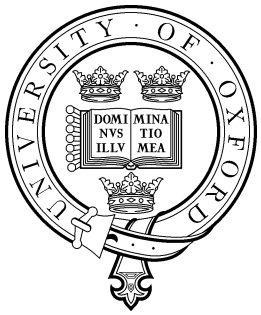
GNNs for Fast Classical Simulation of Quantum Circuits through Optimizing ZX-Graph Decompositions



David Philipps

Keble College

University of Oxford

A thesis submitted for the degree of

MSc in Mathematics and Foundations of Computer Science

Trinity 2025

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Richie Yeung and Matthew Sutcliffe, for their invaluable guidance and support throughout this project, and for proofreading this thesis.

I also wish to thank my pomodoro partner, Pedro B. R. de Costa, for helping to maintain a productive writing process during the final weeks.

Lastly, I wish to extend my deepest gratitude to my family and the German Academic Scholarship Foundation for making this year at Oxford possible for me. My heartfelt thanks also go to my friends at Keble College for making it such a wonderful time.

Abstract

Quantum circuits can be simulated on classical hardware using an algorithm known as stabiliser decomposition. This procedure is based on two principles: first, any quantum circuit can be constructed out of the universal Clifford+T gate set; second, circuits made only of Clifford gates can be simulated efficiently on classical computers. The core idea is to recursively decompose a Clifford+T circuit into a linear combination of circuits with a lower T-gate count (T-count) than the original, ultimately producing an exponential number of Clifford circuits. Each circuit can then be simulated individually, and the results are aggregated to obtain the final output. This thesis examines the application of machine learning models to inform the selection of intermediate decompositions, with the goal of minimising the number of T-gates in the resulting circuits. Even minor improvements in this recursive step can combine to yield significant enhancements in the overall simulation.

Specifically, we work in the framework of Kissinger and van de Wetering [16]. They combined known gadget decompositions with circuit optimisation routines to reduce the T-count. We also draw on the work of Ahmad et al. [4], who developed heuristics that apply these decompositions strategically to maximise simplification. Building on prior research by Koziell-Pipe, Sutcliffe, and Yeung [22], we trained a machine learning model that outperformed even the greedy brute-force baseline for applying the Vertex-Cut decomposition on exponentiated Pauli circuits. This demonstrates its ability to learn efficient, non-local simplification strategies. In addition, we document several machine learning techniques that did not yield promising results. We analyse their shortcomings and discuss potential avenues for future improvement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many modern digital industries rely on vast amounts of computational power. In the past, we have seen this power grow exponentially from year to year. However, this trend, known as Moore's law, which is a driving factor of our modern world, is predicted to slow down over the next decades as classical hardware reaches physical limitations [11]. Quantum computing is a new model of computation that harnesses the counter-intuitive quantum behaviour of physical systems at the atomic level to speed up computations in a way that is not possible using classical hardware. In the current era of this emerging quantum advantage, the classical simulation of quantum circuits is a difficult yet fruitful endeavour. On the practical side, we want to understand the behaviour of quantum circuits in the absence of an actual implementation, since in most cases the latter is either expensive or not yet possible. Conversely, we also need to verify the correctness of implemented quantum circuits by comparing their output distribution to our simulation. On the theoretical side, however, our hopes for a universally efficient simulator are tempered by the $\#P$-hardness result [27] for the simulation of random quantum circuits.

Consequently, a toolkit of different approaches has emerged, each with its own domain of applicability. Statevector simulations are well-known and used in cases where only a few qubits are involved, and tensor-contraction-based techniques excel even in setups with many qubits, as long as the circuit is not too dense [44]. The third, lesser-known family of techniques, based on stabiliser decompositions, offers good results for circuits that contain only a small number of non-Clifford gates. The basic idea is to decompose the circuit into a sum of Clifford circuits, which can be simulated in polynomial time due to the Gottesmann-Knill theorem [1], and then aggregate the results. As this technique matures, efforts are being made to speed up this aggregation [40] and reduce the size of the generated decomposition [5, 38, 9].

Consistent with previous developments in this area, we will work in the graphical language of ZX-calculus to represent quantum circuits as diagrams. These diagrams have a versatile structure and a rich set of rewrite rules that make them an appealing use case for graph-based machine learning. Related work, such as [25, 30, 34], has successfully employed a combination of GNNs and state-of-the-art reinforcement learning techniques to perform local rewrites to optimise ZX diagrams, for example, by reducing the 2-qubit gate count.

In [22], the authors already employed a similar setup to optimise the decision-making process of iterated decomposition. This thesis builds on their work, expanding the toolkit of machine learning approaches and the theoretical understanding, while also employing findings from new research on stabiliser decompositions.

1.1 Outline

This dissertation is organised as follows. Chapter 3 begins by detailing the specific problem addressed with machine learning and defining the baseline models against which our results are compared. Following this, Chapter 4 describes the supervised learning setups developed for decomposition application selection and provides an analysis of their performance. Chapter 5 then reformulates this task as a reinforcement learning problem, discussing several adaptations to the original framework proposed in [22]. In Chapter 6, we introduce a regression-based technique to train the α-estimation component of the selection drivers. We conclude by combining our best-performing models into a single, unified driver capable of leveraging multiple base decompositions.

Chapter 2

Background

This chapter introduces the necessary background on quantum circuits as a model for quantum computation and the methods by which they can be simulated. Furthermore, we will cover the machine learning techniques employed to build our models. For a thorough introduction to the ZX-Calculus, the reader is referred to the book *Picturing Quantum Software* by Kissinger and van de Wetering [17].

2.1 The SCUM view on Quantum Theory

Describing a computational process involves defining the possible states of a system, the actions that can be performed on those states, and the method for reading out the result. In a classical computer, the state space is $\mathbb{B}^n$, which is the set of bitstrings (vectors where each element is either 0 or 1) of size n. These evolve through the application of logic gates such as AND or OR. After the algorithm completes, the result can be read directly from the bits. These concepts are formalised in models of computation such as circuits or Turing machines. Quantum computers, in contrast, adhere to the arguably more complex framework of quantum theory. We introduce the formulation of quantum computation as presented in [17]:

2.1.1 States

The state space of a quantum system is the normalised subspace of a complex Hilbert space, $\mathcal{H}$. For a qubit, the simplest unit of quantum information, this space is $\mathcal{H} = \mathbb{C}^2$. To describe states within this space, we fix a basis that typically corresponds to a physically meaningful property, such as the spin of a neutron. This is called the computational basis, denoted by $\{|0\rangle, |1\rangle\}$, where:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ and } |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Any other qubit state $|\psi\rangle \in \mathbb{C}^2$ can be expressed as a linear combination $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where $\alpha, \beta \in \mathbb{C}$ are called amplitudes and satisfy $|\alpha|^2 + |\beta|^2 = 1$. A state such as $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is said to be in a superposition of the $|0\rangle$ and $|1\rangle$ states.

2.1.2 Compound Systems

When combining two quantum systems with state spaces $\mathcal{H}_1$ and $\mathcal{H}_2$, the state space of the combined system is described by their tensor product, $\mathcal{H}_1 \otimes \mathcal{H}_2$. For two qubits, this results in $\mathbb{C}^2 \otimes \mathbb{C}^2 = \mathbb{C}^{2\cdot 2}$, and for n qubits, it is $\mathbb{C}^{2^n}$. The computational basis for this space is composed of bitstrings $|\vec{b}\rangle = |b_1\rangle \otimes \cdots \otimes |b_n\rangle$ for $\vec{b} \in \mathbb{B}^n$. If a quantum computer were always in a computational basis state, it could be described just like a classical computer. The power of quantum computing, however, stems from the fact that qubits can exist in any superposition of these basis states. Such a superposition can be described by $\Sigma_{\vec{c}\in\mathbb{B}^n} a_{\vec{c}}|\vec{c}\rangle$, where the $a_{\vec{c}} \in \mathbb{C}$ are amplitudes. The vector containing all amplitudes, known as the statevector, is normalised in the 2-norm and uniquely describes the quantum state.

2.1.3 Unitary Evolution

A quantum computer evolves from one quantum state to another via unitary linear maps, which are usually represented by matrices in $\mathbb{C}^{2^n \times 2^n}$. For example, starting with a qubit in the state $|0\rangle$, the state $|+\rangle$ can be obtained using the unitary map:

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

If we have a composite system such as $\mathbb{C}^2 \otimes \mathbb{C}^2$ and apply a unitary U_1 to the first qubit and U_2 to the second, this corresponds to applying the Kronecker product $U_1 \otimes U_2$ to the whole system.

2.1.4 Measurement

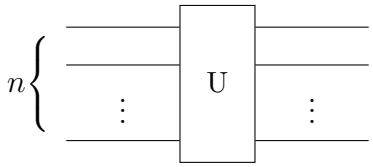
The formalism above describes the internal workings of a quantum computer. However, in stark contrast to a classical machine, we cannot directly access the final quantum state. Measuring the qubits after a computation is a complex process; here, we will only describe a simplified version. To measure a qubit, one must specify a basis, and for simplicity, we assume all qubits are measured in the computational basis.

The Born rule dictates that for a quantum state $\psi = \sum_{\vec{c}\in\mathbb{B}^n} a_{\vec{c}} |\vec{c}\rangle$, a measurement of all qubits will yield the outcome $\vec{c}$ with probability $|a_{\vec{c}}|^2$.

Simulating a quantum computer can now be understood as classically computing these probabilities, given the initial state and the unitary evolution of the computer. First, however, we will introduce a model for describing this evolution.

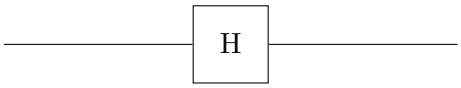
2.2 Quantum Circuits

Quantum circuits are the most popular model for quantum computation. They represent unitary operations on qubits as



where U is a unitary linear operator on the Hilbert space of n qubits, usually represented by a matrix in $\mathbb{C}^{2^n\times 2^n}$. In this formalism, a wire represents a qubit that is acted upon. Some examples of small quantum circuits, which we also call gates, are:

Hadamard Gate



with the unitary operator

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

This gate maps the computational basis states to superpositions. We have

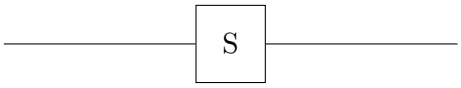
$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle \quad \text{and} \quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$$

for single qubits and

$$(H|0\rangle)^{\otimes n} = \frac{1}{\sqrt{2^n}} \sum_{\vec{b}\in\mathbb{B}^n} |\vec{b}\rangle$$

for n qubits. The set $\{|+\rangle, |-\rangle\}$ is called the Hadamard basis. We want to mention the importance of the duality between the computational basis and the Hadamard basis and refer the reader to [17] for a thorough treatment of the matter.

Phase Gate



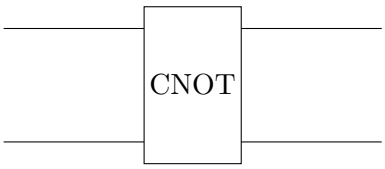
with the unitary operator

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}.$$

Since this gate is diagonal in the computational basis, it has no physically meaningful effect on these basis states. On superpositions, though, it introduces a phase shift, for example,

$$S^2 |+\rangle = S^2 \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle .$$

Controlled-Not Gate

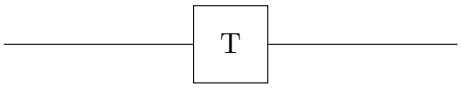


with the unitary operator

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

This two-qubit gate acts as a bitflip on the second qubit if the first qubit is in the $|1\rangle$ state and as the identity otherwise if it is in the $|0\rangle$ state. It therefore plays an important role in emulating classical computation steps on a quantum computer.

T Gate

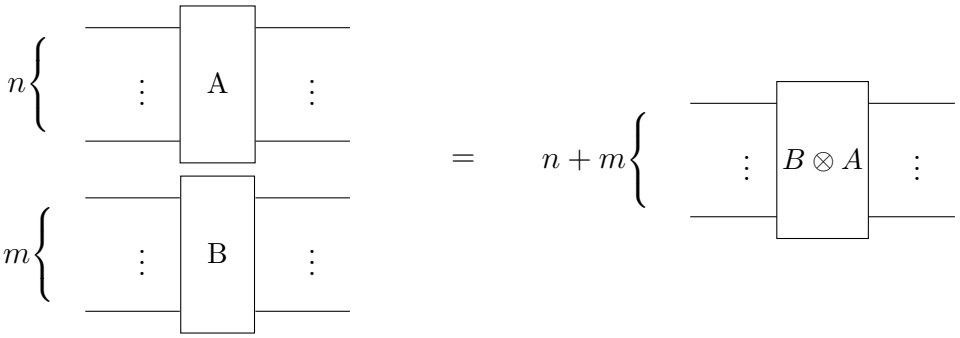


with the unitary operator

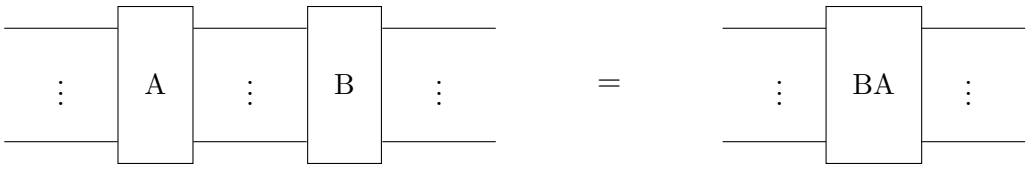
$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix}.$$

This gate has an interesting role in the simulation of quantum circuits, which we explain in the next section.

Circuits can be composed either in parallel,



where $B \otimes A$ denotes the Kronecker product, or, if the number of qubits matches, in sequence,

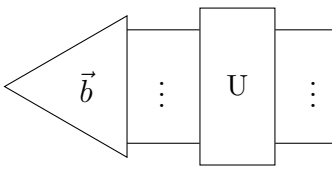


where BA denotes the standard matrix product. Thus, the actions of the gates are read from left to right.

The first three gates (Controlled-Not, Hadamard, Phase) are so-called Clifford gates. It is worth mentioning that they are computationally cheap to implement on physical hardware and in certain error correction codes, as well as classically simulatable. See [29] and [17] for more information on Clifford gates. Together with the T gate, they form the Clifford+T gate set that is known to be approximately universal, meaning that by composing them in parallel and in sequence, any unitary operator can be constructed up to a given precision ε [29]. Therefore, when working with quantum circuits, it usually suffices to consider only those built from these generators.

2.3 Classical Simulation

Given a circuit representation of a unitary operator U on n qubits and an n-qubit classical state $|\vec{b}\rangle$ (where $\vec{b} \in \mathbb{B}^n$), the application of U to $|\vec{b}\rangle$, denoted as



results in the state $U|b\rangle \in \mathbb{C}^{2^n}$. We can write this state in the computational basis as $\sum_{\vec{c}\in\mathbb{B}^n} a_{\vec{c}}|\vec{c}\rangle$, with amplitudes $a_{\vec{c}} \in \mathbb{C}$. If this circuit were implemented on

real hardware and all qubits were measured in the computational basis, the Born rule states that the outcome would be $\vec{c}$ with probability $|a_{\vec{c}}|^2$ [29]. This is called the output distribution of the computation and we let O be the corresponding random variable. In practical use cases, this distribution is sampled multiple times by repeating the computation, and the outcomes are fed to a post-processing step. With access only to classical hardware and the descriptions of the input state $|\vec{b}\rangle$ and circuit C, one might want to perform one of two tasks:

Weak Simulation

A weak simulation (or emulation) of a quantum computation is a probabilistic algorithm A that outputs bitstrings and satisfies

$$\mathcal{P}(A(\vec{b}, C) = \vec{c}) \overset{\varepsilon}{\approx} \mathcal{P}(O = \vec{c}) = |a_{\vec{c}}|^2.^1$$

This task is believed to be hard, as its efficient solution would negate the computational advantage of quantum computers over classical machines. This computational advantage is captured in the complexity class **BQP** of problems that can be solved by weakly simulating a quantum circuit. See [29] for details on this class.

Strong Simulation

Strongly simulating a quantum computation means calculating marginal probabilities over the output distribution. Marginals are probabilities of the form

$$\mathcal{P}(O_I = \vec{b}_I)$$

where I is some set of indices. This is even harder than weak simulation for two reasons. First, a strong simulator can be used to build a weak simulator, by repeatedly calculating marginals and classically sampling from them (cf. [17] p. 202). Second, this task would not be feasible even with a quantum device, as such a device can only be used to create samples, not to access the underlying amplitudes that give rise to the distribution. Strong simulation is also a #P-hard problem [27].

However, when verifying or analysing quantum circuits, it is usually strong simulation that is needed. Therefore, this is the focus of this thesis, and we will use *simulation* synonymously with *strong simulation*. We will now introduce the most popular techniques for this task.

[1] The relation $\overset{\varepsilon}{\approx}$ denotes that the equality holds up to some error margin ε.

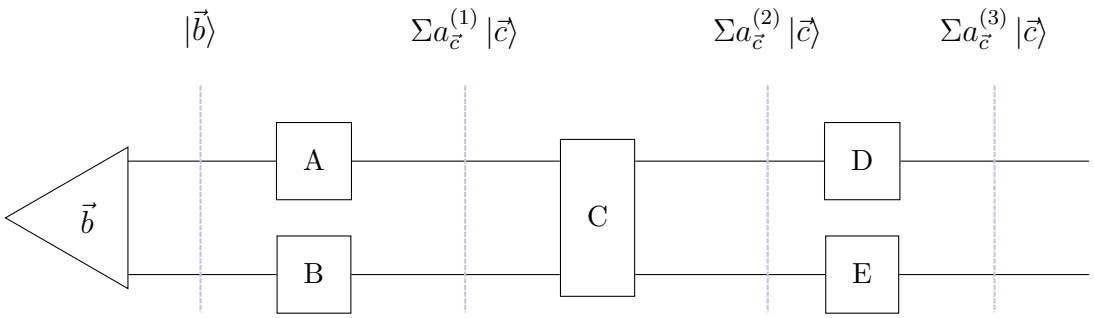


Figure 2.1: In this visualisation of statevector simulation, it is shown how the state of a quantum computer can be described by the corresponding statevectors after each application of a gate.

