A Comparative Analysis of the Blossom Algorithm for Surface Codes

Zicheng Yang[1], Yuheng Ma[1], Yiming Xiao[1], Zikang Lv[1]

[1]School of physics, ZheJiang University
[1]School of computer science and technology, ZheJiang University
[1]School of computer science and technology, ZheJiang University
[1]School of physics, ZheJiang University
(June 9, 2024)

Abstract

Nowadays more and more decoders for quantum error correction are published, many of which are potential in surface code decoding. We compared the decoding performance of different decoders from some vital criteria. Our simulations reveal the blossom algorithm's performance in various quantum memory scales and physical error rate, demonstrating an error threshold of approximately 0.104. The Blossom Algorithm may not distinguish in time complexity compared to some contemporary methods. However, its higher error threshold presents a strategic advantage for specific quantum computing tasks. At the end we conclude with a trade-off of decoders in three aspects: time consuming, error threshold and logical error rate. We anticipate the emergence of a decoder that excels across these three dimensions, which can be the winner. For more information, please visit our official GitHub repository at `https://github.com/Winfred666/SurfaceCode-MWPMDecoder.git`.

I. Introduction

With the popularity of quantum computing spreading in the investors and researchers, quantum error correcting has also been much under scrutiny. Though in the early time, people have the skepticism that weather quantum computing can keep and manipulate a quantum state error-free for a long time [1], the development of quantum error correction making it more optimistic.

Among so many prospective codes of quantum error correcting, the most promising classes is the surface code, which is proposed by Alexei Kitaev in 1997 [2]. The topological concept of the surface code or toric code is to encode the qubit in a 2D lattice of physics qubits system which has some specific boundary condition [3]. In this case, we can influence and correct a wide range of qubits by applying a given way on the torus. Although challenges of fabricating a large-scale quantum memory still exist, the progresses on experimental advances in recent years give us hope to make it a reality [4][5].

Another significant period of a quantum error correcting task is to decode. It is natural that we want to find an optimal solution to figure out the most probably error and correct it for every given measurement, which is called the Maximums Like Algorithm. But it seems unpractical for its exponential operation time [6], so how to balance between performance and running time of the algorithm becomes the priority. A lot of solution has been designed to solve such problems, such as the graph algorithm and neural network [7].

In this paper, we will take an insight of a kind of graph algorithm, the Blossom Algorithm. As well as we will compare it with other decoder algorithm from multiple dimensions.

II. Background of Quantum Error Correction

A standard error correction usually consists of four step [7]:

1. **Encode** : encode the logical quantum state to another defined state, for example:
$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \rightarrow |\psi_E\rangle = \alpha|000\rangle + \beta|111\rangle$$
2. **Detect** : detect the quantum errors by measurement
3. **correct** : correct those errors and decode the qubits to get the corresponding logical state
4. **compute** : compute logical operations on the state by redefining a universal gate set on the code.

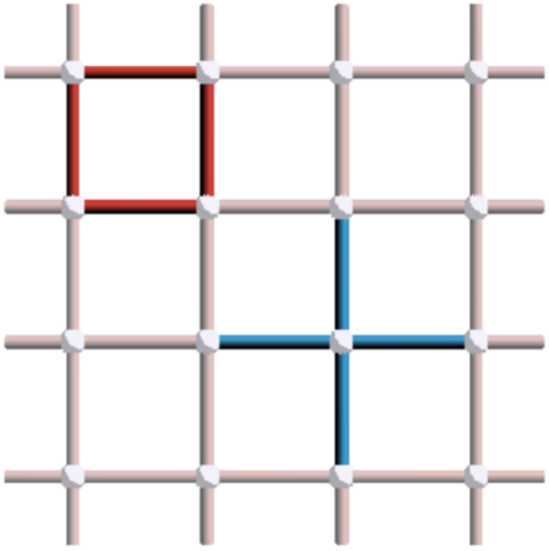


Figure 1: *An illustration of the lattice. The unit marked by the red line or the blue line shows the A_s operator and B_p operator respectively*

