Universiteit Leiden

Master Computer Science

Quantum Circuit Equivalence Verification Using the Generalized Stabilizer Formalism

Name:	Lucas Allison
Student ID:	s2348454
Date:	December 9th, 2024
Specialisation:	Foundations of Computing
1st Supervisor:	Dr. Alfons Laarman
2nd Supervisor:	Dr. Jan Martens
Daily supervisor:	Arend-Jan Quist (MSc)

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Einsteinweg 55
2333 CC Leiden
The Netherlands

Abstract

Verifying the equivalence of quantum circuits plays a crucial role in the compilation and optimization of quantum algorithms. Due to limitations of current quantum hardware, quantum circuits must be compiled and optimized to run efficiently on their target devices while adhering to hardware-specific constraints. Equivalence verification ensures that the modifications introduced during these processes preserve the functionality of the original circuit. Various methods have been proposed to verify equivalence of quantum circuits such as decision diagrams, ZX-calculus and weighted model counting, but as quantum computing advances, the increasing size and complexity of quantum circuits demand the continuous development of more efficient and scalable equivalence verification tools. This thesis presents a novel tool for verifying the equivalence of quantum circuits based on a generalization of the stabilizer formalism. While the traditional stabilizer formalism only applies to stabilizer circuits, its generalized version extends its applicability to arbitrary circuits by representing quantum states through their generalized stabilizers. We introduce four data structures to encode these generalized stabilizers as bitstrings in various configurations and use them to perform equivalence verification of arbitrary quantum circuits. We evaluate our tools performance through an empirical comparison against several existing tools. The results demonstrate that for specific quantum algorithms, such as the Deutsch-Jozsa, GHZ state, and Graph state algorithms, our approach outperforms existing tools. Our tool verifies equivalence for these algorithms faster, while using significantly less memory. For circuits exceeding 1000 qubits, it can require over 800 times less memory. Additionally, we show that our tool is robust against floating point inaccuracies, ensuring reliable results. As quantum algorithms grow in complexity, our tool addresses the demand for efficient quantum circuit equivalence verification by demonstrating scalable performance, particularly in terms of memory, for certain circuit types.

Contents

Chapter 1

Introduction

Quantum computing is an area of great potential because it harnesses the principles of quantum mechanics to perform calculations in a fundamentally different way than classical computers. Quantum computers promise immense computational power for tackling certain classes of problems that are intractable for classical computers. Areas where quantum algorithms could provide significant advantages include cryptography, search and optimization, simulation of quantum systems and solving large linear equation systems [1, 2, 3].

Before quantum computing becomes practically feasible there are many challenges to overcome [4, 5]. Current quantum computers, often referred to as Noisy Intermediate-Scale Quantum (NISQ) devices, are not yet fault tolerant [6, 7]. Noise and decoherence introduce strict constraints on qubit connectivity, the set of available quantum gates, and the depth of circuits that can be executed [8, 9]. While quantum algorithms are typically developed at a high level without taking these constraints into account [10, 11], to be run on actual hardware, they must be "compiled" into circuits that adheres to the constraints of the target device [12]. Additionally, it is essential to optimize circuits as much as possible, as it reduces resource usage — such the number of gates and qubits — enabling more complex algorithms to run on today's hardware. This optimization and compilation introduces substantial transformations to a circuit, making it essential to verify that the resulting circuit preserves the original functionality of the algorithm. Ensuring this functional equivalence prevents errors that could alter the intended quantum operations during execution [13, 14]. Additionally, equivalence verification offers valuable insights into quantum complexity theory. Notably, determining the equivalence of two quantum circuits is a QMA-complete problem [15]. Therefore, attempting to solve this problem using classical methods can provide us with insights into the relationship between classical and quantum complexity classes and can deepen our understanding of the inherent difficulty of QMA problems. While equivalence verification is an important aspect in the implementation of quantum algorithms, it remains a challenging problem with no straightforward solution [15]. As the capabilities of quantum computers increase, the size and complexity of quantum circuits will continue to grow, which requires us to keep developing better equivalence verification tools. In this thesis we address this need by developing a novel tool aimed at providing more efficient and scalable quantum circuit equivalence verification.

To achieve this we focus on optimizing two key metrics: runtime and memory usage, which are critical for the efficiency and scalability of an equivalence verification tool. This leads us to two core subproblems:

- Design a data structure that represents quantum states with a minimal amount of memory.
- Design a data structure that enables efficient execution of quantum operations.

In this thesis we present several data structures aimed at optimizing memory usage and execution efficiency, all based on a generalization of the *stabilizer formalism*. The basic idea behind the stabilizer formalism is that a quantum state can be described by a set of unitaries that "stabilize" the state, called its *stabilizers*. The stabilizers form a group under multiplication, which means we can represent the state by the *generators* of its stabilizers. Moreover, we can perform quantum operations directly on the generators, instead of performing them on the state. The formalism has proven powerful for a specific set of states referred to as *stabilizer states*, which can be compactly represented by their stabilizer generators. Circuits that are initialized in a stabilizer state and only contain *Clifford gates* will always result in a stabilizer state and are therefore referred to as *stabilizer circuits* [16]. These circuits are of particular interest because they can be simulated in polynomial time [17, 18]. However, stabilizer circuits represent only a subset of all quantum circuits. Consequently, prior research has generalized the formalism to arbitrary — that is, both stabilizer and non-stabilizer — quantum circuits [19]. This *generalized stabilizer formalism* is based on the representation and manipulation of quantum states using their *generalized stabilizers*. Unlike traditional stabilizers, which consist exclusively of Pauli strings, generalized stabilizers can be expressed as linear combinations of Pauli strings. We introduce four novel data structures designed to represent generalized stabilizers. These data structures encode the Pauli string summands of a generalized stabilizer as bitstrings and each of them stores these bitstrings in various ways. These data structures support equivalence verification using the approach presented in [20], which presents a method for verifying the equivalence of quantum circuits by simulating them using the (generalized) stabilizer formalism. Additionally, we must address floating-point inaccuracies. Since the coefficients of the generalized stabilizers are represented as floating-point numbers, they introduce imprecisions and rounding errors which can accumulate over time. This adds another crucial subproblem:

- Ensure that floating point inaccuracies do not accumulate to the extent that they impact the correctness of equivalence verification results.

We addressed this problem by creating a wrapper for the basic float-64 datatype that keeps track of the number of operations performed on it. This count is then used to dynamically adjust the error margin when comparing two floating-point values.

We empirically compared our tool, using the best-performing data structure, against the existing quantum circuit equivalence verification tools QCEC [14] and ECMC [21]. Our results demonstrate that for certain quantum algorithms, such as the Deutsch-Jozsa, GHZ state, and Graph state, our method outperforms these tools. Specifically, our tool verifies equivalence faster, while using significantly less memory. For circuits with over 1000 qubits, our tool's maximum resident set size over 800 times smaller than that of QCEC and could be up to 540 smaller than ECMC. For the largest circuits of the aforementioned algorithms ECMC and QCEC would either run out of memory or time out, while our tool continued to successfully verify equivalence. Additionally, when verifying non-equivalent circuits, we demonstrated that our tools classifications are very reliable, as it produced no incorrect classifications in the benchmarks performed. This contrasts with both QCEC and ECMC, which either misclassified some non-equivalent circuits as equivalent or failed to produce results.

Our equivalence verification tool builds on the research presented in [20, 21, 19], introducing two main contributions. First, we developed and empirically evaluated four novel data structures capable of representing generalized stabilizers which can be used for equivalence verification. In contrast to ECMC [19, 21], which relies on a symbolic representation of generalized stabilizers, our data structures explicitly store the Pauli string summands of a generalized stabilizer in memory. Second, we address floating-point inaccuracies by introducing a method to dynamically determine an error margin. We demonstrate the robustness of this approach, as it produces no incorrect results across all tested benchmarks. Many other methods have been proposed to verify equivalence of quantum circuits such as ZX-calculus [22], boolean satisfiability [23, 24], path sums [25] and various types of decision diagrams [14, 26, 27, 28, 29]. Our tool expands the existing set of equivalence verification tools using these methods by offering a memory-efficient solution for specific quantum algorithms.

The outline of this thesis will be as follows: in Chapter 2 we introduce the necessary background on quantum computing, the stabilizer formalism and the problem of quantum circuit equivalence verification. In Chapter 3 we present the data structures we developed to store generalized stabilizer generators and how they are used to verifying the equivalence of quantum circuits. In Chapter 4 we present and discuss the results of our experiments and empirically compare our method with existing tools, specifically QCEC [14] and ECMC [21]. In Chapter 5 we discuss related work. In Chapter 6 we discuss additional theoretical aspects, primarily focusing on extending our tool to support measurement simulation. Finally, in Chapter 7 we present our conclusions and discuss future work.

Chapter 2

Preliminaries

In this chapter we introduce the necessary background knowledge used in the remainder of the thesis. We begin by discussing the fundamentals of quantum computing in Section 2.1, followed by an introduction to the stabilizer formalism in Section 2.2. Finally, in Section 2.2.5, we define quantum circuit equivalence and explain how the stabilizer formalism can be applied to verify the equivalence of quantum circuits. Our preliminary discussions draw from the work presented in [30, 16, 31, 21, 19, 20], which may be referred to for further elaboration.

2.1 Quantum Computing Fundamentals

A quantum computer operates on quantum bits or *qubits*. Unlike bits in classical computing, which can only be in one of two states (0 or 1), the state of a qubit can be described by a two-dimensional complex valued unit vector. If the the state is represented by $\left[\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right]$ this implies $|\alpha|^2 + |\beta|^2 = 1$ must hold. The vectors $\left[\begin{smallmatrix}1\\ 0\end{smallmatrix}\right]$ and $\left[\begin{smallmatrix}0\\ 1\end{smallmatrix}\right]$ are referred to as the computational basis states and can be represented using *Dirac notation* as $|0\rangle$ and $|1\rangle$, respectively. In Dirac notation, a *ket* is written as $|\psi\rangle$ and represents a complex vector. The conjugate transpose of a ket is called a *bra* and is written as $\langle\psi|$. When the state of a qubit is a linear combination of the two computational basis states, i.e., $\left[\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right] = \alpha|0\rangle + \beta|1\rangle$, we refer to the qubit as being in a *superposition* of these states and call the complex values α and β the amplitudes of the qubit.

We can combine an arbitrary number of qubits to create an n-qubit quantum system. Its state is represented by the *Kronecker* or *tensor* product of the individual qubits. The Kronecker product is an operation defined on two matrices of any size, where each element of the first matrix is multiplied with the entire second matrix, the results of which are combined in a larger block matrix. Specifically, given an $n \times m$ matrix A and a $p \times q$ matrix B their Kronecker product yields a block matrix of size $np \times mq$:

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1m}B \\ \vdots & \ddots & \vdots \\ a_{n1}B & \cdots & a_{nm}B \end{bmatrix}$$

With n qubits, this produces a 2^n dimensional complex unit vector. For example, we can combine the qubits $|0\rangle$, $|1\rangle$ and $|1\rangle$ to the 3-qubit state $|0\rangle \otimes |1\rangle \otimes |1\rangle$. Usually, it is simply written as $|011\rangle$. States that are a Kronecker products of only $|0\rangle$ and $|1\rangle$ are called computational basis states. As with single qubits, multi-qubit states can exist in a superposition of computational basis states. Using Dirac notation such a state can be written as the sum of computational basis states: $\Sigma_{x\in\{0,1\}^n}\alpha_x|x\rangle$, where n denotes the number of qubits. Due to the state being a unit vector, it must hold that $\Sigma|\alpha_x|^2 = 1$. An example of a two-qubit state in superposition is the Bell state: $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

An alternative way to represent a state vector is by its *density matrix*. It is a way to represent the state of a quantum system as a matrix. The density matrix of a quantum state $|\psi\rangle$ is defined as $|\psi\rangle\langle\psi|$. Here, $|\psi\rangle\langle\psi|$ is the outer product of the state vector with itself. In many cases it can be useful to work with density matrices instead of state vectors.

Quantum computers change the state of qubits using quantum gates. A *quantum gate* is a $2^n \times 2^n$ unitary matrix acting on an n-qubit state. A unitary matrix is a matrix U such that $U \cdot U^\dagger = U^\dagger \cdot U = I^{\otimes n}$. Here '†' denotes the conjugate transpose of the matrix and $I^{\otimes n}$ denotes the tensor product of n identity gates. If the size of the identity gate is clear from the context we will only write I. For instance, in the case above we would write I instead of $I^{\otimes n}$. A quantum gate U can be applied to a state vector $|\psi\rangle$ yielding a new state $|\psi'\rangle$. The state $|\psi'\rangle$ is obtained by performing a matrix multiplication between U and $|\psi\rangle$. We denote this as $U|\psi\rangle = U \cdot |\psi\rangle = |\psi'\rangle$. If we are using the density matrix representation of the state, then gates are applied by *conjugating* the density matrix with the gate. That is, the new state is obtained by $U\rho U^\dagger$.

A *quantum circuit* is sequence of quantum gates, i.e., the circuit $C = U_1 \cdot U_2 \cdots U_j$ would be composed of j gates. A quantum state can evolve through the circuit by sequentially applying each quantum gate. An important consequence of the unitary property of quantum gates is that they are reversible and therefore quantum circuits are too.

Quantum states are not actually observable, but we have to perform a *measurement* to obtain information about it. A POVM (Positive Operator Valued Measure) is set of positive semi-definite operators M_i that sum to the identity operator: $\Sigma_i M_i = I$. A matrix M is positive semi-definite if for every non-zero real column vector z the scalar $z^\dagger M z$ is positive or zero. When a quantum system is in a state represented by the density matrix ρ and a measurement is performed, the state collapses to one of these operators. The probability that the state will collapse to M_j is given by $\mathrm{Tr}(\rho M_j)$, and after the collapse, the system transitions to the state ρ', where ρ' is given by:

$$\rho' = \frac{M_j \rho M_j}{\mathrm{Tr}(\rho M_j)}$$

To provide an example of computing a collapse probability, consider a single qubit in the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. To perform measurements in the computational basis we use the POVM $\{M_0 = |0\rangle\langle 0|, M_1 = |1\rangle\langle 1|\}$. The probability of collapsing to M_0 is computed as follows:

$$\begin{aligned}\mathrm{Tr}(\rho M_0) &= \mathrm{Tr}\left((\alpha|0\rangle\bar{\alpha}\langle 0| + \beta|1\rangle\bar{\beta}\langle 1|)M_0\right) \\ &= \mathrm{Tr}\left(|\alpha|^2|0\rangle\langle 0|0\rangle\langle 0| + |\beta|^2|1\rangle\langle 1|0\rangle\langle 0|\right) \\ &= |\alpha|^2.\end{aligned}$$

The probability of collapsing to M_1 is computed analogously.

2.2 Stabilizer Formalism

The key idea behind the stabilizer formalism is that we represent a quantum state $|\psi\rangle$ by a group of unitary operators that *stabilize* it. A unitary U stabilizes $|\psi\rangle$ if $U|\psi\rangle = |\psi\rangle$, or in other words, $|\psi\rangle$ is a plus one eigenvector of U. If both U and V stabilize $|\psi\rangle$ then so does UV and U^{-1} [16]. Therefore, the set of unitaries that stabilize a given state form a group under matrix multiplication, which we will denote as $\mathrm{Stab}(|\psi\rangle)$. Although this group can uniquely identify a state, it generally does not provide a more efficient representation than using the state vector itself. However, there are some states that can be represented by a significantly smaller subgroup of the unitaries that stabilize them. These states are known as *stabilizer states*.

2.2.1 Stabilizer States

Before defining stabilizer states we need to define the Pauli group, but in order to do so, we first need to define the *Pauli matrices*. There are four Pauli matrices or gates: $I = e_0 = \left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right]$, $X = e_1 = \left[\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right]$, $Y = e_2 = \left[\begin{smallmatrix}0 & -i\\ i & 0\end{smallmatrix}\right]$ and $Z = e_3 = \left[\begin{smallmatrix}1 & 0\\ 0 & -1\end{smallmatrix}\right]$. These gates form a group under matrix multiplication and can be applied to single qubits. In general we can construct a Pauli group for the n-qubit case as follows:

$$\mathbf{P}_n = \{e^{i\theta\pi/2} e_{j_1} \otimes \cdots \otimes e_{j_n} \mid \theta, j_k \in \{0, 1, 2, 3\}\}$$

In other words: $\mathbf{P}_n$ consists exactly of the matrices that can be obtained by taking the Kronecker product of n Pauli matrices. These products are also referred to as *Pauli strings*. When denoting Pauli strings we will omit the Kronecker product. For example, we will write $X \otimes Z \otimes Z$ as XZZ. When we encounter a Pauli string composed of only identity gates except for one non-identity gate, we will represent it by the single non-identity gate with a subscript indicating its position in the Kronecker product. For instance, $IIYI$ will be written as Y_2, when the number of qubits is clear from the context. Now that we have defined the the Pauli group, we can define stabilizer states:

Definition 1 *An n-qubit quantum state $|\psi\rangle$ is a stabilizer state if there exists a subgroup $S \subseteq Stab(|\psi\rangle)$ containing exactly 2^n elements of the Pauli Group $\mathbf{P}_n$. The group S is referred to as a stabilizer group, and its elements are known as the stabilizers of $|\psi\rangle$.*

The stabilizer group S of a stabilizer state $|\psi\rangle$ uniquely defines the state and therefore we can represent the state through S. Since S forms a group under multiplication, the state can be represented by the *generators* of S. Given that S contains 2^n elements, and any finite group can be generated by a logarithmic number of elements with respect to the size of the group, any stabilizer state can be represented using n stabilizer generators. This is the reason stabilizer states can be represented so efficiently: each generator is a Pauli string consisting of n Pauli matrices, and it can be stored using $2n+1$ bits — two bits per Pauli matrix and one bit for the phase. Only a single bit is needed for the phase since stabilizers can only have phases of ± 1. If a stabilizer $s \in S$ had a phase of $\pm i$, then s^2 would also belong to the stabilizer group, but since $s^2 = -I$, it would not stabilize any state. Therefore, a stabilizer state can be stored using $n(2n+1) \in \mathcal{O}(n^2)$ bits.

There is a class of quantum circuits in which every intermediate state is a stabilizer state. This is especially useful because at any point in the circuits state can be efficiently represented using its stabilizer generators. These circuits are known as *stabilizer circuits*.

2.2.2 Stabilizer Circuits

Stabilizer circuits are a subset of quantum circuits which only use the following elements:

- State preparation of the all-zero state.
- Clifford gates.
- Measurements in the computational basis.

A *Clifford gate* is an element of the *Clifford group*: $\mathbf{C}_n$. This group consists of all the unitaries that normalize the Pauli group, i.e. $\mathbf{C}_n = \{V \in U_{2^n} \mid V\mathbf{P}_n V^\dagger = \mathbf{P}_n\}$. The Clifford group can be generated by three gates, the Hadamard (H), phase shift (S) and controlled-not ($CNOT$) gates, which are defined as follows:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}, S = \begin{bmatrix}1 & 0\\ 0 & i\end{bmatrix} \text{ and } CNOT = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}$$

It is important to note that this set of gates does not form a universal basis and therefore stabilizer circuits only comprise a subset of all quantum circuits. These circuits are known as stabilizer circuits because the initial all-zero state is a stabilizer state, and every intermediate state throughout the circuit also remains a

stabilizer state. This can be seen from the fact that the Clifford group normalizes the Pauli group. Consider an arbitrary Clifford gate C, a stabilizer state $|\psi\rangle$ and one of its stabilizers s. Then the following holds:

$$C|\psi\rangle = Cs|\psi\rangle = CsC^\dagger C|\psi\rangle$$

From this we can see that if s stabilizes the state $|\psi\rangle$, then $CsC^\dagger$ will stabilize the transformed state $C|\psi\rangle$. Additionally, if S is the stabilizer group of $|\psi\rangle$, then $S' = \{CvC^\dagger \mid v \in S\}$ forms the stabilizer group of $C|\psi\rangle$. Since the Clifford group normalizes the Pauli group, S' will consist of exactly 2^n Pauli strings, making it a valid stabilizer group for the state $C|\psi\rangle$. Therefore, the state $C|\psi\rangle$ is a stabilizer state.

Stabilizer circuits can be efficiently simulated on classical computers in polynomial time, due to the Gottesman-Knill theorem [17]. These circuits play a key role in various quantum computing applications, including quantum error correction and quantum communication [30]. In the following section, we will explore the methods for simulating stabilizer circuits.

2.2.3 Simulation of Stabilizer Circuits

To explain how stabilizer circuits are simulated with the stabilizer formalism we will discuss how each element of a stabilizer circuit is simulated. Afterwards we will explain why these simulations can be performed so efficiently. Consider a stabilizer circuit $C = C_1C_2C_3 \dots C_l$ of n qubits. To simulate the circuit we perform the following steps:

State Preparation

We start by representing the all-zero state by its stabilizer generators, which is the set $\{Z_i \mid i \in [0..n]\}$.

Gate Application

In Section 2.2.2 we saw that if a Clifford gate U is applied to a stabilizer state $|\psi\rangle$ with stabilizers S, then the transformed state $U|\psi\rangle$ has stabilizers $S' = USU^\dagger$. Moreover, if G are the generators of S, then $UGU^\dagger$ are the generators of S'. This implies that to simulate the effect of a Clifford gate on a state, we need to conjugate the stabilizer generators of the state, thereby obtaining the stabilizer generators of the transformed state. Therefore, to simulate a Clifford circuit C, for each Clifford gate $C_i \in C$ we conjugate the stabilizer generators of $C_{i-1} \dots C_1|0^n\rangle$ with C_i, thereby obtaining the stabilizer generators of $C_iC_{i-1} \dots C_1|0^n\rangle$. This set of represents the state obtained by applying $C_1, \dots C_i$ to the all-zero state.

Measurements

Before we discuss how we obtain the measurement probabilities, we will first explain how we can express the density matrix of a state in terms of its stabilizers.

In [16] it is shown that the density matrix of the all-zero state can be expressed as a product of the generators of its stabilizers, the group $G_0 = \{Z_i \mid i \in [0..n]\}$. Or, in other words, the density matrix of the all-zero state equals the sum of its stabilizers:

$$\begin{aligned} |0^n\rangle\langle 0^n| &= \frac{1}{2^n} \prod_{Z_j \in G_0} (I + Z_j) \\ &= \frac{1}{2^n} \sum_{g \in \langle G_0 \rangle} g \end{aligned}$$

Notice that after having applied any Clifford circuit C the following holds:

$$\begin{aligned} C|0^n\rangle\langle 0^n| C^\dagger &= C \left(\frac{1}{2^n} \sum_{g \in \langle G_0 \rangle} g \right) C^\dagger \\ &= \frac{1}{2^n} \sum_{g \in \langle G_0 \rangle} C g C^\dagger \end{aligned} \tag{2.1}$$

From this we can see that if a circuit starts in the all-zero state, then the density matrix of a state obtained by applying a Clifford circuit to the all-zero state equals the sum of the conjugated stabilizers of the all-zero state. Since we can obtain any stabilizer state by applying Clifford gates to the all-zero state [16], this means that the density matrix of any stabilizer state equals the sum of its stabilizers. This fact will be useful when we discuss how to obtain the measurement probabilities.

The probability of measuring a $|0\rangle$ for the i^{th} qubit of a stabilizer state with density matrix ρ and stabilizer generators G is given by the formula $p_0 = \text{Tr}(\rho M_0)$, where $M_0 = I^{\otimes i-1} \otimes |0\rangle\langle 0| \otimes I^{\otimes n-i} = \frac{I+Z_i}{2}$, which expands to:

$$\begin{aligned} p_0 &= \text{Tr}(\rho M_0) \\ &= \text{Tr} \left(\frac{1}{2^n} \sum_{g \in \langle G \rangle} g \frac{I + Z_i}{2} \right) \\ &= \frac{1}{2} \text{Tr} \left(\frac{1}{2^n} \sum_{g \in \langle G \rangle} g \right) + \frac{1}{2} \text{Tr} \left(\frac{1}{2^n} \sum_{g \in \langle G \rangle} g Z_i \right) \\ &= \frac{1}{2} \text{Tr}(\rho) + \frac{1}{2^{n+1}} \sum_{g \in \langle G \rangle} \text{Tr}\left(g Z_i\right) \\ &= \frac{1}{2} + \frac{1}{2^{n+1}} \sum_{g \in \langle G \rangle} \text{Tr}\left(g Z_i\right) \end{aligned} \tag{2.2}$$

The probability of measuring a $|1\rangle$ is computed analogously, but with $M_1 = \frac{I-Z_i}{2}$. When measuring the i^{th} qubit of a stabilizer state there are two possibilities:

1. Z_i commutes with all the stabilizer generators and therefore in turn with all of the stabilizers.

2. Z_i anti-commutes with one or more of the stabilizer generators.

In the first case it follows that either Z_i or $-Z_i$ is an element of the stabilizer which yields a measurement of $|0\rangle$ or $|1\rangle$ respectively with probability 1 [30]. This is the case because the only Pauli string that results in a non-zero trace is the one consisting entirely of identity matrices. Any Pauli string containing at least one X or Y matrix will have a trace of 0, as such matrices are anti-diagonal. If a Pauli string contains no X or Y matrices but includes at least one Z matrix, it will be diagonal but still have a trace of 0, as there will be an equal number of diagonal elements that are 1 and -1. Therefore, the stabilizer $\pm Z_i$ is the only stabilizer that will yield an identity matrix for the product in the sum of Equation 2.2. For this reason, if Z_i is in the stabilizer the probability of measuring $|0\rangle$ is 1:

$$\begin{aligned} p_0 &= \frac{1}{2} + \frac{1}{2^{n+1}} \operatorname{Tr}\left(\sum_{g \in \langle G \rangle} g Z_i\right) \\ &= \frac{1}{2} + \frac{1}{2^{n+1}} \operatorname{Tr}(I) \\ &= 1 \end{aligned}$$

In the case that $-Z_i$ is an element of the stabilizer, the probability of measuring $|1\rangle$ is computed analogously and yields 1. The measurement does not disturb the state so the set of stabilizer generators remains unchanged.

