A Rank-Width–Based Approach to Quantum Circuit Simulation via ZX-Calculus

Fedor Kuyanov
Kellogg 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 deepest gratitude to my dissertation supervisor, *Aleks Kissinger*, for his exceptional guidance, encouragement, and insight throughout this research. Aleks introduced me to the remarkable field of Quantum Computing and the powerful framework of ZX-calculus, both of which have shaped the direction of this thesis. His brilliant ideas, sharp intuition, and contagious enthusiasm have continually inspired me to work hard and pursue ambitious goals.

I would also like to thank *David Philipps* for many insightful discussions and helpful suggestions, which helped me develop a clearer understanding of the broader landscape of the field.

Abstract

Quantum computing promises to transform the landscape of computation, yet the limitations of current hardware make classical simulation indispensable for the design, optimisation, and verification of quantum algorithms. This dissertation develops a new simulation strategy based on the ZX-calculus, a graphical formalism for reasoning about quantum circuits. We exploit the ZX diagram's rank-width to perform tensor contractions more efficiently. In particular, we introduce a simulation pipeline that reduces a Clifford+T circuit to a graph-like ZX diagram, extracts a linear rank-decomposition via extended gflow, refines it through simulated annealing, and performs a recursive contraction algorithm along the resulting decomposition tree. The computational complexity of our method depends primarily on the cut-rank structure of the decomposition.

Our benchmarks demonstrate that our method outperforms other state-of-the-art tensor contraction routines by a factor of 10 on random circuits of different sizes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Quantum computing leverages principles of quantum mechanics – such as superposition, interference, and entanglement – to perform computations. This rapidly evolving field enables us to solve complex problems significantly faster than on classical computers. These problems arise in various fields such as cryptography, artificial intelligence, healthcare, and chemistry. Yet, at present, reliable large-scale quantum hardware remains a major challenge to build. In the meantime, simulating quantum circuits on classical computers is crucial for testing and optimising the algorithms before deploying them on actual quantum hardware. Moreover, the techniques used for simulation can provide a deeper understanding of the problem and potentially shed light on how to approach it from a classical perspective.

Graphical languages like ZX/ZH-calculus [6, 12] serve as a helpful framework for reasoning about complex quantum circuits. In addition to being a convenient formalism, they form a basis for various simulation strategies. Quantum circuits can be transformed into a ZX/ZH diagram and optimised using a set of local rules. The sequence of transformations results in a more compact version of the diagram, which may not be directly decomposable into quantum gates. The outcome of the original circuit is then computed classically by evaluating the final diagram using various advanced strategies based on T-count reduction, graph partitioning techniques, etc.

This thesis introduces a new simulation algorithm that takes advantage of the ZX diagram's rank-width to perform tensor contractions more efficiently. As computing rank-width exactly is known to be NP-hard [9], many practical approaches exist for finding rank-decompositions of sufficiently good quality. We start with the rank-decomposition obtained from the extended gflow [1, 19] and apply simulated annealing as in [16]. This enables us to optimise the complexity of our main contraction routine. Finally, we compare the performance of our algorithm with existing strategies on various quantum circuits.

This dissertation is organised as follows.

In Chapter 2, we give a background on quantum computation, tensor networks, and the ZX-calculus. We discuss the rules for converting a quantum circuit into a ZX diagram and provide a complete set of local transformations. We also discuss CNOT-only circuits, where the corresponding unitary is determined by a parity map.

In Chapter 3, we explain some of the known approaches for quantum simulation using ZX-calculus. First, we discuss Clifford circuits, which can be simulated in polynomial time. Then we turn to the general case by considering Clifford+T circuits, as they are computationally universal. We discuss the stabiliser decompositions – an essential tool for reducing T-count at the cost of branching into several instances. We also discuss simulation techniques relying on the connectivity of the diagram. Finally, we discuss other approaches such as greedy strategies and those based on simulated annealing and reinforcement learning.

In Chapter 4, we give all the necessary mathematical background for a clear presentation of our strategy. First, we recall some linear algebra over GF(2) and discuss the notions of rank factorisation and generalised inverse. Then, we define the rank-width and rank-decomposition of a graph, and briefly discuss their applications in classical simulation. We proceed by defining gflow and extended gflow on graphs, which are helpful for obtaining a good starting point for rank minimisation.

In Chapter 5, we illustrate our new method for quantum circuit evaluation using rank-width. First, we give the general pipeline of the strategy. Then, we rigorously prove that extended gflow provides an efficient rank-decomposition. We also discuss the method for further rank optimisation using simulated annealing. Finally, we explain the main tensor contraction routine and discuss its implementation bottlenecks.

Finally, the benchmarks of our strategy are presented in Chapter 6. First, we evaluate the quality of the rank-decompositions found by the annealer. Then, we compare our whole simulation strategy against Quimb and see that on random circuits, our method performs 10 times faster than the baseline.

Chapter 2

Quantum Computing and ZX-Calculus

2.1 Quantum circuits

In this section, we explain the basics of quantum computation. In particular, we discuss qubits, quantum gates, and measurements, which are the building blocks of quantum circuits.

2.1.1 Qubits

A *qubit* is a quantum analogue of a classical bit: its value varies over all normalised complex linear combinations of 0 and 1, being the element of the two-dimensional complex Hilbert space $\mathbb{C}^2$ with the standard inner product. Formally,

Definition 2.1.1. *A* qubit *is a vector* $|\psi\rangle = a\,|0\rangle + b\,|1\rangle = \begin{pmatrix} a \\ b \end{pmatrix} \in \mathbb{C}^2$ *satisfying* $\|\psi\|^2 = |a|^2 + |b|^2 = 1$.

Here we used the Dirac notation, also called the bra–ket notation. Generally, there are two ways to represent a vector $\psi \in \mathbb{C}^2$: either as a column-vector ('ket') $|\psi\rangle = \begin{pmatrix} a \\ b \end{pmatrix}$, or as a row-vector ('bra') $\langle\psi| = \begin{pmatrix} \overline{a} & \overline{b} \end{pmatrix}$, where $\overline{x}$ denotes the complex conjugate of x. The *inner product* $\langle\psi, \phi\rangle$ is then written concisely as $\langle\psi|\phi\rangle$.

There are several orthonormal bases widely used in quantum computing:

1. The *standard computational basis (Z-basis)*, given by $|0\rangle := \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle := \begin{pmatrix} 0 \\ 1 \end{pmatrix}$,
2. The *X-basis*, given by $|+\rangle := \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, |-\rangle := \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$,

3. The *Y-basis*, given by $|+i\rangle := \frac{1}{\sqrt{2}}\begin{pmatrix}1\\ i\end{pmatrix}, |-i\rangle := \frac{1}{\sqrt{2}}\begin{pmatrix}1\\ -i\end{pmatrix}$.

These vectors are the eigenstates of the respective *Pauli matrices*

$$Z = \sigma_z := \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}, \qquad X = \sigma_x := \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}, \qquad Y := \sigma_y := \begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}.$$

Unlike the classical case, the collection of qubits cannot generally be represented as a collection of $\mathbb{C}^2$-vectors. Indeed, such states allow no entanglement and are called *product states*, which form a small subset of all possible states. Instead, the n-qubit state is defined as a normalised vector of $(\mathbb{C}^2)^{\otimes n}$, a tensor product of n copies of $\mathbb{C}^2$. Formally,

Definition 2.1.2. *The tensor product $U \otimes V$, where U, V are Hilbert spaces with orthonormal bases $|u_1\rangle, \ldots, |u_n\rangle$ and $|v_1\rangle, \ldots, |v_m\rangle$, is defined as the set of linear combinations of all possible pairs (u_i, v_j), i.e. $\left\{\sum_{i,j} c_{ij} |u_i\rangle \otimes |v_j\rangle : c_{i,j} \in \mathbb{C}\right\}$. The inner product $\langle c, d\rangle$ is denoted by $\sum_{i,j} \overline{c_{ij}} d_{ij}$.*

The n-qubit state can then be defined as the 2^n-term sum $\sum_{i\in\{0,1\}^n} a_i |i_1 \ldots i_n\rangle$ with $\sum_i |a_i|^2 = 1$, where $|i_1 \ldots i_n\rangle$ is a shorthand notation for $|i_1\rangle \otimes \cdots \otimes |i_n\rangle$. Note that since $(\mathbb{C}^2)^{\otimes n}$ is isomorphic to $\mathbb{C}^{2^n}$, we can assume the state is a ket vector of size 2^n.

2.1.2 Quantum gates

In a quantum computation, the states are transformed into one another using quantum gates.

Definition 2.1.3. *The n-qubit quantum gate is a unitary linear map $U\colon \mathbb{C}^{2^n} \mapsto \mathbb{C}^{2^n}$, i.e., a linear map preserving the inner product.*

In the standard computational basis, this map is expressed by a unitary $2^n \times 2^n$ matrix U, i.e., satisfying $U^*U = I_{2^n}$, where U^* denotes the conjugate transpose. Below are the standard examples of quantum gates.

1. The X-, Y-, and Z-gates $\left(-\boxed{X}-, -\boxed{Y}-, -\boxed{Z}-\right)$ denoted by $\sigma_x, \sigma_y, \sigma_z$.
2. The Hadamard gate $-\boxed{H}-$ denoted by $\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$.
3. The S, T gates $\left(-\boxed{S}-, -\boxed{T}-\right)$ denoted by $\begin{pmatrix}1 & 0\\ 0 & i\end{pmatrix}$ and $\begin{pmatrix}1 & 0\\ 0 & e^{\pi i/4}\end{pmatrix}$, respectively.
4. The CNOT gate  , $|a\rangle \otimes |b\rangle \to |a\rangle \otimes |a \oplus b\rangle$ and extended by the linearity.

2.1.3 Quantum measurements

The last ingredient of quantum circuits is measurements, which allow us to extract classical information from the qubits.

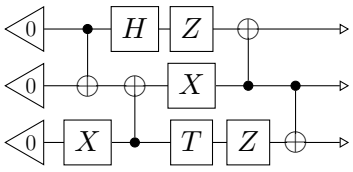
Definition 2.1.4. *A* measurement *is a set of projectors* $\mathcal{M} = \{M_0, \ldots, M_{k-1}\}$ *such that* $\sum_i M_i = I$*. The probability of the* i*-th outcome when we measure the state* $|\psi\rangle$ *is given by the* Born rule*:* $\mathrm{Prob}(i \mid \psi) = \langle\psi| M_i |\psi\rangle$.

In particular, when measuring a single-qubit system in the state $a|0\rangle + b|1\rangle$, we can take $M_0 = |0\rangle\langle 0|$, $M_1 = |1\rangle\langle 1|$ so that the probabilities of getting 0, 1 are $p_0 = |a|^2$, $p_1 = |b|^2$. This is called *measurement in the standard basis*. When measuring n qubits in the state $\sum_i a_i |i_1 \ldots i_n\rangle$ one by one in the standard basis, the probability of getting a bit string k equals $|a_k|^2$. Note that as our state is normalised, all probabilities sum up to 1.

