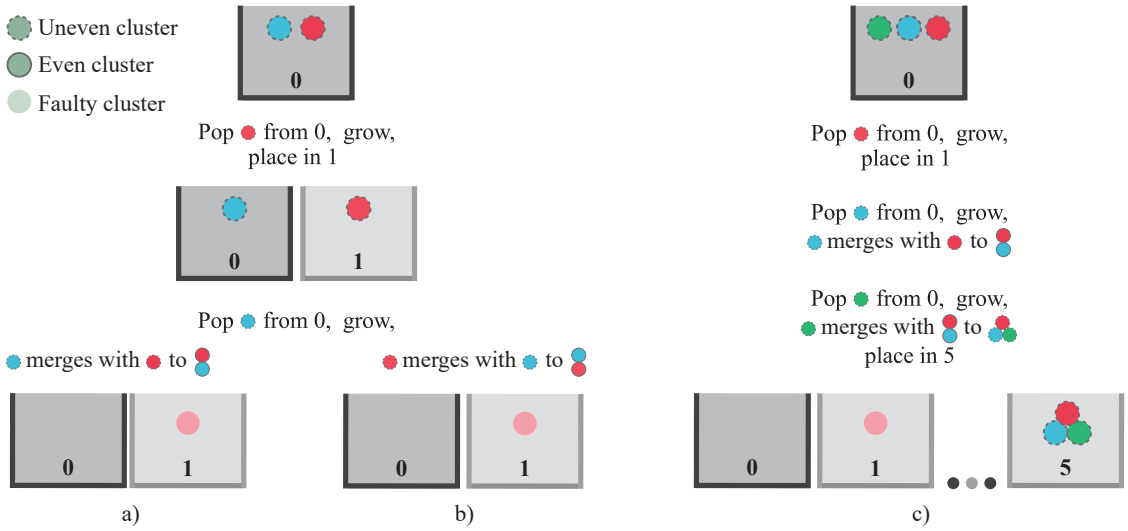


Figure 1.1: Faulty entries of clusters can occur in the buckets, a) cluster that should not be there due to a merge event. Situation a can be solved by checking the parity of the cluster. Checking the parity of the root cluster solves a) and b). Checking the bucket_number of the root cluster solves all.

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 1.2 *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 1.1, C_β will grow before C_α. As the remaining

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 `Grow`(C_μ) this is true. This cluster size S_μ is dependent on the code and the parity of lattice size L. We illustrate in figure 1.2 the clusters C_μ for the toric and planar code. Their maximum odd cluster size S_μ is listed in table 1.1, where $L' = L - 1$ for the planar code.

Lemma 1.3 *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* `Union`$(C_\alpha, C_\beta) = C_{\alpha\beta}$.

	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 1.1: 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.

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

$$k = 2(S_\mu - 1) \tag{1.2}$$

Any cluster with size $S \leq S_\mu$ will be placed into a bucket according to equation 1.1. 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.

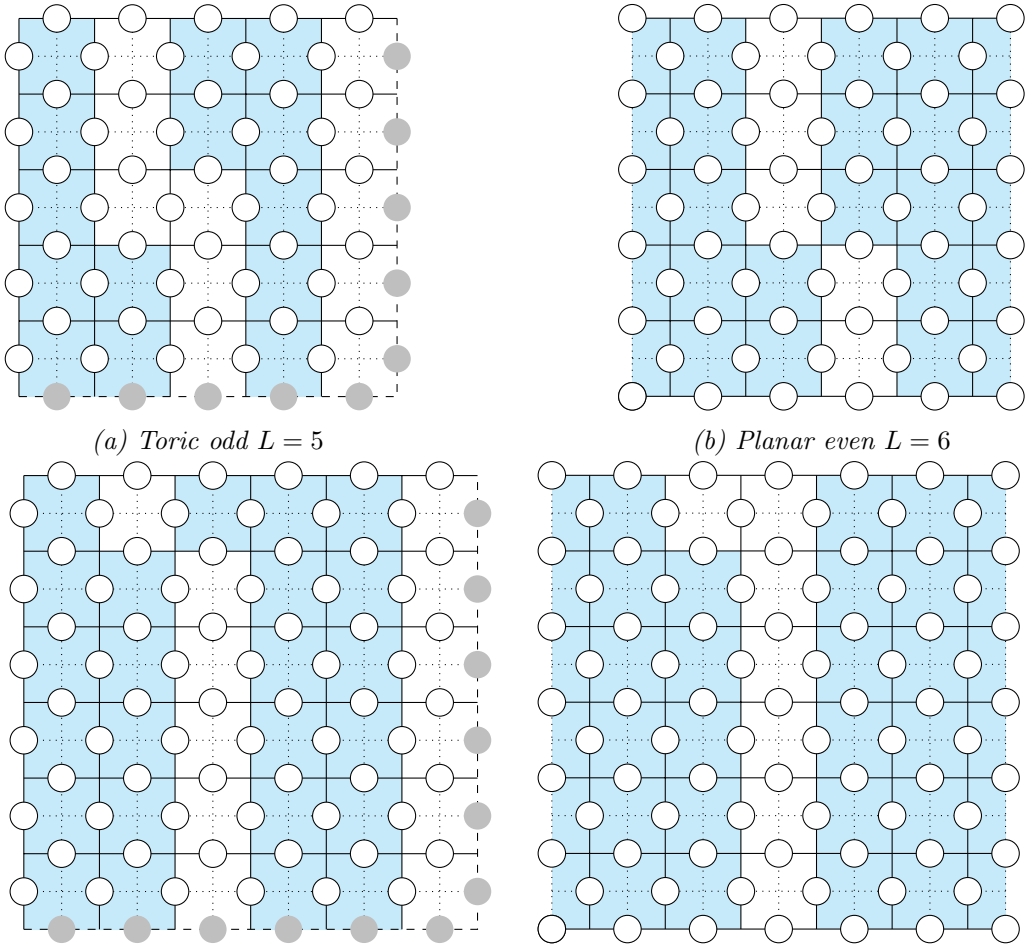
Largest bucket occurrence

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.

1.2.2 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.

