TUM SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

On the Architecture-Aware Synthesis of Pauli Polynomials

David Winderl

TUM SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

On the Architecture-Aware Synthesis of Pauli Polynomials

...

Author:	David Winderl
Supervisor:	Prof. Mendl Christian
Advisor:	Huang Qunsheng
Submission Date:	June 26, 2024

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, June 26, 2024 David Winderl

Acknowledgments

At first I want to thank Keefe (Qunsheng Huang) for the amazing supervision combined with the trustful and lighthearted yet dedicated collaboration from Tokyo to Munich including the one or other ”three A.M.” message, if necessary.
Also, I want to thank Arianne Meijer van de Griend, Richie Yeung and Stefano Gogioso for the collaboration in the PauliOpt Libary and the often long, yet entertaining and from my point of view fruitful discussions. Special thanks to Arianne for pushing me into the right directions at the one or other point.
One special shoutout to Simon Borowski for having to bear my self conversations and providing the apropriate ”yes” at points, where we both knew we had no clue of what we where talking about.
And at last I want to thank Arianne Meijer van de Griend, Richie Yeung, Stefano Gogioso, Simon Borowski and Andreas Dachsberger for hopefully finding all of the typos and the amazing feedback I have received for this work. Thank you guys!

Abstract

The synthesis of quantum circuits from so-called Pauli-Polynomials in the ZX Calculus facilitates quantum circuit optimization. In this work, we will develop three algorithms that can perform this task in an architecture-aware fashion. On top of that, we will develop an architecture-aware synthesis method for synthesizing Clifford-Tableaus. Our results show an increase in the Reduction of CNOT-Gates for larger circuits for both Clifford circuits and circuits described by Pauli-Polynomials, which outlines the capabilities of architecture-aware synthesis in the circuit optimization realm.

Contents

1 Introduction

The field of quantum computing is currently in the noisy intermediate-scale quantum (NISQ) era. This era is distinguished by the presence of qubits that are affected by noise and present limited coherence times, a measure of the time until the information held by a quantum state decays and useful information is lost [43]. Furthermore, the NISQ-era and possibly quantum computation beyond is characterized by restricted connectivity in quantum computing architecture, meaning that not all qubits can interact directly. The restricted architecture necessitates certain conditions on the interactions permitted within quantum circuits designed to perform computations by manipulating qubit states. These factors have led to a profound emphasis on the depth and gate-reduction of quantum circuits to make them compatible with existing NISQ devices [38, 45, 31]. However, the restricted connectivity of these devices often imposes an additional routing step in the optimization process to ensure that the placement of interactions within the quantum circuit aligns with the constraints of the given architecture. Examples of this include transpilation and synthesis tools like tket [45] or BQSKit [56]. Conventional optimization algorithms typically handle this situation through an iterative process incorporating such a routing step. However, recent research suggests that a more integrated approach could yield superior results. Over the past years, it has been demonstrated that combining the routing step directly with circuit optimization leads to more efficient outcomes [48]. In response to these findings, this work focuses on optimizing a more abstract structure within quantum circuits, specifically Pauli-Polynomials, taking into account the inherent constraints of the quantum architecture. We developed three algorithms to achieve this optimization process, termed the architecture-aware synthesis of Pauli-Polynomials. We compared them against state-of-the-art circuit compiling strategies and showing improvements in reducing the CNOT count of larger Pauli-Polynomials.

2 Preliminaries

We will discuss relevant concepts in this thesis throughout the following chapter. Overall, we will start by defining fidelity, then introduce the concept of Parity Maps and Clifford Tableaus. Afterwards, we will introduce the ZX-Calculus, a graphical language for describing linear operations. In the end, we will discuss the VQE-Framework, and derive Pauli-Polynomials from it.

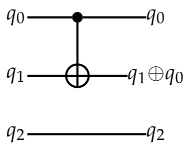
2.1 Fidelity

Fidelity is a central notion in quantum information theory that serves as a metric determining the similarity of two quantum states or quantum operations [40]. Unlike classical systems, where information can be perfectly copied and transmitted, quantum mechanics introduces inherent uncertainties due to the principles of superposition and entanglement. As a result, the traditional notion of similarity based on exact matching needs to be revised in the quantum realm.
In quantum formalism, we can define fidelity, denoted by $F(\sigma, \rho) = Tr\left[\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}}\right]^2$, measures the degree of overlap or agreement between two quantum states, represented by density matrices ρ and σ [40]. It quantitatively assesses how faithfully a quantum state ρ can be transformed into another state σ or vice versa. The fidelity takes values between 0 and 1, with $F(\sigma, \rho) = 1$ indicating perfect similarity, and $F(\sigma, \rho) = 0$ representing complete dissimilarity. In our context, we will utilize fidelity to represent the closeness of our synthesized operators. Here it is worth noting the *helldinger-fidelity* $F(Q, P) = (\sum_i Q_i P_i)^2$, which we can use in terms of shots, which is the output format of quantum devices.

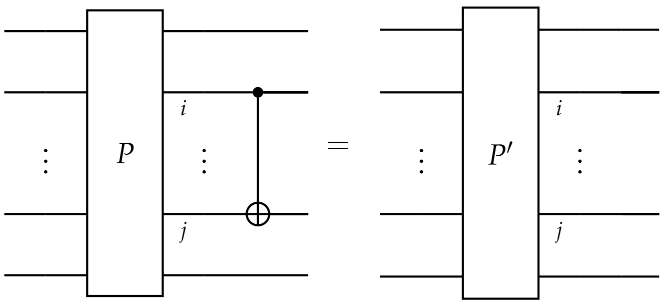
2.2 Parity Maps

Parity Maps are defined as $n \times n$ symplectic matrices over $GF(2)$ [1]. We can utilize Parity Maps to simulate pure CNOT circuits. Assume that we start at a symplectic matrix in $GF(2)$, which we set to identity if no operations had been applied to the quantum circuit. We can now create the Parity Map describing our CNOT-Circuit, by incrementally applying CNOTs to the initial parity map. Generally speaking, we have two options to compose an operation onto a parity map: We can append or prepend a CNOT. Following Patel et al. [41] we can observe, that the application of a CNOT on a quantum circuit is described by a addition modulo two of the control and the target qubit onto the target qubit:

[1]The Galois field describes the finite Field of two element operations

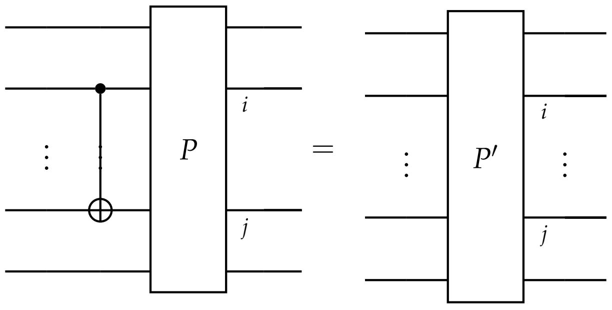


We hence conclude that one can store the application of one CNOT into our parity map by a row addition modulo two:



$$\implies P \xrightarrow{P'_j = P_i \oplus P_j} P'$$

In the case of prepending, the row addition modulo two will be flipped into a column addition modulo two [41]. We can, hence, express the vectorized version of the appending operation as follows:



$$\implies P \xrightarrow{P'_{:,j} = P_{:,i} \oplus P_{:,i}} P'$$

Given our CNOT-Circuit as parity maps in $GF(2)$, we can simulate the effect of our circuit on a state by performing a matrix multiplication. Compare this with the state-vector simulation of the $2^n \times 2^n$- Unitary, and the exponential speedup of this method becomes clear. Nevertheless, it must be noted that parity maps only form a specific subset of quantum circuits.

Synthesis of Parity Maps

The structure of Parity maps as matrices in $GF(2)$ allows the notion that it can be retransformed to identity by performing Gaussian elimination. We can record the individual row operations, forming a pure CNOT circuit. Patel et al. [41], has formulated this concept prooving an upperbound of $\mathcal{O}(\frac{q^2}{\log(q)})$. See Figure 2.1 for an outline of the process. Patels et al.'s work congruently showed a further interesting extension by Kissinger and van de Griend [30], and Nash et al. [39]. Their main contribution was to provide an architecture-aware synthesis algorithm of Parity maps. In the following, we will outline the key concepts of the approach by Kissinger and van de Griend [30].

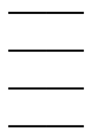
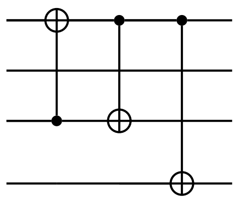
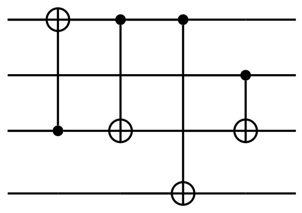
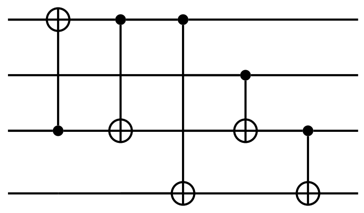
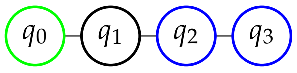
1. Start with the non reduced Parity Map	
$\begin{bmatrix}0 & 1 & 1 & 0\\ 0 & 1 & 0 & 0\\ 1 & 1 & 1 & 0\\ 1 & 0 & 1 & 1\end{bmatrix}$	
2. Cancel the elements in the first column, record the operations	
$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 1 & 1 & 0\\ 0 & 0 & 1 & 1\end{bmatrix}$	
3. Cancel the elements in the second column, record the operations	
$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 1 & 1\end{bmatrix}$	
4. Cancel the elements in the third column, record the operations	
$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$	
5. Note that we are done by this process and do not need to transpose	

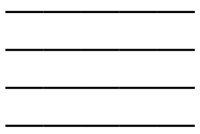
Figure 2.1: Example process of synthesizing a Parity Map using gaussian elimination.

In standard Gaussian elimination, we would simplify a column by adding a row for every nonzero element. After simplifying each column, we would transpose the Parity map and continue with the next column. We can extend this process towards a device topology. In order to do this extension, we have to compute the Steiner tree on the device's architecture graph [33, 30]. We can find a tree traversal by computing a depth-first search from a nonzero node and reversing it. Along this traversal, we check if the parent is not one. In this case, we will add a CNOT to the circuit from child to parent and compute the corresponding row addition, which will "fill" the parent. Otherwise, we will do nothing. As a last step, we will move in a reversed traversal from the pivot element as root canceling all child nodes. See Figure 2.2 for an example process. One can note that this way, all CNOTs have been placed along the edges of our connectivity graph and need no further routing step. Overall this process of Gaussian elimination has been quite competitive in the synthesis of CNOT circuits simply because it does not require further routing and provides an asymptotically upper bound on the CNOTs placed on the quantum circuit. This process, initially formulated by Kissinger and van de Griend [30] as a recursive process, named *recursive-steiner-gauss*, will be referred to as *steiner-gauss* in the following. At last, we want to point the reader towards the *perm-row-col* extension of this process [49, 53]. Here the authors used the fact that a global permutation can be applied to the measurements to optimize the process further.

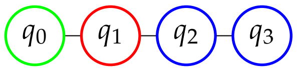
1. Mark all non zero entries in column zero



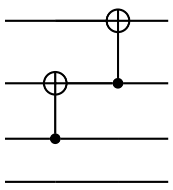
$$\begin{bmatrix} \boxed{0} & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ \boxed{1} & 1 & 1 & 0 \\ \boxed{1} & 0 & 1 & 1 \end{bmatrix}$$



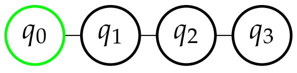
2. "Fill" the steiner tree (it is necessary to additionally place a CNOT at q_1 marked as red)



$$\begin{bmatrix} \boxed{1} & 1 & 0 & 0 \\ \boxed{1} & 0 & 1 & 0 \\ \boxed{1} & 1 & 1 & 0 \\ \boxed{1} & 0 & 1 & 1 \end{bmatrix}$$



3. "Empty" the steiner tree



$$\begin{bmatrix} \boxed{1} & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

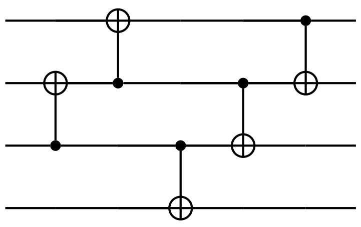


Figure 2.2: Example process of reducing one column using the *steiner-gauss* process.

2.3 Clifford Tableaus and the stabilizer Formalism

Given the three Pauli matrices and the identity matrix:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

The n-qubit Pauli group, G_n, is generated by tensor products of these four operators on n qubits [40]. Note, that we will denote such a tensor product (e.g: $X \otimes Y \otimes Z \otimes Y \otimes I$) as a string listing each operator, as is common notation in literature: $XYZYI$. With this definition, we can describe the Clifford group as the group that maps the Pauli group to itself [1], such that:

$$\forall P \in G_P \quad C^\dagger PC \in G_P$$

This definition implies that all elements of the n-qubit Clifford Group are essentially a mapping between different states of the pauli group. Their set of operators can hence be reasoned as finite. Generally speaking, we can create any Clifford gate by using the Gates $\{H, S, CX\}$:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad S = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} \end{pmatrix} \quad CX = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.1}$$

Since the Clifford group is finite, it cannot describe universal quantum operations. Nevertheless, one can achieve universality by enhancing the Clifford Group with the T-Gate, which yields the group: Clifford+T. Following the stabilizer formalism [1], one of the most important and foundational ideas of quantum error correction, we can obtain a representation that is similar to the Parity Maps for pure CNOT-Circuits. Specifically, we will refer to this as a *Clifford Tableau*.

Stabilizer and Destabilizer Generators

In common literature [40, 1, 24], a unitary operator U stabilizes a state $|\psi\rangle$ if it is an eigenstate of U with an eigenvalue of exactly one. This was utilized in Gottesman et al.'s seminal work to efficiently simulate quantum circuits composed of Clifford gates by representing quantum states via the elements of the Pauli group that stabilize the state of interest. [24] More formally, we can define stabilizer states as the eigenstates of a vector $|\psi\rangle$ that does not change the sign of the prefactor. For example, the state $|0\rangle$ is stabilized by Z since $Z|0\rangle = |0\rangle$. Stabilizers of Pauli products (referred to as stabilizers hereafter) are of particular interest as they form a basis for stabilizer states.
Furthermore, if two unitaries stabilize a state, then the product of these unitaries also stabilizes the state. In other words, a set of stabilizer states forms a group with a generator. We associate a destabilizer as the inverse of the particular generators of the stabilizer group. An n-qubit Pauli product generally has n stabilizer generators and n destabilizer generators.

Clifford Tableaus

We already have seen that, by definition, a Clifford operation can be uniquely characterized by how it transforms a Pauli basis [2]. We also have seen that an n-qubit state is stabilized by n-stabilizers. Hence the idea arises to store the transformation of a Clifford operation of the n-stabilizers by a so-called *Clifford Tableau* [1, 24]. A Clifford tableau is represented as a $2n \times 2n$ matrix over the Galois field $GF(2)$. In this matrix, the first n columns $(x_{r1}, x_{r2}, ..., x_{ri}, ..., x_{rn})$ describe conjugation by a Pauli X gate on the i-th qubit (the destabilizer rows). The latter n columns $(z_{r1}, z_{r2}, ..., z_{ri}, ..., z_{rn})$ describe the conjugation by a Pauli-Z gate on the i-th qubit (the stabilizer rows). The sign change of the qubit i is denoted in an additional column, r. Thus, we can deconstruct each stabilizer/destabilizer row by computing the following:

$$f(k) = (-1)^{r_k} \cdot \prod_{i=1}^{n} Z_i^{z_{ki}} X_i^{x_{ki}}$$

Here, Z_i represents Pauli Z applied to the i-th qubit, analogously X_i for the Pauli X gate. From this formula, we can see the requirement of the sign change arising from the description of $Y = iZX$ in the Tableau formalism. By default, the stabilizer and destabilizer generators of the state $|0 \dots 0\rangle$ are used, for which the Clifford tableau will conveniently form an identity matrix. We can then append and prepend Clifford Gates by decomposing them into H, S, and CNOT gates. For the deviation of the appending and prepending operations for H, S, and CNOT gates, we refer the reader towards Gidney [22]. We want to provide the intuition here that, similar to Parity Maps, one will look at the effect of an H, S, or CNOT Gate on the stabilizer and destabilizer states. Oberserve Figure 2.3 for the appending and Figure 2.4 for the prepending operations. Note that we have omitted the sign vector in those visualisations. More formally, we can follow Aaronson and Gottesman [1]:

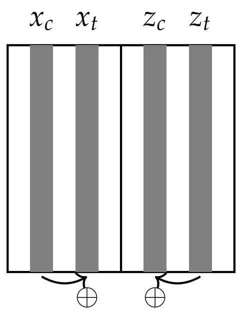
Appending of a H-Gate at qubit a:

$$\begin{aligned} &\forall i \in \text{rows}. \\ &\qquad x_{ia} &&= z_{ia} \\ &\qquad z_{ia} &&= x_{ia} \\ &\qquad r_i &&= r_i \oplus x_{ia} z_{ia} \end{aligned} \tag{2.2}$$

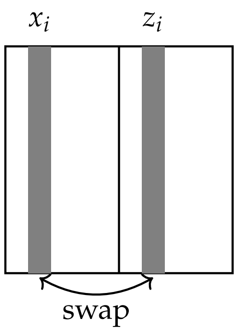
Appending of a S-Gate at qubit a:

$$\begin{aligned} &\forall i \in \text{rows}. \\ &\qquad z_{ia} &&= x_{ia} \oplus z_{ia} \\ &\qquad r_i &&= r_i \oplus x_{ia} z_{ia} \end{aligned} \tag{2.3}$$

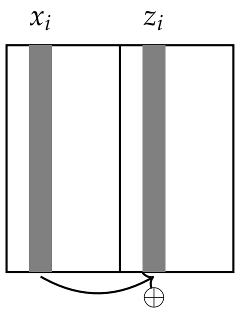
[2]We will refer to elements of the pauli group as Pauli basis in the following.



(a) Append: $CNOT_{c,t}$

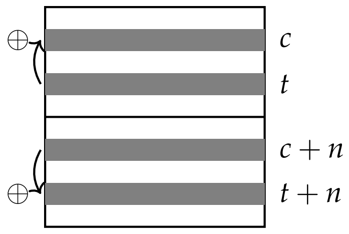


(b) Append: H_i

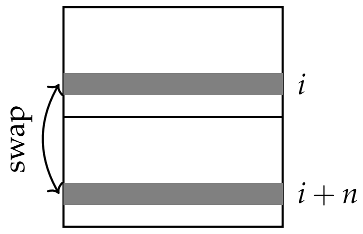


(c) Append: S_i

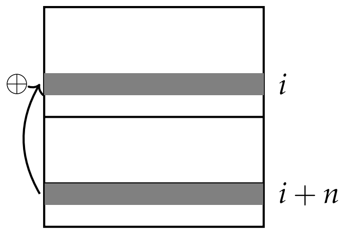
Figure 2.3: Appending operations for the Clifford Gates H, S and CNOT on a Tableau.



(a) Prepend: CNOT$_{c,t}$



(b) Prepend: H_i



(c) Prepend: S_i

Figure 2.4: Prepedning operations for the Clifford Gates H, S and CNOT on a Tableau.

Appending of a CNOT-Gate at control a and target b:

$$\begin{aligned}
&\forall i \in \text{rows}. \\
&\quad x_{ib} = x_{ib} \oplus x_{ia} \\
&\quad z_{ia} = z_{ib} \oplus z_{ia} \\
&\quad r_i = r_i \oplus x_{ia} z_{ia} (x_{ia} \oplus z_{ia} \oplus 1)
\end{aligned} \tag{2.4}$$

Representation in Z/X-Basis

Working with the $2n \times 2n$ symplectic matrix describing a Clifford tableau might be tedious at some point. We hence coaligned our representation with libraries like stim [22], providing a view on the tableau in X- and Z-Basis. Intuitively the X-Basis describes the behavior of all de-stabilizers of the tableau, the Z-Basis of all stabilizers. We can start by noting that a tableau can be described as a block matrix consisting of the XX, XZ, ZX, and ZZ block as follows:

$$\left[\begin{array}{c|c} \text{XX} & XZ \\ \hline ZX & ZZ \end{array}\right]$$

We can then define two matrices, X and Z, by computing $X = XX + 2XZ$ and $Z = ZX + 2ZZ$. One can note, that if the tableau is identity X and Z, will look the following:

$$\text{X:} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{Z:} \quad \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

An arbitrary tableau can be hence presented as follows:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \rightsquigarrow \quad \text{X:} \quad \begin{bmatrix} 0 & 0 & 3 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \qquad \text{Z:} \quad \begin{bmatrix} 0 & 0 & 2 \\ 0 & 2 & 0 \\ 2 & 2 & 0 \end{bmatrix}$$

By the equations and from the example, we can note that if an element of either the Z- or X-Basis is three, we are provided with a change of a Pauli Y; for a two, we can note a change to a Z, and for one a change to an X. We can hence read of the transformation of the Pauli-basis from this representation.