Quantum circuits consist of three parts, from left to right:

1. an input state, usually $|0 \ldots 0\rangle$ by default,
2. a sequence of quantum gates applied to different qubits,
3. single-qubit measurements, typically in the standard basis.

The output of the quantum circuit is the bit string obtained from the measurements. Hence, a quantum algorithm constitutes a nondeterministic process. A circuit example is given below:



(2.1)

2.2 Tensor networks

Another way to view quantum circuits is through the tensor network formalism. Let us briefly introduce all the necessary notions; see [6] and [12] for a complete guide to calculations with tensors and string diagrams.

There are several ways to define tensors in mathematics, such as multilinear maps or using tensor products. In computer science, it is more natural to consider tensors as multidimensional arrays. We will assume there are two kinds of dimensions – inputs and outputs – and write the corresponding indices as subscripts and superscripts, respectively.

Definition 2.2.1. *An (n, m)-tensor T, i.e., having n inputs and m outputs, is a multidimensional array of size $d_1 \times \cdots \times d_n \times d^1 \times \cdots \times d^m$ such that $T^{i^1 \ldots i^m}_{i_1 \ldots i_n} \in \mathbb{C}$ for each $i_k \in [0; d_k)$ and $i^l \in [0; d^l)$, where $d_k, d^l \in \mathbb{N}$ are called input and output dimensions.*

In particular, a tensor having

- 0 inputs and 0 outputs is a scalar,
- 0 inputs and 1 output is a column vector,
- 1 input and 0 outputs is a row vector,
- 1 input and 1 output is a matrix.

The addition of two tensors is defined as the element-wise sum of the arrays. The tensor product and composition turn out to be the special case of tensor contraction, which is defined as follows.

Definition 2.2.2. *Given two tensors A, B of input and output dimensions $\hat{n}_i, \check{n}^i, \hat{m}_i, \check{m}^i$ and two sets of indices $O \subset [1; |\check{n}|], I \subset [1; |\hat{m}|]$, where $|O| = |I| = k$ and $\check{n}^{O_i} = \hat{m}_{I_i}$ for every $i \in [1; k]$, the* contraction *of A, B along O, I is a new tensor $A^O B_I$ such that*

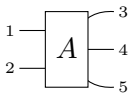
$$\left(A^O B_I\right)^{\check{a}^{\bar{O}} \check{b}}_{\hat{a}\hat{b}_{\bar{I}}} = \sum_{\check{a}^{O_1} = \hat{b}_{I_1} = 0}^{\hat{m}_{I_1} - 1} \cdots \sum_{\check{a}^{O_k} = \hat{b}_{I_k} = 0}^{\hat{m}_{I_k} - 1} A^{\check{a}}_{\hat{a}} B^{\check{b}}_{\hat{b}},$$

where $\hat{a}, \check{a}, \hat{b}, \check{b}$ denote the input and output indices of A and B respectively, and $\bar{O}, \bar{I}$ denote the complements of O and I.

In other words, the contraction is the sum of products of the corresponding values of the initial tensors, where the summation is taken over indices not present in the resulting tensor. For instance, given two matrices A, B, their contraction $A^1 B_1$ corresponds to the matrix product BA (the order is flipped because matrices act on a vector on their right). Another example is the inner product $\langle \psi | \phi \rangle$ being the contraction of $\langle \psi |$ and $| \phi \rangle$.

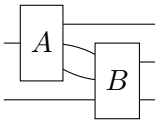
Graphically, tensors are represented as boxes with loose wires. Each wire has a dimension assigned to it (2 by default), and the inputs are located on the left while the outputs are on the right, so that the diagram is read from left to right. In some

literature, diagrams are read from top to bottom or from bottom to top as well. Here is an example of a tensor with input dimensions 1, 2, and output dimensions 3, 4, 5:



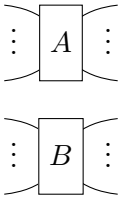
Tensor contraction is represented by joining the corresponding inputs and outputs together:

$$A^{2,3}B_{1,2} =$$



Another special case of tensor contraction is *tensor product*:

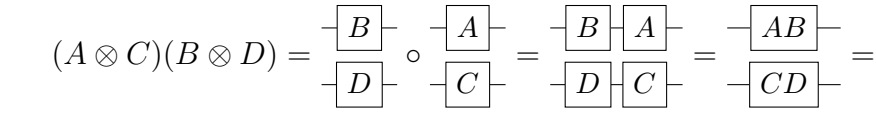
$$A \otimes B = A^{\varnothing}B_{\varnothing} =$$



In general, a *tensor diagram* is an undirected graph with possibly loose edges, where each vertex represents a tensor and each non-loose edge represents a contraction. The value of a tensor diagram can be computed by performing contractions until only one vertex remains. By convention, an empty tensor diagram represents scalar 1.

Tensor diagrams enable us to prove various algebraic identities without worrying about indices. For example, the famous *interchange law* follows directly from the following diagrammatic reasoning:

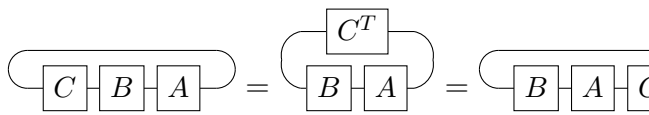
$$(A \otimes C)(B \otimes D) =$$



$$= AB \otimes CD.$$

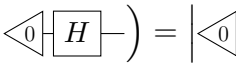
Another example is the *cyclic property of trace*, as $\mathrm{tr}(A)$ corresponds to the self-contraction:

$$\mathrm{tr}(ABC) =$$



$$= \mathrm{tr}(CAB).$$

Quantum circuits can be interpreted as tensor diagrams. Indeed, we can transform each element of a quantum circuit into a tensor with each dimension equal to 2: every single-qubit state is a tensor with a single output, and the gates are either (1, 1)- or (2, 2)-tensors. To determine the probability of a particular measurement outcome $i_1, \ldots, i_n$, by the Born rule, we need to contract (postselect) each qubit k with $\langle i_k|$ and square the norm. This corresponds to taking the modulus squared of the value of the tensor diagram. For example, $\mathrm{Prob}\left(1 \,\middle|\, \langle 0| H \right) =$

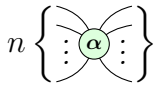


$$= 1/2.$$

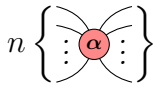
2.3 The ZX-calculus

Quantum circuits and tensor diagrams are powerful models for quantum computation. Yet, to say something meaningful about a particular circuit, one needs to evaluate the whole tensor diagram and work with large sums. Due to the rich zoo of quantum gates, it gets especially non-trivial to determine whether two quantum circuits represent the same computation. The ZX-calculus [6, 12] offers an alternative way of reasoning about quantum circuits by decomposing the gates into smaller building blocks called *spiders*. As there are only two types of spiders (green and red), it is not as difficult to find a complete set of local transformations leaving the underlying tensor unchanged.

Definition 2.3.1 (Green and Red Spiders)**.**



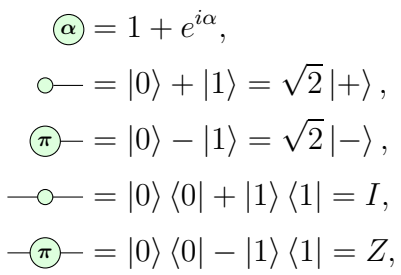
$$:= \underbrace{|0\ldots0\rangle}_{m}\underbrace{\langle 0\ldots0|}_{n} + e^{i\alpha}\underbrace{|1\ldots1\rangle}_{m}\underbrace{\langle 1\ldots1|}_{n},$$



$$:= \underbrace{|+\cdots+\rangle}_{m}\underbrace{\langle +\cdots+|}_{n} + e^{i\alpha}\underbrace{|-\cdots-\rangle}_{m}\underbrace{\langle -\cdots-|}_{n}.$$

Green and red spiders are also called Z- and X-spiders, as they represent the same tensor in the Z- and X-basis. The phase α is omitted when $\alpha = 0$.

Let us consider a few small examples of spiders:

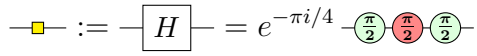


Hereafter we use the symbol $\propto$ to represent equality up to a multiplicative constant. We then have 

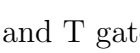
$\propto |k\rangle$ for $k = 0, 1$.

In addition to spiders, although redundant, the Hadamard box has acquired a separate representation. This is because Hadamard gates are so common that expressing them through spiders would be cumbersome.

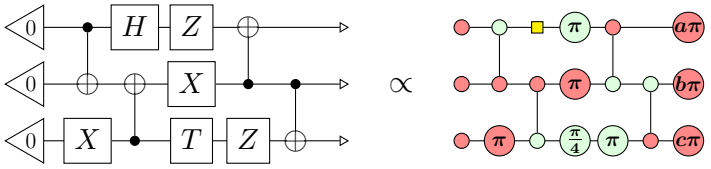
Definition 2.3.2 (Hadamard box or H-box)**.**



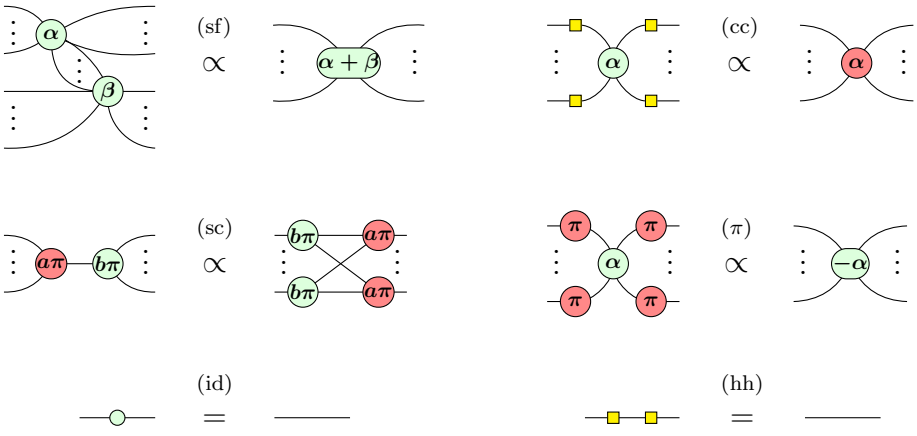
The Z-, X-spiders, and H-boxes can be joined together to form a *ZX diagram*. It turns out that it is possible to represent every quantum circuit in this way. Indeed,

- the input state is replaced by one-legged spiders, with $|0\rangle, |1\rangle$ corresponding to the red ones and $|+\rangle, |-\rangle$ corresponding to the green ones,
- the X, Z, and H gates are replaced by the two-legged π-phase spiders and H-boxes,
- the Y gate equals the composition of X and Z gates (up to a constant factor),
- the S and T gates are given by  and , respectively,
- the CNOT gate is equivalent to  ,
- the measurements in the Z or X basis are replaced by the single-legged red or green spiders, respectively, with phases $i_k\pi$ representing the outcome i_k.