1.2.3 The BCS Union-Find decoder



(a) Toric odd $L = 5$

(b) Planar even $L = 6$

(c) Toric even $L = 6$

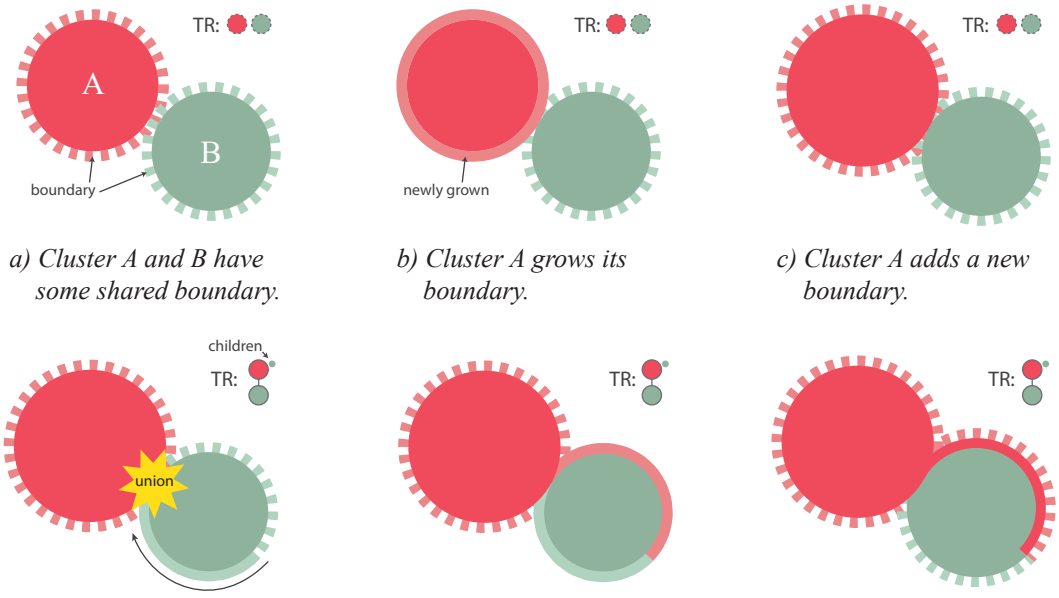
(d) Planar odd $L = 7$

Figure 1.2: 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.

1.3 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 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. 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.



a) Cluster A and B have some shared boundary.

b) Cluster A grows its boundary.

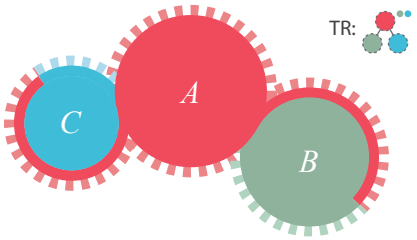
c) Cluster A adds a new boundary.

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

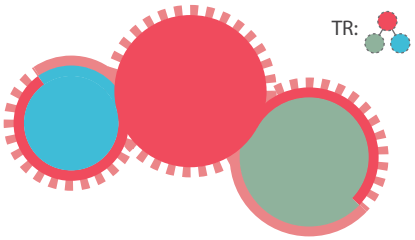
e) New vertices are added to root cluster A.

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

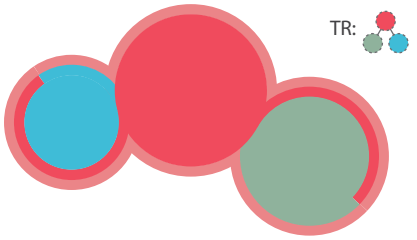
Figure 1.3: The parent-child method for merging boundary lists. By storing a list of pointers of child clusters at the parent cluster, we needn't append the full boundary list from the child to the parent cluster. The tree representation (TR) is shown on the top right.



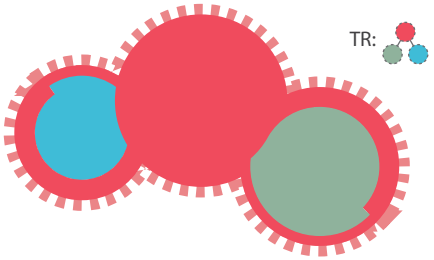
a) Cluster ABC with root cluster A is odd. Boundary of ABC is stored at root A, but also at children B and C.



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

1.4 Growing Edge Priority based on path degeneracy (GEP)

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

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

1.5 Union-Find Balanced Bloom

Not all clusters should be grown at the same time. Larger clusters relatively add more "incorrect edges" to themselves than compared to a smaller cluster (lemma 1.1). 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 1.2. 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 [1]. 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 matching. 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 growth iteration 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 merges between clusters, which 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 1.5.1. We will introduce a new data structure that we call the *node set* of a cluster in 1.5.2. Within this node set, we compute the node *parity* and *delay* in 1.5.3, which sets the order of boundary edge growth. In 1.5.4 through 1.5.6, 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, which we dub the *Union-Find Balanced Bloom decoder* (UFBB), still has a relatively low worst-case quasilinaer time complexity, which is approximated in 1.5.8.

1.5.1 Potential matching weight

To show that not all boundary edges within a cluster should not be grown simultaneously, let us first consider an example. Cluster C_e is defined by vertex set $\mathcal{V}_e = \{v_1, v_2, v_3\}$ (figure 1.5). 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 its surroundings until it actually grows its edges.

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). Inspired by this idea, we introduce the concept of potential matching weight (PMW) of a vertex.

Lemma 1.4 *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 sized odd cluster, the PMW is not equal for all vertices. 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. If the PMW is to be calculated for every vertex that has boundary edges for each cluster in each 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.

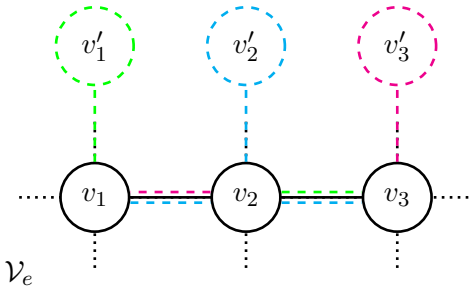


Figure 1.5: Unbalanced matching weight in cluster vertex set $\mathcal{V}$. The matching edges (dashed) correspond to the position of v'. If v' is connected to v_1 or v_3, the resulting matchings have a weight of 2. IF v' is connected to v_2, the resulting matching has a weight of 3.

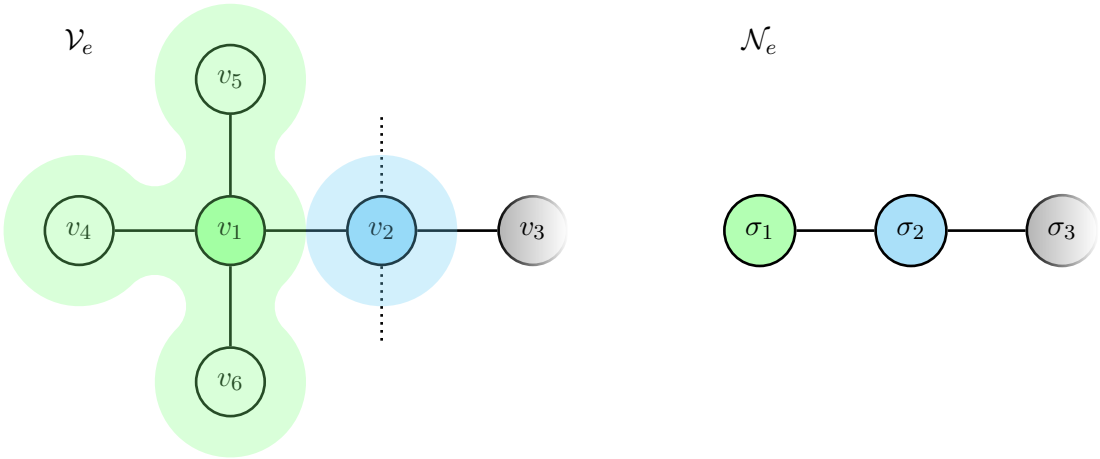


Figure 1.6: A node set $\mathcal{N}$ vs. a vertex set $\mathcal{V}$, both representing the same cluster. Each shaded area covers the vertices of a different node.

