arXiv:2305.08307v1 [quant-ph] 15 May 2023

Fusion Blossom: Fast MWPM Decoders for QEC

Yue Wu and Lin Zhong
Department of Computer Science, Yale University, New Haven, CT

***Abstract*—The Minimum-Weight Perfect Matching (MWPM) decoder is widely used in Quantum Error Correction (QEC) decoding. Despite its high accuracy, existing implementations of the MWPM decoder cannot catch up with quantum hardware, e.g., 1 million measurements per second for superconducting qubits. They suffer from a backlog of measurements that grows exponentially and as a result, cannot realize the power of quantum computation. We design and implement a fast MWPM decoder, called Parity Blossom, which reaches a time complexity almost proportional to the number of defect measurements. We further design and implement a parallel version of Parity Blossom called Fusion Blossom. Given a practical circuit-level noise of 0.1%, Fusion Blossom can decode a million measurement rounds per second up to a code distance of 33. Fusion Blossom also supports stream decoding mode that reaches a 0.7 ms decoding latency at code distance 21 regardless of the measurement rounds.**

I. Introduction

Quantum error correction (QEC) is essential for fault-tolerant quantum computing. The decoder of QEC must be fast enough to avoid exponential backlog effect discussed by Terhal [1]. That is, it must process all the syndrome bits generated by the quantum hardware within a smaller period of time. Such fast decoders are known as *online* decoders. Also, a decoder design should be *scalable* to support large code distances in order to reach the desired logical error rate.

No scalable online MWPM decoders have been reported. Fowler, Adam and Lloyd [2] reported an almost linear-time MWPM decoder without a publicly accessible implementation. Higgott and Gidney recently reported an open-sourced, almost linear-time MWPM decoder at [3]. Since they are sequential algorithms, they will eventually fail to reach the throughput requirement at some large code distance. Fowler also suggested an idea to parallelize the MWPM decoder [4], without providing any empirical data regarding its performance.

We design Fusion Blossom as an online MWPM decoder that scales to arbitrarily large code distance d, using parallelization. Fusion Blossom is inspired by recently reported parallel realizations of the Union-Find (UF) decoder [5–8] and by the relationship between the UF decoder and the MWPM decoder revealed in [9]. Fusion Blossom drastically speeds up QEC decoding to sub-microsecond per measurement round by using parallel CPU cores. Taking a rotated surface code of 0.1% circuit-level noise on a 64-core CPU as an example, it can decode up to $d = 33$ with throughput of one million rounds per second using batch decoding. Using stream decoding, it achieves a constant 0.7 ms average latency at $d = 21$ regardless of the number of measurement rounds. To the best of our knowledge, Fusion Blossom is the first publicly available parallel MWPM decoder [10], implemented in Rust with Python binding [11].

The key ideas of Fusion Blossom are two. First, it recursively divides a decoding problem into two sub-problems that can be solved independently and efficiently fuses their solutions, according to a tree structure computed offline. Second, it leverages a fast sequential MWPM decoder called Parity Blossom, which implements a novel variant of the blossom algorithm. Parity Blossom leverages the property of the *syndrome graph* [9] where the MWPM problem is defined: the syndrome graph is constructed from a much sparser graph called *decoding graph*. Parity Blossom works on the decoding graph to solve the MWPM problem for the syndrome graph.

In an impressive parallel work, Higgott and Gidney [3] present Sparse Blossom, an implementation of blossom algorithm that shares the key idea of Parity Blossom: identifying tight edges using the decoding graph, and the same mathematical foundation. Sparse Blossom features several novel optimizations that are not used by Parity Blossom. This paper presents the following contributions that complement those by Sparse Blossom.

- The mathematical foundation behind Sparse Blossom and Parity Blossom (III).
- Fusion Blossom, a parallel MWPM decoder that could be based on either Sparse Blossom or Parity Blossom (IV).
- A unified framework for implementing matching-based decoders including novel mathematically grounded optimizations (V).

We evaluate Parity Blossom and Fusion Blossom in §VI and discuss related work in §VII. Our implementation is open-source and available at [10].

II. Background

We first define the necessary data structures for decoding a surface code. More information can be found in [9].

A. Quantum Error Correction (QEC) Codes

We aim at decoding codes that can be represented by a data structure called *model graph*. Such codes include many of the topological codes [12]. Such a code consists of data qubits and stabilizers. Data qubits store the quantum information while stabilizers allow errors in data qubits to be observed classically: an error in a data qubit will impact the measurement outcome of the adjacent stabilizers that are designed to detect this type of error.

a) Model Graph: Following [9], we represent such a QEC code with a *model graph* $G_M = (V_M, E_M)$. A vertex $v \in V_M$ corresponds to a stabilizer measurement result. Each edge $e \in E_M$ corresponds to an independent error source and connects with two vertices that correspond to the measurement

outcomes of stabilizers adjacent to this error source. We add a *virtual vertex* to an edge if the corresponding error source only connects with a single vertex. This results in a two-dimensional graph as show in Fig. 1(2). The model graph is sparse because $|E_M|$ is $O(|V_M|)$. Edges in the model graph are weighted. The weight of an edge can be computed from the error model of the corresponding error source $P(e)$ as $w_e = \log\left(\frac{1-P(e)}{P(e)}\right)$.

The model graph can be generalized to be three-dimensional to account for erroneous stabilizer measurement shown in Fig. 4. The third dimension comes from multiple rounds of measurement, with each round represented by the two-dimensional model graph as shown in Fig. 1(2). Vertices corresponding to the measurement outcomes of the same stabilizer in two consecutive rounds are connected with a new edge, which represents the potential measurement error.

b) Error Pattern & Syndromes: When an independent error source in a code experiences an error, it will "flip" the measurement outcome of the two adjacent stabilizers. Because a stabilizer is adjacent to multiple independent error sources, its measurement outcome is determined by the parity of the number of erroneous sources: Only if an odd number of adjacent sources experience error, the stabilizer will have a defect measurement outcome.

Because E_M denote the set of independent error sources in the code, $\mathcal{E} \subseteq E_M$ denotes the subset that experience an error, or *error pattern*. $P(\mathcal{E}), \forall \mathcal{E} \subseteq E_M$ indicates the probability that $\mathcal{E}$ happens. It is the *error model* of the code and can be obtained by characterizing the quantum hardware.

One can compute the error model from the error model for each independent error source, $P(e)$, as below:

$$P(\mathcal{E}) = \prod_{e\in\mathcal{E}} P(e) \prod_{f\in E_M\setminus\mathcal{E}} (1-P(f)) \propto \prod_{e\in\mathcal{E}} \frac{P(e)}{(1-P(e))} \quad (1)$$

Given an error pattern $\mathcal{E}$, $S(\mathcal{E})$ denotes the set of vertices that correspond to the defect measurement outcomes in the decoding graph and it is known as the *syndrome* of $\mathcal{E}$.

Given the syndrome $\mathcal{S}$ and the model graph, a decoder seeks to find an error pattern that produces $\mathcal{S}$. The *decoding graph* is the model graph with the syndrome $\mathcal{S}$ marked, as shown in Fig. 1(3). The Union-Find decoder [13] uses the decoding graph [9] to find an error pattern that can produce the syndrome.

c) Most-Likely Error Decoder: A Most-Likely Error (MLE) decoder tries to find the most likely error pattern that generates the syndrome S.

$$\arg\max_{\mathcal{E}|S(\mathcal{E})=\mathcal{S}} P(\mathcal{E}) = \arg\max_{\mathcal{E}|S(\mathcal{E})=\mathcal{S}} \prod_{e\in\mathcal{E}} \frac{P(e)}{1-P(e)}$$

The MLE decoding problem then becomes a problem for the decoding graph: find a subset of edges $\mathcal{E} \subseteq E_M$ that generates the observed syndrome $S(\mathcal{E}) = \mathcal{S}$ while *maximizing* $P(\mathcal{E}) = \prod_{e\in\mathcal{E}} \frac{P(e)}{(1-P(e))}$. Note that we use $\mathcal{E}$ as error pattern and subset of edges interchangeably because they represent the same thing. Since it's more common to define the summation of weights in graph problems, we can equivalently translate the problem into *minimizing* $W(\mathcal{E}) = \sum_{e\in\mathcal{E}} w_e$.

$$\arg\min_{\mathcal{E}|S(\mathcal{E})=\mathcal{S}} \sum_{e\in\mathcal{E}} w_e$$

d) MWPM Decoder: The Minimum-Weight Perfect Matching (MWPM) decoder is an exact MLE decoder when the error model can be precisely represented by a model graph. Unlike the Union-Find decoder, the MWPM decoder uses the *syndrome graph*, $G(V, E)$, which is generated from the decoding graph by creating an edge between any two defect vertices and removing normal vertices $v \in V_M \setminus \mathcal{S}$ (and their incident edges). That is, $V = \mathcal{S}$ and $E = \{(u, v)|\exists u, v \in \mathcal{S}\}$. The weight of an edge in the syndrome graph is calculated as that of a minimum-weight path between them. As its name suggests, the MWPM decoder finds an MLE error pattern by finding a minimum-weight perfect matching for the syndrome graph. We illustrate the workflow of the MWPM decoder in Fig. 1.

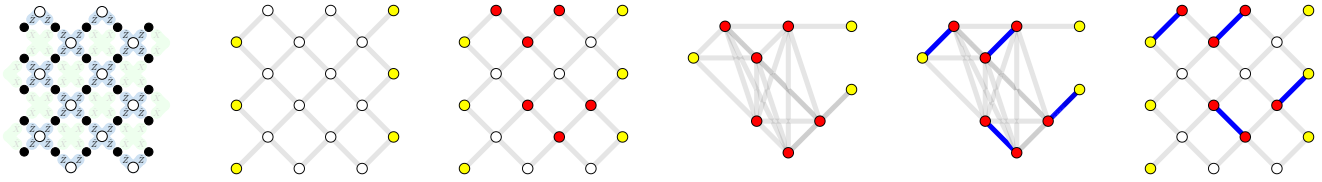
Because the fastest known algorithm to solve the MWPM problem for a general graph is the blossom algorithm [14], most implementations of the MWPM decoder use off-the-shelf MWPM libraries such as Kolmogorov's blossom V library [15] and the Lemon library by Dezső *et al* [16]. These implementations must go through all the stages in Fig. 1. Because the syndrome graph is complete, i.e, $|E| = |V|^2$, even the fastest implementations known [17, 18] have a time complexity of $O(\sqrt{|V|}|E|) = O(|V|^{2.5})$, scaling faster than the number of defect stabilizer measurements $|V|$.

The key idea behind Parity Blossom is that it removes the stage of building the syndrome graph. As a result, Parity Blossom reaches an average runtime of almost $O(|V|)$ given sufficiently low error rate. In doing so, unlike the blossom algorithm, Parity Blossom does not work for general graphs but decoding graphs representing syndromes of QEC codes. We derive this insight from our prior work [9], which shows that the UF decoder can be considered as an approximation of the MWPM decoder. Like the UF decoder, Parity Blossom uses the decoding graph.

B. Blossom Algorithm in General

We next describe the blossom algorithm [14]. We elide details that are irrelevant to our contributions. The blossom algorithm formulate the MWPM problem as an integer linear-programming (ILP) problem. Given any graph $G = (V, E)$ and edge weights $w_e, \forall e \in E$, the MWPM problem solves a perfect matching $x_e, \forall e \in E$ with minimum total weight $\sum_{e\in E} w_e x_e$. A solution is represented by $x_e, \forall e \in E$, with all selected edges $x_e = 1$ and others $x_e = 0$. A perfect matching requires that for every vertex $v \in V$, there is a unique edge with $x_e = 1$ incident to v, and all other incident edges have $x_e = 0$. There is no constraint on the incident edges for a virtual vertex.

The blossom algorithm solves the above ILP problem by first relaxing the integer constraint, becoming a linear-programming (LP) problem. It then adds some more con-



(1) Surface Code (2) Model Graph (3) Decoding Graph (4) Syndrome Graph (5) MWPM (6) Most-Likely Error

Figure 1: The procedure of an MWPM decoder. (2) Create model graph given the code and noise model. Yellow vertices are virtual boundaries. (3) Create decoding graph given model graph and the observed syndrome. The white (red) vertices corresponds to normal (defect) measurement result. (3) Solve an MWPM on the syndrome graph. Blue edges are selected in the MWPM. (4) Translate MWPM into a subset of edges in the decoding graph, which corresponds to a most-likely error.

straints to the LP problem so that all optimal ILP solutions are optimal LP solutions [19]. It solves the following LP problem.

$$\begin{aligned} \min \quad & \sum_{e\in E} w_e x_e && && (1)\\ \text{subject to} \quad & \sum_{e\in\delta(v)} x_e = 1 && \forall v \in V && (1a)\\ & \sum_{e\in\delta(S)} x_e \geqslant 1 && \forall S \in \mathcal{O} && (1b)\\ & x_e \geqslant 0 && \forall e \in E && (1c) \end{aligned}$$

where $\mathcal{O} = \{S | S \subseteq V \wedge |S| > 1 \wedge |S| = 1 \mod 2\}$ and $\delta(S) = \{e | e = (u, v) \in E \wedge ((u \in S \wedge v \notin S) \vee (u \notin S \wedge v \in S))\}$. $e \in \delta(S)$ is called a *hair* of S and has one and only one incident vertex inside S.

The blossom algorithm creatively exploits the *dual* formulation of the same problem.

$$\begin{aligned} \max \quad & \sum_{v\in V} y_v + \sum_{S\in\mathcal{O}} y_S && && (2)\\ \text{subject to} \quad & w_e - \sum_{v\in e} y_v - \sum_{S\in\mathcal{O}|e\in\delta(S)} y_S \geqslant 0 && \forall e \in E && (2a)\\ & y_S \geqslant 0 && \forall S \in \mathcal{O} && (2b) \end{aligned}$$

Definition: Tight Edge. For edge $e \in E$, we say it is tight when $w_e = \sum_{S\in\mathcal{O}^*|e\in\delta(S)} y_S$, where $\mathcal{O}^* = \{S | S \subseteq V \wedge |S| = 1 \mod 2\}$.

