Progressive-Proximity Bit-Flipping for Decoding Surface Codes

Michele Pacenti, *Graduate Student Member, IEEE,* Mark F. Flanagan, *Senior Member, IEEE,* Dimitris Chytas, and Bane Vasić, *Fellow, IEEE*

***Abstract*—Topological quantum codes, such as toric and surface codes, are excellent candidates for hardware implementation due to their robustness against errors and their local interactions between qubits. However, decoding these codes efficiently remains a challenge: existing decoders often fall short of meeting requirements such as having low computational complexity (ideally linear in the code's blocklength), low decoding latency, and low power consumption. In this paper we propose a novel bit-flipping (BF) decoder tailored for toric and surface codes. We introduce the proximity vector as a heuristic metric for flipping bits, and we develop a new subroutine for correcting degenerate multiple errors on adjacent qubits. Our algorithm has quadratic complexity growth and it can be efficiently implemented as it does not require operations on dynamic memories, as do state-of-art decoding algorithms such as minimum weight perfect matching or union find. The proposed decoder shows a decoding threshold of 7.5% for the 2D toric code and 7% for the rotated planar code over the binary symmetric channel.**

***Index Terms*—Surface codes, topological codes, bit flipping, decoding algorithm, quantum error correction.**

I. Introduction

QUANTUM computers make use of the principles of quantum mechanics to perform computations. Quantum states are fragile and very sensitive to errors, and thus it is crucial to implement quantum error correction techniques to protect quantum information. A very important class of quantum codes are topological codes, specifically surface and toric codes [1], as they can be implemented on a planar quantum chip. Decoding of surface codes is typically performed using the minimum-weight-perfect-matching (MWPM) decoder [2]; although MWPM provides excellent decoding performance, its computational complexity makes it infeasible for large-scale implementation. Because of this, many alternative approaches have been studied to lower the decoding complexity of surface codes. The most promising alternative is the union-find (UF) decoder [3], although many other classes of decoders have been proposed, such as tensor network decoders [4] (which are implementations of maximum likelihood decoding), renormalization group decoders [5], neural network based decoders [6], maxSAT decoders [7] and cellular-automaton decoders [8]. On the other hand, message passing decoders such as belief propagation (BP) have shown to be effective for decoding surface codes when paired with MWPM and UF [9], or with post-processing techniques such as ordered statistics decoding (BP-OSD) [10]. Also, serial scheduling and re-initialization of BP has been shown to provide good decoding performance [11].

Bit flipping (BF) decoders are a class of iterative decoding algorithms known to be very fast and efficient [12], although generally they provide lower performance than BP. In general, BF decoders do not perform well on surface codes, mainly because of the very low column weight of the parity check matrix, and also because of *error degeneracy*, which refers to the property of quantum codes where multiple error patterns of the same weight can correspond to the same syndrome. Nevertheless, because of its extremely low complexity, BF is still attractive in the scenario of decoding topological codes; indeed, the latency constraint that a quantum decoder should meet is very tight, to prevent the so-called *backlog* problem [13]. Moreover, it is essential that the decoder has a low power consumption, as it has to be embedded in a cryogenic environment with a strict power budget [14]. This makes hardware implementation of the decoder a crucial aspect; unfortunately, state-of-art decoders such as MWPM or UF do not allow an efficient hardware implementation, mainly because they make use of complex data structures, which require random memory access and dynamic memory allocation, which are known to be less efficient as they introduce latency and increase circuit complexity as well as power consumption. In contrast, the BF algorithm, due to its simplicity, only requires static memories with fixed access, thus having a distinct advantage for hardware implementation.

In this paper, we develop a BF algorithm which is capable of decoding surface codes. In the proposed approach, instead of considering the number of unsatisfied checks as in conventional BF, each bit is assigned a heuristic weight which is the entry of what we call *proximity vector*. The proximity vector is the sum of different contributions called *individual influences*, and each individual influence is associated with an

This work has been conducted as a part of the CoQREATE program which is funded by the National Science Foundation under grant ERC-1941583 and Science Foundation Ireland under the US-Ireland R&D Partnership Programme under grant SFI/21/US-C2C/3750. The work of Michele Pacenti, Dimitris Chytas and Bane Vasić is also supported by the NSF under grants CIF-1855879, CIF-2106189, CCF-2100013, ECCS/CCSS-2027844, ECCS/CCSS-2052751, and in part by Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration and funded through JPL's Strategic University Research Partnerships (SURP) program. Bane Vasić has disclosed an outside interest in his startup company Codelucida to The University of Arizona. Conflicts of interest resulting from this interest are being managed by The University of Arizona in accordance with its policies.

Michele Pacenti, Dimitris Chytas and Bane Vasić are with the Department of Electrical and Computer Engineering, The University of Arizona, Tucson, AZ 85721, USA (e-mail: mpacenti@arizona.edu; dchytas@arizona.edu; vasic@ece.arizona.edu).

Mark F. Flanagan is with the School of Electrical and Electronic Engineering, University College Dublin, Belfield, Dublin 4, Ireland (e-mail: mark.flanagan@ieee.org).

unsatisfied check. Ultimately, bits connected with checks with low entries in the proximity vector will be flipped first, while bits connected with checks with high entries in the proximity vector will be flipped last. After each flip, the proximity vector is updated in an efficient manner. To deal with high-weight consecutive errors, *i.e.*, errors occurring on a set of adjacent qubits, which are particularly harmful for iterative decoders, we design an iterative matching procedure that runs after BF and that is able to correct these errors. We show that our decoder has an asymptotic complexity of $\mathcal{O}(n^2)$, where n is the code's blocklength, and we provide simulation results that show a comparison, in terms of performance, with MWPM, UF and traditional BF. Although our decoder is not able to correct up to the minimum distance of the code, we show that it has a good performance while also being more feasible for hardware implementation than UF and MWPM.

The rest of the paper is organized as follows. In Section II we introduce the preliminaries of quantum error correction. In Section III we present an overview of the main decoding algorithms for topological codes. In Section IV we define the proximity vector and how it can be computed efficiently. Section V provides a detailed description of our proposed decoder. In Section VI we carry out a complexity analysis and a comparison with other state-of-art decoders. In Section VII we analyze the hardware implementation aspects for UF, MWPM and the proposed decoder. Finally, Section VIII presents simulation results.

II. Preliminaries

A. Stabilizer formalism

Consider the n-fold Pauli group,

$$\mathcal{G}_n \triangleq \{cB_1 \otimes \cdots \otimes cB_n : c \in \{\pm 1, \pm i\}, B_j \in \{I, X, Y, Z\}\},$$

where $I = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right], X = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right], Y = \left[\begin{smallmatrix} 0 & -i \\ i & 0 \end{smallmatrix}\right], Z = \left[\begin{smallmatrix} 1 & 0 \\ 0 & -1 \end{smallmatrix}\right]$. Every non-identity Pauli operator $P \in \mathcal{G}_n$ has eigenvalues ± 1 and any two Pauli operators in $\mathcal{G}_n$ either commute or anti-commute with each other. The weight of a Pauli operator P is defined as the number of non-identity elements in the tensor product.

By dropping the phase factor c, the Pauli group $\mathcal{G}_n$ is isomorphic to $\mathbb{F}_2^{2n}$ [10], such that

$$c\bigotimes_{i=1}^{n} X^{x_i} Z^{z_i} \mapsto (x_1, ..., x_n \mid z_1, ..., z_n). \tag{1}$$

In this representation, the commutation relation between two Pauli operators $\mathbf{p}_1 = (\mathbf{x_1}|\mathbf{z_1})$ and $\mathbf{p}_2 = (\mathbf{x_2}|\mathbf{z_2})$ can be computed by checking that the *symplectic inner product* is zero:

$$\mathbf{x_1 z_2}^T + \mathbf{z_1 x_2}^T = 0 \mod 2. \tag{2}$$

A stabilizer group $\mathcal{S}$ is an Abelian subgroup of $\mathcal{G}_n$, and an $[\![n, k, d]\!]$ stabilizer code is a 2^k-dimensional subspace $\mathcal{C}$ of the Hilbert space $(\mathbb{C}^2)^{\otimes n}$ that satisfies the condition $S_i |\Psi\rangle = |\Psi\rangle, \ \forall \ S_i \in \mathcal{S}, |\Psi\rangle \in \mathcal{C}$. Thus, the code $\mathcal{C}$ is defined as the common $+1$-eigenspace of the stabilizer group $\mathcal{S}$. The stabilizer group $\mathcal{S}$ is generated by a set of $n-k$ independent generators $S_1, ..., S_{n-k}$ that can be represented using a matrix $\mathbf{S}$, called the *stabilizer matrix*, whose (i, j) element is given by the Pauli operator corresponding to the j-th qubit in the i-th stabilizer. The minimum distance d is defined as the minimum weight of an element of $N(\mathcal{S}) \setminus \mathcal{S}$, where $N(\mathcal{S})$ is the normalizer[1] of $\mathcal{S}$. Elements in $N(\mathcal{S}) \setminus \mathcal{S}$ are also called *logical operators*.