Here we will focus on the first three steps and outline our work by the following steps:

1. **Introduce errors**
2. **Plot syndromes**
3. **Match the syndromes**
4. **Get the result**

A. toric code

For a better understanding of the toric code, we assume a L×L lattice, where every single qubit lives on the stage, so we have L^2 qubits. The stabilizer group of toric code is generated by two types of operator A_s and B_p [8]:

$$A_s = \bigotimes X_q, B_p = \bigotimes Z_q$$

The X and Z are the Pauli matrix the Hamiltonian of the system is the linear combination of the tow operator:

$$H = -A_s - B_p$$

We show the stabilizer and the lattice in Figure 1.

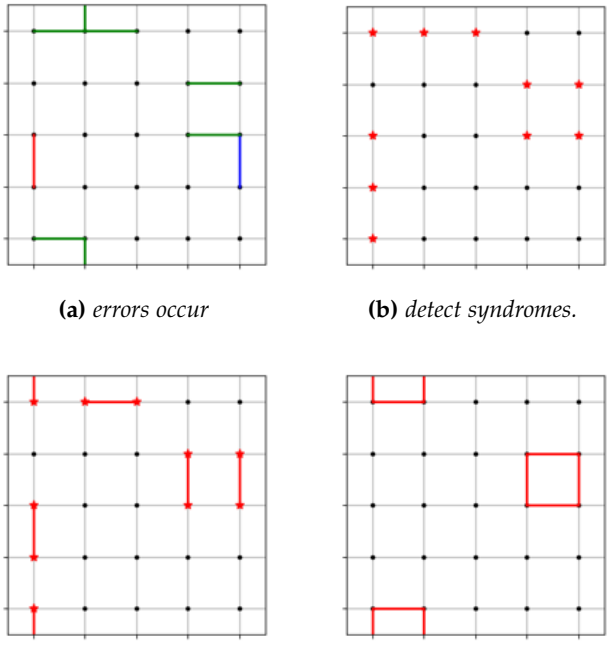
B. Quantum Error

Generally speaking, the noisy and environment disturbance is inevitable in real quantum system. It is impossible for us to clarify the procedure of the coupling between the system and the environment. But in fact, we can ease the challenge by simplifying all the quantum errors into two types of error:

$$\text{Bit-flips error:} |\psi\rangle \rightarrow \hat{X}|\psi\rangle$$

$$\text{Phase-slip error:} |\psi\rangle \rightarrow \hat{Z}|\psi\rangle$$

The X error and the Y error are the fundamental mode of quantum error and they can appear at the same time, which we call Y error. In this paper, we assume that each error has the same probability $p/3$ to occur(in the code we implement ,the p is 0.15).



(a) *errors occur* **(b)** *detect syndromes.*

(c) *matching* **(d)** *decoder*

Figure 2: *the procedure of an error correction. (a)-(d) shows the different stage respectively*

C. Decoder

As shown in Figure 2(a)-(d), when a series of errors happen and we detect this error by the measurement and collect all the syndromes. The task of decoder is to restore the most possible error. The green line ,red line and blue line in Figure 2(a) indicate the X error ,Z error and Y error. It is shown in the figure 2(b) that we detect these error with the measurement we mention above. The next step and the most challenging part is to match the syndromes and decode them to get result. We definitely hope what we get is trivial, but sometimes the logical error or the non-trivial result is inevitable due to the limitation of decoder algorithm [9]. In the next section, we will introduce the Blossom Algorithm and test its logical error rate and compare it with other decoder algorithm.