If an edge e is not tight, its primal x_e must be zero, thanks to the Complementary Slackness theorem. This means that the final solution to the primal problem only includes tight edges.

Definition: Blossom. In the blossom algorithm, $S \in \mathcal{O}$ is a blossom if and only if $y_S > 0$ [15]. Blossoms are defined inductively as below.

1. An odd number of vertices connected in a circle by tight edges form a blossom.
2. An odd number of vertices or blossoms connected in a circle by tight edges form a blossom.

Definition: Node. A blossom or vertex that is not the child of any other blossom is called a node. A node also includes an odd number of vertices. We denote it with the set of its vertices.

1) Blossom algorithm: The blossom algorithm organizes tight edges and nodes in *alternating trees* and matched pairs. The matching solution includes alternating edges in alternating trees and those between matched pairs. A matched pair represents an MWPM solution for the vertices included by the two nodes.

The blossom algorithm starts with an empty matching solution that is feasible to the dual problem and evolves it toward a feasible solution to the primal problem while maintaining the dual feasibility. It terminates when there is no alternating tree: all nodes are in matched pairs.

The primal phase seeks to increase the number of edges in the matching solution: it does so by updating the alternative trees. It computes a vector $\Delta\vec{y}$ of which an element $\Delta y_S \in \{0, +1, -1\}$ is the update for y_S of node S. $\Delta\vec{y}$ is essentially the *direction of update* of the dual variables corresponding to nodes. Because Parity Blossom does not innovate in the primal phase, we refer readers to [15] for details of the primal phase.

Definition: Direction. $\Delta\vec{y}$, the direction of updating $\vec{y}$.

The dual phase seeks to update the dual variables y_S along the direction $\Delta\vec{y}$ computed in the primal phase. In doing so, it must maintain the dual feasibility. That is, it ensures constraints (2a) and (2b) are always true.

Definition: Obstacle. An obstacle is a dual constraint (2a or 2b) that updating a dual variable according to $\Delta\vec{y}$ may violate.

A key job of the dual phase is to detect obstacles and stop before it violates any of the dual constraints. When the dual phase detects an obstacle, it stops after reporting the detected obstacles to the primal phase. Existing implementations of the blossom algorithm detect obstacles using the syndrome graph. A key idea of Parity Blossom and Sparse Blossom [3] is to do it using the decoding graph.

C. Blossom Algorithm in QEC

We show some special properties of the blossom algorithm when it solves the MWPM problem for a syndrome graph.

Theorem: Non-negative Vertex Dual. Given the error probability for any independent error source $P(e) \leqslant 0.5$, dual

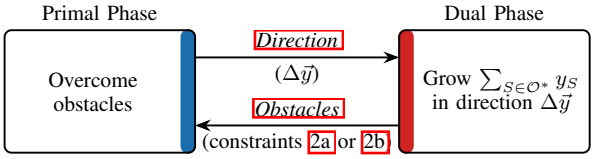


Figure 2: High-level structure of the blossom algorithm

variables y_v, $\forall v \in V$ are non-negative during the whole process of finding the solution by the blossom algorithm.

With **Theorem: Non-negative Vertex Dual**, we can simplify the LP problem for QEC decoding as follows. We define a set that includes both blossoms and single vertices $\mathcal{O}^* = \{S | S \subseteq V, |S| = 1 \mod 2\}$.

$$\max \quad \sum_{S\in\mathcal{O}^*} y_S \tag{3}$$

$$\text{subject to} \quad w_e - \sum_{S\in\mathcal{O}^*|e\in\delta(S)} y_S \geqslant 0 \qquad \forall e \in E \tag{3a}$$

$$y_S \geqslant 0 \qquad \forall S \in \mathcal{O}^* \tag{3b}$$

With the above simplification, the blossom algorithm can be simplified in description so that a single vertex can be treated as a blossom. As a result, the inductive defintion of blossoms is reduced to:

1. A single vertex is a blossom.
2. An odd number of blossoms connected in a circle by tight edges form a blossom.

As a result, we will use blossom to refer to both vertices and proper blossoms when discussing the blossom algorithm in the rest of the paper.

III. Geometric Interpretation

Before describing our decoder designs, we provide their mathematical foundation, which is based on a novel geometric interpretation of the blossom algorithm working on the decoding graph, inspired by [9]. By linking the notion of blossom with a geometric object (Cover) on the decoding graph, we prove key theorems used by our design. Because this interpretation relies on the non-negative weights and non-negative dual variables given in Eq. 3b, it is not applicable to the blossom algorithm working on general graphs.

A. Geometry of Decoding Graph

The geometric interpretation is based on viewing an edge $e = (u, v) \in E_M$ as a straight, continuous line of length $w_e \geqslant 0$. This line consists of points, which do not include u and v. We do consider vertices as points; we say u and v are incident to e, not on e, i.e., $u, v \notin e$. Under the above interpretation, the entire decoding graph comprises of points.

When $w_e > 0$, $\forall p \in e$, p partitions the line into two segments (u, p), of length $w_{(u,p)}$, and (p, v), of length $w_{(p,v)}$. We have $w_{(u,p)} > 0$, $w_{(p,v)} > 0$, and $w_{(u,p)} + w_{(p,v)} = w_e$. We can conveniently view that $p \in e$ breaks e into two edges, each with its own weight. We call these edges *segment edges* when it is necessary to distinguish them from $e \in E_M$. Similarly, two points on e would break it into three segment edges whose weights can be similarly determined and are positive. With this, we can extend the notion of path to two arbitrary points p and q as the set of edges connecting them, some of which are segment edges.

When $w_e = 0$, we call it a zero edge and $e = \varnothing$. Zero edges are necessary for decoding erasure errors [20, 21]. Note that segment edges by definition always have a positive weight.

Definition: Distance. We define the *distance* between two vertices $u, v \in V_M$ on the decoding graph as the weight of a minimum-weight path between them, noted as $\text{Dist}(u, v)$. This definition of distance can be readily extended for two arbitrary points p and q of the decoding graph, according to the path definition above.

A point r on a minimum-weight path between p and q partitions the path into two paths, one between p and r and the other between r and q. They are also the minimum-weight paths between p and r and between r and q, respectively. And we have $\text{Dist}(p, r) + \text{Dist}(r, q) = \text{Dist}(p, q)$.

We note that minimum-weight paths between vertices in the decoding graph are related to the edges in the syndrome graph: the edge weight between two vertices in the syndrome graph is the same as the weight of a minimum-weight path between the corresponding vertices in the decoding graph. The notions of point and distance are key for understanding how the blossom algorithm can be adapted to work on the decoding graph.

Definition: Circle. A circle of radius of d around $v \in V_M$, $C(v, d)$, is defined as the set of points whose distance from v is no greater than d. That is, $C(v, d) = \{p | \text{Dist}(p, v) \leqslant d\}$.

A Circle consists of boundary and inside. The boundary of $C(v, d)$ is simply $\{p | \text{Dist}(p, v) = d\}$. Likewise, the inside $C(v, d)$ is simply $\{p | \text{Dist}(p, v) < d\}$.

B. Blossom on Decoding Graph

We relate dual variables in the blossom algorithm to the geometric objects of the decoding graph.

1) Geometric Notions: The inductive definition of *Blossom* allows a tree representation of a blossom: The blossom is the root; the children of the root are also blossoms; a child can also have its own children and so on. We call the set of vertices and blossoms represented by the (grand)children of this tree the *descendants* of the root blossom.

Definition: Progeny. Given a blossom S, $\mathcal{D}(S)$ is the set that includes S itself and all its descendants. We call it the *progeny* of S. Furthermore, we define

- $\mathcal{D}_v(S) = \{D | D \in \mathcal{D}(S) \wedge v \in D\}$ consists of the members of S's progeny that include vertex v.
- $\mathcal{D}_{u\setminus v}(S) = \{D | D \in \mathcal{D}(S) \wedge u \in D \wedge v \notin D\}$ is the set of progeny members of S that includes u but not v.

Definition: Ancestry. Given a vertex $v \in V$ in the syndrome graph, its *ancestry* $\mathcal{A}(v)$ is the set of all blossoms that include v. Let $\mathcal{A}(u\setminus v)$ denote the subset of u's Ancestry whose members do not include vertex v.

Definition: Cover. Given a blossom S, it covers the set of points defined by the union of circles centered at $\forall v \in S$ with $d = \sum_{D \in \mathcal{D}_v(S)} y_D$. That is,

$$\text{Cover}(S) = \cup_{v \in S} C(v, \sum_{D \in \mathcal{D}_v(S)} y_D).$$

That is, $\text{Cover}(S)$ consists of Circles around $\forall v \in S$. The *boundary* of a Cover consists of points of the Cover that are not inside any of its Circles. Because a Circle consists of a finite number of edges, $\text{Cover}(S)$ also consists of a finite number of edges. For a blossom of a single vertex v, its Cover is simply $\text{Cover}(v) = C(v, y_v)$.

We emphasize that blossoms are defined on the syndrome graph while their Covers are defined on the decoding graph. As a result, the notion of Cover is an important bridge between the decoding and syndrome graphs.

2) Obstacle Detection on Decoding Graph: Because the dual phase detects obstacles based on the syndrome graph, we must find a way to do so on the decoding graph. The key insight and theoretical result of this work is the next theorem, which show exactly how to do it.

First of all, we note that detecting obstacles from dual constraints 2b is independent from the choice of syndrome vs. decoding graphs. Therefore, we only need to focus on those from dual constraints 2a. Second, because obstacles only occur on edges between different *Nodes*, it only needs to watch them to detect obstacles. Formally, we have

Theorem: Tight Edge Detection (Cover). There exists a tight edge between two different nodes S_1 and S_2 if and only if $\text{Cover}(S_1)$ and Cover (S_2) overlap. That is,

$$\exists e = (v_1, v_2) \in E, v_1 \in S_1 \ \wedge \ v_2 \in S_2 \ \wedge \ w_e = \sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S$$
$$\iff \text{Cover}(S_1) \cap \text{Cover}(S_2) \neq \varnothing$$

An obstacle of 2a is detected if such a tight edge exists and $\Delta y_{S_1} + \Delta y_{S_2} > 0$. That is, it can be detected by examining Covers of nodes on the decoding graphs.

C. Parity Blossom

The key idea of Parity Blossom, as well as Sparse Blossom [3], is to detect obstacles using the decoding graph, leveraging the result of **Theorem: Tight Edge Detection (Cover)**. Therefore, Parity Blossom, like Sparse Blossom, uses the existing design of the primal phase, e.g, that of Blossom V [15]. Only in the dual phase, they eschew the use of the syndrome graph. We will describe our implementation of Parity Blossom in §V.

Using the decoding graph to detect obstacles is more advantageous than the syndrome graph given a low physical error rate $p \ll 1$. As explained in §II-A, generating the syndrome graph itself already takes quadratic time $O(|V|^2)$. On the decoding graph, however, large *Covers* are exponentially unlikely with its size, so the average time complexity scales with roughly $O(|V|)$. Note that when $|V|$ is small or when p is large, it might be faster to use the syndrome graph.

IV. Fusion Blossom

We next describe a parallel algorithm of solving the MWPM problem for QEC, called *Fusion Blossom*. Fusion Blossom recursively divides a decoding problem into sub-problems that can be solved independently and then recursively "fuses" their solutions to produce the solution to the original problem. We represent this recursive division/fusion as a full binary tree, called a *fusion tree*. Every leaf in the fusion tree invokes an MWPM solver, while other nodes fuse the solutions from their two children, also leveraging the MWPM solver. In our implementation, the MWPM solver is Parity Blossom.

We next provide the mathematical formulation of division and fusion in §IV-A and §IV-B, respectively. We discuss how Fusion Blossom can make tradeoffs between decoding time and latency in §IV-C.

A. Division

As illustrated by Fig. 3(1), a carefully selected set of vertices $V_b \subset V_M$, e.g., a minimum vertex cut [22] of the decoding graph, can divide a decoding graph into two disjoint graphs that include V_1 and V_2, respectively. The only requirement of V_b is that, there is no edge in the decoding graph that connects vertices from both V_1 and V_2.

With V_b, we can create two sub-problems, one working on the subgraph covering vertices $V_1 \cup V_b$ and the other that covering $V_2 \cup V_b$, as illustrated by Figs. 3(2) to 3(4). Each of the sub-problems treats a vertex from V_b as a virtual vertex that can be matched arbitrary times. This effectively relaxes the parity constraints on vertices V_b in the sub-problems, which will be tightened later by fusion. For the i-th sub-problem, $i \in \{1, 2\}$, the primal and dual formulations as in Eq. 1 and 3, respectively, have E and $\mathcal{O}^*$ as follows.

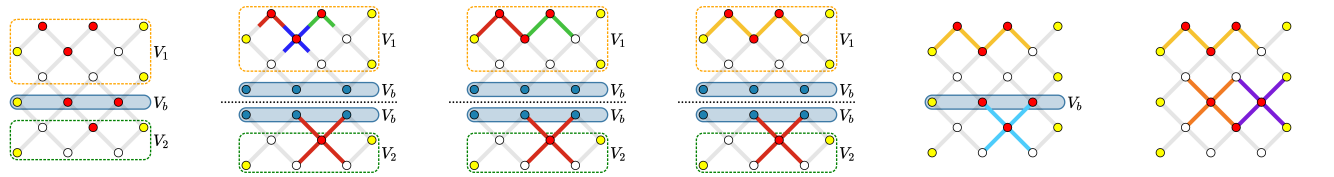
$$E_i = \{e | e = (u, v) \in E \ \wedge \ u, v \in V_i \cup V_b\}$$
$$\mathcal{O}_i^* = \{S | S \in \mathcal{O}^* \ \wedge \ S \subseteq V_i\}$$

The process of division stops when the subproblem is adequately small for invoking the MWPM solver directly. We call such subproblems *leaf problems* and the corresponding subgraphs *leaf partitions*.