1.5.2 Node set representation of 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 Σ. 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 Σ. We say that these vertices are *seeded* in 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 Σ, 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. However, the number of nodes remains the same at 3 nodes (figure 1.6). For all vertices with boundary vertices seeded in the same node, the PMV is thus equal. The calculation of PMW's in the cluster thus does not require to traverse all the vertices, but just the nodes of the cluster. Furthermore, while these 3 nodes represent the cluster, we need only to calculate the PMW for the nodes in the cluster only once. To reach equal PMW in the cluster, we grow only the nodes with the smallest PMW, and delay the growth of nodes with larger PMW.

Lemma 1.5 *The calculation of PMW in the cluster can be limited to the nodes of a cluster, where all vertices seeded in a node have the same PMW.*

Balanced Bloom

We call the growth of the cluster in the subset of boundary edges that is seeded in n_i the *bloom* of node n_i. The flower of n_i is the subset of all vertices in the cluster seeded in the

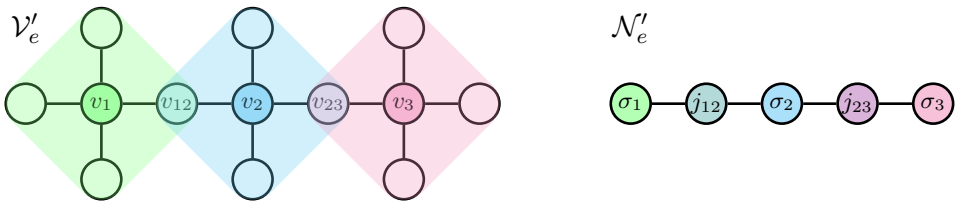


Figure 1.7: Merging vertices v_{12} and v_{23} are seeded in junction nodes j_{12} and j_{23}, respectively, as they lie in equal distance to more than 1 syndrome nodes.

node. Here we use the *object.property* notation to indicate that the property is stored at the parent object. The size of the flower $n_i.s$ is the number of growth iterations a node has grown, and is equal to the maximum weight of edges grown from this node. Note that it does not define the number of vertices in the flower. The combined bloom of all nodes in a cluster is equivalent to the growth of the full cluster, where some nodes are to wait for some iterations based on their delay $n_i.d$.

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. As these nodes bloom and increase in size $n.s$, the cluster moves towards equal PMW. Once equal PMW in the cluster is reached, the growth of a node set is the *balanced bloom* of nodes.

Theorem 1.2 *Every vertex v that is added to a cluster is seeded in some node n. All vertices with boundary edges that are seeded in the same node have the same PMV value. Equal PMV in the cluster is reached by selectively blooming the nodes with the lowest PMW values, as each bloom increases the node size $n.s$ and its PMV.*

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, we make these kind of vertices nodes of themselves, and call them *junction nodes* j. All nodes j have the same characteristics of syndrome nodes σ, and have their own delay and boundary edges seeded in them.

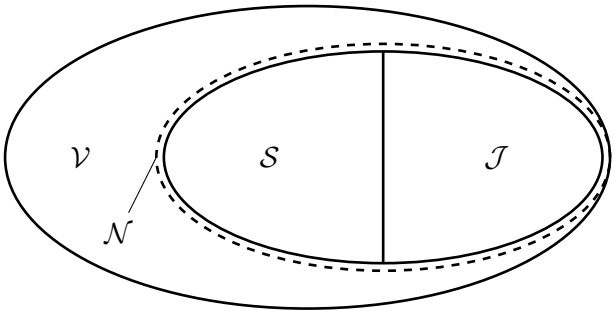


Figure 1.8: The space occupied by the sets of vertices $\mathcal{V}$ and nodes $\mathcal{N}$ (union of syndrome node set $\mathcal{S}$ and junction node set $\mathcal{J}$).

Lemma 1.6 *On a merging vertex v that lie in equal distance to two syndrome nodes from two cluster merging into one, we initiate a junction node j in the node set $\mathcal{N}$. A junction node has the same properties as a syndrome node.*

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, the junction node set, or not a node at all, but never both 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.

$$\begin{aligned} \mathcal{N} \subseteq \mathcal{V} \quad &, \quad S_{\mathcal{N}} \leq S_{\mathcal{V}} \\ \mathcal{S} \quad &\cup \quad \mathcal{J} = \mathcal{N} \\ \mathcal{S} \quad &\cap \quad \mathcal{J} = \varnothing \end{aligned} \tag{1.3}$$

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 $\mathcal{N}_\alpha$ is traversed. We store the node set by its root n_r at C_α.

Theorem 1.3 *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 $\mathcal{V}_\alpha$. The function of $\mathcal{N}_\alpha$ is to store the list of boundary edges at the nodes and growing each node according to the calculated node delay.*

1.5.3 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, for the PMW of each node the entire set needs to be checked on which edges are part of the matching, 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 *relative delay* of a node to its parent, where the traversal allows us to compute its *delay*, which relates closely to the PMW. The delay computation complexity is therefore 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 $S_{\mathcal{N}}$, where all nodes lie on one line, but are allowed to grow in x and y directions. In our example, we only look at the first 3 nodes n_1, n_2, connected by edge ϵ_1, and n_3, connected to n_2 by edge ϵ_2. The node tree continues after n_3 for $S_{\mathcal{N}} - 3$ nodes. Note that edges of the node set are indicated by ϵ, whereas edges of the vertex set are indicated by e.

