Dynamic Qubit Allocation and Routing for Constrained Topologies by CNOT Circuit Re-synthesis

Arianne Meijer - van de Griend
Department of Computer Science
University of Helsinki
Helsinki, Finland
ariannemeijer@gmail.com

Sarah Meng Li
Institute for Quantum Computing
University of Waterloo
Waterloo, Canada
sarah.li@uwaterloo.ca

Many quantum computers have constraints regarding which two-qubit operations are locally allowed. To run a quantum circuit under those constraints, qubits need to be allocated to different quantum registers, and multi-qubit gates need to be routed accordingly. Recent developments have shown that Steiner-tree based compiling strategies provide a competitive tool to route CNOT gates. However, these algorithms require the qubit allocation to be decided before routing. Moreover, the allocation is fixed throughout the computation, i.e. the logical qubit will not move to a different qubit register. This is inefficient with respect to the CNOT count of the resulting circuit.

In this paper, we propose the algorithm *PermRowCol* for routing CNOTs in a quantum circuit. It dynamically reallocates logical qubits during the computation, and thus results in fewer output CNOTs than the algorithms *Steiner-Gauss*[12] and *RowCol*[23].

Here we focus on circuits over CNOT only, but this method could be generalized to a routing and allocation strategy on Clifford+T circuits by slicing the quantum circuit into subcircuits composed of CNOTs and single-qubit gates. Additionally, *PermRowCol* can be used in place of *Steiner-Gauss* in the synthesis of phase polynomials as well as the extraction of quantum circuits from ZX-diagrams.

1 Introduction

Recent strides in quantum computing have made it possible to execute quantum algorithms on real quantum hardware. Contrary to classical computing, efficient quantum circuits are necessary for successful execution due to the decoherence of qubits. If a quantum circuit takes too long to execute, it will not produce any usable results. Moreover, due to poor gate fidelities, each additional gate in the quantum circuit adds a small error to the computation. In the absence of fault-tolerant quantum computers, circuits with more gates produce less accurate results. Therefore, we need to reduce the gate complexity of the executed quantum circuits. This requires resource-efficient algorithms and improved quantum compiling procedures.

When mapping a quantum circuit to the physical layer, one has to consider the numerous constraints imposed by the underlying hardware architecture. For example, in a superconducting quantum computer [20], connectivity of the physical qubits restricts multi-qubit operations to adjacent qubits. These restrictions are known as *connectivity constraints* and can be represented by a *connected graph* (also known as a *topology*). Each vertex represents a distinct physical qubit. When two qubits are adjacent, there is an edge between the corresponding vertices.

Thus, we are interested in improving the routing of a quantum circuit onto a quantum computer. Current routing strategies are dominated by SWAP-based approaches [13, 22, 19, 24]. These strategies move the logical qubits around on different quantum registers. The drawback of this is that every SWAP-gate adds 3 CNOT gates to the circuit (figure 1(a)), adding only more gates to the original circuit. As a

Submitted to:
Quantum Physics and Logic (QPL) 2022

result, it will take much longer to execute a routed quantum circuit, and thus introduce more errors to the computation.

Additionally, these SWAP-based strategies can be replaced by a *bridge template* (or *bridge*) that acts like a remote CNOT. As shown in figure 1(b), the bridge template only requires 4 CNOTs while swapping the qubits would have cost 7 CNOTs (figure 1(c)). The CNOT ladders in the bridge template can be generalized for remote CNOTs with more qubits in between. However, since the bridge template does not move the qubits, we need to make a trade-off between swapping and remote CNOTs when there is a sequence of CNOTs to be routed.

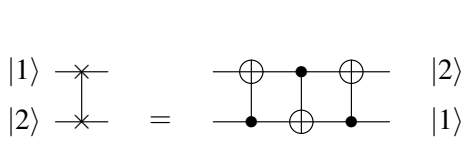


Figure (a): A swap gate implemented by 3 CNOTs.

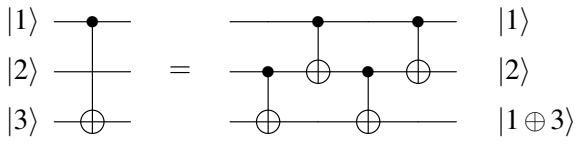


Figure (b): A bridge implemented by 4 CNOTs.

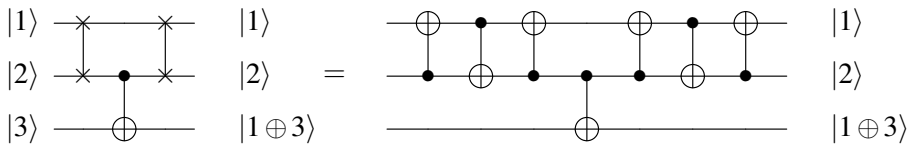


Figure (c): Routing CNOT$(1,3)$ with SWAP gates results in 7 CNOTs.

Figure 1: Visualization of how a SWAP gate and a bridge template is implemented by CNOTs respectively. Note that the bridge gate is the same as the output of the *Steiner-Gauss* algorithm for a single CNOT constrained over a 3-qubit line topology.

Alternatively, we can use Steiner-tree based synthesis [2, 12, 15, 9, 21, 8] to find a generalized bridge gate for a CNOT circuit. The new sequence of CNOTs has the same effect on the logical states as the original CNOT circuit, and every gate is permitted by the hardware's *connectivity constraint*. This is done by changing the rigid representation of the quantum circuit to a more flexible one, with which we could synthesize a routed circuit (section 2). This way, we can make global improvements to the CNOT circuit re-synthesis more easily.

The *Steiner-Gauss* algorithm provides the first Steiner-tree based synthesis approach and it is used to synthesize CNOT circuits [12, 15]. Then, it is used for synthesizing circuits over CNOT and R_z gates [15, 9, 21], and further for circuits over CNOT, R_z, and NOT gates. Finally, this algorithm is generalized for Clifford+T circuits with the *slice-and-build* procotol based on the locality of Hadamard gates in the circuits [8].

One major drawback of the existing Steiner-tree based methods is that they are not flexible with respect to the allocation of qubits. Logical qubits of the synthesized circuit will always be stored in the same physical qubit registers where they were originally allocated. However, this is not always optimal. Figure 2 shows an example where there exists a smaller synthesized circuit by reallocating logical qubits to the physical registers (figure 2(c)) than using *Steiner-Gauss* (figure 2(d)). Note that with the fixed qubit allocation, *Steiner-Gauss* produces less CNOTs than the SWAP-based method (figure 2(b)). Thus, we conclude that the synthesized circuit in figure 2(d) contains implicit SWAP-gates.

Moreover, remapping logical outputs of a quantum circuit to physical registers based on the original qubit allocation can be done by a classical operation. Hence, such operation is considered trivial in

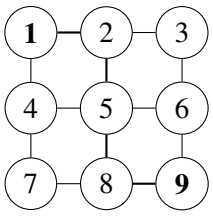


Figure (a): The 9-qubit square grid.

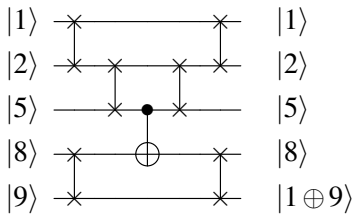


Figure (b): SWAP template with a fixed qubits allocation (CNOT count: 19).

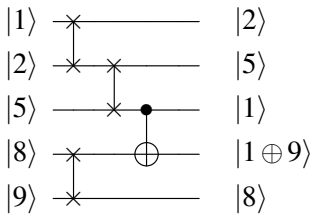


Figure (c): SWAP template with dynamic qubits allocations (CNOT count: 10).

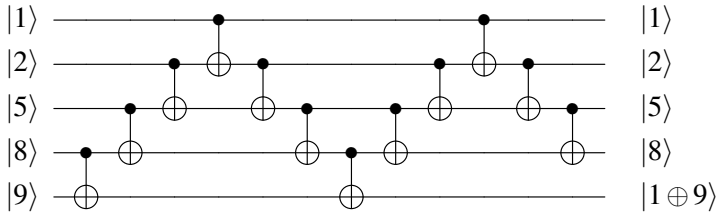


Figure (d): *Steiner-Gauss* with a fixed qubits allocation [12] (CNOT count: 12).

Figure 2: Compare three different routing strategies for CNOT$(1,9)$ under a constrained topology as shown in figure (a).

quantum computing. In other words, preparing a qubit on a register and measuring it from a different register does not influence the computation of a quantum circuit. Thus, it is desirable to have a synthesis procedure that permits dynamic qubit allocations.

The first CNOT synthesis procedures [15, 12] under topological constraints are based on Gaussian elimination: the parity matrix representing the synthesized CNOT circuit is eliminated to the identity matrix (section 2.3). In section 2.4, we show that dynamically changing the qubit allocations is the same as eliminating the parity matrix into the identity matrix up to permutation. To do this, we need to determine to which permutation of the identity matrix to synthesize a priori, which is not a trivial task. However, by adjusting algorithm *RowCol* [23], we can determine the new qubit allocation whilst synthesizing a CNOT circuit.

In this paper, we propose algorithm *PermRowCol*: a new Steiner-tree based synthesis method for CNOT circuits re-synthesis under topological constraints. It dynamically chooses the output qubit allocations. As discussed earlier, this method could be generalized to synthesizing an arbitrary quantum circuit. We will further illustrate this in section 6.2.

The paper is structured as follows. In section 2, we introduce the Steiner-tree based synthesis approach. In section 3, we describe our algorithm *PermRowCol*. In section 4, we show how well it performs against *Steiner-Gauss* [12] and *RowCol* [23]. Then, we discuss the implications of our work in section 5,

and the possible improvements for *PermRowCol* that we are currently working on in section 6. Looking forward, our goal is to fairly compare *PermRowCol* against CNOT re-synthesis algorithms in common compilers such as SABRE [13], Qiskit [22], and TKET [19].

2 Preliminaries

Here we introduce the core concepts required to understand the proposed algorithm *PermRowCol*. In section 2.1, we define the matrix representation of a CNOT circuit. In section 2.2, we describe the concepts of Steiner tree. In section 2.3, we use it for synthesizing a CNOT circuit under a constrained topology. In section 2.4, we explain the core idea behind algorithm *PermRowCol*: the dynamic reallocation of qubits using permutation matrices.

2.1 The parity matrix of a CNOT circuit

In this paper, we consider circuits composed of only CNOTs, and call them *CNOT circuits*. CNOT is short for "controlled not". It acts on two qubits: a control and a target. We write CNOT(c,t) to denote a CNOT applied between a control qubit c and a target qubit t. The control qubit c decides whether a NOT gate is applied to the target qubit t. When $|c\rangle = |0\rangle$, CNOT(c,t) acts trivially on $|t\rangle$, leaving it unchanged. Otherwise, $|t\rangle = |0\rangle$ is changed to $|t\rangle = |1\rangle$ and vice versa. Alternatively, we write that CNOT(c,t) changes $|t\rangle$ to $|c \oplus t\rangle$, where $\oplus$ denotes addition modulo 2, as shown in figure 3.

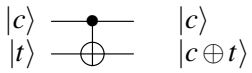
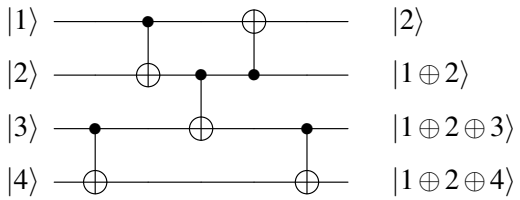


Figure 3: The circuit notation for CNOT(c,t).