B. Fusion

After the sub-problems are solved independently, their solutions form an intermediate state for the original problem in terms of the values of the primal and dual variables. The fusion operation invokes the MWPM solver to find a solution to the original problem starting with this intermediate state.

a) Correctness: We next show that the intermediate state is indeed a valid state for the blossom algorithm. For the primal variables, we remove the matchings to the temporary boundary vertices V_b. Those matched pairs break into alternating trees and search for new matchings. Except for those, the matchings within V_1 or V_2 are preserved. We also create an alternating tree for each defect vertex in V_b. We simply keep the existing dual variables, as shown in Fig. 3(4) to Fig. 3(5), given



(1) Partition (2) Individual Step 1 (3) Individual Step 2 (4) Individual Step 3 (5) Recover V_b (6) Fusion Completed

Figure 3: Fusion Blossom example. The two sub-problems solves their local MWPM (2-4) individually, in parallel. The fusion operation first (5) recovers the temporary boundary vertices and then (6) evolves the intermediate state to a global MWPM.

Theorem: Feasible Dual Variables. Solutions for the two disjoint sub-problems determine the values of the dual variables $y_S, S \in \mathcal{O}_1^* \cup \mathcal{O}_2^*$. These values plus setting y_S to 0 for $S \in \mathcal{O}^* \setminus \mathcal{O}_1^* \cup \mathcal{O}_2^*$ constitute a feasible solution to the original dual problem.

b) Speed: We estimate the average time complexity of fusion operation to be no worse than $O(p|V_b|)$ where $p \ll 1$ is the physical error rate. Note the expected number of defect vertices in V_b is also $O(p|V_b|)$. Our estimate is based on two intuitions. First, the fusion operation only needs to break about the same number of matched pairs from the sub-problem solution to match the defect vertices in V_b. Second, due to the objective of minimum weight, it is more likely to find matches for these vertices close to V_b. We note this estimate is independent of the size of the sub-problems $|V_1|$ and $|V_2|$. We confirm this independence empirically in §VI-B3.

C. Schedule Design: Leaf Partitions and Fusion Tree

When the leaf partitions are properly chosen, there can be multiple ways to fuse their solutions, allowing different tradeoffs between decoding time and latency. In this case, the fusion tree defines the space for scheduling leaf and fusion operations. One particularly relevant case is illustrated in Fig. 4, where the decoding graph is a stream of measurement rounds, each a two-dimensional graph. In this case, a leaf partition is simply a subgraph that consists of M consecutive measurement rounds.

When the measurement rounds of a leaf partition become available, it invokes the MWPM solver to produce a solution. In an online system, as the measurement rounds stream in, the leaf partitions finish one by one. However, there are many ways in which their solutions can be fused, with three examples shown in Fig. 5, each making a different trade-off between decoding time and latency. As illustrated in Fig. 4, we define

- Decoding Time: T, the time from when decoding starts to when it finishes.
- Latency: L, the time from when all measurements are ready to when decoding finishes.
- Measurement Rounds: N, the number of rounds of stabilizer measurements.
- Leaf Partition Size: M, the number of measurement rounds in each leaf partition.

We note the throughput of the system is related to decoding time as N/T: how many rounds it can decode per unit time.

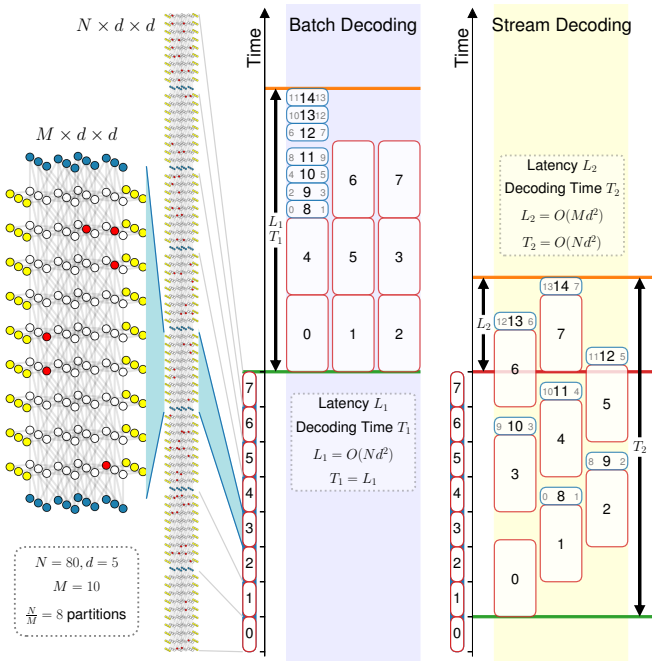


Figure 4: A 3D decoding graph (left), its Batch (center) and Stream (right) Decoding using three CPU cores. The Batch decoding starts when all $N = 80$ rounds of measurements are ready; it fuses solutions according to the Balanced tree in Fig. 5. The Stream decoding starts decoding whenever $M = 10$ rounds of measurements for a leaf node are ready; it fuses solutions according to the Linear tree in Fig. 5.

Batch Decoding. Prior studies generally assumed that the syndrome of all N rounds of measurement is available at the time of decoding, which is known as batch decoding (see Fig. 4 (center)). For batch decoding, the decoding latency and time are the same, i.e., $L = T$, and the decoding time is determined by the longest path from a leaf to the root given enough parallel resources. Therefore, the *balanced tree*, as shown in Fig. 5 (left), is preferable since its longest path (from leaf to root) is the shortest.

Stream Decoding. In contrast to batch decoding, stream decoding starts as soon as enough rounds of measurement are ready for a leaf node (see Fig. 4 (right)). As a result, the decoding latency can be substantially shorter than the decoding time, i.e., $L < T$. More importantly, to determine the decoding

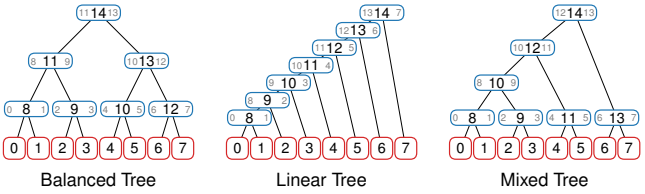


Figure 5: Fusion Trees. The leaves need to be fused recursively into a single root, which represents a global MWPM. Because a parent depends on the children, the paths between leaves and the root determine the decoding time and latency.

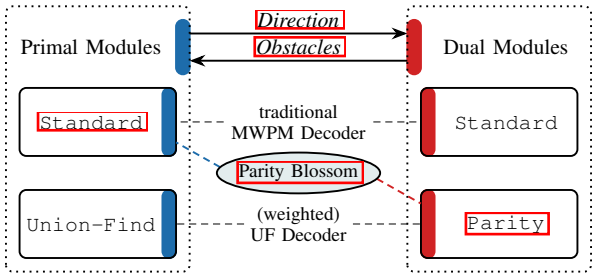


Figure 6: Unified Framework for Matching Decoders.

latency, one can no longer simply consider paths between leaves and the root but must add the time when the rounds of measurement for a leaf is ready (assuming those for the first leaf is ready at time zero). To minimize the decoding latency, one must balance the path length plus the ready time for all paths, allowing a shorter path for a later leaf. For example, when there are enough parallel resources that are fast enough, the path between the last leaf and the root determines the decoding latency. In this case, the *linear tree* (Fig. 5 (center)) is preferable.

With the balanced and linear trees as the two extreme cases in mind, we can create a continuum of trees between them called *mixed trees*. Given the parallel resources and decoding setup, e.g., M and N, one must examine this continuum to find the tree that achieves the shortest decoding latency. To construct a mixed tree, one selects a height in the balanced tree, keeps balanced sub-trees below the height but constructs a linear tree above it. The higher this height, the smaller path difference between earlier and later leaves. For the balanced and linear trees, this mix height is root and leaf, respectively. Fig. 5 (right) shows the mixed tree with the mix height of one. In our latency evaluation (§VI-B2), we use the mixed tree that minimizes the decoding latency.

We note that the mix height can be determined dynamically: the decoder can start with a balanced tree and switch to a linear tree to optimize the performance of the system.

V. Implementation

We next describe our implementation of Parity Blossom and Fusion Blossom, including major ideas for optimizations. We implemented these algorithms in Rust with 12k lines of code.

A. Unified Framework for Matching Decoders

A key idea behind our implementation is to use a unified framework for the blossom algorithm and its variants, including Parity Blossom, Union-Find [13], and more, as illustrated by Fig. 6. As the blossom algorithm iterates between the primal and dual phases as shown in Fig. 2, our unified framework implements them in modules with a narrow, well-defined interface with each other, marked as red and blue in Figs. 2 and 6. The interface allows any primal module to work with any dual module.

This framework serves three purposes. (*i*) First, it shows how these variants and the blossom algorithm are related. (*ii*) Second, it allows code reuse between their implementations. For example, Parity Blossom optimizes the dual module (`Parity`) to work on the decoding graph instead of the syndrome graph (§V-B1). It uses the same primal module (`Standard`) as the original blossom algorithm, with slight modification to support virtual vertices (§V-B2). For another example, the UF decoder and Parity Blossom share the same `Parity` dual module. The UF decoder uses its own primal module (`Union-Find`) that computes the direction approximately, compared to the blossom algorithm [9]. (*iii*) Third, this framework reveals the existence of previously unknown variants that can achieve different trade-offs between accuracy and speed when applied to QEC. It also allows such new variants to be easily implemented. For example, one can design a new primal module based on the `Standard` primal module, which sets a limit to the size of alternating trees. Once an alternating tree reaches the size limit, the module treats it as an invalid cluster like the `Union-Find` primal module. When the size limit is infinite, the decoder is identical to Parity Blossom; when the limit is zero, it is identical to the UF decoder. As a result, by adjusting the size limit, we can produce a continuum of decoders between the UF decoder and Parity Blossom, making different tradeoffs between decoding accuracy and speed.

B. Parity Blossom

As shown in Fig. 6, Parity Blossom uses the `Parity` Dual Module and the `Standard` Primal Module.

1) `Parity` *Dual Module:* Because Parity Blossom uses the decoding graph to detect *Obstacles*, given **Theorem: Tight Edge Detection (Cover)**, the `Parity` dual module must efficiently track the Covers of nodes.

Our first implementation idea comes from the UF decoder [23]: we maintain the boundary edges for each Cover. This is efficient and sufficient for updating Covers because when dual variables are adjusted according to $\Delta\vec{y}$, the boundary edges of their Covers change.

Our second idea removes implementation complication resulting from the fact that some vertices from the decoding graph may belong to multiple Covers. Zero edges, representing erasure errors [20, 21], specifically contribute to this complication because their vertices can belong to many Covers. To ensure that a vertex belongs to at most one "Cover", our idea is to use Pseudo-Covers, derived from Covers as follows.

Definition: Pseudo-Cover. All Covers with a single vertex are Pseudo-Cover. That is, when the blossom algorithm starts, all the Covers on the decoding graph are Pseudo-Covers, each of them with a single defect vertex. (*i*) At the beginning of a dual phase, for a node with $\Delta y_S < 0$, its Pseudo-Cover is derived by removing all boundary non-defect vertices. (*ii*) The Pseudo-Cover for a node S with $\Delta y_S > 0$ is derived by modifying how its Cover grows. When adding a vertex to a growing Pseudo-Cover, the growing stops if the vertex is already inside another Pseudo-Cover. We denote the Pseudo-Cover of $\text{Cover}(S)$ with $\overline{\text{Cover}}(S)$.

Since a vertex belongs to at most one $\overline{\text{Cover}}$, its memory usage is constant. Also, since the incident vertices of an edge $e = (u, v)$ each belongs to at most one $\overline{\text{Cover}}$, there are at most two covered segment edges on e. That is, the memory usage of an edge is also constant. The next theorem says that Pseudo-Covers can also be used to detect tight edges.

Theorem: Tight Edge Detection (Pseudo-Cover). There exists a tight edge between two different nodes S_1 and S_2 with $\Delta y_{S_1} + \Delta y_{S_2} > 0$ if and only if there exists two different nodes S_3 and S_4 with $\Delta y_{S_3} + \Delta y_{S_4} > 0$ whose Pseudo-Covers meet on a decoding graph edge. That is,

$$\begin{aligned} &\exists S_1, S_2, e = (v_1, v_2) \in E, \\ &v_1 \in S_1, v_2 \in S_2, \Delta y_{S_1} + \Delta y_{S_2} > 0, w_e = \sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S \\ \iff \quad &\exists S_3, S_4, e' = (v_3, v_4) \in E_M, \\ &v_3 \in \overline{\text{Cover}}(S_3), v_4 \in \overline{\text{Cover}}(S_4), \\ &\Delta y_{S_3} + \Delta y_{S_4} > 0, e' \subseteq \overline{\text{Cover}}(S_3) \cup \overline{\text{Cover}}(S_4) \end{aligned}$$

Using this theorem, the `Parity` dual module checks whether such S_3 and S_4 exists for each decoding graph edge and reports all detected obstacles to the primal module. We note that the above theorem differs from **Theorem: Tight Edge Detection (Cover)** in a fundamental way: it does not detect all obstacles but at most one for each decoding graph edge.