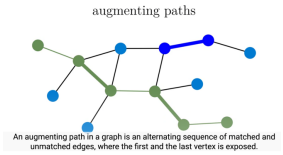
III. Blossom Algorithm

Here, we implement a decoder algorithm called Blossom Algorithm, which is first proposed by Edmonds in 1965. Edmond's idea requires $O(n^2m)$ times, where n is the number of nodes in the graph and m is the number of edges [10]. The Blossom Algorithm has been developed with the time and different improvements have been conducted. It is impossible for us to cover all the creative ideas. Before we discuss the kind of minimum weight matching Blossom Algorithm, we first get an insight of the oldest Blossom Algorithm.

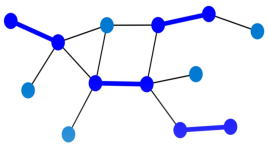
1. **Augmenting Paths**: And then, an augmenting path is required. Like branches on a tree, an augmenting path is an alternating sequence of matched and unmatched edges, where the first and the last vertex are exposed, as shown in Figure 3(a). It can improve the size of the current matching by switching the matched and unmatched edges, as shown in Figure 3(b). And the process of matching can repeat until no free vertices are left, at which point, we know the matching is maximum, as shown in Figure 3(c). We can prove that:

   contains an augmenting path $\leftrightarrow$ matching is not maximum

2. **Blossoms**: But things don't always go well. When there is a blossom, a cycle containing an odd number of "pseudo nodes", where a pseudo node is either a vertex or another blossom, the algorithm fails, as shown in Figure 3(d) and 3(e). We can see that since the augmenting path is longer than the shortest path, it is not found.
3. **Blossom Contraction and Expansion**: In this case, When an odd-length cycle (blossom) is detected, we should contract it into a single vertex, simplifying the problem and after finding a matching, expand the blossoms back to update the matching accordingly. The Figure 4(a)-(d) show the solution when a blossom is detected step by step.
4. **Iterative Process**: Finally, repeat the search for augmenting paths, contraction, and expansion until no more augmenting paths can be found. The whole process and code are presented in Figure 5.

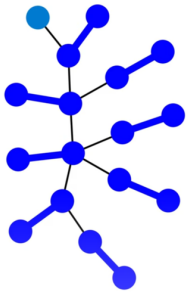


(a) *augmenting path*

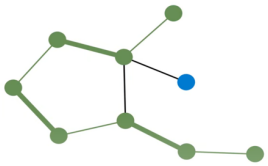


(b) *switch the edges*

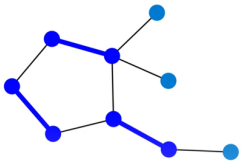
```
def find_augmenting_path(v):
    bfs.start_from_vertex(v)
    while bfs.has_unexplored_vertices():
        bfs.add_vertices_not_in_matching()

        if bfs.found_augmenting_path():
            return bfs.augmenting_path

        bfs.add_vertices_in_matching()

    return []

def improve_matching(G):
    for v in exposed_vertices(G):
        path = find_augmenting_path(v)

        if path != []:
            switch_augmenting_path(path)
            return True

    return False

while True:
    if not improve_matching(G):
        break
```



(c) *the blossom algorithm*

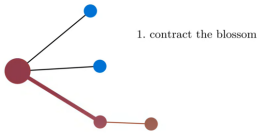


(d) *blossom*

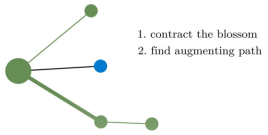


(e) *failed blossom*

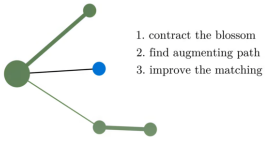
Figure 3: *the basic concepts of blossom algorithm*