On the Synthesis of Clifford Tableaus

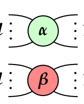
We can note, that similar to parity maps clifford tableaus consist of invertible symplectic matrices, which motivates gaussian elimination for gate synthesis. Here Aaronson and Gottesman [1], have provided the *canonical form theorem*, essentially stating, that any Clifford Tableau can be resynthesized using the following sequence of gates: H-C-P-C-P-C-H-P-C-P-C, providing the Collocary that any Tableau can be syntheisized with $\mathcal{O}(\frac{q^2}{\log q})$ many gates. This theorem has been improved by the heuristic of Bravyi and Maslov [8], which was able to improve the two qubit count up to 64%, compared to Aaronson and Gottesman [1].

The resemblance of Clifford Tableaus and Parity Maps

By examining Clifford Tabelaus and Parity maps in more detail, much resemblance regarding the functionality and purpose can be found. First, symplectic matrices fully describe both in $GF(2)$. Both can effectively simulate the behavior of subtypes of quantum circuits, and we can append and prepend operations to both forms. Hence it might be highly sensible to test approaches that have been quite successful on parity maps on Clifford tableaus. Therefore we want to point the reader towards section 2.2. Note that conceptually we can synthesize a parity map onto a quantum circuit in an architecture-aware fashion by routing the CNOTs according to our Steiner tree estimation. Since in clifford circuits, most single qubit cliffords only change one specific element of a pauli string, sanitizing each row in the Z- and X-Basis could be a sensible application for the re-synthesis of a tableau.

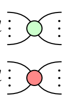
2.4 The ZX Calculus

The ZX Calculus is a rigorous graphical language first defined by Coecke and Duncan [12] as an extension towards categorical quantum mechanics [2]. Generally speaking it describes linear maps $\mathbb{C}^n \to \mathbb{C}^n$ in a penrose-like notation. In recent years it has various applications in different fields of quantum computing such as circuit optimization [13, 16, 21, 48, 14], lattice surgery [15] or the design of quantum error correction codes [10, 11]. A list of publications is maintained at: https://zxcalculus.com/publications.html. We will use the introductory work, of van de Wetering [50] for a brief definition. As mentioned earlier, the ZX Calculus follows a penrose-like notation of so called X or Z Spiders. Those Spiders are defined as follows:



$$\begin{aligned} m \vdots \; \vdots n &= e^{i\alpha} |1\rangle^{\otimes m} \langle 1|^{\otimes n} + |0\rangle^{\otimes m} \langle 0|^{\otimes n} \\ m \vdots \; \vdots n &= e^{i\beta} |-\rangle^{\otimes m} \langle -|^{\otimes n} + |+\rangle^{\otimes m} \langle +|^{\otimes n} \end{aligned} \tag{2.5}$$

Note that by setting the phase argument α or β to zero, we can obtain empty spiders:

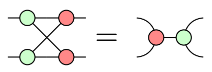


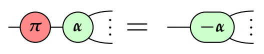
$$\begin{aligned} m \vdots \; \vdots n &= |1\rangle^{\otimes m} \langle 1|^{\otimes n} + |0\rangle^{\otimes m} \langle 0|^{\otimes n} \\ m \vdots \; \vdots n &= |-\rangle^{\otimes m} \langle -|^{\otimes n} + |+\rangle^{\otimes m} \langle +|^{\otimes n} \end{aligned} \tag{2.6}$$

As one can see the different spiders are connected by so called wires, here exists one special ase the H-Wire, describing a Hadamard gate in the ZX-Calculus:

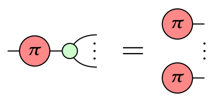
$$= |+\rangle \langle 0| + |-\rangle \langle 1| \quad = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.7}$$

The main prupose of the ZX Calculus is now to provide a set of rewrite rules, which preserve the operators described by the combination of X and Z spiders. See Equations (2.8) to (2.15), for an outline of rules. Note that those rules have been adjusted to fit the notation used by Cowtan et al. [13], and certain types of spiders are defined by their quantum gate counterparts. We want to refer the interested reader to van de Wetering [50] at this point since a detailed discussion of the ZX Calculus with all of its caveats and rules is out of the scope for this thesis.

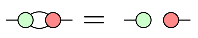

(2.8)

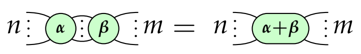


(2.9)

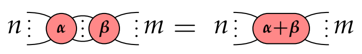


(2.10)

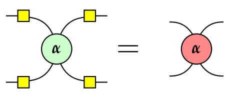

(2.11)



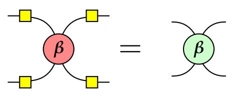
(2.12)



(2.13)



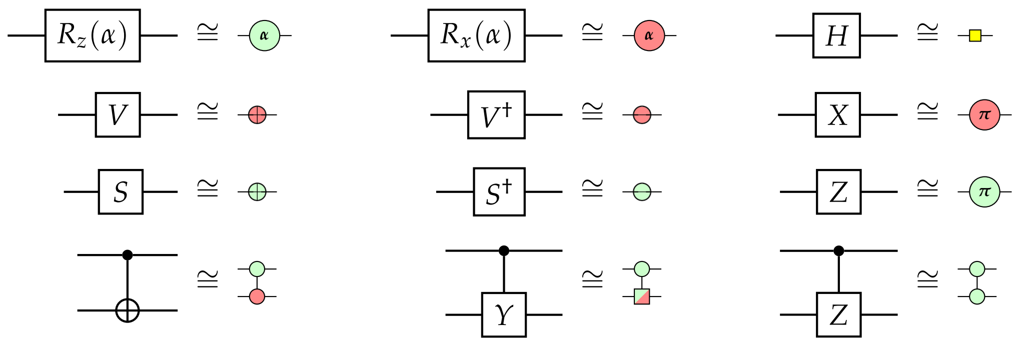
(2.14)



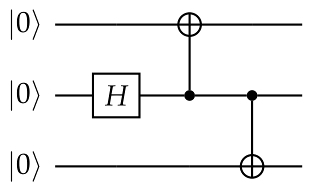
(2.15)

2.4.1 Resemblance of Quantum Circuits and the ZX Calculus

The basis of digital quantum computing are so called quantum circuits, defining a unitary map between an input state and an output state, which we can then measure. We can hence describe every quantum circuit by a ZX Diagram, a simple yet naive process is to convert the circuit towards a circuit with the gate set in $\{R_x, R_z, H, S, S^\dagger, V, V^\dagger, CX, CY, CZ\}$ and then transform it into the ZX-Calculus using the following rules [3]:

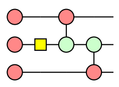


To make this process more clear, we can, for example, describe the GHZ-State by the following quantum circuit:

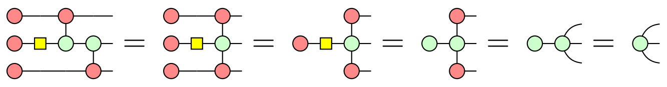


[3]Note that we have adapted the rules from Cowtan et al. [14]

Rewriting this to a ZX Circuit will yield the following diagram:



We can then apply rewrite rules sequentially to obtain the GHZ-State:



While the process of synthesizing ZX Diagrams from quantum circuits is quite intuitive, the opposite is not directly true. de Beaudrap et al. [17] has shown that in general synthesizing a ZX Diagram towards a quantum circuit is #P-Hard. Hence while the ZX Calculus provides a more intuitive reasoning about unitaries, it needs to be treated with caution when it comes to the point of re-synthesis of such diagrams.

2.5 The VQE Framework

The variational quantum eigensolver (VQE) [42] describes a hybrid classical and quantum technique that aims to optimize an upper bound for the lowest expectation value of a physical system (for instance, a molecule or nucleus). Specifically, given a Hamiltonian H, describing the physical system and a parametrized trial wave function $|\psi(\vec{t})\rangle$ (ansatz), the ground state of the physical system is bound by:

$$E_0 \leq \min_{\vec{t}} \frac{\langle\psi(\vec{t})|\, H \,|\psi(\vec{t})\rangle}{\langle\psi(\vec{t})|\psi(\vec{t})\rangle}$$

We hence want to optimize $\vec{t}$ to obtain the ground state of our physical system. We will focus on the chemically-inspired unitary-coupled-cluster ansatz based on single and double excitations (UCCSD). We can hence define T, the excitation operator in the second quantization, as follows:

$$T = \sum_{i=1}^{2} T_i \tag{2.16}$$

$$T_1 = \sum_{i,a} t_a^i a_a^\dagger a_a \tag{2.17}$$

$$T_2 = \sum_{i,j,a,b} t_{ab}^{ij} a_a^\dagger a_b^\dagger a_a a_b \tag{2.18}$$

Here a_p and $a_p^\dagger$ describe the annihilation and creation operators for the orbital index by p. The expansion coefficients t_a^i and t_{ab}^{ij} will be abstracted into a vector $\vec{t}$, which will describe the parameters we want to optimize. A detailed deviation of UCCSD is out of the scope of this thesis; we will hence refer the reader towards [47, 26, 44, 6]. For our purposes, it suffices to note that we will describe our ansatz as:

$$|\Psi(\vec{t})\rangle = U(\vec{t})\,|\psi_0\rangle = \exp\left(T(\vec{t}) - T^\dagger(\vec{t})\right)|\psi_0\rangle \tag{2.19}$$

With $|\psi_0\rangle$, describes the fixed reference state, for which a common choice is the Hartree Fock wavefunction [47]. Since T is a linear combination of fermionic creation and annihilation operators (denoted as $\vec{\tau}$ in the following to confirm the notation of [14]), we can rewrite the operator as:

$$U(\vec{t}) = \exp\left(T(\vec{t}) - T^{\dagger}(\vec{t})\right) = \exp\left(\sum_j t_j(\vec{\tau}_j - \vec{\tau}_j^{\dagger})\right) \tag{2.20}$$

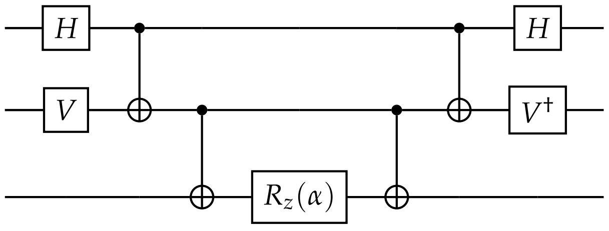
To convert this ansatz towards a quantum circuit, we can firstly rewrite the unitary into a series of unitaries we can apply to our quantum circuit using trotterization[46]:

$$U(\vec{t}) \approx U_{\rho}(\vec{t}) = \left[\prod_j \exp\left(\frac{t_j}{\rho}(\vec{\tau}_j - \vec{\tau}_j^{\dagger})\right)\right]^{\rho} \tag{2.21}$$

Here ρ describes the degree of trotterization, which is mostly set to $\rho = 1$ in the NISQ-era due to limited device capabilities. At last, it is required to map the operators toward a digital quantum device. Herefore, as a first step, encoding strategies such as Jordon-Wigner (JW), Bravyi-Kitaev (BK), or Parity (P) are used [54]. Libraries such as OpenFermion [37] or pennylane [7], provide this natively, providing us with the prefactors α and Pauli strings P_i of a so-called *pauli gadget*:

$$\vec{\tau}_j - \vec{\tau}_j^{\dagger} = i\alpha \sum_{i=0} P_i \quad P_i \in \{X, Y, Z, I\} \tag{2.22}$$

We can now implement such a *pauli-gadegt*, by using R_z, CX, H, V and $V^{\dagger}$ Gates. Take, for instance, the following circuit, descrcibing the Pauli-gadget: $\exp\left(i\frac{\alpha}{2}XYZ\right)$:



(2.23)

At last, it is worth noting Cowtan et al. [14]'s assumption on the trotterization error. Cowtan et al. [14], notes that according to Low et al. [35], the trotter-error of Equation 2.21 is asymptotically bound by:

$$||\delta_{\rho}|| = \mathcal{O}\left(\frac{1}{\rho}(\sum_j ||t_j(\tau_j - \tau_j^{\dagger})||)^2\right) \tag{2.24}$$

Given a well-conditioned reference state, one can assume that low amplitudes will parametrize the ansatz. In other words, we can assume that $\forall j.t_j \ll 1$. Hence the error introduced by trotterization will be small compared to the error of, for instance, two-qubit gates on the physical device we will execute our calculation. Therefore Cowtan et al. [14] assume that we can arbirarly reorder all pauli-terms after trotterization, something we will refer to as n-to-n commutation in the following. On this topic of grouping terms, Gui et al. [27] have provided a further interesting ansatz of term grouping different Pauli gadgets, which minimizes trotter and physical device errors.

2.6 Z- and Pauli-polynomials

In section 2.5, we have seen how a *Pauli-gadget* can be applied on a quantum circuit. Nevertheless, working with quantum circuits consisting of Pauli-gadgets may be tedious to define and can even hinder optimization [13]. We can hence define a *Pauli-gadget* in the ZX-Calculus as in Figure 2.5. The first part of the equation shows the notation of a Pauli-gadget, the second its diagonalization, and the third its decomposition in the ZX-Calculus. We can observe that the decomposition corresponds towards Equation 2.23. With *legs*, we will refer to the connections on the qubits colored in either red (for X), green (for Z), or red-green (for Y). We can chain multiple such Pauli-gadgets together, which will yield a Pauli-polynomial. Pauli-polynomials can be categorized into different classes depending on their types of legs. If we encounter only Z legs, we will refer to the Pauli-polynomial as Z-Polynomial (other authors refer to Z-Polynomials as phase polynomials as van de Griend and Duncan [48]). If we encounter Z and X legs, we will refer to the Pauli-polynomial as ZX-Polynomial. One can acknowledge

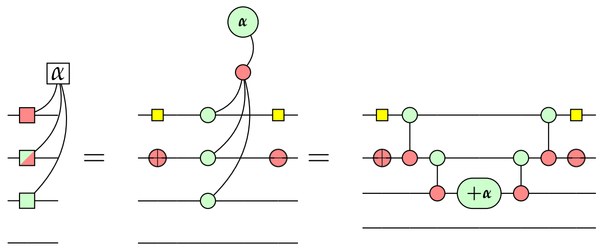


Figure 2.5: Pauli gadget representation of $e^{-i\alpha XYZI}$ per notation in [13] (left) and in ZX-calculus (right)

that the formalism of describing totterized Hamiltonians as Pauli-polynomials is highly expressive by looking at Figure 2.6. Here we have provided a representation of the two-qubit parity encoding of the H2 molecule with sto3g basis. One can observe the corresponding Pauli-polynomial in the upper part of the figure. The quantum circuit is defined below. Additionally, this structure allows us to propagate Clifford operations, conduct statements of commutativity and diagonalize Pauli-polynomials, as we will see in the following.

2.6.1 Propagation of Cliffords

From section 2.3, we can note that a Clifford gate is defined by how it changes a Pauli basis. Following Gogioso and Yeung [23], we can extend this concept towards Pauli-gadgets. In the

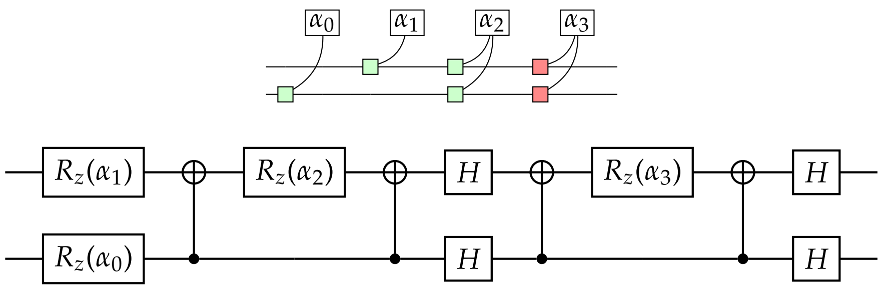
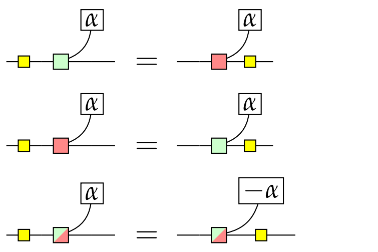


Figure 2.6: Representation of the trotterization of $\exp(-i\vec{\alpha}(IZ + ZI + ZZ + XX))$. Once with a Pauli-polynomial and once with a quantum circuuit

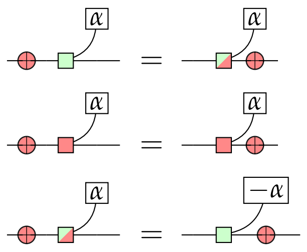
following, we will assume an n-qubit Clifford gate, which acts without loss of generality on the n-first legs of the Pauli-polynomial. Recall that the operator representation of our Pauli-polynomial is defined as $\exp(-i\frac{\theta}{2}P)$, where $P = \bigotimes_{i=0}^{n} P_i$ and $P_i \in \{X, Y, Z, I\}^n$. Applying now a Clifford-Gate to the Pauli-polynomial, we can note that it commutes under matrix exponentiation:

$$
\begin{aligned}
C(\exp(-i\frac{\theta}{2}P))C^{\dagger} &= \\
C\left(\sum_{k=0}^{\infty}\frac{1}{k!}(-i\frac{\theta}{2}P)^k\right)C^{\dagger} &= \\
\sum_{k=0}^{\infty}\frac{1}{k!}C(-i\frac{\theta}{2}P)^k C^{\dagger} &= \\
\exp\left(-i\frac{\theta}{2}C(P)C^{\dagger}\right)
\end{aligned}
$$

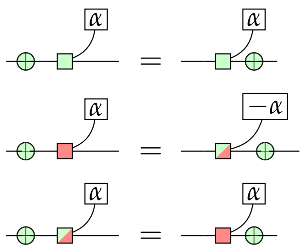
From this calculation, we can conclude that the Clifford gate will affect the pauli string similarly as it would map between two Pauli states. Additionally such a mapping may introduce a either positive or negative sign, which gets absorbed into the phase θ. Hence we can conclude the rewrite rules in Figure 2.7 for single qubit gates and Figure 2.8 for two-qubit gates. In contrast to other authors [13, 14, 55, 23], we have included CY and CZ gates here among the CX Propagation rules to outline the general possibility of propagating Cliffords. At last, we want to point the reader towards Grier and Schaeffer [25]. They have performed a detailed analysis of the Clifford Group, introducing the concept of X-, Y- or Z-preserving Cliffords. We can call a Clifford Gate Z-preserving if it maps Z-basis states to themselves. For instance, the H-Gate is Y-Preserving while the V-Gate is X-Preserving. We confirm both observations by looking at Figure 2.7a and Figure 2.7b respectively. In this context, the CZ-Gate is interesting since it is Z-Preserving. We can see that all possible combinations of ZI, IZ, and ZZ are mapped to themselves.



(a) Propagation Rules for the H-Gate.



(b) Propagation Rules for the V-Gate.

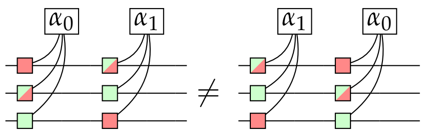


(c) Propagation Rules for the S-Gate.

Figure 2.7: Single Qubit clifford Propagation trough a Pauli-polynomial

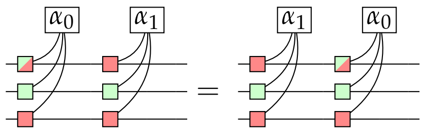
2.6.2 Commutation of Pauli-polynomials

From Yeung [55], we find that two pauli gadgets commute if and only if they share an equal amount of non equal legs. To make this clear, observe that the following pauli gadegts to not commute, since they share three unequal legs:



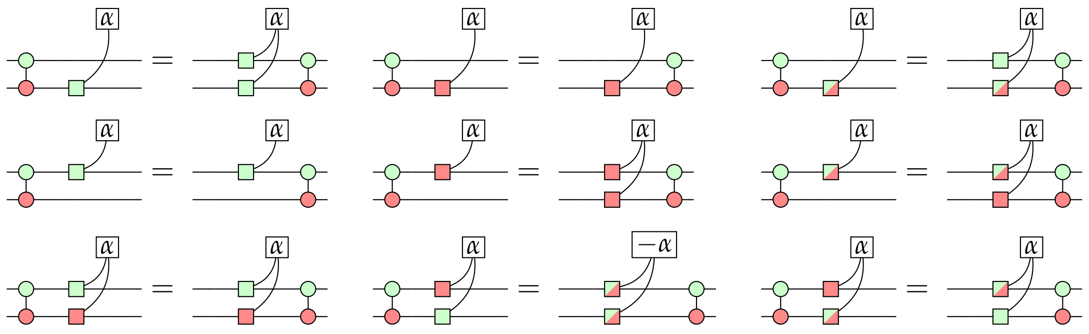
(2.25)

We see, that here they do not commute, so we cannot swap the two gadgets. Nevertheless for the following two gadegts it is possible to swap since they share two legs with differing rotation axis:

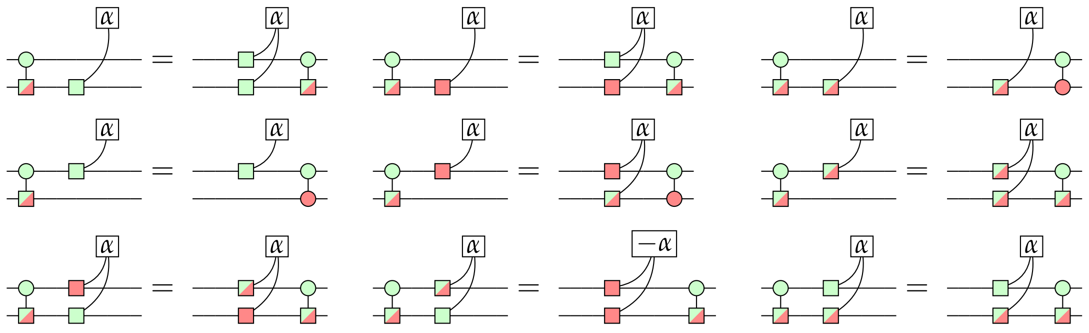


(2.26)

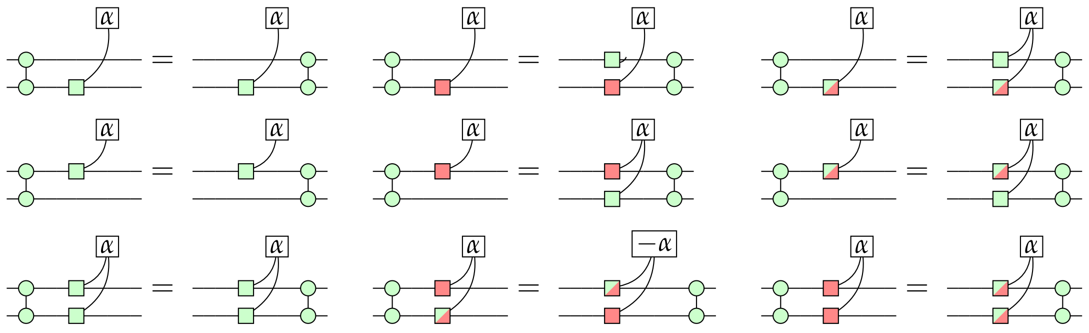
A detailed proof of this concept is out of the scope of this thesis, we herefore refer the reader to Yeung [55].