For a CNOT circuit, we can keep track of it state evolution by checking which qubits appear in the sum (modulo 2) at the circuit output. In figure 4(a), there are 5 CNOTs acting on 4 qubits, whose overall behaviour is described by the sum of some logical qubits on each output wire. We call such a sum a *parity* because it keeps track of whether a logical qubit participates in the sum or not. As such, we can write a parity as a binary string whose length is equal to the number of qubits in the circuit. In this representation, a 0 means that the corresponding logical qubit is not present in the sum, and a 1 means otherwise.



(a) A CNOT circuit composed of 4 qubits.

$$\mathbf{A} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1' & 2' & 3' & 4' \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

(b) Parity matrix for figure 4(a).

Figure 4: The matrix representation of a 4-qubit CNOT circuit.

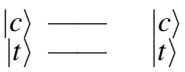


Figure (a): An empty 2-qubit circuit.

$$\begin{array}{c} \\ c \\ t \end{array} \begin{array}{c} \begin{array}{cc} c' & t' \end{array} \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{array}$$

Figure (b): Parity matrix for figure 5(a).

$$\begin{array}{c} \\ c \\ t \end{array} \begin{array}{c} \begin{array}{cc} c' & t' \end{array} \\ \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \end{array}$$

Figure (c): Parity matrix for figure 3.

Figure 5: Properties of parity matrix.

We can use the output parities of a CNOT circuit to create a square matrix where each column represents a parity and each row represents an input qubit. This matrix is called a *parity matrix* and it has some interesting properties. For example, in figure 5, an empty circuit corresponds to the identity matrix. Additionally, applying a CNOT corresponds to adding the target row to the control row in the parity matrix.

This means that we can extract a CNOT circuit from a parity matrix by adding rows of the parity matrix to each other until we obtain the identity matrix. A commonly used algorithm to do this is *Gaussian elimination* [16] which we assume our readers know from linear algebra.

2.2 Steiner tree

We use Steiner trees to enforce the connectivity constraints when synthesizing a new CNOT circuit from the parity matrix. Note that Steiner trees are not strictly necessary to constrain the CNOT circuit synthesis procedure (see e.g. [4] for an alternative) but it is the method we use.

We start at the basics; a *graph* is an order pair $G = (V_G, E_G)$, where V_G is a set of *vertices* and E_G is a set of edges. Each edge is defined as $e = (u, v)$ where $u, v \in V_G$. Each such pair is called an *edge*. The *degree* of a vertex is the number of edges that are incident to that vertex. Graphs also have a *weight* assigned to each edge by a weight function $\omega_E : E_G \to \mathbb{R}$.

The connectivity graph of a quantum computer is generally considered a *simple* graph, meaning that it is an *undirected* graph (i.e., $(u, v) \equiv (v, u)$) with all edge weights equal to 1 that has at most one edge between two distinct vertices (i.e. $\forall (e, e') \in E_G : e \neq e'$), and no self-loops (i.e., $(u, u) \notin E_G$).

Some graphs are *connected*, meaning that for every vertex there exists a sequence of edges (a *path*) from which we can go from that vertex to any other vertex in the graph. A graph that is not connected is *disconnected*. The topology of a quantum computer needs to be connected if we want any pair of arbitrary qubits to interact with each other. A *cut vertex* is a vertex that when removed, the graph will become disconnected. We use *non-cutting vertex* to mean a vertex that is not a cut vertex.

A *subgraph* $G' = (V_G^{'}, E_G^{'})$ of G is a graph that is wholly contained in G such that $V_G^{'} \subseteq V_G$, $E_G^{'} \subseteq E_G$, and $\forall (u, v) \in E_G^{'} : u, v \in V_G^{'}$.

A *tree* is an undirected connected graph that has no path which starts and ends at the same vertex. A tree is *acyclic*. A *minimum spanning tree* T of a connected graph G is a subgraph of G with the same vertices V_G and the subset of the edges E_G such that the sum of the edge weights is minimal and T is still connected.

A Steiner tree is similar to a minimum spanning tree:

Definition 2.1 (**Steiner tree**). Given a graph $G = (V_G, E_G)$ with a weight function ω_E and a set of vertices $S \subseteq V_G$, a Steiner tree $T = (V_T, E_T)$ is a tree that is a subgraph of G such that $S \subseteq V_T$ and the sum of the weights of the edges in E_T is minimized. The vertices in S are called *terminals* while those in $V_T \setminus S$ are called *Steiner nodes*.

Computing Steiner trees is NP-hard and the related decision problem is NP-complete [11]. There are a number of heuristic algorithms that compute approximate Steiner trees [17, 5, 10]. There is a trade-off between the size of the approximate Steiner tree and the algorithm's runtime, so the choice of algorithm is determined by its application. Here we use Prim's and Floyd-Warshall algorithm [6] to build a minimal spanning tree over all terminals using all-pairs-shortest-paths as weights and adding the paths to the tree.

2.3 Synthesizing CNOT circuits for specific topologies

Given the parity matrix of a CNOT circuit, we want to synthesize an equivalent CNOT circuit such that all CNOTs are allowed according to a connectivity graph. From section 2.1, we know that every CNOT corresponds to a row operation in the parity matrix and that we can use Gaussian elimination to turn the parity matrix into the identity matrix. If we keep track of which row operations are performed during the Gaussian elimination process, we obtain a CNOT circuit that is semantically equivalent. *Semantically equivalent* means that the parity matrix of the original circuit is equal to that of the synthesized circuit. Hence, these two circuits have the same input-output behaviour.

If we want the extracted CNOTs to adhere to the given connectivity constraints, we need to adjust our method to allow only row operations that correspond to connected vertices in the topology. There are several methods to do this [12, 15, 16], but our algorithm is based on algorithm *RowCol* [23], so we will explain this method.

The task of CNOT circuit synthesis under topological constraints is to turn a parity matrix into the identity matrix using the allowed row additions. This means that we need to add rows together in the matrix such that each row and column only contains a 1 on the diagonal and 0 everywhere else. We call this process of adding rows *elimination*. *RowCol* is based on the strategy to pick a qubit and eliminate its corresponding column and row such that the row and column are identity and the qubit is no longer needed. Steiner trees are used to find the best path over which the row additions are performed. Then, we can remove the row and column from the parity matrix as well as the vertex from the topology and restart the algorithm on the smaller problem.

To make sure that the topology stays connected, the qubit is chosen to be a non-cutting vertex. Due to the structure of the *RowCol* algorithm, we can remove the qubits in arbitrary order as long as the order does not disconnect the connectivity graph.

A column is eliminated by identifying all 1s in the column and building a Steiner tree T with the diagonal as root and the rows with a 1 as terminals. We want to add the rows with a 1 together such that we end up with an identity column. However, due to the connectivity constraints, we need to use the Steiner nodes to "move" the 1s to the terminals. We do this by traversing T from the bottom up, when we reach a Steiner node, we add its child to itself, making the row at the column we are eliminating into a 1. Then, we traverse T again and adding each row to its children when reached. As a result, only the root will have a 1 in the column we are eliminating and every other row will have a 0. Thus, the column is eliminated. This procedure is also described as pseudo-code in appendix A in algorithm 4.

A row is eliminated in a similar manner. However, it is less straightforward to find which rows need to be added to eliminate the desired row. We can find these rows by solving the system of linear equations defined by the parity matrix. Then, we can once more build a Steiner tree T' with the diagonal as root and the rows to add to the desired row as terminals. Then we traverse T' top down from the root, adding

every Steiner node to its parent. Then, we traverse T' bottom up and add every node to its parent. As a result, the rows belonging to the Steiner node are added twice to its parent and they do not participate in the sum because we add rows modulo 2. Moreover, every terminal is added together and propagated to the root with the bottom up traversal. This procedure is also described as pseudo-code in appendix A in algorithm 5.

By iteratively picking a non-cutting vertex from the graph, eliminating its column and row, and removing these three from the problem, we have a functional CNOT circuit synthesis strategy that takes the given topology into account. Moreover, the algorithm is semantics preserving, so that the parity matrices for both the original and synthesized circuit are equal.

2.4 Dynamic qubit allocation with permutation matrices

Current CNOT synthesis methods from parity matrices synthesize the parity matrix exactly. Meaning that the transformation described by the CNOT circuit is exactly the given parity matrix. Clearly, it makes sense to design an algorithm to preserve the semantics of the parity matrix. However, in the case of using CNOT synthesis methods for routing CNOTs in a quantum circuit, the parity matrix might be too rigid a representation.

Recall that in the parity matrix, the columns corresponds to the parities that need to be created by the quantum circuit. The order of the columns corresponds to the different qubit registers on which those parities are created. That means that if we can arbitrarily change which parities end up on which qubit registers, we can equivalently synthesize a CNOT circuit where the parity matrix has its columns reordered with respect to the original parity matrix. In the case of routing CNOTs, this is exactly possible. Moreover, SWAP-based methods are based on the fact that logical qubits can be moved around to different qubit registers. Crucially, reading the circuit output from different registers is a classical operation and therefore can be considered free in the quantum circuit.

Thus, by synthesizing the exact parity matrix, we are implicitly adding the constraint that each logical qubit needs to end up in the same qubit register as it started out with. Since this constraint is not necessary, we are implicitly swapping the logical qubits back to their original registers, adding unnecessary extra CNOT gates. This point is also illustrated in figure 2 where the synthesized circuit (figure 2d) might have had less CNOTs if we would allow dynamic allocation (figure 2c) even though it already uses less CNOTs than the SWAP strategy with a fixed allocation. Thus, it is better to have an algorithm that is flexible in its qubit allocation. Additionally, if the CNOT synthesis is done as part of a slice-and-build approach where the circuit is cut into pieces (e.g. in [8]) the cost of keeping the logical qubits in the same qubit registers grows linearly with the number of slices.

Unfortunately, it is not trivial to determine an optimal reallocation a priori. In fact, this is why the *Steiner-Gauss* results from [12] used a genetic algorithm to find a better qubit allocation.

Our novel insight is that a parity matrix that is the identity matrix is essentially an allocation map where logical qubit i is stored on the corresponding qubit register i. Then, reordering the columns of the identity matrix corresponds to reading the logical qubits from different quantum registers that are defined by the new column order. An identity matrix with reordered columns is also called a *permutation matrix*. Thus, a parity matrix M that is a permutation matrix can be seen as an *allocation map* where logical qubit i is stored in qubit register j iff $M_{i,j} = 1$. Therefore, it is sufficient to eliminate a parity matrix to a permutation matrix rather than identity. In the next section, we will give our new algorithm to do this.

2.5 Reverse traversal strategy

A benefit of such an algorithm is that we can use the *Reverse Traversal Strategy* (RT) [13]. RT uses the fact that quantum circuits are reversible. Suppose we have a routing procedure that reallocates the qubits to different registers, given an initial allocation. Then we can find a new initial mapping by running our routing procedure on the reverse circuit. If the routing procedure optimizes in the process, we might find a smaller circuit. We can iteratively repeat this processes of routing the reverse circuit to obtain a new initial mapping to possibly find a better circuit.