Applying the mapping (1) to the stabilizer matrix $\mathbf{S}$, we obtain an $(n-k) \times 2n$ binary matrix:

$$\mathbf{H} = [\mathbf{H}_X \mid \mathbf{H}_Z] \tag{3}$$

which we call the *parity check matrix* of $\mathcal{C}$. Since the corresponding stabilizers of $\mathbf{S}$ commute with each other, it is easy to check using (2) that

$$\mathbf{H}_X \mathbf{H}_Z^T + \mathbf{H}_Z \mathbf{H}_X^T = \mathbf{0} \mod 2. \tag{4}$$

B. Error model and syndrome

In the depolarizing error model, errors are Pauli operators such that $\mathbf{e} \in \{I, X, Z, Y\}^{\otimes n}$; each error on the i-th qubit e_i can either be a bit flip (X), a phase flip (Z) or both (Y), each with probability $\epsilon/3$, while the probability of no error (I) is equal to $1-\epsilon$. Using the Pauli-to-binary mapping of (1), a Pauli error $\mathbf{e}$ can be also be modeled as a $1 \times 2n$ binary vector $\mathbf{e} = [\mathbf{e}_X \ \mathbf{e}_Z]$. Given a Pauli error $\mathbf{e}$, the corresponding syndrome $\mathbf{s} \in \{0, 1\}^{n-k}$ can be computed using the symplectic inner product such that

$$\mathbf{s} = \mathbf{e}_X \mathbf{H}_Z^T + \mathbf{e}_Z \mathbf{H}_X^T \mod 2. \tag{5}$$

Obviously, if $\mathbf{e} \in \mathcal{S}$ we have $\mathbf{s} = \mathbf{0}$. More generally, any Pauli error can be expressed as a combination of a *true error* $\mathbf{e}_t$, a stabilizer $\mathbf{S}_i$ and a logical operator $\mathbf{L}_j$ such that $\mathbf{e} = \mathbf{e}_t + \mathbf{S}_i + \mathbf{L}_j$, with $\mathbf{S}_i \in \mathcal{S}$ and $\mathbf{L}_j \in N(\mathcal{S}) \setminus \mathcal{S}$; since, by definition, elements of $\mathbf{L}_j$ and $\mathcal{S}_i$ commute with each other (thus, their syndrome is zero), the syndrome $\mathbf{s}$ is only dependent on the true error $\mathbf{e}_t$; moreover, any error estimate of the type $\hat{\mathbf{e}} = \mathbf{e}_t + \mathbf{S}_i$ is a valid error estimate. This phenomenon is known as *error degeneracy*. It is well known that, if the minimum distance of a stabilizer code is much higher than the weight of its stabilizer elements, there will be many degenerate errors of the same weight [15].

C. Calderbank-Shor-Steane codes

An $[\![n, k_X - k_Z, d]\!]$ Calderbank-Shor-Steane (CSS) code $\mathcal{C}$ is a stabilizer code constructed using two classical $[n, k_X, d_X]$ and $[n, k_Z, d_Z]$ codes C_X and C_Z, respectively, where $d \geq \min\{d_X, d_Z\}$ and $C_Z \subset C_X$ [16]. Note that k_X, k_Z and d_X, d_Z correspond to the dimensions and minimum distances of C_Z and C_X, respectively. The parity check matrix of the CSS code $\mathcal{C}$ has the form

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_Z & \mathbf{0} \\ \mathbf{0} & \mathbf{H}_X \end{bmatrix}, \tag{6}$$

where $\mathbf{H}_X$ and $\mathbf{H}_Z$ are the parity check matrices of C_X and C_Z, respectively, and the commutativity condition of (4) reduces to $\mathbf{H}_X \mathbf{H}_Z^T = \mathbf{0} \mod 2$. To correct depolarizing errors on the qubits, a syndrome $\mathbf{s}$ is computed such that

$$\mathbf{s} = [\mathbf{s}_X \ \mathbf{s}_Z], \tag{7}$$

[1]The normalizer subgroup $N(\mathcal{S})$ of a group $\mathcal{S}$ is the subgroup of $\mathcal{S}$ which is invariant under conjugation.

where $\mathbf{s}_X = \mathbf{e}_X \mathbf{H}_Z^T \mod 2$ and $\mathbf{s}_Z = \mathbf{e}_Z \mathbf{H}_X^T \mod 2$. Because of the structure of $\mathbf{H}$, X and Z errors can be corrected independently using $\mathbf{H}_Z$ and $\mathbf{H}_X$, respectively. In this paper we consider a bit-flip channel, where each qubit experiences an X error with probability p, and remains correct with probability $1-p$. In other words, we fix $\mathbf{e}_Z = \mathbf{0}$, while elements in $\mathbf{e}_X$ can be 1 with probability p or 0 with probability $1-p$. Hence we only consider $\mathbf{H}_Z$ and $\mathbf{s}_X$, which for simplicity we will denote as $\mathbf{H}$ and $\mathbf{s}$, respectively. Note that the bit-flip channel that we consider and the depolarizing error model are closely related: indeed, assuming that X and Z errors are decoded separately, we can model the depolarizing channel as two binary symmetric channels with probability of error $p_x = p_z = \frac{2}{3}\epsilon$; assuming that $d_X = d_Z$, is possible to switch from the logical error rate curve over a binary symmetric channel to the logical error rate under depolarizing noise by re-scaling it of a factor of $3/2$ [17].

It is convenient to introduce the notion of *Tanner graph* [18]. A Tanner graph is a bipartite graph defined from $\mathbf{H}$, such that it has two sets of nodes $V = \{v_1, v_2, ..., v_n\}$ and $C = \{c_1, c_2, ..., c_{n-k}\}$ called *variable nodes* and *check nodes*, respectively, and there is an edge between v_j and c_i if and only if $h_{i,j} = 1$. A check node c_i is said to be *satisfied* if $s_i = 0$, and *unsatisfied* if $s_i = 1$. The degree of a variable or check node is defined as the number of its incident edges. We define the distance between two nodes i and j to be the number of variable nodes belonging to the shortest path between i and j.

D. Surface and toric codes

Surface codes [1] are a widely known class of CSS codes. These are derived from the tessellation of the topological surface in squares, in such a way as to form a lattice. Generally, a surface code is characterized by an $L \times L$ lattice, where L is the size of the horizontal (or vertical) dimension. In the lattice we can identify vertices as X checks, edges as qubits and squares as Z checks. Although there are several types of surface codes, in this paper we focus on the rotated planar codes [19], with parameters $[\![L^2, 1, L]\!]$ (L being the size of the lattice) for the description of the decoder. The matrix $\mathbf{H}_X$ is the incidence matrix between vertices and edges, and the matrix $\mathbf{H}_Z$ is the incidence matrix between squares and edges. The two Tanner graphs corresponding to $\mathbf{H}_X$ and $\mathbf{H}_Z$ are isomorphic and they obviously correspond to a lattice as well, as depicted in Fig. 1 for the rotated planar code. We also consider toric codes, which are again characterized by a $L \times L$ lattice, such that the edges and vertices at the boundaries coincide. The toric code has parameters $[\![2L^2, 2, L]\!]$.

III. Decoding of Toric and Surface Codes

In this section we summarize the state-of-the-art decoders for toric and surface codes, and motivate our contribution.

The decoding of toric and surface codes is usually carried out using the MWPM algorithm [2]. After measuring the syndrome, a complete graph is constructed, where each vertex represents an unsatisfied check; each edge is assigned a weight, which is the minimum number of qubits separating the two checks connected by the edge; this weight assignment

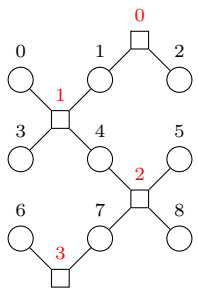


Fig. 1: Tanner graph for the $[\![9, 1, 3]\!]$ rotated planar code. Circles (with black labels) correspond to variable nodes, while squares (with red labels) correspond to check nodes.

is done using Dijkstra's algorithm [20]. Then, the *blossom algorithm* [21] is run on this graph, with the goal of finding the minimum weight perfect matching. Since the overall complexity of the MWPM decoder on an $L \times L$ lattice is $\mathcal{O}(L^6 \log L)$ [22], many efficient implementations have been proposed to reduce this complexity: for instance, in [2] a fully parallel implementation is proposed, able to run with a complexity of $\mathcal{O}(1)$, given a uniform 2-D array of finite speed processing elements and an external memory able to store all the detection events and matching data for the entire duration of a hypothetical computation, while Higgott *et al.* proposed, in [22], that each unsatisfied check could be restricted to be matched within a local neighborhood, or that Dijkstra's algorithm could be optimized to avoid an all-to-all search [23]. The main competitor of MWPM is the UF decoder [3]. This decoding scheme is also divided into two parts: first, the *syndrome validation* process is carried out, where *clusters* are defined to initially correspond to each unsatisfied check, and are then iteratively grown to absorb the nearest check neighbors. A cluster stops growing when it contains an even number of unsatisfied checks, and two clusters merge together if they touch each other. After all the clusters have stopped growing, the *peeling decoder* is run on each cluster. The complexity of the UF decoder, in its efficient implementation proposed in [3], is almost linear in the blocklength. There are also several alternative approaches to UF, which are able to improve the decoding latency. For instance, in [24], the authors propose a spanning tree (ST) matching decoder which is able to correct up to the code distance. In addition, they also propose a lower complexity decoder, called the *rapid-fire decoder*, which is able to improve the decoding latency at the cost of some performance degradation.

Message passing decoders such as BP are, in principle, not suitable for topological codes, mainly because of the presence of weight-4 symmetric stabilizers that behave as trapping sets for the decoder [25]; because of this, several techniques have been developed to improve the performance of BP on these codes. One of the most common approaches is to apply the well-known ordered statistics technique after several rounds of BP [10]; this decoder is known as BP-OSD and has a complexity of $\mathcal{O}(n^3)$, thus making it infeasible for large-scale implementation.