(a) Propagation Rules for the CX-Gate.



(b) Propagation Rules for the CY-Gate.



(c) Propagation Rules for the CZ-Gate.

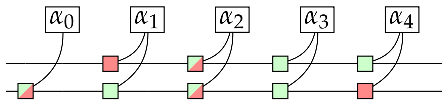
Figure 2.8: Two Qubit Clifford Propagation of the CX-, CY- and CZ-Gate.

2.6.3 Diagonalisation of Pauli-polynomials

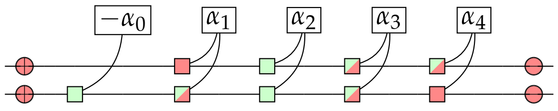
We have already seen that two Pauli-polynomials can commute if they share an even number of *non-equal legs*. We call a region of Pauli-polynomials, where each gadget commutes with the others, a mutually commuting region. We can *diagonalize* such a region, meaning that we convert the Pauli-polynomial into a clifford region, a Z-polynomial and the same clifford region conjugated Cowtan et al. [14] outlines such a process by defining so-called *compatible pairs*. Generally speaking, a compatible pair is a pair of qubits (i, j), for which the following relation holds:

$$\exists A, B \in \{X, Y, Z\} s.t. \quad \forall l \in \{1 \dots m\}, \quad \sigma_{il} \in \{I, A\} \iff \sigma_{jl} \in \{I, B\} \tag{2.27}$$

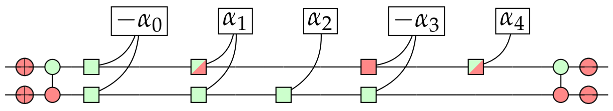
Here σ_{im} describes the leg of the gadget m at qubit i of the Pauli-polynomial, and m are the number of gadgets in the Pauli-polynomial. Given such a compatible pair, we can always diagonalize one of its qubits. For example, assume the following slightly simplified example from Cowtan et al. [14]:



Oberserve, that according to Equation 2.27, we can determine the compatible pair: (Y, Y). This pair indicates that we must propagate a V-Gate at qubit zero and one. We can conclude this by looking at Figure 2.7b. Note that any Y-leg will be transformed into a Z-leg while the X-legs will be preserved. We can observe our Pauli-polynomial after propagation as follows:



At last, we will propagate a CX-Gate through the Pauli-polynomial, effectively diagonalizing qubit one:

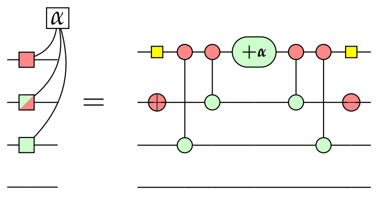


With this knowledge, we can iteratively diagonalize any mutual commuting region by iteratively performing the following three-step procedure:

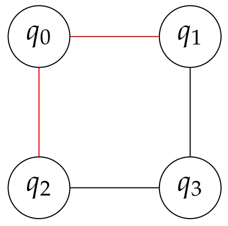
- Check for single-qubit gates, if there is one row with legs only in one of $\{X, Y, Z\}$, we can apply the corresponding clifford according to Figure 2.7. Remove the corresponding qubits from our list
- Check for compatible pairs; if they are present, apply the corresponding Clifford and the not, which will diagonalize the target row. Remove the corresponding qubits from our list.
- If neither the single nor the two-qubit process has succeeded. Decompose the Pauli Gadget with the fewest legs, corresponding to $III \dots Z$. Remove the corresponding qubits.

2.6.4 Architecture Aware Decomposition of Pauli-polynomials

We want to point out to the reader that the decomposition in Figure 2.5 must not necessarily be a ladder of CNOTs. Cowtan et al. [13], has, for instance, found different decompositions of CNOTs, which fueled a more effective synthesis of cliffords. We want to point the reader towards the decomposition of Gogioso and Yeung [23]. They computed a minimal spanning tree for every Pauli gadget in their ZX-Polynomial, which allowed an estimation of the cost of implementing the corresponding gadget on a specific hardware. We can easily extend this concept towards Pauli-polynomials, so for instance, the pauli gadget in Figure 2.5 will look decomposed on a line architecture as in Figure 2.9.



(a) Decomposed Pauli-polynomial



(b) Connectivity Graph

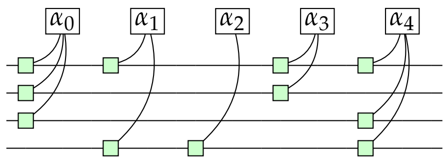
Figure 2.9: Decomposition of the Pauli Gadget: $e^{-i\alpha XYZI}$ for a square architecture. We marked the minimal spanning tree in red in the connectivity graph aside.

2.6.5 Architecture Aware Synthesis of Z-Polynomials

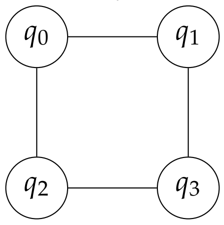
At last, we want to discuss a variant of the *gray-synth* algorithm initially found by Amy et al. [4]. Here van de Griend and Duncan [48] has found an architecture-aware synthesis process, which we will refer to as *steiner-gray-synth*. Their algorithm has a recursive structure consisting of two steps: At once, the *base recursion* and second, the *one recursion* step, which we will elaborate on in the following. We will outline the algorithm's functionality by an example, using the Z Polynomial and connectivity graph in Figure 2.10. Note that this is not a complete example since we execute each step once to outline the functionality. We refer the reader to van de Griend and Duncan [48] for further details and examples.
We start off by defining the Z-Polynomial, a remaining Parity Map, and the circuit as global variables, such that they are acessbile by the base and one recursion.

Base recursion Step

The core concept of the base recursion step is to subdivide the Z-Polynomial into two parts, such that one is as large as possible. We are provided in the function arguments with the remaining columns describing the Z-Polynomial (in our case, this would be: $\alpha_0, \alpha_1, \alpha_2, \alpha_3$ and α_4) and the remaining qubits (in our case: q_0, q_1, q_2 and q_3). Therefore, we pick a noncutting vertex such that we have either the most zeros or ones in one of the partitions. We refer to a



(a) Z-Polynomial



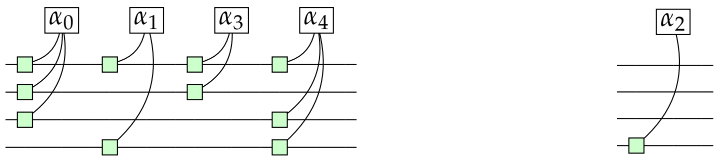
(b) Connectivity graph of the quantum device

Figure 2.10: Z Polynomial and connectivity graph for outlining the process of *steiner-gray-synth*

noncutting vertex as a vertex that will not disconnect the connectivity graph among removal. More formally:

$$i = \underset{q \in \text{Non Cutting}}{\arg\max} \max_{x \in GF(2)} \{c \in \text{Cols} \wedge P_{r,c} = x\} \tag{2.28}$$

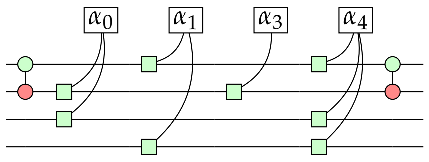
Note that here $P_{r,c} \in GF(2)$ notes if a leg is present at the current gadget (gadget in column c) at the row r. The set, Non Cutting, describes the noncutting vertices in the connectivity graph and Cols the columns (gadgets) we are looking at in the current base recursion step. In our example, we would pick q_0 at the first recursion step since it maximizes Equation 2.28 among all other graph nodes. We, therefore, split the Z-Polynomial using q_0, which will provide us with the following subsets:



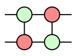
For the gadget α_2, we would run the base recursion decomposing it into a R_z rotation since it has only one leg. The more interesting part is the recursion with the gadgets: $\alpha_0, \alpha_1, \alpha_3$ and α_4. We will utilize all the currently present qubits as we proceed to the one recursion step. For the base recursion, we will remove q_0.

One recursion step

The goal of the one recursion step is to remove as many legs from row q_0 as possible. Note, therefore Figure 2.8a; we observe that if our Pauli-gadget contains only Z-legs, we will either introduce or remove one leg among propagation of a CNOT. We will hence select q_1 among the neighbors of q_0 on the connectivity graph. In this case, we select q_1 among the neighbors since it has the most legs. Next, we place a CNOT: CX_{q_0,q_1} on the circuit and propagate its counterpart through the Z-Polynomial, leaving us with the following structure:



In the end, we will collect the propagated CNOT in a Parity Map, which we can synthesize using the *steiner-gauss* algorithm. We have encountered a single-leg gadget at α_3, meaning that we can remove this gadget and apply the corresponding R_z-Gate on the global quantum circuit. Also note that should we encounter a row with no legs, we can swap them by applying a CNOT twice:



At last, we will split out Z-Polynomial among q_1 and continue to the base recursion by removing q_1 from the qubits as well as to the one recursion by keeping all qubits, but this time with row: q_1.

Architecture-Awareness and further remarks

Since we are picking *CNOT*'s along the connectivity graph until single gadgets are reduced to one leg, this process will map the Z Polynomial towards a quantum circuit that fits the connectivity graph of our device. We further want to point out to the reader the fundamental concept of this algorithm: We first select the qubit, which promises us the ability to remove as many legs as possible; we then select the neighbor, which can remove from this qubit as many legs as possible. This core concept allows the CNOT-Reduction of both the algorithm by van de Griend and Duncan [48] and Amy et al. [4]. Amy et al. [4] even showed in evaluations that their algorithm approximates the optimal solution by a small margin. At last, note that a direct extension towards noncommuting Pauli-polynomials is not possible in this algorithm. Even if we assume n-to-n commutativity, we could still introduce a leg by one of the CNOT-propagation rules in Figure 2.8a. The only alternative in known literature is applying *steiner-gray-synth* or *gray-synth* to an diagonalized Pauli-polynomial. We can make this Pauli-polynomial as large as possible by splitting it into smaller commuting subsets of maximal commuting cliques.

3 Related Work

As discussed earlier, we focus on the architecture-aware synthesis of Clifford Tableaus and Pauli-polynomials. We will outline the current state-of-the-art landscape of Clifford simplification techniques and techniques to synthesize a Pauli-polynomial.

Synthesis of Clifford tableaus

For the synthesis of Clifford Tableaus, our work is based on the stabilizer formalism by Aaronson and Gottesman [1], enhanced by some explanations and introductions of Gidney et al. [22]. Hereby our focus is not on simulating the Clifford group but rather on using Clifford Tableaus as a high-level structure for quantum gate synthesis. In the process of synthesizing clifford Circuits, most researchers relied on so-called normal forms, such as the one we have seen by Aaronson and Gottesman [1]: H-C-P-C-P-C-H-P-C-P-C. Dehaene and De Moor [19] have formulated a courser-grained 5-layer form, which led to improved versions by Maslov and Roetteler [36] and den Nest [20], respectively. Duncan et al. [21] have improved their pyzx library of those normal forms listed above. We also want to point the reader towards Vandaele et al. [51], where they have provided a tableau synthesis algorithm that is optimal concerning the H-count of the resulting circuit. Amy et al. [5], has shown that the synthesis of sum-over-paths is **cp-NP**, but reduced to **P** if we restrict ourselves to Clifford circuits. All in all, one can observe remarkable results in the landscape of Clifford optimization algorithms and that the synthesis of Cliffords may fuel general quantum circuit simplification and synthesis. In this work, we will focus on creating an architecture-aware synthesis algorithm for Clifford Tableaus, which to our knowledge, has never been done before.
At last, we want to point the reader towards the algorithm by Bravyi and Maslov [8], used in our benchmarks as a state-of-the-art baseline. Their method used a combination of template matching, designed explicitly for Clifford circuits, to level out CNOT and SWAP gates. Currently, their algorithm is implemented as the standard when synthesizing a Clifford tableau in the qiskit software stack.

Synthesis of Pauli-polynomials

As Pauli-polynomials naturally describe Hamiltonian simulation on digital quantum devices, the field of synthesis is twofold. At once, researchers focus on reducing the corresponding circuit's CNOT count. Here Cowtan et al. [13]'s work was fundamental in the definition of Pauli-polynomials as phase-circuits, as well as the initial idea of decomposing the CNOT-Ladders introduced, for instance, by Figure 2.6 in a different way, which exposes more CNOT's to the Clifford circuit between two Pauli gadgets. This way, Clifford simplification algorithms could remove more gates than standard optimization could provide. Relating Z-Polynomials a the posed problem definition by grouping individual Pauli gadgets into mutually commuting sets allows the application of algorithms like *gray-synth* by Amy et al. [4] or the *steiner-gray-synth* by van de Griend and Duncan [48], which are naturally more powerful at reducing the CNOT count. On the work of diagonalization of the Pauli-polynomial, de Brugière et al. [18] have found a graph-state based synthesis approach which outperforms Cowtan et al. [14]. At last, we want to point the reader towards Paulihedral [34], a compiler targeted precisely for Hamiltonian simulation. Paulihedral also outperforms the approach by Cowtan et al. [14]. Nonetheless, we used Cowtan et al. [14]'s work as a baseline of our approach since one will see that it is closely related to our synthesis attempts.

4 Architecture Aware Clifford Synthesis

In the following we want to dicuss the architecture aware syntehsis of Clifford tableaus and the algorithm we have developed in this context. Overall one can note, that our process is closely related to a Gaussian elimination process and consists of two major parts. Generally speaking, assume our Clifford tableau as in section 2.3:

$$X: \begin{bmatrix} 0 & 0 & \boxed{3} \\ \boxed{1} & \boxed{1} & 0 \\ \boxed{1} & 0 & 0 \end{bmatrix} \qquad Z: \begin{bmatrix} 0 & 0 & \boxed{2} \\ 0 & \boxed{2} & 0 \\ \boxed{2} & \boxed{2} & 0 \end{bmatrix}$$

Our goal is to reduce this tableau towards identity; hence we want it to be in the following form:

$$X: \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad Z: \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

We note that overall this requires us to remove the *interactions*, we can see in the stabilizer and destabilizer basis (marked in green), and we can see that certain elements of those interactions are not in the correct basis, e.g., three in row zero and column two. Hence for every pivot row, we can subdivide the process into the following two steps:

1. Sanitization of interactions
2. Removal of interactions

4.1 Sanitization of interactions

Examining Figure 2.3, we can see that the application of a H gate swaps the z and the x column of the Clifford tableau, the application of a S gate corresponds towards the addition from the x to the z column of a Clifford tableau. We can obtain the following rules for converting the stabilizer and destabilizer parts of the Clifford tableau towards identity when looking at *one particular row* [1]:

[1]Also note that this process can introduce non-sanitized elements when computed sequentially for all rows.

Basis	Inital value	Applied Gates	Final value
X	0	–	0
	1	–	1
	2	H	1
	3	S	1
Z	0	–	0
	1	H	2
	2	–	2
	3	S, H	2

With this set of rules, we can iterate through the individual columns of the row, applying the respective H and S gates; this way, we will sanitize our row, converting it towards only ones for the stabilizers and two only twos for the destabilizers. See algorithm 1 and algorithm 2 for the methods described in pseudocode.

Algorithm 1: Sanitization process of the Z-Basis

```
Function sanitize_z(r, tableau):
    $Z \leftarrow$ Z matrix of the tableau;
    for $c \leftarrow 0$ to tableau.n_qubits do
        if $Z_{r,c} = 3$ then
            Apply action "S" to qubit $c$;
        end
        if $Z_{r,c} = 1$ then
            Apply action "H" to qubit $c$;
        end
    end
```

Algorithm 2: Sanitization process of the X-Basis

```
Function sanitize_x(r, tableau):
    $X \leftarrow$ X matrix of the tableau;
    for $c \leftarrow 0$ to tableau.n_qubits do
        if $X_{r,c} = 3$ then
            Apply action "S" to qubit $c$;
        end
        if $X_{r,c} = 2$ then
            Apply action "H" to qubit $c$;
        end
    end
```

4.2 Removal of interactions

Removing interactions is critical for synthesizing a Clifford tableau in an architecture-aware fashion since we will apply the CNOT gates required to reduce the specific row to identity. From section 2.2, we have seen that this can be done by computing and traversing a Steiner tree on the devices connectivity graph. Hence, we will assume the current row is thoroughly sanitized, meaning it only consists of ones for the X-Basis and twos for the Z-Basis. We will then select all non-zero entries of the row and add the pivot column (the column with the index of the pivot row) towards the non-zero elements. As a second step, we will compute the Steiner-tree using Kruskals algorithm [32] [2]. We then compute a breath-first search (BFS) of this tree, with the pivot as the root. The following steps differ slightly between the X- and Z-Basis because the control and target qubits per CNOT gate will be swapped. We can justify this swap by observing that according to Figure 2.3, the application of a CNOT in the z columns is an addition to the left, while for the x columns, it is an addition to the right. We will hence outline the process for the X-Basis, noting that it is analogous to the Z-Basis. Given the BFS path of the Steiner tree, one can reverse traverse it twice. We obtain a parent p and child c per iteration step. A CNOT will be applied in the first traversal: $CX_{c,p}$ if the parent is zero. This way, we can ensure that all nodes in the Steiner tree are one after the traversal. All children will be emptied by applying the CNOT: $CX_{p,c}$. This process will ensure that since the Steiner tree is computed on the connectivity graph, CNOTs are placed in an architecture-aware fashion. Secondly, it will, in analogy to the *steiner-gauss*-Process, remove all interactions of the tableau. See algorithm 3 for the process outlined in pseudocode for the X-Basis, algorithm 4 for the process outlined in the Z-Basis.

Algorithm 3: Removal of interactions on the X-Basis

```
1  Function remove_interactions_x(pivot, nodes, tableau):
2      $X \leftarrow$ X matrix of the tableau;
3      nodes $\leftarrow$ nodes $+$ pivot ;          // Add the pivot to nodes if not present
4      tree $\leftarrow$ Compute the Steiner tree on the connectivity graph;
5      traversal $\leftarrow$ DFS through the Steiner tree with pivot as root;
6      foreach $(p, c) \in$ traversal do
7          if $X_{pivot,p} = 0$ then
               // Apply CNOT with c as control and p as target
8          end
9      end
10     foreach $(p, c) \in$ traversal do
           // Apply CNOT with p as control and c as target
11     end
```

[2]In the actual code-base, this is done by the `networkx`-libary [29].

Algorithm 4: Removal of interactions on the Z-Basis

```
1  Function remove_interactions_z(pivot, nodes, tableau):
2      $Z \leftarrow$ Z-matrix of the tableau;
3      nodes $\leftarrow$ nodes $+$ pivot ;          // Add the pivot to nodes if not present
4      tree $\leftarrow$ Compute the steiner tree on the connectivity graph;
5      traversal $\leftarrow$ DFS through the steiner tree with pivot as root;
6      foreach $(p, c) \in$ traversal do
7          if $X_{pivot,p} = 0$ then
               // Apply CNOT with p as control and c as target
8          end
9      end
10     foreach $(p, c) \in$ traversal do
           // Apply CNOT with c as control and p as target
11     end
```

4.3 Sanitization of Signs

From section 2.3, we note that a Clifford tableau also stores the signs introduced by the term $Y = iZX$ in a sign column r. When synthesizing such a tableau, we must also take care of the signs. We can apply a sequence of H and S gates onto the tableau, which will cancel out the signs, by the vectorized rules we have seen in section 2.3. See algorithm 5 for this process.