In the second case, the probability of measuring $|0\rangle$ and $|1\rangle$ are both $\frac{1}{2}$. To show why this holds we will use two facts. First, we can pick any stabilizer generator g that anti-commutes with Z_i, and ensure that it is the only stabilizer generator that anti-commutes with Z_i by replacing any other stabilizer generator g' for which this holds with gg'. Second, if we denote the current state as $|\phi\rangle$, then then it holds for any generator g'' $|\phi\rangle = g''|\phi\rangle$. Using these facts we obtain:

$$\begin{aligned} p_0 &= \operatorname{Tr}(\rho M_0) \\ &= \operatorname{Tr}\left(|\phi\rangle\langle\phi| \frac{I + Z_i}{2}\right) \\ &= \operatorname{Tr}\left(g|\phi\rangle\langle\phi| \frac{I + Z_i}{2}\right) \\ &= \operatorname{Tr}\left(g\rho \frac{I + Z_i}{2}\right) \end{aligned}$$

By using the fact that $g = g^\dagger$, which holds for any stabilizer, and by applying the cyclic property of the trace we obtain:

$$\begin{aligned} p_0 &= \mathrm{Tr}\left(\rho \frac{I + Z_i}{2} g\right) \\ &= \mathrm{Tr}\left(\rho g \frac{I - Z_i}{2}\right) \\ &= \mathrm{Tr}\left(\rho g^\dagger M_1\right) = \mathrm{Tr}(\rho M_1) \\ &= p_1 \end{aligned}$$

Since it must hold that $p_0 + p_1 = 1$, we can deduce that both equal $\frac{1}{2}$. This measurement does alter the state. Therefore, in order to correctly adjust the stabilizer generators we substitute g with Z_i or $-Z_i$ depending on whether we simulated a measurement on $|0\rangle$ or $|1\rangle$, respectively.

Simulation Complexity

We can now see why these circuits can be simulated efficiently. We can efficiently represent a stabilizer state by its stabilizer generators and each operation in a stabilizer circuit can be perform efficiently: the generators of the stabilizers of the all-zero state are n Pauli strings which can easily be constructed. Since Clifford gates normalize the Pauli group, conjugating this set of generators with any number of Clifford gates will always result in a set of n Pauli strings. Because conjugation can be implemented in constant time, by applying the update rules specified in Table 2.1, if we have k Clifford gates in the circuit, simulating them will take $\mathcal{O}(n \cdot k)$ time. Finally, to simulate a measurement of the ith qubit in the worst case Z_i commutes with all the stabilizer generators. In this case determining whether Z_i or $-Z_i$ is a part of the stabilizer requires inverting a matrix, which in theory has a time complexity of $\mathcal{O}(n^\omega)$, where ω is *the exponent of matrix multiplication*, but in practice has a time complexity of $\mathcal{O}(n^3)$.

2.2.4 Arbitrary Quantum Circuit Simulation Using the Stabilizer Formalism

A more negative interpretation of the Gottesman-Knill theorem is that stabilizer circuits by themselves are not useful for producing superpolynomial quantum speedups [31]. Therefore, various research has been conducted into using the stabilizer formalism to simulate arbitrary quantum circuits [21, 32].

As mentioned in Section 2.2.2, the Clifford group can be generated by the H, S and $CNOT$ gates. While they do not form a universal set of basis gates, augmenting them with the T gate, or more generally the $R_z(\theta)$ gate, we do obtain a universal set of quantum gates [33]. These gates are defined as follows:

Table 2.1: Update rules for the transformation of Pauli matrices under Clifford and R_z gate conjugation. Both tables contain three columns. The third column contains the result of conjugating the Pauli matrix depicted in the second column with the gate depicted in the first column.

Gate	Input	Output
H = H†	X	Z
	Y	-Y
	Z	X
S	X	Y
	Y	-X
	Z	Z
S†	X	-Y
	Y	X
	Z	Z
Rz(θ)	X	cos(θ)X + sin(θ)Y
	Y	$-$ sin(θ)X + cos(θ)Y
	Z	Z
Rz(θ)†	X	cos(θ)X - sin(θ)Y
	Y	sin(θ)X + cos(θ)Y
	Z	Z

Gate	Input	Output
CNOT = CNOT†	I_cX_t	I_cX_t
	I_cY_t	Z_cY_t
	I_cZ_t	Z_cZ_t
	X_cI_t	X_cX_t
	X_cX_t	X_cI_t
	X_cY_t	Y_cZ_t
	X_cZ_t	$-Y_cY_t$
	Y_cI_t	Y_cX_t
	Y_cX_t	Y_cI_t
	Y_cY_t	$-X_cZ_t$
	Y_cZ_t	X_cY_t
	Z_cI_t	Z_cI_t
	Z_cX_t	Z_cX_t
	Z_cY_t	I_cY_t
	Z_cZ_t	I_cZ_t

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix} \text{ and } R_z(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}$$

Since any quantum circuit can be constructed using the H, S, $CNOT$, and R_z gates, we will limit our focus to this specific set of gates for the remainder of this thesis. However, we will continue to use the T gate as a representative of the R_z gate, as it is a special instance of the R_z gate, with $\theta = \pi/4$.

In contrast to Clifford gates, when a Pauli string is conjugated with an R_z gate it could yield a linear combination of Pauli strings. How Pauli strings transform under R_z conjugation is depicted in Table 2.1. As an example, conjugating an X matrix with a T gate will yield:

$$TXT^\dagger = \frac{1}{\sqrt{2}}(X + Y)$$

Consider a more complicated example, where we first conjugate the Pauli string $ZIXZX$ with T_3 and subsequently with T_5. The first conjugation yields:

$$T_3(ZIXZX)T_3^\dagger = \frac{1}{\sqrt{2}}(ZIXZX + ZIYZX).$$

Subsequently, conjugating with T_5 will give:

$$T_5 \left(\frac{1}{\sqrt{2}}(ZIXZX + ZIYZX) \right) T_5^\dagger = \frac{1}{2}(ZIXZX + ZIXZY + ZIYZX + ZIYZY).$$

As we can see, the stabilizer formalism does not generally apply to arbitrary quantum circuits, as conjugating a stabilizer group with an R_z gate does not always yield another stabilizer group, thus violating the principles of the stabilizer formalism. To address this limitation, previous research has introduced the *generalized stabilizer formalism*. This formalism defines the concept of a *generalized stabilizer group*, which can represent *generalized stabilizer states*, encompassing all quantum states [19, 34]:

Definition 2 *In an n-qubit quantum system, a generalized stabilizer state $|\psi\rangle$ is the simultaneous eigenvector, with eigenvalue 1, of a group containing 2^n commuting unitary operators S. The set of S is a generalized stabilizer group.*

To simulate arbitrary quantum circuits using the generalized stabilizer formalism, we follow the same approach as for stabilizer circuits. However, instead of representing each (intermediate) quantum state by the generators of its stabilizer group, we describe it using the generators of its generalized stabilizer group. At first glance, from Definition 2, it may appear that simulating arbitrary quantum circuits is no more complex than simulating stabilizer circuits, since the generalized stabilizer group also consists of 2^n elements. The key difference is that the elements of the generalized stabilizer group are not limited to Pauli strings; they can also be linear combinations of these strings. This becomes clear through the following example, where we initially represent the state $|00\rangle$ using its generalized stabilizer generators and then apply an H and T gate to the 0^{th} qubit:

$$\begin{Bmatrix} ZI \\ IZ \end{Bmatrix} \xrightarrow{\text{Apply } H_0} \begin{Bmatrix} X \\ IZ \end{Bmatrix} \xrightarrow{\text{Apply } T_0} \begin{Bmatrix} \frac{1}{\sqrt{2}}(IX + IY) \\ IZ \end{Bmatrix}$$

As illustrated, performing a simulation of an arbitrary n-qubit quantum circuit using the generalized stabilizer formalism still requires us to keep track of n generators, but the size of each generator potentially grows exponentially in the number of R_z gates. This means that at any given point during the simulation the obtained state is represented by the set of generalized stabilizer generators $\{g_i = \sum \alpha_j \cdot p_j \mid i \in [0..n], \alpha_j \in \mathbb{R}, p_j \in \mathbf{P}_n\}$. This means that conjugating a generator that is a linear combination of j Pauli strings will take $\mathcal{O}(j)$ time. This is because we have to conjugate each Pauli string in the sum individually. Therefore, conjugating the generators at a given point in the simulation will now take $\mathcal{O}(n \cdot 2^r)$ time, where r denotes the number of simulated R_z gates. Not only the complexity of gate conjugation increases, but so does that of simulating measurements. We discuss this extensively in Chapter 6.

2.2.5 Quantum Circuit Equivalence

In this section we first introduce the notion of quantum circuit equivalence and subsequently discuss how the stabilizer formalism can be used to verify equivalence of quantum circuits.

Before we formally define quantum circuit equivalence, it is important to note that we will only consider circuits without measurements when verifying equivalence. The reason we don't need to consider measurements is because they only provide us with information about a quantum state, while the unitary gates of the circuit determine how the state evolves. The latter is what we are interested in, as we want to know whether the quantum states produced by the unitary gates of two circuits are equivalent. Omitting measurements does not limit the circuits for which we can verify equivalence, even not for the circuits that contain intermediate measurements. This is because of *the principle of deferred measurement* [30], which allows us to push measurements to the end of a circuit without affecting the final outcome probabilities.

Defining Quantum Circuit Equivalence

To understand quantum circuit equivalence we first have to understand the *global phase*. Given a quantum state, the global phase refers to a complex multiplicative factor that all amplitudes have in common. This factor does not have any observable effects on the measurement probabilities of a state. Importantly, this implies that two quantum states that only differ by a global phase are indistinguishable and are therefore considered equivalent as we can see in the following example which can be generalized:

Consider the two states $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ and $e^{i\theta}|\psi\rangle = |\psi'\rangle = e^{i\theta}(\alpha|0\rangle + \beta|1\rangle)$, where θ is some real number. The probability of observing the state $|0\rangle$ is given by $|\alpha|^2$ and $|e^{i\theta}\alpha|^2 = (e^{i\theta}\alpha)(e^{-i\theta}\alpha^*) = |\alpha|^2$ for the states $|\psi\rangle$ and $|\psi'\rangle$ respectively. Here '$*$' denotes the complex conjugate. Since the probabilities of observing $|1\rangle$ are computed analogously we can see that the factor $e^{i\theta}$ does change the measurement probabilities.

We can now define quantum circuit equivalence. Two quantum circuits are considered to be equivalent if they produce the same output state for each possible input state. That is, given two circuits, C_1 and C_2, and an arbitrary quantum state $|\psi\rangle$ then $C_1 \simeq C_2$ if and only if $C_1|\psi\rangle = e^{i\theta}C_2|\psi\rangle$. If $e^{i\theta}$ is non-zero we say they are *equivalent up to the global phase*.

Verifying Quantum Circuit Equivalence Using the Generalized Stabilizer Formalism

In [20] the authors demonstrate that the stabilizer formalism can be effectively used to verify the equivalence of stabilizer circuits. Moreover, this approach can be extended to verify the equivalence of arbitrary quantum circuits. Before we present

how the method works, it is important to introduce a key theorem of the paper:

Theorem 1 (Thanos et al. [20]) *Let U, V be two unitaries on $n \geq 1$ qubits. Then U is equivalent to V if and only if the following conditions hold:*

1. *for all $j \in \{1, 2, \ldots, n\}$, we have $UZ_jU^\dagger = VZ_jV^\dagger$; and*
2. *for all $j \in \{1, 2, \ldots, n\}$, we have $UX_jU^\dagger = VX_jV^\dagger$.*

The original proof can be found in [20], but a concise version will be presented here. For convenience, we will use the notation $G_Z = \{Z_i \mid i \in \{0, 1, \ldots, n\}\}$ and $G_X = \{X_i \mid i \in \{0, 1, \ldots, n\}\}$ for the remainder of this section.

Proof. If $U \backsimeq V$, then $U = cV$ for some coefficient $c \in \mathbb{C}$ with norm 1. In this case, for all $z_j \in G_Z$, it follows that $Uz_jU^\dagger = cVz_j(cV)^\dagger = cVz_jc^*V^\dagger = |c|^2Vz_jV^\dagger = Vz_jV^\dagger$, where c^* denotes the complex conjugate of c. The proof for the elements of G_X is analogous. The converse direction relies on the fact that any n-qubit state $|\phi\rangle$ can be expressed as a sum of Pauli strings:

$$|\phi\rangle\langle\phi| = \sum_{P_j \in \mathbf{P}_n} \alpha_j P_j$$

We also know that each Pauli string can be expressed as a sum of Pauli strings from the sets G_Z and G_X. In combination with the fact that $Uz_jU^\dagger = Vz_jV^\dagger$ and $Ux_kU^\dagger = Vx_kV^\dagger$ holds, for all $z_j \in G_Z$ and all $x_k \in G_X$, we know that U and V coincide on all Pauli strings by conjugation. This implies that the following holds:

$$U|\phi\rangle\langle\phi|U^\dagger = \sum_{P_j \in \mathbf{P}_n} \alpha_j UP_jU^\dagger = \sum_{P_j \in \mathbf{P}_n} \alpha_j VP_jV^\dagger = V|\phi\rangle\langle\phi|V^\dagger$$

From this we know that $U|\phi\rangle = cV|\phi\rangle$ and thus that $V^\dagger U|\phi\rangle = c|\phi\rangle$, for some constant $c \in \mathbb{C}$ with norm 1, or in other words, $|\phi\rangle$ is an eigenvector of $V^\dagger U$. Since $|\phi\rangle$ is an arbitrary state, we can conclude that any vector is an eigenvector of $V^\dagger U$. Therefore, it holds that $V^\dagger U$ is a multiple of the identity operator. Let c be some complex number with norm 1, then this yields:

$$V^\dagger U = cI \leftrightarrow U = cV$$

□

We can now see how the stabilizer formalism can be used in combination with Theorem 1 to verify the equivalence of stabilizer circuits U and V. First, we simulate both circuits using the stabilizer formalism starting from the all-zero state. Since the stabilizer generators of the all zero-state are exactly G_z at the end of the simulation we obtain the following sets:

$$\{UZ_0U^\dagger, UZ_1U^\dagger, \ldots, UZ_nU^\dagger\} \text{ and } \{VZ_0V^\dagger, VZ_1V^\dagger, \ldots, VZ_nV^\dagger\}$$

We can now easily verify that the first condition of Theorem 1 holds by checking that for each $j \in \{0, 1, \ldots, n\}$ it holds that $UZ_jU^\dagger = VZ_jV^\dagger$. Second, we repeat this process, but start the simulations from the all-plus state, whose stabilizer generators are exactly G_x. This state can be obtained by applying a Hadamard gate to all the qubits of the all-zero state. This allows us to easily verify whether the second condition holds.

This approach can be readily adapted to verify the equivalence of arbitrary quantum circuits. The main difference is that instead of using stabilizer simulations, we use the generalized stabilizer formalism for our simulations, as outlined in Section 2.2.4. Additionally, we will use slightly modified, yet equivalent, conditions from Theorem 1. The first condition can be restated as:

$$\textit{For all } j \in \{1, 2, \ldots, n\} \textit{ it holds that } V^\dagger UZ_jU^\dagger V = Z_j$$

In this case we can perform a generalized stabilizer simulation of the circuit $UV^\dagger$, after which we obtain the following generalized stabilizer group:

$$\{V^\dagger UZ_0U^\dagger V, V^\dagger UZ_1U^\dagger V, \ldots, V^\dagger UZ_nU^\dagger V\}$$

This set can then be used to verify restated condition. The second condition from Theorem 1 can be restated analogously. With this, we can now formally define the main equivalence verification method of this thesis, which will be used in Chapter 3:

Method 1 *To verify the equivalence of two arbitrary quantum circuits U and V perform the following steps:*

1. *Simulate the circuit $UV^\dagger$ using the generalized stabilizer formalism, starting from the all-zero state. Then, with the resulting generalized stabilizer group, verify that for each $j \in \{0, 1, \ldots, n\}$, the condition $V^\dagger UZ_jU^\dagger V = Z_j$ holds.*
2. *Repeat step 1, but start the simulation from the all-plus state. Then, verify that for each $j \in \{0, 1, \ldots, n\}$, the condition $V^\dagger UX_jU^\dagger V = X_j$ holds.*

If all conditions hold, then the circuits U and V are equivalent, otherwise they are not.

Chapter 3

Data Structures for Representing Generalized Stabilizers

In this chapter we discuss the data structures and algorithms that are used to develop our quantum circuit equivalence verification tool. Any further implementation details can be viewed on Github [35]. To verify equivalence using the generalized stabilizer formalism we use Method 1 described in Section 2.2.5. To do this we need a data structure that can represent and conjugate a generalized stabilizer (GS). In Section 3.1 we present the abstract data type that is capable of performing these operations and show why the these operations are sufficient for our approach. In Section 3.2 and Section 3.3 we discuss the four concrete implementations of the ADT. In the former section we discuss all shared functionality, while in the latter section we discuss the implementation details that are unique to each of them.

3.1 Generalized Stabilizer Abstract Data Type

To verify equivalence using Method 1, we need the ability to store and conjugate generalized stabilizers. To achieve this, we define an Abstract Data Type (ADT) that represents a single generalized stabilizer. This data type should support the following operations:

- `new(args: any)`
  Initialize the data structure with any required arguments.
- `set_stabilizer(stabilizer: List[(Pauli String, float)])`
  Set the data structure to represent a given generalized stabilizer. These stabilizers are linear combinations of Pauli strings, so the function takes a list of tuples, where each tuple represents on of its summands: a Pauli string and its corresponding coefficient. The function enforces two constraints: first, all Pauli strings must have the same length, and second, the sum of the squared coefficients must be equal to one.
- `equals_stabilizer(stabilizer: List[(Pauli String, float)])`
  Checks if the provided generalized stabilizer (a list of Pauli string summands and their corresponding coefficients) matches the stabilizer currently represented by the data structure. The function returns `true` if the provided summands and coefficients are equivalent to those stored in the data structure, accounting for floating point imprecision. It only accepts stabilizer

where the length of each Pauli string matches the length of the corresponding strings in the data structure and the sum of the squared coefficients equals one.

- `conjugate(gate: Gate)`
  Conjugates the provided with the generalized stabilizer. Specifically, each Pauli string summand is conjugated independently with this gate. Note that conjugating a Pauli string can result in a linear combination of Pauli strings, so this operation may increase the number of Pauli strings summands stored in the data structure.

A graphical depiction of this functionality is provided in Figure 3.1.

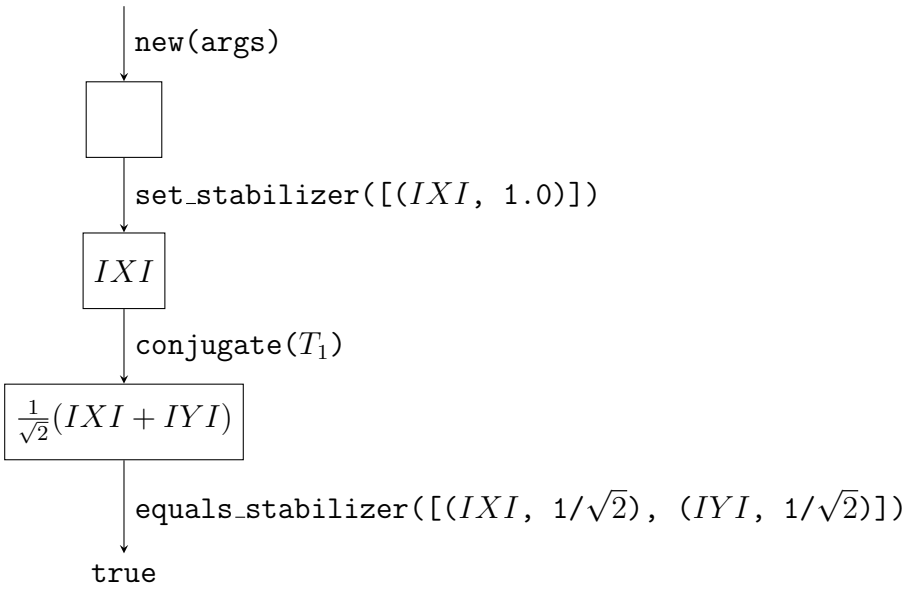

Figure 3.1: Graphical example of the functionality of the ADT of a generalized stabilizer. The arrows are labeled with the executed functions, the boxes represent the contents of the data type.

We will now explain why the functionality of this ADT is sufficient to perform equivalence verification using Method 1. The key point is that this data type can represent any generalized stabilizer. Specifically, to verify the equivalence of two n-qubit quantum circuits, U and V, we need to perform a simulation using the generalized stabilizer formalism of the circuit $UV^\dagger$, starting from both the all-zero and all-plus states. We can represent these using the generators of their generalized stabilizers which are the sets $\{Z_i \mid i \in \{0, 1, \ldots, n\}\}$ and $\{X_i \mid i \in \{0, 1, \ldots, n\}\}$, respectively. We can use the ADT to represent each individual element of these sets. To perform the simulation, we simply need to conjugate the elements of these sets, which is a functionality provided by the ADT. Finally, for each element produced in the simulation, we must check whether it matches a specific generalized stabilizer, which the ADT also supports.

In Algorithm 1 we provide pseudocode to verify equivalence using the discussed ADT, but before we do so, there is one important observation to make. Note that when a generator is conjugated with the circuits $UV^\dagger$, this operation does not influence any of the other generators. As a result, the process of conjugating each generator can be treated independently, making the task inherently parallelizable. The "embarrassingly parallel" nature of this algorithm presents an excellent opportunity for implementing the parallel conjugation of generators, allowing for an easy optimization.

Algorithm 1 Quantum Circuit Equivalence

Input: U: Quantum Circuit, V: Quantum Circuit, n_qubits: int
Output: Equivalent (Bool)

```
1: procedure CircuitEquivalence:
2:     for P_i ∈ {Z_k | k ∈ {0,1,...,n}} ∪ {X_l | l ∈ {0,1,...,n}} in parallel do
3:         GS ← GeneralizedStabilizer.new()
4:         GS.set_stabilizer([(P_i, 1.0)])
5:         for U_j ∈ UV† do
6:             GS.conjugate(U_j)
7:         if not GS.equals_stabilizer([(P_i, 1.0)]) then
8:             return false
9:     return true
```

3.2 Generalized Stabilizer ADT Implementations: Shared Functionality

Some functionality is shared between all the implementations of the generalized stabilizer ADT. Before discussing implementation details unique to each of them, we will first discuss the shared functionality.

3.2.1 Representing Pauli Strings

A single Pauli matrix can be stored using two bits. The I, X, Y, Z matrices are encoded as 00, 10, 11, 01 respectively. The notation is taken from Gottesman and Aaronson [16], where the matrix is $X^{\text{msb}}Z^{\text{lsb}}$, where msb and lsb are the most and least significant bits respectively. Storing Pauli matrices this way allows us to store a Pauli string of length n in $2n$ bits.

3.2.2 H and S Conjugations

We can prevent the need to conjugate all the summands of a generalized stabilizer with the H and S gates by storing the conjugations of the H and S gates in an additional data structure. We store an array, where for each qubit we store a triple of tuples of the form (Pauli matrix, sign). The i^{th} triple represents the

transformations of the X, Y and Z Pauli matrices, respectively, by the H and S gates that target the i^{th} qubit. For example, if we want to conjugate the 2^{nd} qubit with a Hadamard gate, then we conjugate gates of the three tuples stored at index 1. These tuples now represent the changes to all the stored Pauli string summands, without having to explicitly update them. If no previous H or S conjugations are applied the tuples stored at index 1 in the array will be updated as follows:

$$\begin{gathered}\text{Before } H \text{ conjugation} \rightarrow \text{After conjugation} \\ (X,1),(Y,1),(Z,1) \rightarrow (Z,1),(Y,-1),(X,1)\end{gathered}$$

The first tuple on the right hand side now indicates that whenever we encounter a Pauli string with an X matrix at the 2^{nd} qubit we should treat it as a Z matrix, the second tuple indicates that we should treat a Y matrix at the 2^{nd} qubit as a $-Y$ matrix and a final tuple indicates that we should treat a Z matrix at the 2^{nd} qubit as an X matrix. Any arbitrary number H and S gates can be be applied this way. For example, if we now want to conjugate the second qubit with an S gate the triple at index 1 in the array will be updated as follows:

$$\begin{gathered}\text{Before } S \text{ conjugation} \rightarrow \text{After conjugation} \\ (Z,1),(Y,-1),(X,1) \rightarrow (Z,1),(X,1),(Y,1)\end{gathered}$$

Now the first tuple on the right hand side indicates that whenever we encounter a Pauli string with an X matrix at the 2^{nd} qubit we should treat it as a Z matrix, the second tuple indicates that we should treat a Y matrix as an X matrix and the final tuple indicates that we should treat a Z matrix as a Y matrix.

