CMPT 981 (Quantum Circuits and Compilation)
Final Project
Topological Quantum Circuits via Surface Code Braiding: A Review of Recent Developments

Ming Yin

11 December 2022

Abstract

Topological quantum circuits, which use highly error-correcting surface codes, are of great importance in the quest to build a fault-tolerant quantum computer. Over the last decade, significant steps have been made to optimize topological quantum circuits when viewed as braids of logical qubits. In this report, we give an overview of the background and the history of surface codes, followed by a summary of the best topological and non-topological optimization techniques currently known. We produce our own program that runs a recently discovered iterative bridging algorithm, and we discuss potential improvements to our code as well as further work that can be done in the wider topic of topological quantum circuit optimization.

1 Introduction

Surface codes are quantum error-correcting codes (QECCs), originally developed by Bravyi and Kitaev [BK98], and Freedman and Meyer [FM98], from the earlier toric codes proposed by Kitaev [Kit03]. They are of particular interest due to their relatively high tolerance to local errors compared to other QECCs: surface codes have been shown to be resilient against per-operation error rates as high as about 1% [WFH11]. One of the earliest formulations of quantum computation using surfaces codes, via *braid transformations*, was discovered by Raussendorf et al [RHG07] and shown to provide a complete set of operations for any quantum circuit. Thus these codes attract a great deal of interest and study, and are seen as a promising potential way to build a fault-tolerant quantum computer [FMMC12]. A circuit that is described using surface code braiding is commonly called a *topological quantum error-correcting* (TQEC) circuit.

Unfortunately, the trade-off for the high error tolerance provided by surface codes is that a very large number of physical qubits is required to create even one logical qubit. In their 2012 paper that provides a comprehensive introduction to the theory of surface codes, Fowler et al estimate that, given reasonable error rates on physical qubits and for various operations on the surface code, the number of physical qubits required to factorize a 2000-bit number is on the order of at least hundreds of millions [FMMC12]. Therefore, it is of vital importance that optimization techniques are developed to minimize the space-time volume occupied by a TQEC circuit, so that the number of physical qubits required (as well as runtime) can be minimized. Over the last decade, a number of different techniques have

been proposed, many of which can potentially be combined for further optimization. Some of these methods involve topologically deforming the TQEC circuit, while others perform non-topological deformation in a careful way that does not change the action of the circuit.

In this report, we first outline some key ideas to keep in mind when working with surface codes. We then describe a number of different optimization techniques for compressing TQEC circuits, focusing on a bridge-based algorithm described by Hsu et al [HLTC21]. We implement part of their algorithm in Python, and discuss the efficiency and effectiveness of this method.

2 Surface Code Preliminaries

Here, we briefly go over the basic mechanisms of surface codes and braiding. Full details can be found in [FMMC12].

A surface code is defined on a (typically planar) lattice of physical qubits, which are alternately designated the role of *data qubit* or *measurement qubit*. Furthermore, rows of measurement qubits are alternately designated as *measure-X* or *measure-Z* qubits. (See Figure 1.) The *quiescent*, or 'dormant', state of the surface code is created by performing one cycle of the circuits shown in Figure 1 (b) and (c) at all measurement qubits. We denote these circuits by $\mathcal{C}_Z$ and $\mathcal{C}_X$ respectively. This will cause the wavefunction $|\psi\rangle$ of the entire system to be transformed into an eigenstate of all operators of the form $X_aX_bX_cX_d$ and $Z_aZ_bZ_cZ_d$. Here, X and Z are the usual bit-flip and phase-flip Pauli gates, and a, b, c, d refer to the four data qubits neighbouring each measurement qubit. The quiescent state is stabilized by all these operators, and is thus unchanged by any subsequent cycles of the circuits $\mathcal{C}_Z$ and $\mathcal{C}_X$.

We introduce two degrees of freedom by creating a pair of *defects* of the same type. This is done by choosing two measurement qubits of the same type and 'switching them off', i.e. not performing the circuit $\mathcal{C}_Z$ or $\mathcal{C}_X$ at these particular measurement qubits. (See Figure 2.) Informally, we can understand the mathematics of this procedure as follows: before, we had the same number of data qubits as stabilizers, so there was a unique stabilizer state corresponding to the quiescent state (up to phase). Now, after introducing these defects, we have two fewer stabilizers than before, so we now have two extra degrees of freedom of stabilizer states to work with. We can therefore define a logical qubit this way.