4.4 Heuristic Choice of the Pivot

At last, it is required to discuss how to select pivots in our algorithmic structure. Here we first need to note that a pivot needs to be a noncutting vertex on our connectivity graph. This is because we cannot allow interactions using this node after reducing the pivot row in X and Z-Basis. Hence we have to remove it from the graph. Among this removal, the node should not disconnect our graph so that we can synthesize the remaining pivots. Since our primary goal is synthesizing the Clifford tableau with as few CNOTs as possible, we want to choose the row with the minimal Steiner tree. Nevertheless, since the computation of a Steiner-tree is in $\mathcal{O}(q^2)$, where q is the number of qubits and the fact that we have to compute it for every noncutting vertex ($\mathcal{O}(q)$ in the worst-case scenario) this results in a computational overhead of $\mathcal{O}(n^3)$ for choosing the next row, which is undesirable. We hence decided to pre-compute the shortest paths using the Floyd–Warshall algorithm, which provides a lookup table for distances: d. We can then sum up the distances as follows:

$$s_B(r) = \sum_j \begin{cases} d_{r,j} & B_{r,j} \neq 0 \vee r = j \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

Algorithm 5: Removal of interactions on the Z-Basis

```
Function remove_interactions_z(tableau):
    signsz ← Deepcopy of the Z column of the signs of the tableau;
    signsx ← Shallowcopy of the X column of the signs of the tableau;
    for col ← 0 to tableau.n_qubits do
        if $signsz_{col} \neq 0$ then
            Apply H to col;
        end
    end
    for col ← 0 to tableau.n_qubits do
        if $signsz_{col} \neq 0$ then
            Apply S to col;
            Apply S to col;
        end
    end
    for col ← 0 to tableau.n_qubits do
        if $signsz_{col} \neq 0$ then
            Apply H to col;
        end
    end
    for col ← 0 to tableau.n_qubits do
        if $signsx_{col} \neq 0$ then
            Apply S to col;
            Apply S to col;
        end
    end
```

Here B can be the Z or X matrix from the respective basis. Overall we can then compute the heuristic as follows:

$$s(r) = s_X(r) + s_Z(r) \tag{4.2}$$

4.5 The complete elimination process

In the complete process, we can chain the two steps sanitization and removal of interactions in the folllowing way to reduce one column completely:

1. Choose a pivot row r_p
2. Sanitize the X-Basis of r_p
3. Remove the interactions on the X-Basis of r_p
4. Sanitize the Z-Basis of r_p
5. Assure that the reduced X Pivot is one
6. Remove the interactions on the Z-Basis of r_p
7. Assure that the reduced Z Pivot is two and the X Pivot is one of r_p
8. Sanitize the tableau signs

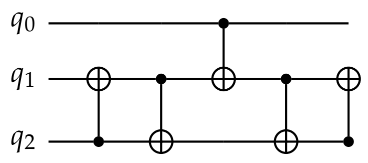
One can observe the pseudocode of the overall process in algorithm 6. Note that we have added two extra steps, which we account to the sanitization process. We can have the edge-case that after sanitizing the Z-Row, the X-Pivot may be three. This can happen when we apply a S-Gate towards the pivot row. After we have sanitized the row in Z-Basis, we can simply undo this by applying another S-Gate towards the pivot qubit. This will not have an effect on the Z-Basis, but will essentially reduce the X-Basis to one again. In the last step we may encounter the same problem. We can fix this by first transforming the S-Gate to a two again (By a sequential application of an H and S Gate analogous to the sanitization process) and next, we can apply an S-Gate in case we encounter a three on the X-Basis again.

4.6 Algorithmic Extension: Allowing Permutations

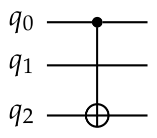
An extension towards synthesizing a Clifford tableau, described above, can be made when we allow n-to-n initial and final swaps on the quantum circuit. Such swaps can be easily removed by permutation of the measurement results from the quantum device. Observe the scenario in Figure 4.1. If we want to re-synthesize this tableau towards a quantum circuit requiring a line architecture, we will yield a circuit with five CNOT-Gates, looking as follows:

Algorithm 6: Clifford Tableau Synthesis

```
1  Function remove_interactions_z(G, tableau):
2      tableau ← tableau$^{-1}$ ;                              // Invert the tableau
3      qc ← A Quantum Circuit, which all operations are appended to;
4      while G has nodes do
           // 1. Choose a pivot row $r_p$
5          $r_p$ ← choose a non cutting vertex from G according to $s(r)$;
           // 2. Sanitize the X-Basis of $r_p$
           // 3. Remove the interactions on the X-Basis of $r_p$
6          sanitize_x($r_p$, tableau);
7          nodes ← Non zero entries of $X_{r_p,:}$;
8          remove_interactions_x($r_p$, nodes, tableau);
           // 4. Sanitize the Z-Basis of $r_p$
           // 5. Assure that the reduced X Pivot is one
           // 6. Remove the interactions on the Z-Basis of $r_p$
           // 7. Assure that the reduced Z Pivot is two and the X Pivot is one
              of $r_p$
9          sanitize_z($r_p$, tableau);
10         if $X_{p_r,p_r} = 3$ then
11             Apply an S gate to $p_r$;
12         nodes ← Non zero entries of $Z_{r_p,:}$;
13         remove_interactions_x($r_p$, nodes, tableau);
14         if $Z_{p_r,p_r} = 3$ then
15             Apply an S gate to $p_r$;
16         if $Z_{p_r,p_r} = 2$ then
17             Apply an H gate to $p_r$;
18         if $X_{p_r,p_r} \neq 1$ then
19             Apply an S gate to $p_r$;
20         Remove $r_p$ from G;
       // 8. Sanitize the tableau sings
21     sanitize_signs(tableau);
22     return qc ;    // Return the quantum circuit which collected the operations
```



This is undesirable, given that if we swap q_1 and q_2 on the connectivity graph, we would only require one CNOT for the same circuit. If we had a Steiner tree with multiple such occurrences, we would introduce multiple unnecessary gates to adjust our circuit towards a bad initial placement of qubits. We can overcome this issue partly heuristically by once computing a Steiner tree and then performing n-times a reverse traversal. If we encounter a parent with a zero and a child with a non-zero node, both swappable, we can swap them. Nevertheless, the reader may note that after applying the CNOT, we can no longer swap the two nodes. Otherwise, one would break the connectivity of the circuit. One can mark the nodes when the first CNOT is applied between them. In this case, we also note a weight of two on the graph's edges connecting the node with the rest of the graph. Initially, all edges of the graph are marked with weights of zero. With this process, one can allow the Steiner-tree approximation to route through "alternative" swappable paths. We took the weight of two for nodes where a CNOT has to be placed since, in the worst case, one has to fill the corresponding element in the tableau and afterward remove the edge. This process of thought will yield algorithm 7. At last, a similar structure can be applied to the choice of the pivot. Here we can scan all swappable nodes regardless of the fact they are cutting or not if and only if we can find a noncutting node in our interaction graph that is swappable. We can then swap the choice of our heuristically-selected row with the noncutting vertex. See algorithm 8 for an outline of the process.



X: $\begin{bmatrix}1 & 0 & 1\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$ Z: $\begin{bmatrix}2 & 0 & 0\\0 & 2 & 0\\2 & 0 & 2\end{bmatrix}$

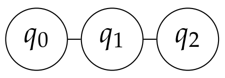


Figure 4.1: Quantum Circuit generating the Clifford tableau in the center plus the respective architecture

Algorithm 7: Extended removal of interactions on the X-Basis

```
1  Function remove_interactions_x(pivot, nodes, tableau, swappable_nodes):
2      $X \leftarrow$ X matrix of the tableau;
3      nodes $\leftarrow$ nodes + pivot ;          // Add the pivot to nodes if not present
4      tree $\leftarrow$ Compute the Steiner tree on the connectivity graph;
5      foreach _ $\in$ tableau.nodes do
6          traversal $\leftarrow$ DFS through the Steiner tree with pivot as root;
7          while traversal do
8              $p, c \leftarrow$ traversal.pop();
9              if $X_{pivot,p} = 0 \wedge p \in swappable\_nodes \wedge c \in swappable\_nodes$ then
10                 Relabel the nodes on the graph and adjust the global permutation;
11             end
12         end
13     end
14     tree $\leftarrow$ Compute the steiner tree on the connectivity graph;
15     traversal $\leftarrow$ DFS through the Steiner tree with pivot as root;
16     foreach $(p, c) \in traversal$ do
17         if $X_{pivot,p} = 0$ then
               // Apply CNOT with c as control and p as target
18         end
19     end
20     foreach $(p, c) \in traversal$ do
           // Apply CNOT with p as control and c as target
21     end
```

Algorithm 8: Clifford Tableau Synthesis

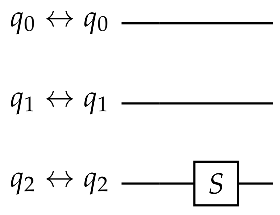
```
1  Function tableau_synth(G, tableau):
2      tableau ← tableau$^{-1}$ ;                         // Invert the tableau
3      perm ← Global permutation qc ← A Quantum Circuit, which all operations are
        appended to;
4      while G has nodes do
          // 1.  Choose a pivot row $r_p$
5         $r_p$ ← choose a non cutting vertex from G according to $s(r)$;
6         if $r_p$ is cutting then
7             $r'_p$ ← Noncutting swappable node from G;
8             Swap $r'_p$ and $r'_p$ on the interaction graph, note the swap on the global
               permutation;
          // 2.  Sanitize the X-Basis of $r_p$
          // 3.  Remove the interactions on the X-Basis of $r_p$
9         sanitize_x($r_p$, tableau);
10        nodes ← Non-zero entries of $X_{r_p,:}$;
11        remove_interactions_x($r_p$, nodes, tableau);
          // 4.  Sanitize the Z-Basis of $r_p$
          // 5.  Assure that the reduced X Pivot is one
          // 6.  Remove the interactions on the Z-Basis of $r_p$
          // 7.  Assure that the reduced Z Pivot is two and the X Pivot is one
              of $r_p$
12        sanitize_z($r_p$, tableau);
13        if $X_{p_r,p_r} = 3$ then
14            Apply an S gate to $p_r$;
15        nodes ← Non zero entries of $Z_{r_p,:}$;
16        remove_interactions_x($r_p$, nodes, tableau);
17        if $Z_{p_r,p_r} = 3$ then
18            Apply an S gate to $p_r$;
19        if $Z_{p_r,p_r} = 2$ then
20            Apply an H gate to $p_r$;
21        if $X_{p_r,p_r} \neq 1$ then
22            Apply an S gate to $p_r$;
23        Remove $r_p$ from G;
      // 8.  Sanitize the tableau sings
24    sanitize_signs(tableau);
25    return qc, perm ;          // Return the quantum circuit which collected the
       operations and its global permutation
```

4.7 Example Execution for one Row

To outline the functionality of our algorithm, we will provide an example execution for the following tableau:

$$\begin{array}{cc} \text{X:} & \text{Z:} \\ \begin{bmatrix} 0 & 0 & 3 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 2 \\ 0 & 2 & 0 \\ 2 & 2 & 0 \end{bmatrix} \end{array} \tag{4.3}$$

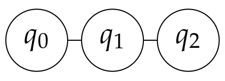
For simplicity, assume that the tableau in Equation 4.3 already describes the inverse. At first, we will pick the row with the least amount of interactions, which in our case is row zero (Denoted as q_0 in the following). We will start with the sanitization of the X-Basis, which yields the row: $\begin{pmatrix} 0 & 0 & 3 \end{pmatrix}$. We will first sanitize the row by applying an S-Gate to q_2:


$q_0 \leftrightarrow q_0$
$q_1 \leftrightarrow q_1$
$q_2 \leftrightarrow q_2$
S

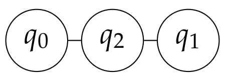
Note that we denoted the global swaps to the left for this quantum circuit. This will produce the following tableau:

$$\begin{array}{cc} \text{X:} & \text{Z:} \\ \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 2 \\ 0 & 2 & 0 \\ 2 & 2 & 0 \end{bmatrix} \end{array} \tag{4.4}$$

The rows we have to compute the Steiner tree for to remove the interactions are $[q_0, q_2]$. This originates from the fact that q_0 is the pivot and q_2 is a non-zero entry. Given the interaction graph:



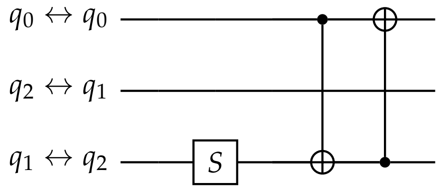
We can observe that the Steiner tree will be the whole graph. Hence, we can find the traversal as $[(q_2, q_1), (q_1, q_0)]$. From this, we can see that our algorithm swaps q_1 and q_2 on the interaction graph, leaving it in the state:



We can hence apply a CX_{q_2,q_0} to the tableau to fill the pivot and one CX_{q_0,q_2}, which will cancel out the remaining interaction. This leaves the tableau in the following state:

$$\begin{array}{cc} \text{X:} & \text{Z:} \\ \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} & \begin{bmatrix} 2 & 0 & 2 \\ 0 & 2 & 0 \\ 0 & 2 & 2 \end{bmatrix} \end{array} \tag{4.5}$$

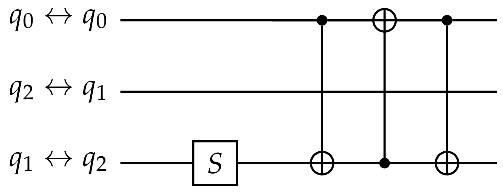
Note in Equation 4.5 that we have automatically reduced column zero in the Z-Basis by reducing row zero in the X-Basis. This holds since all stabilizer and destabilizer states anticommute. The quantum circuit after this operation will be the following:


$q_0 \leftrightarrow q_0$
$q_2 \leftrightarrow q_1$
$q_1 \leftrightarrow q_2$
S

Moving to the Z-Basis, we can note that the row: $\begin{bmatrix}2 & 0 & 2\end{bmatrix}$ consists of twos; hence we do not need to sanitize it. Computing the reverse traversal of on the interaction graph will yield: $[(q_0, q_2)]$, we hence can directly apply a CX_{q_2,q_0} between the two qubits, which will provide us with the final state of the tableau for this row:

$$\text{X:} \quad \begin{bmatrix}1 & 0 & 0\\0 & 1 & 1\\0 & 0 & 1\end{bmatrix} \qquad \text{Z:} \quad \begin{bmatrix}2 & 0 & 0\\0 & 2 & 0\\0 & 2 & 2\end{bmatrix} \tag{4.6}$$

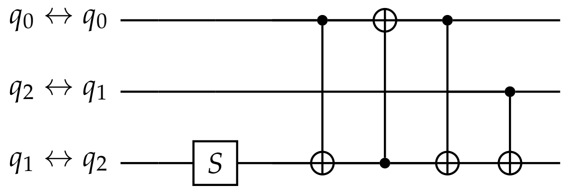
We can find the circuit that reduces this row as:


$q_0 \leftrightarrow q_0$
$q_2 \leftrightarrow q_1$
$q_1 \leftrightarrow q_2$
S

When picking the next pivot, we can see that the score of our interactions will be equal. We hence choose q_1 arbitrarily. We further do not need to perform any sanitization and can entirely focus on the application of the CNOT. In the X-Basis, by taking the row $\begin{pmatrix}0 & 1 & 1\end{pmatrix}$, we can see that this leads towards the reverse-traversal: $[(q_2, q_1)]$. An application of a CNOT: CX_{q_1,q_2}, which will produce the identity tableau:

$$\text{X:} \quad \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} \qquad \text{Z:} \quad \begin{bmatrix}2 & 0 & 0\\0 & 2 & 0\\0 & 0 & 2\end{bmatrix}$$

Since we have already achieved identity, q_3 will have no further effect on the quantum circuit, and we can return the final circuit:


$q_0 \leftrightarrow q_0$
$q_2 \leftrightarrow q_1$
$q_1 \leftrightarrow q_2$
S

4.8 Runtime Analysis

Note that in algorithm 6, we are inverting the Clifford tableau as a first step. This process can be done in $\mathcal{O}(q^3)$ [22]. Next, all sanitizations and the choice of pivot can be done in linear time (we iterate once over the number of qubits): $\mathcal{O}(q)$. Finally, the last and most expensive step is the removal of interactions since this requires the computation of a Steiner tree. Since, generally, this problem is NP-Hard, we relied on heuristic algorithms to approximate the Steiner tree. Specifically, we are approximating it following the implementation in the networkx libary [29], which runs in $\mathcal{O}(|E|\log(|E|)) = \mathcal{O}(q^2\log(q^2))$. At last, since we remove one node from the graph every time on execution, we have to update the distance matrix for our heuristic. We can do this in $\mathcal{O}(q^3)$ using the Floyd-Warshall algorithm. Since we are executing both sanitization and interaction removal once for every node of the Graph, this yields the following runtime:

$$\mathcal{O}(q^3 + q(q^2\log(q^2) + q + q^3)) = \mathcal{O}(q^4) \tag{4.7}$$

5 Architecture Aware Pauli Polynomial Synthesis

In this chapter, we will move toward the synthesis of Pauli-polynomials. Overall, we have provided three algorithms to synthesize the problem. First, an extension towards our work [52] and the work of Gogioso et al. [23], which we will refer to as *synth divide and conquer*. Next, using the architecture-aware synthesis algorithm by van de Griend and Duncan [48] and our architecture-aware synthesis method for Clifford tableaus, we were able to provide an architecture-aware version of the algorithm by Cowtan et al. [14]. At last, we have provided a direct architecture-aware synthesis method for Pauli-polynomials based on the thoughts of van de Griend and Duncan [48]. The following section will outline each method and provide introductory examples of the synthesis attempts.

5.1 Synth divide and conquer

The core idea of Gogioso and Yeung [23] is to remove as many legs as possible from a ZX-Polynomial to the sides, such that we do not require that many CNOTs implement the center part of it. Since most variational ansätze are described by repeating patterns, we can cancel those CNOT regions and further reduce the CNOT count. In our prior work [52], we found that splitting the Pauli-polynomial and trying to ”pull” two-qubit Clifford out shows a more significant effect for larger circuits. We provided a similar approach to our work, which focused on this splitting of Pauli-polynomials as a case study. In the following, we will extend our approach towards Pauli-polynomials. At first, note that overall our approach consists of the following major components:

1. Optimization of the Pauli-polynomial
2. Regrouping of the Pauli-polynomial
3. Splitting of the Pauli-polynomial

5.1.1 Optimization of the Pauli-polynomial

the main goal of the optimization procedure is to remove as many legs as possible of the current Pauli-polynomial. We herefore utilize the propagation rules of Figure 2.8a, Figure 2.8b and Figure 2.8c. We start by assuming the Pauli-polynomial (shortened as PP in equations) is padded with two Clifford regions, C_l and C_r. We can then iterate over all possible combinations of control qubits c and target qubits t and a combination of two-qubit Clifford gates: $g \in [CX, CY, CZ]$. We now want to estimate the increase or decrease in CNOT-Gartes required to implement the Pauli-polynomial for each combination. We refer to this increase or decrease as *effect*, which we will note as $e(c, t, g)$. For Pauli-polynomials, we can estimate this effect by once computing the CNOT-Count of the architecture-aware decomposed circuit representing the Pauli-polynomial:

$$e(PP, c, t, g) = \sum_m e(PP_m, c, t, g) \tag{5.1}$$

Here PP_m describes the m-th gadget of the Pauli-polynomial. We can describe the minimal required CNOTs for synthesizing the Pauli-polynomial as $e(PP_m, c, t, g)$. We can count the CNOTs as outlined in section 2.6. The next factor we have to take into account is the effect on the padded Clifford Regions: $e_{min}(C_l, c, t, g)$ and $e_{min}(C_r, c, t, g)$. We could estimate this by synthesizing the Clifford tableau at every step and counting the number of CNOTs. Nevertheless, this is impractical from a runtime perspective; we hence estimate the upper bound on required CNOTs by the shortest distance between the control and target qubit: $d(c, t) \geq e_{min}(C_l, c, t, g)$ and $d(c, t) \geq e_{min}(C_l, c, t, g)$. Note that this is an loose upper bound on the number of CNOTs applied towards the Clifford since, in the worst case, we have to propagate the Steiner tree once to account for the added CNOT. With this, we can describe the effect a two-qubit Clifford gate will have in terms of the following formula:

$$\begin{aligned} e(c, t, g) &= e_{min}(PP, c, t, g) + e_{min}(C_l, c, t, g) + e_{min}(C_r, c, t, g) \\ &\leq e_{min}(PP, c, t, g) + 2d(c, t) \\ &< 0 \end{aligned}$$

The last statement of this small calculation reflects that we want to *remove* CNOTs from the Pauli-polynomial, i.e., we want to heuristically propagate two-qubit Clifford, which will decrease the effect of CNOTs. We can hence conclude that:

$$2d(c, t) + e_{min}(PP, c, t, g) < 0 \tag{5.2}$$

$$e_{min}(PP, c, t, g) < -2d(c, t) \tag{5.3}$$

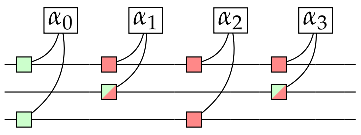
This means that we are only required to compute the effect of propagating the two-qubit Clifford gates through the Pauli-polynomial for optimization. We can then pick the minimal effect among the gates: CX, CY, and CZ, and if this effect is smaller than $2d(c, t)$, where d is the pre-computed distance of the Floyd-Warshall Algorithm, we can propagate the gate and continue. This process is described in algorithm 9.