To make this more concrete consider the generalized stabilizer $0.8YI + 0.6IZ$ and a newly initialized H/S conjugations map. Below we have depicted the former as a set of its summands and the latter as an array:

$$\begin{aligned}&\text{H/S conjugations map: } \big[(X,1),(Y,1),(Z,1), \quad (X,1),(Y,1),(Z,1)\big] \\ &\text{Generalized Stabilizers: } \begin{Bmatrix} 0.8YI \\ 0.6IZ \end{Bmatrix}\end{aligned}$$

Now conjugating the 2^{nd} qubit with an H and subsequently an S gate will yield the following:

$$\begin{aligned}&\text{H/S conjugations map: } \big[(X,1),(Y,1),(Z,1), \quad (Z,1),(X,1),(Y,1)\big] \\ &\text{Generalized Stabilizers: } \begin{Bmatrix} 0.8YI \\ 0.6IZ \end{Bmatrix}\end{aligned}$$

As we can see the summands of the generalized stabilizer that are stored in memory are not updated, only the triples in the H/S conjugations map are. Say we now

want to conjugate the 2nd qubit with a T gate, then we must first apply the compound H/S conjugations to the summands before we conjugate them:

$$\begin{Bmatrix} 0.8YI \\ 0.6IZ \end{Bmatrix} \xrightarrow[H/S \text{ conjugations}]{\text{Apply compound}} \begin{Bmatrix} 0.8YI \\ 0.6IY \end{Bmatrix} \xrightarrow[T \text{ gate}]{\text{Conjugate}} \begin{Bmatrix} 0.8YI \\ \frac{0.6}{\sqrt{2}}(IY - IX) \end{Bmatrix}$$

The advantage of this approach is that H and S conjugates can now be performed in $\mathcal{O}(1)$ time instead of $\mathcal{O}(2^r)$ time, where r represents the number of R_z gates in the circuit. However, each data structure using this method is responsible for taking the H and S conjugations into account when conjugating or using the generalized stabilizer it represents.

3.2.3 Numerical Stability

The coefficients of Pauli strings are stored as floating point numbers, and as many floating point operations are performed on these coefficients (e.g., during conjugation with an R_z gate), it is crucial to account for the error margins that accumulate from floating point arithmetic. Initially, we used a constant error margin for all comparisons, but we found that this approach was not robust. In some cases, the error margin was too large, leading to false positives, while in other cases it was too small, resulting in false negatives.

To address this we implemented a dynamic error margin that scales with the number of operations performed on a floating-point number. Specifically, we designed a wrapper around the standard 64-bit floating-point type that tracks the number of floating-point operations applied to each value. Let $N(x)$ be the counter associated with the number of operations on the floating-point number x. For two floating-point numbers x and y, we consider them equal if the absolute difference between them satisfies:

$$|x - y| < \texttt{max}(\epsilon, \epsilon \cdot (N(x) + N(y)))$$

where ϵ is the maximum error margin for a single floating-point operation, determined by the machine precision. This approach provides a sound overapproximation of the accumulated error. By increasing the margin proportionally to the number of operations, we ensure that the total error margin is large enough to account for all accumulated errors from previous operations.

The downside of this approach is that it can lead to two numbers being incorrectly considered equal when they are not, especially as the cumulative error margin grows. Generally, this is not an issue: the exponential growth of summands limits the number of gates that can be applied to the generators before the equivalence verification process becomes prohibitively slow, preventing the error margin from becoming excessively large. However, incorrect results can still occur. In particular, overapproximation can cause errors when we remove summands during circuit

simulation. To save storage, we remove summands whenever their coefficients can be considered zero based on the method above. Sometimes, due to rotational gates (e.g., through the multiplication of the sine and cosine of an angle), a summand's coefficient may become very small. In such cases a summand could be removed, not because it is truly zero, but because it is incorrectly considered so. This can lead to errors, for instance, if the removed summand would have canceled out another later on.

To address this issue, an additional field is introduced in the wrapper to store the cumulative product of all multiplications from rotational gates. This allows us to track and determine when a coefficient is close to zero due to successive multiplications. The 'FloatingPoint' wrapper now has the following fields:

```
struct FloatingPoint {
    value: f64;
    ops: usize;
    cumulative_product: f64;
};
```

When checking if a coefficient equals zero, we use the `cumulative_product` field to determine whether this product has become very small compared to the error margin. If it has, it is likely not actually zero but simply very small. However, a coefficient might still be zero as a result from other operations (e.g., addition). Therefore, to ensure that we do not keep summands that are actually zero, if the actual 'value' is smaller than the smallest positive float, we can confidently treat it as zero. Determining if a coefficient equals zero is now done as follows:

```
fn FloatingPoint::is_zero(self) -> bool {
    if (self.value.abs() < 2f64::MIN_POSITIVE)
        return true;

    // Overestimate by a factor of 100 to avoid
    // boundary cases, e.g., by floating-point
    // errors in the cumulative product.
    let error_margin = 100 * f64::EPSILON * self.ops;
    if (self.cumulative_product.abs() < error_margin)
        return false;

    return self.value.abs() < f64::EPSILON * self.ops;
}
```

It is important to note that this approach does not guarantee correct results. This is something we discuss in more detail in Chapter 4.

3.3 Generalized Stabilizer ADT Implementations: Data Structures

In the following sections we will discuss the various data structures we developed that implement the functionality of the generalized stabilizer ADT.

3.3.1 Map

The map data structure stores all the Pauli string summands of a generalized stabilizer in a key-value fashion. The key is the Pauli string itself and the value is its associated coefficient. The data structure always keeps two equally sized maps in memory to avoid the continuous allocation of new maps. We refer them as the *source* and *destination* map. Below we present a representation of the data structure:

```
struct GS_Map {
    src_summands: map<PauliString, float64>;
    dst_summands: map<PauliString, float64>;
};
```

The source map always contains the Pauli string summands of the generalized stabilizer it represents, while the destination map is used to store the conjugated summands from the source map. Whenever we conjugate all the Pauli string summands, the source and destination maps are swapped afterwards and the destination map is cleared: its memory is not deallocated, but it only marked as empty.

Consider the case where the data structure represents the generalized stabilizer $0.8ZX + 0.6ZZ$. When we conjugate the generalized stabilizer with a T gate targeting the second qubit each summand of the generalized stabilizer is conjugated individually. First, the summand $0.8ZX$ is conjugated from the source map, which produces two Pauli strings: $\frac{0.8}{\sqrt{2}}ZX$ and $\frac{0.8}{\sqrt{2}}ZY$. As both Pauli strings are not yet present in the destination map, we add the Pauli strings as keys and associate their coefficients as values. Second, we conjugate $0.6ZZ$ from the source map which remains unchanged and can be directly stored in the destination map. Now all the summands are conjugated, we swap the source and destination map and clear the memory of the latter. This example is depicted in Figure 3.2.

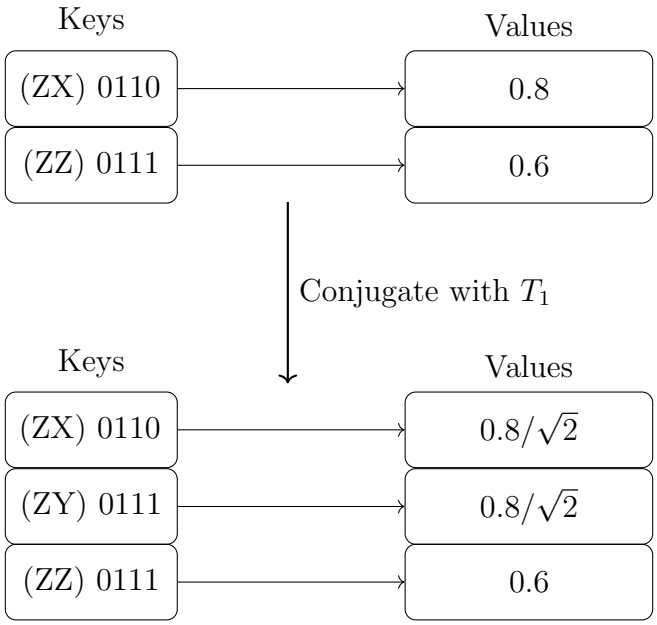

Figure 3.2: Example of conjugating a generalized stabilizer represented by the map-based data structure: The Pauli string summands from the source map (top) are conjugated and stored to the destination map (bottom).

3.3.2 Row-wise Bitvector

The row-wise bitvector data structure manges two vectors: one bitvector which sequentially stores all the Pauli string summands of the generalized stabilizer it represents and one vector containing the coefficient lists associated to each of the summands. The coefficients are associated to the summands by their index, this means that the i^{th} coefficient stored in the vector is associated to the i^{th} Pauli string stored in the bitvector. Below we present a representation of the key fields in the data structure:

```
struct GS_RowWiseBitvec {
    summands: bitvec;
    coefficients: vec<float64>;
};
```

When conjugating stored Pauli string summands, we iterate over the bitvector and extract the relevant bits from each Pauli string (i.e., the bits representing the target matrix). If a $CNOT$ conjugation is being performed, the bits of each Pauli string are updated in place, and the coefficient list associated with the Pauli string is updated accordingly. In the case of an R_z conjugation that produces two Pauli strings, we copy the Pauli string that is being conjugated and its associated coefficient and append it to the end of the bitvector and coefficient

vector, respectively. We set the target matrix of the original Pauli string to an X matrix and that of the copied Pauli string to a Y matrix. We can do this to because it does not matter whether the original target matrix was an X or Y matrix, because in both cases will yield the same two Pauli strings. However, it can be the case that the resulting Pauli strings have different coefficients so both coefficients are updated independently. The reason we implemented it in this way is because it prevents us from having to use an `if` statement. Substituting `if` statements with code that always executes is called *predication* and can improve the performance of the code [36].

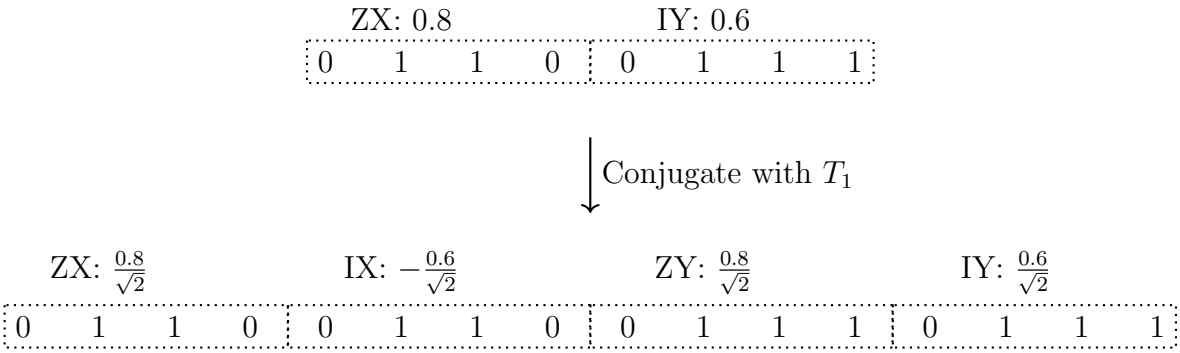

Figure 3.3: Example of conjugating a generalized stabilizer represented by the row-wise bitvector data structure. Above and below the centered arrow the sequence of bits represent the bitvector that stores the Pauli string summands of the generalized stabilizer before and after conjugation with the gate T_1, respectively. Each sequence of bits that represents a Pauli string is surrounded by a dotted box and labeled with the Pauli string it represents and its associated coefficient.

Consider the example depicted in Figure 3.3. Here we can see that our generalized stabilizer consists of two summands: the Pauli strings $0.8ZX$ and $0.6ZY$, which will be conjugated with the gate T_1. First, the Pauli string $0.8ZX$ is conjugated. As the second qubit is an X matrix the conjugation produces two Pauli strings: $\frac{0.8}{\sqrt{2}}ZX$ and $\frac{0.8}{\sqrt{2}}ZY$. We copy the bits representing the ZX Pauli string to the end of the vector and set the second matrix of the original Pauli string to an X matrix (which it already was) and that of the copied Pauli string to a Y matrix. The coefficients are also copied and updated. Then we will conjugate the second Pauli string, which will also yield two Pauli strings: $-\frac{0.6}{\sqrt{2}}IX$ and $\frac{0.6}{\sqrt{2}}IY$. Again, we copy the bits of the Pauli string representing IX, append it to the bitvector, set the second matrix to of the original an copied Pauli string to X and Y respectively and copy and update the coefficients.

Notice, that because Pauli strings are updated in place and new Pauli strings are simply appended to the end of the bitvector it is possible that the same Pauli string is stored more than once. For this reason it is important to periodically remove duplicate Pauli strings from the data structure. We achieve this by sequentially

inserting each Pauli string and its associated coefficient into a map as key and value, respectively. If we want to insert a Pauli string into the map that has already been inserted (i.e., we have encountered a duplicate) we simply sum the coefficients. After having gathered all unique Pauli strings and their coefficients into the map, we scatter them back into a cleared bitvector and coefficient vector. The memory of the vectors is not deallocated, but only marked as empty. We iterate over the key-value pairs in the map and append the Pauli strings to the bitvector and the coefficients to the coefficient vector. Any Pauli string that has a coefficient of zero is ignored.

3.3.3 Column-wise Bitvector

The column-wise bitvector was initially implemented to optimize the conjugations of Clifford gates. It is inspired by DuckDB [37] which uses a columnar storage format: instead of storing the rows of a SQL table sequentially in memory, the columns of the table are stored separately. The idea behind this is that in most cases we only want to access the values of a few columns, in which case iterating solely over the data in the columns as opposed to loading an entire row can be much more efficient; we don't load unnecessary data and it decreases cache misses. The same holds for conjugating Pauli strings with Clifford gates: we do not need to load the entire Pauli string to read and update one or two matrices.

If we are simulating n-qubit circuits, the Pauli strings we store will have a length of n. Therefore, we maintain n bitvectors, which we will refer to as "columns". The i^{th} bitvector stores the i^{th} Pauli matrix of each Pauli string. Additionally, we manage a separate vector that keeps track of the coefficients associated to each of the Pauli strings. Below, we provide a representation data structure:

```
struct GS_ColWiseBitvec {
    columns: array<bitvec>;
    coefficients: vec<float64>;
};
```

This approach is advantageous if we only need to access very few gates of each Pauli string. For example, if we want to conjugate the Pauli string summands with a CNOT with control and target qubit 4 and 5 respectively, then we only have to iterate over the fifth and sixth bitvectors and can ignore the others. In addition, as we know we will need to read and potentially update all the gates stored in these bitvectors, so the subsequent bits that we read will likely be in the cache.

The primary limitation of this approach becomes apparent when we need to access the an entire Pauli string. Say we want to read the entire j^{th} Pauli string. Then we would have to read and concatenate the j^{th} matrix stored by each bitvector. As each bitvector has a separate memory location, this makes the memory access very inefficient. Consider the example in Figure 3.4. We have stored two Pauli strings

$0.8ZX$ and $0.6IY$. We iterate over the matrices stored in the second column as the T gate targets the second qubit. The first matrix we encounter is an X matrix (belonging to the Pauli string ZX), so the conjugation will produce two Pauli strings. We now need to copy the first Pauli string and store it in the data structure. To achieve this, we copy the first matrix stored in each column (or bitvector) and append it to that column. In this case a Z is appended to the end of the first column and a X to the second. We set the second matrix of the original Pauli string to X and the second matrix of the newly stored Pauli string to Y. We clone the associated coefficient and update it accordingly. We perform an analogous process for when we encounter the Y matrix from the IY Pauli string.

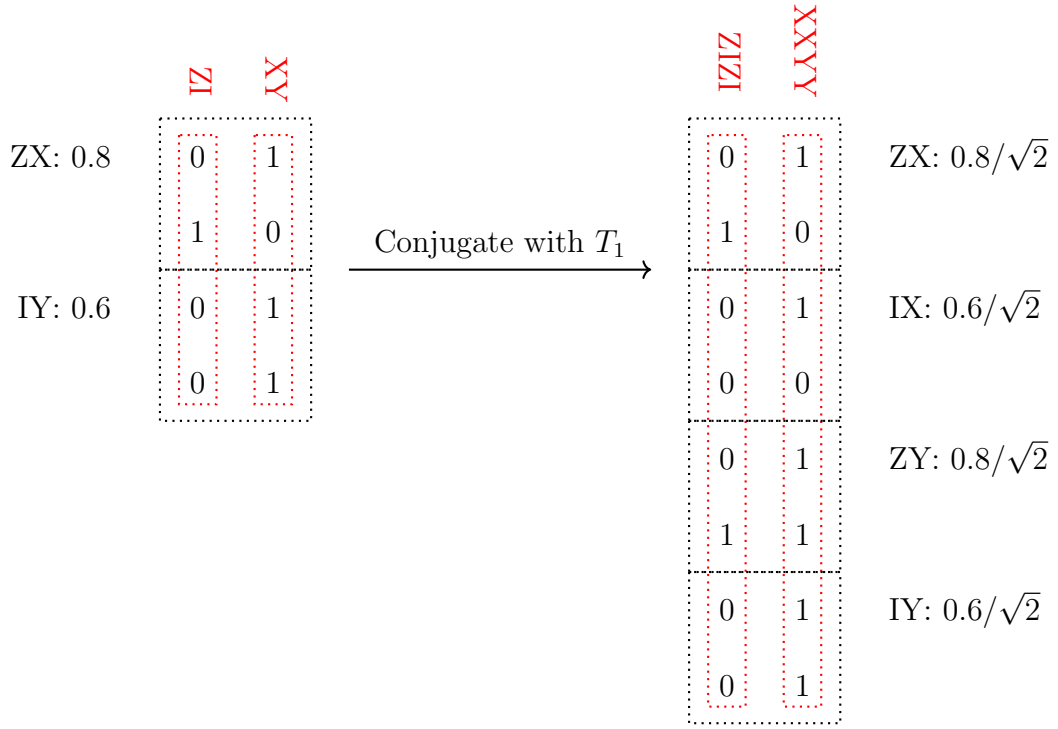

Figure 3.4: Example of conjugating a generalized stabilizer represented by the column-wise bitvector data structure. To the left of the centered arrow we can see how the data structure stores the two Pauli strings summands 0.8ZI and 0.6XY and to the right of it how they transform after a conjugation with a T_1 gate. Each dotted red box represents a column in the data structure and its label indicates the stored Pauli matrices. The black dotted boxes indicate how the columns are combined to form a Pauli string, which are represented next to them with their associated coefficient.

Similar to the row-wise bitvector it can occur that duplicate Pauli strings are stored in it. Removing these duplicates is done in a similar fashion; we gather all Pauli strings in a map and scatter them back in the data structure. For details see Section 3.3.2.

It may seem that this data structure is only beneficial when conjugating the Pauli strings with $CNOT$ gates, but is was implemented before we started tracking H and S conjugations separately (see Section 3.2.2). It was specifically designed for circuits with predominantly Clifford gates, where the benefits of conjugating these gates would outweigh the drawbacks of conjugation non-Clifford gates.

3.3.4 Tree

The tree based data structure is based on the work in [38]. In this data structure, each Pauli string is stored as binary tree. The goal of this data structure is to be as memory efficient as possible. The nodes and leaves of the tree are stored in a table and their storage location is determined by hashing. This way, if two different trees share a common subtree, the common subtree is only stored once. Each tree is divided among three tables:

- **Leaf table**: The table is a bitvector with preallocated memory where all the leafs of the tree are stored. Each entry in the leaf table stores a number of Pauli matrices and one additional bit to indicate whether the bits in the leaf table are taken which will prevent them from being overwritten when other leaves are inserted. The number of Pauli matrices per leaf can be set upon initialization (`pmatrices_per_leaf`).
- **Node table**: The table is a bitvector with preallocated memory where all the internal nodes of the tree are stored. Each entry in the node table consists of two "bookkeeping" bits and a number of "body" bits (`node_body_bits`). In the body bits we store two integers, which represent the offsets in either the node table or the leaf table of the child nodes or leaves. One of the bookkeeping bits is used to indicate this distinction. This offset can be used to find a child node or a leaf. The other bookkeeping bit is set once a node is inserted into the table to mark the bits in the table as occupied.
- **Root table**: The table is an expandable bitvector where each entry stores an offset to the root node of a tree in the node table. This way each entry represents a Pauli string and allows us to keep track of more Pauli strings than we actually store in the node and leaf table. The root table is also used to associate a coefficient with each of the stored Pauli strings.

A representation of the data structure is provided below and an example of how these tables are used to store the generalized stabilizer $0.8ZXXXZY + 0.6ZYXXZY$ is depicted in Figure 3.5.

```
struct GS_Tree {
    coefficients: vec<float64>;
    root_table: bitvec;
    node_table: bitvec;
    leaf_table: bitvec;
};
```

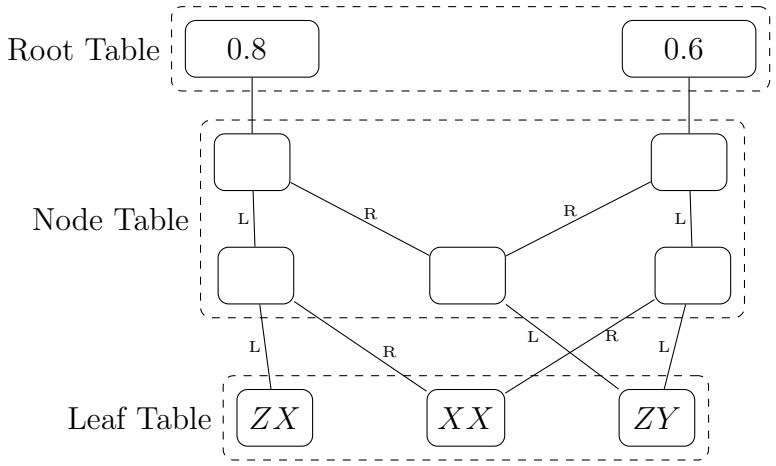

Figure 3.5: Example of storing the generalized stabilizer $0.8ZXXXZY + 0.6ZYXXZY$ using the tree based data structure. The dotted rectangles depict the various tables. Rectangles in these tables represent the entries. The edges from the nodes in the node table contain labels to indicate if the edge is to a right (R) or left (L) child. In the entires of the root table the associated coefficients are depicted, in the entries of the leaf table the stored Pauli matrices.

Both the number of body bits of a node, which we will denote with n, and number of Pauli matrices per leaf, which we will denote with p, can be set upon initialization of the data structure. By default they are set to 24 and the number of qubits divided by four (with a lower bound of 2), respectively. The number of body bits of the nodes predominantly determines the structure of the tree. Since the body bits store two offsets, each individual offset is a number stored within $n/2$ bits. This means that we can store a maximum of $2^{n/2}$ nodes in the node table. Since each node contains $2 + n$ bits we allocate $2^{n/2}(2 + n)$ total bits. As we can address an equal number of leaf nodes and each leaf consists of $2(p + 1)$ bits — two bits per Pauli matrix and one to indicate wether the entry is taken — we allocate $2^{n/2} \cdot 2(p + 1)$ bits for the leaf table.

Note that it is not strictly necessary to have two separate tables for the leaves and for the nodes, however, we have chosen to use two different tables in order to size the leaves and nodes differently. Alternatively, we could have stored the nodes and leaves in a single table, and padded the leaves with zeros to make them the same size as the nodes. However, as we are trying to optimize memory usage, we have

chosen to store the nodes and leaves in separate tables, therefore eliminating the need for pad bits.

Inserting Pauli strings

Inserting a Pauli string is done in a bottom-up fashion. We recursively split the Pauli string into two halves until we reach leaf-sized segments (Pauli strings of length `pmatrices_per_leaf`). These segments are inserted into the leaf table and the offset (or index) of the table entry they are inserted in is returned. Two of these offsets are then combined to form a node, which is inserted into the node table. As the recursion unwinds, each inserted node returns its offset which we pair to form new parent nodes. This process of combining child node offsets continues up the call stack, creating and inserting parent nodes, thus constructing the complete tree structure from leaves to root.

To insert a leaf we hash the bit representation of the Pauli matrices we want to insert and take the hash modulo the maximum number of storable leafs. This gives us an offset within the leaf table. We insert the bits in the leaf table using *linear probing*. This is a collision resolution technique where, upon encountering an occupied table entry, we check subsequent entries sequentially until we find a suitable entry. Specifically, in our case there are three possibilities that can occur at the determined offset:

1. The entry is not marked as taken. We write the bits to that location, mark it as taken and return the offset of the table entry we have written.
2. The entry is marked as taken and the bits at the offset are equal to the bits we want to insert. We return the offset of the table entry without performing any other operations.
3. The entry is marked as taken and the bits at the offset do not equal the bits we want to insert. We increment the offset and repeat this process until we find an entry that satisfies one of the first two conditions.

Nodes are inserted in a similar fashion into the node table as the leafs, but there we determine the hash of the body bits of the node and take this hash modulo the maximum number of storable nodes. This offset is used to insert the node with linear probing and the actual offset of the entry that the node was inserted in is returned. The final offset that is returned is simply appended to the root table, no hashing is performed to insert it.