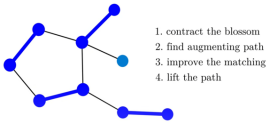
(a) *contract the blossom*



(b) *find augmenting path*



(c) *improve the matching*



(d) *lift the path*

Figure 4: *the contraction and expansion of blossoms. (a)-(d) shows the different step respectively*

```python
def find_augmenting_path(v):
    bfs.start_from_vertex(v)
    while bfs.has_unexplored_vertices():
        bfs.add_vertices_not_in_matching()

        if bfs.found_blossom():
            bfs.contract_blossom()
            find_augmenting_path(v)
            bfs.lift_blossom()

        if bfs.found_augmenting_path():
            return bfs.augmenting_path

        bfs.add_vertices_in_matching()


    return []

def improve_matching(G):
    for v in exposed_vertices(G):
        path = find_augmenting_path(v)

        if path != []:
            switch_augmenting_path(path)
            return True

    return False

while True:
    if not improve_matching(G):
        break
```

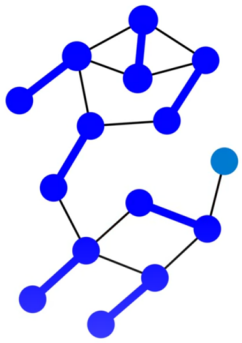


Figure 5: *the whole process and the code*

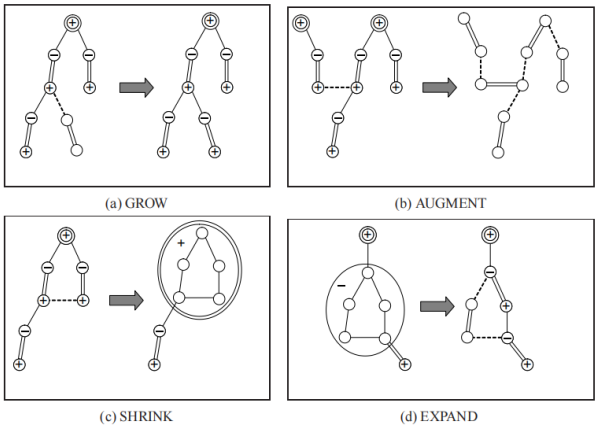


Figure 6: *An illustration of the minimum weight perfect matching*

The minimum weight perfect matching algorithm is originated from the initial idea, but it has some improvement to handle with the practical problems, because there is *weight* in our syndrome graph! Thus, we need a more complicated derived algorithm to solve our problem: Blossom V. The implementation of the algorithm is the following:for every vertex i,it has a weight $l(i)$ and every edge is a coupling of the vertices i, j,it meets the condition:$w(i,j) \leq l(i) + l(j)$.Every perfect minimum weight matching,the sum of the weight edge is:

$$\begin{aligned} \text{val}(M) &= \sum_{(u,v)\in M} w(u,v) \\ &\leq \sum_{(u,v)\in M} (l(u) + l(v)) \\ &\leq \sum_{i=1}^{n} l(i) \end{aligned}$$

Defining z_u as the vertex labeling of u, and we also define $e(u,v)$ an equality edge if $(z_u + z_v = weight(e))$, and at this time the edge labeling of edge is called z_e. It requires $z_e = z_u + z_v - weight(e) = 0$. The augmenting path composed of "equality edges" is continuously expanded, and since all the edges used for expansion are "equality edges", the final maximum weight perfect matching obtained is still all "equality edges". In Figure 6, we demonstrate the whole process of this algorithm in detail. It consists of four steps [10]:

1. **Grow**: If edge (u,v) is tight, $l(u) = +$and $l(v) = \emptyset$ then the tree to which u belongs can be "grown" by acquiring node u and the corresponding matched node, as shown in the Figure 6(a)
2. **Augment**: If edge (u,v) is tight, $l(u) = l(v) = +$ and u, v belong to different trees then the cardinality of matching x can be increased by "flipping" variable x_e for edges e along the path connecting the roots of the two trees, as shown in Figure 6(b). All nodes in the trees become free.
3. **Shrink**: If edge (u,v) is tight, $l(u) = l(v) = +$ and u, v belong to the same tree then there is cycle of odd length that can be shrunk to a blossom, as shown in Figure 6(c). The dual variable for this new blossom is set to 0.
4. **Expand**: If node v is a blossom with $y_e = 0$ and $l(v) = -$ then it can be expanded(Fig.7(d)).