5.1.2 Regrouping of the Pauli-polynomial

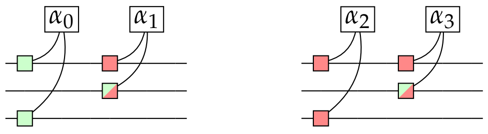
As mentioned earlier, a key element of our algorithm is to split the Pauli-polynomial and further optimize sub-regions. Note that if we do not adjust the Pauli-polynomial correctly, this will lead to suboptimal results. To make this clear assume the following Pauli-polynomial:

Algorithm 9: Optimization process of a Pauli Polynomial

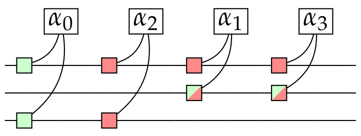
```
Function optimize_pauli_polynomial($c_l, pp, c_r$):
    gates ← $[CX, CY, CZ]$;
    for $c$ ← 0 to $pp.n\_qubits$ do
        for $t$ ← 0 to $pp.n\_qubits$ do
            if $c \neq t$ then
                $g_{min} \leftarrow \arg\min_{g \in \text{gates}} \{e(pp, c, t, g)\}$;
                $e_{min} = e(pp, c, t, g_{min})$;
                if $e_{min} < -2d(c, t)$ then
                    Propagate the gate $g_{min}$ trough the Pauli-polynomial $pp$;
                    Append the gate $g_{min}$ to the left clifford region: $c_l$;
                    Preend the gate $g_{min}$ to the right clifford region: $c_r$;
                end
            end
        end
    end
    return $c\_l$, $pp$, $c\_r$;
```



We can see that if we split it in the middle, we are left with the following two Pauli-polynomials:



We can see that further optimization is not possible among those two Pauli-polynomials. Nevertheless, we can also note that α_1 and α_2 commute; we can hence swap them leaving us with a Pauli-polynomial that allows optimization of CNOTs among splitting:



Heuristically speaking, regrouping to maximize interactions among Pauli-gadgets is sensible. For this purpose, we introduce the concept of *matching-legs*, so legs that are on the same wire, regardless of their type. With this concept, we can compute the following score function for two Pauli gadgets:

- If two legs are on the same wire, we will add 1 to our score
- If no legs are present, we will add -1 to our score
- If there is a mismatch (one leg is present on the wire and one is not), we add -1 to our score

With this score, we can obtain the following order for three Pauli-gadgets a, b, and c:

- Compute the score described above for gadgets a and c: s_{ac}
- Compute the score described above for gadgets a and b: s_{ab}
- The order is then defined as: $b <_a c = s_{ab} < s_{ac}$

In our example above we can see, that $s_{\alpha_0\alpha_1} = -1$ and $s_{\alpha_0\alpha_2} = 1$. We can hence conclude that $\alpha_1 <_{\alpha_0} \alpha_2$. In other words, α_2 is a more desirable neighbor of α_0 than α_1. With this comparison and the fact about commutation, we can obtain an insertion sort-like structure for regrouping the Pauli-polynomial, as outlined in algorithm 10.

Algorithm 10: Regrouping of the Pauli-polynomial

```
Function regroup_pauli_polynomial($PP$):
    col ← 1;
    while $col < n - 1$ do
        $col_p$ ← col − 1;
        $col_c$ ← col;
        $col_n$ ← col + 1;
        while $col_p < n$ and $PP_{col_p} <_{PP_{col_p}} PP_{col_n}$ do
            Swap the two columns;
            $col_p$ ← col;
            col ← $col_n$;
            $col_n$ ← col + 1;
        col ← col + 1
    return $PP$
```

5.1.3 Splitting of the Pauli-polynomial

Since we want to execute our optimization strategy on local regions of the Pauli-polynomial, we developed a divide-and-conquer approach to realize this. We organize our Pauli-polynomial by ″padding″ it with two Clifford regions. Next, we try to ″pull″ out as many two-qubit Clifford as possible from the Pauli-polynomial such that each has a negative effect.

In the next step, we will regroup our Pauli-polynomial and afterward split it, padding the center with an identity Clifford tableau. As a next step, we can recurse on the two subregions that are repeating the pattern described above. We will stop the recursion as soon as we have reached a Pauli-polynomial with two gadgets. In this case, we will optimize the Pauli-polynomial and return it. See Figure 5.1 for a markup of the splitting and algorithm 11 for the respective pseudocode.

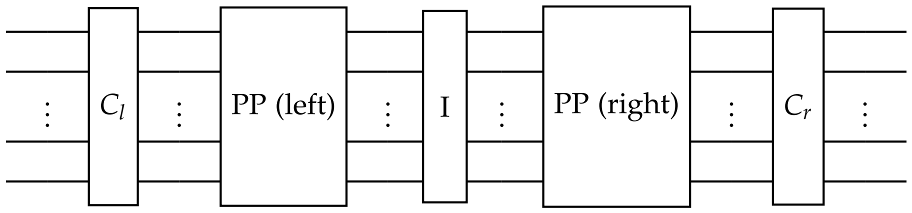


Figure 5.1: Markup circuit for the visualisation process of splitting the Pauli-polynomial (PP) into two subregions: PP (left) and PP (right). I indicates the identity tableau which the two regions are padded with.

Algorithm 11: Divide and conquer approach to synthesize a Pauli-polynomial

```
1  Function divide_and_conquer_synth($pp$):
2      $c_l \leftarrow I$ $c_r \leftarrow I$ $c_l, pp, c_r \leftarrow$ optimize_pauli_polynomial($c_l$, $pp$, $c_r$);
3      regions $\leftarrow$ divide_and_conquer($c_l$, $pp$, $c_r$);
4      $qc \leftarrow$ Convert all regions of the synthesized process to a quantum circuit;
5      return $qc$;
6  Function divide_and_conquer($c_l$, $pp$, $c_r$):
7      if $|pp| \leq 2$ then
8          $c_l, pp, c_r \leftarrow$ optimize_pauli_polynomial($c_l$, $pp$, $c_r$);
9          region $\leftarrow$ Combine $c_l$, $pp$ and $c_r$ to a region;
10         return $region$;
11     end
12     $c \leftarrow I$ // Optimize the current Pauli-polynomial
13     $c, pp, c_r \leftarrow$ optimize_pauli_polynomial($c$, $pp$, $c_r$);
14     $c_l, pp, c \leftarrow$ optimize_pauli_polynomial($c_l$, $pp$, $c$);
       // Regroup the current Pauli-polynomial
15     $pp \leftarrow$ regroup_pauli_polynomial($pp$);
16     $pp_l, pp_r \leftarrow$ Split the Pauli-polynomial;
       // Optimize the subregions
17     region$_l \leftarrow$ divide_and_conquer($c_l$, $pp$, $c$);
18     region$_r \leftarrow$ divide_and_conquer($c$, $pp$, $c_r$);
19     region $\leftarrow$ Combine region$_l$ and region$_r$ such that $c$ only occurrs once;
20     return $region$;
```

5.1.4 Runtime Analysis

In summary, we can note that optimization requires us to iterate over all possible combinations of control and target qubits and then propagate a constant set of two-qubit Clifford (in our case $[CX, CY, CZ]$) through the Pauli-polynomial to estimate the effect. The computational cost of propagation is $\mathcal{O}(nq^2)$; since we execute it for all combinations, we will produce a computational overhead of $\mathcal{O}(nq^4)$ for optimization. Sorting relies on the insertion sort-like structure, where for each comparison, we have to check all of the legs of the Pauli-polynomial; we hence can conclude a complexity of $\mathcal{O}(n^2q)$. So, we have an overall cost of $\mathcal{O}(nq^4 + n^2q)$ per recursion step. To estimate the computational overhead of the whole process, we can define the recurrence relation:

$$T(n) = \begin{cases} nq^4 + n^2q & n \leq 2 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + nq^4 + n^2q & \text{otherwise} \end{cases} \tag{5.4}$$

We can then utilize the Akara-Bazi theorem [3] to obtain the runtime of our algorithm. Noting that $nq^4 + n^2q$ describes the worst-case scenario of our recursion step runtime, we can only obtain an upper bound. We can determine $p = 1$ and assume q as constant since the size of the architecture will not change during execution. This will yield the following upper bound:

$$\begin{aligned} T(n) &\in \mathcal{O}\left(n^1\left(1 + \int_1^n \frac{uq^4 + u^2q}{u^2}\,du\right)\right) \\ &= \mathcal{O}\left(n(1 + q(q^3\log(n) + n - 1))\right) \\ &= \mathcal{O}\left(n + q^4n\log(n) + qn^2 - n\right) \\ &= \mathcal{O}\left(q^4n\log(n) + qn^2\right) \end{aligned}$$

We can hence conclude a runtime of $\mathcal{O}(q^4m\log(m) + qm^2)$, with m, the number of Pauli gadgets in the Pauli-polynomial.

5.2 Architecture-Aware Synthesis of the Unitary Coupled Cluster Ansatz

As mentioned in the future work suggestions by Cowtan et al. [14], an architecture-aware adaption of his algorithm may bring the advantage of reducing unnecessary SWAP-Gates, which are introduced during the routing process for a specific device. We can note that we are already provided with the architecture-aware version of Gray-Synth by van de Griend and Duncan [48] and that our Clifford tableau synthesis also provides this out of the box. In this section, we will focus on the extension of Cowtan et al. [14]'s ansatz in an architecture-aware manner. We will refer to his approach as UCCSD-Synth in the following. Generally speaking, UCCSD-Synth consists of three parts:

- Effective regrouping of the individual gadgets in the Pauli-polynomial into commuting regions
- Diagonalisation of each commuting region
- Synthesis of the Z-Polynomial, which is a result of diagonalization

5.2.1 Regrouping of the individual gadgets

In UCCSD-Synth, we assume that optimizing the CNOT-Count will outweigh the trotter error introduced by possible badly placed Pauli-gadgets. Hence the goal of this step is to regroup the gadgets of the Pauli-polynomial, such that the commuting regions are maximal. Cowtan et al. [14] refer to this as term-sequencing. They realize this process by providing a graph with nodes belonging to all Pauli-gadgets in the Pauli-polynomial. An edge is added in case the gadgets of the two vertices anti-commute. From this process, we can find a set of commuting regions by approximating the graph-coloring problem known as NP-Hard. The different colors then correspond to different commuting sets. As a last step, we can sort the gadgets in the commuting sets, and the commuting sets alphabetically by the type of legs (X-leg, Y-leg, Z-leg, or I-leg). Which will provide us with a set of commuting regions.

5.2.2 Diagonalisation of each commuting region

Next, given a commuting region, we have to diagonalize it. We have already outlined this process in subsection 2.6.3, which we refer the reader to. We adjusted the process slightly by always choosing the two-qubit gates with control and a target of minimal distance to match the architecture better. After diagonalizing the commuting sub-region of the Pauli-polynomial, we can store the operations to the left of the Z-Polynomial in a Clifford-Tableau and optimize them via re-synthesis. Additionally, we can propagate the operations at the right through the rest of the Pauli-polynomial, using the propagation rules from Figure 2.7 and Figure 2.8 collecting them in a Clifford-Tableau at the end.

5.2.3 Synthesis of the Z-Polynomial

As a last step, it remains to synthesize the Z-Polynomial onto a quantum circuit, for which we will utilize the synthesis algorithm by van de Griend and Duncan [48]. We have outlined the functionality in subsection 2.6.4. We want to point out that van de Griend and Duncan [48]'s work stores the CNOTs pulled out in a pure CNOT-Circuit. Again, we can propagate all the CNOT-Gates through the remaining Pauli-polynomial and store them in our final Clifford.

5.2.4 Overall Process

Having identified the major components of synthesizing our Pauli-polynomial, we can combine them into a process. We will first sequence the Pauli-polynomial into different commuting regions. Then, we will first diagonalize each of the k commuting sets. The left part of the Clifford is added towards our final circuit, and the right part is propagated through the Pauli-polynomial. As a last step, we will synthesize the Z-Polynomial of the diagonalization process, apply it towards the circuit, and propagate the remaining CNOTs through the remaining Pauli-polynomial. After doing this for all commuting sets, we can synthesize the final clifford and return the circuit.

5.2.5 Runtime Analysis

Following Cowtan et al. [14], we can note that the process of sequencing requires a computational overhead of $\mathcal{O}(m^2q)$, where m is the number of Pauli-gadgets in our Pauli-polynomial and q is the number of qubits. The diagonalization process runs in $\mathcal{O}(mq^3)$ and the final Clifford synthesis in $\mathcal{O}(q^4)$. At last, it remains to determine the runtime of the synthesis algorithm of van de Griend and Duncan [48]. Here we can determine the steps required by the algorithm by the following two recurrence relations:

$$T_0(n,q) = \begin{cases} 1 & n = 0 \vee q = 0 \\ T_0(n_0, q-1) + T_1(n_1, q) + nq & \text{otherwise} \end{cases} \tag{5.5}$$

$$T_1(n,q) = \begin{cases} 1 & n = 0 \\ T_0(n_0, q-1) + T_1(n_1, q) + nq & \text{otherwise} \end{cases} \tag{5.6}$$

Here n_0 refers to the number of the zeros in a row and n_1 to the length of ones in a row, selected on the specific recursion step. We can see that the most overhead occurs during the selection of a pivot row in both recursion steps. Here we have to count the numbers of zeros and ones for each qubit, which yields a runtime of $\mathcal{O}(nq)$. Generally, determining the steps of the alternating recurrence relations is hard since we must know n_0 and n_1, which are problem specific. Nevertheless, we can find a loose upper bound for both recursions by considering the worst case of always removing just one column from the rows. We can hence find an upper bound for both terms as:

$$T(n) = \begin{cases} 1 & n = 0 \vee q = 0 \\ T(n-1) + nq & \text{otherwise} \end{cases} \tag{5.7}$$

From the previous line of argument, we can conclude that $T_0(n,q) \leq T(n)$ and $T_1(n,q) \leq T(n)$. We can see th¡t this inequality holds true, since in the worst case $n_0 = n - 1$ and $n_1 = 1$ for every step. We can then see that the inequality in Equation 5.7 holds. The closed form of $T(n)$, can then described as:

$$T(n) = \sum_{k=0}^{n} kq \in \mathcal{O}(n^2q) \tag{5.8}$$

We can hence conclude an upper bound of $\mathcal{O}(m^2q)$ on the runtime of *steiner-gray-synth*. We want to point out that according to van de Griend and Duncan [48], their algorithm worked much better on average in practice, which we account for the assumption of reducing one gadget at a time we have taken. Nevertheless, with this knowledge, we can find the overall runtime of the architecture-aware version as follows:

$$\mathcal{O}(m^2q + k(m^2q + q^4 + mq^3) + q^4) = \mathcal{O}(k(m^2q + q^4 + mq^3)) \tag{5.9}$$

5.3 Pauli Steiner Gray Synthesis

After the analysis and implementation of UCCSD-Synth and divide-and-conquer synth, we remarked one particular weak point in the approaches: Both UCCSD and divide-and-conquer require us to split the Pauli-polynomial at positions which may not represent the inner structure of the Pauli-polynomial. On the otherside, a direct extension of the Steiner-gray synth is not feasible since we may introduce new legs at positions we thought we marked as already done, even when assuming n-to-n commutativity. This fact, and the fact that we would have to try every combination of every CNOT at every possible position in-between gadgets, motivated a different viewpoint. In general, the conjecture is that selecting one row and splitting it into legs of certain types is helpful. This is what makes both gray-synth and Steiner-gray-synth so performant on Z-Polynomials. We will add the twist here that instead of propagating Cliffords to the end, we will store them at intermediate positions, which we can combine towards a large quantum circuit. In practice, we will follow the Steiner-gray-synth recursive structure for Pauli polynomials. However, we will compose the circuits from different recursion paths instead of having one global circuit. We can subdivide this process into two major steps:

- The Identity Recursion Step
- The Pauli Recursion Step

At last, note that one base assumption we are making here is that we will have all-to-all connectivity of the Pauli-polynomial regarding commuting gadgets.

5.3.1 Identity recursion step

Here our primary goal is to select a non-cutting row on the connectivity graph that skews our Pauli-polynomial as well as possible. For our heuristic, let us denote the following two functions:

$$max(q) = max_{x\in[I,X,Y,Z]} \left\{ |\{c \in \text{Cols} \quad \text{where } PP_{qc} = x\}| \right\} \tag{5.10}$$

$$min(q) = min_{x\in[I,X,Y,Z]} \left\{ |\{c \in \text{Cols} \quad \text{where } PP_{qc} = x\}| \right\} \tag{5.11}$$

Intuitively, $max(q)$ counts the maximum number of legs of type X, Y, Z, or I, and $min(q)$ is the minimal type. We want to split the Pauli-polynomial, such that we have one of the most prominent regions of X, Y, Z, or I, but we also want to consider smaller regions, which we do not want to be too sparse. With this thought, we can find our heuristic as the difference between $max(q)$ and $min(q)$:

$$i = \underset{q\in\text{Non-cutting Vertices}}{\arg\max} \{max(q) - min(q)\} \tag{5.12}$$

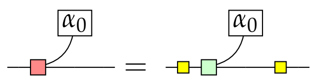
We will hence pick such a row i and then split the Pauli-polynomial into four sets such that each subset only contains a row of X, Y, Z, or I legs. See Figure 5.2, for an example. For the gadgets containing only out-of-identity gates, we can recurse on the identity recurse again, removing the specific qubits. We will recurse on the Pauli Recursion step for X, Y, and Z, noting the row type. Ultimately, we will collect all circuits from the particular recursions, combine them and return them. See algorithm 12 for an outline of the identity recursion step.

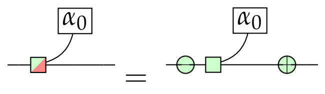
Algorithm 12: Identity Recrusion Step

```
Global: G - Connectivity Graph of the Quantum Device
1  Function identity_recurse(columns, qubits):
2      if $|columns| = 0 \vee |qubits| = 0$ then
3          return An Empty Quantum Circuit;
4      end
5      $G' \leftarrow$ Subgraph of G with qubits;
6      $i \leftarrow \arg\max_{q \in \text{Non-cutting Vertices of } G'} \{max(q) - min(q)\}$;
7      $c_I, c_X, c_Y, c_Z \leftarrow$ Partiton the Pauli-polynomial among $i$;
8      $q_r \leftarrow$ Remove $i$ from qubits;
9      $qc_i \leftarrow$ identity_recurse($c_I$, $q_r$);
10     $qc_x \leftarrow$ precurse_recurse($c_X$, qubits, i, X);
11     $qc_y \leftarrow$ precurse_recurse($c_Y$, qubits, i, Y);
12     $qc_z \leftarrow$ precurse_recurse($c_Z$, qubits, i, Z);
13     return $qc_i + qc_x + qc_y + qc_z$;
```

5.3.2 Pauli Recursion Step

In the Pauli Recursion step, we aim to remove all interactions from the row we have chosen in the identity recursion step (i in the following). We will first select a row by the same heuristic as in Equation 5.12. We will then move on to splitting this row into three sets. One is the identity set, two is the set of the two largest Paulis, and three is the one remaining Pauli. An example split can be observed in Figure 5.3. At first, we will sanitize the row i since we know the Pauli type of this recursion; we can do this by applying a single Clifford gate accordingly to Figure 2.7 in the following way:





For the Z-Case scenario, we do not need to make any modifications. Additionally, this can be applied to the second row containing the individual Paulis. As a last step, we must sanitize the regrouped regions with the two paulis. For this, we can note that the legs in this group are either one of $\{[X,Y],[X,Z],[Z,Y]\}$. We will assume Z, Y as our standard basis; we first aim to convert X, Y legs and X, Z legs into the Z, Y basis. Note that, a Y-leg is invariant towards

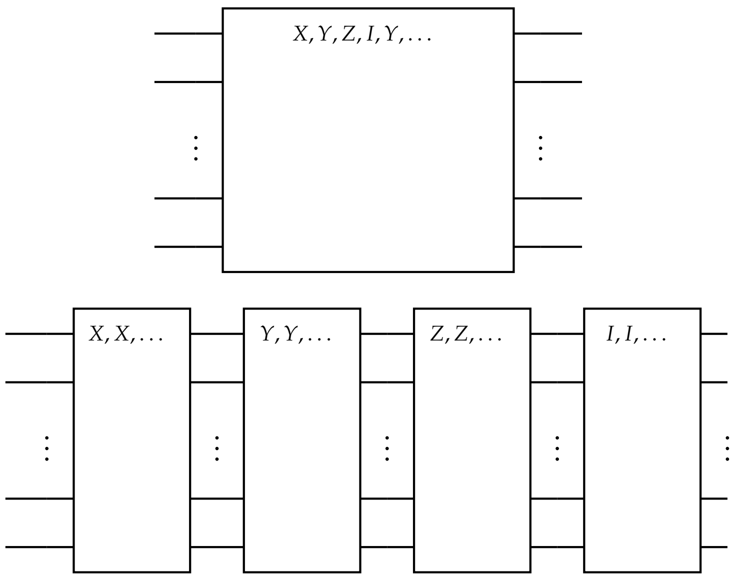


Figure 5.2: Sample process of splitting the Pauli-polynomial during the identity-Recursion step.