This technique can only be used in a routing method that reallocates the qubits. Therefore, RT could not be used for Steiner-tree based methods until now.

3 Algorithm

We propose *PermRowCol*, an Steiner-tree based algorithm that can dynamically reallocate the logical qubits while routing. It does this by eliminating the parity matrix to a permutation matrix instead of the identity matrix. To do this, we build on the *RowCol* [23] algorithm that we explained in section 2.3. The algorithm picks a logical qubit and eliminates its row and corresponding column such that they can be removed from the problem. Our adjustment is rather simple, we disconnect the row and column to be removed such that they don't necessarily overlap at the diagonal. Specifically, we pick the logical qubit corresponding to the row, and a column to be the new register for that logical qubit. Then, we can eliminate both the chosen column and the chosen row such that they can be removed from the problem. We give the pseudo-code for our algorithm in algorithm 1. The algorithm makes use of several subroutines that we give as pseudo-code in algorithm A but we will explain their behaviour here.

We start with a parity matrix M to synthesize over a topology graph G where the labeling of G corresponds to the numbering of rows in M. First, we need to pick the logical qubit that we want to remove from our problem. The only constraint for picking is it needs to be a non-cutting vertex. So, we calculate the set of non-cutting vertices of G. These vertices correspond to the rows we can choose to eliminate. Then, we choose one of those rows. For our results we used the simple heuristic: the row with the least 1s in the parity matrix M (see algorithm 2).

Next, we can choose any of the columns as the qubit register where the logical qubit needs to be stored after calculation, as long as it does not have a qubit assigned to it yet. The heuristic we used is the column that has a 1 on the chosen row and the least amount of 1s in the column (see algorithm 3).

Note that we can choose any arbitrary row and column, as long as the chosen row will not disconnect the graph G when removed and the row and column have not been picked before. Thus, we can replace algorithm 2 and 3 by any other heuristic as long as the non-cutting constraint of the row is satisfied. We discuss the implications of our choice in section 5 and a possible improvement in section 6.

With the row and column chosen, we can start to add rows together such that the row and column only have a 1 at their intersection in the matrix M, i.e. eliminate them. We start with the column and gather the rows with a 1 in that column. Then, we want to build a Steiner tree with the chosen row as root and the gathered rows as nodes. For each Steiner node in the tree, we add its child to it, starting at the leafs. The leafs of the tree corresponds to rows in M with a 1 in the chosen column. By construction, the Steiner nodes have a zero in the chosen column of M. Thus, with this method all Steiner nodes will be turned into a 1. Then, we traverse the tree again from the leafs to the root and adding every parent to its child, resulting in a matrix with all zeros in the chosen column except for at the chosen row (see algorithm 4).

Algorithm 1: *PermRowCol*

Input : Parity matrix M and topology $G(V_G, E)$ with labels corresponding to the rows of M
Output: CNOT circuit C and output qubit allocation P

$P \leftarrow [-1 \cdots -1]$; /* $|V_G|$ times */
$C \leftarrow$ New empty circuit;
while $|V_G| > 1$ **do**
  /* Find the qubits Vs that can (still) be removed without disconnecting G. */
  $Vs \leftarrow NonCuttingVertices(G)$;
  $r \leftarrow$ **ChooseRow**(Vs, M);
  /* Choose qubit register to allocate r to. */
  $c \leftarrow$ **ChooseColumn**$(M, r, [i : i \in [1 \ldots |P|]$ where $P[i] = -1])$;
  $Nodes \leftarrow [i : i \in V_G$ where $M_{i,c} = 1]$;
  $C.add($**EliminateColumn**$(M, G, r, Nodes))$;
  /* Reduce the row if it is not yet done */
  if $\sum_{j \in 1 \ldots |P|} M_{r,j} > 1$ **then**
    $A \leftarrow M$ without row r and without column c;
    $B \leftarrow M[r]$ without column c;
    $X \leftarrow A^{-1}B$; /* Find rows to add to eliminate row r */
    $Nodes \leftarrow [i : i \in V_G$ where $i = r$ or $X[Index(i)] = 1]$;
    $C.add($**EliminateRow**$(M, G, r, Nodes))$;
  end
  /* Update the output qubit allocation */
  $P[c] \leftarrow r$;
  $G \leftarrow$ subgraph of G with vertex r and connecting edges removed;
end
/* Update the last output qubit allocation because the loop ends with 1 qubit in G */
$i \leftarrow [i : i \in [1 \ldots |P|]$ where $P[i] = -1]$; /* Find index with -1 */
$P[i] \leftarrow V_G[0]$;
return C, P

Next, we do a similar thing for the chosen row. First, find which rows need to be added together such that the entire chosen row is filled with 0s except for at the chosen column. We do this by solving the system of linear equations defined by M. Then, we make a new Steiner tree with the chosen row as root and the found rows as nodes. Again we traverse the tree twice, once top-down and only adding Steiner nodes to their parents and once bottom-up and adding all nodes to their parents. Again, this results in adding all gathered rows to the chosen row and thus eliminating the chosen row (see algorithm 5).

Lastly, we update the output qubit allocation to keep track of our choices of row and column.

Since the CNOT generation is based on elementary row operations on M, we construct the Steiner trees based on the rows of M. Once a row in M becomes an identity row (i.e. only contains a single 1), the row is done and we do not need it anymore. Thus, we can remove the corresponding vertex from the topology and restart the algorithm on the smaller problem, ensuring termination of the algorithm.

The time complexity depends on the choice of heuristics and the choice of (approximate) Steiner tree algorithm when eliminating a row and column. The remaining time complexity is dominated by the calculation of the non-cutting vertices. Thus, given N qubits and a topology with E edges, the time complexity for PermRowCol is

$$\begin{aligned} O(PermRowCol) &= O(N)\big(O(NonCuttingVertices) + O(ChooseRow) + O(ChooseColumn) \\ &\qquad + O(EliminateColumn) + O(EliminateRow)\big) \\ &= O(N)\big(O(NonCuttingVertices) + O(ChooseRow) + O(ChooseColumn) \\ &\qquad + 2 * O(Steinertree)\big) \end{aligned}$$

The suggested ChooseRow and ChooseColumn heuristics both have time complexity $O(N^2)$, but these can be replaced by other heuristics that might have different complexities. Similarly, the complexity of the Steiner tree algorithm depends heavily on the choice of algorithm. Building Steiner trees is NP-hard but current topologies are sparse enough that approximate methods perform well. The algorithm we use is based on Prim's and Floyd-Warshall algorithm and it has time complexity $O(N^3)$. Thus, the time complexity for our specific implementation of PermRowCol is

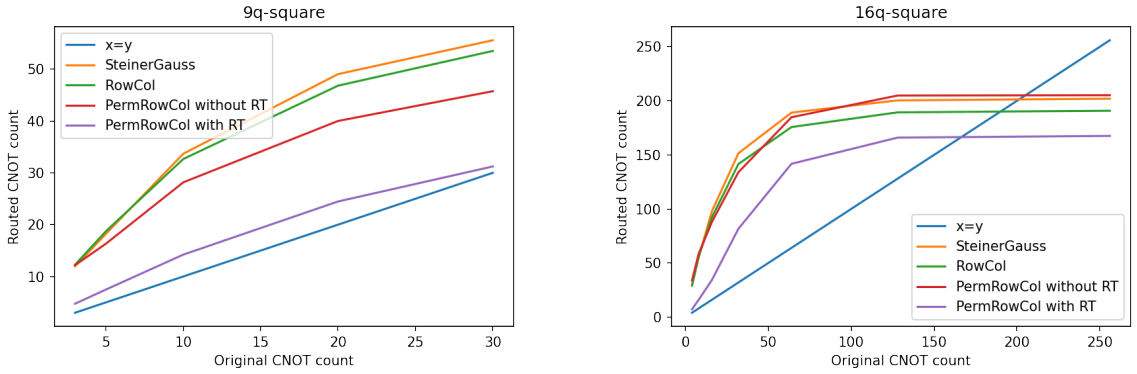
$$O(PermRowCol) = O(N)\big(O(N^2 + EN) + O(N^2) + O(N^2) + 2 * O(N^3)\big) = O(N^4).$$

4 Results

We test our algorithm against *Steiner-Gauss* (implementation from [12]) and *RowCol* to show the effect of dynamically reallocating qubits during synthesis, both with and without *Reverse Traversal* (RT). To do this, we used the same test CNOT circuits that were used for [12]. These are circuits with q number of qubits and d CNOTs that were sampled uniformly at random. The dataset provided by [12] only contained 20 circuits per (q,d)-pair and we used the same script to add 80 more random circuits resulting in 100 circuits per (q,d)-pair. Our implementation of these 3 algorithms, the CNOT circuit generation script, unit tests, as well as the dataset of random CNOT circuits that we used can be found on GitHub[1].

We plot the number of CNOTs in the routed circuit against the number of CNOTs in the original random circuit. The blue $x = y$ line helps to show the routing overhead of the algorithms. If a point is above the blue line, then the routed circuit required more CNOTs than the original circuit. This is expected, but we want as few CNOTs as possible. Similarly, when the original CNOT circuit had many CNOTs, it is

[1] `https://github.com/Aerylia/pyzx/tree/rowcol`



(a) Algorithm performance for the 9-qubit square grid. (b) Algorithm performance for the 16-qubit square grid

Figure 16: These figures show the number of CNOTs generated by *Steiner-Gauss* [12] (orange), *RowCol* [23] (green), and *PermRowCol* without *Reverse Traversal* (RT) (proposed, red) and with RT (proposed, purple) for different fictitious square grid topologies: 9 qubits (16a) and 16 qubits (16b). The blue $x = y$-line can be used to infer the CNOT overhead.

possible for the algorithms to extract less CNOTs than in the original circuit. This means that the original circuit contains redundant CNOTs which are optimized away by the algorithms. This happens because after a certain amount of CNOTs, the parity matrix representing the circuit becomes a random matrix and synthesizing a random parity matrix requires a constant amount of qubits, as discussed in [16].

Figure 16 shows the performance of the three algorithms on two fictitious square grid topologies whereas figure 17 shows the performance on three real device topologies (the connectivity graphs are shown in appendix C figure 19). We can clearly see that the proposed *PermRowCol* algorithm outperforms the other algorithms for small numbers of CNOTs. For large number of CNOTs, it depends on the topology whether the proposed algorithm still outperforms the baselines. We will discuss the reason for this in section 5.

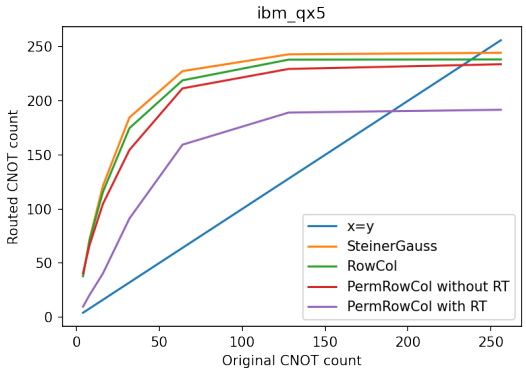
However, when we combine our *PermRowCol* algorithm with the *Reverse Traversal* strategy the method outperforms the baselines on all architectures. As seen by the purple line in figure 16 and 17. Note that because the baseline algorithms cannot reallocate the qubits, RT would have had no effect on the baselines.

Our code for these results are available on the GitHub repository[2].