The u^- represents odd vertex in an alternating tree, while the u^+ represents the even vertex in an alternating tree and $u^\emptyset$ represents an vertex not in any alternating tree.

IV. Cellular Automation Decoder

Besides the Blossom Algorithm, we also get an insight of another decoder, the Cellular Automation Decoder. It offers a dynamical approach to decoding, while others pay attention to the graph problem. The CA decoder draws inspiration from statistical mechanics and field theories, which makes it more “physical” and natural. The whole system can be compared to a dissipative dynamical system driven away from equilibrium. Here we give a brief description of ϕ-automation decoder that drives the system by Coulomb-like potential [11]:

1. **Anyons Creation:** When errors occur, ’anyons’ (or ’syndromes’ we have mentioned in the pas-

sage) emerge at the ends of error strings.

2. **Field Update Rule:** For each cell $\xi \in \Lambda(lattice)$, the field value at ξ, denoted by $\phi(\xi)$, is updated in parallel by taking the average of the field values at all neighboring cells ξ' and adding the charge $q(\xi)$ if there is an anyon present at cell ξ. This can be expressed as:

$$\phi(\xi) = \underset{<\xi',\xi>}{avg} \; phi(\xi') + q(\xi)$$

3. **Anyon Update Rule:** For each cell $x \in V(vertex)$, if there is an anyon present $(i.e., q(x) \neq 0)$, then with $p = 0.5$, the anyon moves to the neighboring cell y that has the highest local field value $\phi(y)$.
4. **Iteration Continuation:** If anyons are still present after the updates, increase the sequence counter τ by 1 and repeat the steps above.
5. **Termination Condition:** The process continues iteratively until there are no more anyons to move, indicating that the error syndromes have been resolved.

V. Test and result

There is two critical parameter to evaluate the performance of a decoder:the error threshold and the time complexity.

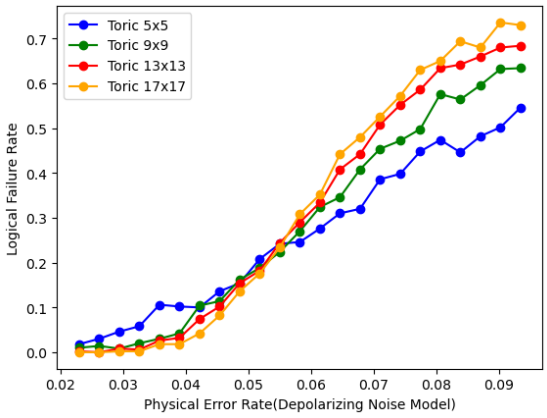
However, one pitfall about test set of surface code is the usage of noisy model. Though depolarizing noise model is the best one to simulate the real condition, most of the test program just generate only bit-flip errors to test the decoder. So in 7b we translate osur physical error rate to the unit of commonly used one.

In this part, we will show our results and compare different decoders with their error threshold and time complexity. You can see more details in our github website, here, we just give a brief introduction. The error threshold is a critical parameter in the design and implementation of fault-tolerant quantum computation. It is related to the threshold theorem, which states that arbitrarily long quantum computations are reliable below a specific error rate that varies with the quantum decoders we use. Above the specific error rate, the reliability of the quantum computation rapidly deteriorates, making it impossible to conduct a fault-tolerant quantum computation. Figure 7(a)-(b) shows the logical error rate varying with the phase-flip and bit-flip error rate under the noise model respectively. For four different scale quantum memories, we set up 25 different groups of error rates, each group of error rates for 500 simulation tests, and calculate the logic error rate. We can calculate the error threshold by recording the intersection of four lines. And it shows the error threshold is about 0.104.

