Progressive-Proximity Bit-Flipping for the 2D Toric Code

Michele Pacenti*, Mark F. Flanagan†, Dimitris Chytas* and Bane Vasić*
*Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ, USA
†School of Electrical and Electronic Engineering, University College Dublin,Belfield, Dublin 4, Ireland
{mpacenti,dchytas}@arizona.edu, mark.flanagan@ieee.org, vasic@ece.arizona.edu

***Abstract*—We propose a novel bit-flipping (BF) decoder tailored for toric codes. We introduce the proximity vector as a heuristic metric for flipping bits, and we develop a new subroutine for correcting a particular class of harmful degenerate errors. Comparing to other decoders, our algorithm is particularly suitable for efficient hardware implementation as it does not require operations on dynamic memories. The proposed decoder shows a decoding threshold of 7.5% for the 2D toric code over the binary symmetric channel.**

***Index Terms*—Toric 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], due to its good performance and low complexity.

Bit flipping (BF) decoders are a class of iterative decoding algorithms known to be very fast and efficient [4], although generally they provide lower performance than Belief Propagation (BP). In general, BF decoders do not perform well on surface codes, mainly because of the 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 [5]. 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 [6]. 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 that are known to be less efficient as they introduce latency and increase circuit complexity as well as power consumption. MWPM, for instance, makes use of Dijkstra's algorithm to construct syndrome graphs on which they perform the perfect matching, which utilizes *priority queues* (dynamic data structures) and is inherently sequential. Similarly, UF relies on dynamic trees. 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 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. We also highlight how our decoder is more suitable for hardware implementation than the alternatives. An extended version of this work can be found at [7].

The rest of the paper is organized as follows. In Section II we introduce the preliminaries of quantum error correction. In Section III we define the proximity vector and how it can be computed efficiently. Section IV provides a detailed description of our proposed decoder. In Section V we carry out a complexity analysis and a comparison with other state-of-art decoders. Finally, Section VI 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, X, Y, Z are called *Pauli operators*. 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. 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 $\mathcal{S} \setminus C(\mathcal{S})$, where $C(\mathcal{S})$ is the centralizer of $\mathcal{S}$. Using the Pauli-to-binary isomorphism [8] we can map the stabilizer matrix $\mathbf{S}$ to a $(n-k) \times 2n$ binary matrix:

$$\mathbf{H} = [\mathbf{H}_X \mid \mathbf{H}_Z] \tag{1}$$

which we call the *parity check matrix* of $\mathcal{C}$.

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 $C_Z \subset C_X$ [9]. 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 stabilizer matrix of the CSS code $\mathcal{C}$ has the form

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_Z & \mathbf{0} \\ \mathbf{0} & \mathbf{H}_X \end{bmatrix}, \tag{2}$$

where $\mathbf{H}_X$ and $\mathbf{H}_Z$ are the parity check matrices of C_X and C_Z, respectively. To correct depolarizing errors on the qubits, a syndrome $\mathbf{s}$ is computed such that

$$\mathbf{s} = [\mathbf{s}_X \ \mathbf{s}_Z], \tag{3}$$

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.

Toric codes [1] are a widely known class of CSS codes. Generally, a toric code is characterized by a $L \times L$ lattice, where L is the size of the horizontal (or vertical) dimension. To an $L \times L$ lattice corresponds a $[\![2L^2, 2, L]\!]$ quantum CSS code. The $\mathbf{H}_X$ matrix is the incidence matrix between vertices (check nodes) and edges (variable nodes), and the $\mathbf{H}_Z$ matrix is the incidence matrix between squares (check nodes) and edges (variable nodes).

A depolarizing channel characterized by channel parameter ϵ (depolarizing error rate), which induces error pattern $\mathbf{e} \in \{I, X, Z, Y\}^n$ on the n qubits. In this paper we consider a bit-flip channel, where each qubit experiences an X error with probability p, and remains correct with probability $1-p$. 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 [10].

Any Pauli error can be expressed as a combination of a *true error* and a stabilizer such that $\mathbf{e} = \mathbf{e}_t + \mathbf{S}_i$, with $\mathbf{S}_i \in \mathcal{S}$; since, by definition, elements of the stabilizer group $\mathcal{S}$ commute with each other (thus, their syndrome is zero), the syndrome $\mathbf{s}$ is only dependent on the true error $\mathbf{e}_t$; nonetheless, this also means that for every true error $\mathbf{e}_t$, there exist a set of Pauli errors $\mathcal{E} = \mathbf{e}_t + \mathcal{S}$ that will lead to the same syndrome. This phenomenon is known as *error degeneracy*.