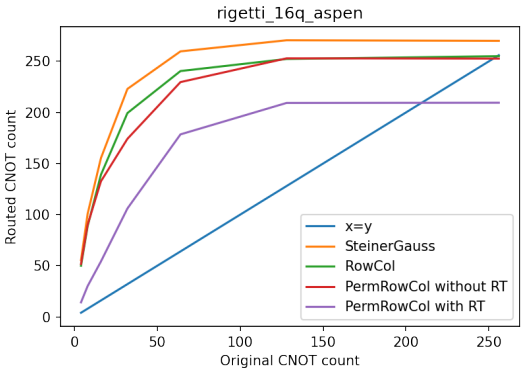
5 Discussion and conclusion

In this paper, we proposed a new algorithm for synthesising CNOT circuits from a parity matrix which dynamically reallocates the qubits to different qubit registers while respecting the connectivity constraints of a quantum computer. This technique is meant for global optimization of quantum circuits. This is the type of optimization that cannot be done with peep-hole optimization. Our research was based on the assumption that the freedom obtained by allowing dynamic allocation in the synthesis procedure can reduce the CNOT count. This assumption has been confirmed and we gave a framework from which an improved algorithm can be constructed.

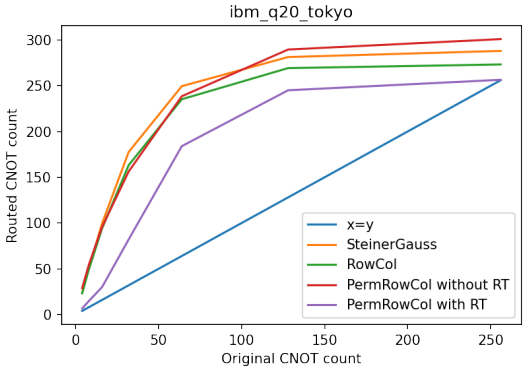
[2] `https://github.com/Aerylia/pyzx/blob/rowcol/demos/PermRowCol%20results.ipynb`



(a) Algorithm performance for the 16-qubit IBM QX5.



(b) Algorithm performance for Rigetti 16Q Aspen.



(c) Algorithm performance for the 20-qubit IBM Tokyo.

Figure 17: These figures show the number of CNOTs generated by *Steiner-Gauss* [12] (orange), *RowCol* [23] (green), and *PermRowCol* without *Reverse Traversal* (RT) (proposed, red) and with RT (proposed, purple) for different real topologies: IBM QX5 (17a), Rigetti Aspen (17b, and IBM Tokyo (17c). The blue $x = y$-line can be used to infer the CNOT overhead.

We have shown that in some but not all cases, the reallocation results in a smaller CNOT overhead with respect to the baselines. However, the addition of *Reverse Traversal* (RT) improves the results noticably.

That being said, there are still improvements to make. Particularly, we need to identify why in some cases, *PermRowCol* performs worse than the original *RowCol*. *RowCol* differs from *PermRowCol* because we pick a different row and column to eliminate, resulting in the reallocation of qubits. If *PermRowCol* produces more CNOT overhead than that of *RowCol*, it means that we picked an inefficient reallocation. Thus, the simple heuristics that we have used in our algorithm does not perform as expected when synthesizing a random parity matrix. This makes sense because our heuristic relies on the number of 1s in each row and column; and in a random parity matrix, the number of 1s in each row and column is approximately the same. Therefore, the *PermRowCol* algorithm in our choice of heuristic may decide to eliminate a random row and column that will simplify the problem to a parity matrix in a way that requires more CNOTs.

This suspicion is strengthened when comparing the algorithms' performance on synthesizing the same circuits but with different topologies. For example, the 16-qubit square grid machine, the IBM

QX5 machine, and the Rigetti Aspen machine all consist of 16 qubits, but have distinct connectivity constraints. As shown in figures 16 and 17, for large circuits (≥ 200 CNOTs), *PermRowCol* generates the most CNOTs with respect to the 16-qubit square grid (appendix C(c)), but the least for Rigetti Aspen and IBM QX5 (appendix C). This might be because the topologies of the real devices are sparser than their fictitious counterparts (see table 1). This means that when removing random non-cutting vertices from the connectivity graph, we will not have many options to choose from for the real devices. Conversely, removing random non-cutting vertices from a square grid will restrict the options less because the vertices are connected through more paths which makes it less likely to pick an inefficient option. However, when synthesizing circuits without any connectivity constraints, appendix B and figure 18 show otherwise. We suspect that this is because picking an inefficient qubit allocation is less detrimental when CNOTs are allowed between arbitrary qubits.

Qubit Count	Topology		Avg. Graph Distance	Average Degree
16	Fictitious devices	16Q-Square	2.5	4
		16Q-Fully Connected	1	15
	Real devices	Rigetti 16Q-Aspen	3.25	2.25
		IBM QX5	3.125	2.75

Table 1: The average graph distance and average vertex degree of topologies in figure 19. It shows that the topologies of real devices (i.e., Rigetti 16Q-Aspen and IBM QX5) have greater average graph distance and smaller average vertex degree than those of fictitious devices (i.e., 16Q-Square and 16Q-Fully Connected). Hence, the topologies of real devices are sparser than their fictitious counterparts.

Additionally, we expect that the heuristics we use for picking the row and column can still be improved. It is not unlikely that the performance of the heuristics heavily depend on the given topology and the given original circuit. This is a combinatorical search space that has not yet been explored within this paper. In stead of proposing a poor one-size-fits-all solution, we encourage the reader to think about what heuristics would work well for their use-case.

Moreover, when combining the reverse traversal (RT) with an A* algorithm [18] for optimizing the choice of row and column, the A* algorithm with RT performs marginally worse than RT on it's own on the 5-qubit devices. In appendix E, we describe these techniques and our results in more details. They seem to indicate that it is equally important for *PermRowCol* to have a good initial qubit mapping as having a good heuristic. Although for repeated iterations of the *PermRowCol* on multiple CNOT-slices of a circuit reduces our ability to pick a good initial allocation for each iteration and will make the choice of heuristics more important again.

In conclusion, we need to find a better method for deciding the dynamic allocation than the heuristic we have used for this paper. This is also why we have not yet compared the performance of *PermRowCol* against established quantum compilers such as Qiskit, TKET, and SABRE. We will discuss possible directions for improvements in the next section.

6 Future work

Algorithm *PermRowCol* is shown to be promising when combining it with Reverse Traversal (RT). To further optimize our synthesis method, we could either directly improve *PermRowCol* (section 6.1) or use *PermRowCol* within other algorithms such that it can be used to synthesize arbitrary quantum circuits (section 6.2).

6.1 Possible improvements of the PermRowCol algorithm

First of all, as discussed in section 5, we need to find a better heuristic for choosing the row and column to eliminate. We expect that the quality of the heuristics depends on the type of algorithm and structure of the topology. However, this still needs to be investigated.

Furthermore, it can still be an interesting research direction is to take into account the different gate error-rates. When building our Steiner-trees we can easily weight the edges by the quality of the gates that they represent. This might result in some interesting error-mitigation techniques. Moreover, due to the dynamic allocation in *PermRowCol*, we might also be able to redesign the algorithm such that it can also take the quality of single qubit gates that are executed after the CNOT circuit into account. This might result in certain qubits not being allocated to certain registers. These kinds of selection rules can easily be added to *PermRowCol* by changing the heuristic for choosing which column to eliminate.

Additionally, *PermRowCol* can still be improved by adding a blockwise elimination method [16]. In fact, the paper that introduced *RowCol* also introduced *size-block elimination* (SBE) that uses a similar strategy but with Steiner-trees and Gray codes. Eliminating blocks of rows and columns might not be very efficient on sparse graphs, but it might be interesting as an alternative to unconstrained Gaussian elimination tasks where a permutation matrix is also a valid solution (e.g. ZX-diagram extraction [3]).

6.2 Extensions to arbitrary quantum circuits

It is clear that CNOTs are not the only gates in arbitrary quantum circuits. So a good research direction is to extend the *PermRowCol* algorithm such that we can route and allocate any quantum circuit. There are two approaches we could take to extend our algorithm to arbitrary quantum circuits: (1) we can synthesize the full circuit from a flexible representation or (2) we can cut the circuit into pieces we can synthesize separately and glue those pieces back together. We will discuss these strategies by building from our *PermRowCol* algorithm to more general quantum circuits until we end up with the class of all quantum circuits.

Our proposed algorithm can only synthesize circuits consisting of only CNOTs. We can extend this to the class of CNOTs and R_z rotations. This class is also called *phase polynomials* and are described by the set of parities at which each R_z occurs as well as a parity matrix describing the output parities of the quantum circuit, this is also called the *sum-over-paths* notation [1]. The key here is that various Steiner-tree-based methods have been proposed for synthesizing the parities for each R_z gate [15, 9, 8, 21]. The remaining parity matrix can then be synthesized by *PermRowCol*. However, phase polynomials can efficiently be simulated on classical computers [1]. To extend phase polynomials to arbitrary quantum circuits, we need to add the H gate which these methods cannot synthesize. Nevertheless, we can cut our circuit into pieces at the H-gates, synthesize the phase polynomial between them, and glue all the pieces back together [8]. Here is where *PermRowCol* can make a difference because the dynamic reallocation of the qubits allows this algorithm to move the H-gates to different quantum registers.

Similarly, we can replace the *Steiner-Gauss* algorithm in [8] for synthesizing CNOTs, R_z, and NOT gates. By also including the NOT gate in the synthesis procedure, it allows to synthesize the phase polynomial before and after the NOT gate to be synthesized in unison. Rather than synthesizing the first phase polynomial, placing the $NOT = HR_z(\pi)H$, and then synthesizing the latter phase polynomial. [8] also proposes a method of generalizing to arbitrary quantum circuits called *slice-and-build*. Which works similarly to the cutting procedure described for extending the phase polynomials.

Next, we can extend the concept of phase polynomials to something called *Pauli exponentials* [7]. It could be considered a generalized version of the sum-over-paths notation. One could think of it as fol-

lows: instead of keeping track of a binary parity at which an R_z gate occurs, it keeps track of *Pauli-string*s that describes the parity at which a gate happens in terms of the Pauli's I, X, Y, Z rather than binary. Note that this is an oversimplification. However, if a Pauli exponential only contains Pauli strings with I and Z, we have a phase polynomial. Quantum circuits can be extracted from Pauli exponentials with an algorithm called *PauliSynth* [7]. An architecture-aware version of *PauliSynth* does not yet exist. However, the Pauli exponential procedure uses the synthesis of phase polynomials as a subroutine. Thus, it is worthwhile to research a method for making that algorithm into an architecture-aware version, possibly with Steiner-trees. Nevertheless, since *PauliSynth* uses Gaussian elimination, it can be worthwhile to replace it with *PermRowCol* even for fully connected topologies (see appendix B figure 18). Although *PauliSynth* was created for synthesizing particular quantum chemistry circuits, it can be argued that Pauli exponentials should be an important primitive for quantum computation [14]. Either way, the restriction of *PauliSynth* is that the Pauli-strings in the Pauli exponential need to be commuting. This is not always the case for arbitrary quantum circuits. Nevertheless, we can simply cut our circuit into sequences of commuting Pauli-strings and Clifford gates and run *PauliSynth* multiple times.