In the literature, X-cut defects are also called *primal* defects, and are typically coloured red. Z-cut defects are called *dual* defects and are typically coloured blue.

A TQEC circuit described by defects can perform logical qubit initialization and measurement, and *braiding* (informally, wrapping one logical qubit around another) can be used to perform gates in the Clifford+T gateset, $\{CNOT, H, T\}$, which is universal. We can represent each of these operations by a diagram describing the time evolution of defects. Figure 3 shows the *canonical braids* for these operations. In these diagrams, the positive time axis points upwards, and the spatial plane of the surface code is shown as a one-dimensional projection. For simplicity, the mechanisms for the Hadamard gate H, the preparation of $|A\rangle$ and $|Y\rangle$ states, and the gate teleportation for implementing a T gate are represented by boxes.

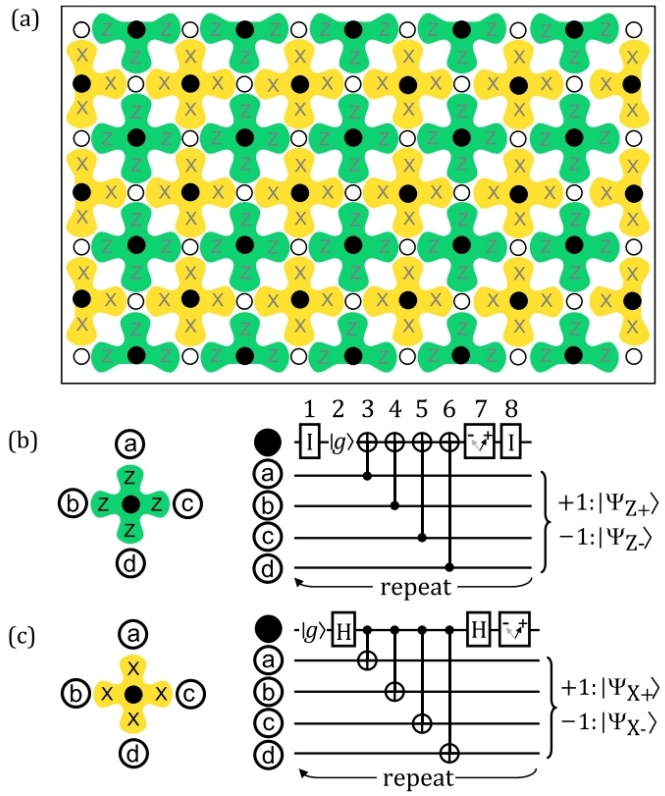


Figure 1: Diagram of a surface code. The circuits in (b) and (c) transform the state of the entire system into an eigenstate of all stabilizers of the form $X_a X_b X_c X_d$ and $Z_a Z_b Z_c Z_d$ [FMMC12].

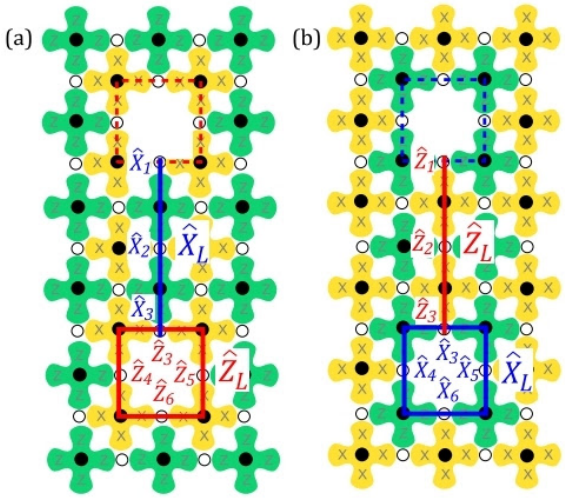


Figure 2: Illustration of the creation of (a) a Z-cut (dual) defect and (b) an X-cut (primal) defect. The logical bit-flip (X_L) and phase-flip (Z_L) operators, which are simply products of appropriate physical X and Z gates, are also shown [FMMC12].