It is convenient to introduce the notion of *Tanner graph* [11]. 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. The two Tanner graphs corresponding to $\mathbf{H}_X$ and $\mathbf{H}_Z$ of the toric code are identical and they obviously correspond to a torus as well.

III. 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*

$$\gamma_i^{(0)}(c_j) = \begin{cases} 1 \text{ for } i = j \\ 0 \text{ otherwise} \end{cases}, \tag{4}$$

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

$$\boldsymbol{\nu}^{(0)} = \boldsymbol{\gamma}^{(0)} \cdot \mathbf{H} . \tag{5}$$

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

$$\begin{cases}\boldsymbol{\gamma}^{(\ell)}(c_j) = \boldsymbol{\nu}^{(\ell-1)}(c_j)\cdot \mathbf{H}^T \\ \boldsymbol{\nu}^{(\ell)}(c_j) = \boldsymbol{\gamma}^{(\ell)}(c_j)\cdot \mathbf{H}\ ,\end{cases} \tag{6}$$

for $\ell = 1, 2, ..., D$. 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:*

$$\begin{cases}\boldsymbol{\nu}^{(D)} = \sum_{i=1}^{m} \boldsymbol{\nu}^{(D)}(c_i) \\ \boldsymbol{\gamma}^{(D)} = \sum_{i=1}^{m} \boldsymbol{\gamma}^{(D)}(c_i)\end{cases}\ . \tag{7}$$

It is also possible to define $\boldsymbol{\nu}^{(D)}$ and $\boldsymbol{\gamma}^{(D)}$ by setting $\boldsymbol{\gamma}^{(0)} = \mathbf{s}$, and then using (5) and (6) directly.

Since the value D will be fixed in the rest of the paper, we simply refer to the influences as $\boldsymbol{\nu}(c_j)$ and $\boldsymbol{\gamma}(c_j)$, and to the vectors as $\boldsymbol{\nu}$ and $\boldsymbol{\gamma}$.

1) Efficient computation of the proximity vector: 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

$$\begin{cases}\boldsymbol{\gamma}' = \boldsymbol{\gamma} - (\boldsymbol{\gamma}(c_j) + \boldsymbol{\gamma}(c_k)) \\ \boldsymbol{\nu}' = \boldsymbol{\nu} - (\boldsymbol{\nu}(c_j) + \boldsymbol{\nu}(c_k)).\end{cases} \tag{8}$$

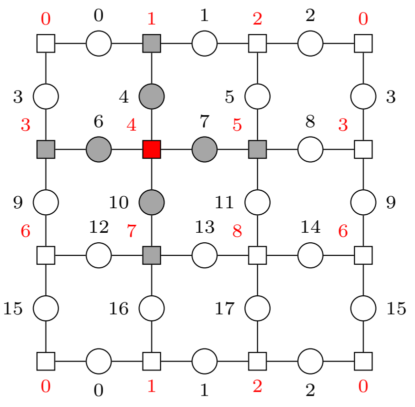
We exploit the quasi-cyclic property of the toric code to update the proximity vector in an efficient manner. The idea is to pre-compute the proximity influence of an arbitrary check node, say $\boldsymbol{\gamma}(c_1), \boldsymbol{\nu}(c_1)$, store it prior to the decoding process, and then compute online $\boldsymbol{\gamma}(c_i), \boldsymbol{\nu}(c_i)$ when needed, by appropriately permuting $\boldsymbol{\gamma}(c_1), \boldsymbol{\nu}(c_1)$. We can label the variable nodes with non-negative integers row-wise in increasing order (so that the first row is labeled from 0 to $L-1$, the second row is labeled from L to $2L-1$, and so on), and we can do the same for the check nodes. Assume that we have already stored $\boldsymbol{\nu}(c_i)$ and $\boldsymbol{\gamma}(c_i)$ for some arbitrary check node c_i, and that we now wish to compute $\boldsymbol{\nu}(c_j)$ and $\boldsymbol{\gamma}(c_j)$, for some $j \neq i$, by appropriately permuting $\boldsymbol{\nu}(c_i)$ and $\boldsymbol{\gamma}(c_i)$. 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 in the following manner:

$$\begin{cases}\sigma_y = \lfloor (j-i)/L \rfloor \\ \sigma_x = j - i \mod L\end{cases}\ , \tag{9}$$