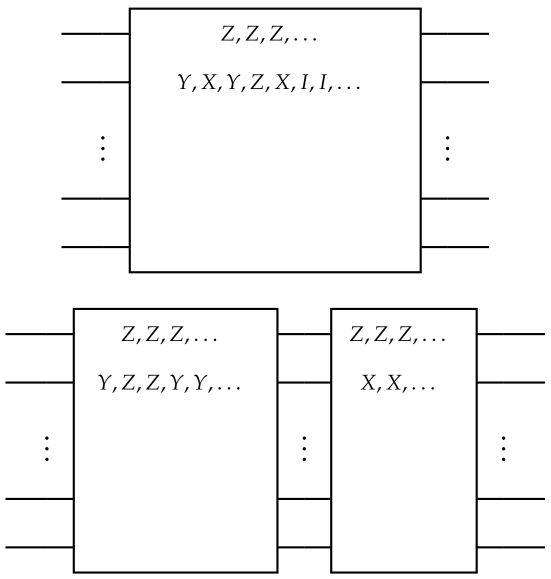
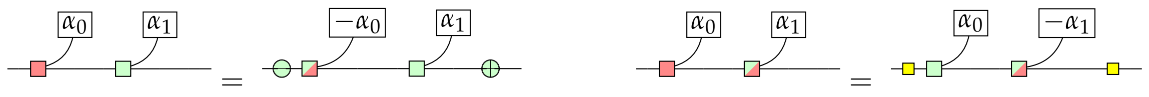
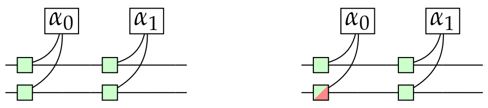


Figure 5.3: Sample process of Splitting the Pauli-polynomial during the Pauli-Recursion step.

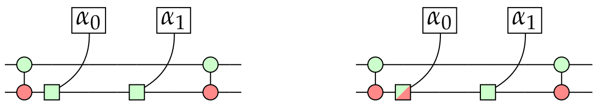
the H-Gate according to Figure 2.7a but will change an X-leg into a Z-leg. Analogous a Z-leg is invariant towards the V-Gate, which will change an X-leg towards a Y-leg. Hence we can hence convert X, Y legs and X, Z legs into Z, Y-legs as follows:



We can then note that all gadgets among the two rows will have one of the following forms:



We can here reduce the row i to identity by applying one CNOT as follows to both of the regions:



The last edge case one has to consider is the column with identity legs on the second qubit. Here we cannot reduce the row i; we will sanitize it and swap it with the next row by applying two CNOTs. Afterward, we will continue with a Z-recurse on the second row. If the second row is non-cutting, we can perform a base-recursion step onto this row and remove it from the number of qubits. See Algorithm algorithm 13 for an outline of the pauli recursion step.

5.3.3 Runtime Analysis

Similarly, in our analysis of van de Griend and Duncan [48], we can find the required steps by the following recurrence relation:

$$T_I(n,q) = \begin{cases} 1 & n = 0 \vee q = 0 \\ T_P(n_x, q) + T_P(n_Y, q) + T_P(n_Z, q) + T_P(n_I, q-1) + nq & \text{otherwise} \end{cases} \tag{5.13}$$

$$T_P(n,q) = \begin{cases} 1 & n = 0 \vee q = 0 \\ T_P(n_{ZY}) + T_I(n_I) + T_P(n - n_{ZY} - n_I) + nq & \text{otherwise} \end{cases} \tag{5.14}$$

Here n_X, n_Y, n_Z, n_I and n_{ZY} describe the sizes of the individual split. We can again state, that in terms of runtime, the worst-case scenario will occur when we split into two large sets of sizes 1 and $n-1$. We can hence find the following recurrence relation again as an upper bound:

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + nq & \text{otherwise} \end{cases} \tag{5.15}$$

We can find again the closed form solution as:

$$T(n) = \sum_{k}^{n} qk \in \mathcal{O}(n^2 q)$$

Algorithm 13: Pauli Recursion Step

```
Global: G - Connectivity Graph of the Quantum Device
Function precurse_recurse(columns, qubits, i, type):
    if $|columns| = 0$ then
        return An Empty Quantum Circuit;
    end
    $G' \leftarrow$ Subgraph of G with qubits;
    $j \leftarrow \arg\max_{q \in \text{Neighbouts of i in } G'} \{max(q) - min(q)\}$;
    $c_I, c_{ZY}, c_O \leftarrow$ Partiton the Pauli-polynomial among $j$;
    $qc_s \leftarrow$ Empty Quantum Circuit;
    $c_{ZY}, c_O \leftarrow$ Sanatize all Columns by applying the respective single cliffords note them
     in $qc_s$;

    $q_r \leftarrow$ Remove $j$ from qubits;
    if $j$ is noncutting in $G'$ then
        $q'_r \leftarrow$ Remove $j$ from qubits;
        $qc_i \leftarrow$ identity_recurse($c_I$, $q'_r$);
    end
    else
        $qc_i \leftarrow CX_{j,i} + CX_{i,j}$;
        Propagate the two CNOTs trought the Pauli-polynmoial;
        $qc_i \leftarrow$ Check for single legged gadgets and add them to the Pauli-polynomial;
        $qc_i \leftarrow$ identity_recurse($c_I$, $q_r$);
        $qc_i \leftarrow CX_{i,j} + CX_{j,i}$;
    end

    $qc_o \leftarrow CX_{i,j}$;
    Propagate the CNOT trough the Pauli-polynomial;
    $qc_o \leftarrow$ Check for single legged gadgets and add them to the Pauli-polynomial;
    $qc_o \leftarrow$ precurse_recurse($c_O$, qubits, j, Z);
    $qc_o \leftarrow CX_{i,j}$;

    $qc_{zy} \leftarrow CX_{i,j}$;
    Propagate the CNOT trough the Pauli-polynomial;
    $qc_{zy} \leftarrow$ Check for single legged gadgets and add them to the Pauli-polynomial;
    $qc_{zy} \leftarrow$ precurse_recurse($c_{ZY}$, qubits, j, Z);
    $qc_{zy} \leftarrow CX_{i,j}$;

    return $qc_s + qc_i + qc_{zy} + qc_o + qc_s^{-1}$;
```

Hence we can conclude an upper bound of $\mathcal{O}(m^2q)$ with m-the number of Pauli-gadgets in the Pauli-polynomial.

6 Experiments

We ran numerical experiments to analyze and verify our algorithms' performance. We once evaluated the two-qubit gate count and fidelity of our Clifford-Synthesis algorithms in our experiments. Second, we put further emphasis on evaluating the synthesis of Pauli-polynomials. For this, we will first provide a qualitative overview of our algorithms, comparing them to the approaches of Cowtan et al. [13, 14]. Next, we will execute all algorithms on a set of random Pauli-polynomials to further assess their capabilities and scaling behavior, and at last, we took the operators of Cowtan et al. [14], which provide a real-world example from our point of view, to confirm the results we have seen on random Pauli-polynomials. Overall we will examine this once assuming a complete architecture to outline the optimization capabilities while also providing data for routed circuits to show the performance of architecture-aware synthesis in this context.

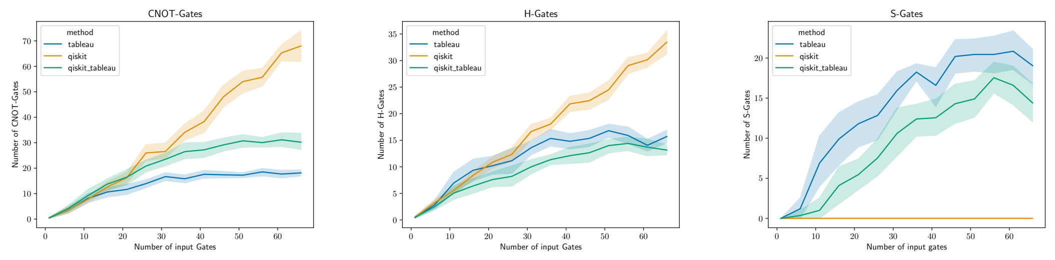
6.1 Clifford Experiments

We executed the tableau synthesis on different architectures, available in the IBM-Software stack [1]. Specifically the backends: *quito*, *guadalupe* and *mumbai*. See **??** for an outline of their connectivity graphs and qubit count. At first, we examined the Gate Count for the various architectures and continued by executing a subset of circuits with 25 gates on the IBM Quito device.

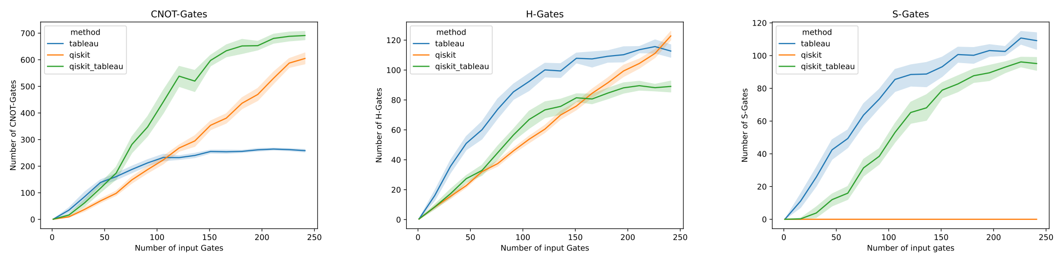
6.1.1 Evaluation of the Gate Count

Therefore, we generated random circuits with the gate set of $\{H, S, CX\}$ since this gate set generates the Clifford group. We varied the gate count, while the number of qubits was fixed by the IBM-Backend. All gates were picked uniformly at random, which means that in our experiments, single qubit gates are more likely to be present in the quantum circuit than CNOTs. At last, we sequentially applied the Cliffords towards a Clifford tableau, which we synthesized once by our approach (referenced as *tableau* in the following). Once with the approach of Bravyi et al. [9], (denoted as *qiskit_tableau* in the following), which is implemented natively in the qiskit library and is supposed to outperform the algorithm of Aaronson and Gottesman [1]. At last, we once transpiled and routed the original circuit using the qiskit `transpile` method (we will refer to this as *qiskit* in the following) towards a gate-set of $\{H, S, CX\}$ and the specific architecture. We did the same for the tableau synthesis of Bravyi et al. [9]. One can find the evaluation results in the Figure 6.1. We can note that upon

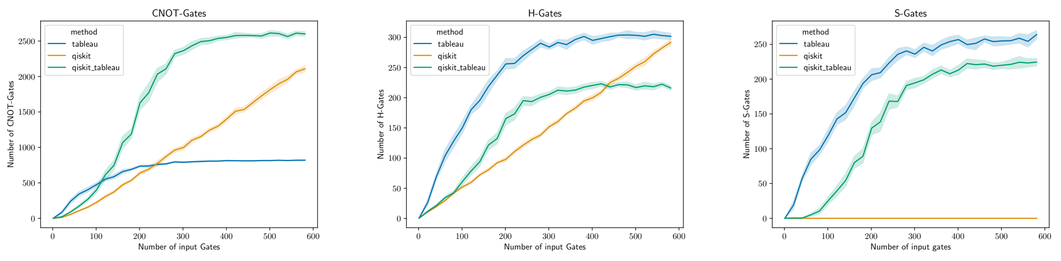
[1]In our implementation, we used qiskit version `0.39.0`, qiskit-aer `0.11.0`. qiskit-ibm-runtime `0.8.0` and qiskit-ibm-provider `0.19.2`. The experiments were conducted on July 16th.



(a) Quito (5 Qubits)



(b) Guadalupe (16 Qubits)



(c) Mumbai (27 Qubits)

Figure 6.1: Comparison of the three different approaches: textitqiskit_tableau, *tableau* and *qiskit*. With respect to different architectures for randomly generated circuits, with a gateset of H, S, CX.

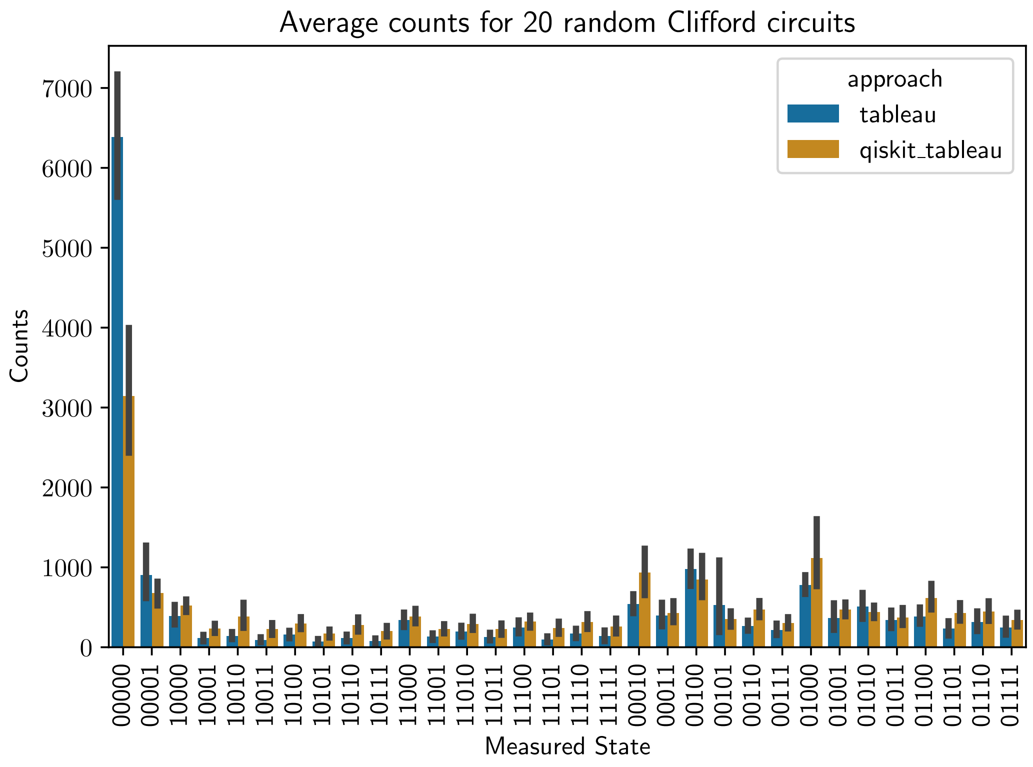


Figure 6.2: Average Counts of our simulation of $CC^\dagger$ for 20 random clifford gates.

convergence, our methods provide an improvement in terms of CNOT-Gates ($^{20}/_{30} \approx 0.66$ for Quito, $^{250}/_{700} \approx 0.35$ for Guadalupe and $^{900}/_{2500} \approx 0.36$ for Mumbai). Nevertheless, we tend to increase both H-Gates and S-Gates, which accounts for the fact that we must sanitize our row at every step. At last, we want to point the reader towards the gap we can observe before convergence when comparing our method to the standard compilation process of Qiskit. We can see that the larger our circuit and architecture grow, the larger this gap grows. So overall we can conclude, that before convergence our algorithms tend to introduce additional CNOTs, compared to other optimization techniques.

6.1.2 Evaluation on Real hardware

To verify our conjecture that reducing the CNOT count will improve the final fidelity of our circuit, we executed random clifford circuits on real hardware. To provide a reasonable estimation of the final fidelity, we point out that if a final state is fully entangled, i.e., the amplitudes all have the same value, it is harder to estimate the influence of noise. Therefore we executed once the produced Clifford circuit and once its inverse on the IBM-Hardware, which should precisely provide us with the input state: $|0\dots0\rangle$, which we can measure the fidelity on. We generated 20 random circuits C and executed $CC^\dagger$ on the quantum device with

16.000 shots. We here compared *qiskit* towards our algorithm. In Figure 6.2, one can see the averaged counts for the 20 experiments. This picture produced a Helliner-fidelity of ≈ 0.1967 for *qiskit_tableau* and a Hellinger-fidelity of ≈ 0.3992 for *tableau*. At last, we computed the correlation between the H, S, and CX-Gates and the Hellinger fidelity for our approach. We here solely relied on our approach since the transpile method of qiskit introduces other gates, such as the V-Gate [2]. Here we found a correlation of ≈ -0.21 for the H-Gates, a correlation of ≈ -0.18 for the S-Gates, and a correlation of ≈ -0.50 for the CX-Gates. From this evaluation, we can conclude that CNOT-Gates will contribute the most towards reducing the fidelity, which strengthens our conjecture of reducing as many CNOT-Gates as possible. At last, we want to focus on the execution time of the individual quantum circuits on the device. Here we found an execution time of *qiskit* of $\approx 7.13 \pm 0.13$ and an execution time of *qiskit_tableau* of $\approx 6.99 \pm 0.08$. We can note that at first, our execution time is, on average less than the one of *qiskit* plus additionally shows a better standard deviation, which we again can account for the fact that we reduced the number of CNOTs describing the Clifford tableau.

6.2 Pauli Polynomial Experiments

For the analysis of Pauli Polynomials in our experiments, we will try to answer the following questions regarding our developed algorithms:

- What are the *analytical* key differences among the approaches?
- How large is the trotterization error of our approaches?
- How do our algorithms perform in optimization concerning a fully connected optimization strategy and routing?
- How do our algorithms perform on real data?

To answer those questions, we will utilize the set of molecules, Cowtan et al. [14] has provided as a real-world example [3]. We will reference those as the *molecule dataset*. In the following, we will answer the first point by providing a comparison with the approach of Cowtan et al. [14]. We will then compare the trotterization error of our approach with the one introduced by Cowtan et al. [14]. Next, we will compare the performance of our algorithms among random Pauli-polynomials. At last, we will perform optimization with *pauli-steiner-gray-synth* and architecture-aware-UCCSD-set synth for a complete, line and circle architecture on the molecule dataset, comparing it to both algorithms of Cowtan et al. [14] with introduced routing. Note that due the limited capabilities of NISQ-Devices a evaluation of Pauli-polynomials on real devices was unfeasible Since in a prior test to our evaluations we reported a fidelity of ≈ 0.0001, which cleary cannot provide clear statements about the optimization technique.

[2]Note, that since our approach only covers certain clifford-circuits, those results might not be generally true.
[3]See: https://github.com/CQCL/tket_benchmarking/tree/master/compilation_strategy

6.2.1 Analytical Overview

Here we will analyze *synth-divide and conquer* (SD&C), *A-UCCSD-set* (architecture-aware-UCCSD-set), *PSGS* (pauli-steiner-gray-synth) combined with the two approaches provided by Cowtan et al. [14]. Specifically, we will reference those as *UCCSD-set* for the set-based approach and *UCCSD-pair* for the pair-based approach. Overall we will look at the runtime of the different approaches if they require the assumption of n-to-n commutativity. Our comparison can be found in Table 6.1. Note that we have denoted the number of gadgets as m and the number of qubits as q. Overall, we have two algorithms, UCCSD-pair and

Algorithm	**Runtime**	**n-to-n commutativity?**
SD&C	$\mathcal{O}(q^4 m \log(m) + qm^2)$	no
A-UCCSD-set	$\mathcal{O}(k(m^2q + q^4 + mq^3))$	yes
PSGS	$\mathcal{O}(m^2q)$	yes
UCCSD-set	$\mathcal{O}(m^2q + mq^3 + mq^3)$[14]	yes (due effectiveness of gray-synth)
UCCSD-pair	Unknown[13]	no

Table 6.1: Analytical comparison among the algorithms used trough-out this work.

SD&C, which do not generally assume n-to-n commutativity among Pauli-gadgets; we can hence provide a fair comparison between those two methods. We placed UCCSD-Set into the category of assuming n-to-n commutativity; we account for this towards the fact that their main optimization step relies on performing gray-synth on as large Z-Polynomials as possible. The structure of PSGS requires n-to-n commutativity, which the user cannot influence. It hence is important to evaluate the trotterization error of this algorithm. We can report PSGS's runtime as the best, compared with A-UCCSD-set and UCCSD-set. For UCCSD-pair, we could not report any runtime. We hence left the comparison with SD&C out of the scope of this discussion.