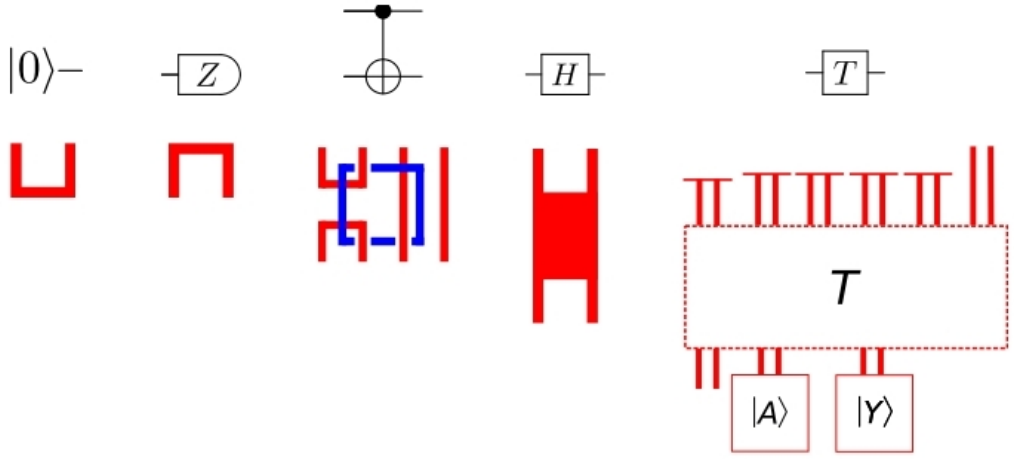


Figure 3: The canonical braid for each basic operation in a TQEC circuit [PF13].

3 Compression of Topological Quantum Circuits

3.1 Background

As mentioned previously, we would like to compress TQEC circuits, both spatially and temporally, to improve their efficiency. Spatial compression means that we can use a smaller lattice and hence fewer physical qubits.

Paetznick and Fowler proposed two optimization techniques in 2013 that use topological deformation of surface code braids [PF13]. Informally, topological deformations include stretching, squashing, bending and rotating strings in the braid, and it can be shown that all surface code braids are invariant under such deformations. First, Paetznick and Fowler describe a force-directed compression algorithm, where the braid is acted on by a gravitational force, which moves everything towards time 0, and a tension force, which shrinks loops. At the time of publication of their paper, the authors were working on a C++ program (called Braidpack) that would run this algorithm, although it does not appear to be publicly available and the current status of the project is unclear. Second, the authors describe a simulated annealing algorithm, where the problem is formulated as a minimization problem with a large number of linear constraints. The algorithm uses a hill-climbing process to look for the subset of constraints that results in the surface code braid that occupies the smallest volume.

Unfortunately, both of these techniques have a number of significant limitations. For example, the force-directed algorithm is vulnerable to becoming stuck in a local minimum, and the final outcome is highly dependent on the initial configuration. Furthermore, these methods only perform topological deformation, which has a relatively small effect compared to non-topological deformation [HLTC21].

An important idea for compression of TQEC circuits was proposed by Asai and Yamashita in 2019 [AY19]. They introduce the process of *modularization*, where the braid is broken up into different modules, each one represented by a cuboid enclosing one primal loop. It is easy to keep track of which dual loops penetrate each primal loop and hence are found in the corresponding module. After creating these modules, the authors propose a number of optimization steps. An optimal placement can be found for the modules by

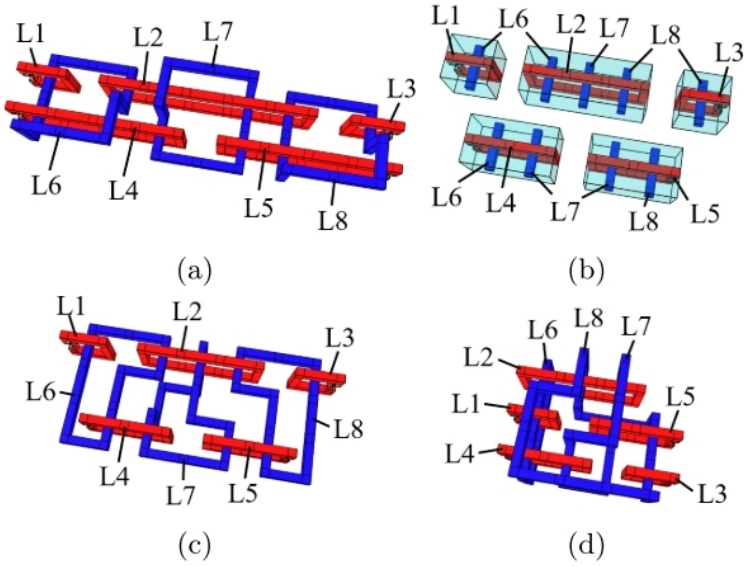


Figure 4: An illustration from Asai and Yamashita showing the modularization process on a TQEC circuit for a swap gate. The diagram in (c) shows the resulting braid if rewiring is done without any prior module placement; the diagram in (d) shows the result from their method [AY19].

treating this problem as a three-dimensional packing problem. Following this, the dual loops can be rewired using standard rewiring techniques to minimize the length of each loop.[1] (See Figure 4.)