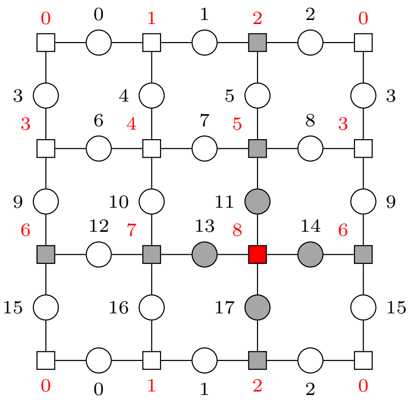
assuming that $i < j$. We can express the index transformation in a closed form, using σ_x and σ_y. Let $k_c \in [1, L^2]$ and $k_v \in [1, 2L^2]$ be the indices of the check and variable nodes, respectively. We can express a coordinate shift using two linear maps from k_c to k'_c and from k_v to k'_v respectively, such that

$$\begin{cases}k'_c = \{\{k_c + \sigma_x\} \mod L + \\ \qquad L\lfloor (k_c - 1)/L \rfloor + \sigma_y L\} \mod L^2 \\ k'_v = \{\{k_v + \sigma_x\} \mod L + \\ \qquad L\lfloor (k_v - 1)/L \rfloor + 2\sigma_y L\} \mod 2L^2\end{cases} \tag{10}$$

Hence, we have $\gamma^{\ell}_{k'_c}(c_j) = \gamma^{\ell}_{k_c}(c_i)$ and $\nu^{\ell}_{k'_c}(c_j) = \nu^{\ell}_{k_c}(c_i)$. An example of this coordinates mapping is illustrated in Fig. 1.



(a) The depth-1 influence of c_4 (highlighted in red) is computed; check and variable nodes such that $\nu(c_4) \neq 0$ and $\gamma(c_4) \neq 0$ are represented in gray, while nodes such that $\nu(c_4) = 0$ and $\gamma(c_4) = 0$ are represented in white.



(b) The influence of Fig. 1(a) is shifted to the node c_8: in this case, $\sigma_x = \sigma_y = 1$.

Fig. 1: Example of shifting of the proximity influence from c_5 to c_8.

2) Auxiliary proximity influence: We also define an auxiliary proximity influence of a check node c_j, that we denote as $\mathbf{v}(c_j)$.

Definition 3. *The auxiliary proximity influence $\mathbf{v}(c_j)$ is a $1 \times n$ vector such that $v_i(c_j)$ 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 $v_i(c_j) = 1$.*

Note that the proximity influences $\boldsymbol{\nu}(c_j)$ and $\mathbf{v}(c_j)$ 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 $\mathbf{v}(c_j)$ 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 $\mathbf{v}(c_j)$ in a similar way to $\boldsymbol{\nu}(c_j)$. Thus, it is sufficient to construct $\mathbf{v}(c_1)$ offline, and apply (10) to create $\mathbf{v}(c_j)$, for any j.

IV. 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 1, and it is composed of two decoding steps: the first is called *Preliminary_BF*, and the second is called *Iterative matching*.

Algorithm 1 Progressive-Proximity Bit-Flipping

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

Algorithm 2 Preliminary_BF

```
Input: $\mathrm{s}$, $\mathbf{H}$
Output: $\hat{\mathrm{e}}$
1: $\hat{\mathrm{e}} \leftarrow \mathbf{0}$
2: $\hat{\mathrm{s}} \leftarrow \mathrm{s}$
3: $\mathbf{u} \leftarrow \mathrm{s} \cdot \mathbf{H}$
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:     $j \leftarrow \arg\min_{i \in \mathcal{S}} \nu'_i$
8:     $\hat{e}_j \leftarrow \hat{e}_j \oplus 1$
9:     $\mathrm{s}' \leftarrow \hat{\mathbf{e}} \cdot \mathbf{H}^T \mod 2$
10:    $\boldsymbol{\nu}' \leftarrow$ Shift_and_remove($\boldsymbol{\nu}', \cdot, \mathrm{s}'$)
11:    $\hat{\mathrm{s}} \leftarrow \mathrm{s} \oplus \mathrm{s}'$
12:    $\mathbf{u} \leftarrow \hat{\mathrm{s}} \cdot \mathbf{H}$
13: end while
14: return $\hat{\mathrm{e}}$, $\hat{\mathrm{s}}$
```