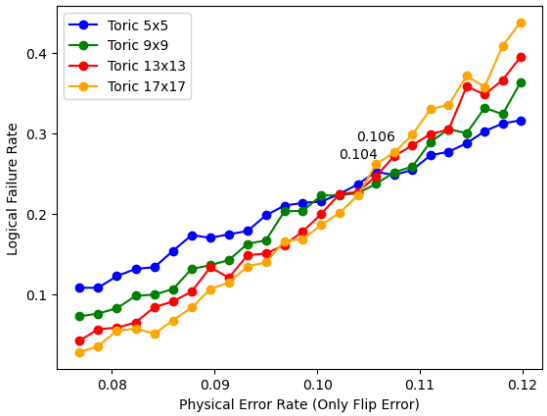
For time complexity, we find our implementation using Blossom V very slow. Though we use local dijksra to build syndrome graph (see in Pymatching refer papers) , which set a parameter M, and defects node will stop connecting with other nodes when its degree is larger than M. However,we just turn the time complexity from $O(n^3 logn)$ [12] to $O(n^2 Mlogn)$, and there are risks of low accuracy.

Luckily, in 2023 , Pymatching2.0, the new library use "sparse blossom algorithm", which has an amortize time of $O(n)$ and high accuracy.

The comparison of time tested by us is shown in 8a and 8b.

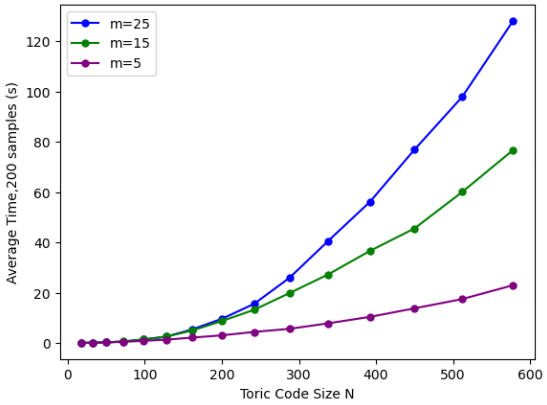


(a)

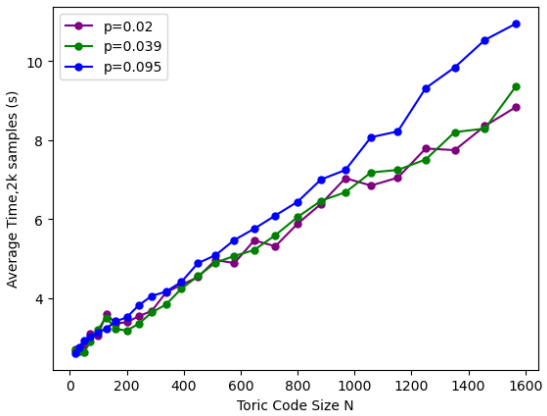


(b)

Figure 7



(a)



(b)

Figure 8: *Time complexity of (a) PyMatching1.2(our implementation) and (b) Pymatching2.0*

Table 1: *Time complexity of different decoders (N is physical qubit number)*

Decoder	Time Complexity
MWPM(original)	$O(n^3 logn)$
MWPM(M-local-dijkstra)	$O(n^2 Mlogn)$
MWPM(sparse blossom)	$O(n)$
Cellular Automaton	$O(n^2)$
Union Find	$O(n)$

VI. Conclusion

A. Our achievement

Our research has successfully implemented the Blossom Algorithm and reproduced the Minimum Weight Perfect Matching (MWPM) algorithm, exploring their roles in quantum error correction. We assessed key performance index, especially the error threshold and time complexity, across a series of modern decoders.