6.2.2 Trotterization Error

To evaluate the development of the introduced trotterization error of our approaches, we will follow the evaluations of Gui et al. [27]. The core idea is to provide a Hamiltonian, with already fixed prefactors among the various pauli-gadgets. For instance, assume the following:

$$H = IZ + ZI + ZZ + XX + YY$$

We will then record the ordering produced by our algorithms, which we can determine by the order of application of the R_z-Gates of the various phases. Let us assume this produces the (permutated) Hamiltonian:

$$H = XX + ZI + YY + IZ + ZZ$$

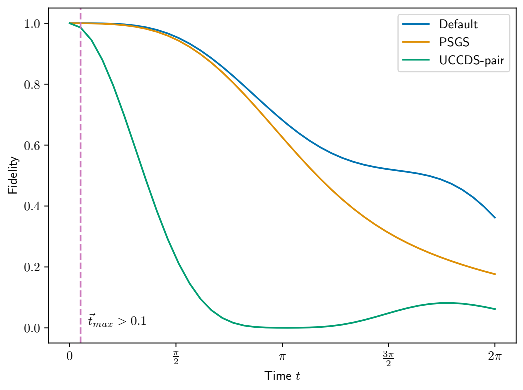
Introducing a time variable, t, we can compute the Hamiltonian simulation $U_{exact} = \exp\left(-i\frac{t}{2}H\right)$. We can then make conjectures about the trotterization error by plotting the Fidelity $F(U_{exact}, U_{circuit})$, between the exact unitary and the unitary of our produced trotterized circuit $U_{circuit}$, parametrized with t. Since the work of Gui et al. [27] focused on Hamiltonian simulation, our results here are to be checked with care since, naturally, UCCSD focuses on finding the ground state of a molecule. Nevertheless, we found this is a straightforward way to provide statements about the trotterization error one introduces into his algorithm using our synthesis methods. For a setup, we took the molecules: *H2_P_631g*, *H4_P_sto3g* and *LiH_P_sto3g* from the molecules dataset. We focused on molecules encoded in the P basis since the prefactors were already provided within the Hamiltonians. We once computed U_{exact} analytically within the qiskit framework and simulated the unitary of the circuit PSGS provided [4] by using the `unitary_simulator` of the qiskit package. For UCCSD set, we employed the sequencing strategies of Cowtan et al. [14], but we did not apply any routing or further optimization processes. We also simulated the circuit using the `unitary_simulator` of qiskit. At last, we have provided the trotterization error for the Hamiltonian with no further ordering as a baseline. See Figure 6.3 for our results. Overall we can see that the regrouping we provided is quite effective on all three molecules we used. Nevertheless, in the end, we tend to decay faster than the default setting, meaning that we will indeed be outperformed by regrouping algorithms like Gui et al. [27]. To verify Cowtan et al. [14]'s conjecture that the trotterization error will be small, we marked the position, where the time times the maximum value of the absolute of the interaction strength will be greater than $0.1 \ll 1$. We found that in this interval, all approaches continued a high degree of fidelity; hence we can confirm the conjecture based on our experiments. We can also see that Cowtan et al. [14]'s approach will degenerate rapidly afterward. At the same time, PSGS can preserve a high fidelity longer.

6.2.3 Optimization Capabilities

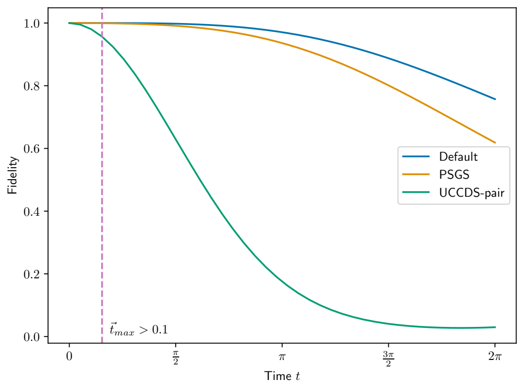
To measure the optimization capabilities, we once evaluated the synthesis on random Pauli-polynomials [5] on 6 and 8 qubits with pauli-gadgets in a range of [10, 20, 30, 40, 50, 70, 90, 100, 200, 300, 500, 1000]. We synthesized every Pauli-polynomial on a line, square, and circle architecture and counted the CNOTs among them. For the algorithms UCCSD-set and UCCSD-pair, we used the routing procedure of tket [45]. We counted the SWAP gates introduced by tket as three CNOTs in our evaluations due to their definition. We then measured the CNOT-Count as $100 \cdot {}^{(\#CX_{naive} - \#CX_{out})}/_{\#CX_{naive}}$ in percent. Here $\#CX_{naive}$ is the CNOT-Count produced by naive architecture-aware decomposition of the Pauli-gadget and $\#CX_{out}$, the CNOT-Count provided by the quantum circuit. See Figure 6.4 for an outline of the performance among a complete architecture. Overall, the algorithms which assumed n-to-n commutativity performed the best. Among those, PSGS provides the best ratio for larger Pauli-polynomials, followed by the A-UCCSD-set. Nevertheless, for smaller Pauli-polynomials, we can see that this trend discontinues, and the A-UCCSD-set even worsens the CNOT-reduction, meaning it introduces additional CNOTs to the circuit. Among the ones, which do not assume n-to-n commutativity, we can see that SD&C is quite bad concerning

[4]Note, that we have omitted SD&C, UCCSD-pair, and A-UCCSD-set since the prior two do not expect n-to-n commutativity and the latter is an extension of UCCSD-set, i.e., we expect a similar trotterization error.

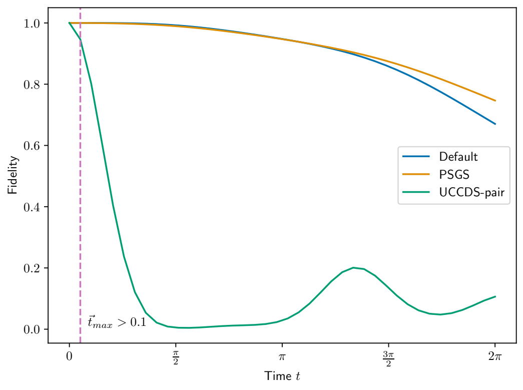
[5]We limited the legs between one and the number of qubits. They where choosen uniformly at random.



(a) Fidelity for H2_P_631g, in the range $[0; 2\pi]$. The vertical line indicating the position where $t \cdot \vec{t_{max}} > 0.1$. Max Interaction: ≈ 0.91

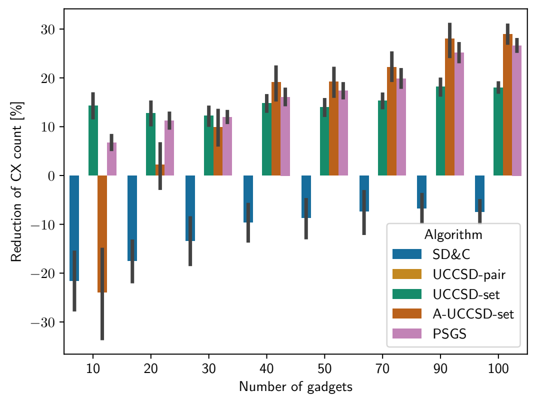


(b) Fidelity for H4_P_sto3g, in the range $[0; 2\pi]$. The vertical line indicating the position where $t \cdot \vec{t_{max}} > 0.1$. Max Interaction: ≈ 0.21

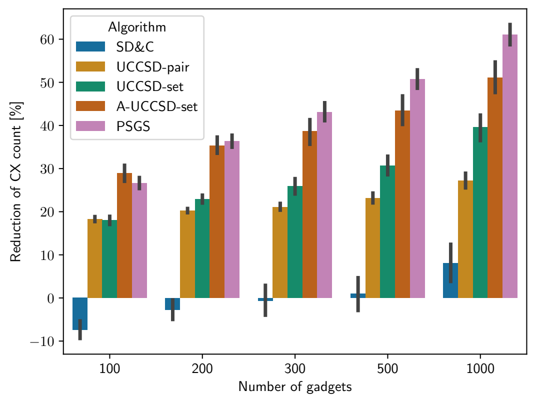


(c) Fidelity for LiH_P_sto3g, in the range $[0; 2\pi]$. The vertical line indicating the position where $t \cdot \vec{t_{max}} > 0.1$. Max Interaction: ≈ 1.0

Figure 6.3: Time evolution for the molecules: H2_P_631g, H4_P_sto3g and LiH_P_sto3g in the interval $[0; 2\pi]$.The first part of this naming scheme introduces the molecule name. The second encoding strategy we applied for formulating the fermionic annihilation and creation operators towards a digital quantum device, which can be either P, JW, or BK. The last part of the naming scheme describes the basis set used for the molecule.



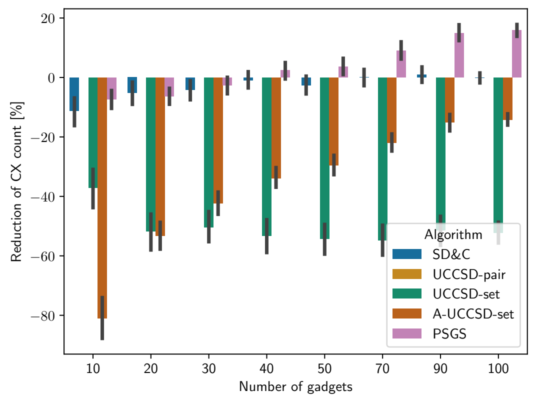
(a) Random Pauli-polynomials with pauli-gadgets in [10, 20, 30, 40, 50, 70, 90, 100].



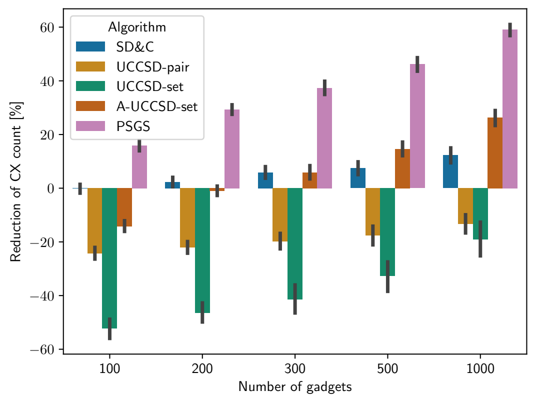
(b) Random Pauli-polynomials with pauli-gadgets in [100, 200, 300, 500, 1000].

Figure 6.4: Evaluation of random Pauli-polynomials for a complete architecture. Once a subset of smaller Pauli-polynomials and once one of larger once.

optimization on a complete architecture and gets outperformed by UCCSD-pair. The fact that both SD&C and A-UCCSD-set utilize clifford tableaus for synthesis and perform poorly could indicate that we are applying to few gates towards the Clifford, leaving too sparse, such that classical optimization would perform better - something we have seen in section 6.1. We obtained quite different results when averaging the CNOT-Reduction throughout the three architectures, line, square, and circle. See Figure 6.5 for our results there. Overall it shows that



(a) Random Pauli-polynomials with pauli-gadgets in [10, 20, 30, 40, 50, 70, 90, 100].



(b) Random Pauli-polynomials with pauli-gadgets in [100, 200, 300, 500, 1000].

Figure 6.5: Evaluation of random Pauli-polynomials for a line, square and circle architecture. Once a subset of smaller Pauli-polynomials and once one of larger once.

due to the placement of CNOTs in the routing process, both UCCSD-set and UCCSD-pair tend to perform worse. We can also see that PSGS performs the best, and among the algorithms assuming n-to-n commutativity. For the once which will preserve commutativity, SD&C

shows not worsen the CNOT-Count but does not improve it. Interestingly UCCSD-pair get shows better scaling behavior than UCCSD-set but converges at a reduction factor of $\approx -20\%$. We can conclude from those plots that architecture-aware synthesis, especially for Pauli-polynomials, can significantly affect the CNOT-Count executed on the quantum device.

6.2.4 Synthesis of Molecules

At last, it remains to test our algorithms on real-world examples. We, therefore, used the molecule dataset and compared UCCSD-set, A-UCCSD-set, and PSGS against the various molecules. We again routed UCCSD-set using tket and counted the swap gates introduced by the routing process by three CNOTs. We once executed the algorithms for a line, circle, and complete architecture, allowing further insights into our algorithms' optimization process and routing capabilities. At last, we have to provide the disclaimer that since our algorithms were prototyped in Python, we ran into scaling issues for larger molecules, so molecules with a qubit size larger than 15. We hence limited ourselves to those molecules. See Table 6.2 for a comparison among a complete architecture, Table 6.4 for a comparison among a line and Table 6.3 for a circle architecture. We can, overall, observe that PSGS outperforms both A-UCCSD-set and UCCSD-set. Additionally, the UCCSD set tends to worsen after being routed toward a specific architecture, which we attribute to the fact that during the routing process, additional CNOTs are introduced - something PSGS can avoid. Overall, our results from the previous section are confirmed; PSGS tends to perform worse on average compared to UCCSD-set for smaller Pauli-polynomials but increases its capabilities when used on larger ones. A-UCCSD-set even tends to outperform PSGS, but we found the best strategy for smaller molecules to be UCCSD-set with routing. At last, we can see that while PSGS is highly performant in reducing the CNOT-Count, the depth reduction for UCCSD-set and A-UCCSD-set performs better.

name	n_qubits	gadgets	UCCSD-set (cx)	UCCSD-set (depth)	PSGS (cx)	PSGS (depth)	A-UCCSD-set (cx)	A-UCCSD-set (depth)
H2_BK_sto3g	4	12	18 (-60.87)	41 (-45.33)	28 (-39.13)	56 (-25.33)	22 (-52.17)	50 (-33.33)
H2_BK_631g	8	84	180 (-66.04)	334 (-56.45)	306 (-42.26)	485 (-36.77)	245 (-53.77)	311 (-59.45)
H4_BK_sto3g	8	160	494 (-57.85)	740 (-52.35)	508 (-56.66)	830 (-46.56)	451 (-61.52)	516 (-66.77)
LiH_BK_sto3g	12	640	2004 (-67.78)	2912 (-61.56)	2462 (-60.42)	3723 (-50.86)	3621 (-41.78)	3467 (-54.24)
NH_BK_sto3g	12	640	2164 (-64.43)	3014 (-59.87)	2456 (-59.63)	3744 (-50.15)	2950 (-51.51)	2866 (-61.84)
H2O_BK_sto3g	14	1000	4045 (-64.60)	5305 (-60.69)	4456 (-61.00)	6440 (-52.29)	6281 (-45.03)	5669 (-58.00)
CH2_BK_sto3g	14	1488	6620 (-59.19)	7540 (-61.25)	6056 (-62.67)	8795 (-54.80)	9688 (-40.28)	8559 (-56.02)
BeH2_BK_sto3g	14	1488	6313 (-62.12)	7272 (-63.07)	6462 (-61.23)	9487 (-51.82)	10907 (-34.56)	9559 (-51.45)
H2_JW_sto3g	4	12	30 (-53.12)	57 (-41.84)	38 (-40.62)	78 (-20.41)	24 (-62.50)	52 (-46.94)
H2_JW_631g	8	84	198 (-71.22)	361 (-61.39)	370 (-46.22)	618 (-33.90)	232 (-66.28)	289 (-69.09)
H4_JW_sto3g	8	160	393 (-70.05)	502 (-71.72)	594 (-54.73)	1026 (-42.20)	397 (-69.74)	479 (-73.01)
NH_JW_sto3g	12	640	1525 (-78.14)	2015 (-77.17)	2090 (-70.04)	3512 (-60.21)	2483 (-64.41)	2354 (-73.33)
LiH_JW_sto3g	12	640	1559 (-77.65)	2046 (-76.85)	2518 (-63.90)	4120 (-53.38)	2346 (-66.37)	2207 (-75.03)
H2O_JW_sto3g	14	1000	2649 (-78.36)	3595 (-76.27)	3150 (-74.26)	5269 (-65.22)	4372 (-64.28)	3926 (-74.08)
BeH2_JW_sto3g	14	1488	4285 (-76.47)	4989 (-77.89)	5846 (-67.89)	8996 (-60.12)	7798 (-57.17)	6781 (-69.94)
CH2_JW_sto3g	14	1488	4052 (-77.75)	4728 (-79.04)	5174 (-71.58)	8169 (-63.79)	7876 (-56.74)	6815 (-69.79)
H2_P_sto3g	2	4	4 (0.00)	20 (122.22)	4 (0.00)	9 (0.00)	4 (0.00)	9 (0.00)
H2_P_631g	6	158	264 (-68.42)	434 (-62.59)	350 (-58.13)	636 (-45.17)	280 (-66.51)	283 (-75.60)
H4_P_sto3g	6	164	268 (-68.54)	445 (-62.73)	364 (-57.28)	666 (-44.22)	258 (-69.72)	296 (-75.21)
NH_P_sto3g	10	630	1929 (-61.08)	2303 (-62.27)	1556 (-68.60)	2703 (-55.72)	2113 (-57.36)	2125 (-65.19)
LiH_P_sto3g	10	630	1929 (-61.08)	2303 (-62.27)	1556 (-68.60)	2703 (-55.72)	2113 (-57.36)	2125 (-65.19)
H2O_P_sto3g	12	1085	3567 (-64.52)	4095 (-65.61)	2742 (-72.73)	4573 (-61.59)	4353 (-56.70)	3880 (-67.41)
BeH2_P_sto3g	12	1085	3723 (-63.41)	4172 (-65.22)	2576 (-74.68)	4067 (-66.10)	4380 (-56.95)	3855 (-67.87)
CH2_P_sto3g	12	2109	6521 (-67.08)	7693 (-67.12)	4752 (-76.01)	8488 (-63.72)	8409 (-57.54)	7400 (-68.37)
H4_P_631g	14	2912	8234 (-73.12)	9682 (-72.64)	9368 (-69.42)	14192 (-59.90)	11397 (-62.79)	9098 (-74.29)
H8_P_sto3g	14	5792	16820 (-72.50)	20014 (-71.69)	11516 (-81.17)	21199 (-70.01)	25044 (-59.05)	20154 (-71.49)
NH3_P_sto3g	14	5792	16820 (-72.50)	20014 (-71.69)	11516 (-81.17)	21199 (-70.01)	25044 (-59.05)	20154 (-71.49)

Table 6.2: Synthesis of the molecule dataset with the algorithms UCCSD-set, PSGS, A-UCCSD-set on a complete architecture.

name	n_qubits	gadgets	UCCSD-set (cx)	UCCSD-set (depth)	PSGS (cx)	PSGS (depth)	A-UCCSD-set (cx)	A-UCCSD-set (depth)
H2_BK_sto3g	4	12	30 (-48.28)	47 (-44.71)	60 (3.45)	91 (7.06)	39 (-32.76)	65 (-23.53)
H2_BK_631g	8	84	484 (-47.51)	543 (-47.79)	580 (-37.09)	729 (-29.90)	746 (-19.09)	679 (-34.71)
H4_BK_sto3g	8	160	1584 (-15.57)	1232 (-42.02)	1128 (-39.87)	1488 (-29.98)	1466 (-21.86)	1104 (-48.05)
LiH_BK_sto3g	12	640	9325 (-18.94)	5634 (-53.36)	5250 (-54.36)	6419 (-46.87)	11472 (-0.28)	6516 (-46.06)
NH_BK_sto3g	12	640	10686 (-10.50)	5840 (-49.64)	5696 (-52.29)	6580 (-43.26)	10133 (-15.13)	5623 (-51.51)
H2O_BK_sto3g	14	1000	22679 (2.83)	11988 (-42.25)	8704 (-60.53)	10539 (-49.23)	23130 (4.88)	12189 (-41.28)
CH2_BK_sto3g	14	1488	41591 (23.53)	17390 (-46.07)	13964 (-58.53)	16069 (-50.16)	30516 (-9.37)	14691 (-54.44)
BeH2_BK_sto3g	14	1488	38666 (18.60)	16442 (-48.10)	13376 (-58.97)	15880 (-49.88)	31365 (-3.79)	14721 (-53.54)
H2_JW_sto3g	4	12	54 (-15.62)	74 (-10.84)	44 (-31.25)	86 (3.61)	45 (-29.69)	73 (-12.05)
H2_JW_631g	8	84	575 (-32.19)	650 (-36.09)	444 (-47.64)	689 (-32.25)	654 (-22.88)	589 (-42.08)
H4_JW_sto3g	8	160	1171 (-33.47)	742 (-63.16)	768 (-56.36)	1180 (-41.41)	1321 (-24.94)	1015 (-49.60)
NH_JW_sto3g	12	640	6415 (-39.25)	3630 (-66.83)	2960 (-71.97)	4356 (-60.20)	9193 (-12.95)	5303 (-51.54)
LiH_JW_sto3g	12	640	6369 (-39.69)	3751 (-68.02)	2952 (-72.05)	4339 (-63.01)	9521 (-9.84)	5250 (-55.25)
H2O_JW_sto3g	14	1000	13204 (-31.17)	7625 (-60.31)	4708 (-75.46)	6794 (-64.64)	16475 (-14.12)	8759 (-54.41)
BeH2_JW_sto3g	14	1488	21787 (-26.24)	10093 (-67.71)	6924 (-76.56)	9709 (-68.94)	25937 (-12.19)	11881 (-61.99)
CH2_JW_sto3g	14	1488	20684 (-29.97)	9477 (-68.60)	6436 (-78.21)	9236 (-69.40)	26909 (-8.89)	12812 (-57.55)
H2_P_sto3g	2	4	4 (0.00)	20 (122.22)	4 (0.00)	9 (0.00)	4 (0.00)	9 (0.00)
H2_P_631g	6	158	627 (-39.01)	610 (-52.86)	528 (-48.64)	826 (-36.17)	609 (-40.76)	564 (-56.41)
H4_P_sto3g	6	164	633 (-40.28)	631 (-53.05)	566 (-46.60)	872 (-35.12)	595 (-43.87)	598 (-55.51)
NH_P_sto3g	10	630	8113 (12.12)	4328 (-46.41)	2184 (-69.82)	3113 (-61.45)	6846 (-5.39)	4418 (-45.29)
LiH_P_sto3g	10	630	8113 (12.12)	4328 (-46.41)	2184 (-69.82)	3113 (-61.45)	6846 (-5.39)	4418 (-45.29)
H2O_P_sto3g	12	1085	18078 (17.65)	8208 (-50.72)	4686 (-69.50)	6275 (-62.33)	14471 (-5.82)	7518 (-54.86)
BeH2_P_sto3g	12	1085	19090 (25.21)	8404 (-48.83)	5320 (-65.11)	6656 (-59.47)	14392 (-5.60)	7617 (-53.62)
CH2_P_sto3g	12	2109	34090 (14.99)	15271 (-52.65)	5950 (-79.93)	9076 (-71.86)	27647 (-6.74)	14646 (-54.59)
H4_P_631g	14	2912	44589 (-8.03)	19036 (-63.18)	10488 (-78.37)	14476 (-72.00)	40499 (-16.46)	18393 (-64.42)
H8_P_sto3g	14	5792	90922 (-5.89)	39910 (-61.41)	12908 (-86.64)	20955 (-79.74)	87114 (-9.83)	39135 (-62.16)
NH3_P_sto3g	14	5792	90922 (-5.89)	39910 (-61.41)	12908 (-86.64)	20955 (-79.74)	87114 (-9.83)	39135 (-62.16)