For example, the circuit (2.1) gets transformed into the following ZX diagram:



Now that we have defined all the necessary components, we need to figure out how to manipulate them. Our goal is to do proofs using the ZX-calculus, such as establishing the equivalence of quantum circuits. The main recipe is to perform local transformations that leave the ZX diagram equivalent up to a constant factor, which can be computed separately. The complete set of rules is given below.

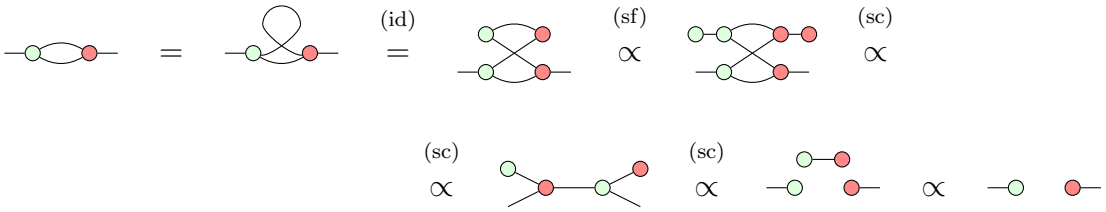


(sf) – spider fusion
(cc) – color change
(sc) – strong complementarity
(π) – π-rule
(id) – identity removal
(hh) – Hadamard cancel

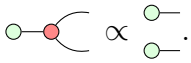
These rules, together with their colour-symmetric versions, form the ZX-calculus. Let us illustrate the application of these rules by proving the following lemma.

Lemma 2.3.1 (Hopf rule)**.**

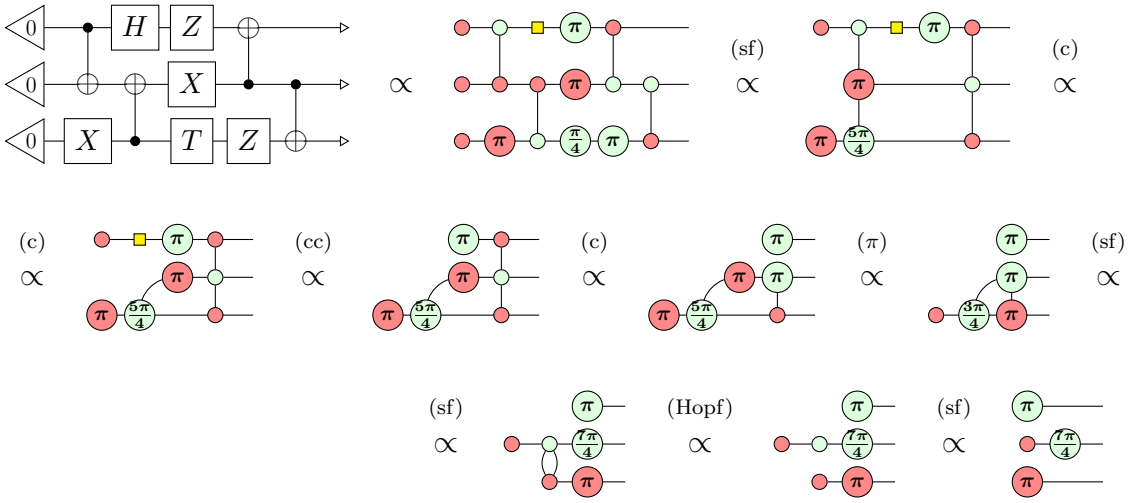
Proof.



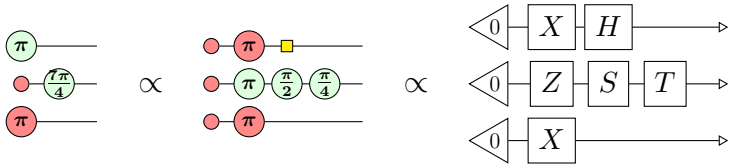
□

Note that we have used the special case of strong complementarity with zero wires on the left:  . This rule is known as just *complementarity* (c).

After a quantum circuit is turned into a ZX diagram, simplification is possible through the rules of the ZX-calculus. Let us turn to our example (2.1) and see what we can accomplish with these rules.



To transform the ZX diagram back to the circuit form, we decompose the $7\pi/4$ phase into $\pi, \pi/2, \pi/4$, and apply (sf) and (cc) to unfuse the all-zero input state:



The resulting circuit is much simpler than the original one.

2.4 Parity maps

Quantum simulation of general circuits is known to be PromiseBQP-complete, meaning that if someone manages to simulate arbitrary quantum circuits efficiently, then building quantum computers would be a waste of time. Yet, some quantum circuits are easy to simulate – for instance, the Clifford ones, which we discuss in the next chapter. In the ZX-calculus, the classes of diagrams that are in some sense easier than general ones are called *fragments*. Let us look at the phase-free fragment of ZX-calculus, i.e., at the ZX diagrams whose spiders are all phase-free. These diagrams correspond to the CNOT circuits and are characterised by parity maps.

Hereafter, we assume $\mathbb{F}_2$ is the field of two elements with addition and multiplication defined as XOR and AND, respectively. A *parity map* is a linear operator $\mathbb{F}_2^n \to \mathbb{F}_2^m$ characterised by a *parity matrix* of size $m \times n$. Let us show that the CNOT circuits (quantum circuits consisting only of CNOTs) are described by parity maps. Indeed, each CNOT gate acts as a parity map on the basis states of its qubits with the parity matrix $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, since $|x\rangle \otimes |y\rangle \mapsto |x\rangle \otimes |x+y\rangle$. Hence it represents a parity map $\mathbb{F}_2^n \to \mathbb{F}_2^n$. When we compose different CNOTs consecutively, the parity matrices get multiplied, resulting in a parity matrix of the composition. Thus, the action of the whole circuit on the basis states is described by a parity matrix. The action on all the remaining states is uniquely determined by the linearity.

Suppose we are given a parity matrix A of size $n \times n$. Does there exist a CNOT circuit equivalent to it? It turns out it exists if and only if A is invertible. Indeed, if A is not invertible, then the map between the basis states is not injective and thus cannot be unitary. If A is invertible, then we can reduce A to the identity by performing elementary row operations:

$$E^{i_k j_k} \dots E^{i_1 j_1} A = I,$$

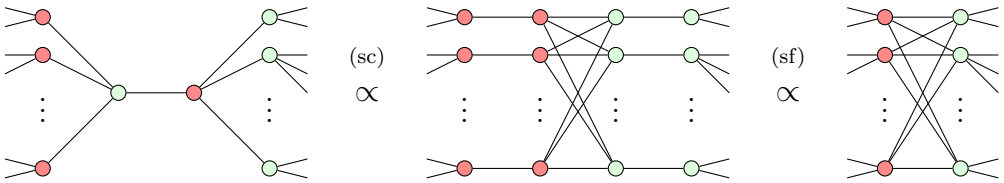
where E^{ij} corresponds to the addition of the j-th row to the i-th row. As $E^{i_t j_t}$ are self-inverse, multiplying both sides by $E^{i_1 j_1} \dots E^{i_k j_k}$ we get

$$A = E^{i_1 j_1} \dots E^{i_k j_k}.$$

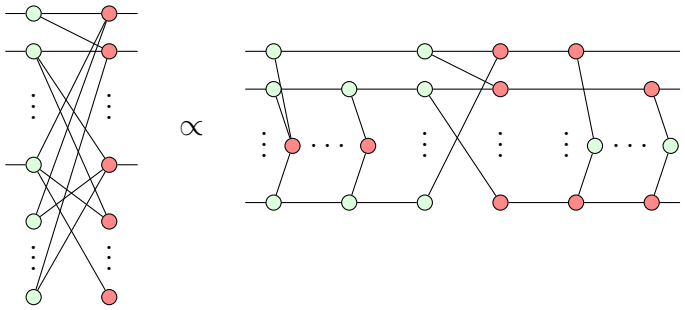
Since $E^{i,j}$ is the parity matrix of the CNOT gate applied to the qubits i, j, it follows that A can be constructed as the composition of CNOTs over $(i_1, j_1), \dots, (i_k, j_k)$.

To sum up, each CNOT circuit is represented by an invertible parity map, and each invertible parity map corresponds to at least one CNOT circuit. But what is the simplest form to which we can reduce its ZX diagram?

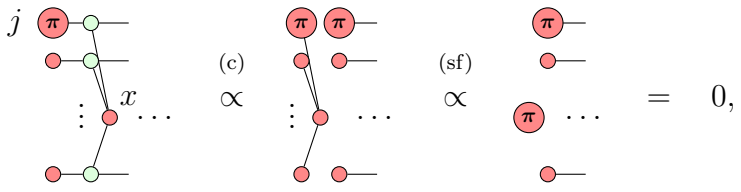
Clearly, the initial ZX diagram of a CNOT-only circuit (without the input state) is phase-free. Let us fuse all the same-colored spiders so that each edge joins different colors. We can assume each input is adjacent to a green spider by adding them using the (id) rule, if necessary. Similarly, each output is adjacent to a red spider. Now take a pair of connected *internal spiders* (not adjacent to inputs and outputs) and apply strong complementarity to them:



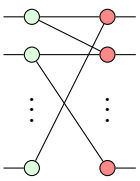
Note that the overall number of spiders is reduced by 2. Keep doing this procedure until every internal spider is connected to *boundary* (non-internal) spiders only. Since the boundary spiders on the left are green, and the ones on the right are red, our ZX diagram has the form



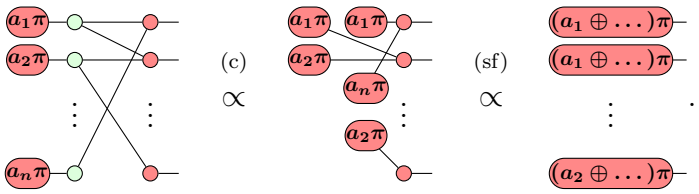
We claim that, in fact, there are no internal red spiders on the left. Assuming the contrary, let x be the first such spider and consider the set of qubits S to which x is connected. Pick $j \in S$ and act with our circuit on the input state $a_i = [i = j]$:



contradicting the unitarity of our circuit. Analogously, by considering the transpose of our map, we argue that there are no internal green spiders on the right. Hence the ZX diagram takes a much simpler form:



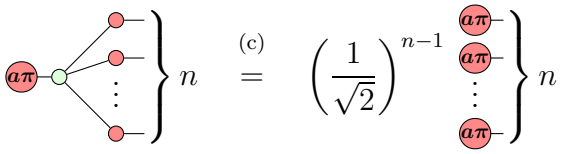
Let us interpret this diagram as a parity map. Plugging in the basis state $|a_1 \dots a_n\rangle$ we have



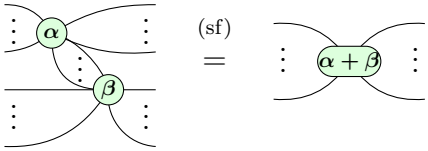
Thus, we can interpret each green spider as COPY and each red spider as XOR. The parity matrix A then corresponds to the biadjecency matrix: $A_{ij} = 1$ iff there is an edge between the i-th output and the j-th input.