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_1, we also add edge ϵ_2 and some value k corresponding to the weight of matchings in the remainder of the cluster. Let us calculate the PMW values for each of nodes n_1, n_2, n_3.

$$\begin{aligned} PMW(n_1) &= \lfloor n_1.s/2 \rfloor + 1 + \epsilon_2 + k \\ PMW(n_2) &= \lfloor n_2.s/2 \rfloor + 1 + \epsilon_1 + \epsilon_2 + k \\ PMW(n_3) &= \lfloor n_3.s/2 \rfloor + 1 + \epsilon_1 + k \end{aligned}$$

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

$$\begin{aligned} \delta(n_2.d) &= PMW(n_2.d) - PMW(n_1.d) \\ \delta(n_3.d) &= PMW(n_3.d) - PMW(n_3.d) \end{aligned}$$

By setting the delay for the first node to some value, for example 0, we can find the node delay.

$$\begin{aligned} n_1.d &= 0 \\ n_2.d &= PMW(n_2) - PMW(n_1) + n_1.d = 2(\lfloor n_2.s/2 \rfloor - \lfloor n_1.s/2 \rfloor + \epsilon_1) + n_1.d \\ n_3.d &= PMW(n_3) - PMW(n_2) + n_2.d = 2(\lfloor n_3.s/2 \rfloor - \lfloor n_2.s/2 \rfloor - \epsilon_2) + n_2.d \end{aligned}$$

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.

$$\begin{aligned} n_1.d &= 0 \\ n_2.d &= 2\big(\lfloor (n_2.s + n_2.g)/2 \rfloor - \lfloor (n_1.s + n_2.g)/2 \rfloor + \epsilon_1\big) - (n_1.g + n_2.g) \bmod 2 + n_1.d \\ n_3.d &= 2\big(\lfloor (n_3.s + n_3.g)/2 \rfloor - \lfloor (n_2.s + n_3.g)/2 \rfloor - \epsilon_2\big) - (n_3.g + n_2.g) \bmod 2 + n_2.d \end{aligned}$$

If we were to consider some nodes n_4, n_5... 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 ϵ_{2i+1} in its delay value, and the substraction of ϵ_{2i} for an even node n_{2i}. Thus we can generalize the delay calculation as the following:

$$n_i.d = n_{i-1}.d + 2\left(\left\lfloor \frac{(n_i.s + n_i.g)}{2} \right\rfloor - \left\lfloor \frac{(n_{i-1}.s + n_i.g)}{2} \right\rfloor + (-1)^{i+1}\epsilon_{i-1}\right) - (n_i.g + n_{i-1}.g) \bmod 2 \quad | \quad n_0.d = 0. \tag{1.4}$$

Using equation 1.4, we can calculate all the relative delays in the 1D tree by traversing the node tree just once from left to right. Note that we had set the initial delay to be $n_0.d = 0$, but it can any arbitrary value. Thus the absolute values of delays has no intrinsic meaning. It is the difference compared the lowest delay value in the cluster that relates to the PMW.

$$PMV(n_i) = n_i.d - \min\{n_0.d, ..., n_{S_{\mathcal{N}}}.d\} + k - n.w \tag{1.5}$$

Recall from theorem 1.2 that the goal of Balanced Bloom is to grow nodes with the lowest PMW first. We therefore do not care about the absolute values of node delays, only their difference to the smallest delay value in the cluster. For each node set, we therefore need to store the *minimal delay value*. This value can be stored at the cluster object as $C.d$.

Node tree parity

The 1D node tree does not accurately represent node trees that occur on 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. But 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 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 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

$$n_\beta.p = \begin{cases} 0, & \text{if } n_\beta \text{ has no children} \\ \left(\sum_j 1 - n_{\gamma,j}.p\right) \bmod 2 \mid \forall n_\gamma \text{ child of } n_\beta, & \text{otherwise,} \end{cases} \tag{1.6}$$

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

Lemma 1.8 *The node parity $n_i.p$ is defined as of 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, rendering node parities within the set to "undefined".*

Node tree delay

The delay equation 1.4 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

$$n_\beta.d = n_\alpha.d + 2\left(\left\lfloor \frac{(n_\beta.s + n_\beta.g)}{2} \right\rfloor - \left\lfloor \frac{(n_\alpha.s + n_\beta.g)}{2} \right\rfloor + (-1)^{n_\beta.p-1+1}\epsilon_\beta\right) - (n_\beta.g + n_\alpha.g) \bmod 2 \quad | \quad n_r.d = 0,\ n_\beta \text{ child of } n_\alpha, \quad (1.7)$$

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 1.8, the delay calculation is only valid if the DFS is performed from the same root n_r as in the parity calculation.

Lemma 1.9 *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 relative differences in delays in neighboring nodes are the some for any node as root $n_r = n \forall n \in \mathcal{N}$. The actual delay value may differ for different roots as we always set $n_r.d = 0$. This is the reason that when blooming a node, we check for $n.w = n.d - C.d$, as $n.d - C.d$ for each node is the same for any node as root. This fact strengthens lemma 1.7.

Junction node parity and delay

Up until now, we have neglected junction-nodes in our story on node parity and delays. But as junction nodes have the same properties as syndrome nodes, there also exists edges seeded in junction nodes, and thus they must be included in the parity and delay calculations. Furthermore, without junction nodes, lemma 1.7 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 1.7 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 1.7 as we are now only interested in the parity component $(-1)^{n_\beta.p-1+1}\epsilon_\beta$. The parity of σ_4 is odd, therefore

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

where $\epsilon = (\sigma_3, \sigma_4)$ is an edge connecting two nodes.

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

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

We see that 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 1.6 for realistic node sets.

$$n_\beta.p = \begin{cases} 0, & \text{if } n_\beta \text{ has no children} \\ \left(\sum_j 1 - n_{\gamma,j}.p\right) \bmod 2 \mid \forall n_\gamma \text{ child of } n_\beta, & n_\beta \equiv \sigma_\beta \\ 1 - \left(\sum_j 1 - n_{\gamma,j}.p\right) \bmod 2 \mid \forall n_\gamma \text{ child of } n_\beta, & n_\beta \equiv j_\beta \end{cases} \tag{1.8}$$

To put this into perspective of lemma 1.8, 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 definition of parity and delay calculation stays unchanged; the parities can to be calculated by a reversed DFS of the node tree from the root with equation 1.8, and the delays by a second DFS with equation 1.7.

Lemma 1.10 *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.*

To perform a reverse DFS of the node tree, we can use a *head recursive* function that calls itself, where the recursiveness is before the required routine. The parity calculation is then the following algorithm.

Algorithm 1: CalcParity

```
Data: node
Result: Defined parities for all children of node
parity = Sum([1− CalcParity(child) ∀ child of node ]) %2
if node ≡ σ then
    node.p = parity
elif node ≡ j then
    node.p = 1− parity
return node.p
```

Degree of delay due to parity inversion

With equation 1.7, we can calculate the appropriate delays in nodes such that if the bloom in these nodes are delayed for that many iterations, the PMW's for every node in the set is equal. We will see how to grow a node set in section 1.5.4. After that, we will see how to join two node sets in the case of a merge of two clusters in section 1.5.5. But before we move on, we already see a problem arising in the parity and delay calculations.