2.3.1 Statevector Simulation

The most straightforward approach to simulating a quantum circuit is to compute the sequence of states the system passes through, following the arrow of time implied by the circuit description. This algorithm, called statevector simulation, works by subdividing the circuit into its sequential layers and applying each, one by one, to the output of the preceding layer, starting with the input state. All intermediate states are represented by their statevector. See Figure 2.1 for a visualisation of the process. The problem with this approach is that the size of the statevector grows exponentially with the number of qubits, making it infeasible to implement this algorithm on even a supercomputer for moderate qubit counts due to memory limitations.

2.3.2 Tensor Networks

The concept of using tensor networks for the efficient simulation of quantum computation was introduced in [24]. However, their applications extend to simulating more complex physical quantum systems (see [31] for a survey) and to the emerging field of quantum machine learning [12]. Tensor networks offer a flexible framework to model various systems while also allowing for efficient[2] simulation, facilitated by libraries such as `quimb` [10].

Tensors are a generalisation of scalars, vectors, and matrices to multidimensional arrays of numbers. Each complex number in the array is addressed via r indices. We call r the *rank* of the tensor. A tensor can be seen as a relationship between r ordered sets, since the r indices for any entry can be seen as a selection of r elements from these

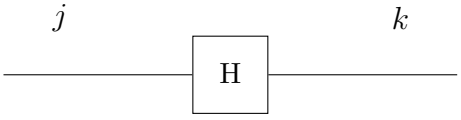
[2]In this context, "efficient" does not imply polynomial time complexity, but rather optimised and parallelised computation, for instance on GPUs.

sets. If the sets are the bases of r vector spaces, this extends to a multilinear relation between the systems. This generalises how matrices describe a linear function between two vector spaces in relation to some bases. In the context of quantum computing, the vector spaces are usually the two-dimensional Hilbert spaces that describe the state space of a qubit and we operate in the computational basis.

Vectors are rank-1 tensors and describe a single system, such as the current state of a qubit. Matrices are rank-2 tensors and describe the relation between two systems, such as between the qubit state before and after some evolution. In the circuit model, we described the application of a two-qubit gate, such as CNOT, with a unitary evolution matrix on the composite system. Using higher-rank tensors, we can also describe it as a relation between the two single-qubit states before the application and the two single-qubit states after. Since this relation is linear in every qubit state, we can use a rank-4 tensor to describe it. This formulation offers multiple advantages for simulation.

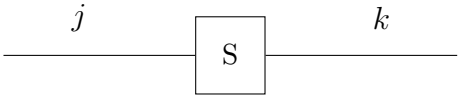
In the following, rank-n tensors are visually depicted as nodes with n open edges, each of which represents a system. In our presentation, we use the lowercase letters on the edges as indices for these systems. Here $j, k, l, m \in \mathbb{B}$, where a 0 corresponds to the basis state $|0\rangle$ and 1 to $|1\rangle$. We describe the Clifford+T gate set as tensors, by giving their entries:

Hadamard Gate



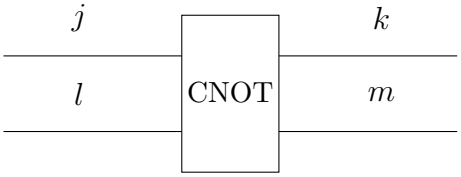
$$= \quad (-1)^{j \cdot k}$$

Phase Gate



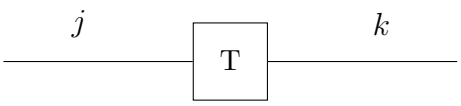
$$= \quad \begin{cases} i, & \text{if } j = k = 1 \\ 1, & \text{if } j = k = 0 \\ 0, & \text{else} \end{cases}$$

Controlled-Not Gate



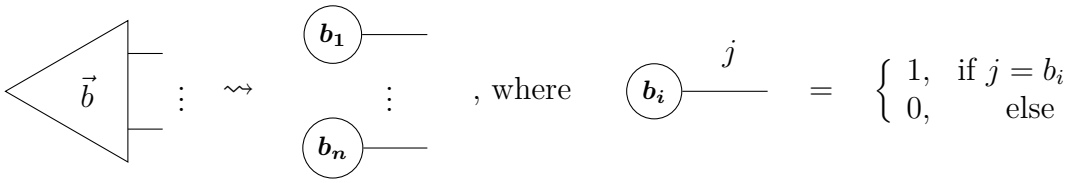
$$= \quad \begin{cases} 1, & \text{if } j = k \text{ and } l = m \oplus j \\ 0, & \text{else} \end{cases}$$

T Gate



$$= \begin{cases} e^{i\frac{\pi}{4}}, & \text{if } j = k = 1 \\ 1, & \text{if } j = k = 0 \\ 0, & \text{else} \end{cases}$$

An input bitstring also has a tensor description as:

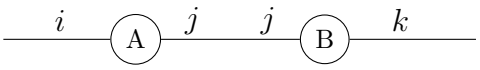


$$= \begin{cases} 1, & \text{if } j = b_i \\ 0, & \text{else} \end{cases}$$

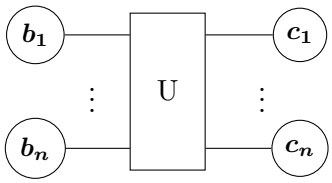
Visually, having tensors $A_{i_1,\ldots,i_n}$ and $B_{j_1,\ldots,j_m}$ next to each other corresponds to the tensor product. This is defined as the rank-$(n+m)$ tensor

$$C_{i_1,\ldots,i_n,j_1,\ldots,j_m} = A_{i_1,\ldots,i_n} \cdot B_{j_1,\ldots,j_m}.$$

Tensors can also be combined by connecting them via an edge:



This is called tensor contraction and corresponds to the tensor that results from summing over the shared system: $C_{i,k} = \sum_{j\in[n]} A_{i,j}B_{j,k}$. As this example shows, tensor contraction is a generalisation of the matrix product. It also holds that the result of contracting the vector of a pre-application state with the matrix of the applied gate is the vector of the post-application state, since it is just the vector-matrix product. An n-qubit quantum circuit can therefore be viewed as a collection of tensors, with the wires being contraction edges. This is then a *tensor network*, and the resulting rank-n tensor describes the final qubit states. The amplitude of a specific computational basis state $|\vec{c}\rangle$ then has the representation



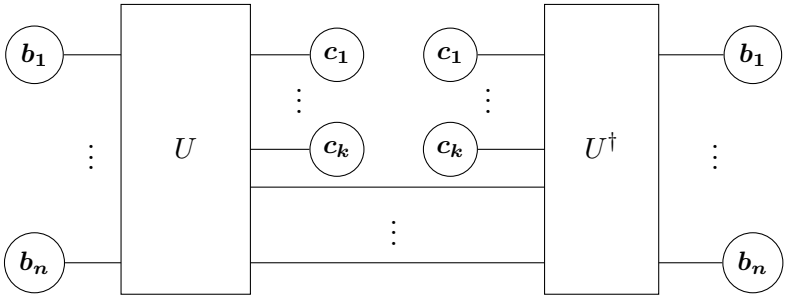
since rank-0 tensors are just complex scalars. To compute the value of this scalar, one needs to compute the intermediate tensors obtained from summing two adjacent tensors over their shared index. This step is called edge contraction, and when done in a sophisticated order, large intermediate tensors and therefore high memory usage

can be avoided. The minimal rank of the largest intermediate tensor over all orderings is determined by the treewidth of the network [24], while storing it takes exponential space. Therefore, the runtime and memory consumption of this technique scale exponentially in this parameter.

To get a marginal probability such as

$$\mathcal{P}(o_1 = c_1, \ldots, o_k = c_k)$$

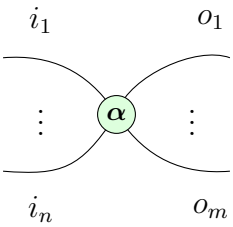
via the Born rule, one would formulate a tensor network as:



For simplicity, we will only consider the computation of amplitudes.

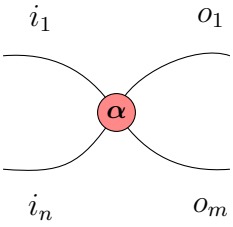
2.3.3 ZX-Calculus

ZX diagrams are a specific type of tensor network that are generated by two families of tensors, the Z-spiders



$$= \begin{cases} e^{i\alpha}, & \text{if } i_1 = \cdots = i_n = o_1 = \cdots = o_m = 1 \\ 1, & \text{if } i_1 = \cdots = i_n = o_1 = \cdots = o_m = 0 \\ 0, & \text{else} \end{cases}$$

and X-spiders

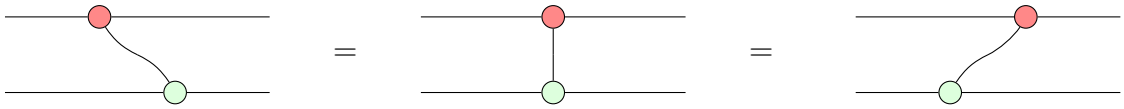


$$= \begin{cases} 2^{-\frac{n+m}{2}}(1 + e^{i\alpha}), & \text{if } i_1 \oplus \cdots \oplus i_n \oplus o_1 \oplus \cdots \oplus o_m = 0 \\ 2^{-\frac{n+m}{2}}(1 - e^{i\alpha}), & \text{else} \end{cases}$$

The $\alpha \in [0, 2\pi)$ on a spider is called its *phase*. Phases that are a multiple of $\frac{\pi}{2}$ are called Clifford, and multiples of π are called Pauli. We also call the corresponding

spider Clifford or Pauli, respectively. Furthermore, spiders with a phase of $\frac{\pi}{4}$ are called T-spiders, and those that have a phase that is an odd multiple of $\frac{\pi}{4}$ are T-like.

The Z- and X-spiders satisfy a useful property called *only-connectivity-matters* (OCM), meaning that there is no distinction between the different legs of a spider. This makes the diagrams very flexible:

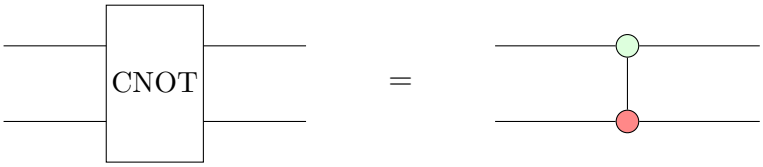


All our basic gates can be written in terms of these spiders

Hadamard Gate

Phase Gate

Controlled-Not Gate



***T* Gate**

We will use the syntactic sugar

called a Hadamard edge.

It follows that a ZX diagram that comes from translating a Clifford+T circuit has only T-like and Clifford spiders. Each diagram also carries a complex scalar that is interpreted as a multiplicative coefficient of the tensor corresponding to the diagram. This way we can also represent the basis states in a simple way:

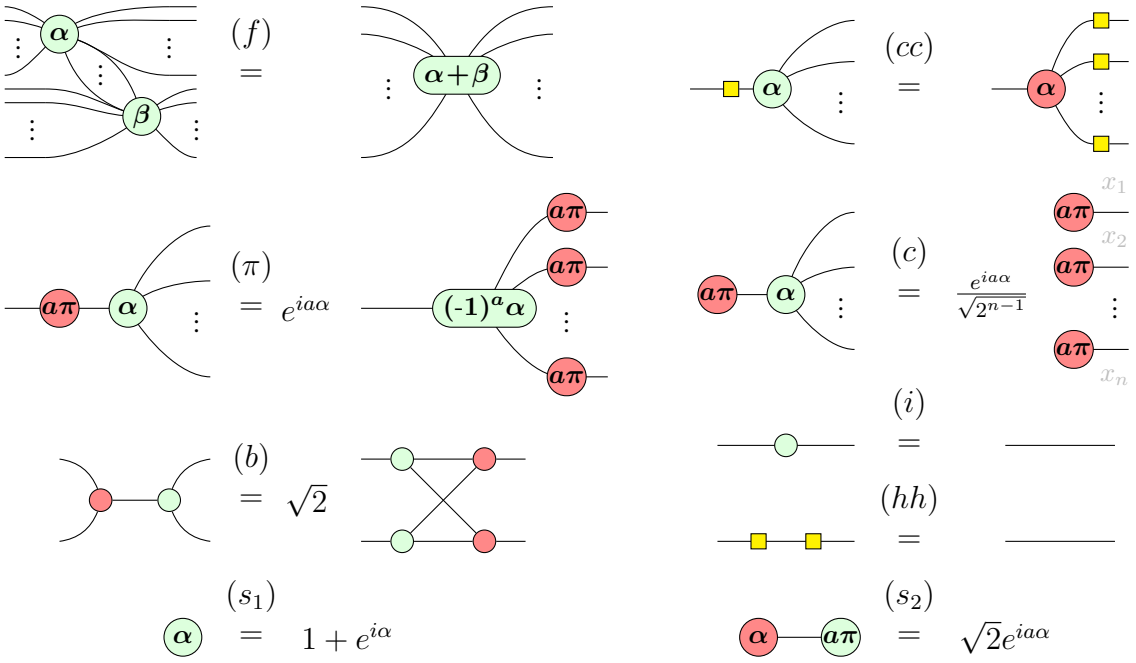
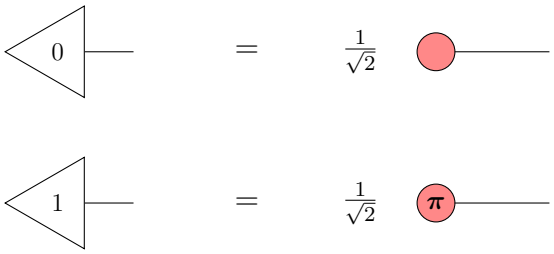


Figure 2.2: The standard rules of the ZX-calculus as presented in [5].



The advantage of ZX diagrams is that they allow for a versatile set of rewrite rules, presented in Figure 2.2. These can be combined into powerful optimisation routines and bring ZX diagrams into normal forms [17]. An example of the latter is the graph-like form, where the diagram only consists of Z-spiders and all edges are simple Hadamard edges. A popular optimisation routine is the `full-reduce` algorithm [15], which rewrites graph-like diagrams to reduce the number of non-Clifford spiders and runs in polynomial time. This algorithm also simplifies the Clifford parts of a diagram, so that all-Clifford diagrams without open edges are reduced to scalars.

2.3.4 Stabiliser Decomposition

Quantum circuits consisting of only H, S, and CNOT gates are called stabiliser or Clifford circuits. The ZX diagram corresponding to such a circuit has only Clifford phases and is therefore also called a Clifford diagram. This class of circuits is special

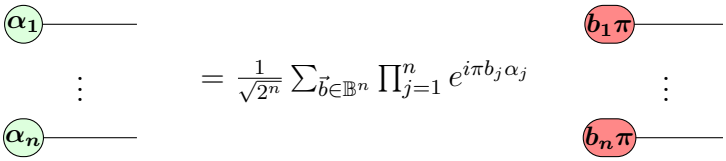
because it allows for polynomial-time simulation, for example, via the aforementioned `full-reduce` algorithm [15]. Recall that the full Clifford+T gate set is universal; therefore, the T-gate is the source of quantum computational advantage and, for simulation, the source of complexity. For general Clifford+T diagrams, the *T-count* is defined as the number of non-Clifford phases, which corresponds to the number of T-like spiders. When naively translating a Clifford+T circuit into a ZX diagram, the T-count of the diagram is exactly the number of T-gates in the original circuit.

We can leverage the efficient simulation of Clifford diagrams to simulate arbitrary Clifford+T circuits by using the concept of a stabiliser decomposition.

Definition 1. A *decomposition* of a ZX diagram D is a set S of ZX diagrams that satisfy $\sum_{s\in S} s = D$. If all elements of S are Clifford diagrams, we call it a *stabiliser decomposition*. We then also refer to the tuple (D, S) as a stabiliser decomposition.

The simplest stabiliser decomposition is the so-called Vertex-Cut (VC):

This decomposition can be generalised to



We also have the following notion

Definition 2. An application of a decomposition (D, S) to a ZX diagram E is a vertex mapping $\psi : D \to E$ such that the subgraph E' of E induced by $\psi(D)$ is isomorphic to D. This gives a decomposition E_ψ, by substituting E' with the decomposition of D:

$$E_\psi = \{E[E'/s] | s \in S\}.$$

If all non-Clifford spiders of E are contained in $\psi(D)$ and (D, S) is a stabiliser decomposition then this is also a stabiliser decomposition.

Using this, we can compute a stabiliser decomposition of any ZX diagram by unfusing[3] all the non-Clifford phases and applying the generalised Vertex-Cut to these. This can be formulated as Algorithm 1.

[3]Unfusing a spider involves applying the (f) rule from Figure 2.2 in reverse to create a new single-legged spider with a specific phase. This phase is subtracted from the original spider's phase.