Updating Pauli Strings

To update a specific matrix in a Pauli string we first recursively traverse the tree to find the leaf that contains the matrix we want to update. Because we know how many Pauli matrices are stored in each leaf, at each recursive call we know which subtree we need to continue to traverse. Once we arrive at the leaf, we copy the Pauli matrices stored in the leaf to a new leaf variable and update the matrix we want to change. We write the new, updated leaf back to the leaf table and return

its offset. Now we need to propagate the changes up the tree. As the recursion unwinds we update each node by reading the body bits of the node into a new node, updating one of the two offsets to store the offset of the newly updated leaf or child node and inserting the node into the node table. The offset of the written node is returned up the call stack, where it is used to update the parent node. This process continues until we reach the root node.

It is important to note that whenever we update a leaf or a node we do not update the entry in place, but we insert a new entry in the table. This is because we do not know whether any other node is pointing to the leaf or node we want to update. If we would update the entry in place, we would potentially break the structure of another tree.

Conjugations

To conjugate the stored Pauli strings with a $CNOT$ gate, we iterate through the root table and for each entry we find and update the target Pauli matrices through the process described above. To ensure that the tree only stores unique root table entries we take a slightly different approach to perform R_z conjugations. We initialize an empty map where we store a root table entry and its location in the root table as a key-value pair. For each entry in the root table we recursively check whether the target matrix is an X or Y matrix. If this is not the case, we simply leave the entry unchanged but add its location in the map. If this is the case we update the target matrix of the Pauli string that the entry references to either an X or Y matrix depending on whether the target matrix of the currently stored Pauli string is a Y or an X matrix, respectively. This update will not remove the existing tree structure or its root table entry, but will only add additional nodes and leaves to ensure that the updated Pauli string is stored. This means that the root table entry of the original entry is still valid and that there will be two trees present representing the original and updated Pauli string. The update function returns a reference to the latter, which will serve as the root table entry for the updated Pauli string. If the entry is already present in the map, we simply sum the coefficient of the newly obtained Pauli string with that of the existing entry, and if it is not present we append the entry to the root table.

Garbage Collection

It is possible for leafs or nodes to be present in their respective table without being referenced by any other nodes. This is a result of performing updates on the stored trees. If there are many unreferenced entries in the tables, this can considerably slow down inserting entries or even prevent an entry from being inserted if the table is full. To ensure we don't encounter these problems we will sporadically perform garbage collection by removing all unreferenced nodes. We maintain counters for both the node and leaf tables, tracking the number of entries in each. Based on these values, we initiate garbage collection at two specific points:

1. Whenever the number of entries in the node or the leaf table exceeds 80 percent of the maximum storable entries in their respective table.

2. Before inserting or updating a Pauli string we check if there are enough available entries to guarantee we can complete the operation, if this is not the case we initiate garbage collection. As we can easily compute the maximum number of nodes and leafs that will be inserted in the table, we can easily check if we will not exceed the maximum available entries of each table.

Garbage collection is performed by recursively retrieving each stored Pauli string and inserting them into newly allocated tables. Inserting the entire Pauli string as opposed to merely copying the referenced nodes and leafs is important, because the hashes of the nodes and leafs are then recomputed. After inserting all Pauli strings, the current tables are deallocated and replaced by the newly populated ones. This process ensures that the node and leaf tables contain only the nodes and leaves that are referenced.

Resizing

It is possible that even after garbage collection the node or leaf table is too full (i.e., we cannot perform an insert or update operation). In this case we need to resize the data structure. We do this by allocating new, larger tables and inserting each stored Pauli string into these new tables. Specifically, each time we resize we increase the number of body bits of a node by 8 (recall that the size of the all three tables follows from this value). The current tables are then replaced by the newly populated tables.

Chapter 4

Experiments

In this chapter we evaluate the performance of our equivalence verification tool, *CTQC*. This tool implements the data structures introduced in Chapter 3 and applies them to verify equivalence using the algorithm detailed in the same chapter. The choice of data structure can be manually specified, but if none is provided, the map-based implementation is selected for non-Clifford circuits, while the row-wise bitvector implementation is used for Clifford circuits. We begin by evaluating the performance of each data structure, followed by benchmarking the "out-of-the-box" version of CTQC against existing equivalence verification tools QCEC [14], developed by the TU Munich, and ECMC [21] from Leiden University. The results of the data structure comparison are discussed in Section 4.1 and the results of the benchmarking with other tools in Section 4.2. All circuits used in the benchmarks were obtained from the MQT Benchmark set or generated by the accompanying code [39] and then transpiled and optimized using Qiskit. All benchmarks where run on a Dell XPS 15 with an Intel i7-9750H chip.

4.1 Evaluating the Performance of Proposed Data Structures

We first compare the performance of our tool, which implements the proposed generalized stabilizer generator data structures and equivalence verification method discussed in Chapter 3. The full results can be found in Appendix A but a subset of the results is presented in Table 4.1. We specifically benchmarked two properties: *runtime* and *memory usage*. The runtime measures the total time it takes for the tool using one of the implemented data structures to verify equivalence, including the time it takes to read the circuits and perform any preprocessing steps. The memory usage is measured in terms of the maximum RSS (Resident Set Size) in megabytes. The maximum RSS is the peak amount of physical RAM a program uses during its execution. It's a good measure for memory usage because it shows the actual impact on system resources, reflecting how much memory is actively in use.

To compactly present the results we use *heatmaps* as shown in Figure 4.1. These offer a concise way to display the performance of a data structure relative to the others. The "relative performance" of each data structure implementation is determined per algorithm rather than per circuit. Performance, based on runtime

Table 4.1: Benchmark results of the various data structures for the largest DJ, GHZ and Graph state circuits and a representative subset of the non-stabilizer circuits from the MQT benchmark set.

Algorithm	#Qubits	Map		Col-wise BV		Row-wise BV		Tree	
		t(sec)	mem(mbs)	t(sec)	mem(mbs)	t(sec)	mem(mbs)	t(sec)	mem(mbs)
Deutsch-Josza	2^{11}	14.4055	13.9	2.2259	22.0	1.731	9.3	16.2668	22.0
	2^{12}	123.141	16.3	11.1899	62.8	7.8248	13.3	107.1632	38.7
	2^{13}	T	T	53.195	216.9	31.3367	23.5	T	T
GHZ State	2^{12}	124.6941	20.7	5.8267	60.1	2.3891	11.2	125.2532	47.3
	2^{13}	T	T	33.1645	212.9	10.649	16.3	T	T
	2^{14}			222.8422	802.8	42.5643	27.4		
	2^{15}			T	T	176.4379	53.0		
Graph State	2^{12}	127.7919	15.1	8.9611	61.5	5.4273	12.2	108.3978	37.1
	2^{13}	T	T	43.0963	214.5	22.3909	18.6	T	T
	2^{14}			253.1204	807.3	90.0806	32.1		
	2^{15}			T	T	364.7836	60.7		
QAOA	2^3	0.0068	0.0	0.0027	0.0	0.038	5.4	0.0025	0.0
	2^4	0.0071	5.1	0.0041	0.0	0.0365	5.1	0.0044	0.0
Quantum Neural Network	2^3	27.0075	642.2	15.0561	238.5	322.8196	337.2	20.7199	647.0
	2^4	M	M	M	M	-	-	M	M
Exact Quantum Phase Estimation	2^3	0.7183	74.8	0.2531	13.4	5.6078	18.0	0.5871	60.3
	2^4	M	M	M	M	-	-	M	M
SU Random Circuit	2^3	15.6331	647.8	9.5276	233.7	289.0859	280.5	11.9401	573.2
	2^4	M	M	M	M	-	-	M	M
W State	2^3	0.0051	0.0	0.002	0.0	0.0163	0.0	0.0051	0.0
	2^4	0.0157	0.0	0.0031	0.0	0.0444	5.1	0.0118	0.0
	2^5	M	M	M	M	-	-	M	M

or memory usage, is determined as follows: For each circuit, the implementation with the lowest time or memory usage receives a score of 1, the next lowest a score of 2, and so forth. These scores are summed for each implementation across all the circuits of a specific algorithm, with lower scores indicating better performance. This method provides a compact overview of implementation performance across the benchmarks, but it is important to keep in mind that it may not capture detailed performance variations of the individual circuits

Discussions. Figure 4.1 shows that the map based implementation outperforms the others in terms of both time and memory usage for most circuits. This advantage is likely because the map based implementation only stores unique Pauli strings as bitvectors, which can easily be retrieved and conjugated, whereas the other implementations may store duplicates or require tree traversal for conjugation. Furthermore, the map based implementation performs especially well for circuits where equivalence can verified up to 16 qubits. In these circuits the generators are relatively small and can be hashed efficiently.

Notably, there are only three algorithms for which any of the implementations can verify equivalence for circuits with more than 16 qubits: the Deutsch-Josza, GHZ and Graph state algorithms. For these algorithms, even significantly larger circuits can be verified, as is shown in Table 4.1. The strong performance on the GHZ and Graph state circuits can be easily attributed to the fact that both are Clifford circuits. We already know that the (generalized) stabilizer formalism is particularly effective for Clifford circuits, so the performance for these circuits is unsurprising. The more interesting result is the performance on the Deutsch-Josza circuits, which contain nearly 47% rotational gates. Initially, we hypothesized that

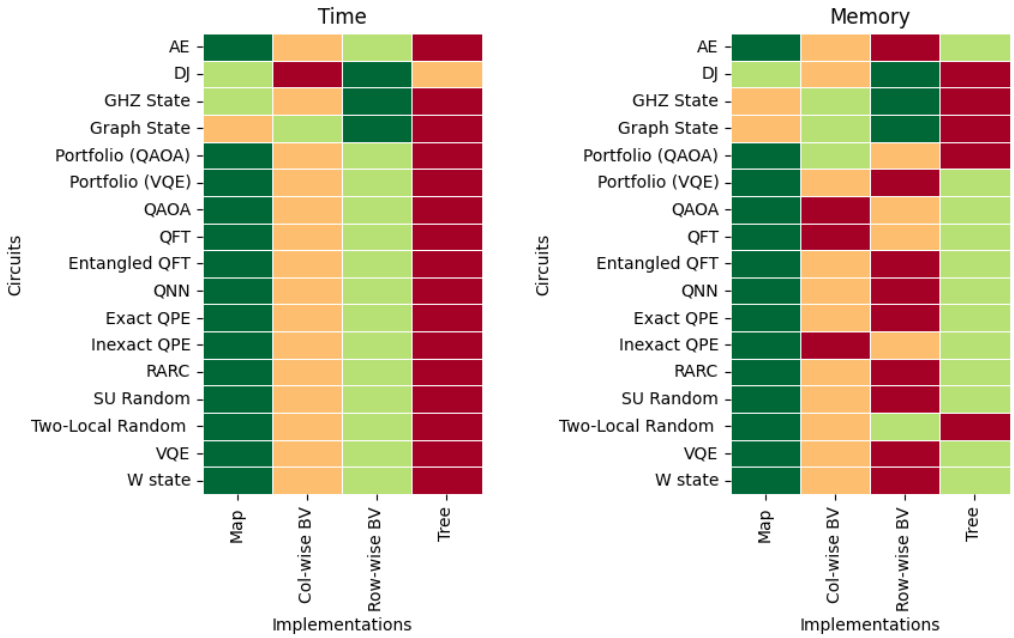

Figure 4.1: Heatmaps for the relative runtime and memory performance of various data structure implementations for the MQT benchmark set, comparing the original and optimized circuits. The algorithms are depicted the vertical axis, the implementations on the horizontal axis. Green indicates lowest time/memory usage, while red indicates the highest, with ties sharing the same color.

this performance could be due to the relatively low circuit depth, averaging 5.6 gates per qubit, which could inherently limit the growth of the number of Pauli strings. However, after further analysis, this explanation seemed unlikely. In a DJ circuit with 2^{12} qubits there are approximately $5.6 \cdot 2^{12}$ gates and as nearly half of them are rotational gates this would be enough to have the number of Pauli strings blow up to the point that the equivalence verification process becomes unfeasible. As it turns out, this behavior is due to the gates that appear in the circuits: the rotational gates in the Deutsch-Jozsa (DJ) circuits only use angles of π and $\pi/2$ modulo π. A rotational gate with the former angle equals the Z gate and with the latter angle corresponds to the $\pm S$ gate, where the sign of the S gate depends on the polarity of the sine of the angle. This means the DJ circuits are effectively Clifford circuits and which explains the performance on these circuits.

Although the map based implementation generally performs best, it does perform best across all circuit types; specifically, it is significantly outperformed by the row-wise bitvector implementation for the Deutsch-Jozsa, GHZ, and Graph state circuits. As shown in Figure 4.1, the data structure verifies equivalence faster, while requiring less memory. This advantage is most apparent for the DJ, GHZ and Graph state circuits, where it also scales to much larger circuits, as seen in

Table 4.1. The better performance of the row-wise bitvector on these circuits is likely due to the fact that these circuits are Clifford circuits and the generators will therefore always remain Pauli strings throughout the circuit conjugation. This means the row-wise bitvector allocates a fixed number of bits for each generator, with updates made in place for each conjugation, resulting in a fast, cache-friendly, and memory-efficient process. In contrast, the map based method requires rehashing the generator after each conjugation and the tree based method involves extensive memory management, both of which introduce overhead. This is also why the column-wise bitvector scales better for these circuits: for each generator, we allocate all bitvectors (columns) once, and then we only need to access the first two bits of each of the bitvectors for conjugation. Although accessing each matrix through different bitvectors — and therefore different memory locations — is less efficient than repeatedly accessing the same bitvector, it remains faster than the map and tree based methods for very large Clifford circuits. Consequently, both bitvector methods can verify equivalence for much larger DJ, GHZ and Graph state circuits.

The tree based generator set performs worst in runtime for nearly all circuits, but generally outperforms both the row-wise and column-wise bitvectors in memory usage. The slow runtime is likely due to the fact that conjugating a Pauli string cannot be done in constant time, as is the case with the other implementations, because it requires traversing the tree. The reason we believe that the tree based implementation does not outperform the map based implementation is two fold. First, in the case where equivalence could only be verified for smaller circuits (≤ 16 qubits), maintaining the tree structure costs significantly more memory than just storing the bits sequentially, thereby negating the benefits of only storing unique subtrees. Second, in the case where equivalence could be verified for large circuits (DJ, GHZ, Graph state) the generators are always Pauli strings. As a consequence the overhead of maintaining the tree structures is again likely more costly than just storing the Pauli strings as sequential bits. We surmise that for circuits where the number of Pauli string summands grow exponentially, the tree based implementation would be the most memory efficient for large circuits as it would be able to utilize its compression capabilities to an even greater extent. However, unfortunately in those cases the runtime would be prohibitively slow.

Since the row-wise bitvector outperforms the map based implementation for Clifford circuits, in the final tool we automatically select the appropriate data structure if non is specified. The row-wise bitvector is used for Clifford circuits, while the map is used for non-Clifford circuits. We will benchmark its performance against existing tools in the next section.

4.2 Evaluating the Performance against Existing Tools

We now compare our complete tool, called CTQC, with the existing tools QCEC and ECMC. The full results of all the benchmarks can be found in Appendix A, but a subset is presented in Table 4.2. Analogous to the benchmarks in Section 4.1 we benchmarked the runtime and memory usage, where the runtime includes the time it takes to read the circuits and perform any preprocessing steps and the memory usage is measured in terms of the maximum RSS usage.

Table 4.2: Benchmark results of the various equivalence verification tools for the largest DJ, GHZ and Graph state circuits and a representative subset of the non-stabilizer circuits from the MQT benchmark set.

Algorithm	**#Qubits**	**CTQC**		**QCEC**		**ECMC**	
		t(sec)	mem(mbs)	t(sec)	mem(mbs)	t(sec)	mem(mbs)
Deutsch-Josza	2^{10}	0.455	6.8	1.0208	5682.2	11.5656	1298.0
	2^{11}	1.731	9.3	M	M	43.7122	5036.4
	2^{12}	7.8248	13.3	-	-	-	-
	2^{13}	31.3367	23.5	-	-	-	-
GHZ State	2^{10}	0.1479	6.2	0.9386	5812.3	2.4931	153.1
	2^{11}	0.6239	8.1	M	M	9.6961	519.5
	2^{12}	2.3891	11.2	-	-	36.9711	1907.9
	2^{13}	10.649	16.3	-	-	149.2334	7502.5
	2^{14}	42.5643	27.4	-	-	M	M
	2^{15}	176.4379	53.0	-	-	-	-
Graph State	2^{10}	0.3727	6.4	1.0076	5522.0	169.1951	127.2
	2^{11}	1.3467	8.1	M	M	209.2035	242.2
	2^{12}	5.4273	12.2	-	-	-	-
	2^{13}	22.3909	18.6	-	-	-	-
	2^{14}	90.0806	32.1	-	-	-	-
	2^{15}	364.7836	60.7	-	-	-	-
Grover's Algorithm without Ancilla	2^{3}	-	-	0.2069	280.0	-	-
	2^{4}	-	-	-	-	-	-
Portfolio Optimization using QAOA	2^{3}	25.0507	216.3	0.0324	78.1	-	-
	2^{4}	M	M	0.0518	87.1	-	-
QAOA	2^{3}	0.0027	0.0	0.0283	78.2	-	-
	2^{4}	0.0041	0.0	0.036	86.0	-	-
Quantum Neural Network	2^{3}	15.0561	238.5	0.0339	87.6	M	M
	2^{4}	M	M	0.0513	152.6	-	-
	2^{5}	-	-	0.1052	85.4	-	-
	2^{6}	-	-	0.2954	364.7	-	-
	2^{7}	-	-	1.164	1193.7	-	-
	2^{8}	-	-	5.592	2248.3	-	-
Exact Quantum Phase Estimation	2^{3}	0.2531	13.4	0.0256	77.9	-	-
	2^{4}	M	M	0.0617	88.5	-	-
	2^{5}	-	-	-	-	-	-
W State	2^{3}	0.002	0.0	0.0256	77.8	0.133	31.3
	2^{4}	0.0031	0.0	0.0355	85.6	0.5789	49.2
	2^{5}	M	M	0.0577	288.7	2.3315	127.9
	2^{6}	-	-	0.101	579.3	7.4094	346.4
	2^{7}	-	-	0.2338	948.4	54.9538	2513.5
	2^{8}	-	-	0.6115	1673.5	579.513	7218.0
	2^{9}	-	-	1.4435	2744.6	M	M
	2^{10}	-	-	6.8719	6066.4	-	-
	2^{11}	-	-	M	M	-	-

Discussions. The heatmaps in Figure 4.2 show that our tool is outperformed by QCEC for most circuits. While our tool remains competitive for circuits with fewer qubits, often achieving similar runtimes to QCEC, it generally struggles to scale to circuits with a larger number of qubits. This fact is depicted in Figure 4.3. Here we can see that see that for most algorithms QCEC is able verify equivalence for larger circuits, while our tool and ECMC have already exceed the time limit or ran out of memory.

Although our tool does not perform well on most circuits, there are three algorithms for which it performs particularly well: the Deutsch-Jozsa, GHZ state and Graph state algorithms. Table 4.2 illustrates that our tool verifies equivalence faster for the circuits of these algorithms, while it also uses less memory. The memory usage for the largest circuits can even be significantly less: up to 800 times less than QCEC and up to 540 less than ECMC. This performance allows our tool to verify equivalence for circuits of these algorithms with much more qubits: our tool is able to verify equivalence for the largest DJ, GHZ and Graph state circuits, while, for example, QCEC runs out of memory for the circuits with 2^{10} qubits. The reason for the strong performance on these circuits is discussed in Section 4.1.

In addition to benchmarking equivalent circuits, we also evaluated non-equivalent circuits. For this, we used the same pairs of original and optimized circuits from the MQT benchmark set, but introduced four types of errors into the optimized circuits:

- *Flipped*: The control and target qubits of a random $CNOT$ gate are swapped.
- *Gate missing*: A random gate is removed from the circuit.
- *Shift-4*: An offset of 10^{-4} is added to the angle of each rotational gate.
- *Shift-7*: An offset of 10^{-7} is added to the angle of each rotational gate.

The performance of the tools on various algorithms is generally consistent with that of standard optimized circuits. However, for some circuits, the tools perform slightly better since the equivalence verification process can stop as soon as a "counter-example" is found. More interesting results from these benchmarks are shown in Table 4.3, which presents the number of correctly and incorrectly classified circuit pairs or instances where no information can be provided.

We consider a classification incorrect if the tool indicates the circuits are equivalent. Before we discuss the results it is important to discuss how we know that a classification is incorrect. After all, what if the error we introduced did not actually alter the functionality of the circuit? While formally verifying (in)correctness is out of the scope of this thesis, we have two reasons to denote certain classifications as incorrect. First, if a tool classifies a circuit pair as equivalent, but the other tools classify it as non-equivalent, we consider the former classification incorrect. Second, specifically in the case of QCEC, we verified equivalence between the original circuit and the optimized version with an error, but if for that circuit we

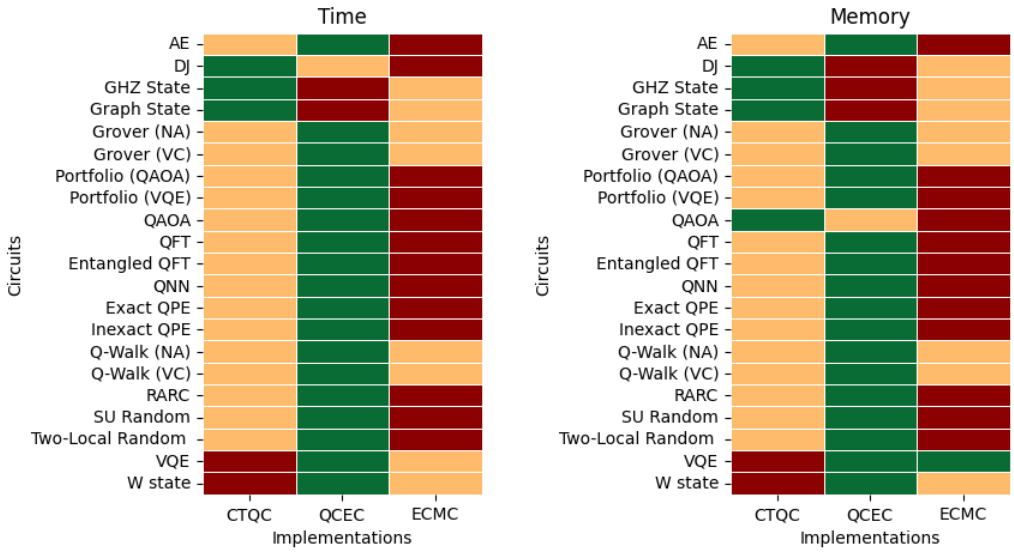

Figure 4.2: Heatmaps for the relative time and memory performance of various equivalence verification tools for circuits from the MQT benchmark set based on the equivalence verification of the original and optimized circuits. Green indicates the lowest relative time/memory usage, while red indicates the highest relative time/memory usage.

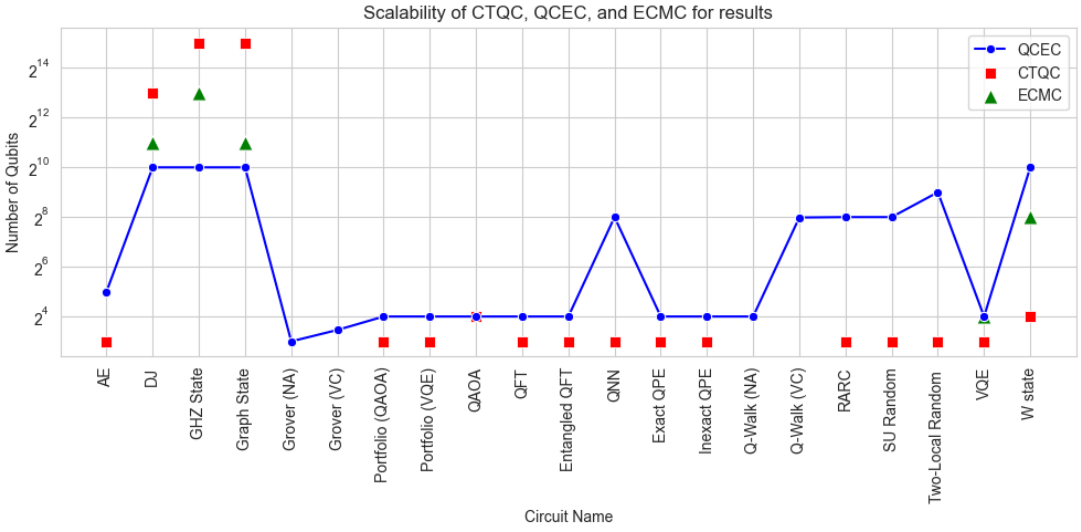

Figure 4.3: Plot showing the largest circuits (in terms of qubits) for which each tool can verify equivalence of each algorithm. The x-axis represents the algorithm, while the y-axis indicates the number of qubits in the circuit.

verified equivalence with QCEC between the optimized circuit and the optimized circuit with an error, QCEC would classify them as non-equivalent.

A classification is deemed incorrect if the tool indicates that the circuits are equivalent, while we "know" they are not. Before presenting the results, it is important to clarify how we determine whether a classification is incorrect. This is particularly relevant because the error we introduced might not necessarily affect the circuit's functionality. Although formally verifying (in)correctness is beyond the scope of this thesis, we rely on two key criteria to identify certain classifications as incorrect. First, if a tool classifies a circuit pair as equivalent while other tools classify the same pair as non-equivalent, we consider the former classification incorrect. The second criteria applies specifically to QCEC. We verified equivalence between the original circuit and its optimized version with an error and, for some circuits pairs, QCEC classified them as equivalent, but when verifying equivalence between the optimized version of this circuit and its error-introduced counterpart, QCEC classified them as non-equivalent. This indicates that the error we introduced did affect the functionality and its original classification is incorrect.