If some odd number of nodes $\mathcal{N}_o$ is attached to n^e of $\mathcal{N}_e$ during a join operation of two node sets, node parities for nodes in subset $\mathcal{N}'_e = \{n_i \in \mathcal{N}_e | n_i \text{ ancestor of } n^e\}$ are flipped, where odd nodes become even and even become odd, which is called *parity inversion*. Per lemma 1.9, the delays in $\mathcal{N}'_e$ are now undefined and need to be recalculated. If before the join operation, $\mathcal{N}_e$ had grown for some iterations where the odd nodes have waited (approaching equal PMW), the even nodes will have some node sizes larger than the odd node sizes $n^e_{even}.s > n^e_{odd}.s$. After the join operations, the parities for nodes in $\mathcal{N}'_e$ flip, and now the previously-even odd nodes have some positive delay. As $n^e_{even}.s > n^e_{odd}.s$, these delays will increase in value per equation 1.7 compared to the previous delay calculation.

As the lattice increases in size, the number of merges between clusters or join operations between node sets will also increase. The node parities for some parts of some node sets will suffer parity inversion during these merges, leading to increasingly larger delay values. The delayed bloom of nodes may therefore not be balanced at all with the current delay equation. We therefore introduce a parameter $K_{bloom} \in [0, 1]$ that determines the degree of delay of a node.

$$n_\beta.d = n_\alpha.d + \left\lceil K_{bloom} \left(2 \left(\left\lfloor \frac{(n_\beta.s + n_\beta.g)}{2} \right\rfloor - \left\lfloor \frac{(n_\alpha.s + n_\beta.g)}{2} \right\rfloor + (-1)^{n_\beta.p-1+1} \epsilon_\beta \right) - (n_\beta.g + n_\alpha.g) \bmod 2 \right) \right\rceil \quad | \quad n_r.d = 0, \;\; n_\beta \text{ child of } n_\alpha, \quad (1.9)$$

From intuition the degree of delay should be set to $K_{bloom} = 1/2$. For this value, the delays in a node set are halved, such that in the case of parity inversion, the delay values from before and after the inversion are kept at minimum. But as the inversion of parities mostly does not occur on all nodes in a set, this is not necessarily true, and other values of K_{bloom} should be explored.

Lemma 1.11 *The degree of delay K_{bloom} determines the part of the calculated delays that is actually assigned to the nodes. This is to minimize the node delays in new delay calculations in nodes that have suffered parity inversion after a join operation with another node set.*

The delay calculation is done by a DFS of the node tree, which can be done by a *tail recursive* function. Here the recursiveness is after the routine, which satisfies the DFS. The delay calculation is then the following algorithm.

Algorithm 2: CalcDelay

```
Data: node, cluster
Result: Defined parities for all children of node
if node has an ancestor then
    calculate node.d with equation 1.9
    if node.d < cluster.d then
        cluster.d = node.d
for child of node:
    CalcDelay(child, cluster)
```

Parity and delay routines

With equation 1.8 and 1.9, 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 1, where we do a reverse DFS of the node tree. The node delays are calculated by calling the tail recursive function `CalcDelay` in algorithm 2, where we do a second DFS of the node tree.

Theorem 1.4 *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).*

1.5.4 Growing a cluster

The boundary list for each cluster is not stored at C, but separately stored at each of the nodes n_i in $\mathcal{N}$. To grow a cluster `Grow`(C), we now 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 $\mathcal{L}_{n_i}$ at node n_i by 1. If this node hasn't waited enough $n_i.w - n_i.d - C.d > 0$, we skip this node, add to the wait $n_i.w = n_i.w + 1$ and apply `Bloom` 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 1.5 *A cluster C is grown by calling `Bloom`(n_r), which first checks for the wait of the current node $n_i.w - n_i.d - C.d > 0$ to grow its boundary edges, and then recursively applies `Bloom` to its children.*

Algorithm 3: Grow

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

```
if node.w = node.d − cluster.d then
    Bloom(node), add all edges edge.support = 2 to F
else:
    node.w + = 1
for child of node:
    Grow(child)
```

1.5.5 Joint of node sets

With the addition of the node set $\mathcal{N}$, during a union of clusters C_α and C_β, we have to additionally combine the node sets $\mathcal{N}_\alpha$ and $\mathcal{N}_\beta$. Let us first make a clear distinction between the various routines. On the vertex set $\mathcal{V}$ we apply `Union`(v^α, v^β), on the two vertices spanning the edge connecting two clusters. On node set $\mathcal{N}$, we introduce here `Joint`(n^α, n^β), which is called on the two nodes n^α, n^β that seeds vertices v^α, v^β, respectively. From now on, when we talk about the "merge clusters C^α and C^β", "the union of vertex sets $\mathcal{V}_\alpha$ and $\mathcal{V}_\beta$" or the "joint of node sets $\mathcal{N}_\alpha$ and $\mathcal{N}_\beta$", we always refer to the combination of these two routines.

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 `Find` function. Similarly, in the node set $\mathcal{N}$, we would also like to apply a set of rules to minimize the calls to `CalcParity` and `CalcDelay`. As the structure of the tree is crucial in computing the parities and relative delays between the nodes, these rules will be quite different than in vertex set $\mathcal{V}$. First of all, we note that while the node set does not change, the parities and delays within the node set stay valid.

Lemma 1.12 *While node set $\mathcal{N}$ is unchanged, the calculated parities and delays within the set are valid.*

Our rules will be dependant on the parities of the joining node sets, which is the number of syndrome-nodes in the set modulo 2. This is due to that junction-nodes do not add to the count of the number of children nodes per lemma 1.10. Note that the parity of a node set $\mathcal{N}.p$ is therefore exactly the same as the parity of a cluster $C.p$, which also refer to the number of syndromes in the cluster.

Lemma 1.13 *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 $C.p$.*

Joint to even node set

Let us first consider the joint operation of two or more node sets, where the resulting node set $\mathcal{N}_e$ is even. If we calculate the parities of a node set with even parity $\mathcal{N}_e$, we will end

up with an odd node $n_r.p = 0$ as root of node set $\mathcal{N}_e$. It therefore does not make sense to talk about node parities within an even node set. Luckily, but not coincidentally, if a node set is even, the cluster is even and therefore will not grow. We could say that we need not to worry about the parties and delays within $\mathcal{N}_e$.

Lemma 1.14 *Node parities become undefined if multiple node sets joins into a new set $\mathcal{N}$ with even parity.*

However, it is entirely possible that another cluster grows, and merges onto the cluster of $\mathcal{N}_e$. In that case, we might think about recovering some of the node parities and delays that were calculated in the subsets of $\mathcal{N}_e$, such that we don't have to traverse $\mathcal{N}_e$ entirely for its parities and delays.

Joint to odd node set

Consider the joint operation of an even $\mathcal{N}^e$ and an odd node set $\mathcal{N}^o$ in nodes n^e, n^o respectively, and assume that this joining is due to the growth of odd cluster $\mathcal{N}^o$ onto an "idle" $\mathcal{N}^e$. The joint of these two sets leaves 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. Note that the child node n^c will become the *sub-root* in subset $'\mathcal{N}^c$, where the ancestry in the subset has been reset in the new sub-root, and is allowed per lemma 1.7.