Algorithm 1 Basic Stabiliser Decomposition

Input: ZX diagram D
Output: The complex scalar c corresponding to D

$T_{new} \leftarrow \emptyset$
for T-like spider s in D **do** ▷ Step 1: Unfuse T-spiders
    Unfuse a T-spider s_{new} from s.
    Add s_{new} to T_{new}.
end for
$P_{decomp} \leftarrow \text{GeneralisedVertexCut}(D, T_{new})$ ▷ Step 2: Apply decomposition
$c_{total} \leftarrow 0$
for Clifford diagram d_i in P_{decomp} **do** ▷ Step 3: Reduce and compute scalars
    $c_i \leftarrow \texttt{full\_reduce}(d_i)$
    $c_{total} \leftarrow c_{total} + c_i$ ▷ Step 4: Sum scalars
end for
return c_{total}

If the input diagram has a T-count of t, this algorithm must evaluate 2^t Clifford diagrams. This performance can be improved by using more sophisticated stabiliser decompositions for the collection of T-spiders created in the first step. For instance, Bravyi, Smith, and Smolin discovered a stabiliser decomposition of size 7 for 6 T-spiders [8], and Qassim et al. proposed a family of decompositions with a size of $O(2^{0.3963n})$ [33], where n is the number of T-gates, leading to a significant improvement in the algorithm's runtime. This motivates the introduction of the efficiency parameter, α.

Definition 3. For a stabiliser decomposition (D, S) the efficiency is defined as

$$\alpha = \frac{\log_2(|S|)}{t},$$

where t is the T-count of D.

The naive and generalised Vertex-Cut has an efficiency of $\alpha = 1$, while the BSS decomposition has $\alpha \approx 0.468$, and the decompositions by Qassim et al. approach $\alpha \approx 0.3963$ as $n \to \infty$. In general, recursively applying a stabiliser decomposition with efficiency α to a diagram with T-count t is theoretically guaranteed to yield a final stabiliser decomposition with approximately $2^{\alpha t}$ terms.

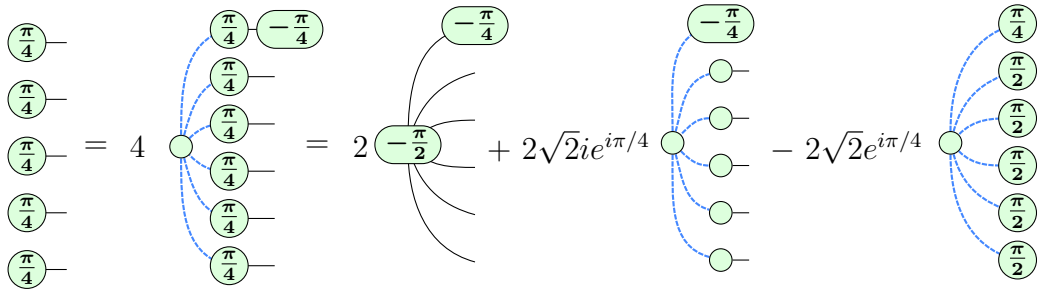
Kissinger and van de Wetering further enhanced the algorithm by introducing intermediate simplification steps [16], see Algorithm 2. This significantly reduces the size of the resulting stabiliser decomposition and therefore the algorithm's runtime. Moreover, the early application of `full-reduce` ensures the resulting Clifford diagrams are already simplified, which avoids redundant work.

Algorithm 2 Recursive BSS Decomposition with Intermediate Optimisation

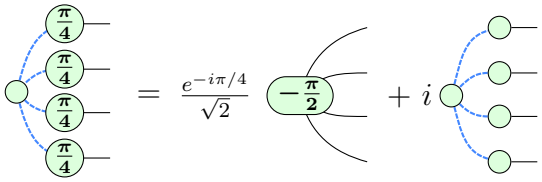
Input: ZX diagram D
Output: The complex scalar c corresponding to D

$D' \leftarrow$ `full_reduce`(D) ▷ Step 1: Simplify the diagram
if T-count$(D') < 6$ **then** ▷ Step 2: Base case for low T-count
    return BruteForceScalar(D')
end if
$T_{BSS} \leftarrow \emptyset$
for six random T-like spiders s in D' **do** ▷ Step 3: Unfuse T-spiders
    Unfuse a T-spider s_{new} from s.
    Add s_{new} to T_{BSS}.
end for
$P_{decomp} \leftarrow$ BSSDecomposition(D', T_{BSS}) ▷ Step 4: Apply BSS
$c_{total} \leftarrow 0$
for diagram d_i in P_{decomp} **do** ▷ Step 5: Recurse and sum results
    $c_i \leftarrow$ recurse(d_i)
    $c_{total} \leftarrow c_{total} + c_i$
end for
return c_{total}

In [19], Kissinger et al. proposed using decompositions that do not immediately result in Clifford diagrams but significantly reduce the T-count. Examples include the Magic5 (M5) decomposition:



and the Cat decompositions, such as Cat4 (with similar versions for Cat3, Cat5, and Cat6 [19]):



For decompositions where all resulting diagrams have the same T-count t', the definition of efficiency can be generalised to:

$$\alpha = \frac{\log_2(|S|)}{t - t'},$$

where t is the original T-count. The Magic5 decomposition has $\alpha \approx 0.396$, which matches the asymptotic efficiency of the decompositions from [7]. Although the Cat4 decomposition has a lower efficiency of $\alpha = 0.25$, its applicability depends on the diagram's structure. This leads to the updated Algorithm 3[4].

Algorithm 3 Recursive Decomposition with Partial Base Decompositions

Input: ZX diagram D
Output: The complex scalar c corresponding to D

$D' \leftarrow$ `full_reduce`(D) ▷ Step 1: Simplify the diagram
if T-count$(D') < 5$ **then** ▷ Step 2: Base case for low T-count
    return BruteForceScalar(D')
end if
$A \leftarrow$ Find all applications of {Magic5, Cat3, Cat4, Cat5, Cat6}.
$\psi_{best} \leftarrow$ Select application from A with the best guaranteed α. ▷ Step 3: Select best application
$C_{set} \leftarrow$ ApplyDecomposition(D', ψ_{best})
$c_{total} \leftarrow 0$
for diagram d_i in C_{set} **do** ▷ Step 4: Recurse and sum results
    $c_i \leftarrow$ recurse(d_i)
    $c_{total} \leftarrow c_{total} + c_i$
end for
return c_{total}

Recent work has focused on finding smaller decompositions by considering larger gadgets [9], treating disconnected components independently [9, 38], and developing heuristics for simple gadget decompositions that yield greater simplifications after the T-count reduction step [39, 4]. The latter approach is based on the observation that applying the same decomposition to different parts of a diagram can lead to varying degrees of simplification by the subsequent `full-reduce` step.

2.3.5 Graph Cuts

As mentioned before, the work of [9] and [38] improved the stabiliser decomposition algorithm by leveraging the behaviour of disconnected components in diagrams. They noticed that if a scalar ZX diagram D can be partitioned into two disconnected components D_1 and D_2, the corresponding scalar of $c \in \mathbb{C}$ of D can be computed as

$$c = c_1 \cdot c_2,$$

[4]We omit the explicit unfusion of spiders from now on.

where c_1 (resp. c_2) is the scalar of D_1 (resp. D_2). The set $\{D_1, D_2\}$ can be seen as a generalised decomposition of D where the scalars of the parts are aggregated via multiplication rather than addition. This decomposition is generally very efficient and therefore always preferred if applicable.

2.3.6 Circuit Classes

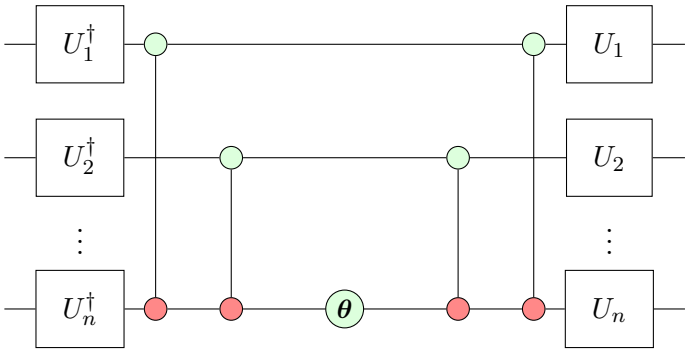
While the Clifford+T gate set is universal, the study of quantum computing often focuses on specific classes of circuits that possess particular structures. We train and evaluate our models on different circuit classes to assess how well they capture the structure of a given class or generalise to an unseen one. This thesis considers several such classes, chosen for their relevance to practical implementation on quantum devices and, consequently, their importance for quantum simulation. Moreover, these classes have been used as benchmarks in previous comparisons of stabiliser decomposition techniques [5, 9, 19].

Exponentiated Pauli Circuits

The three Pauli matrices are

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

A tensor product of Pauli matrices, $\vec{P} = P_1 \otimes \cdots \otimes P_n$ where $P_i \in \{I, X, Y, Z\}$, is called a Pauli string. For any angle θ, the operator $e^{i\theta\vec{P}}$ is a Pauli exponential. If a Pauli string does not contain the identity matrix, its corresponding Pauli exponential can be realised by the circuit:

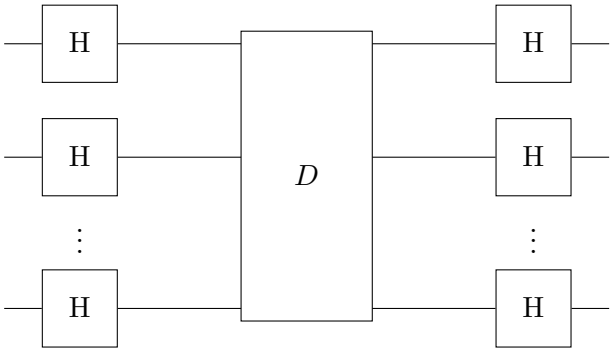


where $U_i = I$ if $P_i = X$, $U_i = H$ if $P_i = Z$, and $U_i = HSH$ if $P_i = Y$. For general Pauli strings, the circuit is applied only to the subset of qubits where P_i is not the identity; the size of this subset is the weight of the Pauli string. Pauli exponentials form a universal gate set and arise naturally in the simulation of certain quantum

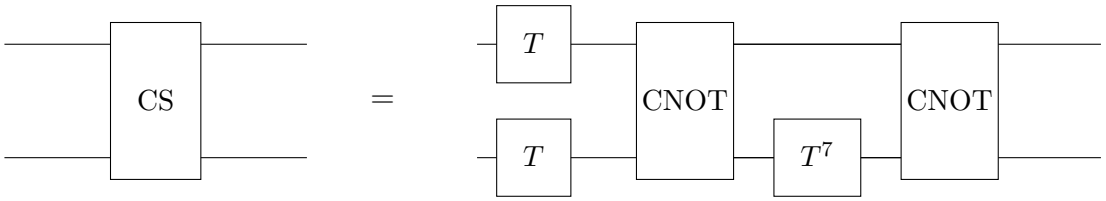
systems and Hamiltonians, as well as in some error correction codes, making them an important target for classical simulation [17]. We generate random exponentiated Pauli circuits by sampling a Pauli string of weight two to four and an angle θ that is an odd multiple of $\frac{\pi}{4}$.

Instantaneous Quantum Polynomial Circuits (IQP)

IQP circuits are a restricted class of circuits believed to be hard to simulate classically, yet feasible to implement on near-term quantum devices [27]. They are defined by the form:



where D is a diagonal operator in the computational basis. We can generate IQP circuits using only Clifford+T gates by constructing D in two layers. The first layer consists of zero to seven T-gates applied randomly to each qubit. The second layer contains zero to three Controlled-Phase (CS) gates between each pair of qubits. A CS gate can be constructed from CNOT and Hadamard gates as follows:



and corresponds to the unitary operator

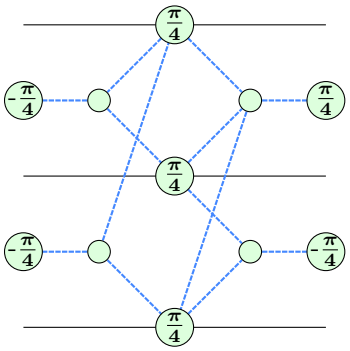
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{pmatrix}.$$

Clifford+T+CCZ

Finally, we consider Clifford+T+CCZ circuits, which are composed of random applications of gates from the Clifford+T set and the CCZ gate. The CCZ gate corresponds to the unitary operator:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}.$$

and is built in the ZX-calculus as:



We generate these circuits by randomly sampling and applying gates to random qubits according to the following distribution:

- T: 5%
- CCZ: 5%
- H: 22.5%
- Phase: 22.5%
- Controlled-Not: 22.5%
- Controlled-Z (CS^2): 22.5%

This set is considered because it was used in [21], and the heuristics from [4] demonstrated good performance on this class of circuits.

2.4 Machine Learning

Having defined the problem of finding efficient applications of decompositions, we now turn to the machine learning tools used to solve it. As opposed to a purely analytical approach, machine learning employs statistical algorithms that learn from data to

find solutions. This section introduces the relevant machine learning domains for this thesis, such as supervised and reinforcement learning, and covers the fundamentals of working with graph-structured data. For a thorough introduction, the reader is referred to [26].

2.4.1 Supervised Learning

Supervised learning is a machine learning paradigm where an algorithm is trained to map inputs to outputs using a dataset of input-output pairs, also known as labelled data. While this is, in principle, the most straightforward approach used in this work, its primary difficulty often lies in generating a high-quality training dataset. Formally: the learning problem is to approximate an unknown target function $f : X \to Y$, given a training dataset $D = \{(x_1, y_1), \ldots, (x_N, y_N)\}$, which consists of N example pairs where each $x_i \in X$ is an input object and $y_i \in Y$ is its corresponding label. The labels satisfy $y_i = f(x_i) + \varepsilon_i$, where the ε_i describe the inherent *noise* of the training data. When this noise is large in comparison to the *training signal* $f(x_i)$ then even the best training methods cannot learn a good approximation for f.

The process begins by defining a *hypothesis space* $\mathcal{H}$, which is the set of all candidate functions $h : X \to Y$. The objective is to select a hypothesis $h \in \mathcal{H}$ that best approximates the true function f. A canonical example is the classification of handwritten digits, where the training set contains pairs of images and their human-assigned numerical labels, and the goal is to select a function that correctly labels unseen images.

To evaluate a given hypothesis, a *loss function* $l : Y \times Y \to \mathbb{R}$ is defined, which quantifies the error between a predicted output $h(x_i)$ and the true output y_i for a single example. The overall performance of a hypothesis h on the training dataset is typically measured by the sum or mean of the individual losses:

$$L(h) = \sum_{i=1}^{N} l(h(x_i), y_i) \tag{2.1}$$

Since the hypothesis space $\mathcal{H}$ is typically vast and continuous, exhaustively searching for the optimal candidate is computationally intractable. Furthermore, finding a perfect global minimum on the training data is not always desirable, as it can lead to overfitting. Instead, optimisation is performed using iterative, gradient-based methods.

The hypothesis space is parameterised as a set of functions $\{h_\omega \mid \omega \in \Omega\}$, where each hypothesis is uniquely determined by a vector of parameters ω. If the loss function is

differentiable with respect to its first argument and the function $h_\omega(x)$ is differentiable with respect to the parameters ω, then the total cost $L(h_\omega)$ is also differentiable with respect to ω. The basic *gradient descent* algorithm starts with an initial parameter guess ω_0 and iteratively updates the parameters by moving in the direction opposite to the gradient of the loss function:

$$\omega_{t+1} = \omega_t - \eta \nabla_\omega L(h_{\omega_t}) \tag{2.2}$$

Here, η is the learning rate, a hyperparameter controlling the update step size. In this thesis, we utilise the PyTorch implementation of the Adam optimiser, a sophisticated variant of this method [13]. We also use stochastic gradient descent, where the gradient is computed on small subsets of the training set in each round.

2.4.2 Reinforcement Learning

In reinforcement learning, data is not sourced from a predefined training set but is generated through iterative interaction with an environment. An agent receives states and responds with actions, which in turn alter the environment's state and yield a reward or punishment. This reinforcement signal is then used to update the agent's policy. This framework has been successfully applied to various scientific fields, such as robotics [20], chemistry [37] and also to problems related to stabiliser decompositions, such as circuit optimisation [25, 30, 34].

Formally: the problem has to be structured as a (deterministic) Markov Decision Process (MDP), which is a tuple $(\mathcal{S}, \mathcal{A}, T, R, p_0, \gamma)$ [32]. We have a set of states $\mathcal{S}$, and for each state $s \in \mathcal{S}$ an action space $\mathcal{A}(s)$, a transition function $T : \mathcal{S} \times \mathcal{A}(s) \to \mathcal{S}$ and an immediate reward $R : \mathcal{S} \times \mathcal{A}(s) \to \mathbb{R}$. Furthermore, p_0 is an initial distribution over the states and γ is a discount factor that weighs future reward against the immediate reward. The agent's policy is described by a probability distribution $\pi_\omega(a, s)$.

To optimise the policy, many episodes are rolled out. During an episode, the initial state distribution and the policy are sampled to generate a sequence of states and actions, which is terminated either by the environment or after a fixed length. After many such episodes, an optimisation routine such as Proximal Policy Optimisation [36] or Q-learning is used to adjust the model's parameters ω to maximise the expected rewards. We will further explain Proximal Policy Optimisation in Chapter 5 and refer to [41] for a full introduction to reinforcement learning.

2.4.3 Neural Networks

In both supervised and reinforcement learning, the core task is to optimise a parameterised function, such as a hypothesis h_ω or a policy π_ω. Neural networks offer a powerful framework for representing these functions, capable of approximating highly complex, non-linear relationships. A common type is the multi-layer perceptron (MLP), organised in layers of neurons. The input is a real-valued vector representing the activation of the first layer. Subsequent neurons compute their activation by taking a weighted sum of the activations from the preceding layer, adding a bias, and applying a non-linear activation function. After this *feed-forward* process, the activation of the last layer is taken as the output of the network. The learnable parameters are the weights and biases, and their gradients can be calculated efficiently using the backpropagation algorithm. Networks with many layers form the basis of Deep Learning [23].