The `Compute_proximity_metric` subroutine takes the syndrome s as input and combines (10) and (7) to get the proximity vectors $\boldsymbol{\gamma}$ and $\boldsymbol{\nu}$. The `Preliminary_BF` algorithm, illustrated in Algorithm 2, is a serial BF decoder which, at each iteration, flips the bit with minimum ν_i that is connected to two unsatisfied checks, and consequently obtains the updated $\boldsymbol{\nu}' = \boldsymbol{\nu} - \boldsymbol{\nu}(c_j) - \boldsymbol{\nu}(c_k)$, where c_j and c_j are the two checks connected to the flipped bit. More details on these subroutines can be found in [7]. The `Iterative_matching` routine, illustrated in Algorithm 3, utilizes 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 $\mathbf{v}(c_i)$[1]. 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 entryγ_j); we call this the *target node* (line 9), and denote its distance from the pivot by δ. After computing the auxiliary vectors for c_j, namely $\mathbf{c}(c_j), \mathbf{v}(c_j)$, we compute $\mathbf{v}(c_i) + \mathbf{v}(c_j)$; 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 $\mathbf{v}(c_i) + \mathbf{v}(c_j)$ 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 $\mathbf{v}(c_i) + \mathbf{v}(c_j)$ 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. 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; specifically, we introduce an extra unsatisfied check c_z (line 22), which is positioned Δx steps to the right (or left) of c_i, and Δy steps below (or above) of c_j. There will be only one path of shortest length connecting c_i with c_z and c_z with c_j, respectively, and we flip the bits corresponding to this two paths. A more comprehensive explanation of the algorithm can be found in [7].

V. Complexity analysis and hardware implementation

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. In [7] we show that the computational complexity of calculating $\boldsymbol{\gamma}$ and $\boldsymbol{\nu}$ can be considered $\mathcal{O}(1)$, and that the `Preliminary_BF` step has a complexity of $\mathcal{O}(n)$. In each iteration of Algorithm 3, all of the operations can be performed in $\mathcal{O}(1)$, except for the `arg min` function which has complexity of $\mathcal{O}(n)$; since the procedure is repeated for each pair of unsatisfied checks, namely $|\mathrm{s}|/2$ times, making the complexity of the algorithm proportional to $\mathcal{O}(n|\mathrm{s}|)$, and since $|\mathrm{s}| = \mathcal{O}(m)$ [12], 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 $\mathcal{O}(n^2)$. Regarding the hardware implementation aspects, 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. 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

[1]In Algorithm 3, we refer with `Shift_influence` and `Shift_and_remove` to two subroutines that utilize (7), (8), and (10) to compute and update the proximity vectors. More details can be found in [7].

Algorithm 3 `Iterative_matching`

```
Input: $\hat{e}$, s, H, $\gamma$
Output: $\hat{e}$
1: $\hat{s} \leftarrow s$
2: $\gamma' \leftarrow \gamma$
3: while $|\hat{s}| > 0$ do
4:     $i \leftarrow \arg\min_{i\in[1,m]} \gamma' \mid \hat{s}_i = 1$        ▷ Pivot.
5:     $t \leftarrow \hat{s}$
6:     $t_i \leftarrow 0$
7:     $c(c_i), v(c_i) \leftarrow$ Shift_influence($i$)
8:     $j \leftarrow j \mid t_j > 0 \wedge c_j(c_i) > 0$
9:     $j \leftarrow \arg\min_{j\in j} c(c_i) \wedge \arg\min_{j\in j} \gamma'$        ▷ Target.
10:    $c(c_j), v(c_j) \leftarrow$ Shift_influence($j$)
11:    $\delta \leftarrow c_j(c_i)$
12:    $f \leftarrow k \mid v_k(c_i) + v_k(c_j) = \delta + 1$
13:    if $|f| = \delta$ then
14:        $\hat{e}_f \leftarrow \hat{e}_f \oplus 1$
15:    else
16:        if $j - i \mod L < i - j \mod L$ then
17:            $\Delta x \leftarrow j - i \mod L$
18:        else
19:            $\Delta x \leftarrow -(i - j \mod L)$
20:        end if
21:        $\Delta y \leftarrow \min\{\lfloor \frac{i-1}{L} \rfloor - \lfloor \frac{j-1}{L} \rfloor \mod L, \lfloor \frac{j-1}{L} \rfloor - \lfloor \frac{i-1}{L} \rfloor \mod L\}$
22:        $z \leftarrow i + \Delta x \mod L + L\lfloor \frac{i-1}{L} \rfloor$
23:        $c(c_z), v(c_z) \leftarrow$ Shift_influence($z$)
24:        $f_1 \leftarrow k \mid v_k(c_i) + v_k(c_z) = \Delta x + 1$
25:        $f_2 \leftarrow k \mid v_k(c_j) + v_k(c_z) = \Delta y + 1$
26:        $\hat{e}_{f_1,f_2} \leftarrow \hat{e}_{f_1,f_2} \oplus 1$
27:    end if
28:    $s' \leftarrow \hat{e} \cdot H^T \mod 2$
29:    $\gamma' \leftarrow$ Shift_and_remove($\cdot, \gamma, s'$)
30:    $\hat{s} \leftarrow s \oplus s'$
31: end while
32: return $\hat{e}$
```