We have not implemented an important optimization used by Sparse Blossom [3] and Blossom V [15]: they sort and process edges based on when they will become tight using priority queues. Instead, our dual module implementation enumerate all edges, like the UF decoder [23].

2) `Standard` *Primal Module:* We base the `Standard` primal module on the implementation of blossom V [15] with three optimizations. We note these optimizations can be generally useful beyond Parity Blossom.

First, it supports virtual vertices. Prior works, using the blossom V library, had to emulate them, which is inefficient.

Second, each time the dual module is invoked, it prepares Pseudo Covers based on the directions of their blossoms, which incurs significant overhead. To amortize this overhead, our `Standard` primal module handle all reported obstacles each time it is invoked, instead of returning to the dual module after handling one.

Third, we grow all alternating trees simultaneously like the UF decoder and Sparse Blossom [3]. It corresponds to "the multiple-tree approach with fixed δ" reported in [15] and applied by the blossom V library to only 5% of the nodes. As shown by the authors of Sparse Blossom, this approach explores fewer edges on average and thus is faster.

C. Fusion Blossom

We implement Fusion Blossom with Parity Blossom as the MWPM solver, and using the Rayon parallel programming library [24]. A manager thread reads the fusion tree from leaves up. It creates a job for each node in the fusion tree when the jobs for the children nodes have returned. The manager thread inserts new jobs into a queue from which a group of worker threads remove jobs and complete them. A job invokes Parity Blossom implemented in the unified framework as the MWPM solver.

During the fusion operation, the MWPM solver is invoked to evolve the intermediate state to an optimum for the fused problem. A naïve implementation would construct the internal data structures for solving the fused problem from the output of solving the children sub-problems, leading to excessive memory copying. Instead, our implementation allows the MWPM solver invoked by a parent to reuse, i.e., operate directly on, the internal data structures of the MWPM solver invoked by its child. As the system progresses from a leaf toward the root in the fusion tree, it maintains the internal data structures for the MWPM solver invoked by each node of the fusion tree, with those of a parent including those of its children.

Organizing the internal data structures hierarchically based on the fusion tree as described above brings an additional opportunity to optimize the MWPM solver when invoked by a parent for the fusion operation. Since the fusion operation only changes the *Pseudo-Covers* in a small region around the boundary vertices V_b, the MWPM solver ideally should only work on data structures related to this small region. We achieve this by allowing the MWPM solver invoked by the parent to "invoke" the MWPM solver of its child to evolve the child's internal data structures. This is possible because the child's internal data structures are maintained in place.

D. Other Optimizations

To improve performance, we implement an optional feature that leverages `unsafe` Rust to bypass safety checks when it is safe to do so at the algorithm level. By enabling the optional `dangerous_pointer` feature, it results in a speedup of approximately 2x compared to the standard build.

Since the initialization time scales with $|V|$ yet the decoding time scales with $p|V|$, it is more practical to reset the decoder between simulation shots rather than creating a new one. Our optimization achieves a reset time of $O(1)$. The key idea is to avoid enumerating all the decoding graph edges to reset the Pseudo Covers stored on them. In implementing this idea, the decoder keeps a global timestamp and each edge has its own timestamp, all initialized to 0. The global timestamp advances

by 1 on a global reset. An edge is invalid if its timestamp does not match the global one. Only when an invalid edge is being accessed, it is reset and its timestamp is updated to the global timestamp.

VI. Evaluation

We evaluate our implementations of Parity Blossom and Fusion Blossom with both macro and micro benchmarks. The evaluation answers the following questions.

- Correctness: Are they exact MWPM decoders?
- Throughput: How many rounds of measurement can be decoded per unit time?
- Latency: How long does it take from when the last round of measurement arrives to when decoding finishes?
- Scalability: How throughput changes with code distance?

We verify the correctness of our implementations by comparing against the blossom V library [15] over millions of randomized test cases with tractable code distances up to 19. We focus on throughput, latency, and scalability in the rest of this section.

A. Setup

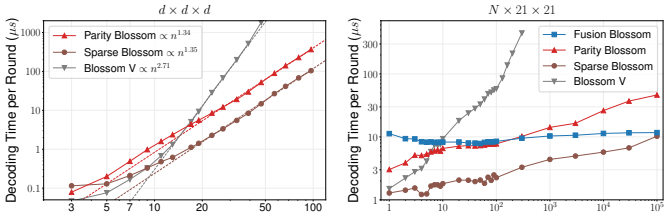
1) Noise Model: We use the circuit-level noise model [25] with a physical error rate of 0.1%. We use a rotated surface code shown in Fig. 1(1). It has $n = d^2$ data qubits and $(d^2-1)/2$ Z (X) stabilizers. Given a syndrome of N noisy rounds of measurement, the Z (X) decoding graph has $(N+1)(d^2-1)/2$ ordinary vertices and $(N+1)(d+1)$ virtual vertices, a total of $(N+1)(d+1)^2/2$. Since the X and Z decoding graphs can be decoded independently, we only use the Z decoding graph for evaluation. For simplicity, we use format $N \times d \times d$ to represent the code.

2) Measurement: We evaluate the decoding speed on a Linux server with dual Intel Xeon Platinum 8375C CPUs, a total of 64 cores, each supporting two hyper-threads. The server is an M6i instance from AWS (Amazon Web Services). Our results do not include the initialization time, during which the one-time, expensive memory allocation is performed. Once initialized, the decoder works on 100 simulation shots consecutively. Between two shots, the decoder is reset with a constant overhead, which is included in the result. Each shot by default includes 10^5 rounds of measurement.

3) Baseline: For Sparse Blossom, we use the authors' own implementation through its Python binding [26] with the same setup as the above, with batch optimization enabled. For the traditional MWPM decoder, we use the blossom V library [15] with the following optimizations. It pre-computes a complete graph of V_M offline to reduce the runtime overhead of constructing the syndrome graph. It also eliminates edges in the complete graph if they have higher weight than the two vertices matching to virtual boundary respectively, because these edges would never be selected in an MWPM.

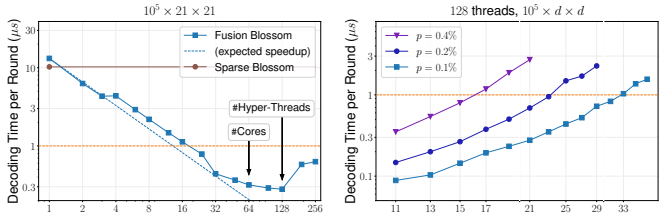
4) Metrics: Given the decoding time T and measurement rounds N, we define

- Throughput: N/T, decoded rounds per unit time.



(a) Code Distance d (b) Measurement Rounds N

Figure 7: Decoding time with a single thread (lower the better).



(a) Number of Threads (b) Code Distance d

Figure 8: Parallel decoding time (lower the better).

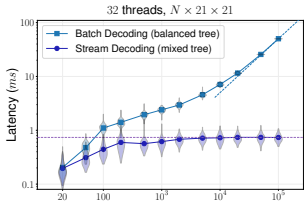
- Decoding time per measurement round: T/N, the inverse of throughput. Since this is easier to compare with the measurement cycle of a quantum hardware, we use it in lieu of throughput in the figures.

B. Results

1) Throughput: We use batch decoding in Fig. 4 for all throughput evaluations, assuming the syndrome is ready when the decoding begins. Instead of throughput, we report data in its inverse, i.e., decoding time per round (T/N).

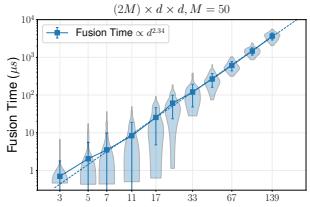
We first show the advantage of using the decoding graph over the syndrome graph, confirming the findings also reported by Higgott and Gidney in [3]. We benchmark the throughput on a single thread. As shown in Fig. 7(a), the decoding time of Parity Blossom and Sparse Blossom scales almost linearly with the number of qubits $n = d^2$, which is the theoretically lower bound. In contrast, the traditional MWPM decoder based on the blossom V library scales poorly with the number of qubits n. Not surprisingly, Parity Blossom is roughly 4x slower than Sparse Blossom in this case, because we have not incorporated some important optimizations (§V-B1).

Second, when the number of rounds N grows in Fig. 7(b), both Parity Blossom and Sparse Blossom see decoding time per round increases [3], due to increasing pressure on the memory hierarchy. Surprisingly, that of Fusion Blossom remains steady as N grows and beats that of Parity Blossom at large $N \geqslant 10^3$, despite that Fusion Blossom is not supposed to enjoy any algorithmic advantage over Parity Blossom with a single thread. This is because Fusion Blossom divides the problem equally into small ones and solving a small problem enjoys better cache locality. On the other hand, a small M incurs more fusion operations and more overhead. Therefore, we empirically find the optimal $M = 100$ and use it as the default leaf partition size.

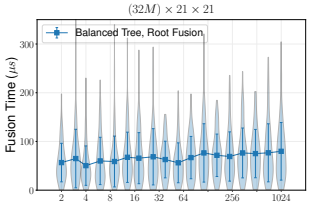


Measurement Rounds N

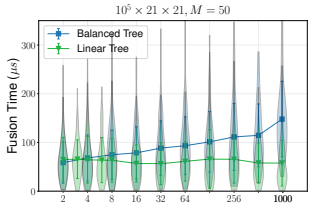
Figure 9: Latency



(a) Code Distance d



(b) Leaf Partition Size M



(c) Number of Leaf Partitions

Figure 10: Fusion time with a single thread (lower the better).

Not surprisingly, Fusion Blossom beats all serial MWPM decoders when more threads are available. As shown in Fig. 8(a), the throughput of Fusion Blossom increases almost linearly with the number of threads, until the maximum number of hyper-threads (128) supported by the processor. At that point, it reaches the minimum decoding time per round (0.3 us). Using all 128 hyper-threads in the processor, Fusion Blossom can decode up to $d = 33$ at $p = 0.1\%$ with less than 1 us decoding time per round, as shown in Fig. 8(b).

2) Latency: We observe a constant latency regardless of the measurement rounds N in the stream decoding (Fig. 4), compared to the linearly growing latency in the batch decoding. We emulate the stabilizer measurement cycle of 1 us, which is similar to that of state-of-the-art superconducting quantum hardware [27]. We use a mixed fusion tree in which each balanced subtree at the mix height has 50 leaves. Each leaf deals with $M = 20$ rounds of measurement. As shown in Fig. 9, the average latency is roughly 0.7 ms regardless of the number of measurement rounds N. For the batch decoding, as predicted in Fig. 4, the latency scales linearly with N.

3) Fusion Time: Given a low physical error rate, a fusion operation only changes a small region around the boundary vertices on average. Thus, the fusion time should only increase with $|V_b| = O(d^2)$, as confirmed by Fig. 10(a), but not M, the number of rounds, as confirmed by Fig. 10(b).

Moreover, because a fusion operation may recursively invoke the children's MWPM solvers until leaf partitions are reached, the structure of the corresponding subtree of the fusion tree impacts fusion time. Fig. 10(c) show this with fusion time for both balanced and linear trees where the X axis is the number of leaf partitions in the subtree.

Interestingly, the mean of fusion time of the balanced tree increases with the number of leaf partitions while that of the linear tree largely remains constant. This, again, is because a fusion operation only changes a small region around the boundary vertices on average. As a result, the operation is most likely to involve two leaf partitions next to each other and increasingly unlikely to involve partitions that are farther away from each other. In the balanced tree, the operation must travel through the entire height of tree to reach any two leaf partitions, even if they are next to each other. In contrast, in the linear tree, the operation is exponentially less likely to travel one level down the tree. For the example in Fig. 10(c) (center), fusion operation 14 is exponentially less likely to involve lower numbered leaf paritioned.

4) Scalability: Finally, we show that given enough ($\Omega(d^{2.68})$) parallel resources, e.g., cores, Fusion Blossom can meet the throughput requirement by any code distance d when the physical error rate p is well below the threshold p_{th}, using both analysis and empirical data.

Let K denote the number of threads, each handling a $(N/K) \times d \times d$ partition on its own core. Note that the threads may reside in different machines, accessing shared memory via network with a constant-factor slowdown, e.g., using shared-memory rack-scale distributed systems like [28]. The decoding time of each thread scales with $O(d^{2.68}N/K)$, according to Fig. 7(a). The solutions from the K concurrent threads can be fused with a balanced tree, with $O(d^{2.34}\log K)$ time, according to Fig. 10(a). The decoding time per round has a complexity of $O(d^{2.68}/K + d^{2.34}\log K/N)$. Thus, given a lower bound of $K = \Omega(d^{2.68})$ and $N = \Omega(d^{2.34}\log K)$, the decoding time per round will be bounded. This analysis assumes N polynomially grows with d. This is reasonable because the lifetime of a logical qubit scales exponentially with d, i.e., $\max N \propto (p_{\text{th}}/p)^{(d+1)/2}$ [2].

Note the scaling factors of $O(d^{2.68})$ and $O(d^{2.34})$ are empirically derived from code distances up to 100 (Figs. 7(a) and 10(a)), which corresponds to about 10^4 physical qubits for each logical qubit, orders of magnitude higher than what is considered to be practical in the near future.

We note that $K = \Omega(d^{2.68})$ does not mean $d^{2.68}$ threads (or cores) are necessary. When estimating how many cores are needed for a large d, one can empirically derive the number for a small d and then extrapolate based on the scaling of $d^{2.68}$. For example, to estimate how many cores are necessary to decode $d = 51$ with $p = 0.1\%$ using the setup in Section VI-A, one can pick a data point in Fig. 8(a) where $d = 21$ roughly needs 20 cores to meat the throughput requirement. We can estimate roughly $20 \times (51/21)^{2.68}$ or 216 cores are necessary for $d = 51$.