Let us now consider arbitrary parity maps from $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$. Suppose we are given a parity matrix A of size $m \times n$ and want to build the corresponding ZX diagram with n inputs and m outputs. Using the interpretation above, for each $A_{ij} = 1$, we join the i-th red spider on the right and the j-th green spider on the left. We know that this diagram is equivalent to the parity map up to a multiplicative scalar. But can we compute this factor exactly?

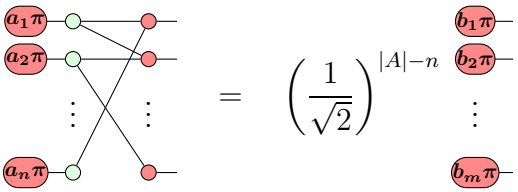
We can do this by first computing the scalar in the complementarity rule. Direct calculation yields



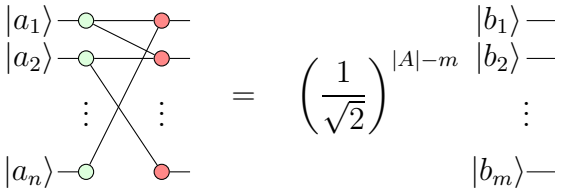
It is also easy to check that spider fusion has scalar 1:



Hence we have



where $b := Aa$ and $|A|$ denotes the number of ones in A. Recall that the red spiders are basis states times $\sqrt{2}$, thus the resulting scalar is $(1/\sqrt{2})^{|A|-m}$:



Chapter 3

Current Simulation Methods

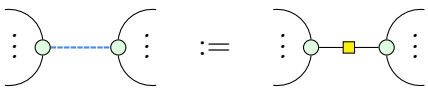
3.1 Clifford circuits

Clifford circuits form a very important subclass of quantum circuits. These circuits are composed entirely of *Clifford gates* – namely the Hadamard (H), S, and CNOT gates. Despite their limited computational power compared to universal quantum circuits, Clifford circuits are widely used in quantum error correction, stabiliser codes, and classical simulation. In this section, we use ZX-calculus to prove the Gottesman-Knill theorem, which states that Clifford circuits can be classically simulated in polynomial time. By reducing a Clifford ZX diagram to the GSLC normal form, we establish a normal form for Clifford circuits and estimate the computational complexity of their classical simulation.

When viewed through the lens of ZX-calculus, Clifford circuits correspond to the *Clifford fragment*, where all phases are integer multiples of $\pi/2$. Indeed, the S gate creates a $\pi/2$-phase green spider, and all other gates are phase-free. Note that the Clifford fragment is the generalisation of the phase-free fragment discussed in Chapter 2.

Let us introduce a few necessary notions and present some basic properties.

Definition 3.1.1. *When two spiders in the ZX diagram are connected via a Hadamard, we illustrate this as a blue dotted line:*



We call such a connection a Hadamard edge.

Lemma 3.1.1. *A pair of parallel Hadamard edges cancels:*

Proof. First, apply the (cc) rule to change the color of the left spider. Then unfuse the green and red spiders, apply the Hopf rule in the middle, and finally fuse these spiders back and apply the (cc) rule again. For more details, see [12, Lemma 5.1.5]. □

Lemma 3.1.2. *Hadamard self-loops are equivalent to adding the π phase:*

Proof. See [12, Lemma 5.1.6]. □

Definition 3.1.2. *A ZX-diagram is* graph-like *when*

- *Every spider is a Z-spider.*
- *Spiders are only connected via Hadamard edges.*
- *There are no self-loops or parallel edges.*
- *Every Z-spider is connected to at most one input and at most one output.*
- *Every input and output wire is connected to a Z-spider.*

Proposition 3.1.1. *An arbitrary ZX diagram can be reduced efficiently to a graph-like diagram.*

Proof. We outline the main steps of the algorithm; see [12, Proposition 5.1.8] for a detailed explanation.

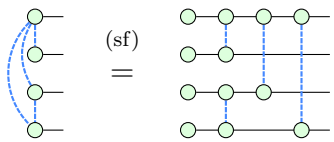
1. Convert all X-spiders to Z-spiders using the (cc) rule.
2. Cancel all pairs of adjacent Hadamards using the (hh) rule.
3. Fuse all spiders using (sf) until there are no neighbouring same-colored spiders left.
4. Remove parallel Hadamard edges using Lemma 3.1.1.
5. Remove self-loops and Hadamard self-loops using Lemma 3.1.2.
6. Add Z-spiders and Hadamards using (id) and (hh) in reverse, so that every input and output is directly connected to a Z-spider and no Z-spiders are connected to multiple inputs/outputs.

□

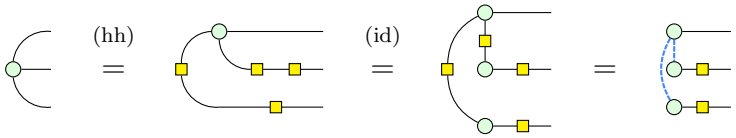
Definition 3.1.3. *A graph-like diagram is a* graph state *when*

- *it has no inputs,*
- *every spider is connected to an output,*
- *and all phases are zero.*

We present an example of a graph state and show how to decompose it into a CZ-only circuit, where vertical Hadamard edges represent the CZ gates:



Many quantum states are not exactly graph states, but are very close to them. For instance, the *GHZ state* is given by



This motivates the following definition.

Definition 3.1.4. *A graph state with local Cliffords (GSLC) is a graph state to which some single-qubit unitaries have been applied on its outputs.*

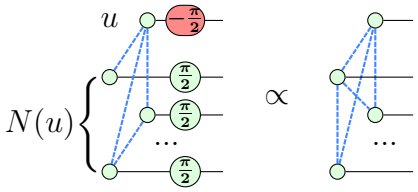
At first glance, GSLC is insufficient to capture the variety of Clifford circuits, since GSLC does not have any inputs. It turns out one can easily reduce Clifford circuits to states by bending the input wires so that the resulting $2n$-qubit state is still Clifford. Hence, we can work with states instead of unitaries without loss of generality.

Ideally, we would like to transform a Clifford state in the graph-like form (due to Proposition 3.1.1) to a graph state. However, graph-like diagrams have some internal spiders that we need to eliminate. A natural way to accomplish this is by applying some local transformations to the graph that follow from the ZX-calculus. Let us now briefly state these transformations, which are proven in [12, Section 5.2]. We begin by studying the operations on graph states.

Definition 3.1.5. *Let G be a graph, and u be a vertex in G. The* local complementation about u*, written as $G \star u$, is the graph with the same vertices and edges as G, except that the neighbourhood of u is complemented: two neighbours v, w are connected in $G \star u$ if and only if they are not connected in G.*

Proposition 3.1.2. *Graph states G and $G \star u$ are equivalent up to single-qubit unitaries applied on u and its neighbours.*

More precisely, we need to apply the red $-\pi/2$-spider to u and green $\pi/2$-spiders to the neighbourhood $N(u)$ so that the diagrams become equivalent. For example,



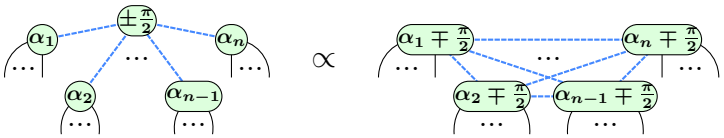
It turns out local complementation is especially useful when acting on a pair of connected vertices.

Definition 3.1.6. *Let G be a graph and let u, v be a pair of connected vertices. We define* pivot of G along uv*, written as $G \wedge uv$, as the graph $G \star u \star v \star u$.*

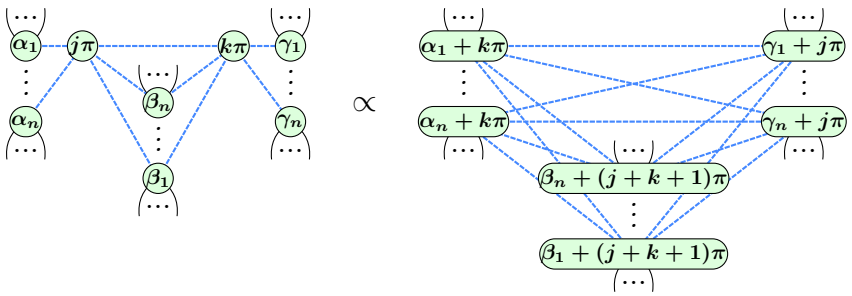
It is quite easy to check that the ordering of u and v does not matter, meaning that $G \star u \star v \star u = G \star v \star u \star v$. Schematically, pivot swaps u, v and complements the edges between the neighbourhoods $A := N(u) \cap N(v)$, $B := N(u) \setminus (N(v) \cup \{v\})$, and $C := N(v) \setminus (N(u) \cup \{u\})$:

Let us now state the rules for removing internal spiders in a graph-like diagram. We present two types of transformations that allow us to remove an internal $\pi/2$-phase spider or delete a pair of connected $0/\pi$-phase spiders.

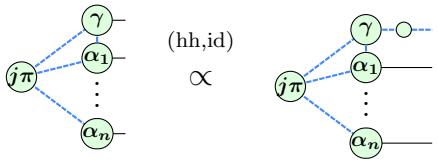
Proposition 3.1.3. *The following local complementation simplification holds:*



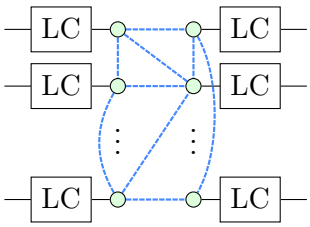
Proposition 3.1.4. *The following pivot simplification holds:*



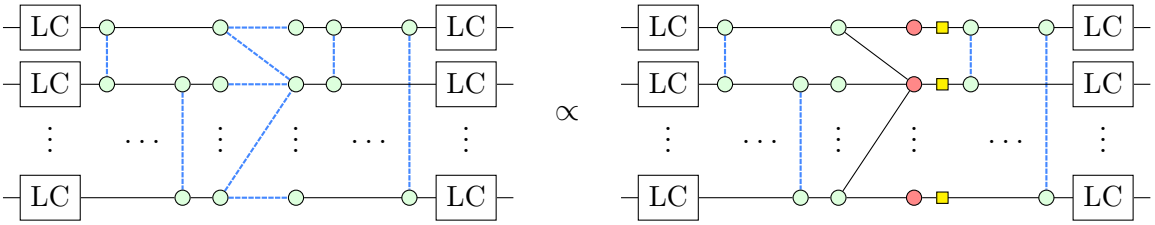
Let us explore what we can achieve by these two simplifications. First of all, if a ZX diagram has an internal $\pm\pi/2$-phase spider, we can delete it by applying local complementation. We keep doing this until there are no internal $\pm\pi/2$-phase spiders left. Afterwards, if there exists a pair of connected $0/\pi$-phase spiders, we remove them using pivot simplification. We end up with only internal $k\pi$-spiders that are not connected to each other. We can get rid of them as well by making an additional boundary spider internal:



If $\gamma = k\pi$, then we can remove both internal spiders with the pivot simplification. If $\gamma = \pm\pi/2$, then first apply local complementation on γ, after which $j\pi$ will acquire a $\pm\pi/2$ phase so that it can also be eliminated by local complementation. Finally, the phases of the boundary spiders can be pushed to the outputs using spider unfusion. As a result, a Clifford state is turned into GSLC. Equivalently, the original Clifford circuit is transformed to the *GSLC normal form*, defined as follows:



Let us decompose this form even further. We can unfuse the connections between spiders in the same layer so that they become the CZ gates, and change the colors of the spiders on the right side in the middle part of the diagram:



Observe that the middle part becomes a parity map, which we already considered in Chapter 2 and know it corresponds to a CNOT-only circuit. Since single-qubit Cliffords are generated by Hadamards and S gates, we obtain the following normal form for Clifford circuits:

$$\mathrm{Had}, \mathrm{S} - \mathrm{CZ} - \mathrm{CNOT} - \mathrm{Had} - \mathrm{CZ} - \mathrm{Had}, \mathrm{S}.$$

Let us finally discuss the *strong simulation* of Clifford circuits: given an n-qubit input, a circuit consisting of m gates, and an n-qubit output, compute the probability of measuring this output. By the Born rule, this reduces to computing the value of the underlying Clifford ZX diagram with no inputs and outputs. The algorithm follows completely from our reasoning above:

1. Convert the ZX diagram to a graph-like diagram.

2. Apply local complementation and pivot simplifications to reduce the graph-like diagram to the GSLC normal form, keeping track of the scalar C. Since there are no inputs and outputs, the GSLC form we obtained is empty, so return C.

We finally estimate the computational complexity of Clifford circuit simulation. Let $N := n + m$ be the size of the Clifford circuit. Note that its ZX diagram has $O(N)$ spiders. The first step takes $O(N^2)$ time, see the proof of Proposition 3.1.1. The most expensive part is step 2, when we apply the local transformations. Since each application of local complementation and pivoting decreases the number of spiders by at least 1, we perform $O(N)$ such operations. Each operation modifies the edges in the neighbourhood, taking $O(N^2)$ graph operations. Hence, the total complexity is $O(N^3)$. It is possible to reduce the complexity to $O(mn^2)$ – see [12, Section 5.4.1].

3.2 Stabiliser decompositions

Throughout our thesis, we will focus on the strong simulation of quantum circuits, which involves computing the probability of a particular outcome. This task is harder than the weak simulation, where the goal is to sample efficiently from the distribution of outcomes. Nevertheless, strong simulation is a much more natural problem within the framework of ZX-calculus.

An efficient simulation algorithm of general quantum circuits would immediately imply the absence of quantum supremacy. It is widely believed that quantum computers have more computational power than classical ones, so it is likely impossible to simulate an arbitrary quantum circuit in polynomial time. Much effort has already been spent on trying to understand where the quantum advantage comes from. One might argue it comes from entanglement, but the Gottesman-Knill theorem contradicts this intuition: Clifford states can be highly entangled but are classically simulable in polynomial time.

To better understand quantum advantage, let us ask ourselves a question: is it possible to express an arbitrary unitary map $\mathbb{C}^{2^n} \to \mathbb{C}^{2^n}$ using only finitely many

primitive operations (gates)? Of course, if we are interested in implementing this map exactly, then by the counting argument, we would require a continuous family of gates, as there are only countably many different quantum circuits when the gate set is finite. Yet, in practice, we are typically interested in approximations to the unitary map up to some additive error ε. The gate set allowing such approximation for any unitary map and any $\varepsilon > 0$ is called *universal*. One example of the universal gate set is $\{\mathrm{CNOT}, H, T\}$, which we will use throughout our dissertation. Note that in addition to Clifford gates, it introduces the T *gate*: $T = \sqrt{S}$. In ZX-calculus, the T gate is given by the $\pi/4$-phase spider. Hence, it suffices to consider the *Clifford + T fragment* of the ZX-calculus, where all phases are integer multiples of $\pi/4$.

The main idea behind the stabiliser decomposition approach is to represent the output state as a linear combination of Clifford states:

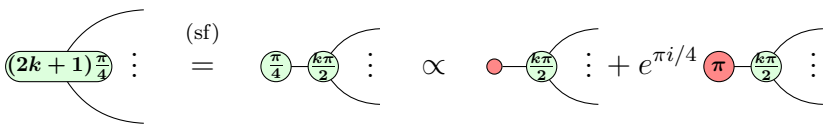
$$|\psi\rangle = \sum_{i=1}^{k} a_i \, |\phi_i\rangle \, .$$

The minimal value of k is called the *stabiliser rank* of $|\psi\rangle$. We can then evaluate the probability of an outcome x by

$$\langle x|\psi\rangle = \sum_{i=1}^{k} a_i \, \langle x|\phi_i\rangle \, ,$$

which costs $\tilde{O}(k)$ operations. Therefore, it is important to find such decompositions with the lowest possible k. Although it is widely believed that the stabiliser rank is exponential for general circuits, in numerous practical cases it yields an efficient simulation algorithm.

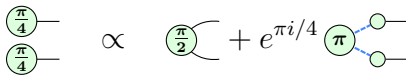
There is an elegant way to represent this method in the ZX formalism. Ideally, if all phases were Clifford, we would run the simulation algorithm discussed in Section 3.1. The only obstacles are the *T-phase spiders*, i.e., the nodes whose phases are odd multiples of $\pi/4$. We can get rid of a single T-phase spider by unfusing a T-state and decomposing it into two terms:



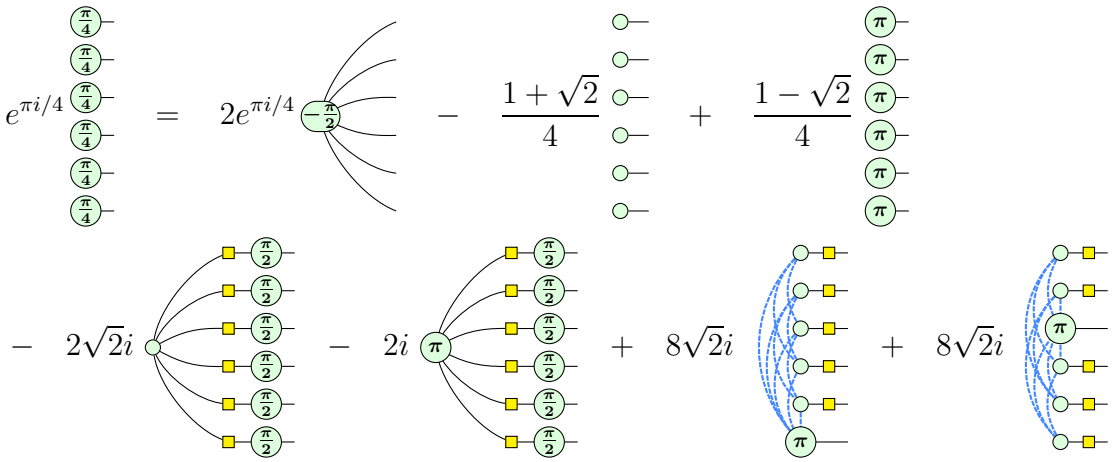
Denote the total number of T-phase spiders by t. With this strategy, the simulation complexity is $\tilde{O}(2^t)$, because we split into two branches having one T-phase spider less – in the end, we obtain a decomposition with $k = 2^t$ terms. This is more efficient

compared to the naive approach, which has complexity $\tilde{O}(2^n)$, only if t is relatively small. Can we do better than this?

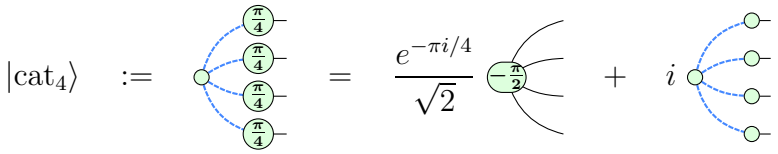
For example, we can delete two T-states at once:

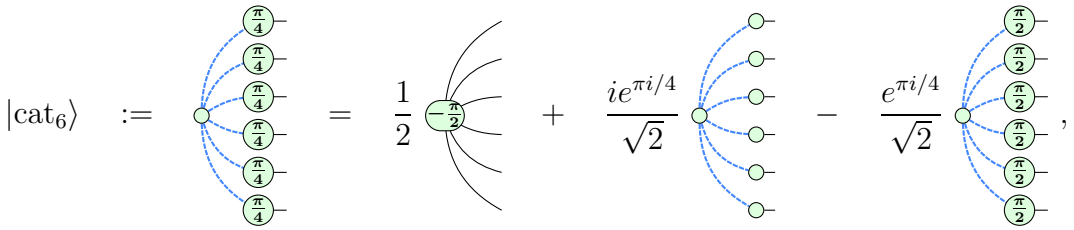


This gives a decomposition into $2^{\alpha t}$ terms where $\alpha = 0.5$. In [4, Eq. 11], Bravyi, Smith, and Smolin gave a decomposition of six T-states into seven stabiliser terms, resulting in $\alpha = \log_2(7)/6 \approx 0.468$:



We are not limited to decomposing a collection of T-states. For instance, it is much more efficient to decompose the *cat states*:

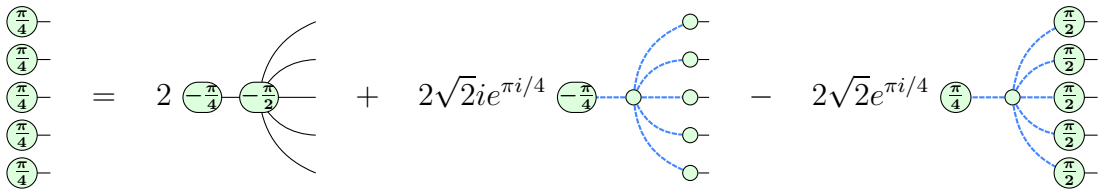




giving $\alpha = 0.25$ and $\alpha \approx 0.264$, respectively. Hence, if we have an internal Clifford spider with degree 4 or 6 (whose neighbours are then non-Clifford), we can efficiently eliminate them all. Still, we cannot use these decompositions to expand other $|\mathrm{cat}_n\rangle$.

Finally, allowing the terms to be not entirely Clifford, we can decompose five

T-states into three terms, each having one T-state, resulting in $\alpha \approx 0.396$:



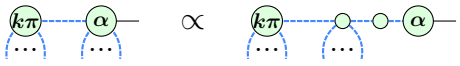
This decomposition is known as *Magic-5*. For a more detailed review of this approach, see [3, 5, 11, 13, 14].

3.3 Graph-based techniques

After turning a ZX diagram into a graph-like diagram, we have three types of spiders: with phase $k\pi$ (aka Pauli spider), with phase $(k + 1/2)\pi$, and the T-phase spiders. As discussed in Section 3.1, we have two kinds of simplifications at our disposal: local complementation and pivoting. The former allows us to get rid of an internal $(k+1/2)\pi$-spider, and the latter enables us to remove two adjacent internal $k\pi$-phase spiders. By applying these rules, we can obtain a smaller graph-like diagram such that

1. each internal spider is either Pauli or T-phase, and
2. there are no adjacent internal Pauli spiders.