For the majority of decoders proposed for topological codes, low decoding error probability comes with the price

of increased decoding complexity. MWPM, for instance, has a complexity that is cubic in the blocklength and it is known to be infeasible in practice; the efficient implementations available, in particular those of [22] and [23], offer a significant speedup that still vanishes for large blocklength. Another major issue in decoders for topological codes, which is often overlooked, is the hardware implementation aspect of the algorithms. In fact, the vast majority of these decoders turn out to be inefficient when implemented on FPGAs. It is well-known that fixed and predictable access to pre-allocated memory is much faster than random access to irregular, pointer-based dynamic memory; in particular, FPGAs are particularly suitable for the former type of memory access. Moreover, modifications of dynamic structures (like adding nodes to a tree, or modifying its connectivity) are also operations which are known to introduce a high amount of latency, even when parallelized, due to the high overhead required. Another aspect to consider is that the usage of dynamic memories is usually more expensive in terms of energy consumption, as they often require additional overhead for their management, as well as more complex hardware. Also, energy consumption is a critical aspect of any quantum error correction decoder, as it runs on a chip which is embedded in a cryogenic environment with a strict power budget. MWPM and spanning tree decoders, for example, make use of Dijkstra's algorithm to construct syndrome graphs on which they perform the perfect matching. Dijkstra's algorithm is not desirable for an efficient FPGA implementation, primarily due to its inherently sequential nature and dependence on dynamic data structures (priority queues).

Similarly, the UF decoder relies on dynamic, pointer-based data structures and complex tree manipulations. The two operations, `union()` and `find()`, often involve traversing and modifying tree structures, which require frequent pointer updates and non-uniform memory access patterns.

In this scenario, it seems reasonable to revisit the BF decoder which utilizes extremely simple update rules that make it much easier to implement in hardware such as FPGA. However, because of the low degree of variable nodes in the Tanner graph of the considered quantum codes (toric codes are regular with variable degree of 2, while surface codes also have degree-1 nodes), and because of error degeneracy, we need to make significant modifications to the traditional BF algorithm in order to decode toric and surface codes. We will show that, although these modifications impact on the asymptotic complexity of the decoder, they significantly improve the performance of the decoder comparing to that which uses traditional BF; at the same time, our algorithm does not make use of any dynamic data structures, and only utilizes elementary operations on arrays of fixed length which can be pre-allocated in memory, thus making our decoder particularly suitable for efficient hardware implementation compared to the alternatives. Nevertheless, the goal of this paper is to merely present and explain the algorithm, without presenting an explicit hardware implementation; an optimized hardware implementation, together with a detailed comparison with those of other decoders, requires a specific in-depth study and is therefore left for future work.

IV. The proximity vector

In this section we introduce the *proximity vector*, a vector that the decoder uses as a bit flipping criterion. We describe the rationale behind it, and we describe how it can be efficiently computed while decoding.

The proximity vector is a heuristic weight assignment to the variable nodes and check nodes in the Tanner graph, and we define it to be the sum of multiple contributions which we call *proximity influences*, which we describe hereafter. Let c_j be an unsatisfied check; we want variable and check nodes neighboring c_j to have the highest weights, while the weights should decrease with increasing distance from c_j. A simple way to achieve this is by computing the proximity influence recursively, as follows:

Definition 1. *Let c_j be an unsatisfied check, and let all the other check nodes be satisfied. We define $\boldsymbol{\gamma}^{(0)}(c_j)$ to be a $1\times m$ vector such that*

$$\gamma_i^{(0)}(c_j) = \begin{cases} 1 \text{ for } i = j \\ 0 \text{ otherwise} \end{cases}, \tag{8}$$

and let $\boldsymbol{\nu}^{(0)}(c_j)$ be a $1 \times n$ vector such that

$$\boldsymbol{\nu}^{(0)} = \boldsymbol{\gamma}^{(0)} \cdot \mathbf{H} \ . \tag{9}$$

Then, let $\boldsymbol{\gamma}^{(\ell)}(c_j)$ and $\boldsymbol{\nu}^{(\ell)}(c_j)$ be defined recursively as

$$\begin{cases} \boldsymbol{\gamma}^{(\ell)}(c_j) = \boldsymbol{\nu}^{(\ell-1)}(c_j) \cdot \mathbf{H}^T \\ \boldsymbol{\nu}^{(\ell)}(c_j) = \boldsymbol{\gamma}^{(\ell)}(c_j) \cdot \mathbf{H} \ , \end{cases} \tag{10}$$

for $\ell = 1, 2, ..., D$, D being a fixed positive integer. We call $\boldsymbol{\nu}^{(\ell)}(c_j)$ the proximity influence of c_j on qubits (or variable nodes) of depth ℓ, and $\boldsymbol{\gamma}^{(\ell)}(c_j)$ the proximity influence of c_j on checks of depth ℓ.

We then combine the proximity influence of each unsatisfied check node to obtain the proximity vectors by adding, for each variable and check node, the values of each proximity influence.

Definition 2. *Let $c_1, ..., c_m$ be all the unsatisfied check nodes, and let $\boldsymbol{\nu}^{(D)}(c_1), ..., \boldsymbol{\nu}^{(D)}(c_m)$ and $\boldsymbol{\gamma}^{(D)}(c_1), ..., \boldsymbol{\gamma}^{(D)}(c_m)$ be the respective proximity influences on qubits and checks. We define $\boldsymbol{\nu}^{(D)}$ and $\boldsymbol{\gamma}^{(D)}$ to be the proximity vectors on qubits and check nodes of depth D, respectively, such that:*

$$\begin{cases} \boldsymbol{\nu}^{(D)} = \sum_{i=1}^m \boldsymbol{\nu}^{(D)}(c_i) \\ \boldsymbol{\gamma}^{(D)} = \sum_{i=1}^m \boldsymbol{\gamma}^{(D)}(c_i) \end{cases} . \tag{11}$$

It is also possible to define $\boldsymbol{\nu}^{(D)}$ and $\boldsymbol{\gamma}^{(D)}$ by setting $\boldsymbol{\gamma}^{(0)} = \mathbf{s}$, and then using (9) and (10) directly.

Since the value of D will be fixed in the rest of the paper, we simply refer to the proximity influences as $\boldsymbol{\nu}(c_j)$ and $\boldsymbol{\gamma}(c_j)$, and to the vectors as $\boldsymbol{\nu}$ and $\boldsymbol{\gamma}$. By construction, the values of the influences $\boldsymbol{\nu}(c_j)$ and $\boldsymbol{\gamma}(c_j)$ are highest for the nearest neighbors of c_j, and decrease with increasing distance of a variable (check) node from c_j. To see this, it is convenient to write $\boldsymbol{\nu}^{(D)}, \boldsymbol{\gamma}^{(D)}$ in the following recursive way. Let $\mathcal{N}(c_j) \subseteq V$ be the neighborhood of check node c_j, and $\mathcal{N}(v_j) \subseteq C$ the neighborhood of variable node v_j; we define $\mathcal{N}^d(c_j) \subseteq V$ to be the depth-d neighborhood of c_j, *i.e.*, the shortest path between c_j and its depth-d neighbors has length $2d-1$; we

can also define $\mathcal{N}^d(v_j) \subseteq C$ in a similar way. For every check node $c_k \in C$ and variable node $v_q \in V$:

$$\begin{cases} \gamma_k^{(i)}(c_j) = \gamma_k^{(i-1)}(c_j) + \sum_{v_z \in \mathcal{N}(c_k)} \nu_z^{(i-1)}(c_j) \\ \nu_q^{(i)}(c_j) = \nu_q^{(i-1)}(c_j) + \sum_{c_z \in \mathcal{N}(v_q)} \gamma_z^{(i-1)}(c_j). \end{cases} \tag{12}$$

For $i = 0$ we have $\nu_q^{(0)}(c_j) = 1$ for all variable nodes $v_q \in \mathcal{N}(c_j)$ and 0 otherwise; for $i = 1$ we have $\nu_q^{(1)}(c_j) = 1$ for all variable nodes $v_q \in \mathcal{N}^2(c_j)$; moreover, $\nu_q^{(1)}(c_j) > 1$ for all variable nodes $v_q \in \mathcal{N}(c_j)$. Let $r > i+1$ and $m < i+1$; in general we have $\nu_q^{(i)}(c_j) = 0$ for all variable nodes in $\mathcal{N}^r(c_j)$, $\nu_q^{(i)}(c_j) = 1$ for all variable nodes in $\mathcal{N}^{i+1}(c_j)$, and $\nu_q^{(i)}(c_j) > 1$ for variable nodes in $\mathcal{N}^m(c_j)$. In other words, the value of the proximity influence for each node at distance $2d-1$ from c_j remains 0 until the $d+1$ iteration, and then increases according to (12). This means that, fixing the number of iterations, the closer is a node to c_j, the higher is the proximity influence for that node. A similar argument can be made for $\boldsymbol{\gamma}(c_j)$.

The proximity vector on qubits $\boldsymbol{\nu}$ and on checks $\boldsymbol{\gamma}$ is the natural superposition of all of the influences exerted by all of the unsatisfied checks. As a result, variable and check nodes which are more distant from unsatisfied nodes (we say more *isolated*) will have a lower weight assigned, while variable and check nodes that are near to many unsatisfied checks will have higher weights assigned. Note that this definition of proximity vector is heuristic, and alternative definitions can be provided to capture the individual influence of a check node. Our definition of proximity vector is convenient for the reason that it only involves integer values, thus it can be stored using a fixed (and low) number of bits.

1) Efficient computation of the proximity vector

In our decoder, it is essential to update the proximity vectors on checks and qubits after each flip: in other words, after a bit is flipped, its neighboring checks will be satisfied[2], and their influence should be removed from the proximity vector. Specifically, if the flipping of variable node v_i causes its two neighboring check nodes c_j and c_k to become satisfied, the updated proximity vectors shall be

$$\begin{cases} \boldsymbol{\gamma}' = \boldsymbol{\gamma} - (\boldsymbol{\gamma}(c_j) + \boldsymbol{\gamma}(c_k)) \\ \boldsymbol{\nu}' = \boldsymbol{\nu} - (\boldsymbol{\nu}(c_j) + \boldsymbol{\nu}(c_k)). \end{cases} \tag{13}$$

To take into account this update of the vector after each flip, one could in principle recompute (11), setting $\boldsymbol{\gamma}^{(0)} = \mathbf{s}'$, where $\mathbf{s}'$ is the updated syndrome after the bit flip; however, this would negatively impact the complexity of the algorithm, as each bit flip would require an additional $\ell(nd_v + md_c)$ operations (multiplication of sparse matrices).