VII. Related Work

As mentioned in §I, Sparse Blossom is a contemporary work closely related to Parity Blossom, sharing the key idea of solving the MWPM problem using the decoding graph. We provide that first rigorous mathematical foundation for this idea and contribute new implementation optimizations.

Related to Fusion Blossom, Fowler [4] presented a parallel design of the MWPM decoder. It partitions the qubits to parallel decoding units of customized hardware. Each decoding

unit handles a sufficiently large number of qubits so that the inter-unit communication is relatively rare. Paradoxically, its success requires both a large number of decoding units (for lower decoding time) and a large number of qubits in each unit (for lower communication overhead). To our best knowledge, perhaps not surprisingly, no implementation or empirical data has been reported for this design. Fusion Blossom, on the other hand, eliminates communications between the partitions and only synchronizes them during the fusion operations. This minimizes the need for communication and is scalable.

There is a literature that seeks to parallelize solving the MWPM problem for general graphs. This literature, however, does not exploit the special structure of the QEC decoding problem as we do. As a result, its results have larger time complexity than Parity Blossom and Fusion Blossom when applied to QEC decoding. For example, Peterson and Karalekas [29] designed and implemented a distributed MWPM algorithm with $O(|V|^4)$ time complexity.

Recently there is a growing interest in approximate algorithms for QEC decoding that sacrifice decoding accuracy to gain speed, e.g., parallelization with parallel-window technique [6, 7, 30] and fast decoders with cryogenic chips [31–34]. Perhaps the most relevant is the (weighted) Union-Find (UF) decoder [23, 35] for which various design [5] and implementation [8] have been reported. The key idea of Parity Blossom draws inspiration from how the UF decoder approximates the MWPM decoder [9].

Acknowledgments

This work was supported in part by Yale University and NSF MRI Award #2216030. The authors are grateful for the insightful discussion with Shruti Puri.

References

[1] B. M. Terhal, "Quantum error correction for quantum memories," *Reviews of Modern Physics*, 2015.

[2] A. G. Fowler, A. C. Whiteside, and L. C. Hollenberg, "Towards practical classical processing for the surface code: Timing analysis," *Physical Review A*, 2012.

[3] O. Higgott and C. Gidney, "Sparse Blossom: correcting a million errors per core second with minimum-weight matching," *arXiv preprint arXiv:2303.15933*, 2023.

[4] A. G. Fowler, "Minimum weight perfect matching of fault-tolerant topological quantum error correction in average $o(1)$ parallel time," *arXiv preprint arXiv:1307.1740*, 2013.

[5] P. Das, C. A. Pattison, S. Manne, D. M. Carmean, K. M. Svore, M. Qureshi, and N. Delfosse, "AFS: Accurate, fast, and scalable error-decoding for fault-tolerant quantum computers," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022.

[6] L. Skoric, D. E. Browne, K. M. Barnes, N. I. Gillespie, and E. T. Campbell, "Parallel window decoding enables scalable fault tolerant quantum computation," *arXiv preprint arXiv:2209.08552*, 2022.

[7] X. Tan, F. Zhang, R. Chao, Y. Shi, and J. Chen, "Scalable surface code decoders with parallelization in time," *arXiv preprint arXiv:2209.09219*, 2022.

[8] N. Liyanage, Y. Wu, A. Deters, and L. Zhong, "Scalable quantum error correction for surface codes using fpga," *arXiv preprint arXiv:2301.08419*, 2023.

[9] Y. Wu, N. Liyanage, and L. Zhong, "An interpretation of union-find decoder on weighted graphs," *arXiv preprint arXiv:2211.03288*, 2022.

[10] "Fusion Blossom: a fast minimum-weight perfect matching (MWPM) solver for quantum error correction (QEC)." [Online]. Available: https://github.com/yale-paragon/fusion-blossom

[11] "Python binding of Fusion Blossom library." [Online]. Available: https://pypi.org/project/fusion-blossom

[12] A. G. Fowler, A. C. Whiteside, A. L. McInnes, and A. Rabbani, "Topological code autotune," *Physical Review X*, 2012.

[13] N. Delfosse and N. H. Nickerson, "Almost-linear time decoding algorithm for topological codes," *Quantum*, 2021.

[14] J. Edmonds and E. L. Johnson, "Matching, Euler tours and the Chinese postman," *Mathematical programming*, 1973.

[15] V. Kolmogorov, "Blossom V: a new implementation of a minimum cost perfect matching algorithm," *Mathematical Programming Computation*, 2009.

[16] B. Dezső, A. Jüttner, and P. Kovács, "LEMON–an open source C++ graph template library," *Electronic Notes in Theoretical Computer Science*, 2011.

[17] S. Micali and V. V. Vazirani, "An $O(\sqrt{|V|}|E|)$ algoithm for finding maximum matching in general graphs," in *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*. IEEE, 1980.

[18] V. V. Vazirani, "A simplification of the MV matching algorithm and its proof," *arXiv preprint arXiv:1210.4594*, 2012.

[19] W. Cook and A. Rohe, "Computing minimum-weight perfect matchings," *INFORMS journal on computing*, 1999.

[20] Y. Wu, S. Kolkowitz, S. Puri, and J. D. Thompson, "Erasure conversion for fault-tolerant quantum computing in alkaline earth rydberg atom arrays," *Nature communications*, 2022.

[21] J. D. Teoh, P. Winkel, H. K. Babla, B. J. Chapman, J. Claes, S. J. de Graaf, J. W. Garmon, W. D. Kalfus, Y. Lu, A. Maiti *et al.*, "Dual-rail encoding with superconducting cavities," *arXiv preprint arXiv:2212.12077*, 2022.

[22] D. B. West *et al.*, *Introduction to graph theory*. Prentice hall Upper Saddle River, 2001, vol. 2, p. 149.

[23] N. Delfosse and G. Zémor, "Linear-time maximum likelihood decoding of surface codes over the quantum erasure channel," *Physical Review Research*, vol. 2, no. 3, p. 033042, 2020.

[24] "Rayon: a data-parallelism library for Rust." [Online]. Available: https://github.com/rayon-rs/rayon

[25] A. J. Landahl, J. T. Anderson, and P. R. Rice, "Fault-tolerant quantum computing with color codes," *arXiv preprint arXiv:1108.5738*, 2011.

[26] O. Higgott and C. Gidney, "PyMatching v2." [Online]. Available: https://github.com/oscarhiggott/PyMatching

[27] "Suppressing quantum errors by scaling a surface code logical qubit," *Nature*, 2023.

[28] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "MIND: In-network memory management for disaggregated data centers," in *Proc. ACM SIGOPS Symposium on Operating Systems Principles*, 2021.

[29] E. C. Peterson and P. J. Karalekas, "A distributed blossom algorithm for minimum-weight perfect matching," *arXiv preprint arXiv:2210.14277*, 2022.

[30] H. Bombín, C. Dawson, Y.-H. Liu, N. Nickerson, F. Pastawski, and S. Roberts, "Modular decoding: parallelizable real-time decoding for quantum computers," *arXiv preprint arXiv:2303.04846*, 2023.

[31] A. Holmes, M. R. Jokar, G. Pasandi, Y. Ding, M. Pedram, and F. T. Chong, "NISQ+: Boosting quantum computing power by approximating quantum error correction," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020.

[32] Y. Ueno, M. Kondo, M. Tanaka, Y. Suzuki, and Y. Tabuchi, "QECOOL: On-line quantum error correction with a superconducting decoder for surface code," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021.

[33] ——, "QULATIS: A quantum error correction methodology toward lattice surgery," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022.

[34] G. S. Ravi, J. Baker, A. Fayyazi, S. Lin, A. Javadi-Abhari, M. Pedram, and F. Chong, "Better than worst-case decoding for quantum error correction," *Bulletin of the American Physical Society*, 2023.

[35] S. Huang, M. Newman, and K. R. Brown, "Fault-tolerant weighted union-find decoding on the toric code," *Physical Review A*, 2020.

APPENDIX A
FACTS & LEMMAS

a) Relevant facts about ***the decoding problem****.*

Fact 1. Edge weights are non-negative in a decoding graph. This is because $w_e = \log \frac{(1-P_e)}{P_e}$ given reasonable physical error rate of $P_e \leqslant 50\%$, $w_e \geq 0$. Consequently, edge weights in a syndrome graph are also non-negative.

We note that even if $P_e > 50\%$, this error is equivalent to an always-happening error plus another error with probability $1 - P_e < 50\%$. We can first apply this error to the syndrome by flipping the incident vertices and then decode with positive weighted edge $w_e = \log \frac{P_e}{(1-P_e)}$.

Fact 2. The triangular relationship holds for the weights in a syndrome graph $G(V, E)$. That is, for any vertices $u, a, b \in V$, $w_{(a,b)} \leqslant w_{(a,u)} + w_{(b,u)}$. This is because syndrome graph weights are constructed from minimum-weight paths in the decoding graph.

b) Relevant facts about ***the blossom algorithm****, also see §II-B.*

Fact 3. The blossom algorithm starts with some feasible dual variables and maintains the feasibility throughout the algorithm. We assume the initial solution is $y_S = 0, \forall S \in \mathcal{O}^*$ given the non-negative weights $w_e \geq 0, \forall e \in E$ (*Fact 1*).

Fact 4. When a dual variable decreases by Δ, it must be a "$-$" node in an alternating tree, and there exists two "$+$" nodes in the tree that connects this "$-$" node with tight edges. The dual variables of these "$+$" nodes must increase by Δ.

c) Relevant facts & lemmas about ***the blossoms****.*

Fact 5. Given two different nodes S_1 and S_2, $S_1 \cap S_2 = \varnothing$ and $\mathcal{D}(S_1) \cap \mathcal{D}(S_2) = \varnothing$. That is, different nodes do not share any vertices, or share any descendants.

Lemma: Root Uniqueness. Any blossom S has a unique node whose *Progeny* includes S, denoted as $\text{Root}(S)$.

Lemma: Ancestry. Similar to the definitions in *Progeny*,

- $\mathcal{A}(v) = \mathcal{D}_v(\text{Root}(v))$.
- $\mathcal{A}(u \backslash v) = \mathcal{D}_{u \backslash v}(\text{Root}(u))$.

Lemma: Distinct-Root Ancestry. If $\text{Root}(u) \neq \text{Root}(v)$, then $\mathcal{A}(u \backslash v) = \mathcal{A}(u)$.

Fact 6. Blossom algorithm maintains all blossoms that have positive dual variables.

$$\begin{aligned}\{S | S \in \mathcal{O}^* \wedge u \in S \wedge v \notin S \wedge y_S \neq 0\} &\subseteq \mathcal{A}(u \backslash v) \\ &\subseteq \{S | S \in \mathcal{O}^* \wedge u \in S \wedge v \notin S\} \\ \{S | S \in \mathcal{O}^* \wedge e \in \delta(S) \wedge y_S \neq 0\} &\subseteq \mathcal{A}(u \backslash v) \cup \mathcal{A}(v \backslash u) \\ &\subseteq \{S | S \in \mathcal{O}^* \wedge e \in \delta(S)\}, \text{ where } e = (u, v)\end{aligned}$$

Fact 7. Given *Fact 6* and $\mathcal{A}(v_1 \backslash v_2) \cap \mathcal{A}(v_2 \backslash v_1) = \varnothing$ we can simplify the dual variable summation,

$$\sum_{\substack{S \in \mathcal{O}^* \\ v_1 \in S \wedge v_2 \notin S}} y_S = \sum_{\substack{S \in \mathcal{O}^* \\ v_1 \in S \wedge v_2 \notin S \wedge y_S \neq 0}} y_S = \sum_{A \in \mathcal{A}(v_1 \backslash v_2)} y_A$$

$$\sum_{\substack{S \in \mathcal{O}^* \\ (v_1, v_2) \in \delta(S)}} y_S = \sum_{\substack{S \in \mathcal{O}^* \\ (v_1, v_2) \in \delta(S) \wedge y_S \neq 0}} y_S = \sum_{A \in \mathcal{A}(v_1 \backslash v_2)} y_A + \sum_{A \in \mathcal{A}(v_2 \backslash v_1)} y_A$$

APPENDIX B
LP SIMPLIFICATION

Theorem: Non-negative Vertex Dual. Given the error probability for any independent error source $P(e) \leqslant 0.5$, dual variables y_v, $\forall v \in V$ are non-negative during the whole process of finding the solution by the blossom algorithm.