3.2 Bridge-Based Non-Topological Compression

In 2021, Hsu et al then combined the idea of modularization with a non-topological deformation technique called *bridge compression* [HLTC21]. Bridge compression was originally discovered by Fowler and Devitt in 2012 [FD12]. Informally, given two (finite extent) dual loops, exactly one bridge connecting them may be added without changing the action of the TQEC circuit. Once the bridge is added, we can topologically deform the two loops by merging the continuous common segment. (See Figure 5.)

Hsu et al propose an iterative bridging algorithm by first modularizing the TQEC circuit, and then searching for the longest continuous common segments of loops by graph-theoretic means. Following the iterative bridging, they also propose using specific algorithms to efficiently perform the module placement and rewiring (dual defect net routing) steps: respectively, using B*-trees and A* search. Remarkably, their experimental results, using RevLib benchmarks [Jun], show an average reduction in space-time volume by 83% compared to previous work by Lin et al [LYLP18].

[1] Note that we can just as well modularize according to the dual loops, and then rewire the primal loops. For convenience, however, we will think of modularization as always being done according to the primal loops.

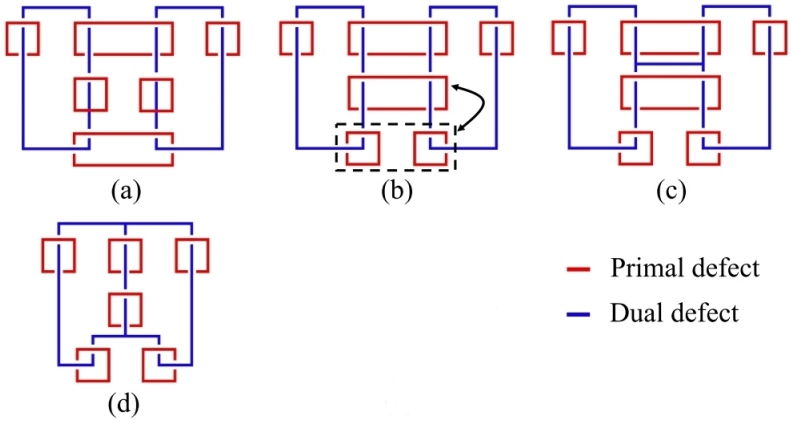


Figure 5: An illustration of dual bridging from Hsu et al. In (c), after some topological deformation, the dual loops are connected by introducing a dual bridge. Then the continuous common segment can be merged, as shown in (d) [HLTC21].

4 Code for Iterative Bridging

In this project, we re-implemented the iterative bridging algorithm described by Hsu et al, as their original C++ code appears not to be publicly available. Our program is written in Python and can be downloaded from GitLab [Yin22]. The code listing can also be found in Appendix A.

The input is a JSON file called `primal-modules.json`. This contains data specifying the number of modules according to primal loops, the number of dual loops in the circuit, and an ordered list for each dual loop specifying its penetration of primal loops. The output is a list of chain sets, each one corresponding to a dual loop after being modified by the iterative bridging algorithm. We closely follow the pseudocode on page 460 (3) of Hsu et al [HLTC21], with only minor modifications to lines 14–17.

We use virtually the same conventions as Hsu et al, with the only major exception being that we start indexing at 0 to avoid confusion within the Python code. Vertices of the graph are represented by a pair of numbers (i, j), which corresponds to the notation v_j^i used in their paper. The chain set of a loop is an ordered list of chains $[(i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n)]$, corresponding to their notation $p_{j_1}^{\prime i_1} - \cdots - p_{j_n}^{\prime i_n}$. Here, each module m_i' is simplified so that the primal loop is penetrated by just two connected pins, denoted by $p_1^{\prime i}$ and $p_2^{\prime i}$.