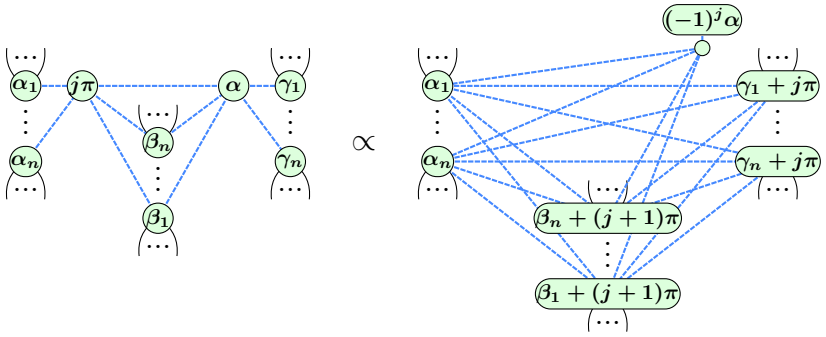
It turns out we can delete an internal $k\pi$-spider which is connected to a boundary spider with an arbitrary phase. The key idea is to use the identity rule and spider unfusion to transform the boundary spider into an internal one:



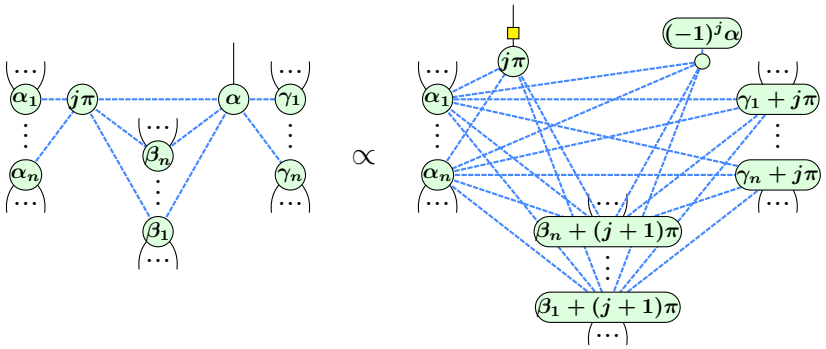
Now, the pivot simplification is possible on the emerged pair of internal Pauli spiders. Note that we will get a $k\pi$-spider connected to the same neighbourhood as the original Pauli spider. However, the α-phase spider is now directly connected to an output and can therefore be discarded for the purposes of diagram simplification.

Suppose we have an internal Pauli node connected to a T-phase node. If the latter is internal, then applying spider unfusion, the identity rule, and pivoting, we get the

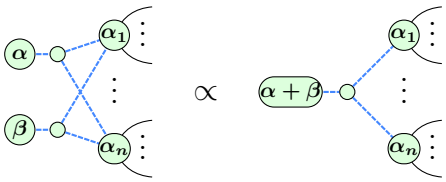
following rule:



If the non-Clifford spider is boundary, then the rule is

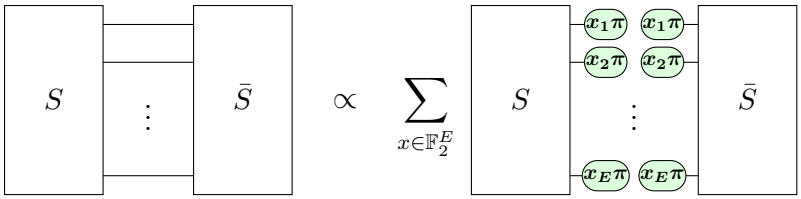


The structure on the top-right – a single-legged spider connected to a phaseless node by a Hadamard edge – is called a *phase gadget*. Phase gadgets have a nice merging property, called *gadget-fusion*:



Let us turn to another approach called *graph cuts decompositions*. Suppose we are fortunate enough, and our ZX diagram splits into two disconnected components, S and $\bar{S}$. In this case, we can independently simulate S and $\bar{S}$ and then multiply their amplitudes. Assuming the simulation over n vertices costs $\tilde{O}(2^{\alpha n})$ for some $\alpha > 0$, the complexity of our computation is then $\tilde{O}\left(2^{\alpha|S|} + 2^{\alpha(n-|S|)}\right)$, which is much better than $\tilde{O}\left(2^{\alpha|S|} \cdot 2^{\alpha(n-|S|)}\right)$ if $|S|, |\bar{S}|$ are sufficiently large. Yet, we are typically not that lucky and can only hope for a cut $(S, \bar{S})$ which separates E edges for small enough E. Then, we can use the following decomposition to bring the simulation cost down

to $\tilde{O}\left(2^E \cdot (2^{\alpha|S|} + 2^{\alpha|\bar{S}|})\right)$:



The combination of this strategy with stabiliser decompositions is a powerful method for quantum circuit evaluation, though it introduces a few challenges. First, how to find a good cut $(S, \bar{S})$ and what measure can be used to assess its quality? One way to do this is by considering the T-count of S and $\bar{S}$. Improving this strategy in various practical cases is an active topic of research, see [17, 20, 21, 22].

3.4 Advanced approaches

The graph transformations discussed in the previous section may yield different graph-like diagrams depending on the order in which the rules are applied. As a result, the cost of the calculations depends heavily on the choice of the rule at each step. Several strategies exist that guide optimal rule selection, as described in [17]. The majority of these strategies aim to minimize the *cost function* of the ZX diagram, which estimates roughly the number of operations required to compute the underlying tensor diagram. For example, we can set it to $f(g) := 2^{|E(g)|}$, as this corresponds to the number of terms in the sum.

The idea behind the greedy approach is to choose the rule that minimises the cost function right after its application. We store the best ZX diagram according to $f(g)$ and update it after exploring each possible local transformation. We finish when every rule increases the cost function.

A better approach is based on simulated annealing. This heuristic is inspired by the physical process of cooling a material down to its minimal energy. At each step, we apply a random rule to the current diagram g, obtaining a new diagram g'. If $f(g') < f(g)$, then g' is chosen as the new candidate. Otherwise, it is selected with probability $\exp\left(-\{f(g') - f(g)\}/T\right)$, where T denotes the temperature that decreases during our process according to the cooling schedule. This approach is superior to the greedy strategy because it is not restricted to the best options after each step and better minimises the cost in the long term.

Another strategy utilises genetic algorithms, which simulate natural selection in biology. We store the population of the diagrams (aka organisms) and model the

birth of the new generation with the *crossover* operation. This operation takes a single ZX diagram as input and produces a new ZX diagram by applying a random rule. Then, we discard the worst diagrams according to the cost function, so that the population size remains constant. We iterate these steps until the desired optimal solution is achieved.

Finally, let us briefly discuss the approach to T-count minimisation used by Google [18]. First, they isolate the non-Clifford part within a CNOT+T circuit defined on a larger qubit register [7]. Then, they compute its *signature tensor*, which reflects the information about the phase polynomial of the CNOT+T circuit. They proceed by computing the *Waring decomposition* of the signature tensor, i.e., the decomposition into the sum of rank-1 tensors with the fewest possible terms. They use reinforcement learning to minimise the rank by introducing the *TensorGame*, where moves correspond to subtracting rank-1 terms, and the goal is to achieve zero as quickly as possible. The rank found by this method directly corresponds to the T-count of the new circuit synthesised from the decomposition.

Chapter 4

Mathematical Background

4.1 Linear algebra over $\mathbb{F}_2$

In this section, we outline the main notions and methods from linear algebra over $\mathbb{F}_2$ that are essential for presenting our strategy clearly and concisely. We will discuss Gaussian elimination, the row echelon form of a matrix, and the notion of generalised inverse, which is the analogue of a pseudoinverse matrix over finite fields.

4.1.1 Gaussian elimination

Let $A \in \mathbb{F}_2^{n \times m}$. We are interested in the simplest form to which we can transform A using primitive row operations. These operations include:

- Add the j-th row to the i-th row, and
- Swap rows i, j.

Note that the application of each operation corresponds to the multiplication of the matrix A on the left. In the case of row additions, we multiply A by the matrix $E_{i,j}$ with 1-s on the diagonal and at (i, j), and for the swap we multiply by the permutation matrix $P_{i,j}$. Hence, we are decomposing $A = R^{(k)} \dots R^{(1)} A'$, where $R^{(i)}$ is one of the options above, and aim to make A' as simple as possible. The best form for A' turns out to be the *reduced row echelon form*, as defined below. Let us define the row echelon form first.

Definition 4.1.1. *A matrix is said to be in the* row echelon form *if it has the form*

$$\left[\begin{array}{c|c|c|c|ccc|c|c} 0 & 1 & * & * & * & \dots & * & * \\ 0 & 0 & 0 & 1 & * & \dots & * & * \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & 1 & * \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \end{array}\right].$$

The highlighted columns, having the leading 1-s, are called pivot columns.

Definition 4.1.2. *A matrix is said to be in the* reduced row echelon form *if it has the form*

$$\left[\begin{array}{c|c|c|c|ccc|c|c} 0 & 1 & * & 0 & * & \dots & 0 & * \\ 0 & 0 & 0 & 1 & * & \dots & 0 & * \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & 1 & * \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \end{array}\right].$$

Proposition 4.1.1. *Every matrix $A \in \mathbb{F}_2^{n\times m}$ can be transformed to the reduced row echelon form by primitive row operations.*

Proof. Let us transform A to the row echelon form first. If A is zero, then all is done. Let j_0 be the first nonzero column of A, and let i_0 be such that $A_{i_0,j_0} = 1$. If $i_0 \neq 1$, then swap the first row and the i_0-th row so that $A_{1,j_0} = 1$. For each i such that $A_{i,j_0} = 1$, add row 1 to row i. Note that we have $A_{i,j_0} = 0$ for each $i \geq 2$ and $A_{i,j} = 0$ for each $j < j_0$. Apply the same procedure to the submatrix $A_{i_0+1:,j_0+1:}$.

To transform A to the reduced echelon form, iterate over the pivot columns of A from right to left and set everything above the leading 1 to zero by performing row additions from bottom to top. □

The process described above is known as Gaussian elimination. If we also allow primitive column operations, then we can reduce A even further to the following form:

$$\Sigma_A = \left(\begin{array}{c|c} I_r & 0 \\ \hline 0 & 0 \end{array}\right),$$

where r is the rank of A, and I_r denotes an identity $r \times r$ matrix. Indeed, first move the pivot columns to the front using swaps, and then zero out everything else on the

right with column additions. As column operations correspond to multiplying A on the right, we can express A in the following way:

$$A = R^{(k)} \dots R^{(1)} \cdot \Sigma_A \cdot C^{(1)} \dots C^{(l)} = R_A \Sigma_A C_A,$$

where R_A and C_A are invertible $n \times n$ and $m \times m$ matrices. Note that the computational complexity of Gaussian elimination is $O(nm \cdot \min(n, m))$, and the cost of computing the decomposition above is $O((n+m)^2 \cdot \min(n, m))$.