Proof. Given *Fact 1* and *Fact 3*, the initial state satisfies that $y_u \geqslant 0, \forall u \in V$. Whenever a vertex dual variable y_u decreases $\Delta y_u < 0$, it must be a node. According to *Fact 4*, there must exists two other nodes S_a and S_b increasing. Since there are tight edges between a node and u, we can assume edges (a, u) and (b, u) are the constraints of tight edges from the two nodes. Obviously $(u, a), (u, b) \in \delta(u)$, i.e. y_u contributes to the slack of both edges (u, a) and (u, b). According to *Fact 5*, there are no other node containing u since $\{u\}$ is a node, and any non-zero dual variable that includes a must be child node of S_a. According to *Fact 4*, there are two tight constraints.

$$y_u + y_a + \sum_{S \in \mathcal{O} | (u,a) \in \delta(S)} y_S = w_{(u,a)} \tag{2}$$

$$y_u + y_b + \sum_{S \in \mathcal{O} | (u,b) \in \delta(S)} y_S = w_{(u,b)} \tag{3}$$

According to *Fact 3* and *Fact 4*, there are three constraints for the update amount Δ between vertices a, b and u.

$$y_u - \Delta + y_a + \sum_{S \in \mathcal{O} | (u,a) \in \delta(S)} y_S + \Delta \leqslant w_{(u,a)} \tag{4}$$

$$y_u - \Delta + y_b + \sum_{S \in \mathcal{O} | (u,b) \in \delta(S)} y_S + \Delta \leqslant w_{(u,b)} \tag{5}$$

$$y_a + \sum_{S \in \mathcal{O} | (u,a) \in \delta(S)} y_S + \Delta + y_b + \sum_{S \in \mathcal{O} | (u,b) \in \delta(S)} y_S + \Delta \leqslant w_{(a,b)} \tag{6}$$

Constraints (4) and (5) are automatically satisfied with any Δ value, but constraint 6 requires that

$$\Delta \leqslant \left(w_{(a,b)} - y_a - \sum_{S \in \mathcal{O} | (u,a) \in \delta(S)} y_S - y_b - \sum_{S \in \mathcal{O} | (u,b) \in \delta(S)} y_S \right) / 2$$

Given $w_{(a,b)} \leqslant w_{(u,a)} + w_{(u,b)}$ (*Fact 2*) and Eqs. (2) and (3),

$$\Delta \leqslant y_u$$

That means in the next stage $y'_u = y_u - \Delta \geq 0$. Since blossom algorithm starts with $y_u \geqslant 0$ and there is no chance of decreasing any y_u below zero at any point of the algorithm, we can conclude that $y_u \geqslant 0$ stands throughout the algorithm. □

APPENDIX C
PROOFS OF DECODING GRAPH

Lemma: Edge Max Point. For a (segment) edge $e = (s, t)$ of weight w and a point $p \notin e$,

$$\max_{q \in e} \text{Dist}(p, q) = \min(\text{Dist}(p, s), \text{Dist}(p, t)) + w/2 + |\text{Dist}(p, s) - \text{Dist}(p, t)|/2$$

Proof. By the definition of *Distance*, we have $|\text{Dist}(p, t) - \text{Dist}(p, s)| \leqslant w$. We can denote

$$|\text{Dist}(p, t) - \text{Dist}(p, s)| = w - \delta$$

where $0 \leqslant \delta \leqslant w$. Without loss of generality, we assume $\text{Dist}(p, s) \leqslant \text{Dist}(p, t)$. $\forall q \in e$, let $w_1 = \text{Dist}(q, s)$. We have

$$\begin{aligned}\text{Dist}(p, q) &= \min(\text{Dist}(p, s) + w_1, \text{Dist}(p, t) + w - w_1) \\ &= \min(\text{Dist}(p, s) + w_1, (\text{Dist}(p, s) + w - \delta) + w - w_1) \\ &= \text{Dist}(p, s) + \min(w_1, 2w - \delta - w_1)\end{aligned}$$

Thus, we have

$$\begin{aligned}\max_{q \in e} \text{Dist}(p, q) &= \text{Dist}(p, s) + w - \delta/2 \\ &= \min(\text{Dist}(p, s), \text{Dist}(p, t)) + w - \delta/2\end{aligned}$$

□

Lemma: Edge Min Point. For a (segment) edge $e = (s, t)$ of weight w and a point $p \notin e$,

$$\min_{q \in e} \text{Dist}(p, q) = \min(\text{Dist}(p, s), \text{Dist}(p, t))$$

Lemma: Edge Max-Min Bound. For a (segment) edge e of weight w and a point $p \notin e$,

$$\max_{q \in e} \text{Dist}(p, q) - \min_{q \in e} \text{Dist}(p, q) \geqslant w/2.$$

Lemma: Circle-Edges. A circle $C(v, d)$ on the decoding graph is a union of a finite number of vertices and (segment) edges with their incident points.

Proof. There are a finite number of vertices V_M. Given an edge $e = (s, t)$ of weight w, if $w = 0$, the edge is empty $e = \varnothing$; If $w > 0$, there are four situations regarding its relationship:

- if $\forall p \in e$, $\text{Dist}(p, v) > d$, e is outside the circle.
- if $\forall p \in e$, $\text{Dist}(p, v) \leqslant d$, $e \cup \{s, t\}$ is inside the circle.
- if $\exists r \in e$, $\text{Dist}(s, v) \leqslant d$, and $\exists t \in e$, $\text{Dist}(t, v) > d$, there must be one or two points that are of distance d to v. When there is a single point, it partitions e into two segments: one inside the circle and the other outside.
- When there are two points that are of distance d to v, they partition e into three segments: the middle segment lies outside the circle while the other two inside.

With the above, we can conclude that a circle covers a finite number of vertices and (segment) edges with their incident points. That is, a circle is a union of a finite number of sets each represented by an closed intervals of (segment) edge. □

APPENDIX D
PROOFS OF BLOSSOM

The blossom algorithm solves the MWPM problem for the *syndrome* graph. Our Parity Blossom algorithm solves the same problem but work on the decoding graph. Therefore, to prove that Parity Blossom solves the same problem, we relate dual variables in the blossom algorithm to the geometric objects of the decoding graph.

We imagine the syndrome graph is overlaid over its decoding graph. As $V \subseteq V_M$, a vertex in the syndrome graph is aligned with its correspondent in the decoding graph, as shown in Fig. 11.

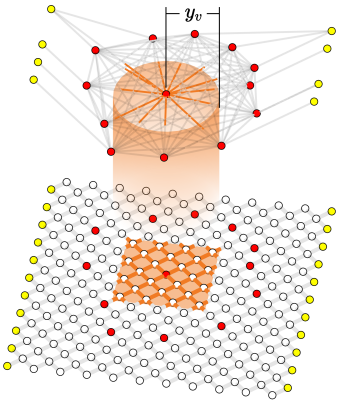


Figure 11: Overlaying syndrome graph on top of its decoding graph. The center vertex is v. The dual variable y_v on the syndrome graph corresponds to a *Circle* of the same radius on the decoding graph. In the decoding graph, the circle $C(v, y_v)$ consists of all the orange (segment) edges. Despite its square appearance, it is actually a "circle" in Manhattan geometry.

We define

$$C(v) = C(v, \sum_{A \in \mathcal{A}(v)} y_A)$$
$$C(u \backslash v) = C(u, \sum_{A \in \mathcal{A}(u \backslash v)} y_A)$$

Lemma: Root Cover. If S is a node, i.e., $S = \text{Root}(S)$,

$$\text{Cover}(S) = \cup_{v \in S} C(v).$$

Lemma: Distinct-Root Circle. If $\text{Root}(u) \neq \text{Root}(v)$, then $C(u \backslash v) = C(u)$.

Lemma: Tight Edge. An edge (v_1, v_2) in the syndrome graph becomes tight if and only if $C(v_1 \backslash v_2)$ and $C(v_2 \backslash v_1)$ on the decoding graph overlap. That is,

$$w_e = \sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S \iff C(v_1 \backslash v_2) \cap C(v_2 \backslash v_1) \neq \varnothing$$

Proof. **Sufficiency.** Assume on the decoding graph $\exists p \in C(v_1, \sum_{A \in \mathcal{A}(v_1 \backslash v_2)} y_A) \cap C(v_2, \sum_{A \in \mathcal{A}(v_2 \backslash v_1)} y_A)$. By the def-

inition of Circle,

$$\text{Dist}(v_1, p) \leqslant \sum_{A \in \mathcal{A}(v_1 \setminus v_2)} y_A$$
$$\text{Dist}(v_2, p) \leqslant \sum_{A \in \mathcal{A}(v_2 \setminus v_1)} y_A$$

According to the triangular relationship, the weight of the edge between v_1 and v_2 in the syndrome graph has $w_e = \text{Dist}(v_1, v_2) \leqslant \text{Dist}(v_1, p) + \text{Dist}(v_2, p)$. This and *Fact 7* have

$$w_e \leqslant \sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S$$

Since the edge slackness constraints (3b) also says $\geqslant$ of the above inequality, we have

$$w_e = \sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S$$

Necessity. Suppose an edge $e = (v_1, v_2)$ is tight, there exists a minimum-weight path from v_1 to v_2 in the decoding graph that consists of edges $e_1 = (v_1, a), e_2, \cdots, e_n = (b, v_2)$ where

$$\sum_{i=1}^{n} w_{e_i} = w_e = \sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S$$

Since $\text{Dist}(v_1, v)$ is continuous and monotonic as v moves from v_1 to v_2 along the minimum-weight path, $\exists p \in e_j$ that split the edge e_j into two parts weighted w_1 and w_2, where

$$\text{path } (v_1, p): \quad \sum_{i=1}^{j-1} w_{e_i} + w_1 = \sum_{S \in \mathcal{O}^* | v_1 \in S \wedge v_2 \notin S} y_S$$
$$\text{path } (p, v_2): \quad w_2 + \sum_{i=j+1}^{n} w_{e_i} = \sum_{S \in \mathcal{O}^* | v_1 \notin S \wedge v_2 \in S} y_S$$

Given the distance definition and *Fact 7*,

$$\text{Dist}(v_1, p) \leqslant \sum_{S \in \mathcal{O}^* | v_1 \in S \wedge v_2 \notin S} y_S = \sum_{A \in \mathcal{A}(v_1 \setminus v_2)} y_A$$
$$\text{Dist}(v_2, p) \leqslant \sum_{S \in \mathcal{O}^* | v_1 \notin S \wedge v_2 \in S} y_S = \sum_{A \in \mathcal{A}(v_2 \setminus v_1)} y_A$$

That is, $\exists p$ belongs to both $C(v_1 \setminus v_2)$ and $C(v_2 \setminus v_1)$. □

Theorem: Node Cover Finite Overlap. Given two nodes S_1 and S_2, $\text{Cover}(S_1) \cap \text{Cover}(S_2)$ is a finite set.

Proof. Because S_1 and S_2 are nodes,

$$\text{Cover}(S_1) \cap \text{Cover}(S_2) = \left(\bigcup_{v_1 \in S_1} C(v_1) \right) \cap \left(\bigcup_{v_2 \in S_2} C(v_2) \right).$$

Suppose $|\text{Cover}(S_1) \cap \text{Cover}(S_2)|$ is infinite, there must exist two vertices $v_1 \in S_1$ and $v_2 \in S_2$ such that $|C(v_1) \cap C(v_2)|$ is infinite.

Since a circles includes a finite number of edges, there must exists a (segment) edge f of a nonzero weight $w > 0$ with

$$f \subseteq C(v_1) \cap C(v_2).$$

By the definition of Circle, we have

$$\forall p \in f, \text{Dist}(v_1, p) \leqslant \sum_{A \in \mathcal{A}(v_1)} y_A \quad \text{and}$$
$$\text{Dist}(v_2, p) \leqslant \sum_{A \in \mathcal{A}(v_2)} y_A$$

Moreover, with *Lemma: Edge Max-Min Bound*,

$$\exists p \in f, \text{Dist}(v_2, p) \leqslant \sum_{A \in \mathcal{A}(v_2)} y_A - w/2$$

Given the triangular inequality of the distance function $\text{Dist}(v_1, v_2) \leq \text{Dist}(v_1, p) + \text{Dist}(v_2, p)$,

$$\text{Dist}(v_1, v_2) + w/2 \leqslant \sum_{A \in \mathcal{A}(v_1)} y_A + \sum_{A \in \mathcal{A}(v_2)} y_A.$$

With *Fact 7* and *Lemma: Distinct-Root Ancestry*,

$$\sum_{A \in \mathcal{A}(v_1)} y_A + \sum_{A \in \mathcal{A}(v_2)} y_A = \sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S$$

The syndrome graph edge $e = (v_1, v_2)$ has a weight $w_e = \text{Dist}(v_1, v_2)$, thus

$$w_e + w/2 \leqslant \sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S \implies w_e < \sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S$$

The above violates the edge slackness constraints (3b). As the result, the theorem must be true. □

Theorem: Tight Edge Detection (Cover). There exists a tight edge between two different nodes S_1 and S_2 if and only if $\text{Cover}(S_1)$ and $\text{Cover}(S_2)$ overlap. That is,

$$\exists e = (v_1, v_2) \in E, v_1 \in S_1 \ \wedge \ v_2 \in S_2 \ \wedge \ w_e = \sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S$$
$$\iff \text{Cover}(S_1) \cap \text{Cover}(S_2) \neq \varnothing$$

Proof. Since S_1 and S_2 are different nodes, we have $C(v_1 \setminus v_2) = C(v_1)$ and $C(v_2 \setminus v_1) = C(v_2)$ per *Lemma: Distinct-Root Circle*.

Necessity. With *Lemma: Tight Edge*,

$$w_e = \sum_{S \in \mathcal{O}^* | (v_1, v_2) \in \delta(S)} y_S$$
$$\implies C(v_1 \setminus v_2) \cap C(v_2 \setminus v_1) \neq \varnothing$$
$$\implies C(v_1) \cap C(v_2) \neq \varnothing$$
$$\implies \text{Cover}(S_1) \cap \text{Cover}(S_2) \neq \varnothing$$

The last step is true because the *Cover* of a node consists of circles of all its vertices, per *Lemma: Root Cover*,

$$\text{Cover}(S) = \cup_{v \in S} C(v)$$

Sufficiency.

$$\begin{aligned}&\text{Cover}(S_1) \cap \text{Cover}(S_2) \neq \varnothing \\ &\Longrightarrow \exists v_1 \in S_1, v_2 \in S_2, C(v_1) \cap C(v_2) \neq \varnothing \\ &\Longrightarrow C(v_1 \backslash v_2) \cap C(v_2 \backslash v_1) \neq \varnothing \\ &\Longrightarrow w_{(v_1,v_2)} = \sum_{S \in \mathcal{O}^* | (v_1,v_2) \in \delta(S)} y_S\end{aligned}$$

□

APPENDIX E
PROOFS OF FUSION BLOSSOM

$V_b \subset V_M$ divides a decoding graph $G(E_M, V_M)$ into two disjoint graphs that include V_1 and V_2, respectively if there is no edge in the decoding graph that connects vertices from both V_1 and V_2. As a result, it divides the MWPM problem into two disjoint sub-problems. For the i-th sub-problem, $i \in \{1, 2\}$, the primal and dual formulations as in Eq. 1 and 3, respectively, have E and $\mathcal{O}^*$ as follows.

$$\begin{aligned}E_i &= \{e | e = (u, v) \in E \ \wedge \ u, v \in V_i \cup V_b\} \\ \mathcal{O}_i^* &= \{S | S \in \mathcal{O}^* \ \wedge \ S \subseteq V_i\}\end{aligned}$$

Theorem: Feasible Dual Variables. Solutions for the two disjoint sub-problems determine the values of the dual variables $y_S, S \in \mathcal{O}_1^* \cup \mathcal{O}_2^*$. These values plus setting y_S to 0 for $S \in \mathcal{O}^* \setminus \mathcal{O}_1^* \cup \mathcal{O}_2^*$ constitute a feasible solution to the original dual problem.