Table 6.3: Synthesis of the molecule dataset with the algorithms UCCSD-set, PSGS, A-UCCSD-set on a circle architecture.

name	n_qubits	gadgets	UCCSD-set (cx)	UCCSD-set (depth)	PSGS (cx)	PSGS (depth)	A-UCCSD-set (cx)	A-UCCSD-set (depth)
H2_BK_sto3g	4	12	33 (-50.00)	53 (-46.46)	60 (-9.09)	90 (-9.09)	46 (-30.30)	77 (-22.22)
H2_BK_631g	8	84	583 (-50.84)	556 (-59.62)	544 (-54.13)	706 (-48.73)	682 (-42.50)	643 (-53.30)
H4_BK_sto3g	8	160	1847 (-17.98)	1355 (-48.98)	996 (-55.77)	1323 (-50.19)	1601 (-28.91)	1131 (-57.42)
LiH_BK_sto3g	12	640	10052 (-31.99)	5730 (-64.97)	4726 (-68.02)	5830 (-64.36)	12098 (-18.15)	6384 (-60.97)
NH_BK_sto3g	12	640	11326 (-25.78)	6068 (-64.10)	4888 (-67.97)	6233 (-63.12)	10490 (-31.26)	5621 (-66.74)
H2O_BK_sto3g	14	1000	23688 (-17.80)	12146 (-61.61)	8756 (-69.62)	10610 (-66.47)	21829 (-24.25)	10777 (-65.94)
CH2_BK_sto3g	14	1488	46717 (10.37)	18376 (-60.21)	12006 (-71.63)	14681 (-68.21)	31879 (-24.68)	14989 (-67.54)
BeH2_BK_sto3g	14	1488	43056 (3.58)	17228 (-62.01)	12594 (-69.70)	15053 (-66.80)	32916 (-20.81)	15126 (-66.64)
H2_JW_sto3g	4	12	55 (-14.06)	76 (-23.23)	44 (-31.25)	86 (-13.13)	53 (-17.19)	80 (-19.19)
H2_JW_631g	8	84	637 (-42.30)	610 (-54.07)	484 (-56.16)	729 (-45.11)	590 (-46.56)	542 (-59.19)
H4_JW_sto3g	8	160	1509 (-29.62)	829 (-68.02)	768 (-64.18)	1180 (-54.48)	1426 (-33.49)	1059 (-59.14)
NH_JW_sto3g	12	640	7567 (-45.01)	4030 (-74.12)	2960 (-78.49)	4356 (-72.03)	8236 (-40.15)	4560 (-70.72)
LiH_JW_sto3g	12	640	7399 (-46.23)	4066 (-73.71)	2992 (-78.26)	4379 (-71.69)	8299 (-39.69)	4906 (-68.28)
H2O_JW_sto3g	14	1000	13581 (-46.78)	7710 (-72.82)	4708 (-81.55)	6794 (-76.05)	14144 (-44.58)	7574 (-73.30)
BeH2_JW_sto3g	14	1488	25531 (-33.01)	11119 (-73.59)	6996 (-81.64)	9781 (-76.77)	26694 (-29.96)	11993 (-71.52)
CH2_JW_sto3g	14	1488	24235 (-36.41)	10322 (-75.54)	6436 (-83.11)	9236 (-78.11)	28310 (-25.72)	12376 (-70.67)
H2_P_sto3g	2	4	4 (0.00)	20 (122.22)	4 (0.00)	9 (0.00)	4 (0.00)	9 (0.00)
H2_P_631g	6	158	809 (-40.34)	662 (-61.22)	540 (-60.18)	843 (-50.62)	531 (-60.84)	498 (-70.83)
H4_P_sto3g	6	164	820 (-40.92)	664 (-62.21)	566 (-59.22)	884 (-49.69)	606 (-56.34)	535 (-69.55)
NH_P_sto3g	10	630	9727 (3.88)	4729 (-56.37)	2030 (-78.32)	3116 (-71.25)	7303 (-22.01)	4565 (-57.89)
LiH_P_sto3g	10	630	9727 (3.88)	4729 (-56.37)	2030 (-78.32)	3116 (-71.25)	7303 (-22.01)	4565 (-57.89)
H2O_P_sto3g	12	1085	21783 (10.03)	9122 (-59.40)	4642 (-76.55)	6260 (-72.14)	14989 (-24.29)	7626 (-66.06)
BeH2_P_sto3g	12	1085	22450 (13.35)	9258 (-58.81)	4766 (-75.94)	6246 (-72.21)	14866 (-24.94)	7503 (-66.62)
CH2_P_sto3g	12	2109	39439 (2.18)	16475 (-62.47)	4716 (-87.78)	7894 (-82.02)	28609 (-25.88)	14865 (-66.13)
H4_P_631g	14	2912	48980 (-21.54)	19733 (-71.85)	8920 (-85.71)	13054 (-81.38)	42132 (-32.51)	18977 (-72.93)
H8_P_sto3g	14	5792	101769 (-18.29)	41715 (-70.17)	10652 (-91.45)	18538 (-86.74)	90113 (-27.65)	39063 (-72.07)
NH3_P_sto3g	14	5792	101769 (-18.29)	41715 (-70.17)	10652 (-91.45)	18538 (-86.74)	90113 (-27.65)	39063 (-72.07)

Table 6.4: Synthesis of the molecule dataset with the algorithms UCCSD-set, PSGS, A-UCCSD-set on a line architecture.

7 Conclusion

In this work, we have developed three overall architecture-aware synthesis methods. Our Clifford Tableau Synthesis algorithm shows a reasonable scaling behavior and outperforms the state-of-the-art algorithms implemented in qiskit. This algorithm enabled us to develop other synthesis methods, namely synth-divide-and-conquer, and an architecture-aware-UCCSD-set-based algorithm. Overall we have seen that both suffice their purpose concerning routing, nevertheless show weak points concerning optimization, which we account for the gap in the Clifford tableaus for smaller circuits. Further research could shed light on a possible connection and how to avoid this detail. Regarding PSGS, we have seen outstanding performance on random circuits and real-world examples combined with a reasonable asymptotically bound runtime. Additionally, it is worth to point our that PSGS was able to provide a better decay of the fidelity when comparing the order one trotterization of the hamiltonian with the implemented unitary. Nevertheless, we have seen that this algorithm is relatively weak in reducing the depth of a circuit and tends to perform worse for smaller circuits. Here the algorithms of Cowtan et al. [14] outperforms pauli-steiner-gray-synth, which indicates that one could apply the diagonalization process at every base-recursion-step. This could possibly reduce the CNOT-Count further and yield the best of both worlds. Apart from that, future work aspects could be to integrate tests of different Hamiltonians, like the Quantum Alternating Operator Ansatz [28, 14]. Based on our experiments, we can conclude that architecture-aware optimization of quantum circuits is beneficial for both the synthesis of Pauli-polynomials and Clifford-Tableaus.

List of Figures

List of Tables

Bibliography

[1] Scott Aaronson and Daniel Gottesman. Improved simulation of stabilizer circuits. *Physical Review A*, 70(5), nov 2004. doi: 10.1103/physreva.70.052328.

[2] Samson Abramsky and Bob Coecke. A categorical semantics of quantum protocols, 2007.

[3] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. *Comput. Optim. Appl.*, 10:195–210, 1998. doi: 10.1023/A:1018373005182.

[4] Matthew Amy, Parsiad Azimzadeh, and Michele Mosca. On the controlled-NOT complexity of controlled-NOT–phase circuits. *Quantum Science and Technology*, 4(1):015002, sep 2018. doi: 10.1088/2058-9565/aad8ca. URL `https://doi.org/10.1088%2F2058-9565%2Faad8ca`.

[5] Matthew Amy, Owen Bennett-Gibbs, and Neil J. Ross. Symbolic synthesis of clifford circuits and beyond, 2022.

[6] Panagiotis Kl. Barkoutsos, Jerome F. Gonthier, Igor Sokolov, Nikolaj Moll, Gian Salis, Andreas Fuhrer, Marc Ganzhorn, Daniel J. Egger, Matthias Troyer, Antonio Mezzacapo, Stefan Filipp, and Ivano Tavernelli. Quantum algorithms for electronic structure calculations: Particle-hole hamiltonian and optimized wave-function expansions. *Physical Review A*, 98(2), aug 2018. doi: 10.1103/physreva.98.022322. URL `https://doi.org/10.1103%2Fphysreva.98.022322`.

[7] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, et al. Pennylane: Automatic differentiation of hybrid quantum-classical computations, 2022.

[8] Sergey Bravyi and Dmitri Maslov. Hadamard-free circuits expose the structure of the clifford group. *IEEE Transactions on Information Theory*, 67(7):4546–4563, jul 2021. doi: 10.1109/tit.2021.3081415.

[9] Sergey Bravyi, Ruslan Shaydulin, Shaohan Hu, and Dmitri Maslov. Clifford circuit optimization with templates and symbolic pauli gates. *Quantum*, 5:580, nov 2021. doi: 10.22331/q-2021-11-16-580. URL `https://doi.org/10.22331%2Fq-2021-11-16-580`.

[10] Titouan Carette, Dominic Horsman, and Simon Perdrix. SZX-Calculus: Scalable Graphical Quantum Reasoning. In Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen, editors, *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*, volume 138 of *Leibniz International Proceedings in Informatics*

(LIPIcs), pages 55:1–55:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-117-7. doi: 10.4230/LIPIcs.MFCS.2019.55. URL `http://drops.dagstuhl.de/opus/volltexte/2019/10999`.

[11] Nicholas Chancellor, Aleks Kissinger, Joschka Roffe, Stefan Zohren, and Dominic Horsman. Graphical structures for design and verification of quantum error correction, 2023.

[12] Bob Coecke and Ross Duncan. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, 13(4):043016, apr 2011. doi: 10.1088/1367-2630/13/4/043016. URL `https://doi.org/10.10882F1367-26302F132F42F043016`.

[13] Alexander Cowtan, Silas Dilkes, Ross Duncan, Will Simmons, and Seyon Sivarajah. Phase gadget synthesis for shallow circuits. *Electronic Proceedings in Theoretical Computer Science*, 318:213–228, may 2020. doi: 10.4204/eptcs.318.13.

[14] Alexander Cowtan, Will Simmons, and Ross Duncan. A generic compilation strategy for the unitary coupled cluster ansatz, 2020.

[15] Niel de Beaudrap and Dominic Horsman. The zx calculus is a language for surface code lattice surgery, 2020.

[16] Niel de Beaudrap, Xiaoning Bian, and Quanlong Wang. Techniques to reduce $\pi/4$-parity-phase circuits, motivated by the ZX calculus. *Electronic Proceedings in Theoretical Computer Science*, 318:131–149, may 2020. doi: 10.4204/eptcs.318.9. URL `https://doi.org/10.42042Feptcs.318.9`.

[17] Niel de Beaudrap, Aleks Kissinger, and John van de Wetering. Circuit extraction for ZX-diagrams can be # p-hard. In Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, volume 229 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 119:1–119:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.ICALP.2022.119. URL `https://drops.dagstuhl.de/opus/volltexte/2022/16460`.

[18] Timothée Goubault de Brugière, Simon Martiel, and Christophe Vuillot. A graph-state based synthesis framework for clifford isometries, 2022.

[19] Jeroen Dehaene and Bart De Moor. Clifford group, stabilizer states, and linear and quadratic operations over gf(2). *Phys. Rev. A*, 68:042318, Oct 2003. doi: 10.1103/PhysRevA.68.042318. URL `https://link.aps.org/doi/10.1103/PhysRevA.68.042318`.

[20] M. Van den Nest. Classical simulation of quantum computation, the gottesman-knill theorem, and slightly beyond, 2009.

[21] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John van de Wetering. Graph-theoretic simplification of quantum circuits with the ZX-calculus. *Quantum*, 4:279, jun 2020. doi: 10.22331/q-2020-06-04-279. URL `https://doi.org/10.223312Fq-2020-06-04-279`.

[22] Craig Gidney. Stim: a fast stabilizer circuit simulator. *Quantum*, 5:497, jul 2021. doi: 10.22331/q-2021-07-06-497.

[23] Stefano Gogioso and Richie Yeung. Annealing optimisation of mixed zx phase circuits, 2022.

[24] Daniel Gottesman. Stabilizer codes and quantum error correction, 1997.

[25] Daniel Grier and Luke Schaeffer. The classification of clifford gates over qubits. *Quantum*, 6:734, jun 2022. doi: 10.22331/q-2022-06-13-734. URL `https://doi.org/10.223312Fq-2022-06-13-734`.

[26] Harper R. Grimsley, Daniel Claudino, Sophia E. Economou, Edwin Barnes, and Nicholas J. Mayhall. Is the trotterized UCCSD ansatz chemically well-defined? *Journal of Chemical Theory and Computation*, 16(1):1–6, dec 2019. doi: 10.1021/acs.jctc.9b01083. URL `https://doi.org/10.1021%2Facs.jctc.9b01083`.

[27] Kaiwen Gui, Teague Tomesh, Pranav Gokhale, Yunong Shi, Frederic T. Chong, Margaret Martonosi, and Martin Suchara. Term grouping and travelling salesperson for digital quantum simulation, 2021.

[28] Stuart Hadfield, Zhihui Wang, Bryan O'Gorman, Eleanor Rieffel, Davide Venturelli, et al. From the quantum approximate optimization algorithm to a quantum alternating operator ansatz. *Algorithms*, 12(2):34, feb 2019. doi: 10.3390/a12020034. URL `https://doi.org/10.3390%2Fa12020034`.

[29] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11–15, Pasadena, CA USA, 2008. URL `http://conference.scipy.org/proceedings/SciPy2008/paper_2/`.

[30] Aleks Kissinger and Arianne Meijer van de Griend. Cnot circuit extraction for topologically-constrained quantum memories, 2019.

[31] Aleks Kissinger and John van de Wetering. PyZX: Large Scale Automated Diagrammatic Reasoning. In Bob Coecke and Matthew Leifer, editors, *Proceedings 16th International Conference on Quantum Physics and Logic, Chapman University, Orange, CA, USA., 10-14 June 2019*, volume 318 of *Electronic Proceedings in Theoretical Computer Science*, pages 229–241. Open Publishing Association, 2020. doi: 10.4204/EPTCS.318.14.

[32] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison Wesley, 2006.

[33] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for steiner trees. *Acta Informatica*, 15(2):141–145, 1981. doi: 10.1007/BF00288961. URL `https://doi.org/10.1007/BF00288961`.

[34] Gushu Li, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding, and Yuan Xie. 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*, pages 554–569, 2022.

[35] Guang Hao Low, Vadym Kliuchnikov, and Nathan Wiebe. Well-conditioned multiproduct hamiltonian simulation, 2019.

[36] Dmitri Maslov and Martin Roetteler. Shorter stabilizer circuits via bruhat decomposition and quantum circuit transformations. *IEEE Transactions on Information Theory*, 64(7): 4729–4738, 2018. doi: 10.1109/TIT.2018.2825602.

[37] Jarrod R. McClean, Nicholas C. Rubin, Kevin J. Sung, Ian D. Kivlichan, Xavier Bonet-Monroig, Yudong Cao, Chengyu Dai, E. Schuyler Fried, Craig Gidney, Brendan Gimby, Pranav Gokhale, Thomas Haner, Tarini Hardikar, Vojtěch Havlíček, Oscar Higgott, Cupjin Huang, Josh Izaac, Zhang Jiang, Xinle Liu, Sam Mcardle, Matthew Neeley, Thomas O'Brien, Bryan O'Gorman, Isil Ozfidan, Maxwell D. Radin, Jhonathan Romero, Nicolas P.D. Sawaya, Bruno Senjean, Kanav Setia, Sukin Sim, Damian S. Steiger, Mark Steudtner, Qiming Sun, Wei Sun, Daochen Wang, Fang Zhang, and Ryan Babbush. Openfermion: the electronic structure package for quantum computers. *Quantum Science and Technology*, 5:034014, 6 2020. ISSN 2058-9565. doi: 10.1088/2058-9565/AB8EBC. URL `https://iopscience.iop.org/article/10.1088/2058-9565/ab8ebchttps://iopscience.iop.org/article/10.1088/2058-9565/ab8ebc/meta`.

[38] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1), may 2018. doi: 10.1038/s41534-018-0072-4. URL `https://doi.org/10.1038%2Fs41534-018-0072-4`.

[39] Beatrice Nash, Vlad Gheorghiu, and Michele Mosca. Quantum circuit optimizations for NISQ architectures. *Quantum Science and Technology*, 5(2):025010, mar 2020. doi: 10.1088/2058-9565/ab79b1.

[40] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge, 2000. URL `/bib/nielsen/Nielsen2000/QC10th.pdf`.

[41] Ketan N Patel, Igor L Markov, and John P Hayes. Optimal synthesis of linear reversible circuits. *Quantum Inf. Comput.*, 8(3):282–294, 2008.

[42] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5(1), jul 2014. doi: 10.1038/ncomms5213. URL `https://doi.org/10.1038%2Fncomms5213`.

[43] John Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, aug 2018. doi: 10.22331/q-2018-08-06-79. URL `https://doi.org/10.22331%2Fq-2018-08-06-79`.

[44] Jonathan Romero, Ryan Babbush, Jarrod R. McClean, Cornelius Hempel, Peter Love, and Alán Aspuru-Guzik. Strategies for quantum computing molecular energies using the unitary coupled cluster ansatz, 2018.

[45] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t$|$ket$\rangle$: a retargetable compiler for NISQ devices. *Quantum Science and Technology*, 6(1):014003, nov 2020. doi: 10.1088/2058-9565/ab8e92. URL `https://doi.org/10.1088%2F2058-9565%2Fab8e92`.

[46] Masuo Suzuki. General theory of fractal path integrals with applications to many-body theories and statistical physics. *Journal of Mathematical Physics*, 32(2):400–407, 02 1991. ISSN 0022-2488. doi: 10.1063/1.529425. URL `https://doi.org/10.1063/1.529425`.

[47] Jules Tilly, Hongxiang Chen, Shuxiang Cao, Dario Picozzi, Kanav Setia, Ying Li, Edward Grant, Leonard Wossnig, Ivan Rungger, George H. Booth, and Jonathan Tennyson. The variational quantum eigensolver: A review of methods and best practices. *Physics Reports*, 986:1–128, nov 2022. doi: 10.1016/j.physrep.2022.08.003. URL `https://doi.org/10.1016%2Fj.physrep.2022.08.003`.

[48] Arianne Meijer van de Griend and Ross Duncan. Architecture-aware synthesis of phase polynomials for nisq devices, 2020.

[49] Arianne Meijer van de Griend and Sarah Meng Li. Dynamic qubit routing with cnot circuit synthesis for quantum compilation, 2023.

[50] John van de Wetering. Zx-calculus for the working quantum computer scientist, 2020.

[51] Vivien Vandaele, Simon Martiel, Simon Perdrix, and Christophe Vuillot. Optimal hadamard gate count for clifford+t synthesis of pauli rotations sequences, 2023.

[52] David Winderl, Qunsheng Huang, and Christian B. Mendl. A recursively partitioned approach to architecture-aware zx polynomial synthesis and optimization, 2023.

[53] Bujiao Wu, Xiaoyu He, Shuai Yang, Lifu Shou, Guojing Tian, Jialin Zhang, and Xiaoming Sun. Optimization of CNOT circuits on limited-connectivity architecture. *Physical Review Research*, 5(1), jan 2023. doi: 10.1103/physrevresearch.5.013065. URL `https://doi.org/10.1103%2Fphysrevresearch.5.013065`.

[54] Yanqiu Xiao, Fudong Zhang, Guangzhen Cui, Development Khaled AbdElazim, Ramadan Moawad, Essam Elfakharany, Mark Steudtner, and Stephanie Wehner. Fermion-to-qubit mappings with varying resource requirements for quantum simulation. *New Journal of Physics*, 20:063010, 6 2018. ISSN 1367-2630. doi: 10.1088/1367-2630/AAC54F. URL `https://iopscience.iop.org/article/10.1088/1367-2630/aac54fhttps://iopscience.iop.org/article/10.1088/1367-2630/aac54f/meta`.

[55] Richie Yeung. Diagrammatic design and study of ansätze for quantum machine learning, 2020.

[56] Ed Younis, Costin C Iancu, Wim Lavrijsen, Marc Davis, Ethan Smith, and USDOE. Berkeley quantum synthesis toolkit (bqskit) v1, 4 2021. URL `https://www.osti.gov//servlets/purl/1785933`.