2.4.4 Graph Neural Networks

Graph neural networks (GNNs) are the core of the models developed in this work. They are machine learning models designed for graphs and are inherently invariant to node permutations, which promotes good generalisation. Given a set of input node features, they compute output features via a process called message-passing. In this process, intermediate features are computed over multiple layers according to the formula:

$$h_u^{t+1} = \textbf{update}\left(h_u^t, \bigoplus_{v \in N_u} \textbf{message}(h_u^t, h_v^t)\right)$$

where h_u^t is the feature of node u at layer t, N_u is the neighbourhood of u, **message** and **update** are differentiable functions, and $\oplus$ is a permutation-invariant aggregation operator such as sum, mean, max or an attention mechanism.

In this work, we use the following well-established GNN architectures:

Graph Convolutional Network [14]

GCNs update the node representations by aggregating their neighboring node features, normalising them, applying a linear transformation, and finally applying some non-linear activation function. Formally

- $\textbf{message}(h_u^t, h_v^t) = \frac{1}{\sqrt{\deg(u)\cdot\deg(v)}} W^t \cdot h_v^t$, where W is a matrix with learnable entries

- $\oplus$ is a simple sum
- **update** treats the current feature as a neighbor and applies some activation function σ such as ReLU to the aggregated node features

Combined:

$$h_u^{t+1} = \sigma\left(W^t \cdot \sum_{v \in N_u \cup \{u\}} \frac{1}{\sqrt{\deg(u) \cdot \deg(v)}} h_v^t\right).$$

Graph Attention Network (GAT) [42]

GATs are very similar to GCNs, with the distinction that the neighboring node features are not normalised, but weighted via an attention mechanism:

$$h_u^{t+1} = \sigma\left(\sum_{v \in N_u \cup \{u\}} \alpha_{uv}^t W^t \cdot h_v^t\right),$$

where

$$\alpha_{uv} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T[W^t h_u^t || W^t h_v^t]))}{\sum_{k \in N_u \cup \{u\}} \exp(\text{LeakyReLU}(\mathbf{a}^T[W^t h_u^t || W^t h_k^t]))}.$$

Usually, one uses multiple attention heads $\mathbf{a}$ and concatenates the results.

Graph Isomorphism Network (GIN) [43]

GINs are designed to be very expressive in terms of structural information. They aggregate information by summing its own feature with those of its neighbours while applying a learnable bias to its own feature. They then update by feeding the aggregated information into a multi-layer perceptron. Formally

$$h_u^{t+1} = \mathbf{MLP}\left((1 + \varepsilon^t) \cdot h_u^t + \sum_{v \in N_u} h_v^t\right).$$

2.4.5 GNNs on ZX diagrams

We apply GNNs to graph-like ZX diagrams to learn efficient choices for partial stabiliser decompositions. The task is simplified because we only need to consider graph-like diagrams where phases are multiples of $\frac{\pi}{4}$. We treat spiders as nodes and encode their phases as an 8-dimensional one-hot vector for the input features. Information about spider type (all are Z-spiders) and edge type (all are Hadamard edges) is omitted as it is uniform. This encoding is consistent with the work of [22].

For this work, we further enrich the input feature vector by adding random features [2], features of biconnected components [46], and subgraph homomorphism counts of rings [6]. Depending on the approach, the GNN then generates node embeddings or graph-level features used to rank nodes or make other decisions.

Chapter 3

Problem Definition

To understand where machine learning can be applied, we recall the general recursive Algorithm 4 for obtaining a stabiliser decomposition of a given ZX diagram using known base decompositions and graph cuts. Step four, which we term the *selection*

Algorithm 4 General Recursive Stabiliser Decomposition

Input: ZX diagram D
Output: The complex scalar c corresponding to D

$D' \leftarrow$ `full_reduce`(D) ▷ Step 1: Simplify the diagram
if D' is empty **then** ▷ Step 2: Base Case for empty diagrams
    return Scalar(D')
end if
$D_{components} \leftarrow$ ConnectedComponents(D')
if $|D_{components}| > 1$ **then** ▷ Step 3: Handle disconnected components
    $c_{total} \leftarrow 1$
    for component d_i in $D_{components}$ **do**
        $c_i \leftarrow$ recurse(d_i)
        $c_{total} \leftarrow c_{total} \times (c_i)$
    end for
    return c_{total}
end if
$\psi \leftarrow$ SelectDecompositionApplication(D') ▷ Step 4: Selection driver
$P_{decomp} \leftarrow$ ApplyDecomposition(D', ψ)
$c_{total} \leftarrow 0$
for diagram d_i in P_{decomp} **do** ▷ Step 5: Recurse and sum results
    $c_i \leftarrow$ recurse(d_i)
    $c_{total} \leftarrow c_{total} + c_i$
end for
return c_{total}

driver, is the focus of our improvement efforts. Given a starting diagram D, the applications that the driver chooses lead to a decomposition tree, where each node

represents an intermediate diagram and the corresponding application. The number of leaves in that tree corresponds to the size of the computed stabiliser decomposition, which has some efficiency α. We call this the global α that the driver achieved. We can measure the general performance of a selection driver by the expectation value of this global α over some dataset.

3.1 Local α

A greedy way to achieve a good (low) global α is by selecting applications that have a high reduction in T-count and a low number of parts. This can be measured by the local α of this application. Let D be a diagram with a given T-count t^*. After applying a decomposition and running `full-reduce`, the resulting set of diagrams, say S_i, may contain diagrams with varying T-counts. This makes the standard definition of α inapplicable. To address this, we introduce a general measure of local efficiency, α_{loc}, for a given decomposition within a larger recursive procedure. Let $t_1, \ldots, t_k$ be the T-counts of the diagrams in a decomposition of D. Assuming the overall procedure is expected to yield a stabiliser decomposition of size $2^{\alpha t^*}$ for some α, we use the approximation:

$$2^{\alpha t^*} = \sum_{i=1}^{k} 2^{\alpha t_i}. \tag{3.1}$$

We then define α_{loc} as the unique positive solution α to this equation.

Lemma 4. *For any $t^* \in \mathbb{N}$ and $t_1, \ldots, t_k < t^*$ with $k > 1$, Equation (3.1) has a unique positive solution.*

Proof. Dividing both sides by $2^{\alpha t^*}$, we see that the solutions of Equation (3.1) coincide with the roots of the function:

$$f(\alpha) = 1 - \sum_{i=1}^{k} 2^{\alpha(t_i - t^*)}.$$

We observe that $f(0) = 1 - k < 0$, since $k > 1$. Also, because $t_i - t^* < 0$, we have:

$$\begin{aligned} \lim_{\alpha \to \infty} f(\alpha) &= 1 - \sum_{i=1}^{k} \lim_{\alpha \to \infty} 2^{\alpha \overbrace{(t_i - t^*)}^{<0}} \\ &= 1 - 0 = 1. \end{aligned}$$

By the Intermediate Value Theorem, a root must exist. Furthermore, the derivative is:

$$f'(\alpha) = -\sum_{i=1}^{k} \ln(2)(t_i - t^*) 2^{\alpha(t_i - t^*)} > 0 \quad \forall \alpha \in \mathbb{R},$$

since each term $(t_i - t^*)$ is negative. Thus, f is strictly increasing, and the root is unique. □

For instances where all diagrams in the decomposition have the same T-count t', our α_{loc} matches the standard efficiency definition, $\alpha = \frac{\log_2(k)}{t^* - t'}$. This is because:

$$\begin{aligned}
\sum_{i=1}^{k} 2^{\alpha t_i} &= k \cdot 2^{\alpha t'} \\
&= 2^{\log_2(k)} \cdot 2^{\frac{\log_2(k) t'}{t^* - t'}} \\
&= 2^{\frac{\log_2(k) t'}{t^* - t'} + \log_2(k)} \\
&= 2^{\frac{\log_2(k)(t' + t^* - t')}{t^* - t'}} \\
&= 2^{\frac{\log_2(k) t^*}{t^* - t'}} \\
&= 2^{\alpha t^*}.
\end{aligned}$$

To recap, we can measure three kinds of efficiency for a selection driver: firstly, the theoretically guaranteed α of the used base decompositions. Secondly, the local α of the individual decomposition applications. Lastly, the global α of the overall decomposition.

While the drivers in [5, 39] search for structures that guarantee simplification and therefore offer a bound for the local α, the process can also be brute-forced by computing the local α for all possible applications. We call this greedy brute-force algorithm Sherlock:

Given a base set B of efficient decompositions (e.g., Vertex-Cut, Magic5, BSS, Cat decompositions), Sherlock selects the best application as follows:

1. Identify all possible applications, ψ_i, of decompositions from B to the current diagram D.

2. For each application ψ_i, use `full-reduce` to simplify the resulting diagrams in D_{ψ_i}.

3. For each resulting set of diagrams D_{ψ_i}, compute its α_{loc} and select the application that minimises this value.

Note that for an application of Magic5, the resulting α_{loc} is guaranteed to be less than or equal to the theoretical efficiency of Magic5, which is approximately 0.396. We use the name Sherlock-VC for a Sherlock variant that only uses Vertex-Cut and Sherlock-M5 for one that only uses Magic5 decompositions.

3.2 Structure of a Selection Driver

Given a set of base decompositions (e.g., Vertex-Cut, Magic5, and the Cat family), we subdivide the overall driver into multiple parts so that we can train machine learning models for these parts. We have two layers: first, individual selectors for each type of decomposition, and then estimators that evaluate the proposed selections. Figure 3.1 illustrates this process. The baseline methods used for comparison can also be understood within this framework:

1. The method of Kissinger et al. [19] selects a random T-like vertex for a Vertex-Cut, five T-like spiders for a Magic5 if possible, and a random Cat-n decomposition with priorities for n in the decreasing order of 4, 6, 5, and 3. The α's of these options are then estimated by the base decomposition's theoretical α, regardless of the specific structure of the diagram D. We refer to this selection driver as *Kissinger et al.*.

2. The final algorithm of Ahmad and Sutcliffe in [5], which we call *DynamicT*, has the same selectors and estimators for Magic5 and Cat decompositions but enhances the Vertex-Cut process. This is crucial, as the Kissinger method only applies Vertex-Cut when the T-count is below 5. DynamicT computes different heuristics for possible Vertex-Cuts[1]. These heuristics form a lower bound on the simplification achievable with `full-reduce`, which in turn provides an upper bound on the global α. The vertex with the best guarantees is then selected, and the guarantee used as the estimator. The restriction of this driver to only Vertex-Cuts is called *DynamicT-VC-only*.

3. The Sherlock driver estimates the performance of all possible applications by computing the local efficiency, α_{loc}, and selecting the application with the best value. In practice, we limit consideration to 100 possible Magic5 applications, as the action space grows too fast ($O(n^5)$, where n is the T-count of the diagram).

In the following sections, we present machine learning techniques to build models for each of the selection and estimation processes and discuss their potential as well as current shortcomings. GNN-based models promise strong pattern recognition capabilities on graph-structured data. If trained successfully, they may learn to approximate the heuristics from [5], discover novel multi-step strategies, and potentially

[1]Some of these are technically an application of the reversed strong complementarity rule followed by a Vertex-Cut, but for simplicity, we treat them as Vertex-Cuts.

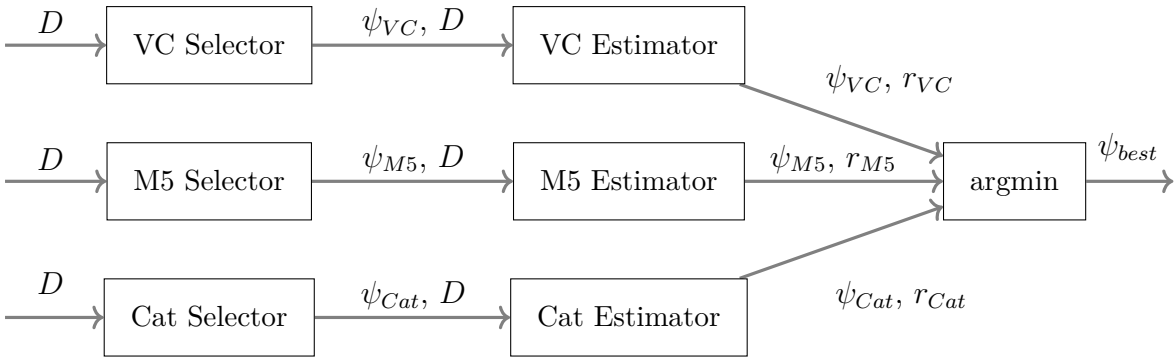


Figure 3.1: Depiction of a selection driver using the Vertex-Cut, Magic5 and Cat decompositions. The diagram D is fed into individual selectors for each decomposition. The selected application, $\psi_{...}$, and the original diagram are then evaluated by the corresponding estimators. The application with the lowest estimate is chosen as the overall output.

even outperform Sherlock. While Sherlock's brute-force approach should theoretically outperform methods such as DynamicT and Kissinger due to its use of richer information, it has significant shortcomings. Firstly, its time complexity is prohibitive, as it requires running the `full-reduce` algorithm for every possible application (e.g., $O(n^5)$ Magic5 applications), making it computationally infeasible for large diagrams. Secondly, Sherlock is a greedy algorithm, optimising for the maximum simplification in a single step. This may not always be optimal, as a decomposition yielding diagrams with higher T-counts could be more beneficial if those diagrams permit superior follow-up decompositions. An additional advantage of trained models is their ability to run on GPUs, which allows for native parallelisation.

Chapter 4

Supervised Selection Learning

This section presents our work on using supervised learning to train models for the selection driver. We construct the training data by generating random circuits from the previously discussed classes: exponentiated Pauli circuits, IQP circuits, and Clifford+T+CCZ circuits. For each diagram, we consider every possible application of our base decompositions (or a sufficiently large sample in the case of Magic5). For each application, we compute the global α that would be achieved by applying it and subsequently decomposing the results using Sherlock with the same base decomposition. This data is then used to train our models, each with a setup specific to the decomposition it handles, on the different circuit classes.

4.1 Setup for Vertex-Cut Selection

For the VC-model, we designed a GNN-based architecture that computes a relative rank for each non-Clifford spider. In practice, the spider with the highest rank is selected for the Vertex-Cut; therefore, the model is trained to produce ranks that correlate with the quality of the corresponding Vertex-Cut. The architecture consists of three stages. First, a multi-layer perceptron (MLP) encodes the input features into initial vertex embeddings. Second, a GNN processes these embeddings, aggregating structural information from the graph to produce final vertex embeddings. Third, an MLP converts these final embeddings into scalar ranks. The ranks of Clifford vertices are masked to a large negative value to prevent their selection. Figure 4.1 provides a sketch of this architecture.

To complete the setup, we define a loss function for training. For a given graph, the loss function takes the model's output ranks and the global α's for every Vertex-Cut application as input. We present the two candidates that produced the best-performing models:

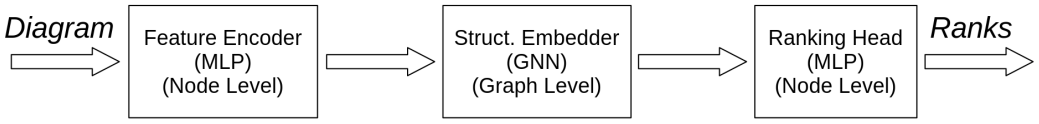


Figure 4.1: Illustration of the Vertex-Cut selection model architecture. The input diagram is encoded as described in Section 2.4.5 and then fed through multiple layers with learnable parameters.

Kullback-Leibler (KL) divergence

The KL divergence is a measure of the difference between two probability distributions and is widely used as a loss function for machine learning [28]. It is defined as

$$D_{KL}(p||q) = \sum_y p(y) \log \frac{p(y)}{q(y)},$$

where p is a target distribution and q is the model distribution. To use it in our setup, we convert both the model's output and the target values from the training data into probability distributions. The model's training objective is to minimise the KL divergence between these two distributions. The model's output ranks $\vec{r} = (r_1, \ldots, r_n)$ are transformed into a probability distribution over the non-Clifford spiders using the softmax function

$$\text{softmax}(\vec{r})_i = \frac{e^{r_i}}{\sum_{j=1}^{n} e^{r_j}}.$$

This distribution represents the model's confidence in each spider being the optimal choice. To create the target distribution, we use the negative of the global α for each possible Vertex-Cut as a score, and similarly apply the softmax function to these scores.

Expected global α (EGA)

For this loss function, we again convert the model's output ranks $\vec{r}$ into a probability distribution $\vec{p}$ using the softmax function. The loss is then calculated as the expected value of the global α, which is the sum of the global α values from the training data, each weighted by its corresponding probability from the model's output distribution:

$$l(\vec{r}, \vec{\alpha}) = \sum_{i=1}^{n} \alpha_i p_i$$

This loss directly represents the expected global α one would achieve by sampling a Vertex-Cut from the model's predicted distribution.

4.2 Setup for Magic5 Selection

Selecting a good Magic5 application involves choosing five distinct non-Clifford spiders, making the model architecture and training procedure more complex than for Vertex-Cutting. We propose two models: a simple and an advanced version.

The simple model uses the same GNN architecture as the VC-model to assign a rank to each spider. In practice, the five spiders with the highest ranks are selected for the Magic5 application. During training, a target score is calculated for each spider by taking the negative of the average global α's over all sampled Magic5 applications in which that spider participated. If a spider was not part of any sampled application, its target score is set to -0.396 (the theoretical efficiency of Magic5), while Clifford spiders are assigned a large negative value. The loss function is the KL divergence between the softmax distributions of the model's predicted ranks and the target scores.