Instead, we exploit the labeling we have assigned to variable and check nodes in the surface code. The idea is to pre-compute the proximity influence of an arbitrary check node over a larger surface code, say $\boldsymbol{\gamma}(c_1), \boldsymbol{\nu}(c_1)$, store it prior to the decoding process, and then compute online $\boldsymbol{\gamma}(c_i), \boldsymbol{\nu}(c_i)$ when needed by appropriately permuting $\boldsymbol{\gamma}(c_1)$ and $\boldsymbol{\nu}(c_1)$. We illustrate this process for the rotated surface code, although it can be extended for other types of surface codes. Assume that we label the variable nodes with non-negative integers row-wise in increasing order and we do the same for the check nodes; an example is provided in Fig. 1 for a rotated code with $L = 3$. Consider the Tanner graphs $\mathcal{G}_1$ of a rotated code with $L = L_1$, and $\mathcal{G}_2$ of an $L_2' \times L_2''$ surface code , such that $L_2 > L_1$ and $L_2'' > L_1$. Let $\mathcal{L}_v^{(1)}, \mathcal{L}_c^{(1)} \in \mathbb{N}^{L_1^2}$ be the labels of the variable and check nodes of $\mathcal{G}_1$, and $\mathcal{L}_v^{(2)}, \mathcal{L}_c^{(2)} \in \mathbb{N}^{L_2^2}$ be the labels of the variable and check nodes of $\mathcal{G}_2$, respectively. There are two injective mappings $\phi_c : \mathcal{L}_c^{(1)} \to \mathcal{L}_c^{(2)}$ and $\phi_v : \mathcal{L}_v^{(1)} \to \mathcal{L}_v^{(2)}$ from the labels of the check and variable nodes of $\mathcal{G}_1$ to the labels of the check and variable nodes of $\mathcal{G}_2$. Let $i, j \in \mathcal{L}_c^{(1)}$. Assume that we have calculated $\boldsymbol{\nu}(c_{\phi_c(i)})$ and $\boldsymbol{\gamma}(c_{\phi_c(i)})$ for some arbitrary check node c_i using (10) on $\mathcal{G}_2$, and that we now wish to compute $\boldsymbol{\nu}(c_j)$ and $\boldsymbol{\gamma}(c_j)$, for some $j \neq i$, by appropriately permuting $\boldsymbol{\nu}(c_{\phi_c(i)})$ and $\boldsymbol{\gamma}(c_{\phi_c(i)})$. Let ρ_c be the number of check in each row of $\mathcal{G}_2$ and ρ_{vo}, ρ_{ve} be the number of variable nodes in even and odd rows of $\mathcal{G}_2$, respectively. By exploiting the indexing we have defined for the check and variable nodes, we can define a *vertical* shift σ_y and an *horizontal* shift σ_x:

$$\begin{cases} \sigma_y = \lfloor \frac{\phi_c(j)}{\rho_c} \rfloor - \lfloor \frac{\phi_c(i)}{\rho_c} \rfloor, \\ \sigma_x = \phi_c(j) \mod \rho_c - \phi_c(i) \mod \rho_c , \end{cases} \tag{14}$$

Note that σ_x and σ_y can also assume negative values. Depending on the surface code we are considering, it is possible to express the index transformation in closed form, using σ_x and σ_y. Let $k_c, k_c' \in \mathcal{L}_c^{(2)}$ and $k_v, k_v' \in \mathcal{L}_v^{(2)}$. We can express a coordinate shift using two linear maps from k_c to k_c' and from k_v to k_v' respectively, such that

$$\begin{cases} k_c' = k_c + \sigma_x + \sigma_y \rho_c \\ k_v' = k_v + \sigma_x + \sigma_y(\rho_{vo} + \rho_{ve}). \end{cases} \tag{15}$$

Thus, we have $\nu_{k_v'}(c_{\phi_c(j)}) = \nu_{k_v}(c_{\phi_c(i)})$ and $\gamma_{k_c'}(c_{\phi_c(j)}) = \gamma_{k_c}(c_{\phi_c(i)})$, for all k_c' and k_v'; finally, we apply the inverse mappings ϕ_c^{-1}, ϕ_v^{-1} to all the elements in the codomain of these functions, such that $\nu_q(c_j) = \nu_{\phi_v(q)}(c_{\phi_c(j)})$ and $\gamma_p(c_j) = \gamma_{\phi_c(p)}(c_{\phi_c(j)})$ for any $q \in \mathcal{L}_v^{(1)}$ and $p \in \mathcal{L}_c^{(1)}$. This procedure is illustrated in Algorithm 1, where $j \in \mathcal{L}_c^{(1)}$ is the label of the check nodes of which we want to compute the proximity vectors. An example of the application of Algorithm 1 is illustrated in Fig. 2. Specifically, we pick $\mathcal{G}_1$ to be a $L_1 = 3$ rotated code and $\mathcal{G}_2$ to be a 4×5 surface code. The example illustrates how to shift the proximity vectors from $c_{\phi_c(0)=8}$ to $c_{\phi_c(2)=15}$. In Fig. 2(a) the proximity metrics $\boldsymbol{\gamma}(c_{\phi_c(0)})$ and $\boldsymbol{\nu}(c_{\phi_c(0)})$ are assumed to be pre-computed; in Fig. 2(b) we want to compute $\boldsymbol{\gamma}(c_2)$ and $\boldsymbol{\nu}(c_2)$, which is mapped to $\boldsymbol{\gamma}(c_{\phi_c(2)=15})$ and $\boldsymbol{\nu}(c_{\phi_c(2)=15})$. We compute $\sigma_x = 1$ and $\sigma_y = 1$ and apply (15) to each variable and check node. The reader can verify that, for instance, check node 7 is mapped to check node 14, and thus to check node $\phi_c^{-1}(14) = 1$ in the $L = 3$ surface code. On the other hand, check node 2 is mapped to check node 9, which doesn't belong to the image of ϕ_c, and thus $\boldsymbol{\gamma}(c_{\phi^{-1}(c_9)}) = \emptyset$. The values for L_2', L_2'' are such that the mapping in (14) gives the correct result for every σ_x, σ_y, which happens for $L_2', L_2'' > L + D$.

2) Auxiliary proximity influence

We also define an auxiliary proximity influence of a check node c_j, that we denote as $\boldsymbol{\alpha}(c_j)$.

[2]Because of the nature of our decoder, each bit flip never generates new unsatisfied checks.

Algorithm 1 `Shift_influence`

Input: j
Global: $\boldsymbol{\gamma}(c_{\phi_c(1)})$, $\boldsymbol{\nu}(c_{\phi_c(1)})$, L_2, ϕ_v, ϕ_c
Output: $\boldsymbol{\gamma}(c_j)$, $\boldsymbol{\nu}(c_j)$

1: $\mathbf{k}_v, \mathbf{k}_c \leftarrow [1, L_2^2]$
2: Compute σ_x, σ_y using (14)
3: Compute $\mathbf{k}'_v, \mathbf{k}'_c$ using (15)
4: $\boldsymbol{\nu}_{\phi_v^{-1}(\mathbf{k}'_v)}(c_j) \leftarrow \boldsymbol{\nu}_{\mathbf{k}_v}(c_{\phi_c(1)})$
5: $\boldsymbol{\gamma}_{\phi_c^{-1}(\mathbf{k}'_c)}(c_j) \leftarrow \boldsymbol{\gamma}_{\mathbf{k}_c}(c_{\phi_c(1)})$
6: **return**

Definition 3. *The auxiliary proximity influence $\boldsymbol{\alpha}(c_j)$ is a $1 \times n$ vector such that $\alpha_i(c_j)$ is the length of the shortest path between the variable node v_i and the check node c_j. For example, if a variable node v_i is directly connected to c_j, then $\alpha_i(c_j) = 1$.*

Note that the proximity influences $\boldsymbol{\nu}(c_j)$ and $\boldsymbol{\alpha}(c_j)$ share the same non-zero positions, but in general they have different values: while $\boldsymbol{\nu}(c_j)$ will have higher values for the variable nodes close to c_j, in $\boldsymbol{\alpha}(c_j)$ the variable nodes connected to c_j will have value 1, those at distance 2 will have value 2, and so on. We can compute $\boldsymbol{\alpha}(c_j)$ in a similar way to $\boldsymbol{\nu}(c_j)$. Thus, it is sufficient to construct $\boldsymbol{\alpha}(c_1)$ offline, and apply Algorithm 1 to create $\boldsymbol{\alpha}(c_j)$, for any j. We illustrate in Section V-B how we make use of the auxiliary proximity influence to correct errors. Finally, we define two subroutines, illustrated in Algorithm 2 and Algorithm 3. The `Compute_proximity_vector` subroutine takes as input the syndrome $\mathbf{s}$ and generates the proximity vectors $\boldsymbol{\nu}$ and $\boldsymbol{\gamma}$, using Algorithm 1 to compute individual influences and summing them up according to (11). The `Shift_and_remove` subroutine takes as input the residual syndrome $\mathbf{s}'$, as well as the proximity vectors $\boldsymbol{\nu}$ and $\boldsymbol{\gamma}$, and updates them using (13).