We ran our program with a number of different simple TQEC circuits. Appendix B shows the outputs for two of them: first, we test our code by verifying that it produces the correct output for the example circuit in Hsu et al; then we run our program for a homemade circuit of $CNOT$ gates.

5 Discussion

Our code was shown to correctly perform bridging on TQEC circuits, prioritizing bridges that would allow longer continuous common segments to be merged. As it currently stands, the time complexity of our algorithm is not ideal. While most of the functions seem to have a worst-case complexity that is polynomial, of order 3 or 4, there may be a bottleneck in the `find_valid_path` function, which currently has a worst-case time complexity of about

$O(2^{n_P} n_P^2 n_D)$, where n_P and n_D are the numbers of primal and dual loops respectively. This exponential blow-up is because the function considers every possible critical path, and any pair of vertices corresponding to the endpoint pins of one module can be swapped without altering the validity of the path. Further analysis of the underlying theory may allow us to improve the efficiency of this part of the code, e.g. if it turns out not to be necessary to check every possible path. Moreover, some of the other functions could be improved. For example, the graph construction code could probably be optimized so that, rather than creating a new graph from scratch each time, the existing graph would simply be modified. Additional data structures, in particular a `Loop` class, would help to reduce some of the redundancy caused by data being lost and having to be recalculated later in the program.

Additionally, it would be interesting to study in more detail the remaining steps of the algorithm proposed by Hsu et al. Their module placement and rewiring algorithms seem to play a significant role in allowing such massive reductions in space-time volume. We could also think about combining these methods with other techniques, both topological and non-topological, proposed by other researchers in the field.

A wider problem to consider, even with these significant improvements, is whether braiding is necessarily the most promising way to use surface codes for building a fault-tolerant quantum computer. Current optimization techniques result in TQEC circuits that would probably still need tens of millions of physical qubits, for example, to factorize a 2000-bit number using the method in Fowler et al [FMMC12]. A different formulation known as *lattice surgery* has been attracting significant levels of interest in recent years, and has been demonstrated to be remarkably efficient in physical qubit cost [Lit19]. It would be interesting to more closely examine and compare the differences between these two surface code techniques.

A Code Listing

Here is the file `bridging.py` as of 11 December 2022. Any subsequent changes to the program can be found on the GitLab page [Yin22].

```
import json
import itertools as it
import pandas as pd
from typing import List, Tuple


class Chain:
    def __init__(self, pins: List[Tuple]):
        self.pins = pins

    def __str__(self):
        return str(self.pins)


# TODO Create Loop class so that data doesn't have to keep being recalculated


class Vertex:
    def __init__(self, i, j):
        self.i = i
        self.j = j

    def __str__(self):
        return f'({self.i}, {self.j})'