Proof. Apparently $\mathcal{O}_1^* \cap \mathcal{O}_2^* = \varnothing$ and $\mathcal{O}_1^* \cup \mathcal{O}_2^* \subseteq \mathcal{O}^*$. Likewise, $E_1 \cup E_2 \subseteq E$.

The MWPM solutions for the sub-problems must satisfy the following by definition

$$\begin{aligned}\sum_{S \in \mathcal{O}_1^* | e \in \delta(S)} y_S \leqslant w_e, \ \forall e \in E_1 \\ \sum_{S \in \mathcal{O}_2^* | e \in \delta(S)} y_S \leqslant w_e, \ \forall e \in E_2\end{aligned}$$

Additionally, since any vertex in V_b does not appear in any blossom of $\mathcal{O}_1^*$ or $\mathcal{O}_2^*$,

$$\begin{aligned}e \notin \delta(S), \ \forall S \in \mathcal{O}_1^*, \ \forall e \in E_2 \\ e \notin \delta(S), \ \forall S \in \mathcal{O}_2^*, \ \forall e \in E_1\end{aligned}$$

These four can be combined as

$$\sum_{S \in \mathcal{O}_1^* \cup \mathcal{O}_2^* | e \in \delta(S)} y_S \leqslant w_e, \ \forall e \in E_1 \cup E_2$$

Because y_S is 0 for $S \in \mathcal{O}^* \setminus (\mathcal{O}_1^* \cup \mathcal{O}_2^*)$, we can further simplify the above as

$$\sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S \leqslant w_e, \ \forall e \in E_1 \cup E_2$$

We only need to prove the constraint is also met for $\forall e \in E \setminus (E_1 \cup E_2)$, which can be noted as $e = (v_1, v_2)$ where $v_1 \in V_1$ and $v_2 \in V_2$. e corresponds to a minimum-weight path between v_1 and v_2 in the original decoding graph. Since V_b divides the decoding graph into two disjoint parts, this path must go through $\exists v_b \in V_b$. Since a minimum-weight path go through v_1, v_b, v_2, we have $w_{(v_1,v_2)} = w_{(v_1,v_b)} + w_{(v_2,v_b)}$. Because $(v_1, v_b) \in E_1$ and $(v_2, v_b) \in E_2$, we have

$$\begin{aligned}\sum_{S \in \mathcal{O}_1^* | (v_1,v_2) \in \delta(S)} y_S \leqslant w_{(v_1,v_b)} \\ \sum_{S \in \mathcal{O}_2^* | (v_1,v_2) \in \delta(S)} y_S \leqslant w_{(v_2,v_b)}\end{aligned}$$

Again because y_S is 0 for $S \in \mathcal{O}^* \setminus (\mathcal{O}_1^* \cup \mathcal{O}_2^*)$, we have

$$\sum_{S \in \mathcal{O}^* | (v_1,v_2) \in \delta(S)} y_S \leqslant w_{(v_1,v_b)} + w_{(v_2,v_b)} = w_{(v_1,v_2)}$$

That is,

$$\sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S \leqslant w_e, \ \forall e \in E \setminus (E_1 \cup E_2)$$

Overall, the dual variables are feasible in (3a) and (3b)

$$\begin{aligned}w_e - \sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S \geqslant 0, \ \forall e \in E \\ y_S \geqslant 0, \ \forall S \in \mathcal{O}^*\end{aligned}$$

□

APPENDIX F
PROOFS OF PARITY DUAL MODULE

We optimize the algorithm by tracking *Pseudo-Covers* (§V-B1) instead of *Covers*. Pseudo-Covers are similar to Covers, but differ by only a few boundary vertices. The primary complication introduced by Pseudo-Cover stems from the presence of zero edges, as all the vertices connected by zero edges can belong to an arbitrary number of Covers. Additional complications arise when Fusion Blossom partitions the decoding graph, as some of the zero edges may not be known to a Pseudo-Cover when it is constructed from a sub-problem. Therefore, it is natural to question whether Pseudo-Covers are equally effective as Covers in detecting *Obstacles*.

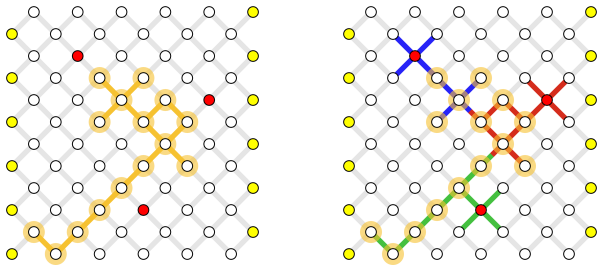
Here we prove the correctness of using Pseudo-Covers, as stated in **Theorem: Tight Edge Detection (Pseudo-Cover)**. We start with a few definitions and lemmas exclusively used in the proof of this theorem. We will use the example in Fig. 12 throughout. Pseudo-Covers have the following properties.

$$\begin{aligned}\overline{\text{Cover}}(S) \subseteq \text{Cover}(S) \\ \text{Cover}(S) \setminus \overline{\text{Cover}}(S) \subseteq V_M \\ \overline{\text{Cover}}(S_1) \cap \overline{\text{Cover}}(S_2) \cap V_M = \varnothing, \ \forall S_1 \neq S_2\end{aligned}$$

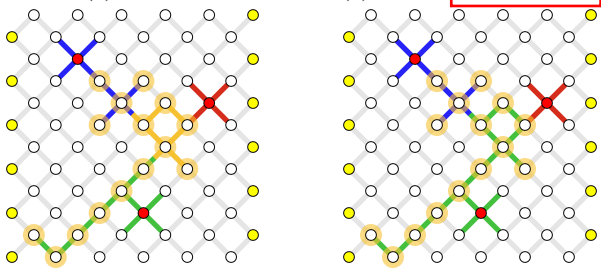
Lemma: No Free Zero Edge Vertex. Given a zero edge $e = (u, v) \in E_M$, if there is a node S such that $u \in \overline{\text{Cover}}(S)$, there is a node T such that $v \in \overline{\text{Cover}}(T)$.

Definition: Island. We define the *island* of a vertex v as the set of vertices that have zero distance to v, noted as

$$\text{Island}(v) = \{u | \text{Dist}(u, v) = 0, u \in V_M\}$$



(1) Island (2) Invalid *Pseudo-Covers*



(3) $\Delta y_R = -1$, $\overline{\text{Cover}}(R)$ (4) $\Delta y_G = +1$, $\overline{\text{Cover}}(G)$

Figure 12: Example of maintaining Pseudo-Covers to detect at least one obstacle. (1) The *Island* is marked in yellow circles, connected by zero edges in yellow. When a vertex in the Island is inside a Pseudo-Cover, we mark half of the incident edges with the color of the node. (2) Suppose the nodes are R,G,B for red, green and blue respectively. Suppose $\Delta y_R = -1$, $\Delta y_G = \Delta y_B = +1$. Although there is an obstacle between G and B given $\text{Cover}(G) \cap \text{Cover}(B) \neq \varnothing$, their initial yet invalid Pseudo-Covers $\overline{\text{Cover}}(G)$ and $\overline{\text{Cover}}(B)$ do not touch on any decoding graph edge, isolated by $\overline{\text{Cover}}(R)$ as shown in the figure, thus forbidding an obstacle to be detected. In order to let $\overline{\text{Cover}}(G)$ and $\overline{\text{Cover}}(B)$ touch, the algorithm constructs valid Pseudo-Covers by (3) removing boundary vertices from $\overline{\text{Cover}}(R)$ given $\Delta y_R = -1$ and (4) absorbing as many boundary vertices as possible into $\overline{\text{Cover}}(G)$ given $\Delta y_G = +1$. After that, $\overline{\text{Cover}}(G)$ and $\overline{\text{Cover}}(B)$ touch on one decoding graph edge and thus detects an obstacle.

Obviously, $v \in \text{Island}(v)$. Also, $|\text{Island}(v)| > 1$ iff $\exists e = (u, v) \in E_M$, $w_e = 0$. That is, an island is non-trivial only when there are zero edges. An example is shown in Fig. 12(1), where the vertices in yellow circles constitute an island, connected by yellow zero edges.

Definition: Occupancy. We define the *Occupancy* of a vertex v as the set of nodes whose Cover includes v, noted as

$$\text{Occupancy}(v) = \{S | v \in \text{Cover}(S) \wedge S \text{ is node}\}$$

A member of the Occupancy is called an Occupier.

Lemma: Occupier Boundary. If $v \in V_M$ has more than one occupiers, it must be on the boundary of the Covers of all its occupiers.

Proof. Assume $S_1, S_2 \in \text{Occupancy}(v)$. We have $v \in \text{Cover}(S_1) \cap \text{Cover}(S_2)$. We prove by contradiction, assuming v is not on the boundary of $\text{Cover}(S_1)$.

Since v is strictly inside $\text{Cover}(S_1)$, there exists a defect vertex $u_1 \in S_1$ such that v is strictly inside the circle $C(u_1)$. That is, $\text{Dist}(u_1, v) < \sum_{A \in \mathcal{A}(u_1)} y_A$. Since $v \in \text{Cover}(S_2)$, there exists a defect vertex $u_2 \in S_2$ such that $v \in C(u_2)$. That is, $\text{Dist}(u_2, v) \leqslant \sum_{A \in \mathcal{A}(u_2)} y_A$. We have

$$\begin{aligned}\text{Dist}(u_1, u_2) &\leqslant \text{Dist}(u_1, v) + \text{Dist}(u_2, v) \\ &< \sum_{A \in \mathcal{A}(u_1)} y_A + \sum_{A \in \mathcal{A}(u_2)} y_A = \sum_{S \in \mathcal{O}^* | (u_1, u_2) \in \delta(S)} y_S\end{aligned}$$

This violates the dual constraint (2a) of $e = (u_1, u_2) \in E$, contradiction. Thus, the theorem must be true. □

We can extend the notion of Occupancy to an Island: the Occupancy of Island(v) is the same as Occupancy(v). This is because

$$v \in \text{Cover}(S) \iff \text{Island}(v) \subseteq \text{Cover}(S)$$

As shown in example Fig. 12(1), the Occupancy of the island is $\varnothing$, while in Fig. 12(2) the Occupancy of the island is $\{R, G, B\}$.

Lemma: Pseudo-Cover Touching. Consider a vertex $v \in V_M$, if there exists two node $S_1, S_2 \in \text{Occupancy}(v)$ with $\Delta y_{S_1} + \Delta y_{S_2} > 0$, there exists two nodes $S_3, S_4 \in \text{Occupancy}(v)$ with $\Delta y_{S_3} + \Delta y_{S_4} > 0$ whose Pseudo Covers border each other. That is,

$$\begin{aligned}&\exists S_1, S_2 \in \text{Occupancy}(v), \Delta y_{S_1} + \Delta y_{S_2} > 0 \\ \implies \quad &\forall S_3 \in \text{Occupancy}(v), \Delta y_{S_3} > 0, \\ &\exists S_4 \in \text{Occupancy}(v) \setminus \{S_3\}, \Delta y_{S_4} \geqslant 0, e = (a, b) \in E_M, \\ &a \in \overline{\text{Cover}}(S_3), b \in \overline{\text{Cover}}(S_4), \\ &e \subseteq \overline{\text{Cover}}(S_3) \cup \overline{\text{Cover}}(S_4)\end{aligned}$$

Proof. Because v has at least two occupiers (S_1 and S_2), v and all members of Island(v) must be on the boundaries of the Covers of all its occupiers, per *Lemma: Occupier Boundary*.

According to how a Pseudo Cover is derived, $\forall S \in \text{Occupancy}(v)$ with $\Delta y_S < 0$, we have $\overline{\text{Cover}}(S) \cap \text{Island}(v) = \varnothing$, because all zero edges and their vertices must be removed from Cover(S) to create $\overline{\text{Cover}}(S)$.

For every $S_3 \in \text{Occupancy}(v)$ with $\Delta y_{S_3} > 0$, there are only two cases.

- In the first case, Island(v) does not overlap with any other Pseudo-Covers beyond $\overline{\text{Cover}}(S_3)$. We have $\text{Island}(v) \subseteq \overline{\text{Cover}}(S_3)$ given the island is a connected graph and how Pseudo-Covers are derived. Since there exists another node $S_4 \in \text{Occupancy}(v) \setminus \{S_3\}$ with $\Delta y_{S_4} \geqslant 0$ and $\text{Island}(v) \cap \overline{\text{Cover}}(S_4) = \varnothing$, there must exist a decoding graph edge $e = (a, b) \in \text{Cover}(S_4)$ where $a \in \text{Island}(v) \subseteq \overline{\text{Cover}}(S_3)$ and $b \in \overline{\text{Cover}}(S_4)$. That is, $a \in \text{Island}(v)$ must be removed from $\text{Cover}(S_4)$ to form $\overline{\text{Cover}}(S_4)$. Because $b \notin \text{Island}(v)$, w_e must be greater than zero. As a result, $e \subseteq \overline{\text{Cover}}(S_4)$. Therefore, $e \subseteq \overline{\text{Cover}}(S_4) \subseteq \overline{\text{Cover}}(S_3) \cup \overline{\text{Cover}}(S_4)$, and $\Delta y_{S_3} + \Delta y_{S_4} > 0$.