The advanced model employs a sequential selection process, selecting the five spiders one by one. It is composed of five copies of the simple model's architecture, which operate sequentially. During inference, the first copy computes ranks for all spiders, and the one with the highest rank is selected. This choice is encoded as a one-hot vector and added as a new node feature to the graph. The second copy then takes this augmented graph as input and selects the next spider. This process is repeated three more times, with the ranks of already-selected spiders being masked out in subsequent steps, until five spiders are chosen for the Magic5 application. Using this architecture the model can learn to base the selection of a spider on the previous selections.

Each of the five model components is trained with an adapted version of the simple model's procedure. To compute the loss for the i-th component, we first randomly sample a valid partial selection S of $i-1$ spiders. This selection is added to the graph as a one-hot encoded feature. The target score for any remaining spider is then calculated by averaging the true global α values only over those Magic5 applications that contain both the spider in question and all spiders in S. As before, if no such applications exist in the data, the target score is set to -0.396.

Details on the implementation, the training process, and hyperparameters for all setups can be found in the appendix.

	Exponentiated Pauli	Clifford+T+CCZ
#Qubits	15	15
Depth	6-50	10-200
Weight	2-4	-

Table 4.1: Settings used for generating the test sets of the different circuit classes, if applicable.

4.3 Iterated Training

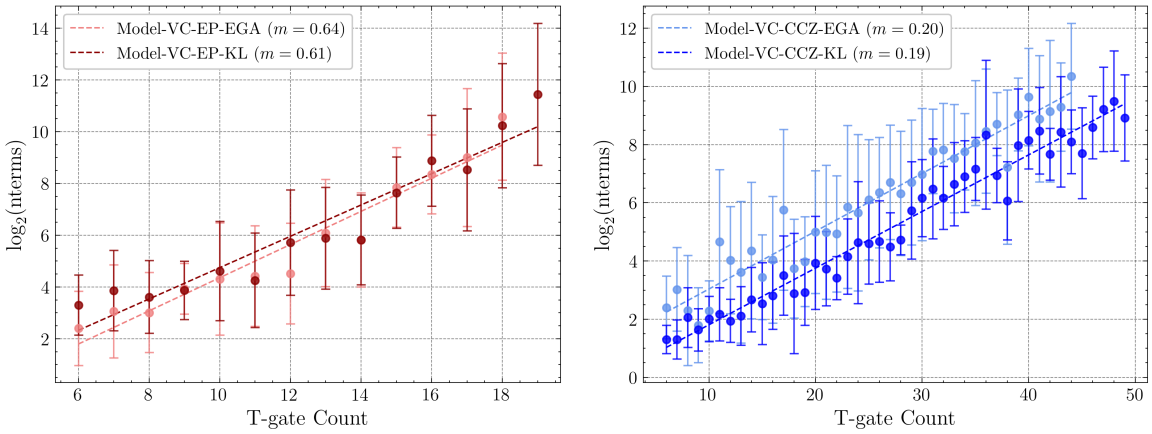
The training procedure described above teaches the models to select decompositions that perform well under the assumption that subsequent steps will be guided by Sherlock's decision mechanism. To mitigate this inherent bias, we introduce an iterated training scheme. After an initial training phase, we generate a new training dataset. In this new dataset, the follow-up decomposition choices are made by the trained models themselves, rather than by Sherlock. The models are then further trained on this new data, and this entire process can be repeated multiple times.

4.4 Results

We evaluate our models' performance using the benchmarking capabilities of QuiZX. The test set is constructed by generating random circuits from the specified circuit classes, using the parameters detailed in Table 4.1. These circuits are simplified using `full-reduce` until we obtain 10 samples for each T-count ranging from 6 to 30. We then run the stabiliser decomposer on each graph once for each of the selection drivers. Any decomposition that takes longer than 120 seconds is terminated. The performances are presented as box plots, with the T-count on the x-axis and the base-2 logarithm of the resulting stabiliser decomposition size on the y-axis. A linear fit is also added to the data points of a given driver to indicate the trend and the slope m of this fit corresponds to the expected global α of the driver on this dataset.

Experiment 1: Loss Function Comparison

First, we want to compare the two loss functions used for training the Vertex-Cut selection models. In Figure 4.2, the performance of the corresponding models on unseen graphs of the same circuit class is shown. We see that while the difference on exponentiated Pauli circuits is not significant, on Clifford+T+CCZ circuits the KL-based model performs better than the EGA-based one. Therefore, we continue only with the KL-based models.



(a) On Exponentiated Pauli Circuits

(b) On Clifford+T+CCZ Circuits

Figure 4.2: Comparison of the trained Vertex-Cut models with KL- or EGA-based loss.

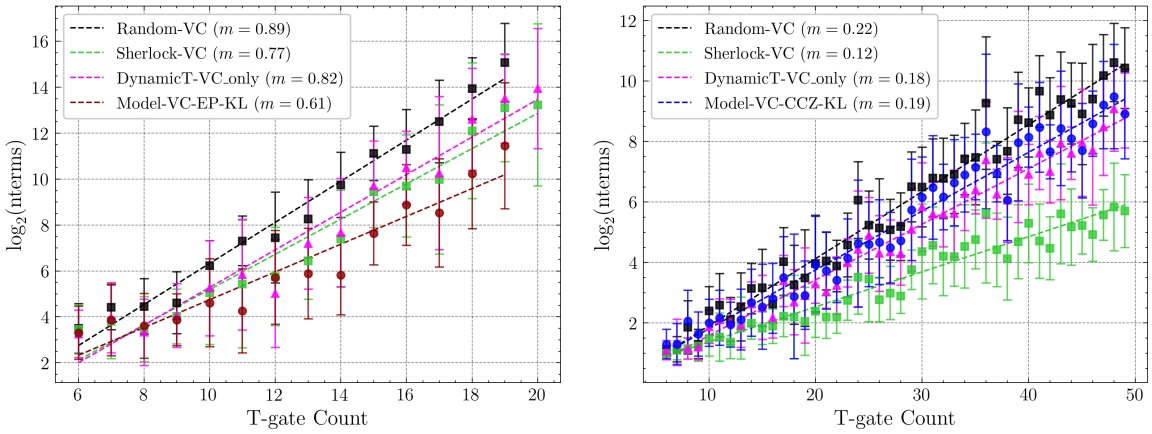
Experiment 2: Vertex-Cut Performance

Next, we want to compare the Vertex-Cut selection models to the other heuristics for Vertex-Cut selection. Figure 4.3 presents the results of this comparison. The results show that the Vertex-Cut model trained on exponentiated Pauli circuits with KL-divergence loss outperforms both the DynamicT heuristic from [5] and the Sherlock driver. This outcome is particularly promising, as it indicates the model learnt not only patterns for immediate simplification but also more complex strategies that provide advantages in subsequent decomposition steps. This result exceeds our expectations for this simple training setup.

In contrast, the results also show that the model trained on Clifford+T+CCZ circuits only barely outperformed random selection on that circuit class.

Experiment 3: Vertex-Cut Generalisation

Following up on the promising performance of the model trained on exponentiated Pauli circuits, we investigate whether the patterns that this model learnt can also be applied on Clifford+T+CCZ circuits. We therefore took the best-performing Vertex-Cut models from EP and Clifford+T+CCZ circuits and compared them on those classes, as well as IQP circuits. The results in Figure 4.4 show that the models only barely outperform random selection on unseen circuit classes and always underperform the model trained on this class. Furthermore, both underperform the DynamicT heuristic on IQP circuits.



(a) On Exponentiated Pauli Circuits

(b) On Clifford+T+CCZ Circuits

Figure 4.3: Comparison of the trained Vertex-Cut models and the baselines Sherlock-VC (greedy brute force), DynamicT-VC-only [4], and a random selector.

This highlights an expected challenge in generalising to unseen circuit types. Furthermore, a model trained on a joint dataset of all classes performed worse on the individual test sets than the specialised models, suggesting that the training signals from different circuit classes may be conflicting.

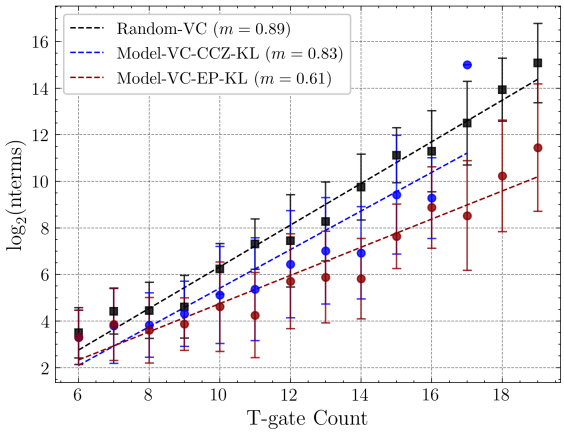
Experiment 4: Vertex-Cut Iteration

As described in Section 4.3, we tried to improve the models via an iterated data generation process to reduce the bias in the training data. However, as shown in Figure 4.5, the model's performance did not significantly change in this process.

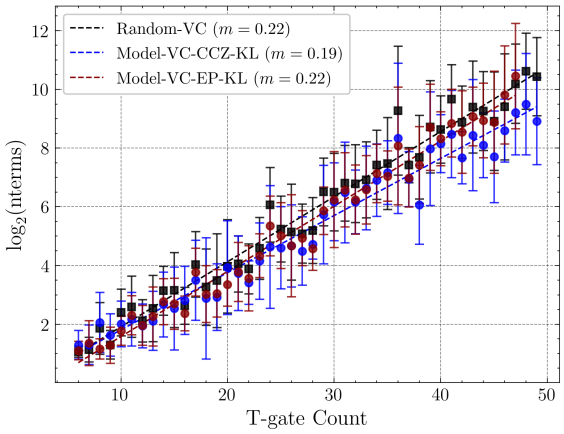
Experiment 5: Magic5

For the Magic5 decomposition, the presented supervised learning techniques did not produce effective models, as can be seen in Figure 4.6. We trained the models on exponentiated Pauli circuits and evaluated on unseen circuits of the same class. When restricting the training data to only a few diagrams and evaluating the model on those graphs, the advanced model showed the capability to memorise the best applications. However, this knowledge did not generalise to the whole circuit class. This indicates a weak, or even conflicting, training signal that is largely indistinguishable from the inherent noise in the data.

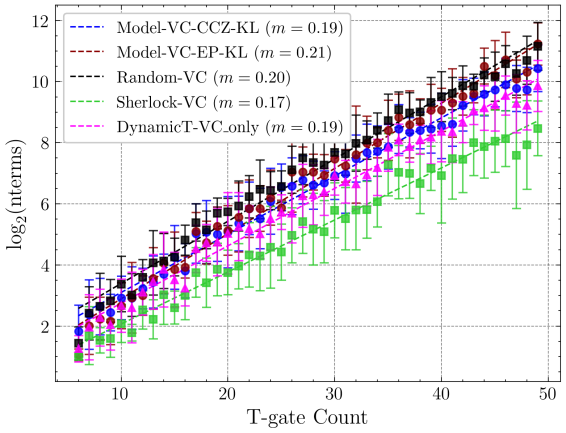
This poor performance is likely attributable to the highly complex nature of the Magic5 action space. This conclusion is supported by the observation that similar



(a) On Exponentiated Pauli Circuits



(b) On Clifford+T+CCZ Circuits



(c) On IQP Circuits

Figure 4.4: Comparison of the trained Vertex-Cut models across circuit classes

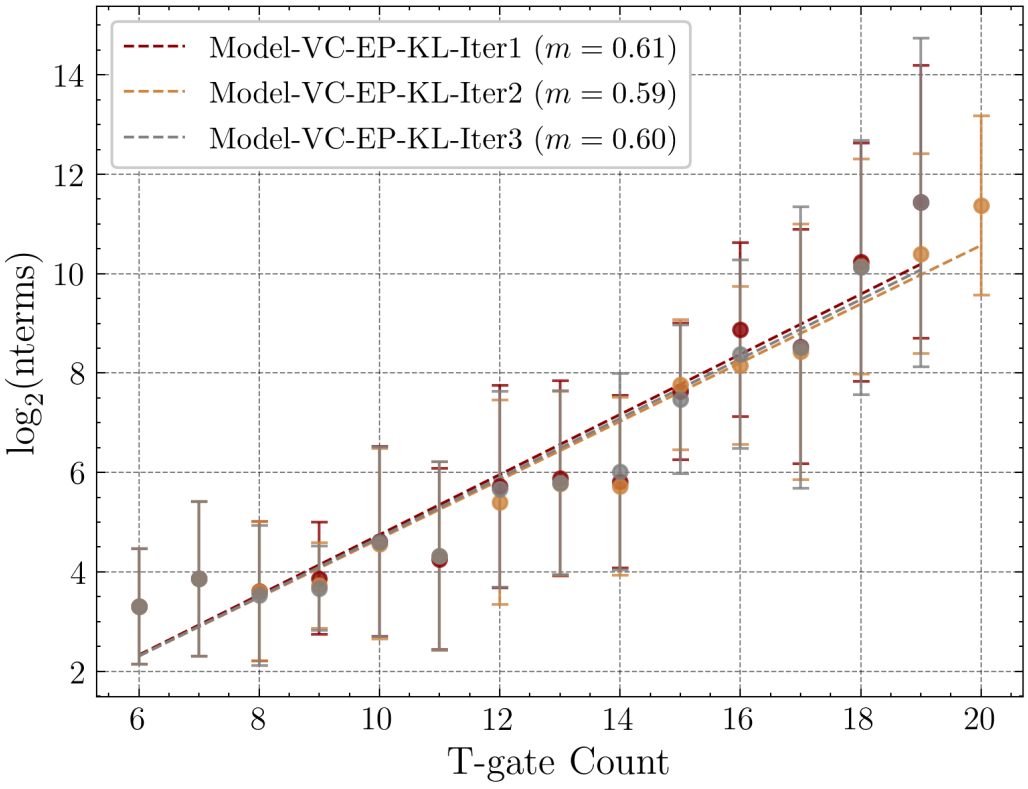


Figure 4.5: Comparison of the performance of iterations of the Vertex-Cut model trained on EP circuits with KL loss.

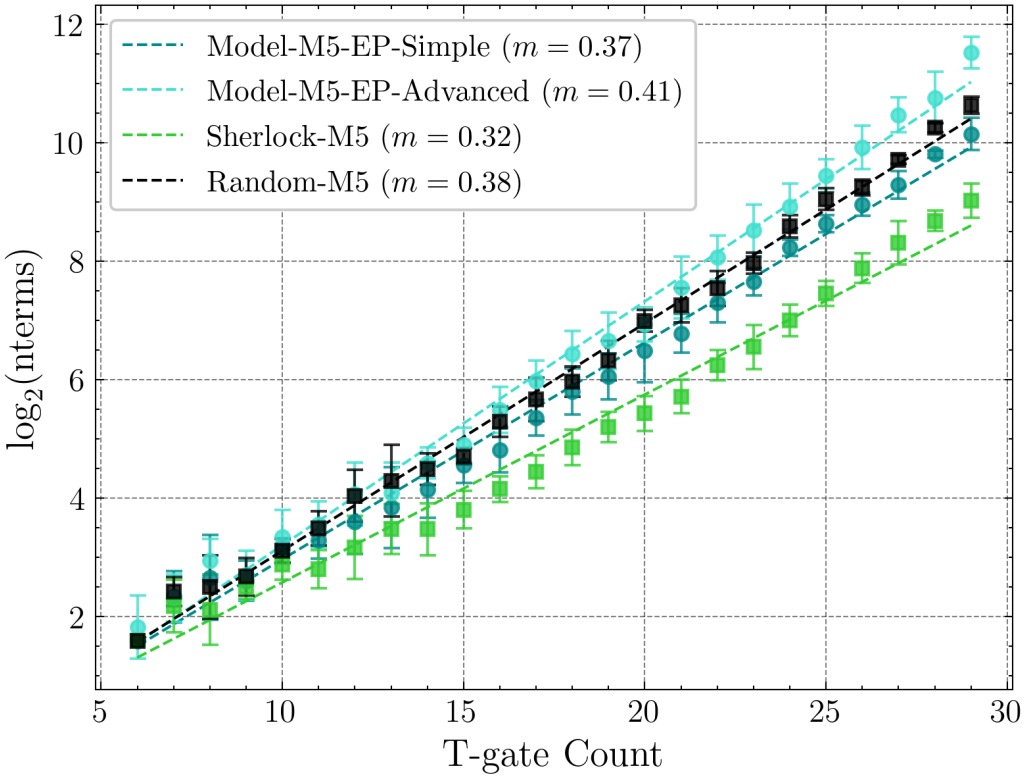


Figure 4.6: Comparison of the trained M5 models with Sherlock and a Random driver using only M5 on exponentiated Pauli circuits. The models were trained on this class of circuits.

negative results were obtained across a range of model architectures, loss functions, and circuit classes.

Chapter 5

Reinforcement Learning

This section presents work on using reinforcement learning (RL) to train models for the selection driver. In the RL paradigm, models learn by interacting with an environment and receiving feedback via a reward system. This approach has the advantage of not requiring a pre-generated dataset of approximately optimal selections, thereby avoiding the biases introduced by the data generation process, which we considered in Section 4.3.

In the following, we give a formal definition of the training environment used in [22] to train a Magic5 selection model and discuss adaptations intended to improve performance.