Table 4.3: Table showing the equivalence classifications for the QCEC, CTQC, and ECMC tools on non-equivalence benchmarks. For each type of non-equivalent circuit and verification tool, the table lists the number of circuits correctly identified as non-equivalent (true negatives), incorrectly identified as equivalent (false positives), or where information could not be determined (no information).

Error type	**Classification**	**CTQC**	**QCEC**	**ECMC**
Flipped	true-negative	53	50	36
	false-positive		4	
	no information		3	
Gate Missing	true-negative	53	48	34
	false-positive		4	
	no information		5	
Shift-4	true-negative	25	13	16
	false-positive			
	no information		2	
Shift-7	true-negative	25	1	8
	false-positive			8
	no information		10	

As we can see QCEC frequently fails to provide information or incorrectly classifies circuits as equivalent, while ECMC often misclassifies circuits with shift-7 errors. We believe this is due to how each tool handles floating point inaccuracies. Floating point numbers are common in quantum circuit equivalence verification, leading to situations where calculations yield only approximate results. To address this,

tolerance thresholds for floating point comparisons are necessary. The choice and application of these thresholds can significantly impact the classification. To illustrate the effects of this in more detail consider the following two circuits:

$$H R_z\left(\frac{\pi}{4}\right) R_z\left(-\frac{\pi}{4+\alpha}\right) \quad \text{and} \quad H$$

When $\alpha = 0$ the circuits are trivially equivalent. However, when $\alpha = 0.1$ they are not, and all tools correctly classify it as such. However, as we make α smaller the tools start giving wrong results. Specifically, CTQC, QCEC, and ECMC incorrectly classify the circuits as equivalent when α is 10^{-15}, 10^{-7}, and 10^{-4}, respectively. Our tool, which uses a dynamic error margin that adjusts based on the number of operations (see Section 3.2.3), may even begin to yield incorrect results sooner as the number of gates in the circuits, and thus the error margin, increases. For instance, if we add 8000 T gates between the H and R_z gates, our tool would incorrectly classify the circuits as equivalent when $\alpha = 10^{-10}$.

For ECMC is it rather apparent that the incorrect classifications are caused by floating point approximations, as all incorrectly classified circuits contain very small errors in the rotational gates, but we believe this is also the cause for QCEC's misclassifications. Although the flipped and (in some cases) gate missing errors may not appear directly related to floating point inaccuracies, we suspect they are a contributing factor in this case. The effects of the circuit errors might be very minimal and these small deviations could be misinterpreted as negligible floating point inaccuracies. Notably, all of QCEC's misclassifications occur in the W state circuits with 128 qubits or more. We believe this is due to the high number of rotational gates in these circuits, which can produce small floating point values in state representations such as the state vector or density matrix, and as the number of qubits increases so does the number of gates. Consequently, an error from a single gate may have a minimal impact on these already small floating point values. The only other circuits with rotational gates and more than 64 qubits for which QCEC does not time out or exceed memory are the DJ circuits. The reasons mentioned earlier could explain why QCEC is unable to provide information for most of these circuits. In all other cases where QCEC cannot provide any information we also think this is due to floating point inaccuracies. It can be the case that the other methods used by QCEC, such as the simulation, classify the circuits as equivalent, while the ZX-checker indicates that they are not. The ZX-checker cannot prove non-equivalence, but these false-negatives are very rare, thus throwing an exception informing us that no information can be provided.

Chapter 5

Related Work

In this chapter we discuss various related work. In Section 5.1 we describe the improvements suggested by Gottesman and Aaronson to previous stabilizer simulation methods. In Section 5.2 we discuss the STIM simulator, which is a high performant stabilizer circuit simulator. In Section 5.3 we discuss ECMC, a quantum circuit simulator and equivalence verification tool based on the stabilizer formalism. In Section 5.4 we discuss QCEC, an equivalence verification tool that uses different methods in parallel to verify equivalence. In Section 5.5 we discuss the Quipu simulator, which uses a novel concept called "stabilizer frames" to simulate arbitrary quantum circuits using the generalized stabilizer formalism. Finally, in Section 5.6 we discuss the quantum circuit simulator "Abstraqt", which is based on the generalized stabilizer formalism and limits the exponential growth of Pauli strings by compression multiple summands into a single abstract element.

5.1 Gottesman and Aaronson

In the paper "Improved simulation of stabilizer circuits" [16] Gottessman and Aaronson suggest several improvements for the stabilizer-based simulation described in paper of the Gottesman-Knill theorem [17]. As the stabilizer formalism is extensively discussed in Chapter 2, we focus here on the most important improvement they suggest: keeping track of the *destabilizers* generators in addition to the the stabilizers generators.

The destabilizers generators are all the operators that in combination with the stabilizer generators generate the complete Pauli group. Recall, that in the case of a deterministic measurement determining whether Z_i or $-Z_i$ is a part of the stabilizer requires inverting a matrix which in practice has a time complexity of $\mathcal{O}(n^3)$. Keeping track of the destabilizer generators allows us to simulate deterministic measurements in $\mathcal{O}(n^2)$ time. To explain why this is the case define $S = \{Z_0, Z_1, \ldots, Z_n\}$ and $D = \{X_0, X_1, \ldots, X_n\}$ as the ordered sets of stabilizer generators and destabilizer generators of the initial state, respectively. For these sets it holds that $s_j \in S$ anti-commutes with $d_j \in D$, but commutes with all other destabilizers. After any number of Clifford gates this fact will still hold. We know that either Z or $-Z$ is in the stabilizer, but not both, as in that case $-ZZ = -I$ would be in the stabilizer, which is a contradiction. Therefore, there exists a subset of the stabilizer generators that, when multiplied together, will yield $\pm Z$. To find which generators are part of this product we can use the fact

that Z_i will anti-commute with the j^{th} destabiliser generator if and only if the j^{th} stabilizer generator is a part of the generators that comprise the product of $\pm Z_a$.

Proof. Assume Z_i anti-commutes with the j^{th} destabiliser generator, d_j. Since d_j anti-commutes only with s_j it follows that s_j must be part of the product in order for $d_j Z_i = -Z_i d_j$ to hold. Now assume s_j is part of generators that comprise the product of Z_i. Since d_j anti-commutes with s_j and does not anti-commute with any other stabilizer generator it follows that Z_i must anti-commute with d_j. □

Now to determine whether Z_i or $-Z_i$ is in the stabilizer, we multiply all the stabilizer generators s_j for which it holds that d_j anti-commutes with Z_j. This product will produce $\pm Z$ which immediately provides us with the result of the measurement.

Tracking the destabilizers is particularly useful for simulations but not for our equivalence verification method. While verifying the equivalence of two circuits, U and V, we do verify that they coincide on conjugation on the Pauli strings $\{X_0, X_1, \ldots, X_n\}$, or the destabilizers. However, performing this verification individually for each Pauli string produces the same outcome as verifying them simultaneously, with no direct advantage gained from the simultaneous approach.

5.2 STIM

STIM [18] is a high performant stabilizer circuit simulator. It has been shown that it can be effectively used to verify equivalence between stabilizer circuits in [20]. The simulator is based upon the work from Gottesman and Aaronson presented in [16], but has three main improvements:

1. The time complexity of deterministic measurements is improved from quadratic to linear.
2. A cache friendly layout is used to store the stabilizer generators and SIMD (Single Instruction, Multiple Data) instructions are used to update them. SIMD instructions allow for the execution of the same operation on multiple data elements simultaneously, typically within a single processor cycle, which can greatly enhance performance.
3. The circuit is fully simulated up until a measurement and the reference state is used to bulk sample the measurement outcomes.

The first improvement is achieved by simulating the circuit "backwards". STIM relies on the fact that measuring an observable M after some Clifford circuit C is equivalent to measuring $C^\dagger M C$ at the start of the circuit. Consider that we want to simulate a measurement of the a^{th} qubit. If we denote ρ as the density matrix of the all-zero state and define $M_0 = \frac{1}{2}(I + Z_a)$, then the probability of measuring $|0\rangle$ for the a^{th} qubit is given by:

$$\mathrm{Tr}(\rho C M_0 C^{-1}) = \frac{1}{2} + \frac{1}{2}\,\mathrm{Tr}\left(\rho C^\dagger Z_a C\right)$$

Using the fact that the Clifford group normalizes the Pauli group — which implies that $C^\dagger Z_a C$ is a Pauli string — and the fact that the density matrix of the all-zero state is the sum of all the Pauli strings that consist exclusively of I and Z matrices, it is not difficult to show that the probability of measuring $|0\rangle$ simplifies to two cases:

1. If $C^\dagger Z_a C$ is a Pauli string with only I and Z matrices, the measurement is deterministic and the probability of measuring $|0\rangle$ is zero if the sign of the $C^\dagger Z_a C$ negative and one otherwise.
2. If $C^\dagger Z_a C$ is a Pauli string with at least one X or Y matrices, the measurement is non-deterministic and the probability of measuring $|0\rangle$ is $\frac{1}{2}$.

Notice, that in the deterministic case the complexity of the measurement is linear, as it only requires us to check whether all Pauli matrices in the observable are I or Z matrices. This is in contrast to the previous work where the complexity was quadratic.

The second improvement is achieved by storing the Pauli strings in a matrix where the i^{th} row contains the i^{th} matrix of each Pauli string. This allows contiguous memory access when conjugating specific matrices of the Pauli strings and SIMD instructions can be used to update multiple strings at once. For locality reasons, it is important that for each Pauli matrix, which is stored using an x and a z bit (See Section 3.2.1), the x and z bits are stored together. This is in contrast to the approach in [16], where first all the x bits are stored and subsequently all the z bits. Finally, STIM uses a dense representation of the matrix, that is, all bits are stored explicitly. These aspects allow predictable, sequentially memory access allowing for efficient cache usage and SIMD instructions.

The third improvement is achieved by fully simulating the circuit up until a measurement and using this reference state to bulk sample the measurement outcomes. STIM can perform hundreds of simulations in parallel due to its use of SIMD instructions, making bulk sampling very efficient.

5.3 ECMC

ECMC [19, 21] is a quantum circuit simulator and equivalence verification tool of arbitrary quantum circuits based on the stabilizer formalism. We compared the performance of ECMC with our method in Chapter 4. Instead of explicitly storing the stabilizer generators, a symbolic representation is used. Pauli strings are encoded using $2n+1$ boolean variables. The j^{th} matrix in the string is encoded by the variables x_j and z_j and a single variable r is used to encode the sign. These variables are combined to a single boolean formula by concatenating them

with the "and" operator. A Pauli string can be recovered from this formula by determining a satisfying assignment, where the assignment of a and a' to x_j and z_j represents the Pauli matrix $X^a Z^{a'}$ and the assignment of 0 or 1 to r represents 1 or -1, respectively. For example, the Pauli string $-ZY$ is encoded by the formula $r\bar{x_1}z_1x_2z_2$ as its only satisfying assignment is $r = 1$, $x_1 = 0$, $z_1 = 1$, $x_2 = 1$ and $z_2 = 1$. Conjugations of Clifford gates can easily be performed by negating variables, but T gate conjugations are slightly more complex. Whenever a Pauli string is conjugated with T_j and x_j is present in the boolean formula of the Pauli string (i.e., the j^{th} matrix is either an X or Y matrix), we know the result will produce a sum of Pauli strings. Both can still easily be represented by a boolean formula, but an additional variable u_i is added to indicate that the i^{th} matrix conjugation produced this string, and as it is part of a sum it thus has a coefficient of $\frac{1}{\sqrt{2}}$. For example, the result of the conjugation of $-ZY$ with T_1 is represented by the formulas $r\bar{x_1}z_1x_2z_2u_i$ and $r\bar{x_1}z_1x_2\bar{z_2}u_i$, representing the Pauli strings $-\frac{1}{\sqrt{2}}ZY$ and $-\frac{1}{\sqrt{2}}ZX$, respectively. This method easily extends to the rotational gates, but then two additional variables are used, u_{i1} and u_{i2}, which, upon an assignment of 1, represent the cosine and sin of the angle of the gate, respectively.

This symbolic approach is combined with *weighted model counting* (WMC) to perform equivalence verification. WMC is a generalization of model counting. In model counting, we determine the number of satisfying assignments for a Boolean formula φ. Given φ with n Boolean variables, there are 2^n possible truth assignments, and model counting identifies how many of these satisfy φ. In WMC, each assignment $\vec{x}$ is associated with a weight $W(\vec{x})$, and the goal is to compute the weighted sum of all satisfying assignments. For a formula φ with satisfying assignments $\vec{x} \models \varphi$, WMC computes:

$$\sum_{\vec{x} \models \varphi} W(\vec{x})$$

In ECMC the weight function W is defined as $W(r_k^l) = -1, W(\overline{r_k^l}) = 1, W(u^0) = W(u^1) = \frac{1}{\sqrt{2}}$ and $W(\overline{u^0}) = W(\overline{u^1}) = 1$. Given a unitary operator U, any Pauli string P^0 and define $P^m = UP^0U^\dagger$. Let F_{P^m} be its symbolic representation as previously discussed. In [21] it is shown that, using this weight function, the weight of any Pauli string summand P of P^m can be obtained via $MC_W(F_{P^m} \wedge F_P)$, where MC_W denotes performing weighted model counting with the weight function W, and F_P is the symbolic representation of P. This allows us to compute $\frac{1}{2^n}\operatorname{Tr}(P^m P)$. Furthermore, it is demonstrated that if $\frac{1}{2^n}\operatorname{Tr}(P^m P) = 1$, then $P^m = P$. Combining this with Theorem 1, equivalence verification through weighted model counting reduces to verifying that for all $j \in \{1, 2, \ldots, n\}$ and $G \in \{Z, X\}$, the following holds:

$$MC_W(F_{G_j^m} \wedge F_{G_j}) = 1$$

5.4 QCEC

QCEC [40] is a quantum circuit equivalence verification tool. We compared the performance of QCEC with our method in Chapter 4. The tool verifies equivalence based on two different methods.

The first method is proposed in [14] and is based on the use of decision diagrams. Decision diagrams store data in a tree structure where each node corresponds to a variable and the edges denote possible values for these variables. The value found at a leaf node, obtained by traversing a specific path through the tree, represents the value associated with the variable assignment defined by that path. In decision diagrams equivalent subtrees are merged, which often results in a very compact representation of the data. Matrices can be represented using decision diagrams by recursively dividing the matrix into four submatrices. A node in the decision diagram represents a (sub)matrix and an edge from that node will yield a node representing on of its four submatrices or a leaf containing an actual complex numbers from the entire matrix. An example of the identity matrix stored using a decision diagram is provided in Figure 5.1. Not only can matrices be represented using decision diagrams, but matrix operations can also be defined on them. This allows them to be used for tasks such as simulating quantum circuits.

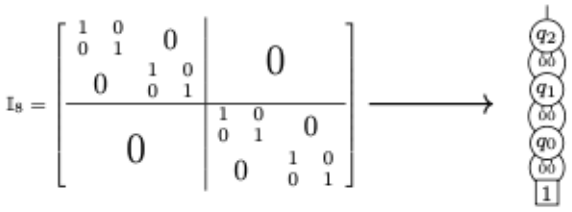

Figure 5.1: Example of the identity matrix stored as a decision diagram. Taken from [14].

The method relies on verifying equivalence in two stages. In the first stage, simulations using decision diagrams and random inputs are performed to check whether both circuits produce the same state. This stage can not guarantee equivalence, unless the entire input space is simulated, but a counterexample can be discovered, showing that the circuits are not equivalent. The authors of [14] show that contrary to classical circuits, where the chance of finding a counterexample using this approach is very small, this method can be quite effective for quantum circuits. If no counterexample is found, QCEC moves on to the second stage, referred to as "alternating equivalence verification". In this stage, the authors use the fact that decision diagrams are an efficient data structures for representing sparse matrices, such as the identity matrix, and they leverage a fundamental property of quantum circuits: *reversibility*. The authors note that if two circuits, U and V are equivalent, then sequentially multiplying the gates of U and V^{-1} should yield the identity gate again, i.e., $U \cdot V^{-1} = I$. However, the key observation is that this can expressed as follows: $U \cdot V^{-1} = U \cdot I \cdot V^{-1}$. Due to the associativity of matrix multiplication

gates of U or V^{-1} can be multiplied with the identity gate in arbitrary order, as long as the gates of U are "applied from the left" and the gates of V^{-1} are "applied from the right". If the circuits are equivalent it is possible to apply the gates in such a fashion that will frequently yield the identity gate. Since sparse matrices (e.g. the identity matrix) can be efficiently represented by decision diagrams and matrix multiplication can be efficiently implemented for them, this allows for a very fast way of verifying equivalence. It is important to note however that if the two circuits are not equivalent the first stage will not always find a counterexample. In this case the decision diagram representing the "middle" matrix can still grow exponentially, prohibiting fast equivalence verification.

The second method is proposed in [22] and is based on ZX-calculus. ZX-calculus is a graphical language that can be used to represent and reason about quantum circuits using diagrams made up of "spiders" connected by "wires". *Spiders* are nodes in a diagram that represent specific quantum operations, and *wires* represent qubits or quantum states to which these operations are applied. A Z spider is depicted by a green node and represents operations in the Z-basis and a X spider is depicted by a red node and represents operations in the X-basis. Which specific operation it represents is determined by an additional parameter α. Specifically, if the spider has k inputs and l output, then the operations are defined as follows:

$$Z(\alpha) = \left|0^k\right\rangle + e^{i\alpha}\left|1^l\right\rangle \text{ and } X(\alpha) = \left|+^k\right\rangle + e^{i\alpha}\left|-^l\right\rangle$$

Wires connect spiders and represent qubits. A wire can be seen as a qubit to which the operations (spiders) are applied. Quantum gates are translated into a corresponding combination of spiders and wires. An example of a circuit represented using ZX calculus can be seen in Figure 5.2.

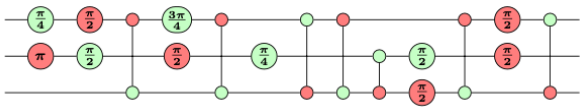

Figure 5.2: Example simulation of a small quantum circuit using abstractions. Taken from [41].

The diagrams allow for equivalent transformations to simplify or modify the quantum circuit without altering the underlying quantum operations. This is particularly useful for circuit optimization and proving equivalence between quantum circuits. In [42] it an algorithm is provided that transforms diagrams into their *reduced gadget form*. For two Clifford circuits represented by ZX-diagrams C and C' it holds that they are equivalent if the reduced gadget form of the diagram $C^\dagger C'$ equals the identity diagram. This result does not extend to arbitrary quantum circuits, where it is possible for two equivalent circuits U and U' that the reduced gadget form of $U^\dagger U'$ does not equal the identity diagram. This implies that for arbitrary quantum circuit a reduction to reduced gadget form can prove equivalence, but can not prove non-equivalence.

In [22] various benchmarks have been run from which conclusions could be drawn about the performance of the two approaches. Reducing ZX-diagrams will always finish in polynomial time and have a guaranteed maximum memory usage. The downsides of this is that it does not scale very well for large circuits. The performance of decision diagrams depends highly on the specific circuits. For some circuits, even if they are very large, the circuit structure allows for a compact representation with decision diagrams, while for other circuits this is not the case. Because of this simulation and equivalence verification with decision diagrams can be much faster, but this is not guaranteed. This led to QCEC using using a hybrid strategy of both methods. First, a few simulations with decision diagrams are performed with random inputs. If the two circuits yield different states for any of these simulations, we know the circuits are not equivalent. Second, the alternating equivalence verification method and the ZX-calculus based approach are run along side each other. If the former approach finds a quick solution, which can be the case if the decision diagram stays compact, then we do not have to wait for the ZX reduction, but but if the decision diagrams grows exponentially, then the ZX-calculus provides a reliable alternative with consistent runtime.

5.5 Quipu

Quipu [32] is a quantum circuit simulator based on the stabilizer formalism that uses a novel concept called *stabilizer frames* to more compactly represent stabilizer generators. Although this thesis focuses on equivalence verification, methods for simulating circuits using the stabilizer formalism remain relevant, as the only difference between simulating a circuit and verifying equivalence is the measurements. For this reason, we will discuss the method proposed in [32]. The key difference between the approach used in this thesis and Quipu lies in how the stabilizer generators are stored. In our method, all stabilizer generators are stored explicitly, whereas in Quipu, they may be stored implicitly and can be derived from the stabilizer frame. This compact representation can be beneficial for circuits where the stabilizer generators frequently increase in size due to the application of non-Clifford gates, but it also introduces additional overhead when updating the stabilizer generators.

Before we can define stabilizer frames we first need to provide some definitions. The *cofactor* of the j^{th} qubit of a quantum state $|\psi\rangle$ is a projection of the j^{th} qubit onto the computational basis states $|0\rangle$ or $|1\rangle$ indicated by $|\psi^{j=0}\rangle$ or $|\psi^{j=1}\rangle$ respectively. In other words $|\psi^{j=0}\rangle$ and $|\psi^{j=1}\rangle$ are exactly the states after a measurement in the computational basis results in $|0\rangle$ and $|1\rangle$, respectively.

A stabilizer matrix $\mathcal{M}$ associated to a state stabilizer state is a matrix containing the stabilizer generators of the state. An important observation the authors make, is that the phases of the generators are imperative to identifying the state. Therefore, if we store a single stabilizer matrix $\mathcal{M}$ and two phase vectors σ_i and σ_j we can actually represent two stabilizer states: The state identified by the generators $\sigma_i\mathcal{M}$

and $\sigma_j\mathcal{M}$. More generally, using a single stabilizer matrix it is possible to represent 2^n different stabilizer states by using associated phase vectors.

A *stabilizer frame* represents a quantum state $|\psi\rangle$ as a sum of stabilizer states. A single stabilizer state is identified in the frame by the stabilizer matrix $\mathcal{M}$, a phase vector σ_j and a global phase a_j. Consider Figure 5.3. Here we can see that the state $|\psi\rangle$ consists of a sum of two stabilizer states $a_1(|00\rangle + |01\rangle)$ and $a_2(|10\rangle + |11\rangle)$ identified by the stabilizer generators $\sigma_1\mathcal{M}$ and $\sigma_2\mathcal{M}$ and two global phases a_1 and a_2 in the sum. Note that usually we do not need to store the global phase of a stabilizer state, however as we are representing a state as a sum of stabilizer states the global phases of the individual stabilizer states becomes relative and must be stored.

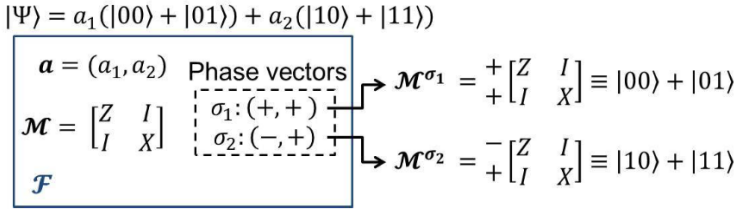

Figure 5.3: A two qubit state $|\psi\rangle$ represented using a stabilizer frame. Taken from [32].

To manipulate a state represented by a frame several *frame operations* are defined. First, `ROTATE(`$\mathcal{F}$`,` C`)` simulates the application of a Clifford gate C on a quantum state represented by the stabilizer frame $\mathcal{F}$ by updating the stabilizer generators stored in its stabilizer matrix $\mathcal{M}$ and the associated phase vectors through the usual update rules (See Section 2.2.4). The global phases also have to be updated accordingly. Second, `COFACTOR(`$\mathcal{F}$`,` c`)` explicitly represents a state as a sum of the its cofactors of the c^{th} qubit. The necessity for this operation is twofold: it facilitates measurements and allows for the simulation of non-Clifford gates. For the latter, consider Figure 5.4. Here we can see that the the state $|\Psi\rangle$ which is represented by a single stabilizer matrix and phase vector can be represented by the sum of the cofactors of its first two qubits (represented within the dotted box). To apply a non-Clifford gate we only need to update each cofactor individually. In the example figure, to simulate a Toffoli gate we only update the cofactor $|\Psi^{c_1c_2=11}\rangle$ to represent $|111\rangle$ (indicated by the arrow), as the states represented by the other cofactors remain unchanged. To simulate measurements the outcome probability is calculated as the sum of the normalized outcome probabilities of each state.

Although a single stabilizer frame is sufficient, it can be beneficial to store a single frame as multiple frames. The operation `COALESCE(`$\mathcal{F}$`)` provides this functionality. It searches for multiple phase vectors associated to the same stabilizer matrix, which can be *coalesced* into a single phase vector associated to another stabilizer matrix. An example is provided in Figure 5.5. The dotted rectangles indicate the phase vectors that are merged into a new frame pointed to by the arrows. The gates above the arrows indicate how the generators in the stabilizer matrix transform.