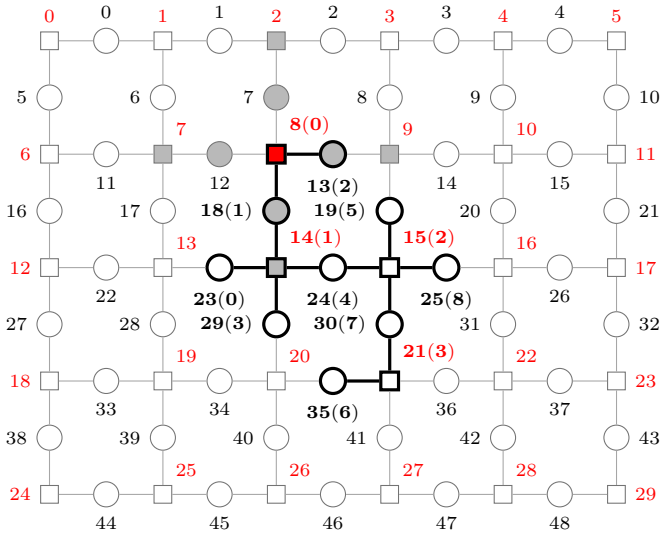
Algorithm 2 `Compute_proximity_vector`

Input: $\mathbf{s}$
Global: $\boldsymbol{\gamma}(c_1)$, $\boldsymbol{\nu}(c_1)$, L
Output: $\boldsymbol{\nu}$, $\boldsymbol{\gamma}$

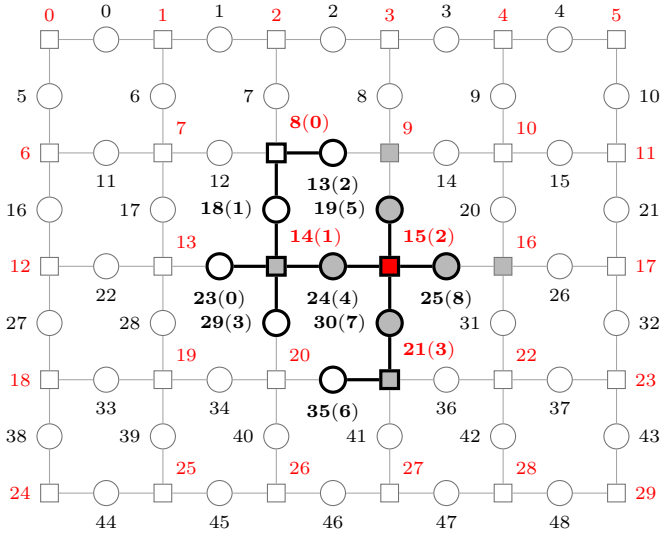
1: $\mathbf{k} \leftarrow k \mid s_k = 1$
2: $\boldsymbol{\nu}, \boldsymbol{\gamma} \leftarrow \mathbf{0}$
3: **for** $k \in \mathbf{k}$ **do**
4: $\quad \boldsymbol{\gamma}(c_k), \boldsymbol{\nu}(c_k) \leftarrow$ `Shift_influence`(k)
5: $\quad \boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma} + \boldsymbol{\gamma}(c_k)$
6: $\quad \boldsymbol{\nu} \leftarrow \boldsymbol{\nu} + \boldsymbol{\nu}(c_k)$
7: **end for**
8: **return**

V. Progressive-Proximity Bit-Flipping

We are now ready to describe our proposed decoder, which we call *Progressive-Proximity Bit-Flipping* (PPBF). It is illustrated in Algorithm 4, and it is composed of two decoding steps: the first is called *Preliminary BF*, and the second is called *Iterative matching*; these two algorithms are illustrated in Algorithm 5 and Algorithm 6, respectively.



(a) The depth-1 influence of c_8 (highlighted in red) is computed on a 4×5 surface code. The node label of c_8 corresponds to the node c_0 in the $L = 3$ rotated code.



(b) The influence of Fig. 2(a) is shifted to the node c_{15} on the 4×5 surface code: in this case, $\sigma_x = 1$ and $\sigma_y = 1$. Then, the values of the shifted metrics are assigned to the nodes of the $L = 3$ rotated code according to the illustrated mapping between labels.

Fig. 2: Example of shifting of the proximity influence. The labeling on the variable (black) and check (red) nodes inside the parentheses is the labeling for the $L = 3$ rotated code, while the labeling outside the parentheses is the one for the 4×5 surface code. In this example, we compute the influence of $c_{15(2)}$ by shifting the influence of $c_{8(0)}$. The nodes colored in gray are the ones with a non-zero proximity metric; notice how some of the values are discarded after shifting, i.e. the values of c_9 and c_{16}.

A. Preliminary BF

In the preliminary BF step of Algorithm 5, the vector $\mathbf{u}$ containing the number of unsatisfied checks for each variable node is first computed (line 3), and only variable nodes

Algorithm 3 `Shift_and_remove`

Input: $\boldsymbol{\nu}$, $\boldsymbol{\gamma}$, $\mathbf{s}'$
Global: $\boldsymbol{\gamma}(c_1)$, $\boldsymbol{\nu}(c_1)$, L
Output: $\boldsymbol{\nu}'$, $\boldsymbol{\gamma}'$

1: $\mathbf{j} \leftarrow j \mid s'_j = 1$
2: $\boldsymbol{\nu}' \leftarrow \boldsymbol{\nu}$
3: $\boldsymbol{\gamma}' \leftarrow \boldsymbol{\gamma}$
4: **for** $k \in \mathbf{j}$ **do**
5: $\quad \boldsymbol{\gamma}(c_k), \boldsymbol{\nu}(c_k) \leftarrow$ `Shift_influence`(k)
6: $\quad \boldsymbol{\nu}' \leftarrow \boldsymbol{\nu}' - \boldsymbol{\nu}(c_k)$
7: $\quad \boldsymbol{\gamma}' \leftarrow \boldsymbol{\gamma}' - \boldsymbol{\gamma}(c_k)$
8: **end for**
9: **return**

Algorithm 4 Progressive-Proximity Bit-Flipping

Input: $\mathbf{s}$, $\mathbf{H}$
Output: $\hat{\mathbf{e}}$

1: $\boldsymbol{\nu}, \boldsymbol{\gamma} \leftarrow$ `Compute_proximity_metric`$(\mathbf{s})$
2: $\hat{\mathbf{e}}, \hat{\mathbf{s}}, \boldsymbol{\nu}' \leftarrow$ `Preliminary_BF`$(\mathbf{s}, \mathbf{H}, \boldsymbol{\nu})$
3: $\hat{\mathbf{e}} \leftarrow$ `Iterative_matching`$(\hat{\mathbf{s}}, \mathbf{H}, \boldsymbol{\gamma})$
4: **return**

involved in two unsatisfied checks will be considered for flipping. Among all the variable nodes i such that $u_i = 2$, we flip the one which achieves the minimum value of the proximity vector ν'_i (lines 7-8); then, the proximity vector is updated using Algorithm 3, such that the influence of the checks satisfied after the flip is removed from $\boldsymbol{\nu}'$ (line 10). The residual syndrome is then computed (line 11) and another iteration is performed, until there are no variable nodes i such that $u_i = 2$.

Algorithm 5 `Preliminary_BF`

Input: $\mathbf{s}$, $\mathbf{H}$, $\boldsymbol{\nu}$
Output: $\hat{\mathbf{e}}$, $\hat{\mathbf{s}}$, $\boldsymbol{\nu}'$

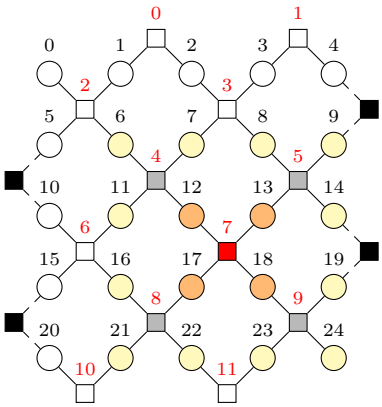
1: $\hat{\mathbf{e}} \leftarrow \mathbf{0}$
2: $\hat{\mathbf{s}} \leftarrow \mathbf{s}$
3: $\mathbf{u} \leftarrow \mathbf{s} \cdot \mathbf{H}$
4: $\boldsymbol{\nu}' \leftarrow \boldsymbol{\nu}$
5: $\mathcal{S} \leftarrow \{i \in [1, n] : u_i = 2\}$
6: **while** $\mathcal{S} \neq \emptyset$ **do**
7: $\quad j \leftarrow \arg\min_{i \in \mathcal{S}} \nu'_i$
8: $\quad \hat{e}_j \leftarrow \hat{e}_j \oplus 1$
9: $\quad \mathbf{s}' \leftarrow \hat{\mathbf{e}} \cdot \mathbf{H}^T \mod 2$
10: $\quad \boldsymbol{\nu}' \leftarrow$ `Shift_and_remove`$(\boldsymbol{\nu}', \cdot, \mathbf{s}')$
11: $\quad \hat{\mathbf{s}} \leftarrow \mathbf{s} \oplus \mathbf{s}'$
12: $\quad \mathbf{u} \leftarrow \hat{\mathbf{s}} \cdot \mathbf{H}$
13: **end while**
14: **return** $\hat{\mathbf{e}}$, $\hat{\mathbf{s}}$