Lastly, we can use *PermRowCol* in the extraction of ZX-diagrams [3]. ZX-calculus is universal for quantum computation, so we can write any quantum circuit as a ZX-diagram. Then, we can make the diagram into a normal form from which to extract an optimized circuit. In the extraction procedure, they use Gaussian elimination, which can be replaced by *PermRowCol*. Even though the extraction procedure doesn't take any topologies into account, it can still be worthwhile to use *PermRowCol*. In ZX-calculus, only connectivity matters, so reordering the outputs of the extracted circuit is equivalent to bending wires and therefore free. Thus, it is better to end up with crossing wires than to extract CNOT gates. In appendix B, we show that *PermRowCol* can result in less CNOTs with respect to Gaussian elimination and *RowCol*. Therefore, *PermRowCol* is an algorithm with the potential to improve many quantum circuits.

Acknowledgements

The authors like to thank Jukka K. Nurminen, Douwe van Gijn, and Dustin Meijer for proof-reading.

References

[1] Matthew Amy, Parsiad Azimzadeh & Michele Mosca (2018): *On the controlled-NOT complexity of controlled-NOT–phase circuits*. Quantum Science and Technology 4(1).

[2] Matthew Amy & Vlad Gheorghiu (2020): *staq—A full-stack quantum processing toolkit*. Quantum Science and Technology 5(3). arXiv:1912.06070.

[3] Miriam Backens, Hector Miller-Bakewell, Giovanni de Felice, Leo Lobski & John van de Wetering (2021): *There and back again: A circuit extraction tale*. Quantum 5.

[4] Timothée Goubault de Brugière, Marc Baboulin, Benoît Valiron, Simon Martiel & Cyril Allouche (2020): *Quantum CNOT circuits synthesis for NISQ architectures using the syndrome decoding problem*. In: Quantum CNOT circuits synthesis for NISQ architectures using the syndrome decoding problem, Springer, pp. 189–205.

[5] Jarosław Byrka, Fabrizio Grandoni, Thomas Rothvoß & Laura Sanità (2013): *Steiner tree approximation via iterative randomized rounding*. Journal of the ACM (JACM) 60(1).

[6] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest & Clifford Stein (2022): *Introduction to algorithms*. MIT press.

[7] Alexander Cowtan, Will Simmons & Ross Duncan (2020): *A generic compilation strategy for the unitary coupled cluster ansatz*. arXiv preprint. arXiv:2007.10515.

[8] Vlad Gheorghiu, Sarah Meng Li, Michele Mosca & Priyanka Mukhopadhyay (2020): *Reducing the CNOT count for Clifford+ T circuits on NISQ architectures*. arXiv preprint. arXiv:2011.12191.

[9] Arianne Meijer-van de Griend & Ross Duncan (2020): *Architecture-aware synthesis of phase polynomials for NISQ devices*. arXiv preprint. arXiv:2004.06052.

[10] Frank K Hwang & Dana S Richards (1992): *Steiner tree problems*. Networks 22(1).

[11] Richard M Karp (1972): *Reducibility among combinatorial problems*. In: Complexity of computer computations, Springer, pp. 85–103.

[12] Aleks Kissinger & Arianne Meijer van de Griend (2020): *CNOT circuit extraction for topologically-constrained quantum memories*. Quantum Information and Computation 20(7-8). arXiv:1904.00633.

[13] Gushu Li, Yufei Ding & Yuan Xie (2019): *Tackling the qubit mapping problem for NISQ-era quantum devices*. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 1001–1014. arXiv:1809.02573.

[14] Gushu Li, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding & Yuan Xie (2022): *Paulihedral: a generalized block-wise compiler optimization framework for Quantum simulation kernels*. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 554–569.

[15] Beatrice Nash, Vlad Gheorghiu & Michele Mosca (2020): *Quantum circuit optimizations for NISQ architectures*. Quantum Science and Technology 5(2). arXiv:1904.01972.

[16] Ketan N Patel, Igor L Markov & John P Hayes (2004): *Optimal synthesis of linear reversible circuits*. Quantum Information & Computation 8(3), doi:10.26421.

[17] Gabriel Robins & Alexander Zelikovsky (2005): *Tighter bounds for graph Steiner tree approximation*. SIAM Journal on Discrete Mathematics 19(1).

[18] Stuart Russell & Peter Norvig (2010): *Artificial Intelligence: A Modern Approach*, 3 edition. Prentice Hall.

[19] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington & Ross Duncan (2020): *t|ket>: a retargetable compiler for NISQ devices*. Quantum Science and Technology 6(1). arXiv:2003.10611.

[20] Roberto Stassi, Mauro Cirio & Franco Nori (2020): *Scalable quantum computer with superconducting circuits in the ultrastrong coupling regime*. npj Quantum Information 6(1). arXiv:1910.14478.

[21] Vivien Vandaele, Simon Martiel & Timothée Goubault de Brugière (2022): *Phase polynomials synthesis algorithms for NISQ architectures and beyond*. Quantum Science and Technology. arXiv:2104.00934.

[22] Robert Wille, Rod Van Meter & Yehuda Naveh (2019): *IBM's Qiskit tool chain: Working with and developing for real quantum computers*. In: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, pp. 1234–1240, doi:10.23919/DATE.2019.8715261.

[23] Bujiao Wu, Xiaoyu He, Shuai Yang, Lifu Shou, Guojing Tian, Jialin Zhang & Xiaoming Sun (2021): *Optimization of CNOT circuits under topological constraints*. arXiv preprint. arXiv:1910.14478.

[24] Xiangzhen Zhou, Yuan Feng & Sanjiang Li (2022): *Quantum Circuit Transformation: A Monte Carlo Tree Search Framework*. ACM Transactions on Design Automation of Electronic Systems (TODAES). arXiv:2008.09331.

A Subroutines

For reference, this section contains the pseudo-code of all subroutines used in algorithm 1.

Algorithm 2: ChooseRow: Subroutine for choosing which row to eliminate.

Input : Parity matrix M, vertices Vs to choose from that correspond to rows in M
Output: The *row* to eliminate.
`/* Pick the row with the least 1s in` M `*/`
$row \leftarrow argmin_{v \in Vs}(\sum M[v])$;
return *row*

Algorithm 3: ChooseColumn: Subroutine for choosing which column to eliminate.

Input : Parity matrix M, columns $ColsToEliminate$ to choose from that correspond to columns in M, chosen row *row*
Output: The *col* to eliminate.
`/* Pick the column with the least 1s in` M `if the column has an 1 on` *row* `*/`
$row \leftarrow argmin_{c \in ColsToEliminate}(\sum M[:][c]$ if $M[row][c] = 1$ else $|ColsToEliminate|)$;
return *row*

Algorithm 4: EliminateColumn: Subroutine for eliminating a column. Here, $SteinerTree$ is a routine which creates a Steiner tree over graph G with root *root* and nodes $Nodes$. $PostOrderTraverseEdges$ traverses the given tree in post-order and returns the edges rather than the vertices.

Input : Parity matrix M, graph G representing the connectivity constraints, *root* vertex, $Nodes$ to build the Steiner tree over
Output: List of CNOTs C to add to the circuit.
$C \leftarrow []$;
$Tree \leftarrow SteinerTree(G, root, Nodes)$;
for $edge \in PostOrderTraverseEdges(Tree)$ **do**
  if $M[edge[0]][col] = 0$ **then**
    $C.add(CNOT(edge[0], edge[1]))$; `/* Make Steiner nodes into 1s */`
    $M[edge[0]] \leftarrow M[edge[0]] + M[edge[1]] \bmod 2$;
  end
end
for $edge \in PostOrderTraverseEdges(Tree)$ **do**
  $C.add(CNOT(edge[1], edge[0]))$; `/* Make the column into identity */`
  $M[edge[1]] \leftarrow M[edge[0]] + M[edge[1]] \bmod 2$;
end
return C

Algorithm 5: EliminateRow: Subroutine for eliminating a row. Here, $SteinerTree$ is a routine which creates a Steiner tree over graph G with root $root$ and nodes $Nodes$. $PreOrderTraverseEdges$ traverses the given tree in pre-order and returns the edges rather than the vertices. Similarly, $PostOrderTraverseEdges$ traverses the tree in post-order and returns the edges when traversed for the second time.

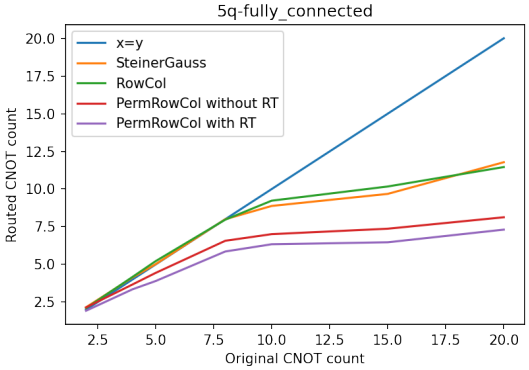
Input : Parity matrix M, graph G representing the connectivity constraints, $root$ vertex, $Nodes$ to build the Steiner tree over
Output: List of CNOTs C to add to the circuit.
$C \leftarrow []$;
$Tree \leftarrow SteinerTree(G, root, Nodes)$;
for $edge \in PreOrderTraverseEdges(Tree)$ **do**
  if $edge[1] \notin Nodes$ **then**
    $C.add(CNOT(edge[0], edge[1]))$ $M[edge[0]] \leftarrow M[edge[0]] + M[edge[1]] \bmod 2$;
  end
end
for $edge \in PostOrderTraverseEdges(Tree)$ **do**
  $C.add(CNOT(edge[0], edge[1]))$ $M[edge[0]] \leftarrow M[edge[0]] + M[edge[1]] \bmod 2$;
end
return C

B Unconstrained performance

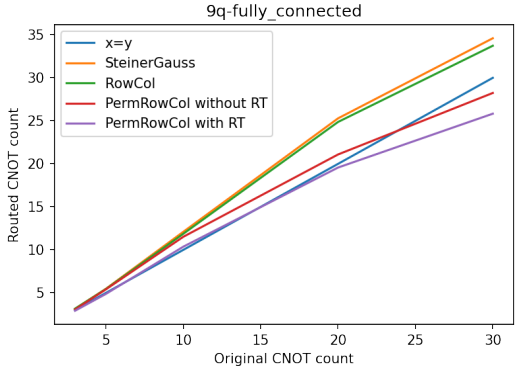
As additional information, figure 18 compares the algorithm performance in the unconstrained case, i.e. a fully connected topology. Although this use case does not include the routing of the CNOTs, there are two use cases for which this is interesting. Both of these cases need to allow reallocation of qubits to make use of *PermRowCol*. Restricting *PermRowCol* to a fixed allocation is equivalent to the *RowCol* algorithm.

First of all, we can use *PermRowCol* without topological restrictions in case we have a circuit with many CNOTs (with respect to the number of qubits) to optimize the number CNOTs.