- Island(v) overlaps with a Pseudo-Cover beyond that of S_3. There must exists a zero edge $e = (a, b) \in E_M$ that $a, b \in$ Island(v) and a and b are covered by Pseudo-Covers of S_3 and another nodes S_4, respectively. We have $\Delta y_{S_4} \geqslant 0$ because $\overline{\text{Cover}}(S) \cap \text{Island}(v) = \varnothing$ for any $\Delta y_S < 0$. We have $e = \varnothing \subseteq \overline{\text{Cover}}(S_3) \cup \overline{\text{Cover}}(S_4)$, and $\Delta y_{S_3} + \Delta y_{S_4} > 0$.

□

Lemma: Edge Fully Cover. If a decoding graph edge $e = (u, v) \in E_M$ is covered by the union of two different Pseudo-Covers S_1, S_2 and u, v are inside $\overline{\text{Cover}}(S_1), \overline{\text{Cover}}(S_2)$ respectively, then the Covers of S_1 and S_2 overlap. That is,

$$e \subseteq \overline{\text{Cover}}(S_1) \cup \overline{\text{Cover}}(S_2), u \in \overline{\text{Cover}}(S_1), v \in \overline{\text{Cover}}(S_2)$$
$$\Longrightarrow \text{Cover}(S_1) \cap \text{Cover}(S_2) \neq \varnothing$$

Proof. We prove there exists a point in Cover$(S_1) \cap$Cover(S_2). We have $u \in \overline{\text{Cover}}(S_1) \subseteq \text{Cover}(S_1)$ and $v \in \overline{\text{Cover}}(S_2) \subseteq \text{Cover}(S_2)$.

Clearly, if $u \in \text{Cover}(S_2)$ or $v \in \text{Cover}(S_1)$, the vertex u or v belongs to Cover$(S_1) \cap$ Cover(S_2).

If not, then $w_e > 0$ per definition of *Island* and *Occupancy*. Since $v \notin \text{Cover}(S_1)$, Cover$(S_1) \cap (e \cup \{u, v\})$ is a closed edge segment from u to $p_1 \in (u, v)$. Similarly, Cover$(S_2) \cap (e \cup \{u, v\})$ is a closed edge segment from v to $p_2 \in (u, v)$. We have $e \subseteq \overline{\text{Cover}}(S_1) \cup \overline{\text{Cover}}(S_2) \subseteq \text{Cover}(S_1) \cup \text{Cover}(S_2)$. Thus, we have Dist$(u, p1) +$ Dist$(v, p2) \geqslant w_e$, because otherwise there exists a point on segment edge $p \in (p_1, p_2) \subseteq e$ where $p \notin \text{Cover}(S_1) \cup \text{Cover}(S_2)$. Also, Dist$(u, p1) +$ Dist$(v, p2) \leqslant w_e$ given **Theorem: Node Cover Finite Overlap**, because otherwise Cover(S_1) and Cover(S_2) overlap on segment edge (p_1, p_2) with infinite points. Thus, we have Dist$(u, p1) +$ Dist$(v, p2) = w_e$, i.e., point $p_1 = p_2$ belongs to both Cover(S_1) and Cover(S_2).

□

Theorem: Tight Edge Detection (Pseudo-Cover). There exists a tight edge between two different nodes S_1 and S_2 with $\Delta y_{S_1} + \Delta y_{S_2} > 0$ if and only if there exists two different nodes S_3 and S_4 with $\Delta y_{S_3} + \Delta y_{S_4} > 0$ whose Pseudo-Covers meet on a decoding graph edge. That is,

$$\exists S_1, S_2, e = (v_1, v_2) \in E,$$
$$v_1 \in S_1, v_2 \in S_2, \Delta y_{S_1} + \Delta y_{S_2} > 0, w_e = \sum_{S \in \mathcal{O}^* | e \in \delta(S)} y_S$$
$$\Longleftrightarrow \quad \exists S_3, S_4, e' = (v_3, v_4) \in E_M,$$
$$v_3 \in \overline{\text{Cover}}(S_3), v_4 \in \overline{\text{Cover}}(S_4),$$
$$\Delta y_{S_3} + \Delta y_{S_4} > 0, e' \subseteq \overline{\text{Cover}}(S_3) \cup \overline{\text{Cover}}(S_4)$$

Proof. **Sufficiency.** We prove that when $S_1 = S_3$, $S_2 = S_4$, there exists such a tight edge e. Given **Theorem: Tight Edge Detection (Cover)** and $\Delta y_{S_1} + \Delta y_{S_2} > 0$, we only need to prove Cover$(S_3) \cap$ Cover$(S_4) \neq \varnothing$. Although their Pseudo-Covers do not overlap on vertices, i.e., $\overline{\text{Cover}}(S_3) \cap \overline{\text{Cover}}(S_4) \cap V_M = \varnothing$, we have Cover$(S_3) \cap$ Cover$(S_4) \neq \varnothing$ given *Lemma: Edge Fully Cover*.

Necessity. Given **Theorem: Tight Edge Detection (Cover)**, we have Cover$(S_1) \cap$ Cover$(S_2) \neq \varnothing$ and $\Delta y_{S_1} + \Delta y_{S_2} > 0$. Since $\Delta y_S \in \{0, \pm 1\}$, without loss of generality, we have $\Delta y_{S_1} = +1$ and $\Delta y_{S_2} \in \{0, +1\}$. Also, there exists a point $p \in$ Cover$(S_1) \cap$ Cover(S_2).

If such a point belongs to an edge $p \in e' = (v_3, v_4) \in E_M$, we have $w_e > 0$ and Dist$(v_3, p) > 0$, Dist$(v_4, p) > 0$. Given **Theorem: Node Cover Finite Overlap**, v_3 and v_4 must belong to different Covers. Without loss of generality, we assume $v_3 \in$ Cover(S_1) and $v_4 \in$ Cover(S_2). Thus, segment edge $(v_3, p) \subseteq$ Cover(S_1) and $(v_4, p) \subseteq$ Cover(S_2). Given edge segments contain no vertex and Cover$(S) \setminus \overline{\text{Cover}}(S) \subseteq V_M$, we have $(v_3, p) \subseteq \overline{\text{Cover}}(S_1)$, $(v_4, p) \subseteq \overline{\text{Cover}}(S_2)$ and $p \in \overline{\text{Cover}}(S_1) \cap \overline{\text{Cover}}(S_2)$. Thus,

$$e' = (v_3, p) \cup \{p\} \cup (p, v_4) \subseteq \overline{\text{Cover}}(S_1) \cup \overline{\text{Cover}}(S_2)$$

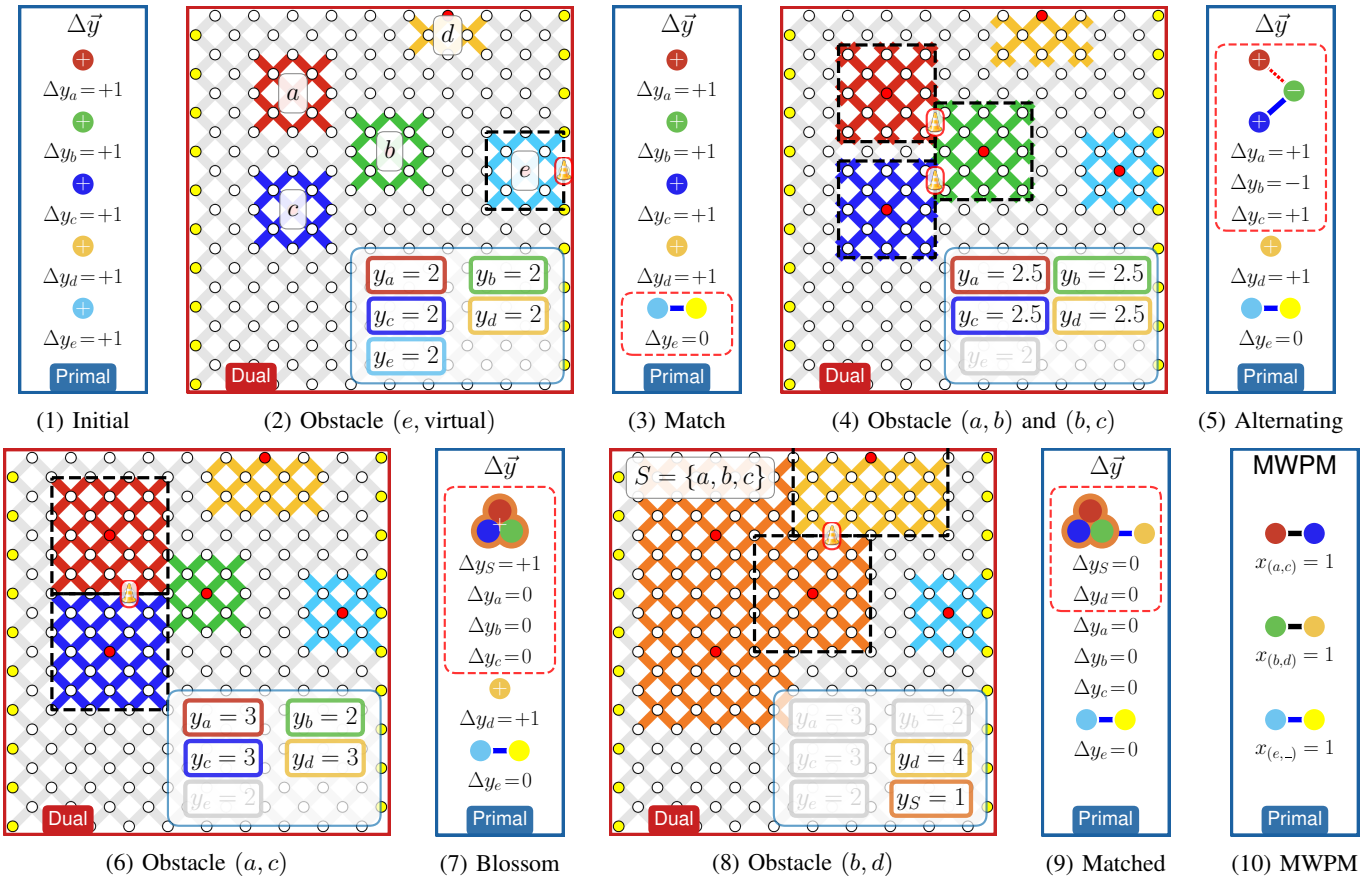
Since $v_3, p \in$ Cover(S_1), $v_4 \notin$ Cover(S_1), v_3 is not on the boundary of Cover(S_1). Given Cover$(S) \setminus \overline{\text{Cover}}(S)$ only consists of boundary vertices per definition of *Pseudo-Cover*, we have $v_3 \in \overline{\text{Cover}}(S_1)$. Similarly, $v_4 \in \overline{\text{Cover}}(S_2)$. That is, there exist $S_3 = S_1$ and $S_4 = S_2$ satisfying the conditions.

If such a point is a vertex v, we have $S_1, S_2 \in$ Occupancy(v). Now we can simply invoke *Lemma: Pseudo-Cover Touching* to complete the proof. Note that in this case, S_3 and S_4 are not necessarily S_1 and S_2.

□

Appendix G
Parity Blossom Example

An example of Parity Blossom is shown in Fig. 13, demonstrating the whole procedure from receiving the defect vertices to calculating the MWPM.



(1) Initial (2) Obstacle $(e, \text{virtual})$ (3) Match (4) Obstacle (a, b) and (b, c) (5) Alternating

(6) Obstacle (a, c) (7) Blossom (8) Obstacle (b, d) (9) Matched (10) MWPM

Figure 13: Blossom on Decoding Graph using Parity Blossom as example. The primal phase works on the matchings and outputs directions $\Delta\vec{y}$. The dual phase works on the *Covers* of nodes (in different colors) and outputs *Obstacles* (with icon). When an obstacle is detected between vertices u and v, the black dashed lines show the *Circles* $C(u)$ and $C(v)$ as part of their individual Covers. (1) The algorithm starts with direction $\Delta y_v = +1,\ \forall v \in V$ and initial dual variables $y_S = 0,\ \forall S \in \mathcal{O}^*$. (2) Dual phase grows $2\Delta\vec{y}$ and finds an obstacle between vertex e and a virtual vertex on the right. (3) Primal phase overcomes the obstacle by matching e with the virtual vertex, and set $\Delta y_e = 0$. (4) Dual phase grows $\frac{1}{2}\Delta\vec{y}$ and finds two obstacles at tight edges (a, b) and (b, c). (5) Primal phase constructs an alternating tree with alternating grow and shrink $\Delta y_a, \Delta y_b, \Delta y_c = +1, -1, +1$ to overcome the obstacles. (6) Dual phase grows $\frac{1}{2}\Delta\vec{y}$ and finds an obstacle between (a, c). (7) Primal phase constructs a blossom $S = \{a, b, c\}$ with $\Delta y_S = +1$. (8) Dual phase grows $\Delta\vec{y}$ and finds an obstacle between (b, d). It's found by first detecting $\text{Cover}(S) \cap \text{Cover}(\{d\}) \neq \varnothing$ and then detecting $C(b \backslash d) \cap C(d \backslash b) \neq \varnothing$. (9) Primal phase matches S to b. All nodes are matched. (10) Primal phase expands the blossoms and outputs an MWPM.