B. The iterative-matching routine

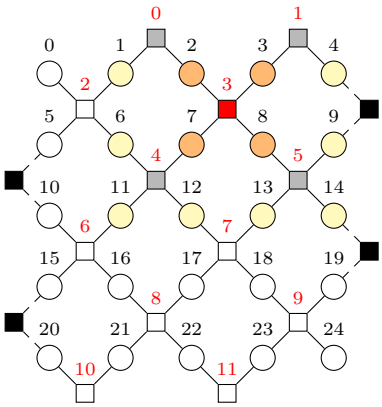
Here we present the iterative matching routine, which is specified in Algorithm 6. A matching decoder is a decoding algorithm specifically tailored for surface codes, which aims to pair together couples of unsatisfied checks, estimating the qubits on the shortest path connecting them as being in error. This comes from the fact that every error on the surface code can be interpreted as a path on the lattice, and the two endpoints of this path are the two unsatisfied checks generated by the error; however, errors occurring on boundary qubits only generate one unsatisfied check. To address this issue, it is common to add additional "dummy" unsatisfied check nodes to the boundaries of the surface code, as proposed in [2] (notice that this is not necessary for the toric code). These checks are illustrated in black in Fig. 3. We utilize the proximity vector $\boldsymbol{\gamma}$ to match pairs of unsatisfied checks together; specifically, we start by identifying the unsatisfied check c_i with the lowest proximity vector entry γ_i (line 4) calling it a *pivot node*, and compute its auxiliary proximity influence $\boldsymbol{\alpha}(c_i)$. Among all the other unsatisfied checks, we pick the one at the smallest distance from the pivot (if there is more than one candidate at the same distance, we choose the check c_j with lowest proximity vector entry γ_j); we call this the *target node* (line 9), and denote its distance from the pivot by δ. After computing the auxiliary vector $\boldsymbol{\alpha}(c_j)$, we compute $\boldsymbol{\alpha}(c_i) + \boldsymbol{\alpha}(c_j)$; the result of this operation is a vector such that if the k-th element is equal to $\delta + 1$, the variable node v_k belongs to the shortest path between c_i and c_j. If the number of entries of $\boldsymbol{\alpha}(c_i) + \boldsymbol{\alpha}(c_j)$ that are equal to $\delta + 1$ is equal to δ, it means that there is only one shortest path between c_i and c_j, that corresponds to the most likely error matching that syndrome; on the other hand, if the number of entries of $\boldsymbol{\alpha}(c_i) + \boldsymbol{\alpha}(c_j)$ that are equal to $\delta + 1$ is larger than δ, it means that the corresponding error is degenerate, as there is more than one shortest path between c_i and c_j. An example of this is illustrated in Fig. 3, where there are two possible paths of length 2 between c_3 and c_7, and the number of variable nodes such that $\boldsymbol{\alpha}(c_3) + \boldsymbol{\alpha}(c_7) = 3$ is more than 2. In the first case, it is sufficient to flip all the variable nodes associated with the value $\delta + 1$, while in the latter case one of the possible degenerate errors must be chosen. We distinguish these two scenarios in line 13. To pick one among the possible paths, we compute the auxiliary influence of a third check node c_z, of position such that it is horizontally aligned with c_i and vertically aligned with c_j; lines 16-22 are dedicated to computing the position z. In particular, we define σ_x to be the *horizontal shift* between c_i and c_j and σ_y to be the *vertical shift* between c_i and c_j. Once $\boldsymbol{\alpha}(c_z)$ is computed in lines 22-23, it is easy to check that the variable nodes v_k such that $\alpha_k(c_i) + \alpha_k(c_z) = \sigma_x + 1$ are those connecting c_i and c_z, and those such that $\alpha_k(c_j) + \alpha_k(c_z) = \sigma_y + 1$ are those connecting c_j and c_z, thus we can flip all of them simultaneously in line 26.

C. Decoding of toric codes

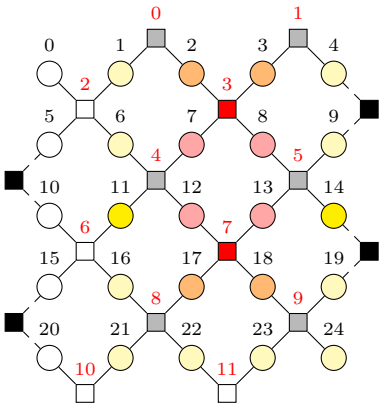
In the case of toric codes it is possible to simplify the computation of the proximity vectors. Consider a $[\![2L^2, 2, L]\!]$ toric code. The proximity vector of an arbitrary check node is pre-computed on the same code (there is no need to use a larger code), therefore for each node label i we have $\phi(i) = i$; to perform the permutation from a node i to a node j, σ_x and σ_y are computed using (14), and for each



(a) Proximity influence of depth 2 for the check node c_7; different colorings of variable nodes correspond to their distance from c_7: orange nodes are at distance 1, yellow at distance 2.



(b) Proximity influence of depth 2 for c_3.



(c) Combination of the previous two proximity influences: variable nodes which were assigned yellow and orange are now colored red; those nodes which were assigned yellow and yellow are now deep yellow. Finally, the variable nodes which were assigned orange/white or yellow/white have maintained their color. The set of red variable nodes coincide with the union of all the possible error patterns that satisfy the matching, *i.e.*, $\{v_7, v_{12}\}$ and $\{v_8, v_{13}\}$.

Fig. 3: Example of the usage of the auxiliary proximity influence of checks in Algorithm 6. In the example, c_3 and c_7 are unsatisfied checks, and the decoder has to find the shortest path between them, assuming it starts with c_7 as pivot.

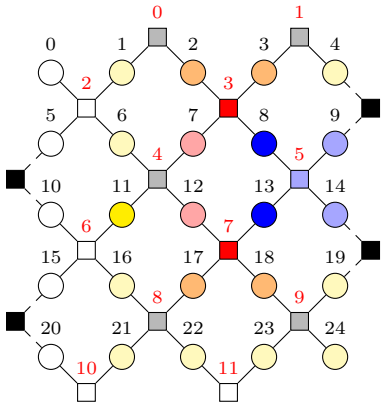


Fig. 4: Addition of the extra check as described in Section V-B. The blue variable nodes are the depth-1 influence of the extra check, and the darker ones are those which will be flipped. In the example, $\sigma_x = \sigma_y = 1$.

$k_c, k_v, k_c', k_v' \in [1, 2L^2]$ we apply

$$\begin{cases} k_c' = \{\{k_c + \sigma_x\} \mod L + \\ \qquad L\lfloor (k_c - 1)/L \rfloor + \sigma_y L\} \mod L^2 \\ k_v' = \{\{k_v + \sigma_x\} \mod L + \\ \qquad L\lfloor (k_v - 1)/L \rfloor + 2\sigma_y L\} \mod 2L^2 \end{cases} . \tag{16}$$

To explain this procedure, we note that the toric code is composed of L rows of L check nodes, for a total of L^2 checks, and $2L$ rows of L variable nodes, for a total of $2L^2$ variable nodes. The labels in each row are shifted cyclically modulo L by the term $k_c + \sigma_x \mod L$, which returns a value in $[0, L-1]$. Adding $L\lfloor (k_c - 1)/L \rfloor$ returns a value in the same row of k_c but shifted by σ_x. Finally, a shift of $\sigma_y L$ modulo L^2 is applied to get the final label which will be in the same column but a different row. A similar reasoning is applied for k_v. The rest of the algorithm remains the same.

VI. Complexity Analysis

In this section, we analyze the complexity of the proposed PPBF decoder. The computation of the proximity vector for c_1 is done offline and only once, thus it does not contribute to the computational complexity of the algorithm.

A. Complexity of Algorithms 1, 2 and 3

Algorithms 1, 2 and 3 are extensively used in every decoding iteration, therefore it is crucial that their computational complexity is low. Algorithm 1 simply applies (14) and (15) to shift the proximity influences; this can be done with complexity $\mathcal{O}(1)$. Algorithm 3 applies Algorithm 1 to update the proximity vector; since we update the proximity vector after every flip, Algorithm 1 is applied twice (every flip satisfies two check nodes) for every Algorithm 3 call. Thus, Algorithm 3 has the same order of complexity as Algorithm 1. Finally, since Algorithm 2 is applied just once offline to compute $\boldsymbol{\gamma}(c_1)$ and $\boldsymbol{\nu}(c_1)$, it does not contribute to the decoding complexity; in any case, similarly for Algorithm 3, it only involves calling Algorithm 1 $|\mathbf{s}|$ times, thus it has the same complexity. We then conclude that the computational complexity of Algorithms 1, 2 and 3 is $\mathcal{O}(1)$. We also want to stress that, since the entries

Algorithm 6 `Iterative_matching`

Input: $\hat{\mathbf{e}}$, $\mathbf{s}$, $\mathbf{H}$, $\boldsymbol{\gamma}$
Global: ϕ_c, ϕ_v
Output: $\hat{\mathbf{e}}$

1: $\hat{\mathbf{s}} \leftarrow \mathbf{s}$
2: $\boldsymbol{\gamma}' \leftarrow \boldsymbol{\gamma}$
3: **while** $|\hat{\mathbf{s}}| > 0$ **do**
4: $\quad i \leftarrow \underset{i\in[1,m]}{\arg\min}\ \boldsymbol{\gamma}' \mid \hat{s}_i = 1$ ▷ Pivot.
5: $\quad \mathbf{t} \leftarrow \hat{\mathbf{s}}$
6: $\quad t_i \leftarrow 0$
7: $\quad \boldsymbol{\alpha}(c_i) \leftarrow$ `Shift_influence`(i)
8: $\quad \mathbf{j} \leftarrow j \mid t_j > 0 \wedge \mathbf{c}_j(c_i) > 0$
9: $\quad j \leftarrow \underset{j\in\mathbf{j}}{\arg\min}\ \mathbf{c}(c_i) \wedge \underset{j\in\mathbf{j}}{\arg\min}\ \boldsymbol{\gamma}'$ ▷ Target.
10: $\quad \boldsymbol{\alpha}(c_j) \leftarrow$ `Shift_influence`(j)
11: $\quad \delta \leftarrow c_j(c_i)$
12: $\quad \mathbf{f} \leftarrow k \mid v_k(c_i) + v_k(c_j) = \delta + 1$
13: $\quad$ **if** $|\mathbf{f}| = \delta$ **then**
14: $\quad\quad \hat{e}_{\mathbf{f}} \leftarrow \hat{e}_{\mathbf{f}} \oplus 1$
15: $\quad$ **else**
16: $\quad\quad$ Get σ_x using (14)
17: $\quad\quad$ Get σ_y using (14)
18: $\quad\quad z \leftarrow \phi_c^{-1}(\phi_c(z) + \sigma_x \rho_c)$
19: $\quad\quad \boldsymbol{\alpha}(c_z) \leftarrow$ `Shift_influence`(z)
20: $\quad\quad \mathbf{f}_1 \leftarrow k \mid v_k(c_i) + v_k(c_z) = \Delta x + 1$
21: $\quad\quad \mathbf{f}_2 \leftarrow k \mid v_k(c_j) + v_k(c_z) = \Delta y + 1$
22: $\quad\quad \hat{e}_{\mathbf{f}_1,\mathbf{f}_2} \leftarrow \hat{e}_{\mathbf{f}_1,\mathbf{f}_2} \oplus 1$
23: $\quad$ **end if**
24: $\quad \mathbf{s}' \leftarrow \hat{\mathbf{e}} \cdot \mathbf{H}^T \mod 2$
25: $\quad \boldsymbol{\gamma}' \leftarrow$ `Shift_and_remove`$(\cdot, \boldsymbol{\gamma}, \mathbf{s}')$
26: $\quad \hat{\mathbf{s}} \leftarrow \mathbf{s} \oplus \mathbf{s}'$
27: **end while**
28: **return** $\hat{\mathbf{e}}$

of the proximity vectors are integers with bounded values, and since Algorithms 1, 2 and 3 only involve elementary operations such as integer summation, the memory usage and the runtime of the algorithm will generally be low compared to those of other algorithms with similar computational complexity, but that use floating point calculations.