If the subset $'\mathcal{N}^e$ consists of only two odd node sub-subsets $''\mathcal{N}^o_0$,$''\mathcal{N}^o_1$, where n_0, n_1 are the joining nodes, the ancestry in $''\mathcal{N}^o_0$ is preserved and n_1 is the sub-root of $''\mathcal{N}^o_1$. We see that the parities in all ancestors of n_0 are flipped. Let's consider the cases and find whether we can minimize the parity and delay calculation in $'\mathcal{N}^e$.

For case 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. This means that the parities in $'\mathcal{N}^o$ do not change per lemma 1.8, and the delays in $'\mathcal{N}^o$ are still valid as per lemma 1.9. In $'\mathcal{N}^e$, as the ancestry path has changed, we are certain to traverse $'\mathcal{N}^e$ from the sub-root n^e to calculate the delays in this subset which is in the order of $S_{'\mathcal{N}^e}$.

In case b), as an odd number of nodes of $'\mathcal{N}^o$ is attached to n^e, it means that parities of all ancestor of n^e are flipped. As the ancestry in $'\mathcal{N}^o$ has changed, we are certain to traverse $'\mathcal{N}^o$ from the sub-root n^o to calculate the delays which is in the order of $S_{'\mathcal{N}^o}$. The node parity changes in $'\mathcal{N}^e$ will be dependant on the location of n^e in the ancestry compared to n^1 and n^2, and all children nodes of these parity changes will have to recalculate their delays. Let's call the number of nodes needs to calculate parity and delays in $'\mathcal{N}^e$ a value $S_e \leq S_{'\mathcal{N}^e}$, leaving the total number of operations in the order of $S_e + S_{'\mathcal{N}^o}$.

For $'\mathcal{N}^e$ consisting of two subsets, keeping track of the parity changes between n^e, n^0 and n^1 is still an easy task, and we might gain in minimization in operations in case b) compared to case a) for some value S_e such that $S_e + S_{'\mathcal{N}^o} < S_{'\mathcal{N}^e}$. But as the number of subsets in $'\mathcal{N}^e$ increases, the task of finding the ancestry paths of parity changes becomes analogous to traversing $'\mathcal{N}^e$ entirely $S_e \to S_{'\mathcal{N}^e}$. To simplify, we always choose case a.

Theorem 1.6 *The union of node sets $\mathcal{N}^\alpha, \mathcal{N}^\beta$ on nodes n^α, n^β respectively is performed with* `Joint`(n^α, n^β). *If the resulting node set $\mathcal{N}$ is odd, one of $\mathcal{N}^\alpha$ and $\mathcal{N}^\beta$ is odd while the other is even, and* `Joint`(n^α, n^β) *makes the node of the even set n^e a child of the node of the odd set n^o. If the resulting node set $\mathcal{N}$ is even, the choice is arbitrary.*

1.5.6 Multiple joints per bucket

Clusters with same vertex set size $S_\mathcal{V}$ lie in the same bucket b_i and thus are grown together per theorem 1.1. Let's call the growth of all clusters in a bucket a *growth iteration*. As the state of fully grown edges are only checked after each growth iteration including possibly many clusters, there may be multiple joint events within the same growth iteration. If the node parities and delays are to be calculated on all the union events in the fusion list $\mathcal{F}$, some node parity and delay calculations may be invalid and unnecessary.

Consider an example with 5 odd clusters $C_1, ...C_5$ with node sets $\mathcal{N}_1, ...\mathcal{N}_5$. The union of C_1 and C_2 to C_{12} is odd-odd and requires no parity-delay calculation. The union of C_{12} and C_3 is even-odd, and we calculate the parities and delays in the $\mathcal{N}_{12}$. The union of C_{123} and C_4 is odd-odd and the union of C_{1234} and C_5 is even odd, and we calculate the parities and delays in $\mathcal{N}_{1234}$. The earlier computation in $\mathcal{N}_{12}$ was therefore unnecessary and possible invalid.

To circumvent this even parity multiplicity, we must make sure to only apply the calculation to the largest even node set in some sequence of joint operations. To do this, we first 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, as long as the sum is even.

Lemma 1.15 *An odd node set $\mathcal{N}$ that is the result of some joint operations 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 $''\mathcal{N}$.*

This even subset $'\mathcal{N}^e$ is the undefined part of $\mathcal{N}^o$ in which we must calculate the parities and delays. As we can only be sure that the subset $'\mathcal{N}^e$ is complete after all unions of clusters are complete, we cannot apply and parity-delay calculations during the unions. We suspend these calculations as much as possible by doing them just before a cluster is grown from a bucket.

Lemma 1.16 *Parity and delay calculations are only performed on a the undefined part of a node set when a cluster is grown, not directly after a joint operation.*

The only task now is to store where the even subset $'\mathcal{N}^e$ starts in the ancestry of subset $\mathcal{N}^o$. For each joint operation between odd node set $'\mathcal{N}^o$ and even node set $'\mathcal{N}^e$ on nodes n^o, n^e per theorem 1.6, we store the sub-root $'n^e_r$ of subset $'\mathcal{N}^e$ to a list $\mathcal{C}$ at the root node of the resulting set $\mathcal{N}^{res}$ of cluster C^{res}. If cluster C is selected for growth as per theorem 1.1, we first check for nodes in $n^r.\mathcal{C}$ at root and apply `CalcParity`(n_i) and `CalcDelay`(n_i) for all nodes $n^i \in n^r.\mathcal{C}$ to calculate parities and delays in undefined parts of the set. We then call `Bloom`(n_r) per theorem 1.5.

Theorem 1.7 *Undefined parts of an odd node set $\mathcal{N}^o$ are defined as a set $\mathcal{C}_{\mathcal{N}^o}$ of sub-roots from which all children (including sub-roots) are undefined, and is stored at root node n_r^o. If before it has grown, node set $\mathcal{N}^o$ is joint with another odd node set, which then act as the even set ${}^{\uparrow}\mathcal{N}^e$ in a larger joint event, the sub-root of ${}^{\uparrow}\mathcal{N}^e$ is stored at $n_r^{\uparrow\mathcal{N}^o}$. Delay and parity calculations will then traverse all undefined paths only once.*

1.5.7 Pseudocode

Now we have the full description of the *Balanced Bloom* alteration of the UF decoder, we can present its pseudocode in algorithm 4. The recursive `Grow` function of algorithm 3 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 `Joint`, after each bucket iteration.