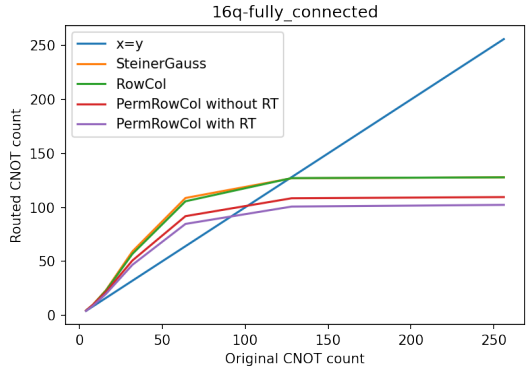
Secondly, *PermRowCol* can be used in cases when the CNOT circuit is not known beforehand and needs to be synthesized from a given parity matrix. For example, this can be done as part of the *GraySynth* [1] algorithm that is used in the *PauliSynth* [7] algorithm for generating UCCSD circuits in quantum chemistry. Alternatively, it can be used in place of Gaussian elimination as part of ZX-diagram extraction [3]. As we have already discussed in section 6.2.



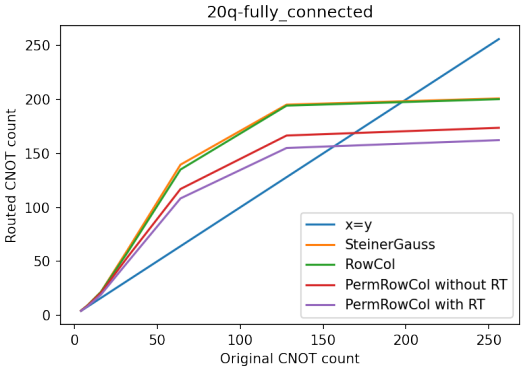
(a) Algorithm performance for the 5-qubit fully connected graph.



(b) Algorithm performance for the 9-qubit fully connected graph.



(c) Algorithm performance for the 16-qubit fully connected graph.



(d) Algorithm performance for the 20-qubit fully connected graph.

Figure 18: These figures show the number of CNOTs generated by *Steiner-Gauss* [12] (orange), *RowCol* [23] (green), and *PermRowCol* (proposed, red) for the unconstrained case, i.e. a complete graph of 5, 9, 16, and 20 qubits where every pair of distinct vertices is connected by a unique edge. The blue $x = y$-line can be used to infer the CNOT overhead.

C Topologies of real devices

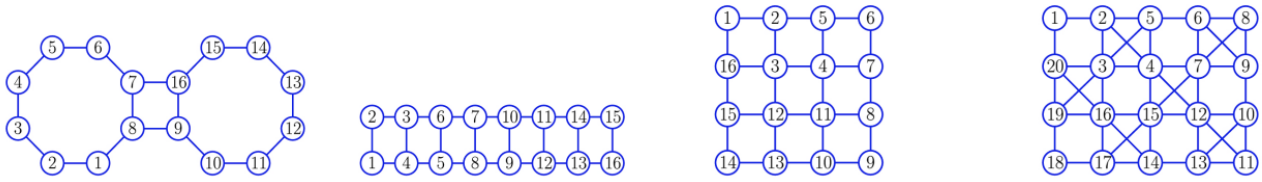
We show in figure 19 the different topologies for the real quantum computers that we used for the connectivity constraints in our experiments.

D Results in table form

For completeness, we show the results from figure 16 and 17 in the form of table 2.

CNOT	**Topology**	***Steiner-Gauss***	***RowCol***	***PermRowCol***	***PermRowCol+RT***
3	9q-square	11.96 (298.67 %)	12.26 (308.67 %)	12.17 (305.67 %)	4.74 (58.00 %)
5	9q-square	18.23 (264.60 %)	18.74 (274.80 %)	16.33 (226.60 %)	7.48 (49.60 %)
10	9q-square	33.69 (236.90 %)	32.67 (226.70 %)	28.17 (181.70 %)	14.22 (42.20 %)
20	9q-square	49.05 (145.25 %)	46.82 (134.10 %)	40.01 (100.05 %)	24.47 (22.35 %)
30	9q-square	55.58 (85.27 %)	53.52 (78.40 %)	45.75 (52.50 %)	31.23 (4.10 %)
4	16q-square	29.90 (647.50 %)	29.02 (625.50 %)	33.86 (746.50 %)	7.21 (80.25 %)
8	16q-square	55.14 (589.25 %)	55.70 (596.25 %)	58.49 (631.13 %)	15.96 (99.50 %)
16	16q-square	97.85 (511.56 %)	92.24 (476.50 %)	87.95 (449.69 %)	34.34 (114.63 %)
32	16q-square	151.54 (373.56 %)	141.57 (342.41 %)	133.99 (318.72 %)	81.68 (155.25 %)
64	16q-square	189.09 (195.45 %)	175.74 (174.59 %)	184.84 (188.81 %)	141.75 (121.48 %)
128	16q-square	200.40 (56.56 %)	189.32 (47.91 %)	204.89 (60.07 %)	165.97 (29.66 %)
256	16q-square	201.95 (−21.11 %)	190.83(−25.46 %)	205.16(−19.86 %)	167.55(−34.55 %)
4	rigetti 16q aspen	55.59 (1289.75 %)	50.00 (1150.00 %)	51.77 (1194.25 %)	14.17 (254.25 %)
8	rigetti 16q aspen	100.85(1160.63 %)	88.78 (1009.75 %)	90.12 (1026.50 %)	30.13 (276.63 %)
16	rigetti 16q aspen	155.25 (870.31 %)	138.99 (768.69 %)	132.58 (728.63 %)	54.15 (238.44 %)
32	rigetti 16q aspen	223.03 (596.97 %)	199.41 (523.16 %)	174.21 (444.41 %)	106.04 (231.38 %)
64	rigetti 16q aspen	259.77 (305.89 %)	240.51 (275.80 %)	229.71 (258.92 %)	178.55 (178.98 %)
128	rigetti 16q aspen	270.64 (111.44 %)	252.33 (97.13 %)	252.95 (97.62 %)	209.31 (63.52 %)
256	rigetti 16q aspen	270.07 (5.50 %)	255.06 (−0.37 %)	252.69 (−1.29 %)	209.52(−18.16 %)
4	ibm qx5	37.87 (846.75 %)	37.96 (849.00 %)	40.54 (913.50 %)	9.62 (140.50 %)
8	ibm qx5	72.34 (804.25 %)	71.43 (792.88 %)	66.81 (735.13 %)	20.62 (157.75 %)
16	ibm qx5	121.33 (658.31 %)	115.63 (622.69 %)	104.60 (553.75 %)	40.31 (151.94 %)
32	ibm qx5	184.57 (476.78 %)	174.74 (446.06 %)	154.56 (383.00 %)	91.17 (184.91 %)
64	ibm qx5	227.48 (255.44 %)	218.93 (242.08 %)	211.51 (230.48 %)	159.43 (149.11 %)
128	ibm qx5	242.96 (89.81 %)	238.07 (85.99 %)	229.47 (79.27 %)	189.13 (47.76 %)
256	ibm qx5	244.43 (−4.52 %)	238.25 (−6.93 %)	233.83 (−8.66 %)	191.73(−25.11 %)
4	ibm q20 tokyo	24.04 (501.00 %)	23.14 (478.50 %)	28.88 (622.00 %)	6.71 (67.75 %)
8	ibm q20 tokyo	50.58 (532.25 %)	49.09 (513.63 %)	54.29 (578.63 %)	14.72 (84.00 %)
16	ibm q20 tokyo	99.70 (523.13 %)	94.17 (488.56 %)	95.66 (497.88 %)	30.08 (88.00 %)
32	ibm q20 tokyo	177.58 (454.94 %)	163.22 (410.06 %)	156.28 (388.38 %)	82.09 (156.53 %)
64	ibm q20 tokyo	249.51 (289.86 %)	235.23 (267.55 %)	238.52 (272.69 %)	183.99 (187.48 %)
128	ibm q20 tokyo	281.34 (119.80 %)	269.25 (110.35 %)	289.53 (126.20 %)	245.02 (91.42 %)
256	ibm q20 tokyo	288.01 (12.50 %)	273.23 (6.73 %)	300.92 (17.55 %)	256.48 (0.19 %)

Table 2: This table shows the performance of *Steiner-Gauss* [12, 15], *RowCol* [23], *PermRowCol* (proposed), and *PermRowCol* with *Reverse Traversal strategy* (proposed) for different topologies. The shown numbers represent the average CNOT count over 100 circuits and the average CNOT overhead with respect to the original circuit in brackets behind it.



(a) Rigetti 16Q-Aspen (b) IBM QX5 (c) 16q-Square (d) IBM QX20 Tokyo

Figure 19: Topologies of existing quantum computers that we use for testing our algorithm. Images are taken from [4].

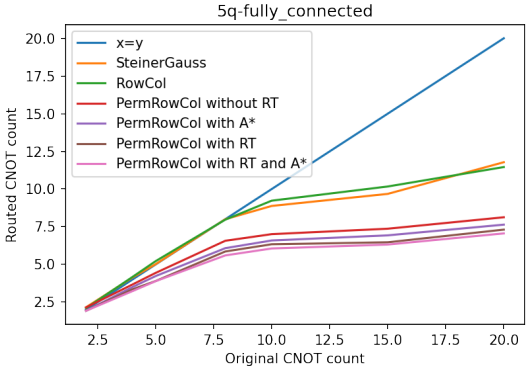
E A* results

In this appendix, we show the results for using the A* algorithm [18] for choosing the row and column in the iterations of *PermRowCol*. We do this for different 5-qubit topologies because the overhead of A* becomes too long for larger topologies.

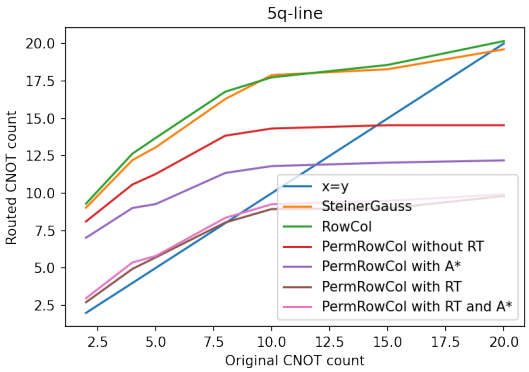
We added the A* algorithm into *PermRowCol* using a priority queue. Initially, we push the original problem into the queue with priority equal to 0. Then, while the queue is not empty, we remove an instance from the queue, reduce the matrix for each combination of chosen row and column and push the resulting smaller problem to the queue with the size of the circuit until now as priority. We continue this process until a solution has been found. By construction, the resulting solution will be the smallest circuit that can be found with *PermRowCol* regardless of the choice in ChooseRow and ChooseColumn heuristics, but the complexity is exponential in the number of qubits.

To reduce the runtime of the algorithm, we restrict the number of new problem instances created at each iteration by a parameter called *choiceWidth*. Instead of expanding each possible combination of row and column, we only expand the top *choiceWidth* options where we rank the options using the original ChooseRow and ChooseColumn heuristics. Where the ChooseRow has priority. Additionally, we limit the size of the queue such that problem instances low in the queue are removed from memory since they probably don't need to be expanded before a solution is found. For these results, we used *choiceWidth*= 4 and *max_size*= 10. Resulting in an algorithm with time complexity $O(choiceWidth^{O(PermRowCol)} = O(4^{n^4})$.