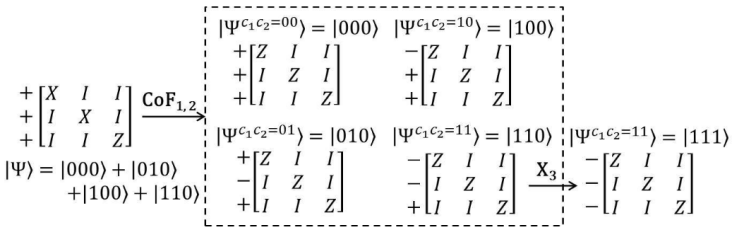

Figure 5.4: Cofactoring the state $|\psi\rangle$ and applying a Toffoli gate. Taken from [32].

When we have two frames with equivalent stabilizer frames we can merge them into a single frame again.

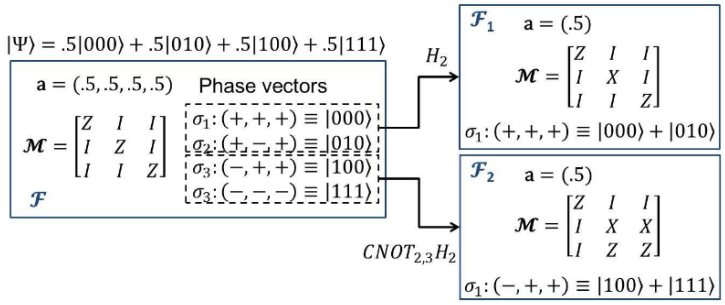

Figure 5.5: Derivation of multiple frames from a single stabilizer frame. Taken from [32].

To better illustrate how these operations can be used to simulate a quantum circuit we will explain the example depicted in Figure 5.6[1].

In the top left we see the example circuit and intermediate states. In the bottom left we can see the frame $\mathcal{F}$ which represents the state $|\Psi\rangle$. We first explain how the Toffoli gate is applied to the state $|\Psi\rangle$, which can also depicted in Figure 5.4, but here we will provide a more detailed explanation using Figure 5.6. In order to apply a Tollofi gate to obtain the state $|\Psi'\rangle$ we first determine the cofactors of $\mathcal{F}$ for each qubit and apply the Toffoli gate to them. If we cofactor $\mathcal{F}$ for each qubit — i.e., perform the operation `COFACTOR(`$\mathcal{F}$`,` c`)` for each $c \in \{0, 1, 2\}$ — we are explicitly representing each state $|j\rangle$, with $j \in \{0, 1\}^3$, with a non-zero coefficient in the stabilizer frame as a combination of the stabilizer matrix and phase vectors. This means that we obtain the stabilizer matrix $\mathcal{M} = Z^{\otimes 3}$ with the phase vectors $\sigma_1 = (+, +, +)$, $\sigma_2 = (+, -, +)$, $\sigma_3 = (-, +, +)$ and $\sigma_4 = (-, -, +)$ representing the states $|000\rangle$, $|010\rangle$, $|100\rangle$, $|110\rangle$. All states have an associated global phase variable $a_i = 0.5$ associated to them to account for the amplitudes of the states in $|\Psi\rangle$. Now that we have cofactored $\mathcal{F}$, we can apply the Toffoli gate to each individual cofactor. Since the Toffoli gate only changes the summand $.5|110\rangle$ in

[1]Note that in the upper left box the final summand of the states $|\Psi'\rangle$ and $|\Psi''\rangle$ should be $.5|111\rangle$ instead of $.5|110\rangle$

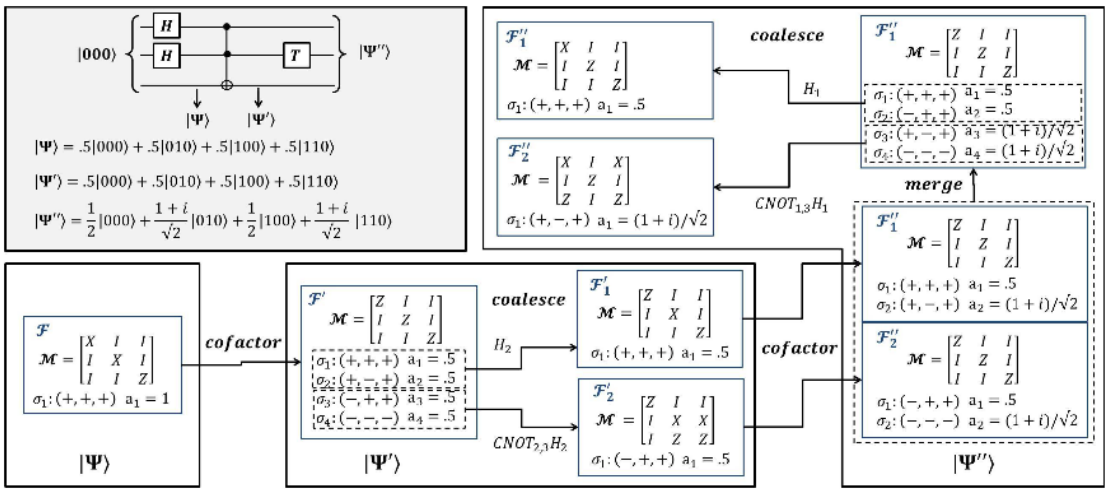

Figure 5.6: Example simulation of a small quantum circuit using stabilizer frames. Taken from [32].

$|\psi\rangle$ to $.5|111\rangle$, we only need to update σ_4 in our cofactored frame from $(-,-,+)$ to $(-,-,-)$. The resulting frame $\mathcal{F}'$ can be seen in the bottom-middle rectangle of the figure.

We can coalesce $\mathcal{F}'$ into $\mathcal{F}'_1$ and $\mathcal{F}'_2$. This is depicted by the arrows in the bottom-middle rectangle. Both arrows indicate that two phase vectors, which are surrounded by a dotted box, are merged into a single phase vector associated to a different stabilizer matrix. $\mathcal{F}'_1$ and $\mathcal{F}'_2$ now represent the summands $0.5(|000\rangle+|010\rangle)$ and $0.5(|100\rangle + |111\rangle)$ of $|\psi'\rangle$, respectively.

To apply the T gate to the state $|\Psi\rangle'$ we first cofactor both $\mathcal{F}'_1$ and $\mathcal{F}'_2$ into $\mathcal{F}''_1$ and $\mathcal{F}''_2$ and update each cofactor individually. In $\mathcal{F}''_1$ the phase vectors σ_1 and σ_2 represent the summand $0.5|000\rangle$ and $0.5|010\rangle$ of the state $|\psi'\rangle$. The T gate changes the amplitude of $|010\rangle$ from 0.5 to $\frac{1+i}{\sqrt{2}}$, so we associate the coefficients $a_1 = 0.5$ and $a_2 = \frac{1+i}{\sqrt{2}}$ to σ_1 and σ_2, respectively, to indicate this change. In $\mathcal{F}''_2$ we perform an analogously operation, except here the phase vectors σ_1 and σ_2 represent the summands $0.5|100\rangle$ and $0.5|111\rangle$ of the state $|\psi'\rangle$. The T gate changes the amplitude of $|111\rangle$ from 0.5 to $\frac{1+i}{\sqrt{2}}$, so we associate the coefficients $a_1 = 0.5$ and $a_2 = \frac{1+i}{\sqrt{2}}$ to σ_1 and σ_2, respectively, to indicate this change. These stabilizer frames already represent the state $|\psi''\rangle$, but it is possible to use a more compact representation. Since $\mathcal{F}''_1$ and $\mathcal{F}''_2$ both contain the same stabilizer matrix, we can merge them into $\mathcal{F}''_1$. Afterwards, we can coalesce $\mathcal{F}''_1$ back into two stabilizer frames $\mathcal{F}''_1$ and $\mathcal{F}''_2$ to obtain the final representation of the state $|\psi''\rangle$.

5.6 Abstraqt

Abstraqt is a stabilizer based quantum circuit simulator. In order to limit the exponential growth of stabilizers due to the simulation of non-Clifford gates abstraqt merges multiple Pauli strings into a single "abstract element".

An *abstraction* consist of an *abstract set* $(\boldsymbol{\mathcal{X}}, \leq)$, which is ordered set where each element represents elements from a *concrete set* $(2^{\mathcal{X}}, \leq)$, which is also an ordered set. Concrete elements can be obtained from abstract elements through the *concretization function* $\gamma : \boldsymbol{\mathcal{X}} \mapsto 2^{\mathcal{X}}$. An *abstract transformer* $f^{\#} : \boldsymbol{\mathcal{X}} \mapsto \boldsymbol{\mathcal{X}}$ defines how an operation transforms abstract elements.

In abstraqt real numbers are abstracted through intervals. In this case $\mathcal{X} = \mathbb{R}$ and $\boldsymbol{\mathcal{X}}$ is the set of of intervals on $\mathbb{R}$. An element $[l, u] \in \boldsymbol{\mathcal{X}}$ represents the elements $\gamma([l, u]) = \{x \mid x \in [l, u]\} \in \mathcal{X}$. In turn complex numbers can be abstracted by using polar notation and using intervals to abstract the real parts of the numbers. In this case the abstract complex number $e^{[l_1,u_1]+[l_2,u_2]i}$ represents the complex numbers $\{e^{r_1+r_2i} \mid r_1 \in [l_1, u_1], r_2 \in [l_2, u_2]\}$.

Pauli strings can be abstracted as $\boldsymbol{\mathcal{P}} = i^v \cdot \boldsymbol{\mathcal{P}^{(0)}} \otimes \boldsymbol{\mathcal{P}^{(1)}} \otimes \cdots \otimes \boldsymbol{\mathcal{P}^{(n-1)}}$, where $v \in \boldsymbol{\mathcal{Z}_4}$ and $\boldsymbol{\mathcal{P}^{(k)}} \subseteq \{X, Y, Z, I\}$. Here $\boldsymbol{\mathcal{Z}_4}$ denotes the abstract set of the concrete set $\mathbb{Z}_4$. The elements of $\boldsymbol{\mathcal{Z}_4}$ are sets consisting of elements of $\mathbb{Z}_4$ and can therefore represent multiple complex numbers of the form i^x with $x \in \mathbb{Z}_4$. The abstract element $\boldsymbol{\mathcal{P}}$ can therefore represent the following set of Pauli strings:

$$\gamma(\boldsymbol{\mathcal{P}}) = \{i^v \cdot \otimes_{i=0}^{n-1} \mathcal{P}^{(i)} \mid v \in \boldsymbol{v}, \mathcal{P}^{(i)} \in \boldsymbol{\mathcal{P}^{(i)}}\}$$

In turn, as we can abstract Pauli strings, we can also abstract density matrices. This is because the density matrix of any pure state can be expressed as a sum of Pauli strings.

To facilitate simulation using these abstract representations various abstract transformers have been defined, such as gate conjugation and compression. In the former the Pauli strings of the abstract element $\boldsymbol{\mathcal{P}}$ are conjugated and in the latter intervals and abstract Pauli strings are merged by taking their "union".

Since multiple elements are merged into a single abstract representation, the measurements obtained will only be approximations. However, this is not a bug, but a feature: the tool trades precision for efficiency. This makes it particularly useful in scenarios where a full simulation is intractable, as Abstraqt may still be able to provide valuable insights into specific circuit properties.

Chapter 6

Discussions

In this Chapter we discuss an alternative way too look at Method 1 and discuss the theory of simulating measurements using the generalized stabilizer formalism, which can be used in future research. We discuss the former in Section 6.1 and the latter in Section 6.2 and Section 6.3.

6.1 Equivalence Verification in the Pauli Basis

While Method 1 uses the (generalized) stabilizer formalism to perform equivalence verification with Theorem 1, it worth noting that verifying equivalence using Theorem 1 does not inherently require the (generalized) stabilizer formalism. In [21] an alternative perspective is presented, showing that the density matrix of any pure quantum state can be expressed as a sum of Pauli strings. This implies that if two circuits coincide on conjugation of the Pauli basis, they must be equivalent. While the method to verify equivalence in both cases remains exactly the same, it is an elegant way to explain the method, as it avoids the complexity of the stabilizer formalism.

6.2 Simulating Measurements with the Generalized Stabilizer Formalism

In Section 2.2 we showed that the density matrix of a state obtained by applying a Clifford circuit to the all-zero state can be represented as a sum of the conjugated stabilizers of the initial all-zero state, which is expressed by Equation (2.1). This result remains valid when an arbitrary quantum circuit is applied. However, in this case, the stabilizers we obtain may be linear combinations of Pauli strings. Recall that the probability of measuring a $|0\rangle$ for a quantum state with density matrix ρ and stabilizer generators G, obtained by conjugating the Pauli stabilizer generators of the all-zero state, is computed by:

$$p_0 = \frac{1}{2}\operatorname{Tr}(\rho) + \frac{1}{2^{n+1}} \sum_{g \in \langle G \rangle} \operatorname{Tr}\left(g \cdot Z_i\right)$$

Per definition $\operatorname{Tr}(\rho) = 1$ and each $g \in \langle G \rangle$ might be a sum of Pauli strings, i.e., $g = x_1 p_1 + \cdots + x_j p_j$, where x_k and p_k denote the coefficient and the Pauli string

of the k^{th} summand of g. Also, recall that for any Pauli string p it holds that $\text{Tr}(pZ_i) = 2^n$ if $p = Z_i$ and $\text{Tr}(pZ_i) = 0$ otherwise. Using these facts we can further rewrite the equation above. To do this concisely, we will use *Iverson Notation*: A bracket with a boolean statement, e.g. $[x = y]$, has a value of 1 if the condition is true and 0 if it is false [43].

$$\begin{aligned}
p_0 &= \frac{1}{2} + \frac{1}{2^{n+1}} \sum_{g \in \langle G \rangle} \sum_k \text{Tr}\left(x_k p_k Z_i\right) \\
&= \frac{1}{2} + \frac{1}{2^{n+1}} \sum_{g \in \langle G \rangle} \sum_k x_k \,\text{Tr}(p_k Z_i) \\
&= \frac{1}{2} + \frac{1}{2^{n+1}} \sum_{g \in \langle G \rangle} \sum_k x_k \,\text{Tr}(I) \cdot [p_k = Z_i] \\
&= \frac{1}{2} + \frac{1}{2^{n+1}} \sum_{g \in \langle G \rangle} \sum_k x_k 2^n \cdot [p_k = Z_i] \\
&= \frac{1}{2} + \frac{1}{2} \sum_{g \in \langle G \rangle} \sum_k x_k \cdot [p_k = Z_i]
\end{aligned}$$

As we can see, in order to determine the probability of measuring $|0\rangle$ we have to sum the coefficients of each Pauli string summand of each stabilizer that is equal to Z_i. If we perform a simulation by only storing the stabilizer generators of a state, we can not easily sum the coefficients of stabilizer summands. If we naively want to construct the entire generalized stabilizer group, we would have to compute the product of any subset of the stabilizer generators. In the worst case, a stabilizer generator consist of 2^t Pauli strings summands, where t is the number of T gates in the circuit. This means that computing the entire generalized stabilizer group will take $\mathcal{O}(2^{nt})$ time, where n is the number of qubits in the circuit. In practice, this is infeasible for any circuit with more than a few qubits.

In addition to this, each measurement potentially increases the size of the generalized stabilizer group by a factor of four. Assume the simulated measurement outcome is 0, then the state is updated by the following formula:

$$\begin{aligned}
\rho' &= \frac{M_0 \rho M_0}{p_0} \\
&= \frac{1}{p_0} \left(\frac{I + Z}{2} \cdot \rho \cdot \frac{I + Z}{2} \right) \\
&= \frac{1}{4p_0} (\rho + Z\rho + \rho Z + Z\rho Z)
\end{aligned}$$

The state update is analogous when a measurement outcome of 1 is simulated, but then M_1 and $p_1 = 1 - p_0$ are used instead of M_0 and p_0, respectively.

6.3 Simulating Measurements with the Generalized Stabilizer Formalism: Alternative Approach

The stabilizer circuit simulator "STIM" uses an alternative approach to simulate measurements, which we discuss in Section 5.2. In this section, we will discuss how this approach can be applied to arbitrary quantum circuits.

Let U be any arbitrary quantum circuit with n qubits, M an arbitrary observable and denote ρ to be the density matrix of the all-zero state. If we start in the all-zero state and apply the quantum circuit U, then the probability of measuring the observable M is given by the following formula:

$$\operatorname{Tr}\left(U\rho U^\dagger M_a\right) = \operatorname{Tr}\left(\rho U^\dagger M_a U\right)$$

As we can see the probability of measuring the observable M after applying the circuit U is the same as the probability of measuring the observable $U^\dagger M_a U$ for the all-zero state. This can be quite convenient if we want to know the probability of measuring $|0\rangle$ for the a^{th} qubit after applying the circuit U. In this case the observable would be $\frac{1}{2}(I + Z_a)$, which give us the following formula:

$$\begin{aligned}
\operatorname{Tr}\left(\rho U^\dagger \frac{I + Z_a}{2} U\right) &= \frac{1}{2}\operatorname{Tr}\left(\rho U^\dagger I U\right) + \frac{1}{2}\operatorname{Tr}\left(\rho U^\dagger Z_a U\right) \\
&= \frac{1}{2}\operatorname{Tr}\left(\rho\right) + \frac{1}{2}\operatorname{Tr}\left(\rho U^\dagger Z_a U\right) \\
&= \frac{1}{2} + \frac{1}{2}\operatorname{Tr}\left(\rho U^\dagger Z_a U\right)
\end{aligned}$$

There are two observations which will allow us to rewrite this formula further. First, we know that ρ equals the sum of the stabilizers of the all-zero state (See Section 2.2.3). These are exactly all the Pauli strings that are a tensor product of only I and Z matrices. We will denote this set of stabilizers as $\text{Stab}(|0^n\rangle)$. Second, as U is any arbitrary quantum circuit, we know that $U^\dagger Z_a U$ might be a sum of Pauli strings. Using these facts we obtain the following:

$$
\begin{aligned}
\frac{1}{2} + \frac{1}{2}\operatorname{Tr}\left(\rho U^\dagger Z_a U\right) &= \frac{1}{2} + \frac{1}{2}\operatorname{Tr}\left(\rho \Sigma_j \alpha_j P_j\right) \\
&= \frac{1}{2} + \frac{1}{2}\Sigma_j \alpha_j \operatorname{Tr}\left(\rho P_j\right) \\
&= \frac{1}{2} + \frac{1}{2}\Sigma_j \alpha_j \operatorname{Tr}\left(\Sigma_{S_k \in \mathrm{Stab}(|0^n\rangle)} S_k P_j\right) \\
&= \frac{1}{2} + \frac{1}{2}\Sigma_j \alpha_j \Sigma_{S_k \in \mathrm{Stab}(|0^n\rangle)} \operatorname{Tr}(S_k P_j) \\
&= \frac{1}{2} + \frac{1}{2}\Sigma_j \alpha_j [\exists S_k \in \mathrm{Stab}(|0^n\rangle) \text{ s.t. } S_k = P_j]
\end{aligned}
$$

Notice that once we have determined $U^\dagger Z_a U$, we can easily determine the probability of measuring $|0\rangle$ for the a^{th} qubit. This is because we only have to sum the coefficients of all the Pauli string summands of $U^\dagger Z_a U$ that are a tensor product of exclusively I and Z matrices, because we know that these Pauli strings are elements of $\mathrm{Stab}(|0^n\rangle)$. Additionally, we only need to conjugate the Pauli string Z_a as I commutes with all Pauli strings.

If we denote the number of T gates in a circuit with t, then this method reduces the complexity of simulating a single measurement to $\mathcal{O}(2^t)$, which is a significant improvement over the $\mathcal{O}(2^{nt})$ complexity of the previous approach. This means that, if it is tractable for a given circuit to conjugate its gates with the Pauli string Z_a, then it is also possible to obtain the measurement probabilities of the a^{th} qubit.

The challenge of this approach is not how to determine a single measurement, it is how to determine all measurements. Consider, that we simulated a measurement of the a^{th} qubit and we now want to determine the measurement probabilities of the b^{th} qubit. We can not simply use the observable $\frac{1}{2}(I + Z_b)$, as this would not take our simulated result of qubit a into account. STIM solves this problem by changing the circuit in such a way that qubit a is forced to be in the state $|0\rangle$ or $|1\rangle$, depending on the simulated measurement, and then simulating the measurement of qubit b. For stabilizer circuits this can easily be done using a Hadamard and X gate, but for circuits that contain rotational gates, this is not possible. However, it is possible to use a different observable. Consider that we simulated a measurement of $|0\rangle$ for the a^{th} qubit, then we could determine the probability of measuring $|0\rangle$ for both the a^{th} and b^{th} qubit using the observable $\frac{1}{2}(I + Z_a) \otimes \frac{1}{2}(I + Z_b)$. Similarly, we can determine the probability of measuring $|0\rangle$ for the a^{th} qubit and $|1\rangle$ for the b^{th} qubit using the observable $\frac{1}{2}(I + Z_a) \otimes \frac{1}{2}(I - Z_b)$. Notice that from these probabilities we can deduce the measurement probabilities for the b^{th} qubit, given a measurement of $|0\rangle$ for the a^{th} qubit. This method can be generalized to determine the measurement probabilities of any qubit, while taking any number of prior simulated measurements into account. The problem with this approach is that the number of Pauli strings in the observable grows exponentially

in the number of simulated measurements: for a given measurement we have to factor in the result of all previous measurements in the observable, which requires us to tensor $\frac{1}{2}(I \pm Z_j)$ for each qubit j that has been measured. As a result, the complexity of simulating all measurements of a circuit using this approach does not improve over the previous approach and remains $\mathcal{O}(2^{nt})$.

Chapter 7

Conclusions and Future Research

In this thesis we introduced four novel data structures which can be used to verify the equivalence of arbitrary quantum circuits using the generalized stabilizer formalism. We implemented and empirically benchmarked these data structures, comparing their performance. These results contributed to the development of our equivalence verification tool, CTQC, which dynamically selects the most suitable data structure based on whether the circuits contain exclusively Clifford gates. We evaluated CTQC against two existing tools: QCEC [14] and ECMC[21]. Our experiments demonstrated that for certain quantum algorithms — such as the Deutsch-Josza, GHZ state, and Graph state algorithms — our method outperforms these existing tools. Specifically, CTQC verifies equivalence faster, while using significantly less memory. For circuits with more than 1000 qubits from the aforementioned algorithms, the maximum resident set size of our tool was up to 540 times smaller than ECMC and 800 times smaller than QCEC. Additionally, when verifying equivalence between non-equivalent circuits we showed that CTQC's classifications are very reliable, as there where no incorrect classifications in the performed benchmarks. This is in contrast to both QCEC and ECMC, which incorrectly classified some non-equivalent circuits as equivalent or failed to provide results.

While the benchmarks demonstrate that CTQC only outperforms QCEC on Clifford circuits — which is unsurprising given the efficiency of the stabilizer formalism for these circuits — we believe our tool can also be more effective for certain non-Clifford circuits. Future research could focus on developing a metric to determine an upper bound on the number of R_z gates for each qubit that may lead to a duplication in the number of summands of a generator. For instance, if multiple consecutive R_z gates target the same qubit, the number of summands in a generator will at most double. Similarly, if the i^{th} qubit is not (potentially) entangled with the j^{th} qubit, any R_z gate targeting the j^{th} qubit would not affect the number of summands in the generators Z_i and X_i during the verification process. Experiments could be conducted to identify thresholds under which our tool — or, more broadly, equivalence verification tools based on generalized stabilizers — remain effective. This metric could enable the development of a hybrid approach that combines multiple methods for equivalence verification.

Future work could also focus on refining floating point error management to further improve the accuracy of equivalence verification tools. Our approach, which uses a dynamic error margin and takes additional precautions with small values

produced by rotational gates, has demonstrated robustness across the benchmarks. However, for circuits with rotational gates our tool can only verify equivalence for small circuits (in terms of qubit count) and as the number of gates increases, the cumulative error margin is overestimated proportionally, potentially causing errors. Future research could examine if the approach can be effectively used in other equivalence verification techniques, such as decision diagram-based simulations used in QCEC, and evaluate its scalability for larger circuits.

Another possibility for future research is to explore more efficient techniques for simulating measurements using our method. While we have thoroughly covered the theoretical aspects in Chapter 6, we also highlighted the challenges associated with simulating measurements in practice. It would be valuable to explore more efficient methods for simulating measurements, as this would allow us to not only verify the equivalence of quantum circuits but also to simulate them.

As the capabilities of quantum computers increase, the number of qubits and the complexity of circuits will also grow. Consequently, verifying equivalence will place increasing demands on both time and memory resources, highlighting the need for efficient equivalence verification tools. We empirically show that our tool, for specific circuit types, will offer better scalability compared to current solutions, making it a valuable contribution to the suite of quantum circuit equivalence verification tools.

Bibliography

[1] A. Montanaro, "Quantum algorithms: an overview," *npj Quantum Information*, vol. 2, p. 15023, Jan 2016.

[2] A. M. Dalzell, S. McArdle, M. Berta, P. Bienias, C.-F. Chen, A. Gilyén, C. T. Hann, M. J. Kastoryano, E. T. Khabiboulline, A. Kubica, G. Salton, S. Wang, and F. G. S. L. Brandão, "Quantum algorithms: A survey of applications and end-to-end complexities," 2023.

[3] S. Zhang and L. Li, "A brief introduction to quantum algorithms," *CCF Transactions on High Performance Computing*, vol. 4, p. 53–62, Feb. 2022.

[4] A. D. Corcoles, A. Kandala, A. Javadi-Abhari, D. T. McClure, A. W. Cross, K. Temme, P. D. Nation, M. Steffen, and J. M. Gambetta, "Challenges and opportunities of near-term quantum computing systems," *Proceedings of the IEEE*, vol. 108, p. 1338–1352, Aug. 2020.