```

```python
class Graph:
    def __init__(self, mat: pd.DataFrame = None):
        # Adjacency matrix
        if mat is None:
            self.mat = pd.DataFrame()
        else:
            self.mat = mat.copy()
        # Currently not used
        self.common_modules = []

    def add_vertex(self, vertex: Vertex):
        self.mat[str(vertex)] = 0
        self.mat.loc[str(vertex)] = 0

    def add_edge(self, edge: Tuple[Vertex]):
        v0, v1 = edge
        try:
            self.mat.loc[str(v0)][str(v1)] = 1
            self.mat.loc[str(v1)][str(v0)] = 1
        except KeyError:
            pass

    def is_edge(self, edge: Tuple[Vertex]):
        v0, v1 = edge
        return bool(self.mat[str(v0)][str(v1)])


def construct_graph(b_chain_sets: List[List[Chain]], l_chain_set: List[Chain]):
    graph = Graph()

    # Vertex construction

    # Find vertices from common modules between b and l
    all_b_pins = []
    for b_chain_set in b_chain_sets:
        for chain in b_chain_set:
            all_b_pins += chain.pins

    all_l_pins = []
    for chain in l_chain_set:
        all_l_pins += chain.pins

    common_pins = list(set(all_b_pins) & set(all_l_pins))
    for p in common_pins:
        i, j = p
        graph.add_vertex(Vertex(i, j))

        # Also keep track of common modules between b and l
        if i not in graph.common_modules:
            graph.common_modules.append(i)

    # Find vertices from common endpoint pins among loops in b
    for l0, l1 in it.combinations(b_chain_sets, 2):
        l0_endpoints = []
        l1_endpoints = []

        for chain in l0:
            l0_endpoints += [chain.pins[0], chain.pins[-1]]
        for chain in l1:
            l1_endpoints += [chain.pins[0], chain.pins[-1]]

        common_pins = list(set(l0_endpoints) & set(l1_endpoints))

        for p in common_pins:
            i, j = p
            graph.add_vertex(Vertex(i, j))

    # Edge construction
```

```
    all_chain_sets = b_chain_sets + [l_chain_set]

    # Join endpoints in each loop
    for l in all_chain_sets:
        all_endpoints = []
        for chain in l:
            all_endpoints.append([chain.pins[0], chain.pins[-1]])

        for c0, c1 in it.combinations(all_endpoints, 2):
            for end0, end1 in it.product(c0, c1):
                edge = (Vertex(end0[0], end0[1]), Vertex(end1[0], end1[1]))
                graph.add_edge(edge)

    # Connect vertices corresponding to consecutive pins in a chain
    for l in all_chain_sets:
        for chain in l:
            for p0, p1 in zip(chain.pins, chain.pins[1:]):
                edge = (Vertex(p0[0], p0[1]), Vertex(p1[0], p1[1]))
                graph.add_edge(edge)

    return graph


def find_valid_path(graph: Graph, critical_verts: List[Vertex],
                    b_chain_sets: List[List[Chain]], l_chain_set: List[Chain]):
    # Try every valid connecting order until we find one that preserves the loops
    verts_split = zip(critical_verts[::2], critical_verts[1::2])

    # Can swap endpoint pins within a pair
    with_swaps = []
    for pair in verts_split:
        with_swaps.append((pair, pair[::-1]))

    for pre_path in it.product(*with_swaps):
        path = list(it.chain(*pre_path))
        move_on = False
        for edge in zip(path, path[1:]):
            if not graph.is_edge(edge):
                move_on = True
                break

        if move_on:
            continue
        else:
            # Check if the path preserves the loops
            all_chain_sets = b_chain_sets + [l_chain_set]
            for l in all_chain_sets:
                ordered_modules = []
                for chain in l:
                    ordered_modules += [pin[0] for pin in chain.pins[::2]]

                # Is the path contained in the ordered list of modules?
                ordered_modules_cycle = it.cycle(ordered_modules)
                path_modules = [v.i for v in path[::2]]
                path_length = len(path_modules)
                is_preserved = False

                for _ in ordered_modules:
                    subset = list(it.islice(ordered_modules_cycle,
                                            0, path_length))
                    if path_modules == subset or path_modules[::-1] == subset:
                        is_preserved = True
                        break

                if is_preserved:
                    return list(path)

    return None
```

```
163
164
165 def create_max_priority_queue(base_loops: List[List[Chain]],
166                                candidate_loops: List[List[Chain]]):
167     max_priority_queue = []
168
169     these_modules = set([pin[0] for loop in base_loops for chain in loop
170                          for pin in chain.pins])
171
172     for index, l in enumerate(candidate_loops):
173         those_modules = set([pin[0] for chain in l
174                              for pin in chain.pins])
175         common_modules = these_modules & those_modules
176         num_common_modules = len(common_modules)
177         if num_common_modules > 0:
178             loop_data = {'index': index,
179                          'num_common_modules': num_common_modules,
180                          'common_modules': common_modules}
181             max_priority_queue.append(loop_data)
182
183     max_priority_queue.sort(key=lambda data: data['num_common_modules'],
184                             reverse=True  )
185     return max_priority_queue
186
187
188 # This code is awful lmaoooo
189 def update_chain_sets(b_chain_sets: List[List[Chain]], path: List[Vertex]):
190     for edge in zip(path, path[1:]):
191         for index, l in enumerate(b_chain_sets):
192             l_copy = l.copy()
193
194             # Merge if valid
195             first_chain = None
196             for chain in l_copy:
197                 # Bruhhh I hate how I've written this
198                 if (edge[0].i, edge[0].j) in [chain.pins[0], chain.pins[-1]] or \
199                         (edge[1].i, edge[1].j) in [chain.pins[0], chain.pins[-1]]:
200                     first_chain = chain
201
202                     try:
203                         first_index = \
204                             [chain.pins[0],
205                              chain.pins[-1]].index((edge[0].i, edge[0].j))
206                     except ValueError:
207                         first_index = \
208                             [chain.pins[0],
209                              chain.pins[-1]].index((edge[1].i, edge[1].j))
210
211                     break
212
213             if first_chain is not None:
214                 l_copy.remove(first_chain)
215                 second_chain = None
216                 for chain in l_copy:
217                     if (edge[0].i, edge[0].j) in [chain.pins[0], chain.pins[-1]] or \
218                             (edge[1].i, edge[1].j) in [chain.pins[0], chain.pins
    [-1]]:
219                         second_chain = chain
220
221                         try:
222                             second_index = \
223                                 [chain.pins[0],
224                                  chain.pins[-1]].index((edge[0].i, edge[0].j))
225                         except ValueError:
226                             second_index = \
227                                 [chain.pins[0],
228                                  chain.pins[-1]].index((edge[1].i, edge[1].j))
229
```

```
                        break

                if second_chain is not None:
                    l_copy.remove(second_chain)
                    # Merge!
                    new_pins = first_chain.pins[::(1 if first_index == 1 else -1)] +\
                        second_chain.pins[::(1 if second_index == 0 else -1)]
                    new_chain = Chain(new_pins)
                    l_copy.append(new_chain)

                    b_chain_sets[index] = l_copy


def iterative_bridging(filename):
    # Read JSON data file
    modular_data = json.load(open(filename))

    # Create starting chain set for each loop
    dual_loops = []
    for loop in modular_data['loop_penetration']:
        chain_set = []
        for m in loop:
            chain_set.append(Chain([(m, 0), (m, 1)]))
        dual_loops.append(chain_set)

    all_bridge_structures = []
    # Main loop - while dual_loops is not empty
    while dual_loops:
        # Initial bridge structure b
        bridge_structure = [dual_loops.pop(0)]

        # Get max priority queue
        max_priority_queue = create_max_priority_queue(bridge_structure,
                                                       dual_loops)

        # While max_priority_queue is not empty
        while max_priority_queue:
            loop_data = max_priority_queue[0]
            l_chain_set = dual_loops[loop_data['index']]
            # Construct bridge graph
            graph = construct_graph(bridge_structure, l_chain_set)

            # Determine the critical vertices
            critical_verts = []
            for m in loop_data['common_modules']:
                critical_verts += [Vertex(m, 0), Vertex(m, 1)]

            path = find_valid_path(graph, critical_verts,
                                   bridge_structure, l_chain_set)
            if path is not None:
                bridge_structure.append(l_chain_set)
                update_chain_sets(bridge_structure, path)

                dual_loops.remove(l_chain_set)
                # Get new max priority queue
                max_priority_queue = create_max_priority_queue(bridge_structure,
                                                               dual_loops)

        all_bridge_structures += bridge_structure

    return all_bridge_structures


if __name__ == '__main__':
    all_bridge_structures = iterative_bridging('primal-modules.json')

    print('Here are the chain sets for each loop:\n------------------')
    for index, chain_set in enumerate(all_bridge_structures):
```

```
        to_print = '{{ '
        to_print += ', '.join([str(chain_repr)
                               for chain_repr in [chain.pins for chain in chain_set]])
        to_print += ' }}'
        print(to_print)
```