4.1.2 Rank factorisation and generalised inverse

Definition 4.1.3. *Let $A \in \mathbb{F}_2^{n \times m}$. Rank of A is defined as the minimal r such that there exist $B \in \mathbb{F}_2^{n \times r}, C \in \mathbb{F}_2^{r \times m}$ with $A = BC$.*

Let us see how to obtain this decomposition algorithmically, given $A \in \mathbb{F}_2^{n \times m}$. First, compute the decomposition $A = R_A \Sigma_A C_A$ from above. Let r be the number of 1-s in Σ_A. Set B to the first r columns of R_A and C to the first r rows of C_A. Since Σ_A has I_r in its top left corner and is zero everywhere else, we have

$$A = \left(\begin{array}{c|c} B & * \end{array} \right) \left(\begin{array}{c|c} I_r & 0 \\ \hline 0 & 0 \end{array} \right) \left(\begin{array}{c} C \\ \hline * \end{array} \right) = BC.$$

The complexity of this routine is $O((n+m)^2 r)$.

In the general case, an $n \times m$ matrix A may not be invertible; nevertheless, we would like to have some analogue of the inverse matrix with the same properties that A^{-1} has when it exists. If $A \in \mathbb{C}^{n \times m}$, then the *pseudoinverse* (aka Moore-Penrose inverse) does the job. However, we are working over $\mathbb{F}_2$ and the pseudoinverse matrix might not exist. It turns out the solution is the *generalised inverse*, defined as follows.

Definition 4.1.4 (generalised inverse)**.** *Given $A \in \mathbb{F}_2^{n \times m}$, $A^g \in \mathbb{F}_2^{m \times n}$ is called the* generalised inverse *of A if it satisfies the following condition:*

$$AA^gA = A.$$

Note that the generalised inverse is not necessarily unique. To compute a valid A^g from A efficiently, we utilise the decomposition $A = R_A \Sigma_A C_A$ once again and take $A^g = C_A^{-1} \Sigma_A^T R_A^{-1}$ (recall that R_A, C_A are invertible). Then we have

$$AA^gA = R_A \Sigma_A C_A C_A^{-1} \Sigma_A^T R_A^{-1} R_A \Sigma_A C_A = R_A \Sigma_A \Sigma_A^T \Sigma_A C_A = R_A \Sigma_A C_A = A.$$

4.2 Rank-width

Rank-width was introduced by Oum and Seymour [8] as the replacement for clique-width. Similarly to clique-width, rank-width measures the difficulty of decomposing a graph into a tree-like structure. It uses the notions of *branch-decomposition* and *cut-rank*, defined as follows.

Definition 4.2.1. *Let M be a finite set, called the* ground set *of the decomposition. A* branch-decomposition *is a pair (T, L), where T is a tree with each non-leaf node having degree 3, and L is the bijection from M to the leaves of T.*

Branch-decompositions are usually applied to graphs, so the ground set is often the vertex set of a graph. An example of a graph and a branch-decomposition is given below.

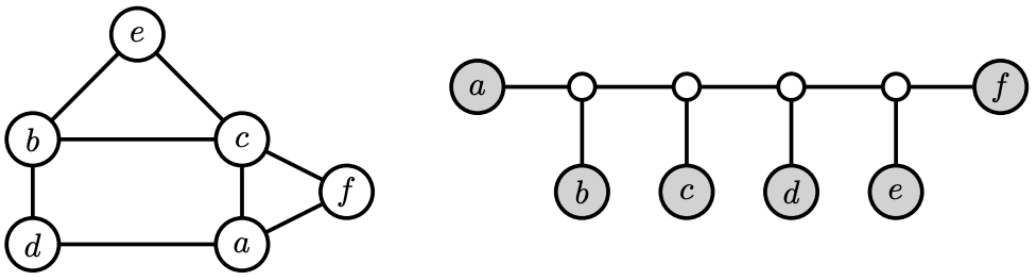


Figure 4.1: [16, Figure 2.1] A graph and a possible branch-decomposition with the vertices of the graph as the ground set.

Definition 4.2.2. *Given an undirected graph $G = (V, E)$ and a subset $X \subset V$, the* cut-rank $\rho_G(X)$ *is the rank of the adjacency matrix between X and $V \setminus X$, viewed as a matrix over $\mathbb{F}_2$.*

For example, if $G = K_4$ and $X = \{1, 2\}$, then the adjacency matrix between X and $V \setminus X$ is $\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$, hence $\rho_G(X) = 1$. If $X = \varnothing$ or $X = V(G)$, then $\rho_G(X)$ is zero, since the adjacency matrix is empty.

If we combine the two notions above, we get the definition of the *rank-decomposition* and *rank-width*.

Definition 4.2.3. *Let $G = (V, E)$ be an undirected graph. A* rank-decomposition *is a branch-decomposition with the ground set V, where each edge of the tree is assigned the cut-rank $\rho_G(X)$ for $(X, V \setminus X)$ being the partition of the leaves induced by the removal of this edge. The* width *of the rank-decomposition is the maximum cut-rank over all edges of the tree.*

Definition 4.2.4. *The* rank-width *of an undirected graph G, written as* $\mathrm{rw}(G)$*, is the minimum width of its rank-decomposition.*

Here is an example of a graph and its optimal rank-decomposition, showing the graph has rank-width 2:

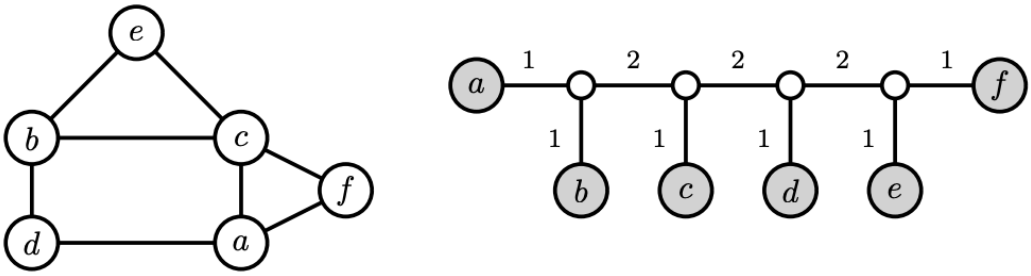


Figure 4.2: [16, Figure 3.2] A graph and an optimal rank-decomposition of width 2.

In fact, the rank-decomposition above is *linear*, because the cuts induced by the internal edges have a linear structure. Formally,

Definition 4.2.5. *A rank-decomposition is called* linear *if its non-leaf nodes form a bamboo. Consequently, a linear rank-decomposition is given by the ordering of the vertices of the graph, with the cut-ranks taken between a prefix and a suffix of the ordering.*

Let us discuss the motivation for applying rank-decompositions to ZX diagram evaluation within our approach. Since each edge of the tree introduces a vertex cut of the diagram, we can use the graph-cuts decomposition technique from Section 3.3 for simulation. The key feature of our approach is that we are utilising the cut-rank of the partition rather than the number of edges between the parts. Indeed, we can factorise the parity map between the parts into two factors, with the intermediate dimension r equal to the cut-rank. Graphically, we have two parity maps joined by r parallel wires, so in principle, the simulation cost can be reduced to 2^r.

Can we compute the rank-width efficiently? In [9], Oum shows that computing rank-width exactly is NP-hard, and the decision problem of determining whether the rank-width is at most k, is NP-complete. So, we can only hope to compute the rank-width approximately. In [2], there is an attempt to calculate an upper and a lower bound for the rank-width of a given graph. The upper bound algorithm iteratively improves the decomposition using "a mix of greedy and random decisions". A better approach is described in [16], which is based on simulated annealing. As part of our work, we integrated it into QuiZX, a Rust library for quantum circuit optimisation.

4.3 GFlow

In this section, we introduce the notions of gflow and extended gflow that came from *measurement-based quantum computing* (MBQC). They provide conditions under which a measurement pattern on a graph state can be executed deterministically. This is especially useful for designing fault-tolerant protocols and compiling MBQC algorithms from abstract circuits. Extended gflow, a refinement of gflow, applies to cases when measurements are restricted to Pauli planes (XY, YZ, or XZ). Also, extended gflow is particularly useful for extracting circuits back from a ZX diagram [1, 19]. Indeed, the existence of extended gflow, unlike gflow, is preserved during the rewrite rules of the ZX-calculus.

Both these flows operate on a graph with inputs and outputs, which is called an *open graph*.

Definition 4.3.1. *An open graph is a triplet (G, I, O), where $G = (V, E)$ is an undirected graph, and $I, O \subset V$ are respectively called input and output vertices.*

Typically, a flow consists of a strict partial order $\prec$ over the vertices, corresponding to the measurement order ($i \prec j$ if i is measured before j), and a function $g\colon V(G) \setminus O \to \{0,1\}^{V(G)\setminus I} \setminus \{\varnothing\}$ that associates each non-output vertex v with its correction set, to handle the outcomes after measuring v. For $S \subset V(G)$, denote the *odd neighbourhood* of S by

$$\mathrm{Odd}(S) := \{u \in V(G)\colon |N(u) \cap S| \equiv 1 \pmod 2\}.$$

Definition 4.3.2 (gflow)**.** *$(g, \prec)$ is a* gflow *of an open graph (G, I, O), if and only if*

1. *if $j \in g(i)$ then $i \prec j$,*
2. *if $j \in \mathrm{Odd}(g(i))$ then $j = i$ or $i \prec j$,*
3. *$i \in \mathrm{Odd}(g(i))$.*

Informally, $g(u)$ contains the vertices after u such that there is an odd number of edges between $g(u)$ and u, and there is an even number of edges between $g(u)$ and any vertex before u. The visualisation of this property is given in Figure 4.3.

Let us give the formal definition of the extended gflow, which operates on a *labelled open graph*: a tuple (G, I, O, λ), where $\lambda\colon V(G) \setminus O \to \{\mathrm{XY}, \mathrm{XZ}, \mathrm{YZ}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}\}$ is the labelling function assigning a measurement plane or Pauli to each non-output vertex.

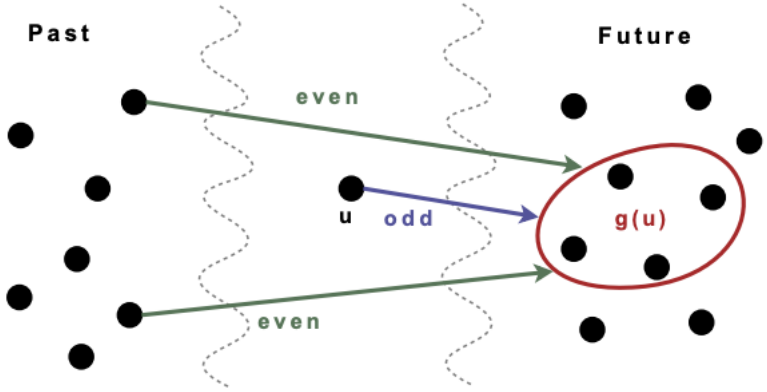


Figure 4.3: [15, Figure 2] A graphical interpretation of gflow $(g, \prec)$.