[5] G. Kalai, "The argument against quantum computers," 2019.

[6] J. W. Z. Lau, K. H. Lim, H. Shrotriya, and L. C. Kwek, "Nisq computing: where are we and where do we go?," *AAPPS Bulletin*, vol. 32, p. 27, Sep 2022.

[7] S. Brandhofer, S. Devitt, T. Wellens, and I. Polian, "Special session: Noisy intermediate-scale quantum (nisq) computers—how they work, how they fail, how to test them?," in *2021 IEEE 39th VLSI Test Symposium (VTS)*, pp. 1–10, 2021.

[8] J. Preskill, "Quantum computing in the nisq era and beyond," *Quantum*, vol. 2, p. 79, Aug. 2018.

[9] W. Finigan, M. Cubeddu, T. Lively, J. Flick, and P. Narang, "Qubit allocation for noisy intermediate-scale quantum computers," 2018.

[10] D. A. Sofge, "A survey of quantum programming languages: History, methods, and tools," 2008.

[11] P. Selinger, "A brief survey of quantum programming languages," in *Functional and Logic Programming* (Y. Kameyama and P. J. Stuckey, eds.), (Berlin, Heidelberg), pp. 1–6, Springer Berlin Heidelberg, 2004.

[12] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," 2017.

[13] M. Amy and V. Gheorghiu, "staq — a full-stack quantum processing toolkit," *Quantum Science and Technology*, vol. 5, p. 034016, June 2020.

[14] T. Peham, L. Burgholzer, and R. Wille, "Equivalence checking of quantum circuits with the zx-calculus," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 12, p. 662–675, Sept. 2022.

[15] D. Janzing, P. Wocjan, and T. Beth, ""non-identity-check" is qma-complete," *International Journal of Quantum Information (IJQI)*, vol. 3, 09 2005.

[16] S. Aaronson and D. Gottesman, "Improved simulation of stabilizer circuits," *Physical Review A*, vol. 70, Nov. 2004.

[17] D. Gottesman, "Stabilizer codes and quantum error correction," 1997.

[18] C. Gidney, "Stim: a fast stabilizer circuit simulator," *Quantum*, vol. 5, p. 497, July 2021.

[19] J. Mei, M. Bonsangue, and A. Laarman, "Simulating quantum circuits by model counting," 2024.

[20] D. Thanos, T. Coopmans, and A. Laarman, "Fast equivalence checking of quantum circuits of clifford gates," 2023.

[21] J. Mei, T. Coopmans, M. Bonsangue, and A. Laarman, "Equivalence checking of quantum circuits by model counting," 2024.

[22] T. Peham, L. Burgholzer, and R. Wille, "Equivalence checking of quantum circuits with the zx-calculus," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 12, p. 662–675, Sept. 2022.

[23] L. Berent, L. Burgholzer, and R. Wille, "Towards a sat encoding for quantum circuits: A journey from classical circuits to clifford circuits and beyond," Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[24] R. Wille, N. Przigoda, and R. Drechsler, "A compact and efficient sat encoding for quantum circuits," in *IEEE AFRICON 2013*, IEEE AFRICON Conference, Institute of Electrical and Electronics Engineers Inc., 2013. IEEE AFRICON 2013 ; Conference date: 09-09-2013 Through 12-09-2013.

[25] M. Amy, "Towards large-scale functional verification of universal quantum circuits," *Electronic Proceedings in Theoretical Computer Science*, vol. 287, p. 1–21, Jan. 2019.

[26] L. Burgholzer and R. Wille, "Improved dd-based equivalence checking of quantum circuits," in *ASP-DAC 2020 - 25th Asia and South Pacific Design Automation Conference, Proceedings*, Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC, pp. 127–132, Institute of Electrical and Electronics Engineers Inc., Jan. 2020. Publisher Copyright: © 2020 IEEE.; 25th Asia and South Pacific Design Automation Conference, ASP-DAC 2020 ; Conference date: 13-01-2020 Through 16-01-2020.

[27] G. F. Viamontes, I. L. Markov, and J. P. Hayes, "Checking equivalence of quantum circuits and states," 2007.

[28] L. Vinkhuijzen, T. Coopmans, D. Elkouss, V. Dunjko, and A. Laarman, "Limdd: A decision diagram for simulation of quantum computing including stabilizer states," *Quantum*, vol. 7, p. 1108, Sept. 2023.

[29] C.-Y. Wei, Y.-H. Tsai, C.-S. Jhang, and J.-H. R. Jiang, "Accurate bdd-based unitary operator manipulation for scalable and robust quantum circuit verification," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, (New York, NY, USA), p. 523–528, Association for Computing Machinery, 2022.

[30] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2011.

[31] S. Aaronson, "Lecture 28, tues may 2: Stabilizer formalism." https://www.scottaaronson.com/qclec/28.pdf. Accessed: 2023-10-11.

[32] H. J. García and I. L. Markov, "Simulation of quantum circuits via stabilizer frames," 2017.

[33] C. M. Dawson and M. A. Nielsen, "The solovay-kitaev algorithm," 2005.

[34] Y. Zhang, Y. Tang, Y. Zhou, and X. Ma, "Efficient entanglement generation and detection of generalized stabilizer states," *Phys. Rev. A*, vol. 103, p. 052426, May 2021.

[35] L. Allison, "ctqc." https://github.com/lucasallison/ctqc, 2024.

[36] A. Clements, *Computer Organization and Architecture: Themes and Variations*. United States: Cengage Learning, international ed. ed., 2014.

[37] M. Raasveldt and H. Mühleisen, "Duckdb: an embeddable analytical database," in *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, (New York, NY, USA), p. 1981–1984, Association for Computing Machinery, 2019.

[38] A. Laarman, J. van de Pol, and M. Weber, "Parallel recursive state compression for free," *CoRR*, vol. abs/1104.3119, 2011.

[39] N. Quetschlich, L. Burgholzer, and R. Wille, "MQT Bench: Benchmarking software and design automation tools for quantum computing," *Quantum*, 2023. MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/.

[40] T. M. Chair for Design Automation, "mqt-qcec." https://github.com/cda-tum/mqt-qcec, 2024.

[41] B. Bichsel, A. Paradis, M. Baader, and M. Vechev, "Abstraqt: Analysis of quantum circuits via abstract stabilizer simulation," *Quantum*, vol. 7, p. 1185, Nov. 2023.

[42] A. Kissinger and J. van de Wetering, "Reducing the number of non-clifford gates in quantum circuits," *Physical Review A*, vol. 102, Aug. 2020.

[43] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*. USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 1994.

Appendices

Appendix A

Benchmark Results

In this chapter we present the benchmark complete results for the MQT benchmark set. The first columns of the tables list the various algorithms that occur in the benchmark set. Each algorithm has multiple instances, each with a different number of qubits. This is indicated in the second column. The third and fourth columns show the total number of gates $\#G$ and the percentage of which are Rz ($\#Rz$) gates in the original circuit. In the following two columns, the same information is shown for the circuit types that are used to verify equivalence against. This is either the optimized version (opt) or the optimized version with an introduced error (See Section 4.2). The last columns show the time and RSS usage of the various tools. In each row the best time and memory usage is highlighted in green. The following symbols are used throughout the tables:

- '-': Indicates a timeout, or if preceeded by a timeout or exception it indicates that the verification process was not executed for this circuit. When benchmarking equivalent circuits a timeout of 10 minutes was used, while benchmarking non-equivalent circuits a timeout of 3 minutes was used.
- 'M': Indicates that the tool exceeded the 8GB RSS memory limit.
- 'E': The tool encountered an exception.
- 'N': No result could be determined by the tool. This is specific to QCEC, which sometimes cannot assess the equivalence of the circuits.
- 'W': The circuits were incorrectly classified as equivalent or non-equivalent.

Table A.1: Results for verifying equivalence between original and optimized circuits from the MQT benchmark set using the proposed generalized stabilizer datastructures

Algorithm	**#Qubits**	**transp**		**opt**		**Col-wise BV**		**Map**		**Tree**		**Row-wise BV**	
		$\#G$	$\%R_z$	$\#G$	$\%R_z$	t(sec)	mem(mbs)	t(sec)	mem(mbs)	t(sec)	mem(mbs)	t(sec)	mem(mbs)
Amplitude Estimation	2^3	296	67	296	67	1.5878	70.7	0.4801	22.9	10.056	31.3	0.9996	59.9
	2^4	924	65	924	65	M	M	M	M	-	-	M	M
Deutsch-Josza	2^3	66	67	33	34	0.0035	0.0	0.0009	0.0	0.0012	0.0	0.0037	0.0
	2^4	114	60	63	27	0.0044	0.0	0.0016	0.0	0.0016	0.0	0.0013	0.0
	2^5	242	62	131	29	0.0017	0.0	0.0016	0.0	0.0043	0.0	0.0026	0.0
	2^6	466	60	259	27	0.0033	0.0	0.0039	0.0	0.0152	5.3	0.0034	0.0
	2^7	890	58	509	25	0.0097	5.1	0.0115	5.2	0.0515	6.0	0.0117	5.1
	2^8	1762	57	1015	25	0.0808	5.7	0.0505	5.7	0.1933	6.8	0.0298	5.3
	2^9	3402	55	2001	24	0.1166	6.6	0.2845	7.0	0.5905	8.8	0.1016	5.2
	2^{10}	7058	57	4067	25	0.5258	9.9	1.8669	9.0	2.7325	13.2	0.455	6.8
	2^{11}	14114	57	8135	25	2.2259	22.0	14.4055	13.9	16.2668	22.0	1.731	9.3
	2^{12}	28362	57	16305	25	11.1899	62.8	123.141	16.3	107.1632	38.7	7.8248	13.3
	2^{13}	57058	57	32695	25	53.195	216.9	-	-	-	-	31.3367	23.5
GHZ State	2^3	8	0	8	0	0.0025	0.0	0.0008	0.0	0.0013	0.0	0.001	0.0
	2^4	16	0	16	0	0.0012	0.0	0.0009	0.0	0.0046	0.0	0.0027	0.0
	2^5	32	0	32	0	0.0019	0.0	0.0016	0.0	0.0063	0.0	0.0019	0.0
	2^6	64	0	64	0	0.0065	0.0	0.0036	0.0	0.0095	0.0	0.0021	0.0
	2^7	128	0	128	0	0.015	0.0	0.0079	0.0	0.0322	5.5	0.0074	0.0
	2^8	256	0	256	0	0.0487	5.0	0.0351	5.2	0.1302	6.4	0.0171	4.8
	2^9	512	0	512	0	0.133	6.1	0.2647	6.4	0.6244	8.3	0.0673	5.1
	2^{10}	1024	0	1024	0	0.3004	9.4	2.0184	8.4	3.2457	12.7	0.1479	6.2
	2^{11}	2048	0	2048	0	1.1636	20.6	15.7728	12.3	18.5608	25.0	0.6239	8.1
	2^{12}	4096	0	4096	0	5.8267	60.1	124.6941	20.7	125.6565	47.4	2.3891	11.2
	2^{13}	8192	0	8192	0	33.1645	212.9	-	-	-	-	10.649	16.3
	2^{14}	16384	0	16384	0	222.8422	802.8	-	-	-	-	42.5643	27.4
	2^{15}	32768	0	32768	0	-	-	-	-	-	-	176.4379	53.0
Graph State	2^3	32	0	16	0	0.0012	0.0	0.0011	0.0	0.0014	0.0	0.0011	0.0
	2^4	64	0	32	0	0.0017	0.0	0.0031	0.0	0.0043	0.0	0.0019	0.0
	2^5	128	0	64	0	0.002	0.0	0.0043	0.0	0.0036	0.0	0.0026	0.0
	2^6	256	0	128	0	0.0027	0.0	0.0062	0.0	0.0133	0.0	0.0053	0.0
	2^7	512	0	256	0	0.0215	0.0	0.0116	0.0	0.0338	5.9	0.0107	0.0
	2^8	1024	0	512	0	0.0787	5.0	0.0468	5.3	0.1394	6.5	0.0334	4.8
	2^9	2048	0	1024	0	0.179	6.1	0.3184	6.6	0.6543	8.5	0.1383	5.1
	2^{10}	4096	0	2048	0	0.4867	9.4	2.3934	8.6	3.3359	12.8	0.3727	6.4
	2^{11}	8192	0	4096	0	1.9565	20.7	16.777	12.8	18.1756	20.9	1.3467	8.1
	2^{12}	16384	0	8192	0	8.9611	61.5	127.7919	15.1	108.2165	37.2	5.4273	12.2
	2^{13}	32768	0	16384	0	43.0963	214.5	-	-	-	-	22.3909	18.6
	2^{14}	65536	0	32768	0	253.1204	807.3	-	-	-	-	90.0806	32.1
	2^{15}	131072	0	65536	0	-	-	-	-	-	-	364.7836	60.7
Portfolio Optimization using QAOA	2^3	508	56	508	56	41.8186	654.9	25.0507	216.3	425.7628	353.2	32.8958	640.6
	2^4	1592	48	1592	48	M	M	M	M	-	-	M	M
Portfolio Optimization using VQE	2^3	540	42	414	55	3.7859	135.3	2.3047	100.1	39.2353	106.6	2.7049	156.6
	2^4	1656	28	1026	44	M	M	M	M	-	-	M	M
QAOA	2^3	200	64	200	64	0.0068	0.0	0.0027	0.0	0.038	5.4	0.0025	0.0
	2^4	400	64	400	64	0.0071	5.1	0.0041	0.0	0.0365	5.1	0.0044	0.0
Quantum Fourier Transform	2^3	160	53	139	46	1.0216	76.0	0.3548	19.4	8.9193	22.4	0.7166	74.8
	2^4	640	57	535	48	M	M	M	M	-	-	M	M
Entangled Quantum Fourier Transform	2^3	168	50	147	43	1.9677	146.5	0.5833	36.4	13.7146	46.9	1.5016	213.1
	2^4	656	55	551	47	M	M	M	M	-	-	M	M
Quantum Neural Network	2^3	399	59	323	44	27.0075	642.2	15.0561	238.5	322.8196	337.2	20.7199	647.0
	2^4	1183	51	1002	40	M	M	M	M	-	-	M	M
Exact Quantum Phase Estimation	2^3	167	51	167	51	0.7183	74.8	0.2531	13.4	5.6078	18.0	0.5871	60.3
	2^4	655	55	655	55	M	M	M	M	-	-	M	M
Inexact Quantum Phase Estimation	2^3	167	51	167	51	1.7367	107.4	0.4857	20.6	12.5797	29.7	1.2877	116.9
	2^4	655	55	655	55	M	M	M	M	-	-	M	M
Real Amplitude Random Circuit	2^3	372	61	241	39	2.4271	108.2	1.5944	61.2	31.0792	70.2	1.7019	75.7
	2^4	936	48	677	28	M	M	M	M	-	-	M	M
SU Random Circuit	2^3	372	61	241	39	15.6331	647.8	9.5276	233.7	289.0859	280.5	11.9401	573.2
	2^4	936	48	680	29	M	M	M	M	-	-	M	M
Two-Local Random Circuit	2^3	372	61	372	61	2.5611	61.7	1.6087	62.4	29.6189	67.7	1.7495	52.0
	2^4	936	48	936	48	M	M	M	M	-	-	M	M
Variational Quantum Eigensolver	2^3	230	74	230	74	0.7186	97.7	0.313	50.1	5.05	68.5	0.4716	89.7
	2^4	462	73	462	73	M	M	M	M	-	-	M	M
W State	2^3	158	63	141	51	0.0051	0.0	0.002	0.0	0.0163	0.0	0.0051	0.0
	2^4	334	63	301	51	0.0157	0.0	0.0031	0.0	0.0444	5.1	0.0118	0.0
	2^5	686	64	621	51	M	M	M	M	-	-	M	M

Table A.2: Results for verifying equivalence between original and optimized circuits from the MQT benchmark set.

Algorithm	#Qubits	transp		opt		CTQC		QCEC		ECMC	
		$\#G$	$\%R_z$	$\#G$	$\%R_z$	t(sec)	mem(mbs)	t(sec)	mem(mbs)	t(sec)	mem(mbs)
Amplitude Estimation	2^3	296	67	296	67	0.4801	22.9	0.0299	78.3	-	-
	2^4	924	65	924	65	M	M	0.0469	89.1	-	-
	2^5	3140	63	3140	63	-	-	0.0829	102.3	-	-
	2^6	11412	62	11412	62	-	-	E	E	-	-
Deutsch-Josza	2^3	66	67	33	34	0.0037	0.0	0.0245	78.0	0.0196	7.6
	2^4	114	60	63	27	0.0013	0.0	0.0307	78.2	0.0278	7.5
	2^5	242	62	131	29	0.0026	0.0	0.0438	84.9	0.0558	7.5
	2^6	466	60	259	27	0.0034	0.0	0.0701	141.0	0.1372	26.8
	2^7	890	58	509	25	0.0117	5.1	0.1214	740.8	0.4621	41.7
	2^8	1762	57	1015	25	0.0298	5.3	0.2299	1271.8	2.2228	65.9
	2^9	3402	55	2001	24	0.1016	5.2	0.4627	2600.1	12.9232	348.2
	2^{10}	7058	57	4067	25	0.455	6.8	1.0208	5682.2	11.5656	1298.0
	2^{11}	14114	57	8135	25	1.731	9.3	M	M	43.7122	5036.4
	2^{12}	28362	57	16305	25	7.8248	13.3	-	-	-	-
	2^{13}	57058	57	32695	25	31.3367	23.5	-	-	-	-
GHZ State	2^3	8	0	8	0	0.001	0.0	0.0259	200.5	0.0103	7.6
	2^4	16	0	16	0	0.0027	0.0	0.0304	84.3	0.0123	7.6
	2^5	32	0	32	0	0.0019	0.0	0.0433	84.5	0.0176	7.6
	2^6	64	0	64	0	0.0021	0.0	0.0697	262.0	0.0329	7.6
	2^7	128	0	128	0	0.0074	0.0	0.1246	754.4	0.0717	15.4
	2^8	256	0	256	0	0.0171	4.8	0.2345	1513.1	0.2016	38.3
	2^9	512	0	512	0	0.0673	5.1	0.4612	2928.5	0.671	62.4
	2^{10}	1024	0	1024	0	0.1479	6.2	0.9386	5812.3	2.4931	153.1
	2^{11}	2048	0	2048	0	0.6239	8.1	M	M	9.6961	519.5
	2^{12}	4096	0	4096	0	2.3891	11.2	-	-	36.9711	1907.9
	2^{13}	8192	0	8192	0	10.649	16.3	-	-	149.2334	7502.5
	2^{14}	16384	0	16384	0	42.5643	27.4	-	-	M	M
	2^{15}	32768	0	32768	0	176.4379	53.0	-	-	-	-
Graph State	2^3	32	0	16	0	0.0011	0.0	0.0246	84.2	0.0129	7.5
	2^4	64	0	32	0	0.0019	0.0	0.0306	84.6	0.0194	7.6
	2^5	128	0	64	0	0.0026	0.0	0.0444	84.5	0.0302	7.6
	2^6	256	0	128	0	0.0053	0.0	0.072	522.9	0.0684	7.5
	2^7	512	0	256	0	0.0107	0.0	0.1257	718.2	0.1548	31.8
	2^8	1024	0	512	0	0.0334	4.8	0.234	1218.6	1.9509	42.7
	2^9	2048	0	1024	0	0.1383	5.1	0.4897	2682.9	0.9034	57.3
	2^{10}	4096	0	2048	0	0.3727	6.4	1.0076	5522.0	169.1951	127.2
	2^{11}	8192	0	4096	0	1.3467	8.1	M	M	209.2035	242.2
	2^{12}	16384	0	8192	0	5.4273	12.2	-	-	-	-
	2^{13}	32768	0	16384	0	22.3909	18.6	-	-	-	-
	2^{14}	65536	0	32768	0	90.0806	32.1	-	-	-	-
	2^{15}	131072	0	65536	0	364.7836	60.7	-	-	-	-
Grover's Algorithm without Ancilla	2^3	10736	63	10736	63	-	-	0.2069	280.0	-	-
	2^4	2087278	67	2087278	67	-	-	-	-	-	-
Grover's Algorithm with V-Chain	2^3	7799	62	7757	63	M	M	0.1542	103.3	-	-
	2^4	1174493	66	1171391	66	-	-	-	-	-	-
Portfolio Optimization using QAOA	2^3	508	56	508	56	25.0507	216.3	0.0324	78.1	-	-
	2^4	1592	48	1592	48	M	M	0.0518	87.1	-	-
Portfolio Optimization using VQE	2^3	540	42	414	55	2.3047	100.1	0.0326	84.3	-	-
	2^4	1656	28	1026	44	M	M	0.0471	88.4	-	-
QAOA	2^3	200	64	200	64	0.0027	0.0	0.0283	78.2	-	-
	2^4	400	64	400	64	0.0041	0.0	0.036	86.0	-	-
Quantum Fourier Transform	2^3	160	53	139	46	0.3548	19.4	0.0259	77.6	-	-
	2^4	640	57	535	48	M	M	1.9776	417.0	-	-
	2^5	2560	59	1675	49	-	-	-	-	-	-
Entangled Quantum Fourier Transform	2^3	168	50	147	43	0.5833	36.4	0.0268	78.2	-	-
	2^4	656	55	551	47	M	M	1.6494	423.0	-	-
	2^5	2592	58	1707	48	-	-	-	-	-	-
Quantum Neural Network	2^3	399	59	323	44	15.0561	238.5	0.0339	87.6	M	M
	2^4	1183	51	1002	40	M	M	0.0513	152.6	-	-
	2^5	3903	44	3512	37	-	-	0.1052	85.4	-	-
	2^6	13951	39	13144	36	-	-	0.2954	364.7	-	-
	2^7	52479	37	50835	35	-	-	1.164	1193.7	-	-
	2^8	203263	35	199958	34	-	-	5.592	2248.3	-	-
Exact Quantum Phase Estimation	2^3	167	51	167	51	0.2531	13.4	0.0256	77.9	-	-
	2^4	655	55	655	55	M	M	0.0617	88.5	-	-
	2^5	2591	58	2591	58	-	-	-	-	-	-
Inexact Quantum Phase Estimation	2^3	167	51	167	51	0.4857	20.6	0.0314	78.1	-	-
	2^4	655	55	655	55	M	M	0.061	89.0	-	-
	2^5	2591	58	2591	58	-	-	-	-	-	-
Quantum Walk without Ancilla	2^3	10937	62	10913	62	-	-	0.2166	275.6	-	-
	2^4	219567	66	219543	66	-	-	27.5308	752.5	-	-
	2^5	2453631	67	2453607	67	-	-	-	-	-	-

Table A.3: Results for verifying equivalence between original and optimized circuits from the MQT benchmark set (continued).

Algorithm	**#Qubits**	**transp**		**opt**		**CTQC**		**QCEC**		**ECMC**	
		$\#G$	$\%R_z$	$\#G$	$\%R_z$	t(sec)	mem(mbs)	t(sec)	mem(mbs)	t(sec)	mem(mbs)
Quantum Walk with V-Chain	2^3	2337	42	2085	48	-	-	0.0547	87.7	-	-
	2^4	11457	44	10213	49	-	-	0.2228	393.2	-	-
	2^5	50433	44	44901	50	-	-	1.594	620.6	-	-
	2^6	211329	45	188005	50	-	-	15.4077	1337.9	-	-
	2^7	864897	45	769125	50	-	-	210.4146	3409.1	-	-
Real Amplitude Random Circuit	2^3	372	61	241	39	1.5944	61.2	0.028	78.2	-	-
	2^4	936	48	677	28	M	M	0.0442	89.6	-	-
	2^5	2640	34	2125	18	-	-	0.0831	133.4	-	-
	2^6	8352	22	7325	11	-	-	0.1995	505.8	-	-
	2^7	28992	13	26941	6	-	-	0.7351	1083.1	-	-
	2^8	107136	7	103037	3	-	-	3.2834	2063.6	-	-
SU Random Circuit	2^3	372	61	241	39	9.5276	233.7	0.028	85.6	-	-
	2^4	936	48	680	29	M	M	0.0434	160.4	-	-
	2^5	2640	34	2127	19	-	-	0.0879	370.0	-	-
	2^6	8352	22	7327	11	-	-	0.2103	614.1	-	-
	2^7	28992	13	26944	6	-	-	0.7619	1084.8	-	-
	2^8	107136	7	103039	3	-	-	3.9336	2127.2	-	-
Two-Local Random Circuit	2^3	372	61	372	61	1.6087	62.4	0.0287	78.2	-	-
	2^4	936	48	936	48	M	M	0.0429	86.7	-	-
	2^5	2640	34	2640	34	-	-	0.0781	257.2	-	-
	2^6	8352	22	8352	22	-	-	0.1895	199.0	-	-
	2^7	28992	13	28992	13	-	-	0.6302	989.7	-	-
	2^8	107136	7	107136	7	-	-	2.7349	1936.1	-	-
	2^9	410880	4	410880	4	-	-	14.7887	4238.4	-	-
Variational Quantum Eigensolver	2^3	230	74	230	74	0.313	50.1	0.0293	78.2	0.9504	61.4
	2^4	462	73	462	73	M	M	0.0369	78.2	3.7851	185.3
W State	2^3	158	63	141	51	0.002	0.0	0.0256	77.8	0.133	31.3
	2^4	334	63	301	51	0.0031	0.0	0.0355	85.6	0.5789	49.2
	2^5	686	64	621	51	M	M	0.0577	288.7	2.3315	127.9
	2^6	1390	64	1261	51	-	-	0.101	579.3	7.4094	346.4
	2^7	2798	64	2541	51	-	-	0.2338	948.4	54.9538	2513.5
	2^8	5614	64	5101	51	-	-	0.6115	1673.5	579.513	7218.0
	2^9	11246	64	10221	51	-	-	1.4435	2744.6	M	M
	2^{10}	22510	64	20461	51	-	-	6.8719	6066.4	-	-
	2^{11}	45038	64	40941	51	-	-	M	M	-	-

Table A.4: Results for verifying equivalence between circuits from the MQT benchmark set. The optimized circuits contain an error: the control and target qubit of random $CNOT$ gate have been flipped.