5.1 Reinforcement Learning Environment

As mentioned in Section 2.4.2, the problem must be formulated as a Markov Decision Process (MDP), defined by the tuple $(\mathcal{S}, \mathcal{A}, T, R, p_0, \gamma)$. We formalise the ideas from [22]: The state space $\mathcal{S}$ consists of graph-like ZX diagrams. For a given state (diagram) $D \in \mathcal{S}$, the action space $\mathcal{A}(D)$ comprises all possible applications φ of Magic5. The transition function $T : \mathcal{S} \times \mathcal{A}(s) \to \mathbb{P}(\mathcal{S})$ is defined by applying the chosen decomposition and then simplifying the resulting subdiagrams using `full-reduce`:

$$T(D, \varphi) = \{\texttt{full-reduce}(d) \mid d \in E_\varphi\}.$$

This formulation deviates from a standard MDP, as a single action leads to a set of new states rather than a single one. However, the training procedures for MDPs can be generalised to work with this environment. The reward R for an application φ is defined as the negative of the local efficiency, $-\alpha_{loc}$, of the corresponding decomposition. The initial distribution p_0 corresponds to the process of sampling diagrams

from a given class; in [22], the authors sampled from random Clifford+T circuits, grid-like ZX diagrams and random graphs. The reward discount factor γ is treated as a hyperparameter.

To compute the agent's policy distribution $\pi(s, a)$, the authors of [22] employ a two-stage process involving a feature extractor and a policy network. First, the ZX diagram is converted into a graph with node features, as described in Section 2.4.5. The feature extractor, a Graph Attention Network (GAT), processes these features to compute vertex embeddings. These embeddings are then fed into the policy network, which iterates through GAT and MLP layers to compute scalar output features for each node. These scalars are converted into a probability distribution using a softmax function, which can then be sampled (e.g., five times for a Magic5 application).

Additionally, the vertex embeddings are passed to a value network (the critic), which shares the same structure as the policy network but includes an extra pooling layer at the end to compute a single scalar value for the given graph state.

5.2 Proximal Policy Optimisation

To train this selection model, the authors of [22] employ Proximal Policy Optimisation (PPO) [36] with an adapted version of Generalised Advantage Estimation (GAE) [35]. Following the Actor-Critic architecture, PPO trains the policy network and the value network in parallel: The value network is trained to estimate the expected cumulative reward (the value) from a given state. The policy network is then updated based on the *advantage*, which measures whether the actual reward received after an action was better or worse than the estimated value. The policy network is trained to increase the probability of actions that led to a positive advantage and decrease the probability for those with a negative advantage. This use of a relative advantage signal, rather than the absolute reward, reduces the variance of the updates and leads to significantly more stable training. We give a rough sketch of the computation steps; more details can be found in [36].

An episode begins by having the selection driver model interact with the environment, starting from a randomly sampled Clifford+T diagram D. This interaction generates a decomposition tree where the root is D, the internal nodes are intermediate diagrams together with the applied decomposition, and the leaves are the final computed scalars, which will be ignored during training.

For each non-leaf node v in this tree, let $C(v)$ be the non-leaf children of v in the decomposition tree, s_v the state corresponding to v, and a_v the action corresponding

to v. The generalised advantage estimate $\hat{A}_v$ is calculated based on the temporal difference (TD) error δ_v, which measures the difference between the estimated value of a state and the target value:

$$\delta_v = R(v) - V(s_v) + \frac{\gamma}{|C(v)|} \sum_{w \in C(v)} V(s_w),$$

where $V(s_v)$ is the value network's estimate for state v. The advantage of v is then recursively computed as a discounted sum of these errors:

$$\hat{A}_v = \delta_v + \frac{\gamma\lambda}{|C(v)|} \sum_{w \in C(v)} \delta_w,$$

where $\lambda \in [0, 1]$ is the GAE hyperparameter.

The overall PPO loss function combines three components: a policy loss, a value loss, and an entropy loss. The loss is repeatedly computed and the weights ω updated accordingly. For a decomposition tree T, it is computed as:

$$L^{PPO}(\omega) = \frac{1}{|T|} \sum_{v \in T} \left[L_v^{\text{Policy}}(\omega) - c_1 L_v^{\text{Value}}(\omega) + c_2 L_v^{\text{Entropy}}(\omega) \right],$$

where c_1 and c_2 are weighting coefficients. Each component is defined as follows:

Policy Loss

The policy loss uses a clipped surrogate objective to prevent excessively large policy updates. It is defined using the probability ratio $r_v(\omega) = \frac{\pi_\omega(a_v, s_v)}{\pi_{\omega_{\text{old}}}(a_v, s_v)}$, where ω_{old} are the pre-update model weights during the episode. The loss is:

$$L_v^{\text{Policy}}(\omega) = \min \left(r_v(\omega)\hat{A}_v, \text{clip}(r_v(\omega), 1 - \varepsilon, 1 + \varepsilon)\hat{A}_v \right),$$

where ε is a hyperparameter that defines the clipping range.

Value Loss

The value loss, or value function loss, is a mean-squared error that encourages the value network to accurately predict the expected future rewards:

$$L_v^{\text{Value}}(\omega) = \left(V_\omega(s_v) - (\hat{A}_v + V(s_v)) \right)^2.$$

Entropy Loss

The entropy loss encourages exploration by penalising overly confident policies. It is the negative of the entropy of the policy distribution:

$$L_v^{\text{Entropy}}(\omega) = \sum_{a \in \mathcal{A}(s_v)} \pi_\omega(a, s_v) \log \pi_\omega(a, s_v).$$

5.3 Adaptations

The authors of [22] successfully trained a Magic5 selection model that outperformed random selection on random Clifford+T circuits. However, they observed no significant improvement when comparing their model to the full heuristic from [19], which includes Cat decompositions. To address this, we explored a range of adaptations to the reinforcement learning setup with the goal of training models that yield better performance.

Experiment 6: Multiple Base Decompositions

The work of [4] demonstrated that prioritising a Vertex-Cut over a Magic5 application, when the former is guaranteed to yield simplifications, significantly enhances the performance of the stabiliser decomposition algorithm. This finding motivated an expansion of the environment's action space to include Vertex-Cut and Cat applications. Consequently, the agent's architecture was adapted to follow the structure depicted in Figure 3.1; that is, it first selects potential applications, then ranks them, and finally executes the best one. However, a preliminary experiment limited to Vertex-Cut and Magic5 revealed that the model first learnt to exclusively select Magic5 and subsequently optimised its policy only within that portion of the action space. This obstacle might be overcome by pre-training the different components of the model separately or by incentivising the agent to explore different base decompositions. We therefore decided to first focus on training the Magic5 and Vertex-Cut selectors independently.

Experiment 7: Advanced Magic5 Selection

Initial efforts to improve the setup focused on designing a more expressive model capable of learning complex selection strategies. As detailed in Section 2.4.5, we enhanced the node features to include structural information, drawing on research from [2], [46], and [6]. We also experimented with Shortest Path Networks [3], motivated by a study suggesting that SPNs could more reliably distinguish between efficient and inefficient Magic5 applications than other GNN architectures. The corresponding study, which was conducted in the context of a mini-project, can be found in Section A.2.

When these enhancements did not lead to significant performance gains, we shifted our focus to the agent's architecture. Instead of computing a single probability distribution over all spiders and sampling it five times, as in the original work, we im-

plemented a sequential selection process analogous to the advanced model described in Section 4.2. That is, we have for a Magic5 selection $[v_1, v_2, v_3, v_4, v_5]$ on a diagram d

$$\begin{aligned}\pi_\omega([v_1, v_2, v_3, v_4, v_5], d) =& \pi_\omega^{(5)}([v_5], d \mid [v_1, v_2, v_3, v_4]) \\ & \cdot \pi_\omega^{(4)}([v_4], d \mid [v_1, v_2, v_3]) \\ & \cdot \pi_\omega^{(3)}([v_3], d \mid [v_1, v_2]) \\ & \cdot \pi_\omega^{(2)}([v_2], d \mid [v_1]) \\ & \cdot \pi_\omega^{(1)}([v_1], d \mid []),\end{aligned}$$

where each factor in this product is computed with a separate model. This change complicated the calculation of the entropy loss for PPO, as it is computationally unfeasible to compute all conditional selection probabilities. We therefore approximated the entropy by calculating it for each selection step individually rather than over the entire joint action space. This advanced agent, faced with a much larger action space, was unable to extract a clear learning signal and failed to learn a good policy. Its performance on a validation set did not significantly improve with training. This mirrors the results of its supervised learning counterpart and may be due to the fact that the learning algorithm does not attribute the rewards to the individual selections.

Experiment 8: Replacing Generalised Advantage Estimation

Our next attempt to improve the learning process involved modifying the loss function to generate a more effective learning signal. The first approach was to alter the advantage estimation by disregarding future rewards. Since the ultimate goal is a low overall α for the entire decomposition rather than a series of locally optimal steps, we adapted the algorithm to define the reward R for an intermediate action a_v as the negative of the global α of the stabiliser decomposition of s_v after the rollout concludes. This simplifies the formula for the advantage estimate $\hat{A}_v$ to:

$$\hat{A}_v = R(v) - V(s_v).$$

Secondly, we attempted to compute the advantages by comparing the model's rewards not against the value network's estimate, but against the α value that the Sherlock algorithm would achieve. Under this scheme, a positive advantage would indicate that the model outperformed Sherlock on a given diagram, theoretically providing a more informative training signal. However, both of these approaches destabilised the training process and failed to produce models that outperformed previous iterations.

Experiment 9: Graph Cuts

Finally, we adapted the environment to include the Graph Cut decomposition by Codsi [9] as discussed in Section 2.3.5. This changes the transition function T to

$$T(D, \varphi) = \bigcup_{d \in E_\varphi} \texttt{components}(\texttt{full-reduce}(d))$$

where the `components` function maps a diagram to the set of its connected components. As expected, due to the effectiveness of the graph cuts, the Vertex-Cut models trained in this augmented reinforcement learning environment performed better on a validation set. However, they did not significantly outperform the models trained using supervised learning as described in the previous section.

Chapter 6

Efficiency Estimation through Regression

In the previous section, we demonstrated that graph neural networks can be trained to select efficient applications of Vertex-Cuts and discussed avenues to achieve the same for Magic5 decompositions. To bring those two models together as depicted in Figure 3.1, a way is needed to compare these proposed applications. Calculating the local alpha (α_{loc}) is one option, but it has two significant drawbacks. Firstly, computing this value requires performing the actual decomposition, a CPU-bound process that negates the parallelisation advantages offered by GPU-based GNN selection. Secondly, the observation that our Vertex-Cut model outperforms the Sherlock selection procedure on exponentiated Pauli circuits suggests that there are important patterns that extend beyond local simplifications and that these are learnable by GNNs.

Accordingly, this section focuses on training a model to estimate the global efficiency achieved by our decomposition models, given an initial diagram and a specific decomposition application. This task is framed as a regression problem, as the target feature (global efficiency) is a continuous scalar value. We train two separate models, one for Vertex-Cuts and one for Magic5, using the respective training sets of exponentiated Pauli circuits from Chapter 4.

6.1 Setup of the Regression Model

The regression models take a ZX diagram as input, augmented with an additional node feature that identifies the proposed decomposition application in a one-hot encoding. The architecture consists of a feature-encoding MLP, followed by several GNN layers. A graph-level embedding is then produced by taking the mean of the

Estimator	Exponentiated Pauli	Clifford+T+CCZ
α_{loc}	0.296	0.275
VC-Model-Exp-Pauli	0.935	—
VC-Model-Clifford+T+CCZ	—	0.6658

Table 6.1: Pearson correlation coefficients of the global α and different estimators for Vertex-Cut.

Estimator	Exponentiated Pauli Circuits
α_{loc}	0.352
M5-Model-Exp-Pauli	0.539

Table 6.2: Pearson correlation coefficients of the global α and different estimators for Magic5.

intermediate node features. Finally, this embedding is passed through an MLP that outputs the scalar estimate.

We use the mean squared error (MSE) between the model's estimates, $f_\omega(d_i)$, and the labelled global efficiencies, α_i, from the training data as the loss function:

$$L(f_\omega) = \frac{1}{N}\sum_{i=1}^{N}(f_\omega(d_i) - \alpha_i)^2,$$

where N is the size of the training set. The model weights are adjusted to minimise this error using the Adam optimisation routine.

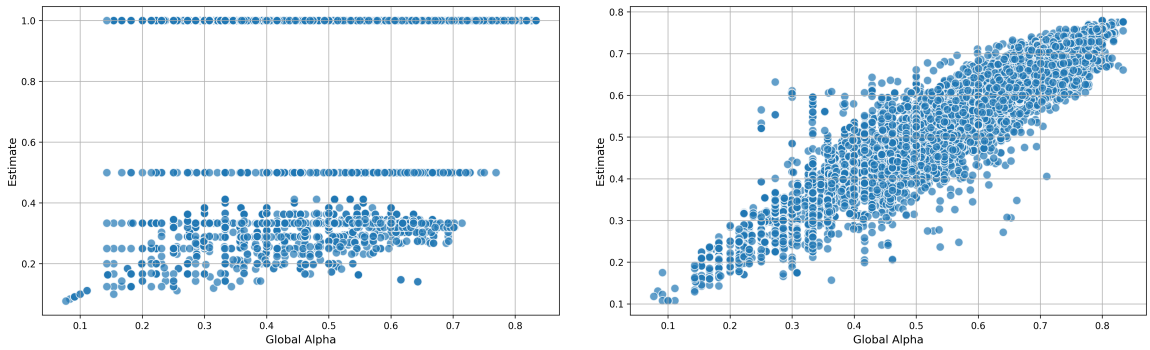
6.2 Evaluation of the Regression Model

We evaluate the utility of the model's outputs for estimating the global α on unseen[1] diagrams and compare it to the predictive power of the local efficiency, α_{loc}. Figure 6.1 through Figure 6.4 display scatter plots of these relationships, while Table 6.1 and Table 6.2 present the corresponding Pearson correlation coefficients

$$corr_{X,Y} = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \overline{Y})^2}}.$$

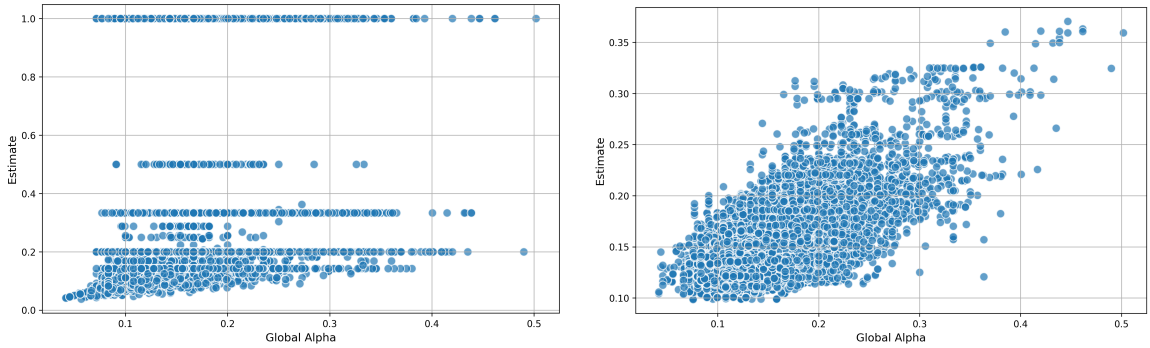
In both cases, the model's estimate has a higher correlation with the global α than the local estimate α_{loc}. The difference is greater for Vertex-Cut than for Magic5, which may be due to the more complex action space.

[1]Our train and test sets contain 1000 diagrams each.



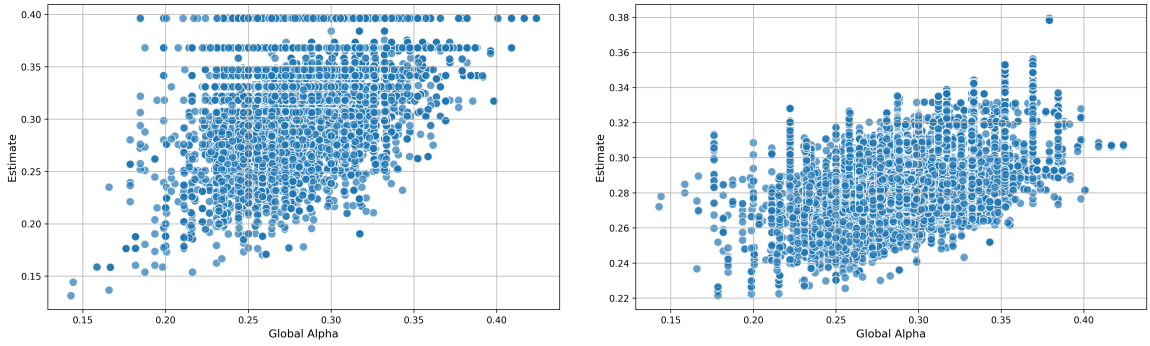
(a) α_{loc} estimate (b) Model estimate

Figure 6.1: Scatter plot of the global α, the estimate α_{loc} and the estimate of the Vertex-Cut estimator model on unseen exponentiated Pauli circuits with Vertex-Cut.



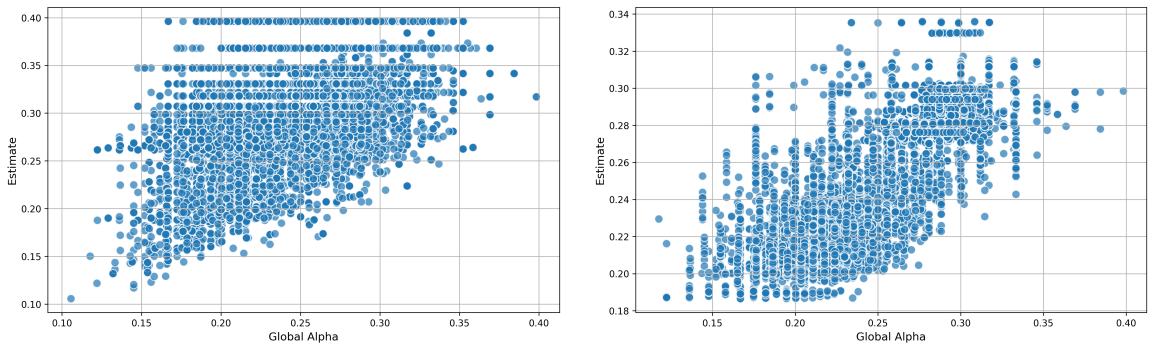
(a) α_{loc} estimate (b) Model estimate

Figure 6.2: Scatter plot of the global α, the estimate α_{loc} and the estimate of the Vertex-Cut estimator model on unseen Clifford+T+CCZ circuits with Vertex-Cut.