B. BF decoding complexity

Assuming that comparing elements of $\mathbf{u}$ and $\boldsymbol{\nu}$ has a negligible impact on the complexity, the only meaningful contribution comes from Algorithm 3 and the computation of $\mathbf{u}$ (lines 10 and 12 of Algorithm 5), which both have complexity $\mathcal{O}(n)$.

C. Iterative matching complexity

In each iteration of Algorithm 6, all of the operations can be performed in $\mathcal{O}(1)$, except for the `argmin` function which is performed in $\mathcal{O}(n)$; since the procedure is repeated for each pair of unsatisfied checks, namely $|\mathbf{s}|/2$ times, making the complexity of the algorithm proportional to $\mathcal{O}(n|\mathbf{s}|)$, and since $|\mathbf{s}| = \mathcal{O}(m)$ [22], m being the number of check nodes, and since $m = \mathcal{O}(n)$ (for instance, for the toric code we have $m = n/2$) we have that the complexity of our decoder is $\mathcal{O}(n^2)$. The MWPM complexity is in the order of $\mathcal{O}(n^{12}\log n^2)$, but efficient implementations are available that achieve very small runtime for relatively small values of n [22]. The Union Find decoder [3] achieves an almost-linear complexity of $\mathcal{O}(\alpha(n)n)$, where $\alpha(n)$ is the inverse of Ackermann's function [26], which has been shown to be linear for practical values of n.

VII. Hardware implementation aspects

The major advantage of our algorithm in comparison to UF, MWPM and ST is that it does not require the storage of data structures that require dynamic memory allocation. In this paragraph, we first analyze the UF, MWPM and ST decoders in terms of their usage of dynamic memory, then we show how PPBF, in contrast, only requires static access to pre-allocated memory, thus providing advantages in terms of latency, hardware usage, and energy consumption.

The UF algorithm uses tree structures to represent clusters. Each tree is expanded at each iteration, and eventually trees are merged together. The functions that operate on trees in the UF algorithm are called `union`() and `find`(). The function `find`() can be decomposed in three fundamental operations: i) *traversal*, where the function navigates through the nodes of the tree, ii) *comparison*, where the function checks if the node is the root or not, and iii) *path compression*, where the structure of the tree is modified such that each leaf points directly to the root. The function `union`() can be decomposed into the operations of i) *find*, where the function applies `find`() to a pair of nodes to find their respective root, ii) *comparison*, where the function compares the size of two trees (the smallest is embedded in the largest), iii) *linking*, where the function updates the pointers of the nodes to merge the trees, and iv) *size update*, where the size of the new tree is updated. We also highlight that the size and number of trees that need to be stored is not known *a priori*, and the access to each tree is frequent and unpredictable. On the other hand, MWPM and ST both make use of Dijkstra's algorithm, which also makes use of trees as data structures. At each iteration, the node at the shortest distance is extracted from the tree, and the tree is updated after the node is extracted. This operation has a similar nature to the *traversal* and *path compression* steps of UF. Since at each iteration one single node is extracted, and the tree consequently updated, Dijkstra's algorithm is intrinsically sequential and thus hard to make parallel. Moreover, the adjacency matrix of the syndrome graph created with Dijkstra's algorithm has unknown dimension, and thus requires dynamic memory allocation for storage. In comparison, PPBF only requires the storage of $\boldsymbol{\gamma}(c_1)$ and $\boldsymbol{\nu}(c_1)$, which are two integer vectors of length n, and the value L which is an integer. Moreover, the knowledge of the map ϕ is required, which it is represented by two length-n vectors. The algorithm does not make use of dynamic data structures, but only uses arrays and integers of fixed dimension which can be pre-allocated in memory, increasing the hardware efficiency. The algorithm has a fixed and predictable access to the arrays, indeed all the functions which act on arrays are element-wise operations, which can be made parallel (except for the $\arg\min$ function). Finally, the proximity vectors can be

This article has been accepted for publication in IEEE Transactions on Communications. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCOMM.2024.3454029

Decoder	Memory alloc.	Data type	Access to data
UF	Dynamic	Pointer-based	Irregular
MWPM/ST	Dynamic	Pointer-based	Irregular
PPBF	Static	Array	Element-wise

TABLE I: Summary of the memory usage of the analyzed decoders.

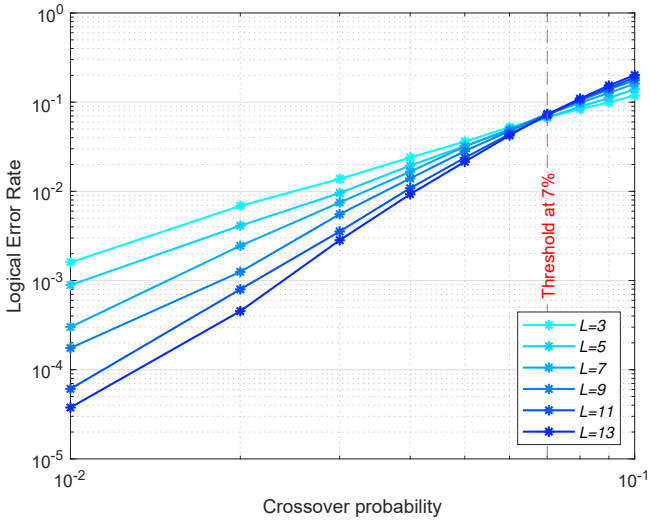


Fig. 5: Performance of our decoder on planar rotated codes of different sizes, assuming a BSC. The threshold, which is around 7%, is highlighted.

normalized and stored with finite precision. For these reasons, our algorithm possesses unique advantages for hardware implementation compared to the competing approaches. In Table I we summarize our analysis. UF and MWPM/ST decoder make use of pointer-based data structures, such as trees, which are accessed in an irregular manner and also modified by the algorithm, thus requiring dynamic memory allocation. In comparison, PPBF only makes use of static data structures as arrays, which can be pre-allocated, and that are modified element-wise only.

VIII. Results

In this section we present simulation results for toric and rotated surface codes. We perform our simulation assuming a bit-flip channel and assume perfect syndrome measurements, and we compare our decoder with MWPM and UF. We have investigated, through simulations, the best value of D for each rotated and toric code. We found that, as long as $D \geq L$, its value does not impact significantly on the performance of the decoder. The reason for the condition $D \geq L$ is to ensure that all the check and variable nodes have a non-zero proximity vector regardless of the check node of reference; if this is not the case, there might be situations in line 9 of Algorithm 6 where no target node is detected. For each data point in the plotted curves, the simulation was run until either 100 logical errors were obtained or 10^5 error vectors were processed. In Fig. 5 we plot simulation results of our decoder on rotated planar codes over the BSC. The threshold can be seen to occur around 7%. In Fig. 6 we plot simulation results of our decoder on toric codes over the bit-flip channel. For comparison, the

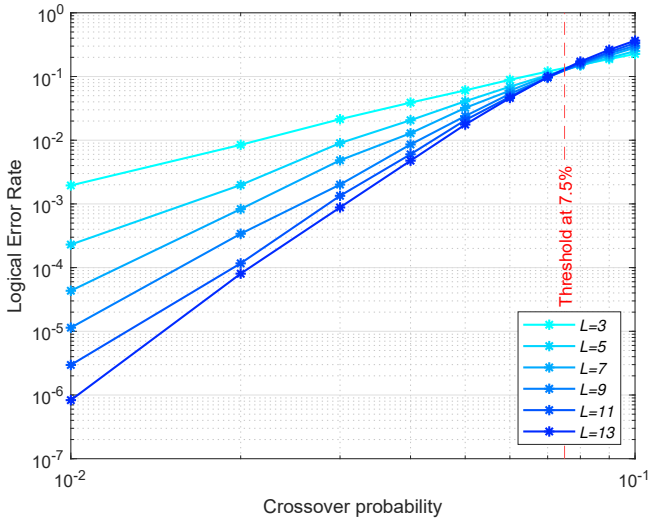


Fig. 6: Performance of the proposed decoder on toric codes of different sizes, assuming a BSC. The threshold, which is around 7.5%, is highlighted.

threshold of MWPM on toric code is 10.3% [1], while that of UF is 9.9% [3]. As can be seen from the figure, the threshold is around 7.5%; to obtain the threshold for the depolarizing channel, it is sufficient to multiply the threshold value by 3/2 [17]. In Figs. 7 and 8 we highlight the comparison between PPBF, traditional BF and MWPM for the rotated and toric codes with $L = 13$ (for the latter, we also add the performance of the UF decoder for comparison). For the traditional BF, in each iteration we flip all of the bits involved in two unsatisfied checks, and we run it for a maximum of 100 iterations. As expected, MWPM, having higher complexity, achieves the best performance, both in terms of threshold and waterfall; the UF presents slightly worse performance than MWPM, although it has significantly lower complexity. Our decoder exhibits a different slope compared to the other decoders, meaning that it does not take fully advantage of the distance of the code. Nevertheless, it still achieves good performance, comparable to that of MWPM and UF; we also need to stress that our decoder is a hard-decision decoder, while MWPM and UF both utilize soft information. Comparing to the classical BF decoder, the PPBF shows a significantly lower error rate.