Algorithm 4: UFBalancedBloom

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.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 F
9          else:
10             node.w + = 1
11         for child of node:
12             repeat lines 7-12 on child
13     for edge in F:
14         Union(v1, v2) for edge = (v1, v2)
15         Joint(n1, n2) for v1, v2 seeded in nodes n1, n2
16     Place(cluster) ∀ odd clusters
```

1.5.8 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`. As these two functions are always called together per theorem 1.7, we can just introspect the number of calls to one of them, and call this contribution the *delay* complexity. The second contribution will be caused by `Grow` of algorithm 3, 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.

Delay complexity

Consider an odd cluster represented by node set ${}^{k-1}\mathcal{N}^o$ with set size $S_{{}^{k-1}\mathcal{N}^o}$ that is the result of union between a number of clusters ${}^kC_i$ with node subsets ${}^k\mathcal{N}_i$. Here k indicates a *generation*, where larger k indicates a more distant descendent generation of smaller clusters. As ${}^{k-1}\mathcal{N}^o$ is odd, it will be selected for growth. And because it consists of a number of subsets, ${}^{k01}\mathcal{N}^o$ is bound to consist of an odd subset ${}^k\mathcal{N}_0^o$ and an even subset ${}^k\mathcal{N}^e$ (lemma 1.15) on which we are to calculate the parities and delays (theorem 1.7).

Fragmentation of a node set Let us call this division of odd set into smaller odd and even subsets the *partial fragmentation* f_1 of ${}^{k-1}\mathcal{N}^o$. We can apply another partial fragmentation f_2 of ${}^k\mathcal{N}^e$ into 2 odd subsets $\mathcal{F}'_k = \{{}^k\mathcal{N}_1^o, {}^k\mathcal{N}_2^o\}$, and call the 2 fragmentations f_1, f_2 of ${}^k\mathcal{N}^o$ into a set of node sets $\mathcal{F}_k = \{{}^1\mathcal{N}_0^o, {}^1\mathcal{N}_1^o, {}^1\mathcal{N}_2^o\}$ a *fragmentation step* f. Each odd subset ${}^k\mathcal{N}_i^o$ of $\mathcal{F}_k$ can continue to be partially fragmented by f_1 into ${}^{k+1}\mathcal{N}_{i,0}^{o,o}$ and ${}^{k+1}\mathcal{N}_i^{o,e}$ the same way. 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.

Lemma 1.17 *Let the separation of an odd node set ${}^{k-1}\mathcal{N}^o$ into subsets $\mathcal{F}'_k = \{{}^k\mathcal{N}_0^o, {}^k\mathcal{N}^e\}$ be the partial fragmentation f_1 and subsequently into subsets $\mathcal{F}_k = \{{}^k\mathcal{N}_0^o, {}^k\mathcal{N}_1^o, {}^k\mathcal{N}_2^o\}$ be f_2 of $\mathcal{N}$. The combination of the two is a fragmentation step f.*

$$\mathcal{F}_k = f({}^{k-1}\mathcal{N}^o) = f_2(f_1({}^{k-1}\mathcal{N}^o)) = f_2(\{{}^k\mathcal{N}_0^o, {}^k\mathcal{N}^e\}) = \{{}^k\mathcal{N}_0^o, {}^k\mathcal{N}_1^o, {}^k\mathcal{N}_2^o\} \quad | \quad S_{{}^k\mathcal{N}_i^o} \geq 3 \tag{1.10}$$

Each odd node set of $\mathcal{F}_k$ can undergo the same fragmentation step into odd subsets, leaving us again with a set of node subsets $\mathcal{F}_{k+1}$. We can do this some p times until our resulting set of node sets $\mathcal{F}_p$ consists only of smallest possible node subsets $\mathcal{N}^o$ where $S_{\mathcal{N}^o} = 1$. To find the worst case complexity, we want to maximize the delay complexity within ${}^{k-1}\mathcal{N}^o$; we are to find the sequence of joint operations that maximizes the sum of even node sets sizes $S_{\mathcal{N}^e}$ in all partial fragmentations in the *full fragmentation* F of $\mathcal{N}^o$.

Looking at the fragmentation from the other way, we have a set of size 1 node sets that undergo joint operations in each partial fragmentation. In f_2, two odd node sets join, and we do not add to the count of N_delay. In f_1, an odd and an even note sets join, and we have to calculate the delays in the even node set before moving on to the next joint operation.

Lemma 1.18 *Let the full fragmentation of $\mathcal{N}$ be*

$$F(\mathcal{N}^o) = \underbrace{f(f(...f(\mathcal{N})))}_{p \text{ times}} = \{{}^p\mathcal{N}_1^o, {}^p\mathcal{N}_1^o, {}^p\mathcal{N}_2^o, ..., {}^p\mathcal{N}_{N_\sigma}^o\} \quad | \quad S_{{}^p\mathcal{N}_i^o} = 1, \tag{1.11}$$

where along each fragmentation step k a partial fragmentation set $\mathcal{F}'_k$ is produced, the number of delay calculations is

$$N_{delay} = \sum_{k=1}^{p} \sum \{S_{k\mathcal{N}^e} | \forall \ even \ {}^k\mathcal{N}^e \in \mathcal{F}'_k\}. \tag{1.12}$$

Partial fragmentation number Note that here we ignore the fact that the partial fragmentations of some node set may not result in two but many subsets. Let us call the number of odd subsets the *fragmentation number* N_f. For partial fragmentation f_1, the separation of the odd node set ${}^{k-1}\mathcal{N}^o$ must be in 1 odd and 1 even subset per lemma 1.15, thus $N_{f_1} = 2$. For partial fragmentation f_2, the separation of even set ${}^k\mathcal{N}^e$ can be in $2n_o$ odd and n_e even subsets. But any even subset will be subjected to the same partial fragmentation f_2 in the full fragmentation, reducing the fragmentation number to $N_{f_2} = 2n_o$.

To find N_{f_2}, let us consider two cases where $n_o = 1$ or $n_o = 2$. If an even node set $\mathcal{N}^e$ is fragmented with ${}^kN_{f_2} = 2$, a fragmentation step of f_2, f_1 will be

$$\begin{aligned} \mathcal{F}^e_k &= \{{}^k\mathcal{N}^o_1, {}^k\mathcal{N}^o_2\}, \\ \mathcal{F}'^e_{k+1} &= \{{}^{k+1}\mathcal{N}^{o,o}_{1,0}, {}^{k+1}\mathcal{N}^{o,e}_1, {}^{k+1}\mathcal{N}^{o,o}_{2,0}, {}^{k+1}\mathcal{N}^{o,e}_2\}. \end{aligned}$$

For $N_{f_2} = 4$, a fragmentation step will be