B Example Outputs

B.1 Example Circuit from Hsu et al

Our input file `primal-modules.json` is as follows:

```
{
    "num_modules": 6,
    "num_dual_loops": 3,
    "loop_penetration": [
        [0, 1, 3],
        [1, 2, 5],
        [1, 4, 3]
    ]
}
```

Our program outputs the following:

```
Here are the chain sets for each loop:
------------------
{{ [(0, 0), (0, 1)], [(1, 0), (1, 1), (3, 0), (3, 1)] }}
{{ [(4, 0), (4, 1)], [(1, 0), (1, 1), (3, 0), (3, 1)] }}
{{ [(1, 0), (1, 1)], [(2, 0), (2, 1)], [(5, 0), (5, 1)] }}
```

B.2 Homemade $CNOT$ Circuit

Table 1 shows the circuit that we created, and Figure 6 shows the corresponding TQEC circuit.

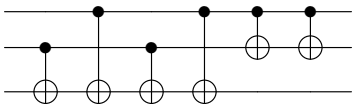


Table 1: Our homemade circuit consisting of $CNOT$ gates.

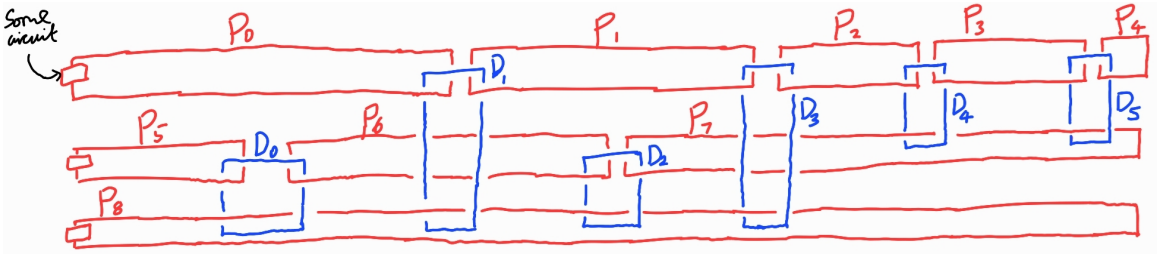


Figure 6: The TQEC circuit corresponding to our homemade circuit. The primal loops are labelled $P_0, \ldots, P_8$, and the dual loops are labelled $D_0, \ldots, D_5$.

Our input file `primal-modules-1.json` is as follows:

```
{
    "num_modules": 9,
    "num_dual_loops": 6,
    "loop_penetration": [
        [5, 8, 6],
        [0, 8, 1],
        [6, 8, 7],
        [1, 8, 2],
        [2, 7, 3],
        [3, 7, 4]
    ]
}
```

Our program outputs the following:

```
Here are the chain sets for each loop:
------------------
{{ [(5, 0), (5, 1)], [(8, 0), (8, 1), (6, 0), (6, 1)] }}
{{ [(7, 0), (7, 1)], [(6, 1), (6, 0), (8, 1), (8, 0)] }}
{{ [(0, 0), (0, 1)], [(8, 0), (8, 1), (1, 0), (1, 1)] }}
{{ [(2, 0), (2, 1)], [(1, 1), (1, 0), (8, 1), (8, 0)] }}
{{ [(3, 0), (3, 1)], [(2, 0), (2, 1), (7, 0), (7, 1)] }}
{{ [(4, 0), (4, 1)], [(3, 0), (3, 1), (7, 0), (7, 1)] }}
```

References

[AY19] Kota ASAI and Shigeru YAMASHITA. Compaction of topological quantum circuits by modularization. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E102.A(4):624–632, 2019.

[BK98] S. B. Bravyi and A. Yu. Kitaev. Quantum codes on a lattice with boundary, 1998.

[FD12] Austin G. Fowler and Simon J. Devitt. A bridge to lower overhead quantum computation, 2012.

[FM98] Michael H. Freedman and David A. Meyer. Projective plane and planar quantum codes, 1998.

[FMMC12] Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A*, 86(3), sep 2012.

[HLTC21] Chen-Hao Hsu, Wan-Hsuan Lin, Wei-Hsiang Tseng, and Yao-Wen Chang. A bridge-based compression algorithm for topological quantum circuits. pages 457–462, 12 2021.

[Jun] Lisa Jungmann. Revlib - an online resource for reversible benchmarks (`http://www.informatik.uni-bremen.de/rev_lib/`).

[Kit03] A.Yu. Kitaev. Fault-tolerant quantum computation by anyons. *Annals of Physics*, 303(1):2–30, jan 2003.

[Lit19] Daniel Litinski. A game of surface codes: Large-scale quantum computing with lattice surgery. *Quantum*, 3:128, mar 2019.

[LYLP18] Yibo Lin, Bei Yu, Meng Li, and David Z. Pan. Layout synthesis for topological quantum circuits with 1-d and 2-d architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(8):1574–1587, 2018.

[PF13] Adam Paetznick and Austin G. Fowler. Quantum circuit optimization by topological compaction in the surface code, 2013.

[RHG07] R Raussendorf, J Harrington, and K Goyal. Topological fault-tolerance in cluster state quantum computation. *New Journal of Physics*, 9(6):199–199, jun 2007.

[WFH11] David S. Wang, Austin G. Fowler, and Lloyd C. L. Hollenberg. Surface code quantum computing with error rates over 1%. *Physical Review A*, 83(2), feb 2011.

[Yin22] Ming Yin. Gitlab repository: iterative bridging for surface codes (`https://gitlab.com/minghuayin1729/iterative-bridging-for-surface-codes`), 2022.