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 normalized and stored with finite precision. For these reasons, our algorithm possesses unique advantages for hardware implementation compared to the competing approaches.

VI. Results

In this section we present simulation results for toric code. We perform our simulation assuming a bit-flip channel and assume perfect syndrome measurements, and we compare our decoder with MWPM and UF. For our simulations we fix $D = \lfloor L/2 \rfloor$. 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. 2 we plot simulation results of our decoder on toric codes over the bit-flip channel. As can be seen from the figure, the threshold is around 7.5%[2]; to obtain the threshold for the depolarizing channel, it is sufficient to multiply the threshold value by 3/2 [10]. In Fig. 3 we highlight the comparison between our decoder, traditional BF, MWPM and UF for the toric code with $L = 13$. 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 still achieves good performance, comparable to that of MWPM and UF, while achieving low complexity and latency; we also need to stress that our decoder is a hard-decision decoder, while

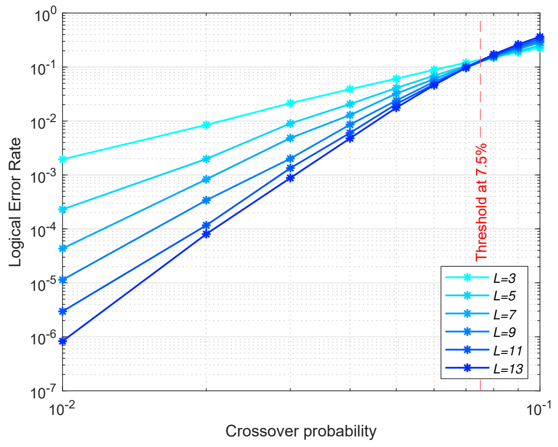


Fig. 2: Performance of the proposed decoder on toric codes of different sizes, assuming a BSC channel. The threshold, which is around 7.5%, is highlighted.

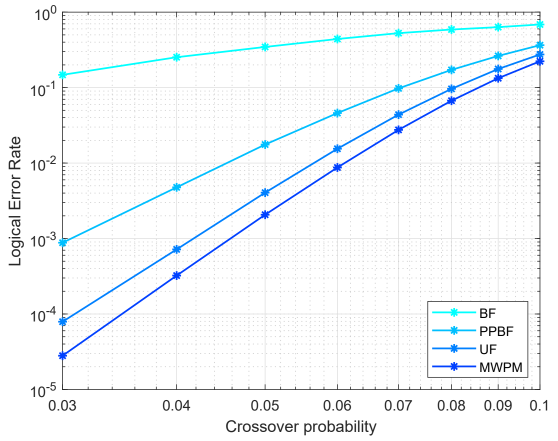


Fig. 3: Performance comparison between our decoder, traditional BF, MWPM and UF for distance 13 toric code over BSC.

[2]For threshold we mean the depolarizing probability below which the decoder error probability approaches zero while increasing the distance of the code.

MWPM and UF both utilize soft information. Comparing to the classical BF decoder, the PPBF shows a significantly lower error rate.

VII. Conclusion

We have presented a new decoder for toric codes which is able to achieve a very good decoding performance while allowing efficient hardware implementation. 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.

Acknowledgments

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.

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

[5] 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.

[6] 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.

[7] M. Pacenti, M. F. Flanagan, D. Chytas, and B. Vasic, "Progressive-proximity bit-flipping for decoding surface codes," 2024. [Online]. Available: https://arxiv.org/abs/2402.15924

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

[9] 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.

[10] 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.

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

[12] 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.