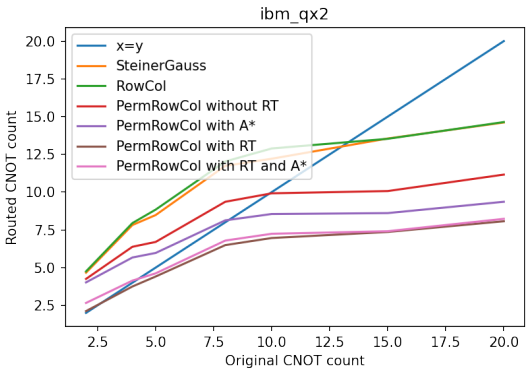
The results are shown in figure 20 where we see that A* does not perform better than than the Reverse Traversal (RT) strategy, even when combining the two algorithms.



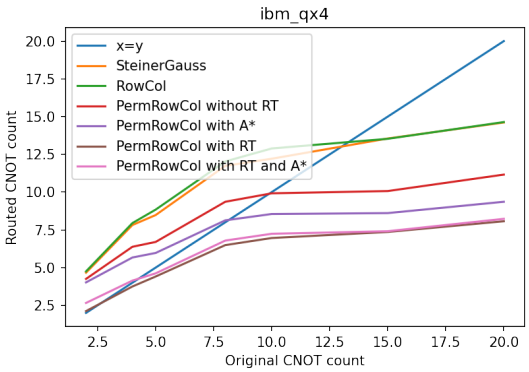
(a) Algorithm performance for the 5-qubit fully connected graph.



(b) Algorithm performance for the 5-qubit line graph.



(c) Algorithm performance for the 5-qubit IBM QX2 device.



(d) Algorithm performance for the 5-qubit IBM QX4 device.

Figure 20: These figures show the number of CNOTs generated by *Steiner-Gauss* [12] (orange), *RowCol* [23] (green), and different variants of *PermRowCol* (proposed) on different 5-qubit topologies. The different variants of *PermRowCol* are the original (red), with A* algorithm (purple), with Reverse Traversal (RT) (brown), and with both A* and RT (pink). The blue $x = y$-line can be used to infer the CNOT overhead.

F Example execution of *PermRowCol*

In the section, we execute algorithm *PermRowCol* with inputs in figure G. We start with a parity matrix $\mathbf{A}$ to synthesize over the topology graph G, and reduce the problem to eliminating $\mathbf{A}$ to a permutation matrix. The labelling of vertices in G corresponds to the numbering of rows in $\mathbf{A}$.

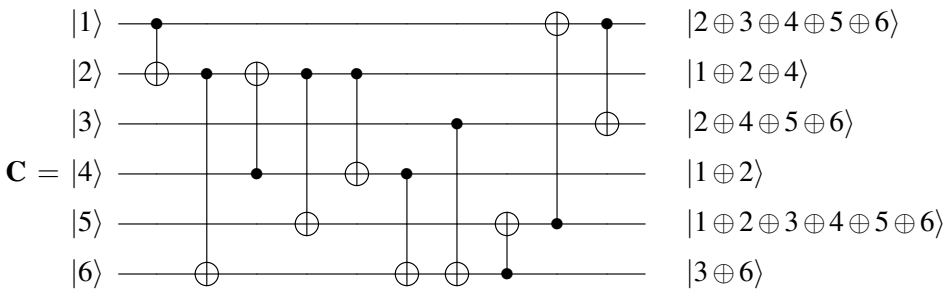


Figure (a): A CNOT circuit composed of 6 qubits.

$$\mathbf{A} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Figure (b): The parity matrix of (a).

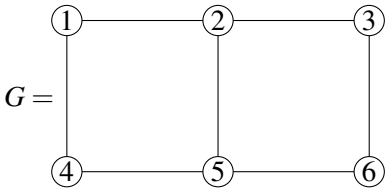


Figure (c): The 6-qubit square grid G.

Figure G: The CNOT circuit $\mathbf{C}$ in figure (a) can be exactly represented by the 6×6 parity matrix $\mathbf{A}$ in figure (b). Under the constrained topology G in figure (c), the Steiner-tree based algorithm *PermRowCol* accounts for G and re-synthesizes $\mathbf{C}$.

F.1 Notations

We declare notations that will be used in appendices F.2 and F.3.

According to algorithm *PermRowCol*, the chosen logical qubit corresponds to a row in $\mathbf{A}$. Its index is denoted by r. The chosen new physical register for r corresponds to a column in $\mathbf{A}$. Its index is denoted by c. V_s is a set of non-cutting vertices of G under which $\mathbf{A}$ is synthesized. S is the set of terminal nodes corresponding the indices of all non-zero entries in a row or column. $R(i, j)$ denotes a row operation on $\mathbf{A}$ such that row i is added to row j, while row i remains unchanged.

The output qubit allocation table 3 keeps track of the row and column being selected at each elimination step.

Logical qubit/r	1	2	3	4	5	6
Physical register/c						

Table 3: The output qubit allocation table

F.2 Parity matrix for a CNOT circuit

Consider a CNOT circuit composed of n qubits, where CNOT(c,t) has control c and target t. Based on the specification in section 2.1, CNOT(c,t) corresponds to a row operation $R(t,c)$ such that row t is added to row c, while row t remains unchanged.

In addition to the parity matrix defined in this paper, we note an alternative characterization for CNOT circuits [16, 15, 8]. Let's call it as the *alt-parity matrix* for a CNOT circuit. Similar to the parity matrix characterization, a square matrix is constructed to record the output parities of the circuit. Each row represents a parity, and each column represents the input qubit. By construction, given a CNOT circuit, its alt-parity matrix is the transpose of its parity matrix. Accordingly, CNOT(c,t) corresponds to a row operation $R(c,t)$ such that row c is added to row t, while row c remains unchanged. However, the synthesis procedure constructs the circuit in reverse.

For example, in figure F, we compare the differences between the two notations and the corresponding CNOT synthesis algorithms. In figure F.c, we see that the column-wise representation of the parities (figure F.b) results in taking row operations that correspond to the CNOTs in order, while adding the target to the control: $R(2,1)R(3,2) \sim CNOT(1,2)CNOT(2,3)$. Alternatively, in figure F.e, we see that the row-wise representation of the parities (figure F.d) results in taking row operations that correspond to the CNOTs in reverse order, while adding the control to the target: $R(2,3)R(1,2) \sim CNOT(1,2)CNOT(2,3)$.

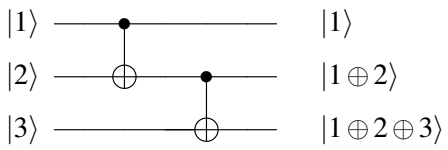


Figure (a): A CNOT circuit composed of 3 qubits and 2 CNOT gates. The labels of input qubits are denoted on the left of the circuit. Accordingly, the output parities are denoted on the right of the circuit.

$$\mathbf{A} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1' & 2' & 3' \end{array} \\ \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure (b): The parity matrix of (a) where each column represents an output parity and each row represents an input qubit.

$$\mathbf{A} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1' & 2' & 3' \end{array} \\ \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \xrightarrow{R(2,1)} \mathbf{A}' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1' & 2' & 3' \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \xrightarrow{R(3,2)} I = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1' & 2' & 3' \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure (c): Given the parity matrix in (b), CNOT$(1,2)$ corresponds to the row operation $R(2,1)$ and CNOT$(2,3)$ corresponds to the row operation $R(3,2)$.

$$\mathbf{B} = \begin{array}{c} \\ 1' \\ 2' \\ 3' \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \end{array}$$

Figure (d): The alt-parity matrix of (a) where each row represents an output parity and each column represents an input qubit.

$$\mathbf{B} = \begin{array}{c} \\ 1' \\ 2' \\ 3' \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \end{array} \xrightarrow{R(2,3)} \mathbf{B}' = \begin{array}{c} \\ 1' \\ 2' \\ 3' \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \xrightarrow{R(1,2)} I = \begin{array}{c} \\ 1' \\ 2' \\ 3' \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure (e): Given the parity matrix in (b), CNOT$(2,3)$ corresponds to the row operation $R(2,3)$ and CNOT$(1,2)$ corresponds to the row operation $R(1,2)$.

Figure F: Figure(a) demonstrates a circuit composed of CNOT$(1,2)$ and CNOT$(2,3)$, which results in some output parities. They are described by the parity matrix $\mathbf{A}$ in (b), or equivalently, by the alt-parity matrix $\mathbf{B}$ in (e). $\mathbf{B}^\top = \mathbf{A}$.

F.3 *PermRowCol* walkthrough

Elimination step 1 Before the first elimination step, the parity matrix $\mathbf{A}$ and the constrained topology G are shown in figure G.

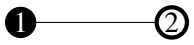
Choose the row and column to eliminate: The set of non-cutting vertices of G is $V_s = \{1,2,3,4,5,6\}$. Then $r = 1$ and $c = 4$ since

	$1'$	$2'$	$3'$	$4'$	$5'$	$6'$	Sum	Row
1	0	1	0	1	1	0	3	✓
2	1	1	1	1	1	0	5	
3	1	0	0	0	1	1	3	
4	1	1	1	0	1	0	4	
5	1	0	1	0	1	0	3	
6	1	0	1	0	1	1	4	

$\Rightarrow$

	$1'$	$2'$	$3'$	$4'$	$5'$	$6'$
1	0	**1**	0	**1**	**1**	0
2	1	1	1	1	1	0
3	1	0	0	0	1	1
4	1	1	1	0	1	0
5	1	0	1	0	1	0
6	1	0	1	0	1	1
Sum	\	3	\	2	6	\
Column				✓		

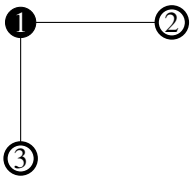
Eliminate the chosen row and column: We start by eliminating column $4'$ to $e_1^\intercal$, then $S = \{1,2\}$. The Steiner tree $\mathscr{T}_{1,S}$ has root 1 and a set of terminals S:

$\mathscr{T}_{1,S} =$ 

Thus, algorithm *PermRowCol* assigns CNOT$(2,1)$. It follows that

$$\mathbf{A} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array} \xrightarrow{R(1,2)} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array}. \tag{1}$$

Next, we eliminate row 1 to e_4. From solving the system of linear equations while emitting column $4'$, row 1 is formed by rows 2 and 4. Thus $S = \{1,2,4\}$. The Steiner tree $\mathscr{T}_{1,S}$ has root 1 and a set of terminals S:

$\mathscr{T}_{1,S} =$ 

Thus, algorithm *PermRowCol* assigns CNOT(1,2)CNOT(1,4). It follows that

$$(1) = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array} \xrightarrow{R(2,1)} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array} \tag{2}$$

$$\xrightarrow{R(4,1)} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array} \tag{3}$$

Update the output qubit allocation: The output qubit allocation after eliminating row 1 and column $4'$ is updated in table 4.

Logical qubit/r	1	2	3	4	5	6
Physical register/c	4					

Table 4: Logical qubit 1 is stored in the physical register 4.

Elimination step 2 After the first elimination step, the parity matrix $\mathbf{A}'$ and the constrained topology G' are shown in figure H.

$$\mathbf{A}' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Figure (a): The updated parity matrix $\mathbf{A}'$.

$G' =$

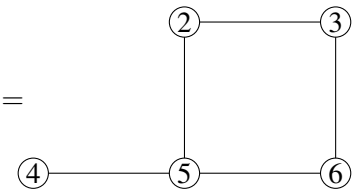


Figure (b): The 5-qubit grid G'.