(a) α_{loc} estimate (b) Model estimate

Figure 6.3: Scatter plot of the global α, the estimate α_{loc} and the estimate of the Magic5 estimator model on unseen exponentiated Pauli circuits with Magic5.



(a) α_{loc} estimate

(b) Model estimate

Figure 6.4: Scatter plot of the global α, the estimate α_{loc} and the estimate of the Magic5 estimator model on unseen Clifford+T+CCZ circuits with Magic5.

6.3 Combining Vertex-Cut and Magic5

Given the performance of the abovementioned estimators, we can now build a final driver that uses all the trained models following the architecture of Figure 3.1. We combine our best-performing models on exponentiated Pauli circuits, by feeding the candidates of the selection models into the corresponding estimation models and selecting the candidate with the smallest estimate. Since we do not have a trained Cat selection model, we ignore this base decomposition for now.

The hypothesis is that this model, while primarily relying on Magic5 applications, can make use of good Vertex-Cut selections and estimate when they are worth selecting over a Magic5.

Experiment 10: Combined Models

First, we compare our model to the constituent selection models. The results of this experiment can be seen in Figure 6.5, which show that the combined model does not significantly outperform the simple Magic5 selection model. However, this was to be expected, since the Vertex-Cut models drastically underperform Magic5 models on exponentiated Pauli circuits.

Second, we compare it to other drivers that combine Magic5 and Vertex-Cut decompositions. Figure 6.6 shows that our combined model has a similar performance to the full DynamicT heuristic, but is significantly outperformed by Sherlock. This shows that there is room for improvement when selecting the Magic5 and when comparing among different base decompositions.

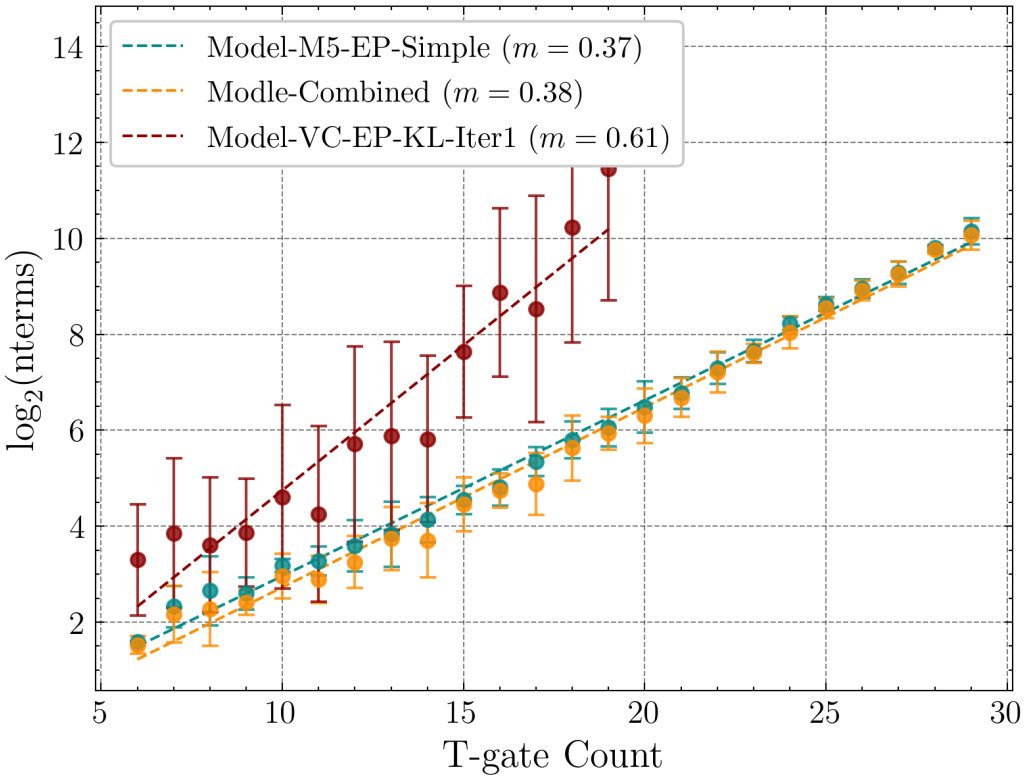


Figure 6.5: Comparison of the combined model and its constituent parts.

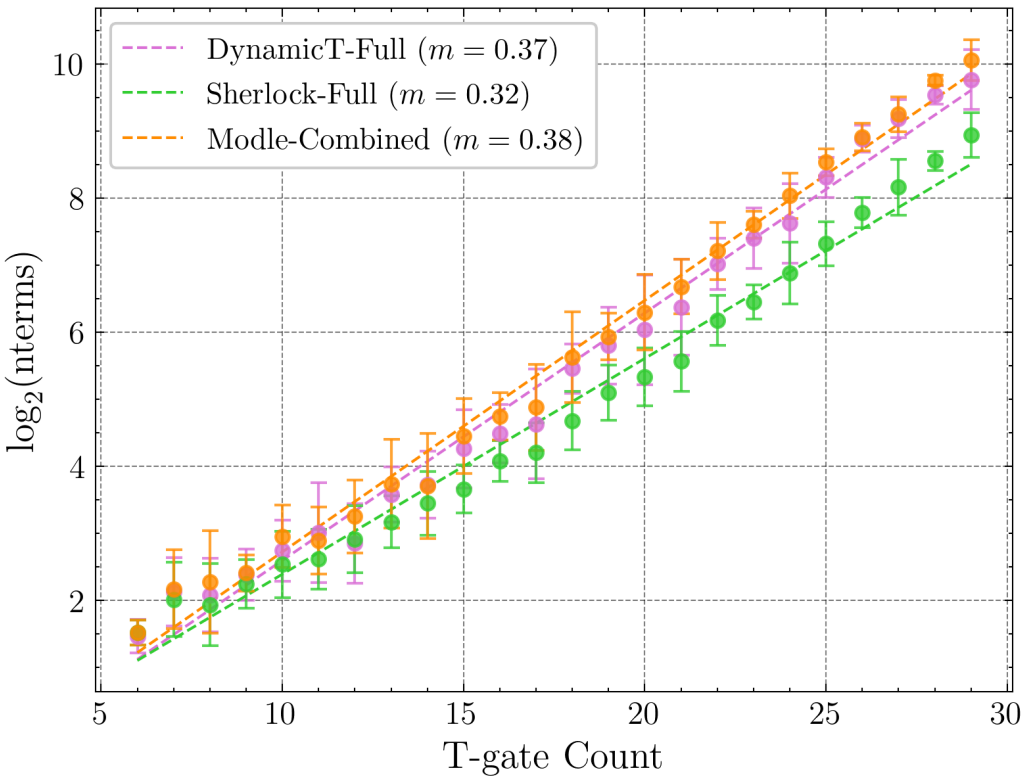


Figure 6.6: Comparison of the combined model and the full Sherlock and DynamicT heuristics.

Chapter 7

Conclusion and Future Work

This work explored a wide range of machine learning applications for the problem of computing stabiliser decompositions of ZX diagrams. Building on the insight from [4] that different applications of the Vertex-Cut decomposition can yield vastly different performance, we successfully trained models that surpassed existing methods. Specifically, our supervised learning model for Vertex-Cut selection on exponentiated Pauli circuits outperformed not only the advanced heuristics from [5] but also the locally optimal Sherlock driver. However, this success proved difficult to translate to other circuit classes or to more complex base decompositions such as Magic5, for which our models did not achieve competitive performance.

The significant performance gap observed between random selection and the Sherlock driver across all tested circuit classes and base decompositions underscores the substantial potential for optimisation in this area, making it a promising avenue for future research. Our positive results with Vertex-Cut and the α-estimation of Chapter 6 demonstrate that graph neural networks are indeed capable of recognising the complex, non-local patterns that lead to good decompositions. This suggests that with greater computational resources and further refinements to the training process, more general and powerful models can be developed. Future work should particularly address the challenge of representing the vast action space of Magic5 applications more effectively, as our sequential selection architecture proved ineffective. Since Cat decompositions have the best guaranteed α of the known simple base decompositions, future models should also make use of these as base decompositions.

It is important to note that while our models demonstrate superior scaling in terms of decomposition size, they were not yet faster than existing drivers in practice. This is because the current QuiZX benchmark implementation does not leverage the inherent parallelisation capabilities of GNNs on GPUs. Integrating these capabilities to fully

utilise the GPU would be a crucial step toward achieving a practical acceleration of the simulation process.

Furthermore, applying techniques from explainable AI for GNNs, as discussed in [45], could offer valuable insights into the concrete patterns learnt by our successful models. These insights could then be used to improve future training procedures or even be distilled into new, robust classical heuristic algorithms.

Bibliography

[1] Scott Aaronson and Daniel Gottesman. Improved Simulation of Stabilizer Circuits. *Physical Review A*, 70(5):052328, November 2004. arXiv:quant-ph/0406196. URL: http://arxiv.org/abs/quant-ph/0406196, doi:10.1103/PhysRevA.70.052328.

[2] Ralph Abboud, İsmail İlkan Ceylan, Martin Grohe, and Thomas Lukasiewicz. The Surprising Power of Graph Neural Networks with Random Node Initialization, June 2021. arXiv:2010.01179 [cs]. URL: http://arxiv.org/abs/2010.01179, doi:10.48550/arXiv.2010.01179.

[3] Ralph Abboud, Radoslav Dimitrov, and Ismail Ilkan Ceylan. Shortest Path Networks for Graph Property Prediction. In *Proceedings of the First Learning on Graphs Conference*, pages 5:1–5:25. PMLR, December 2022. ISSN: 2640-3498. URL: https://proceedings.mlr.press/v198/abboud22a.html.

[4] Wira Azmoon Ahmad. Efficient Heuristics for Classical Simulation of Quantum Circuits Using ZX-Calculus, 2024.

[5] Wira Azmoon Ahmad and Matthew Sutcliffe. Dynamic T-decomposition for classical simulation of quantum circuits, December 2024. arXiv:2412.17182 [quant-ph]. URL: http://arxiv.org/abs/2412.17182, doi:10.48550/arXiv.2412.17182.

[6] Giorgos Bouritsas, Fabrizio Frasca, Stefanos Zafeiriou, and Michael M. Bronstein. Improving Graph Neural Network Expressivity via Subgraph Isomorphism Counting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(1):657–668, January 2023. URL: https://ieeexplore.ieee.org/abstract/document/9721082, doi:10.1109/TPAMI.2022.3154319.

[7] Sergey Bravyi and David Gosset. Improved Classical Simulation of Quantum Circuits Dominated by Clifford Gates. *Physical Review Letters*, 116(25):250501, June 2016. Publisher: American Physical Society.

URL: https://link.aps.org/doi/10.1103/PhysRevLett.116.250501, doi:10.1103/PhysRevLett.116.250501.

[8] Sergey Bravyi, Graeme Smith, and John A. Smolin. Trading Classical and Quantum Computational Resources. *Physical Review X*, 6(2):021043, June 2016. Publisher: American Physical Society. URL: https://link.aps.org/doi/10.1103/PhysRevX.6.021043, doi:10.1103/PhysRevX.6.021043.

[9] Julien Codsi. Cutting-Edge Graphical Stabiliser Decompositions for Classical Simulation of Quantum Circuits, 2022. URL: https://www.maths.ox.ac.uk/system/files/inline-files/J%20Codsi%2021-22.pdf.

[10] gray. quimb: A python package for quantum information and many-body calculations. *ResearchGate*, 2018. URL: https://www.researchgate.net/publication/327432424_quimb_A_python_package_for_quantum_information_and_many-body_calculations, doi:10.21105/joss.00819.

[11] Aimee R. Hatfield and Abdel-Hameed A. Badawy. Moore's Law: What Comes Next? In Hamid R. Arabnia and Leonidas Deligiannidis, editors, *Software Engineering Research and Practice and e-Learning, e-Business, Enterprise Information Systems, and e-Government*, pages 195–206, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-031-86644-9_15.

[12] William Huggins, Piyush Patil, Bradley Mitchell, K Birgitta Whaley, and E Miles Stoudenmire. Towards quantum machine learning with tensor networks. *Quantum Science and Technology*, 4(2):024001, January 2019. Publisher: IOP Publishing. URL: https://dx.doi.org/10.1088/2058-9565/aaea94, doi:10.1088/2058-9565/aaea94.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017. arXiv:1412.6980 [cs]. URL: http://arxiv.org/abs/1412.6980, doi:10.48550/arXiv.1412.6980.

[14] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks, February 2017. arXiv:1609.02907 [cs]. URL: http://arxiv.org/abs/1609.02907, doi:10.48550/arXiv.1609.02907.

[15] Aleks Kissinger and John van de Wetering. Reducing the number of non-Clifford gates in quantum circuits. *Physical Review A*, 102(2):022406, August 2020. Publisher: American Physical Society. URL: https://link.aps.org/doi/10.1103/PhysRevA.102.022406, doi:10.1103/PhysRevA.102.022406.

[16] Aleks Kissinger and John van de Wetering. Simulating quantum circuits with ZX-calculus reduced stabiliser decompositions. *Quantum Science and Technology*, 7(4):044001, July 2022. Publisher: IOP Publishing. URL: https://dx.doi.org/10.1088/2058-9565/ac5d20, doi:10.1088/2058-9565/ac5d20.

[17] Aleks Kissinger and John van de Wetering. *Picturing Quantum Software: An Introduction to the ZX-Calculus and Quantum Compilation*. Preprint, 2024.

[18] Aleks Kissinger and John van de Wetering. PyZX: Large Scale Automated Diagrammatic Reasoning. *Electronic Proceedings in Theoretical Computer Science*, 318:229–241, May 2020. arXiv:1904.04735 [quant-ph]. URL: http://arxiv.org/abs/1904.04735, doi:10.4204/EPTCS.318.14.

[19] Aleks Kissinger, John van de Wetering, and Renaud Vilmart. Classical simulation of quantum circuits with partial and graphical stabiliser decompositions. *LIPIcs, Volume 232, TQC 2022*, 232:5:1–5:13, 2022. arXiv:2202.09202 [quant-ph]. URL: http://arxiv.org/abs/2202.09202, doi:10.4230/LIPIcs.TQC.2022.5.

[20] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, September 2013. Publisher: SAGE Publications Ltd STM. doi:10.1177/0278364913495721.

[21] Mark Koch, Richie Yeung, and Quanlong Wang. Speedy Contraction of ZX Diagrams with Triangles via Stabiliser Decompositions, July 2023. arXiv:2307.01803 [quant-ph]. URL: http://arxiv.org/abs/2307.01803, doi:10.48550/arXiv.2307.01803.

[22] Alexander Koziell-Pipe, Richie Yeung, and Matthew Sutcliffe. Towards Faster Quantum Circuit Simulation Using Graph Decompositions, GNNs and Reinforcement Learning. In *The 4th Workshop on Mathematical Reasoning and AI at NeurIPS'24*, October 2024. URL: https://openreview.net/forum?id=54060pbCKY.

[23] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015. Publisher: Nature Publishing Group. URL: https://www.nature.com/articles/nature14539, doi:10.1038/nature14539.

[24] Igor L. Markov and Yaoyun Shi. Simulating Quantum Computation by Contracting Tensor Networks. *SIAM Journal on Computing*, 38(3):963–981, January 2008. Publisher: Society for Industrial and Applied Mathematics. URL: https://epubs.siam.org/doi/abs/10.1137/050644756, doi:10.1137/050644756.

[25] Alexander Mattick, Maniraman Periyasamy, Christian Ufrecht, Abhishek Y. Dubey, Christopher Mutschler, Axel Plinge, and Daniel D. Scherer. Optimizing Quantum Circuits via ZX Diagrams using Reinforcement Learning and Graph Neural Networks, April 2025. arXiv:2504.03429 [cs]. URL: http://arxiv.org/abs/2504.03429, doi:10.48550/arXiv.2504.03429.

[26] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. Adaptive computation and machine learning. The MIT Press, Cambridge, Mass. London, 2012.

[27] Ramis Movassagh. The hardness of random quantum circuits. *Nature Physics*, 19(11):1719–1724, November 2023. Publisher: Nature Publishing Group. URL: https://www.nature.com/articles/s41567-023-02131-2, doi:10.1038/s41567-023-02131-2.

[28] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022. URL: http://probml.github.io/book1.

[29] Michael A. Nielsen and Isaac L. Chuang. Quantum Computation and Quantum Information: 10th Anniversary Edition, December 2010. ISBN: 9780511976667 Publisher: Cambridge University Press. URL: https://www.cambridge.org/highereducation/books/quantum-computation-and-quantum-information/01E10196D0A682A6AEFFEA52D53BE9AE, doi:10.1017/CBO9780511976667.

[30] Maximilian Nägele and Florian Marquardt. Optimizing ZX-Diagrams with Deep Reinforcement Learning. *Machine Learning: Science and Technology*, 5(3):035077, September 2024. arXiv:2311.18588 [quant-ph]. URL: http://arxiv.org/abs/2311.18588, doi:10.1088/2632-2153/ad76f7.

[31] Román Orús. Tensor networks for complex quantum systems. *Nature Reviews Physics*, 1(9):538–550, September 2019. Publisher: Nature Publishing Group. URL: https://www.nature.com/articles/s42254-019-0086-7, doi:10.1038/s42254-019-0086-7.