Algorithm	**#Qubits**	**transp**		**flipped**		**CTQC**		**QCEC**		**ECMC**	
		$\#G$	$\%R_z$	$\#G$	$\%R_z$	t(sec)	mem(mbs)	t(sec)	mem(mbs)	t(sec)	mem(mbs)
Amplitude Estimation	2^3	296	67	296	67	0.2429	14.3	0.0286	78.3	-	-
	2^4	924	65	924	65	M	M	0.0682	91.3	-	-
	2^5	3140	63	3140	63	-	-	-	-	-	-
Deutsch-Josza	2^3	66	67	33	34	0.001	0.0	0.0247	84.5	0.0173	7.4
	2^4	114	60	63	27	0.0012	0.0	0.0315	84.7	0.0359	7.1
	2^5	242	62	131	29	0.0013	0.0	0.0446	85.0	0.0626	7.3
	2^6	466	60	259	27	0.0016	0.0	0.0721	159.2	0.1763	23.9
	2^7	890	58	509	25	0.0016	0.0	N	N	0.8161	25.4
	2^8	1762	57	1015	25	0.0021	0.0	N	N	2.5174	69.5
	2^9	3402	55	2001	24	0.0032	5.5	N	N	10.0384	329.9
	2^{10}	7058	57	4067	25	0.0034	5.0	1.3625	6034.4	13.7428	1241.9
	2^{11}	14114	57	8135	25	0.0079	5.0	M	M	34.9062	374.1
	2^{12}	28362	57	16305	25	0.0099	5.1	-	-	-	-
	2^{13}	57058	57	32695	25	0.0248	22.9	-	-	-	-
GHZ State	2^3	8	0	8	0	0.0008	0.0	0.025	84.2	0.0098	7.7
	2^4	16	0	16	0	0.0016	0.0	0.031	84.2	0.0119	7.6
	2^5	32	0	32	0	0.001	0.0	0.044	84.6	0.0167	7.6
	2^6	64	0	64	0	0.0014	0.0	0.069	228.1	0.0283	7.6
	2^7	128	0	128	0	0.002	0.0	0.1188	761.1	0.0577	7.6
	2^8	256	0	256	0	0.0055	0.0	0.2507	1178.0	0.1489	33.2
	2^9	512	0	512	0	0.006	0.0	0.5689	3137.2	0.5091	61.4
	2^{10}	1024	0	1024	0	0.0177	6.3	2.3329	6069.8	1.7283	152.6
	2^{11}	2048	0	2048	0	0.0839	7.9	M	M	6.7064	506.5
	2^{12}	4096	0	4096	0	0.3583	11.1	-	-	21.1165	1295.4
	2^{13}	8192	0	8192	0	1.1832	17.9	-	-	128.0016	7148.8
	2^{14}	16384	0	16384	0	0.0421	6.4	-	-	-	-
	2^{15}	32768	0	32768	0	65.0309	53.4	-	-	-	-
Graph State	2^3	32	0	16	0	0.0016	0.0	0.0251	84.3	0.012	7.6
	2^4	64	0	32	0	0.0021	0.0	0.0321	84.8	0.0165	7.6
	2^5	128	0	64	0	0.0027	0.0	0.0453	84.8	0.0282	7.6
	2^6	256	0	128	0	0.0022	0.0	0.0746	410.8	0.0599	7.7
	2^7	512	0	256	0	0.0024	0.0	0.1763	814.8	0.1314	33.4
	2^8	1024	0	512	0	0.0055	0.0	0.2412	963.5	0.3034	45.3
	2^9	2048	0	1024	0	0.0157	5.1	0.5753	3088.6	0.983	84.7
	2^{10}	4096	0	2048	0	0.0736	6.4	1.2388	6026.7	2.216	130.3
	2^{11}	8192	0	4096	0	0.3099	8.2	M	M	10.8704	241.3
	2^{12}	16384	0	8192	0	0.0134	10.3	-	-	-	-
	2^{13}	32768	0	16384	0	2.3162	17.5	-	-	-	-
	2^{14}	65536	0	32768	0	20.4039	30.5	-	-	-	-
	2^{15}	131072	0	65536	0	25.1613	57.5	-	-	-	-
Portfolio Optimization using QAOA	2^3	508	56	508	56	14.0081	181.3	1.2703	349.9	-	-
	2^4	1592	48	1592	48	M	M	-	-	-	-
Portfolio Optimization using VQE	2^3	540	42	414	55	1.153	84.2	0.1899	298.3	-	-
	2^4	1656	28	1026	44	M	M	-	-	-	-
QAOA	2^3	200	64	200	64	0.0023	0.0	0.0274	84.9	-	-
	2^4	400	64	400	64	0.002	0.0	0.0477	87.5	-	-
Quantum Fourier Transform	2^3	160	53	139	46	0.2761	16.4	0.0264	84.7	-	-
	2^4	640	57	535	48	M	M	1.6507	423.0	-	-
	2^5	2560	59	1675	49	-	-	-	-	-	-
Entangled Quantum Fourier Transform	2^3	168	50	147	43	0.2583	14.6	0.0356	229.9	-	-
	2^4	656	55	551	47	M	M	18.5828	852.1	-	-
	2^5	2592	58	1707	48	-	-	-	-	-	-
Quantum Neural Network	2^3	399	59	323	44	8.9276	179.4	0.0355	243.3	M	M
	2^4	1183	51	1002	40	M	M	-	-	-	-
Exact Quantum Phase Estimation	2^3	167	51	167	51	0.091	5.8	0.0403	85.2	-	-
	2^4	655	55	655	55	M	M	0.1542	381.1	-	-
	2^5	2591	58	2591	58	-	-	-	-	-	-
Inexact Quantum Phase Estimation	2^3	167	51	167	51	0.1955	12.1	0.0305	79.4	-	-
	2^4	655	55	655	55	M	M	1.215	545.7	-	-
	2^5	2591	58	2591	58	-	-	-	-	-	-
Quantum Walk without Ancilla	2^3	10937	62	10913	62	-	-	0.2953	306.1	-	-
	2^4	219567	66	219543	66	-	-	78.5985	902.5	-	-
	2^5	2453631	67	2453607	67	-	-	-	-	-	-
Quantum Walk with V-Chain	2^3	2337	42	2085	48	M	M	0.7978	334.4	-	-
	2^4	11457	44	10213	49	-	-	-	-	-	-
Real Amplitude Random Circuit	2^3	372	61	241	39	0.7915	45.5	0.666	324.4	-	-
	2^4	936	48	677	28	M	M	0.0508	90.4	-	-
	2^5	2640	34	2125	18	-	-	-	-	-	-
SU Random Circuit	2^3	372	61	241	39	4.9076	166.1	0.03	85.4	-	-
	2^4	936	48	680	29	M	M	0.3222	410.6	-	-
	2^5	2640	34	2127	19	-	-	-	-	-	-

Table A.5: Results for verifying equivalence between circuits from the MQT benchmark set. The optimized circuits contain an error: the control and target qubit of random $CNOT$ gate have been flipped (continued).

Algorithm	**#Qubits**	**transp**		**flipped**		**CTQC**		**QCEC**		**ECMC**	
		$\#G$	$\%R_z$	$\#G$	$\%R_z$	t(sec)	mem(mbs)	t(sec)	mem(mbs)	t(sec)	mem(mbs)
Two-Local Random Circuit	2^3	372	61	372	61	0.826	47.5	0.2275	307.4	-	-
	2^4	936	48	936	48	M	M	-	-	-	-
Variational Quantum Eigensolver	2^3	230	74	230	74	0.145	36.3	0.0289	85.2	1.0014	61.8
	2^4	462	73	462	73	M	M	0.2391	350.5	3.8236	184.4
W State	2^3	158	63	141	51	0.0023	0.0	0.0268	84.8	0.1531	31.1
	2^4	334	63	301	51	0.6741	356.8	0.0389	85.0	0.9942	64.8
	2^5	686	64	621	51	M	M	0.0567	91.0	2.5579	132.7
	2^6	1390	64	1261	51	-	-	0.1033	455.6	10.6007	568.1
	2^7	2798	64	2541	51	-	-	W	W	94.3515	4679.7
	2^8	5614	64	5101	51	-	-	W	W	-	-
	2^9	11246	64	10221	51	-	-	W	W	-	-
	2^{10}	22510	64	20461	51	-	-	W	W	-	-
	2^{11}	45038	64	40941	51	-	-	M	M	-	-

Table A.6: Results for verifying equivalence between circuits from the MQT benchmark set. The optimized circuits contain an error: a random gate has been removed from the circuit.

Algorithm	#Qubits	transp		gm		CTQC		QCEC		ECMC	
		#G	$\%R_z$	#G	$\%R_z$	t(sec)	mem(mbs)	t(sec)	mem(mbs)	t(sec)	mem(mbs)
Amplitude Estimation	2^3	296	67	295	67	0.2728	14.0	0.0294	85.3	-	-
	2^4	924	65	923	65	M	M	0.1949	322.6	-	-
	2^5	3140	63	3139	63	-	-	0.0976	403.5	-	-
	2^6	11412	62	11411	62	-	-	E	E	-	-
Deutsch-Josza	2^3	66	67	32	35	0.0011	0.0	0.0263	84.5	0.0192	7.4
	2^4	114	60	62	26	0.0016	0.0	0.035	84.7	0.0276	7.5
	2^5	242	62	130	29	0.0016	0.0	0.046	85.2	0.0566	7.4
	2^6	466	60	258	27	0.002	0.0	0.078	88.9	0.1367	26.4
	2^7	890	58	508	25	0.0022	0.0	N	N	0.4779	39.7
	2^8	1762	57	1014	25	0.0276	5.1	N	N	2.2064	65.1
	2^9	3402	55	2000	24	0.0195	5.0	N	N	13.7389	351.3
	2^{10}	7058	57	4066	25	0.1852	6.8	2.2781	6040.0	4.7257	320.0
	2^{11}	14114	57	8134	25	0.2555	9.3	M	M	44.4552	5036.5
	2^{12}	28362	57	16304	25	3.9314	13.3	-	-	-	-
	2^{13}	57058	57	32694	25	24.6788	23.0	-	-	-	-
GHZ State	2^3	8	0	7	0	0.0011	0.0	0.0259	84.3	0.0099	7.5
	2^4	16	0	15	0	0.0022	0.0	0.0324	84.6	0.0122	7.4
	2^5	32	0	31	0	0.0028	0.0	0.0454	84.5	0.0176	7.4
	2^6	64	0	63	0	0.0021	0.0	0.0706	271.7	0.0284	7.4
	2^7	128	0	127	0	0.0034	0.0	0.1196	863.5	0.0681	15.3
	2^8	256	0	255	0	0.0039	0.0	0.2427	1061.8	0.1588	33.1
	2^9	512	0	511	0	0.0091	0.0	0.4945	2698.5	0.4131	51.7
	2^{10}	1024	0	1023	0	0.053	6.1	0.9044	5311.3	2.0838	146.3
	2^{11}	2048	0	2047	0	0.0138	7.8	M	M	7.5323	490.2
	2^{12}	4096	0	4095	0	0.8347	10.9	-	-	27.3429	1838.0
	2^{13}	8192	0	8191	0	3.612	16.1	-	-	81.4822	5037.6
	2^{14}	16384	0	16383	0	5.5478	28.4	-	-	-	-
	2^{15}	32768	0	32767	0	10.8387	49.8	-	-	-	-
Graph State	2^3	32	0	15	0	0.001	0.0	0.0243	78.1	0.0126	7.4
	2^4	64	0	31	0	0.0026	0.0	0.0324	84.8	0.0204	7.4
	2^5	128	0	63	0	0.0031	0.0	0.0465	86.5	0.0304	7.4
	2^6	256	0	127	0	0.0024	0.0	0.0726	361.3	0.0702	15.3
	2^7	512	0	255	0	0.0035	0.0	0.1742	762.7	0.1888	34.6
	2^8	1024	0	511	0	0.0036	0.0	0.317	1704.3	0.3105	39.9
	2^9	2048	0	1023	0	0.0151	4.8	0.5213	3097.3	1.2434	76.4
	2^{10}	4096	0	2047	0	0.0816	6.4	1.8085	6028.3	-	-
	2^{11}	8192	0	4095	0	0.2891	8.2	M	M	-	-
	2^{12}	16384	0	8191	0	1.1335	12.1	-	-	-	-
	2^{13}	32768	0	16383	0	9.5845	17.8	-	-	-	-
	2^{14}	65536	0	32767	0	15.1475	32.8	-	-	-	-
	2^{15}	131072	0	65535	0	59.8538	59.1	-	-	-	-
Portfolio Optimization using QAOA	2^3	508	56	507	57	14.1299	181.3	2.0959	349.7	-	-
	2^4	1592	48	1591	48	M	M	-	-	-	-
Portfolio Optimization using VQE	2^3	540	42	413	54	2.3722	127.0	N	N	-	-
	2^4	1656	28	1025	44	M	M	-	-	-	-
QAOA	2^3	200	64	199	65	0.0016	0.0	0.032	88.6	-	-
	2^4	400	64	399	64	0.003	0.0	0.0374	85.4	-	-
Quantum Fourier Transform	2^3	160	53	138	46	0.2177	16.1	0.0274	84.8	-	-
	2^4	640	57	534	48	M	M	N	N	-	-
Entangled Quantum Fourier Transform	2^3	168	50	146	44	0.2705	19.3	0.0318	192.1	-	-
	2^4	656	55	550	47	M	M	1.3519	522.8	-	-
	2^5	2592	58	1706	48	-	-	-	-	-	-
Quantum Neural Network	2^3	399	59	322	44	8.9272	179.8	0.6876	303.7	M	M
	2^4	1183	51	1001	40	M	M	-	-	-	-
Exact Quantum Phase Estimation	2^3	167	51	166	50	0.2089	12.0	0.0277	85.0	-	-
	2^4	655	55	654	56	M	M	0.344	497.8	-	-
	2^5	2591	58	2590	58	-	-	-	-	-	-
Inexact Quantum Phase Estimation	2^3	167	51	166	50	0.1816	12.3	0.0285	79.0	-	-
	2^4	655	55	654	56	M	M	118.2213	1418.2	-	-
	2^5	2591	58	2590	58	-	-	-	-	-	-
Quantum Walk without Ancilla	2^3	10937	62	10912	62	-	-	0.5104	324.0	-	-
	2^4	219567	66	219542	66	-	-	137.1825	1113.3	-	-
	2^5	2453631	67	2453606	67	-	-	-	-	-	-
Quantum Walk with V-Chain	2^3	2337	42	2084	48	M	M	2.571	327.2	-	-
	2^4	11457	44	10212	49	-	-	-	-	-	-
Real Amplitude Random Circuit	2^3	372	61	240	39	0.7904	45.0	0.0322	89.0	-	-
	2^4	936	48	676	28	M	M	-	-	-	-
SU Random Circuit	2^3	372	61	240	39	4.8355	171.4	0.2042	316.2	-	-
	2^4	936	48	679	29	M	M	-	-	-	-
Two-Local Random Circuit	2^3	372	61	371	61	0.769	45.8	0.5172	335.8	-	-
	2^4	936	48	935	48	M	M	0.0699	180.4	-	-
	2^5	2640	34	2639	34	-	-	-	-	-	-
Variational Quantum Eigensolver	2^3	230	74	229	73	0.1734	42.1	0.0288	85.3	0.9661	62.1
	2^4	462	73	461	73	M	M	W	W	W	W

Table A.7: Results for verifying equivalence between circuits from the MQT benchmark set. The optimized circuits contain an error: a random gate has been removed from the circuit (continued).

Algorithm	**#Qubits**	**transp**		**gm**		**CTQC**		**QCEC**		**ECMC**	
		$\#G$	$\%R_z$	$\#G$	$\%R_z$	t(sec)	mem(mbs)	t(sec)	mem(mbs)	t(sec)	mem(mbs)
W State	2^3	158	63	140	51	0.0012	0.0	0.0276	84.7	0.1683	31.2
	2^4	334	63	300	51	0.0017	0.0	0.0348	85.2	0.5926	48.6
	2^5	686	64	620	50	M	M	0.0644	159.2	2.379	129.1
	2^6	1390	64	1260	50	-	-	0.1124	594.3	7.3463	346.7
	2^7	2798	64	2540	51	-	-	W	W	84.7567	3237.0
	2^8	5614	64	5100	50	-	-	W	W	-	-
	2^9	11246	64	10220	50	-	-	W	W	-	-
	2^{10}	22510	64	20460	50	-	-	W	W	-	-
	2^{11}	45038	64	40940	51	-	-	M	M	-	-

Table A.8: Results for verifying equivalence between circuits from the MQT benchmark set. The optimized circuits contain an error: 10^{-4} has been added to the phases of the rotation gates.

Algorithm	**#Qubits**	**transp**		**shift4**		**CTQC**		**QCEC**		**ECMC**	
		$\#G$	$\%R_z$	$\#G$	$\%R_z$	t(sec)	mem(mbs)	t(sec)	mem(mbs)	t(sec)	mem(mbs)
Amplitude Estimation	2^3	296	67	296	67	2.0323	87.5	-	-	-	-
	2^4	924	65	924	65	M	M	-	-	-	-
Deutsch-Josza	2^3	66	67	33	34	0.0008	0.0	N	N	0.0234	7.4
	2^4	114	60	63	27	0.0015	0.0	N	N	0.0344	7.4
	2^5	242	62	131	29	0.002	0.0	-	-	0.0765	31.0
	2^6	466	60	259	27	0.0026	0.0	-	-	0.1863	37.8
	2^7	890	58	509	25	0.0075	5.1	-	-	0.5697	40.5
	2^8	1762	57	1015	25	0.0306	5.4	-	-	2.6792	94.6
	2^9	3402	55	2001	24	0.2189	6.5	-	-	4.1621	358.9
	2^{10}	7058	57	4067	25	1.5348	9.0	-	-	17.1163	1324.2
	2^{11}	14114	57	8135	25	12.1692	14.0	-	-	67.7668	5111.2
	2^{12}	28362	57	16305	25	17.8883	16.6	-	-	-	-
	2^{13}	57058	57	32695	25	-	-	-	-	-	-
Portfolio Optimization using QAOA	2^3	508	56	508	56	19.9938	181.3	83.676	349.8	-	-
	2^4	1592	48	1592	48	M	M	-	-	-	-
Portfolio Optimization using VQE	2^3	540	42	414	55	10.2394	165.8	132.0622	349.4	-	-
	2^4	1656	28	1026	44	M	M	-	-	-	-
QAOA	2^3	200	64	200	64	0.36	40.9	2.2478	286.7	-	-
	2^4	400	64	400	64	0.9671	95.1	-	-	-	-
Quantum Fourier Transform	2^3	160	53	139	46	0.241	17.8	0.026	78.2	-	-
	2^4	640	57	535	48	M	M	-	-	-	-
Entangled Quantum Fourier Transform	2^3	168	50	147	43	0.2788	23.8	0.0509	267.3	-	-
	2^4	656	55	551	47	M	M	-	-	-	-
Quantum Neural Network	2^3	399	59	323	44	10.3287	188.0	4.3314	307.6	M	M
	2^4	1183	51	1002	40	M	M	-	-	-	-
Exact Quantum Phase Estimation	2^3	167	51	167	51	0.1237	11.2	0.0838	274.3	-	-
	2^4	655	55	655	55	-	-	-	-	-	-
Inexact Quantum Phase Estimation	2^3	167	51	167	51	0.2569	14.3	0.0927	267.7	-	-
	2^4	655	55	655	55	M	M	-	-	-	-
Real Amplitude Random Circuit	2^3	372	61	241	39	4.7936	168.3	147.5878	331.6	-	-
	2^4	936	48	677	28	M	M	-	-	-	-
SU Random Circuit	2^3	372	61	241	39	5.7147	185.1	9.3239	329.0	-	-
	2^4	936	48	680	29	M	M	-	-	-	-
Two-Local Random Circuit	2^3	372	61	372	61	8.5217	165.4	91.7066	321.9	-	-
	2^4	936	48	936	48	M	M	-	-	-	-
Variational Quantum Eigensolver	2^3	230	74	230	74	3.5237	160.5	66.1867	287.6	1.2429	68.2
	2^4	462	73	462	73	M	M	-	-	4.8133	229.2
W State	2^3	158	63	141	51	0.5853	62.0	0.0268	85.1	0.1941	34.2
	2^4	334	63	301	51	M	M	-	-	0.7208	55.1
	2^5	686	64	621	51	-	-	-	-	3.108	148.9
	2^6	1390	64	1261	51	-	-	-	-	9.5291	416.1
	2^7	2798	64	2541	51	-	-	-	-	96.5288	3419.6
	2^8	5614	64	5101	51	-	-	-	-	-	-

Table A.9: Results for verifying equivalence between circuits from the MQT benchmark set. The optimized circuits contain an error: 10^{-7} has been added to the phases of the rotation gates.

Algorithm	**#Qubits**	**transp**		**shift7**		**CTQC**		**QCEC**		**ECMC**	
		$\#G$	$\%R_z$	$\#G$	$\%R_z$	t(sec)	mem(mbs)	t(sec)	mem(mbs)	t(sec)	mem(mbs)
Amplitude Estimation	2^3	296	67	296	67	1.905	90.1	N	N	-	-
	2^4	924	65	924	65	M	M	-	-	-	-
Deutsch-Josza	2^3	66	67	33	34	0.0011	0.0	0.0261	81.8	W	W
	2^4	114	60	63	27	0.0012	0.0	N	N	W	W
	2^5	242	62	131	29	0.0011	0.0	-	-	W	W
	2^6	466	60	259	27	0.0025	0.0	-	-	W	W
	2^7	890	58	509	25	0.0072	5.1	-	-	W	W
	2^8	1762	57	1015	25	0.0305	5.6	-	-	W	W
	2^9	3402	55	2001	24	0.2006	6.8	-	-	4.083	358.6
	2^{10}	7058	57	4067	25	1.387	9.2	-	-	16.6816	1324.5
	2^{11}	14114	57	8135	25	12.2582	13.9	-	-	65.4598	5111.8
	2^{12}	28362	57	16305	25	71.5098	17.1	-	-	-	-
	2^{13}	57058	57	32695	25	-	-	-	-	-	-
Portfolio Optimization using QAOA	2^3	508	56	508	56	19.7688	189.6	-	-	-	-
	2^4	1592	48	1592	48	M	M	-	-	-	-
Portfolio Optimization using VQE	2^3	540	42	414	55	10.2377	166.0	-	-	-	-
	2^4	1656	28	1026	44	M	M	-	-	-	-
QAOA	2^3	200	64	200	64	0.359	43.7	N	N	-	-
	2^4	400	64	400	64	0.9396	81.1	-	-	-	-
Quantum Fourier Transform	2^3	160	53	139	46	0.2385	18.8	N	N	-	-
	2^4	640	57	535	48	M	M	-	-	-	-
Entangled Quantum Fourier Transform	2^3	168	50	147	43	0.2831	22.9	N	N	-	-
	2^4	656	55	551	47	M	M	-	-	-	-
Quantum Neural Network	2^3	399	59	323	44	10.3185	179.4	N	N	M	M
	2^4	1183	51	1002	40	M	M	-	-	-	-
Exact Quantum Phase Estimation	2^3	167	51	167	51	0.1161	10.9	N	N	-	-
	2^4	655	55	655	55	-	-	-	-	-	-
Inexact Quantum Phase Estimation	2^3	167	51	167	51	0.2436	13.4	N	N	-	-
	2^4	655	55	655	55	M	M	-	-	-	-
Real Amplitude Random Circuit	2^3	372	61	241	39	4.6352	165.7	-	-	-	-
	2^4	936	48	677	28	M	M	-	-	-	-
SU Random Circuit	2^3	372	61	241	39	5.9306	168.2	-	-	-	-
	2^4	936	48	680	29	M	M	-	-	-	-
Two-Local Random Circuit	2^3	372	61	372	61	8.2396	170.2	-	-	-	-
	2^4	936	48	936	48	M	M	-	-	-	-
Variational Quantum Eigensolver	2^3	230	74	230	74	3.5347	155.1	N	N	1.1873	70.1
	2^4	462	73	462	73	M	M	-	-	5.0224	230.3
W State	2^3	158	63	141	51	0.5992	62.0	N	N	W	W
	2^4	334	63	301	51	M	M	-	-	W	W
	2^5	686	64	621	51	-	-	-	-	3.1131	149.1
	2^6	1390	64	1261	51	-	-	-	-	8.927	416.7
	2^7	2798	64	2541	51	-	-	-	-	94.8188	3419.5
	2^8	5614	64	5101	51	-	-	-	-	-	-