IX. Conclusion

We have presented a new decoder for surface and toric codes which is able to achieve a very good decoding performance with reasonably low decoding complexity, which is particularly suitable for efficient hardware implementation compared to state-of-art decoders. First, we have defined a novel heuristic called the proximity influence, which assigns weights to variable and check nodes in the neighborhood of an unsatisfied check; we also showed that the proximity influence of a check node c_j can be efficiently obtained by appropriately permuting the influence of an arbitrary node c_1 which is computed offline and stored in memory. The total contribution of the influences of all of the unsatisfied checks is called the proximity vector, and we exploit it as a metric for flipping

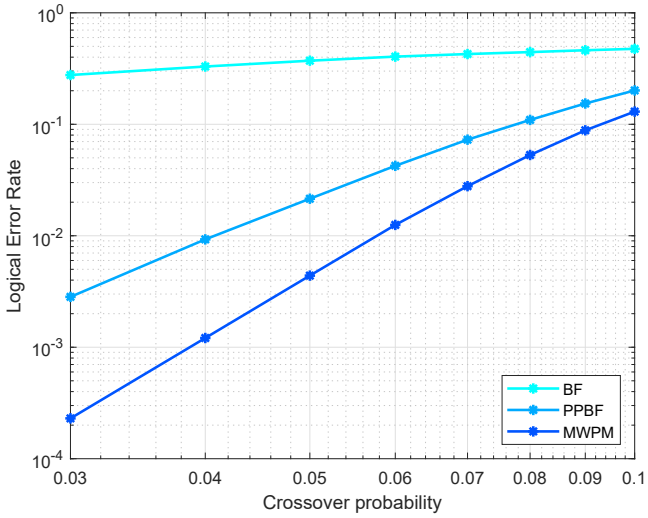


Fig. 7: Comparison of BF, PPBF and MWPM for the distance 13 rotated code.

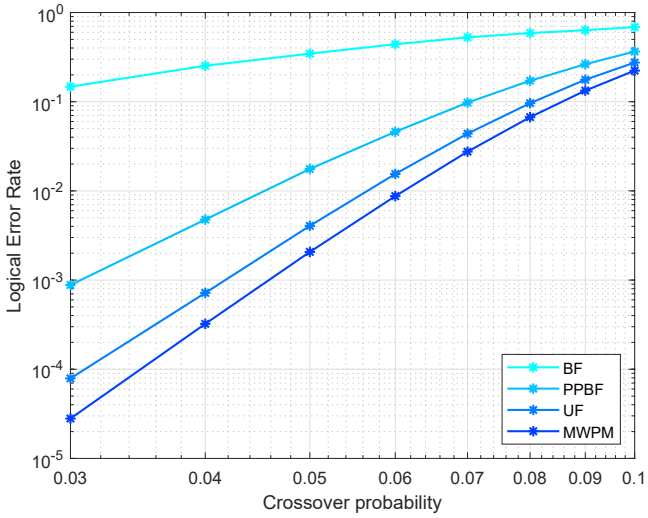


Fig. 8: Performance comparison between our decoder, traditional BF, MWPM and UF for the distance 13 toric code over the BSC.

bits. To deal with error degeneracy and low variable node degrees of toric and surface codes, we designed the iterative matching procedure, which is employed after a round of serial BF. Future work could include improving the performance of the decoder using decoding diversity, *i.e.*, running several rounds of PPBF, each one using a different variation of the proximity metric, and choosing as error estimate the one with least weight. Also, methods to incorporate soft information on the qubits or the syndrome into the proposed decoding algorithm could lead to performance improvement. Adapting the decoder to work on more general QLDPC codes could also could also be an interesting direction for future research.

REFERENCES

[1] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, "Topological quantum memory," *Journal of Mathematical Physics*, vol. 43, no. 9, pp. 4452–4505, Aug. 2002.

[2] A. G. Fowler, "Minimum weight perfect matching of fault-tolerant topological quantum error correction in average $O(1)$ parallel time," Oct. 2014. [Online]. Available: http://arxiv.org/abs/1307.1740

[3] N. Delfosse and N. H. Nickerson, "Almost-linear time decoding algorithm for topological codes," *Quantum*, vol. 5, p. 595, Dec. 2021.

[4] S. Bravyi, M. Suchara, and A. Vargo, "Efficient algorithms for maximum likelihood decoding in the surface code," *Physical Review A*, vol. 90, no. 3, p. 032326, Sep. 2014.

[5] G. Duclos-Cianci and D. Poulin, "Fast Decoders for Topological Quantum Codes," *Physical Review Letters*, vol. 104, no. 5, p. 050504, Feb. 2010.

[6] S. Gicev, L. C. L. Hollenberg, and M. Usman, "A scalable and fast artificial neural network syndrome decoder for surface codes," *Quantum*, vol. 7, p. 1058, Jul. 2023. [Online]. Available: http://dx.doi.org/10.22331/q-2023-07-12-1058

[7] L. Berent, L. Burgholzer, P.-J. H. S. Derks, J. Eisert, and R. Wille, "Decoding quantum color codes with MaxSAT," 2023. [Online]. Available: https://arxiv.org/abs/2303.14237

[8] M. Herold, E. T. Campbell, J. Eisert, and M. J. Kastoryano, "Cellular-automaton decoders for topological quantum memories," *npj Quantum Information*, vol. 1, no. 1, pp. 1–8, Oct. 2015.

[9] O. Higgott, T. C. Bohdanowicz, A. Kubica, S. T. Flammia, and E. T. Campbell, "Improved Decoding of Circuit Noise and Fragile Boundaries of Tailored Surface Codes," *Physical Review X*, vol. 13, no. 3, p. 031007, Jul. 2023.

[10] P. Panteleev and G. Kalachev, "Degenerate Quantum LDPC Codes With Good Finite Length Performance," *Quantum*, vol. 5, p. 585, Nov. 2021.

[11] K.-Y. Kuo and C.-Y. Lai, "Exploiting degeneracy in belief propagation decoding of quantum codes," *npj Quantum Information*, vol. 8, no. 1, pp. 1–9, Sep. 2022.

[12] P. Ivaniš, S. Brkić, and B. Vasić, "Suspicion Distillation Gradient Descent Bit-Flipping Algorithm," *Entropy*, vol. 24, no. 4, p. 558, Apr. 2022.

[13] 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)*, May 2020, pp. 556–569.

[14] S. Krinner, S. Storz, P. Kurpiers, P. Magnard, J. Heinsoo, R. Keller, J. Lütolf, C. Eichler, and A. Wallraff, "Engineering cryogenic setups for 100-qubit scale superconducting circuit systems," *EPJ Quantum Technology*, vol. 6, no. 1, pp. 1–29, Dec. 2019.

[15] A. A. Kovalev and L. P. Pryadko, "Fault tolerance of quantum low-density parity check codes with sublinear distance scaling," *Physical Review A*, vol. 87, no. 2, p. 020304, Feb. 2013.

[16] A. R. Calderbank and P. W. Shor, "Good quantum error-correcting codes exist," *Physical Review A*, vol. 54, no. 2, pp. 1098–1105, Aug. 1996.

[17] D. MacKay, G. Mitchison, and P. McFadden, "Sparse-graph codes for quantum error correction," *IEEE Transactions on Information Theory*, vol. 50, no. 10, pp. 2315–2330, Oct. 2004.

[18] R. Tanner, "A recursive approach to low complexity codes," *IEEE Transactions on Information Theory*, vol. 27, no. 5, pp. 533–547, Sep. 1981.

[19] D. Horsman, A. G. Fowler, S. Devitt, and R. V. Meter, "Surface code quantum computing by lattice surgery," *New Journal of Physics*, vol. 14, no. 12, p. 123011, Dec. 2012.

[20] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," in *Edsger Wybe Dijkstra: His Life,Work, and Legacy*, 1st ed. New York, NY, USA: Association for Computing Machinery, Jul. 2022, vol. 45, pp. 287–290.

[21] J. Edmonds, "Paths, Trees, and Flowers," *Canadian Journal of Mathematics*, vol. 17, pp. 449–467, Jan. 1965.

[22] O. Higgott, "PyMatching: A Python Package for Decoding Quantum Codes with Minimum-Weight Perfect Matching," *ACM Transactions on Quantum Computing*, vol. 3, no. 3, pp. 16:1–16:16, Jun. 2022.

[23] O. Higgott and C. Gidney, "Sparse Blossom: correcting a million errors per core second with minimum-weight matching," Mar. 2023. [Online]. Available: http://arxiv.org/abs/2303.15933

[24] D. Forlivesi, L. Valentini, and M. Chiani, "Spanning tree matching decoder for quantum surface codes," *IEEE Communications Letters*, vol. 28, no. 7, pp. 1509–1513, 2024.

[25] N. Raveendran and B. Vasić, "Trapping Sets of Quantum LDPC Codes," *Quantum*, vol. 5, p. 562, Oct. 2021.

[26] R. E. Tarjan, "Efficiency of a Good But Not Linear Set Union Algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, Apr. 1975.