$$\begin{aligned} \mathcal{F}^e_k &= \{{}^k\mathcal{N}^o_1, {}^k\mathcal{N}^o_2 \, {}^k\mathcal{N}^o_3 \, {}^k\mathcal{N}^o_4\}, \\ \mathcal{F}'^e_{k+1} &= \{{}^{k+1}\mathcal{N}^{o,o}_{1,0}, {}^{k+1}\mathcal{N}^{o,e}_1, {}^{k+1}\mathcal{N}^{o,e}_{2,0}, {}^{k+1}\mathcal{N}^{o,e}_2, {}^{k+1}\mathcal{N}^{o,e}_{3,0}, {}^{k+1}\mathcal{N}^{o,e}_3, {}^{k+1}\mathcal{N}^{o,o}_{4,0}, {}^{k+1}\mathcal{N}^{o,e}_4\}. \end{aligned}$$

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 as more fragmentation steps are applied. As the sum of even node set sizes in each fragmentation step is the same, increasing N_{f_2} will decrease the number of fragmentation steps and thus the number of delay calculations N_{delay} per equation 1.12. Thus our decision of $N_{f_2} = 2$ in lemma 1.17 is correct.

Partial fragmentation ratio To complete the fragmentation description, 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 $\mathcal{F}_{k-1}$ with respect to the size of ${}^k\mathcal{N}^o$, where $R_i S_{{}^{k-1}\mathcal{N}^o}$ is the size of subset ${}^k\mathcal{N}^o_i$. Note that R_0 corresponds to the odd subset from f_1, and R_1, R_2 to the odd subsets in f_2.

Lemma 1.19 *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*

$$R_j = \frac{S_{{}^k\mathcal{N}_j^o}}{S_{{}^{k-1}\mathcal{N}^o}} \tag{1.13}$$

Recall lemma 1.16 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 union or joint operations can occur. If no joint operations occur, the node set stays unchanged, and the cluster is allowed to continue growth without delay calculations per lemma 1.12. We want to minimize n_v, as each added vertex here is not a node that can possibly count towards N_{delay}, and it is therefore preferable that some joint operations do occur during growth.

Take the first fragmentation sets $\mathcal{F}'_k = \{\mathcal{N}_0^o, \mathcal{N}^e\}$ and $\mathcal{F}_k = \{\mathcal{N}_0^o, \mathcal{N}_1^o, \mathcal{N}_2^o\}$ of some cluster defined by node set ${}^{k-1}\mathcal{N}^o$. These partial fragmentation correspond to 2 joint operations, between two odd clusters $\mathcal{N}_1^o, \mathcal{N}_2^o$ in f_2, and between odd and even clusters $\mathcal{N}_0^o, \mathcal{N}^e$ in f_1.

If we want to minimize n_v in f_2, 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_1, the joint event is caused by growth of $\mathcal{N}_0^o$ in either some bucket $b_j > b_i$ where $S_{\mathcal{V}_0} > S_{\mathcal{V}_1}$, or growth in the same bucket b_i where $S_{\mathcal{V}_0} = S_{\mathcal{V}_1}$. This leaves us with $S_{\mathcal{V}_0} \geq S_{\mathcal{V}_1} = S_{\mathcal{V}_2}$.

To maximize N_{delay}, we want to maximize $S_{\mathcal{N}^e} = S_{\mathcal{N}_1^o} + S_{\mathcal{N}_2^o}$ in f_1. Recall from equation **??** 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 1.20 *A fragmentation step of ${}^{k-1}\mathcal{N}^o$ is maximized in $S_{{}^k\mathcal{N}^e}$ if the fragmentation ratios take the values $R_j = \frac{1}{3}$.*

Time complexity To find the time complexity cased by the delay calculations, we are left with a single variable p, the number of fragmentation steps that can be taken. 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 1.20, which can be calculated easily.

$$p = \log_3(S_{\mathcal{N}^o}) \tag{1.14}$$

In each fragmentation step $\mathcal{F}'_k$, f_1 is equivalent to the joint of odd node sets with even node sets where the sum of odd sets sizes is

$$\sum\{S_{{}^k\mathcal{N}^o} | \forall \text{ odd } {}^k\mathcal{N}^o \in \mathcal{F}'_k\} = \frac{1}{3}S_{\mathcal{N}^o}, \tag{1.15}$$

and the sum of even node set of sizes is

$$\sum\{S_{{}^k\mathcal{N}^e} | \forall \text{ even } {}^k\mathcal{N}^e \in \mathcal{F}'_k\} = \frac{2}{3}S_{\mathcal{N}^o}. \tag{1.16}$$

This approximation is true as we have taken $S_{\mathcal{N}} = S_{\mathcal{V}}$ and $n_v = 0$. Filling in equation 1.14 and 1.16 in 1.12, we find that

$$\begin{aligned} N_{delay} &\leq \sum_{k=1}^{p} \sum \{S_{{}^k\mathcal{N}^e} | \forall \text{ even } {}^k\mathcal{N}^e \in \mathcal{F}'_k\}. \\ &= \sum_{k=1}^{\log_3(S_{\mathcal{N}^o})} \frac{2}{3} S_{\mathcal{N}^o} \\ &= \frac{2}{3} S_{\mathcal{N}^o} \log_3(S_{\mathcal{N}^o}) \end{aligned} \tag{1.17}$$

The node set size of set $\mathcal{N}^o$ is bounded by the lattice size N. The worst case time complexity of the delay computation is bounded by $\mathcal{O}(N \log_3(N))$. The real worst-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 ${}^kN_i^o$ in each fragmentation set $\mathcal{F}_k$ is subjected to growth in each partial fragmentation, and that we start out 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}.

$$N_{bloom} \leq \sum_{p}^{k=1} \sum \{S_{{}^k\mathcal{N}} | {}^k\mathcal{N} \in \mathcal{F}_k\} \tag{1.18}$$

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

$$\sum \{S_{{}^k\mathcal{N}} | {}^k\mathcal{N} \in \mathcal{F}_k\} = S_{\mathcal{N}}. \tag{1.19}$$

By filling in p we find that

$$\begin{aligned} N_{bloom} &\leq \sum_{p}^{k=1} \sum \{S_{{}^k\mathcal{N}} | {}^k\mathcal{N} \in \mathcal{F}_k\} \\ &= \sum_{k=1}^{\log_3(S_{\mathcal{N}^o})} S_{\mathcal{N}} \\ &= S_{\mathcal{N}^o} \log_3(S_{\mathcal{N}^o}), \end{aligned} \tag{1.20}$$

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

1.5.9 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 1.

Algorithm 5: CalcParity for surfaces with boundaries

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

```
parity = Sum([1− CalcParity(child) ∀ child of node ]) %2
if node ≡ σ then
    node.p = parity
elif node ≡ j then
    node.p = 1− parity
elif node ≡ β then
    node.p = 1
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.

1.5.10 Erasure noise

The inspiration for the UF decoder is the Peeling decoder [3], 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.

Bibliography

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

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

[3] 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).