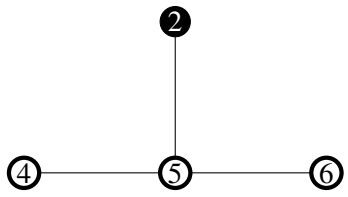
Figure H: Under the constrained topology G' in figure (b), algorithm *PermRowCol* further eliminates the parity matrix $\mathbf{A}'$ in figure (a).

Choose the row and column to eliminate: The set of non-cutting vertices of G' is $V_s = \{2,3,4,6\}$. Then $r = 2$ and $c = 3$ since

$$\mathbf{A}' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array} \begin{array}{c} \text{Sum} \\ \backslash \\ 2 \\ 3 \\ 4 \\ \backslash \\ 4 \end{array} \begin{array}{c} \text{Row} \\ \\ \checkmark \\ \\ \\ \\ \end{array} \Rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ \text{Sum} \\ \text{Column} \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \\ \begin{array}{cccccc} 5 & \backslash & 4 & \backslash & \backslash & \backslash \end{array} \\ \begin{array}{cccccc} & & \checkmark & & & \end{array} \end{array}$$

Eliminate the chosen row and column: We start by eliminating column $3'$ to $e_2^\intercal$, then $S = \{2,4,5,6\}$. The Steiner tree $\mathscr{T}_{2,S}$ has root 2 and a set of terminals S:

$\mathscr{T}_{2,S} =$



Thus, algorithm *PermRowCol* assigns CNOT(4,5)CNOT(6,5)CNOT(5,2). It follows that

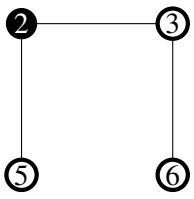
$$\mathbf{A}' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array} \xrightarrow{R(5,4)} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array} \tag{4}$$

$$\xrightarrow{R(5,6)} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{5}$$

$$\xrightarrow{R(2,5)} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}. \tag{6}$$

Next, we eliminate row 2 to e_3. From solving the system of linear equations while emitting columns $3'$ and $4'$, row 2 is formed by rows 3, 5, and 6. Thus $S = \{2,3,5,6\}$. The Steiner tree $\mathscr{T}_{2,S}$ has root 2 and a set of terminals S:

$\mathscr{T}_{2,S} =$



Thus, algorithm *PermRowCol* assigns CNOT(3,6)CNOT(2,3)CNOT(2,5). It follows that

$$(6) = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \xrightarrow{R(6,3)} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{7}$$

$$\xrightarrow{R(3,2)} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{8}$$

$$\xrightarrow{R(5,2)} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{9}$$

Update the output qubit allocation: The output qubit allocation after eliminating row 2 and column $3'$ is updated in table 5.

Logical qubit/r	1	2	3	4	5	6
Physical register/c	4	3				

Table 5: Logical qubit 2 is stored in the physical register 3.

Elimination step 3 After the second elimination step, the parity matrix $\mathbf{A}''$ and the constrained topology G'' are shown in figure I.

$$\mathbf{A}'' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure (a): The updated parity matrix $\mathbf{A}''$.

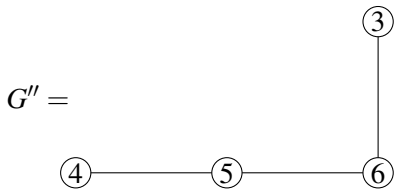


Figure (b): The 4-qubit line G''.

Figure I: Under the constrained topology G'' in figure (b), algorithm *PermRowCol* further eliminates the parity matrix $\mathbf{A}''$ in figure (a).

Choose the row and column to eliminate: The set of non-cutting vertices of G'' is $V_s = \{3,4,5,6\}$. Then $r = 4$ and $c = 2$ since

$$\mathbf{A}'' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{c} \begin{array}{c} \text{Sum} \\ \backslash \\ \backslash \\ 2 \\ 1 \\ 1 \\ 1 \end{array} \\ \begin{array}{c} \text{Row} \\ \\ \\ \\ \checkmark \\ \\ \\ \end{array} \end{array} \Rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ \text{Sum} \\ \text{Column} \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ \backslash & 1 & \backslash & \backslash & \backslash & \backslash \\ & \checkmark & & & & \end{pmatrix} \end{array}$$

Eliminate the chosen row and column: In fact, column $2'$ and row 4 is $e_4^\intercal$ and e_2 respectively, this step is complete.

Update the output qubit allocation: The output qubit allocation after eliminating row 4 and column $2'$ is updated in table 6.

Logical qubit/r	1	2	3	4	5	6
Physical register/c	4	3		2		

Table 6: Logical qubit 4 is stored in the physical register 2.

Elimination step 4 After the third elimination step, the parity matrix $\mathbf{A}'''$ and the constrained topology G''' are shown in figure J.

$$\mathbf{A}''' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure (a): The updated parity matrix $\mathbf{A}'''$.

$G''' =$

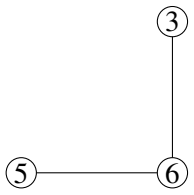


Figure (b): The 3-qubit line G'''.

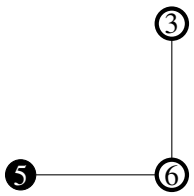
Figure J: Under the constrained topology G''' in figure (b), algorithm *PermRowCol* further eliminates the parity matrix $\mathbf{A}'''$ in figure (a).

Choose the row and column to eliminate: The set of non-cutting vertices of G''' is $V_s = \{3,5\}$. Then $r = c = 5$ since

$$\mathbf{A}''' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{c} \text{Sum} \\ \backslash \\ \backslash \\ 2 \\ \backslash \\ 1 \\ \backslash \end{array} \begin{array}{c} \text{Row} \\ \\ \\ \\ \\ \checkmark \\ \end{array} \quad \Rightarrow \quad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ \text{Sum} \\ \text{Column} \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ \backslash & \backslash & \backslash & \backslash & 2 & \backslash \end{pmatrix} \\ \\ \\ \\ \\ \\ \\ \checkmark \end{array}$$

Eliminate the chosen row and column: We start by eliminating column $5'$ to $e_5^\intercal$, then $S = \{3,5\}$. The Steiner tree $\mathscr{T}_{5,S}$ has root 5 and a set of terminals S:

$\mathscr{T}_{5,S} =$



Thus, algorithm *PermRowCol* assigns CNOT(6,3)CNOT(3,6)CNOT(6,5). It follows that

$$\mathbf{A}''' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \xrightarrow{R(3,6)} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array} \tag{10}$$

$$\xrightarrow{R(6,3)} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array} \tag{11}$$

$$\xrightarrow{R(5,6)} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}. \tag{12}$$

Since row 5 is in fact e_5, this step is complete.

Update the output qubit allocation: The output qubit allocation after eliminating row 5 and column $5'$ is updated in table 7.

Logical qubit/r	1	2	3	4	5	6
Physical register/c	4	3		2	5	

Table 7: Logical qubit 5 is stored in the physical register 5.

Elimination step 5 After the fourth elimination step, the parity matrix $\mathbf{A}''''$ and the constrained topology G'''' are shown in figure K.

$$\mathbf{A}'''' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure (a): The updated parity matrix $\mathbf{A}''''$.

$G'''' =$ 

Figure (b): The 2-qubit line G''''.

Figure K: Under the constrained topology G'''' in figure (b), algorithm *PermRowCol* further eliminates the parity matrix $\mathbf{A}''''$ in figure (a) to a permutation matrix.

Choose the row and column to eliminate: The set of non-cutting vertices of G'''' is $V_s = \{3,6\}$. Then $r = 3$ and $c = 6$ since

$$\mathbf{A}'''' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{c} \text{Sum} \\ \backslash \\ \backslash \\ 1 \\ \backslash \\ \backslash \\ 2 \end{array} \quad \begin{array}{c} \text{Row} \\ \\ \\ \checkmark \end{array} \quad \Rightarrow \quad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ \text{Sum} \\ \text{Column} \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ \backslash & \backslash & \backslash & \backslash & \backslash & 1 \\ & & & & & \checkmark \end{pmatrix} \end{array}$$

Eliminate the chosen row and column: We start by eliminating column $6'$ to $e_3^\intercal$, then $S = \{3,6\}$. The Steiner tree $\mathscr{T}_{3,S}$ has root 3 and a set of terminals S:

$\mathscr{T}_{3,S} =$ 

Thus, algorithm *PermRowCol* assigns CNOT$(6,3)$. It follows that

$$\mathbf{A}'''' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \xrightarrow{R(3,6)} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}. \tag{13}$$

Since row 3 is in fact e_6, this step is complete.

Update the output qubit allocation: The output qubit allocation after eliminating row 3 and column $6'$ is updated in table 8.

Logical qubit/r	1	2	3	4	5	6
Physical register/c	4	3	6	2	5	

Table 8: Logical qubit 3 is stored in the physical register 6.

F.4 Output from *PermRowCol*

After the fifth elimination step, *PermRowCol* terminates as there is precisely one vertex left in the constrained topology. The parity matrix $\mathbf{A}$ is reduced to a permutation P. Accordingly, the final output qubit allocation is shown in table 9.

$$P = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Logical qubit/r	1	2	3	4	5	6
Physical register/c	4	3	6	2	5	1

Table 9: Qubit allocation after resynthesizing the CNOT circuit $\mathbf{C}$ under the constrained topology G.

F.5 Examination of the output validity

By concatenating CNOTs produced from each elimination step, the re-synthesized circuit $\mathbf{C}'$ and the corresponding parity matrix $\mathbf{M}$ is shown in figure L. With the input parity matrix $\mathbf{A}$ and the permutation matrix P output by *PermRowCol*, we have $\mathbf{M}P = \mathbf{A}$. This corresponds to the qubit allocation after re-synthesizing the CNOT circuit $\mathbf{C}$ over G, as described by table 9. Hence, our re-synthesized circuit $\mathbf{C}'$ is equivalent to circuit $\mathbf{C}$ up to column permutation specified by P.

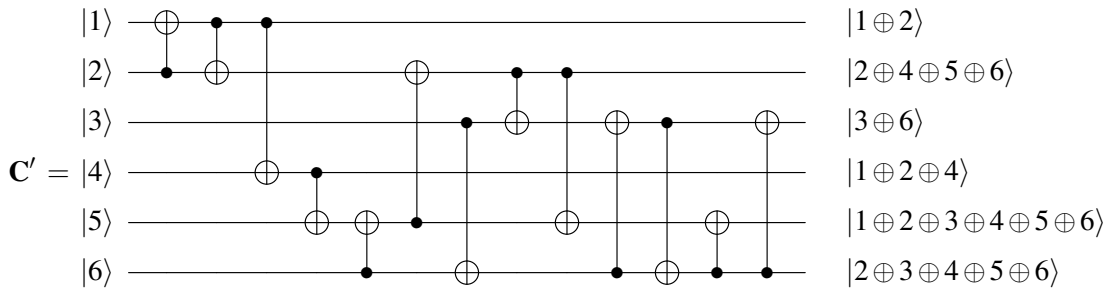


Figure (a): The re-synthesized CNOT circuit $\mathbf{C}'$ after running *PermRowCol* with input parity matrix and constrained topology defined in figure G.

$$\mathbf{M} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Figure (b): The parity matrix of (a).

Figure L: The CNOT circuit $\mathbf{C}'$ in figure (a) can be exactly represented by the 6×6 parity matrix $\mathbf{M}$ in figure (b).