[32] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, August 2014. Google-Books-ID: VvBjBAAAQBAJ.

[33] Hammam Qassim, Hakop Pashayan, and David Gosset. Improved upper bounds on the stabilizer rank of magic states – Quantum, 2021. URL: https://quantum-journal.org/papers/q-2021-12-20-606/.

[34] Jordi Riu, Jan Nogué, Gerard Vilaplana, Artur Garcia-Saez, and Marta P. Estarellas. Reinforcement Learning Based Quantum Circuit Optimization via ZX-Calculus, May 2025. arXiv:2312.11597 [quant-ph]. URL: http://arxiv.org/abs/2312.11597, doi:10.48550/arXiv.2312.11597.

[35] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation, October 2018. arXiv:1506.02438 [cs]. URL: http://arxiv.org/abs/1506.02438, doi:10.48550/arXiv.1506.02438.

[36] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, August 2017. arXiv:1707.06347 [cs]. URL: http://arxiv.org/abs/1707.06347, doi:10.48550/arXiv.1707.06347.

[37] Chence Shi, Minkai Xu, Zhaocheng Zhu, Weinan Zhang, Ming Zhang, and Jian Tang. GraphAF: a Flow-based Autoregressive Model for Molecular Graph Generation, February 2020. arXiv:2001.09382 [cs]. URL: http://arxiv.org/abs/2001.09382, doi:10.48550/arXiv.2001.09382.

[38] Matthew Sutcliffe. Smarter k-Partitioning of ZX-Diagrams for Improved Quantum Circuit Simulation, September 2024. arXiv:2409.00828 [quant-ph]. URL: http://arxiv.org/abs/2409.00828, doi:10.48550/arXiv.2409.00828.

[39] Matthew Sutcliffe and Aleks Kissinger. Procedurally Optimised ZX-Diagram Cutting for Efficient T-Decomposition in Classical Simulation. *Electronic Proceedings in Theoretical Computer Science*, 406:63–78, August 2024. URL: http://arxiv.org/abs/2403.10964v2, doi:10.4204/EPTCS.406.3.

[40] Matthew Sutcliffe and Aleks Kissinger. Fast classical simulation of quantum circuits via parametric rewriting in the ZX-calculus, February 2025. arXiv:2403.06777 [quant-ph]. URL: http://arxiv.org/abs/2403.06777, doi:10.48550/arXiv.2403.06777.

[41] Richard S Sutton and Andrew G Barto. Reinforcement Learning: An Introduction. *1992*, 1992.

[42] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks, February 2018. arXiv:1710.10903 [stat]. URL: http://arxiv.org/abs/1710.10903, doi:10.48550/arXiv.1710.10903.

[43] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful are Graph Neural Networks?, February 2019. arXiv:1810.00826 [cs]. URL: http://arxiv.org/abs/1810.00826, doi:10.48550/arXiv.1810.00826.

[44] Kieran Young, Marcus Scese, and Ali Ebnenasir. Simulating Quantum Computations on Classical Machines: A Survey, November 2023. arXiv:2311.16505 [quant-ph]. URL: http://arxiv.org/abs/2311.16505, doi:10.48550/arXiv.2311.16505.

[45] Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. Explainability in Graph Neural Networks: A Taxonomic Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(5):5782–5799, May 2023. URL: https://ieeexplore.ieee.org/abstract/document/9875989, doi:10.1109/TPAMI.2022.3204236.

[46] Bohang Zhang, Shengjie Luo, Liwei Wang, and Di He. Rethinking the Expressive Power of GNNs via Graph Biconnectivity, February 2024. arXiv:2301.09505 [cs]. URL: http://arxiv.org/abs/2301.09505, doi:10.48550/arXiv.2301.09505.

Appendix A

Appendix

A.1 Implementation Details

The code for this thesis can be found on GitHub under

`https://github.com/Derbeldumm/GNNs-For-StabiliserDecomps`.

To a large extent, it builds on the codebase that Koziell-Pipe, Yeung and Sutcliffe used in [22]. The quizx fork used for benchmarking is available under `https://github.com/Derbeldumm/quizx`.

Model Architecture

In each model, the input encoder is an MLP with 2 hidden layers. The ranking heads have 3 hidden layers of size D, $\frac{D}{4}$ and $\frac{D}{16}$, where D is the embedding dimension. All activation functions are ReLU.

Hyperparameters

When training the models with supervised learning, we used the capabilities of Weights and Biases to perform a Bayesian hyperparameter optimisation in the ranges presented in Table A.1. The additional PPO-specific hyperparameters for the reinforcement learning part are presented in Table A.2.

For supervised and reinforcement learning, we generated a validation set of 100 circuits and after training 100 models, we selected one that had a low loss value and a good performance on the validation set. For the efficiency estimation models, we had a validation set of 100 circuits and implemented early stopping when the loss on the validation set started to increase for 5 consecutive epochs.

Hyperparameter	Values
GNN model	[GIN, GAT, GCN]
GNN Layers	4 - 10
Embedding Dimension	[128, 256, 512, 1024]
Learning Rate	1e-4 - 1e-2
Weight Decay	1e-8 - 1e-3
Dropout	1e-2 - 0.5
Gradient Norm Clipping	0.5
Batch Size	128
Epochs	30
Runs	100
Training Dataset Size	10,000

Table A.1: Hyperparameter ranges used to train the supervised learning models.

Hyperparameter	Values
PPO Clip Range	0.1 - 0.3
GAE gamma	0.9 - 0.99
GAE lambda	0.9 - 0.99
Entropy Loss coefficient	1e-3 - 3e-2
Value Loss coefficient	0.5 - 0.8
PPO Updates per Rollout	1 - 20
Steps per Rollout	512 - 4048
GAE Normalisation	[False, True]
Gradient Norm Clipping	0.5 - 2

Table A.2: Hyperparameter ranges used to train the reinforcement learning models.

A.2 Study on the Expressiveness of Graph Neural Networks

This appendix includes a previous study on the expressiveness of graph neural networks for stabiliser decompositions that was conducted in the context of a Mini-Project for the course "Graph Representation Learning" at the University of Oxford in Michaelmas Term 2024. This study is included because it has not been published elsewhere. It is not part of the primary work for this thesis and should be considered separate.

A.2.1 Abstract

This mini-project explores how well different Graph Neural Network (GNN) architectures can express the algebraic structures underpinning quantum circuit decompositions. By studying a simple classification task—determining whether a proposed decomposition is efficient—we benchmark several GNN models (Graph Convolutional Networks, variants with self-weights and random node initialisation, Shortest Path Networks, and Graph Attention Networks). Among the tested models, Shortest Path Networks demonstrate the highest average accuracy (0.8), indicating strong representational capabilities, even greater than those of GATs, which already have been employed in a more complicated variant of this task.

A.2.2 Introduction

We aim to investigate the ability of different Graph Neural Network (GNN) architectures to express specific algebraic structures. More specifically, we focus on the algebraic structures present in the classical simulation of quantum circuits. One way to perform this classical simulation is by applying a series of decompositions and rewriting rules [19] on the circuit. The main challenge is to choose the most efficient place in the circuit to decompose at each step — a process we call Decomposition Proposal. Because of the nature of quantum circuits, Decomposition Proposal employs complex algebraic structures and currently lacks a known efficient algorithm. However, recent work has explored leveraging the power of GNNs and Proximal Policy Optimisation (PPO) to propose circuit decompositions that are more efficient than basic heuristics [22]. In this mini-project, we study how much of these algebraic structures different GNN architectures are able to express. To this end, we study

their performance on a simplified classification problem: given a circuit and a decomposition, predict whether the decomposition is efficient. Those that do well in this setup are expected to also work well for Decomposition Proposal.

A.2.3 Method

A.2.3.1 Background on Quantum Circuit Simulation

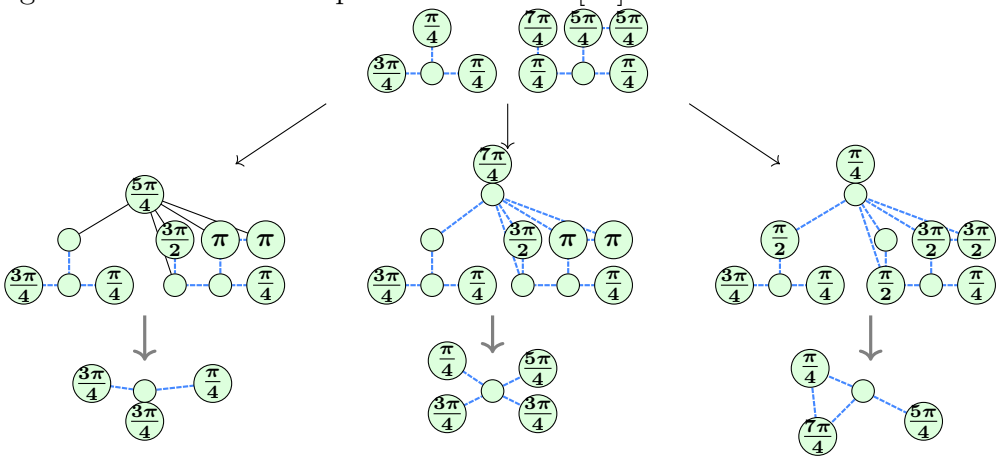
The overall goal is to calculate the amplitude of a specific output vector for a given circuit. Such a calculation is efficiently solvable for quantum circuits containing only the Clifford gate set. Adding another gate called the T-gate yields two major effects: (1) The gate set becomes approximately universal, meaning one can approximate any other quantum circuit using only Clifford+T gates; (2) no efficient classical simulation technique is known for circuits with T-gates.

While it is generally expected that simulating arbitrary quantum computations classically is difficult, it remains interesting — particularly in algorithm development — to study the behaviour of small instances without running them on a (still largely unavailable) quantum device. One classical approach converts the circuit into a ZX-diagram and then performs a series of simplification steps, reducing the diagram to a single scalar that yields the desired amplitude [19]. This approach succeeds for Clifford circuits, but T-gates cannot be simplified in the same way. To handle T-gates, one can decompose the diagram into smaller subdiagrams, each containing fewer T-gates than the original diagram. In the worst case, this has to be repeated $O(n)$ times, so the technique yields an exponential number of classically simulatable subdiagrams, each contributing to the amplitude. One such decomposition is the magic-5 decomposition. Applying a decomposition removes a fixed number of T-gates but may also make further diagrammatic simplifications possible, removing additional gates. The ratio between the number of subdiagrams and the number of removed T-gates defines the efficiency of the decomposition. Depending heavily on the underlying algebraic structure of the diagram, different decomposition choices can vary significantly in efficiency. Choosing an optimal decomposition yields exponentially fewer subdiagrams, enabling the classical simulation of larger circuits.

A.2.3.2 Example

The following example shows a ZX diagram containing eight T-gates (nodes with phases not multiples of 0.5π). The magic-5 decomposition creates three subdiagrams with four T-gates each. When further simplifying two of them end up with only

three T-gates. Choosing another way to decompose may have led to all three sub-diagrams containing four T-gates (meaning a less efficient decomposition). For more background on these decompositions refer to [19].



A.2.3.3 Data Generation:

We use the pyzx library [18] to generate all circuits and to calculate the efficiency of decompositions. Specifically, we generate Clifford+T circuits with 15 to 25 qubits, sampled from a normal distribution N(20,5) for the number of qubits, and a depth sampled from U(10,30) and scaled by the number of qubits. Those circuits are then fully reduced and brought into graph form (refer to the example). The dataset is filtered so that half the instances have the baseline efficiency of 0.396 (this happens when after the initial decomposition no further T-gates get removed by simplifications), and the other half have a lower (better) efficiency. In the node features we encode the phases and whether a node is marked for the magic-5 decomposition.

A.2.3.4 Models

We implemented a range of models that all consist of some message passing part and a succinct global classification head. The pooling is done via global mean pooling. Between all layers, we apply an activation function. The following models were chosen as according to the lecture they differ in their theoretical expressive power.

Graph Convolutional Network (GCN):

As a baseline model, we implemented a standard MPNN with Convolutional Layers from [14].

GCN + Self-Weights:

On top of each Convolutional Layer, we add the current node features with trainable linear weights.

GCN + Self-Weights + Random Node Initialisation:

This model extends the GCN with self-weights by randomly initializing eight additional node features sampled from $U(0, 1)$ [2].

Shortest Path Network:

We apply a shortest-path-based network approach [3] that assigns weights to each ring (up to k=3), followed by an MLP. This yields the message passing node update:

$$\mathbf{h}_u^{(t+1)} = MLP\left(W_0\mathbf{h}_u^{(t)} + \sum_{i=1}^{3} W_i \frac{1}{|\mathcal{N}_i(u)|} \sum_{v\in\mathcal{N}_i(u)} \mathbf{h}_v^{(t)}\right)$$

Graph Attention Networks:

At last, we tested a Graph Attention Network consisting of four attention heads and added self-loops. Between layers, we also insert normalisation layers to mirror the structure of the feature extractor used in [22] more closely.

A.2.3.5 Hyperparameters

We use 1,000 circuits as the training set. Training is performed using Adam with the following hyperparameters:

- Hidden dimensions: 32
- 5 message-passing layers
- 3-layer global classification head
- ReLU activation
- Learning rate: 0.0005
- Batch size: 32
- Early stopping with patience 15 on a validation set of 100
- Maximum of 100 epochs

Final accuracy is measured on a test set of 1,000 circuits.

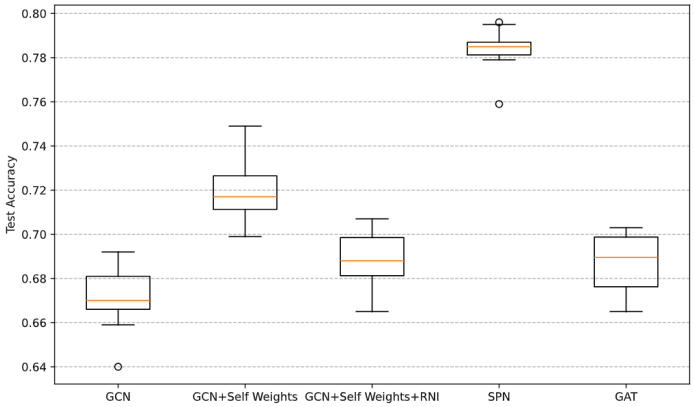
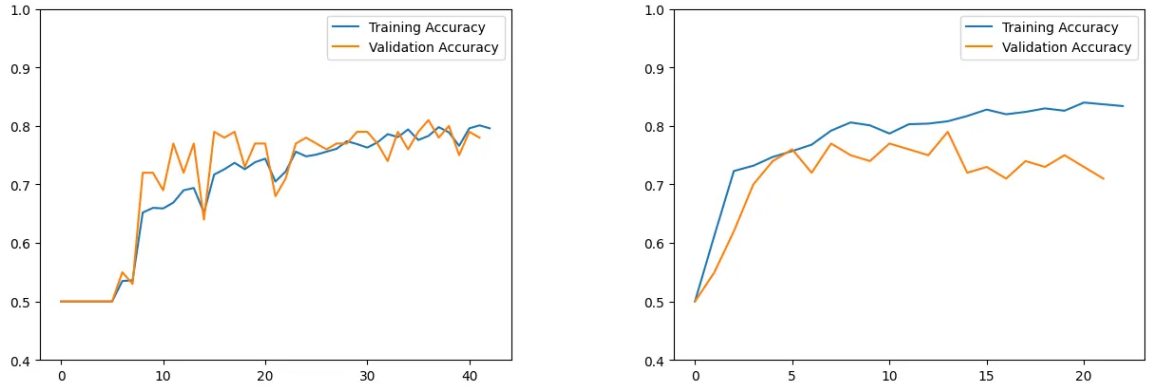


Figure A.1: Comparison of Model Performance over 10 Runs



(a) Shortest Path Network

(b) Graph Attention Network

Figure A.2: Representative training runs

A.2.4 Empirical Results

All models successfully learnt from the data. The baseline Graph Convolutional Network (GCN) achieved a mean test accuracy of 0.65. Adding self-weights improved performance to 0.7, while random node initialisation did not yield further gains but rather hindered learning. Graph Attention Networks (GATs) showed promising performance on the training set but exhibited overfitting tendencies, so the test scores were worse. Shortest Path Networks (SPNs) achieved the best performance with a mean test accuracy of 0.8. Figure A.2 shows representative training runs for SPN and GAT, while Figure A.1 should present the performance of all models with standard deviations over 10 runs for a comprehensive comparison. Note that a GAT based approach was used in [22] for Decomposition Proposal but our results imply that SPNs may be the better choice to work on quantum circuits.

A.2.5 Conclusions

This study provides preliminary insights into the expressive power of various GNN architectures for analyzing quantum circuits. Our results suggest that SPNs are particularly well-suited for this task, so they should be considered for further studies.

It is important to acknowledge the limitations of this study as a mini-project. Hyperparameter tuning was only done manually and without proper methodology. The computational resources restricted the number of layers and experimental runs. Therefore, the observed performance limitations of some models may be attributable to these constraints rather than inherent limitations in their expressive power. Furthermore, the training procedure would benefit from the use of regularisation techniques. Future work could address these issues to build more confidence in our findings. One could also investigate which value of k for the SPN gives the best results while being computationally feasible. Additionally, evaluating the models' ability to predict the actual efficiency of decompositions, rather than just a binary classification, would provide a more nuanced understanding of their capabilities. This could then be integrated with the framework presented in [22] to do Decomposition Proposal.

Furthermore, this study focused solely on magic-5 decompositions. Exploring other decomposition strategies, such as cat-decompositions, could be beneficial for theoretical understanding as well as practical performance. Progress in this area would greatly increase the feasibility of simulating quantum circuits on classical hardware.