B. Outlook

Our findings highlight the Blossom Algorithm's robust error threshold, a significant advantage for quantum computing applications. Despite its time complexity not being as competitive, this advantage clearly sets it apart. Through comparative analysis, we've identified performance trade-offs among decoders. The ideal decoder—one that excels in error threshold, time consuming, and logical error rate—remains an aspirational goal. We anticipate that future breakthroughs will deliver a decoder with exceptional performance on all fronts, therefore, greatly enhancing the potential of quantum computing. Or we can expect to see the emergence of different decoding methods or joint application of multiple decoders in practical quantum computing under different circumstances.

References

[1] Serge Haroche and Jean-Michel Raimond. "Quantum Computing: Dream or Nightmare?" In: *Physics Today* 49.8 (Aug. 1996), pp. 51–52. ISSN: 0031-9228, 1945-0699. DOI: 10.1063/1.881512. (Visited on 05/30/2024).

[2] A.Yu. Kitaev. "Fault-tolerant quantum computation by anyons". In: *Annals of Physics* 303.1 (Jan. 2003), pp. 2–30. ISSN: 0003-4916. DOI: 10.1016/s0003-4916(02)00018-0. URL: http://dx.doi.org/10.1016/S0003-4916(02)00018-0.

[3] Eric Dennis et al. "Topological quantum memory". In: *Journal of Mathematical Physics* 43.9 (Sept. 2002), pp. 4452–4505. ISSN: 1089-7658. DOI: 10.1063/1.1499754. URL: http://dx.doi.org/10.1063/1.1499754.

[4] Jerry M. Chow et al. "Implementing a strand of a scalable fault-tolerant quantum computing fabric". In: *Nature Communications* 5.1 (June 2014). ISSN: 2041-1723. DOI: 10.1038/ncomms5015. URL: http://dx.doi.org/10.1038/ncomms5015.

[5] R. Barends et al. "Superconducting quantum circuits at the surface code threshold for fault tolerance". In: *Nature* 508.7497 (Apr. 2014), pp. 500–503. ISSN: 1476-4687. DOI: 10.1038/

nature13171. URL: http://dx.doi.org/10.1038/nature13171.

[6] Kao-Yueh Kuo and Chung-Chin Lu. *On the Hardnesses of Several Quantum Decoding Problems*. 2013. arXiv: 1306.5173 [quant-ph].

[7] Arthur Pesah. *Decoders for Topological Quantum Error Correction*. Nov. 2020. URL: https://arthurpesah.me/assets/pdf/case-study-surface-code.pdf (visited on 06/05/2024).

[8] Xiaotong Ni. "Neural Network Decoders for Large-Distance 2D Toric Codes". In: *Quantum* 4 (Aug. 2020), p. 310. ISSN: 2521-327X. DOI: 10.22331/q-2020-08-24-310. URL: http://dx.doi.org/10.22331/q-2020-08-24-310.

[9] Fern H E Watson and Sean D Barrett. "Logical error rate scaling of the toric code". In: *New Journal of Physics* 16.9 (Sept. 2014), p. 093045. ISSN: 1367-2630. DOI: 10.1088/1367-2630/16/9/093045. URL: http://dx.doi.org/10.1088/1367-2630/16/9/093045.

[10] Vladimir Kolmogorov. "Blossom V: a new implementation of a minimum cost perfect matching algorithm". In: *Mathematical Programming Computation* 1 (2009), pp. 43–67. URL: https://api.semanticscholar.org/CorpusID:17864814.

[11] Michael Herold et al. "Cellular-automaton decoders for topological quantum memories". In: *npj Quantum Information* 1.1 (Oct. 2015). ISSN: 2056-6387. DOI: 10.1038/npjqi.2015.10. URL: http://dx.doi.org/10.1038/npjqi.2015.10.

[12] Nicolas Delfosse and Naomi H. Nickerson. "Almost-linear time decoding algorithm for topological codes". In: *Quantum* 5 (Dec. 2021), p. 595. ISSN: 2521-327X. DOI: 10.22331/q-2021-12-02-595. URL: http://dx.doi.org/10.22331/q-2021-12-02-595.