Definition 4.3.3 (Extended gflow)**.** *Given a labelled open graph* (G, I, O, λ) *such that* $\lambda(u) \in \{\mathrm{XY}, \mathrm{XZ}, \mathrm{YZ}\}$ *for each* $u \in V(G) \setminus O$*, an* extended gflow *is a tuple* $(g, \prec)$ *satisfying*

1. *if* $j \in g(i)$ *then* $j = i$ *or* $i \prec j$*,*
2. *if* $j \in \mathrm{Odd}(g(i))$ *then* $j = i$ *or* $i \prec j$*,*
3. *if* $\lambda(i) = \mathrm{XY}$ *then* $i \notin g(i)$ *and* $i \in \mathrm{Odd}(g(i))$*,*
4. *if* $\lambda(i) = \mathrm{XZ}$ *then* $i \in g(i)$ *and* $i \in \mathrm{Odd}(g(i))$*,*
5. *if* $\lambda(i) = \mathrm{YZ}$ *then* $i \in g(i)$ *and* $i \notin \mathrm{Odd}(g(i))$*.*

Note that the gflow is the special case of the extended gflow where each vertex has the measurement plane XY. Unfortunately, gflow does not always exist in the case of ZX diagrams obtained with the entire ruleset of the ZX-calculus [1]. For example, the rules that create the phase gadgets break the gflow. However, the existence of the extended gflow is preserved, and there exists a polynomial-time algorithm for finding it [1, 15]. This algorithm is already implemented in PyZX [10], a Python library for circuit optimisation using the ZX-calculus. We utilise this routine in our dissertation to obtain a good initial rank-decomposition of the ZX diagram, as explained in Chapter 5.

Chapter 5

New simulation strategy

In this chapter, we present our new simulation strategy, which is based on the rank-width of the diagram. First, we outline the main steps of our algorithm and then examine each of them more closely. We also discuss the implementation details of the main routine, including its computational complexity.

5.1 Pipeline

Suppose we are given a quantum circuit, and our task is to strongly simulate it. That is, given an input state and an effect, we need to compute the value of the underlying tensor diagram. Our approach consists of the following steps:

1. Convert the quantum circuit into a Clifford+T ZX diagram.
2. Reduce the ZX diagram to a graph-like form and optimise it using the rules of the ZX-calculus.
3. Compute the initial rank-decomposition of the graph-like diagram using extended gflow.
4. Improve the rank-decomposition using simulated annealing.
5. Run the main simulation routine on the resulting decomposition.

In step 1, we use the conventions from Chapter 2. As discussed in Section 3.2, Clifford+T fragment of the ZX-calculus is computationally universal, meaning that an equivalent (up to an additive error ε) ZX diagram always exists for arbitrarily small $\varepsilon > 0$. Moreover, the size of the ZX diagram is polynomial in the circuit size and $\log(1/\varepsilon)$ by the Solovay-Kitaev theorem.

In step 2, we employ the techniques from Sections 3.1 and 3.3 to obtain a compact graph-like diagram. In particular, we first convert the ZX diagram to a graph-like form, then apply local complementation and pivoting to eliminate the majority of the Clifford spiders, and finally isolate the phase gadgets and fuse the same-legged ones, if any. If the fusion took place, then the emerged Clifford spider can be further eliminated via local complementation or pivoting. Steps 1 and 2 have already been implemented in PyZX as functions pyzx.Circuit.to_graph() and pyzx.full_reduce(Graph).

In step 3, we compute a rank-decomposition of the graph-like diagram via the function pyzx.gflow.gflow(Pauli=True), which returns an even more general order than the extended gflow, called *Pauli flow* [19]. We use this partial ordering of vertices to construct a linear rank-decomposition, whose width is bounded by the number of qubits in the original circuit. A detailed explanation of this step is given in Section 5.2.

Step 4 focuses on improving the rank-decomposition after step 3. We accomplish this by employing the techniques from [16]. Although it is unclear how to provide a formal guarantee for the outcome of this step, in some practical cases, the improvement is quite significant – see Section 6.1 for benchmarks. A detailed description of this step is given in Section 5.3.

Step 5 is the key part of our strategy. Once we have obtained a good rank-decomposition of the diagram, we do the simulation by recursively traversing the decomposition tree. See Section 5.4 for a detailed review of our algorithm.

5.2 Initial rank-decomposition

In this section, we use the notation from Section 4.3. By q we denote the number of qubits in the original circuit, that is, the number of inputs and outputs in the graph-like diagram. We now prove the following claim.

Proposition 5.2.1. *If an open graph (G, I, O) with $|I| = |O| = q$ admits an extended gflow $(g, \prec)$ with labels λ, then it admits a linear rank-decomposition with width at most q, computable in polynomial time.*

Proof. First, we consider the *linear extension* of the partial order $\prec$, i.e., a sequence of vertices v_i such that for each $i < j$ we have $v_i \not\succ v_j$. This order can be computed by arranging the layers of $\prec$ one after another.

We prove that v_i constitutes a linear rank-decomposition with width $\leq q$ by induction from right to left. By $M^{(i)}$ we denote the adjacency matrix of size $(|V|-i)\times i$

between $\{v_{i+1}, \ldots, v_{|V|}\}$ and $\{v_1, \ldots, v_i\}$, respectively, and set $r_i := \mathrm{rk}(M^{(i)})$. Let us show that for each $i \in [2; |V|]$, we have $r_{i-1} \leq r_i$ for $v_i \notin O$ and $r_{i-1} \leq r_i + 1$ for $v_i \in O$.

Note that $M^{(i-1)}$ can be obtained from $M^{(i)}$ by deleting its last column and appending a row, which represents the edges between v_i and $v_1, \ldots, v_{i-1}$. Both $M^{(i-1)}$ and $M^{(i)}$ are depicted as follows:

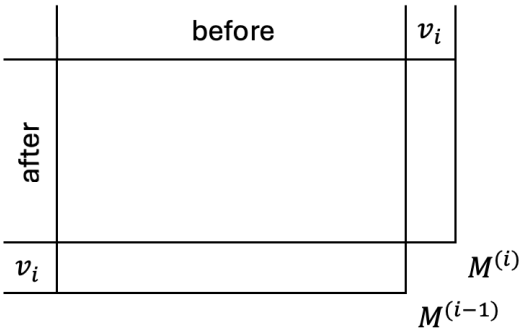


Figure 5.1: Consecutive adjacency matrices from a linear rank-decomposition.

It follows that $r_{i-1} \leq r_i + 1$. We now assume $v_i \notin O$ and show that $r_{i-1} \leq r_i$ by considering two cases:

1. $\lambda(v_i) = \mathrm{XY}$. By Definition 4.3.3, we have $v_i \notin g(v_i)$, $v_i \in \mathrm{Odd}(g(v_i))$, and for each $j < i$ we have $v_j \notin \mathrm{Odd}(g(v_i))$. Take some $u \in g(v_i)$ and add all the rows from $g(v_i) \setminus \{u\}$ to the row u, so that it has a single one at the i-th column. Then, zero out the entire i-th column so that $M^{(i)}$ becomes block-diagonal. See Figure 5.2 for the illustration. Now consider the submatrix $\widetilde{M}$ between $v_{i+1} \ldots v_{|V|}$ and $v_1 \ldots v_{i-1}$, which corresponds to the 'after–before' section in the first diagram of Figure 5.2. Note that the row operations preserve the ranks of $\widetilde{M}$, $M^{(i)}$, and $M^{(i-1)}$. Due to the diagonalisation of $M^{(i)}$ as shown in the last diagram of Figure 5.2, we have $\mathrm{rk}(\widetilde{M}) = \mathrm{rk}(M^{(i)}) - 1$. As $M^{(i-1)}$ differs from $\widetilde{M}$ by one extra row, we have $\mathrm{rk}(M^{(i-1)}) \leq \mathrm{rk}(\widetilde{M}) + 1 = \mathrm{rk}(M^{(i)})$.

2. $\lambda(v_i) \in \{\mathrm{XZ}, \mathrm{YZ}\}$. By Definition 4.3.3, we have $v_i \in g(v_i)$, and for each $j < i$ we have $v_j \notin \mathrm{Odd}(g(v_i))$. Hence, by adding the rows from $g(v_i) \setminus \{v_i\}$ to the row v_i, we can zero out its 'before' part – see Figure 5.3. Define $\widetilde{M}$ as in the previous case. Clearly, $\mathrm{rk}(M^{(i-1)}) = \mathrm{rk}(\widetilde{M}) \leq \mathrm{rk}(M^{(i)})$.

It follows that $\max(r_i) \leq |O| = q$, as desired. As explained in [1], extended gflow can be computed in polynomial time. Hence, our linear rank-decomposition is also computable in polynomial time. □

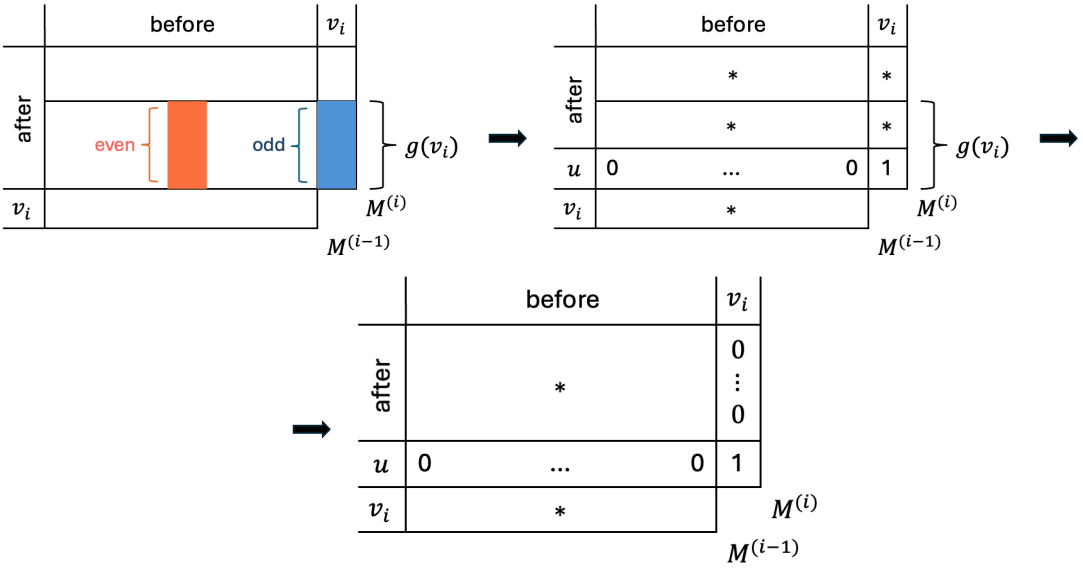


Figure 5.2: Block diagonalisation of the adjacency matrix in the XY case.

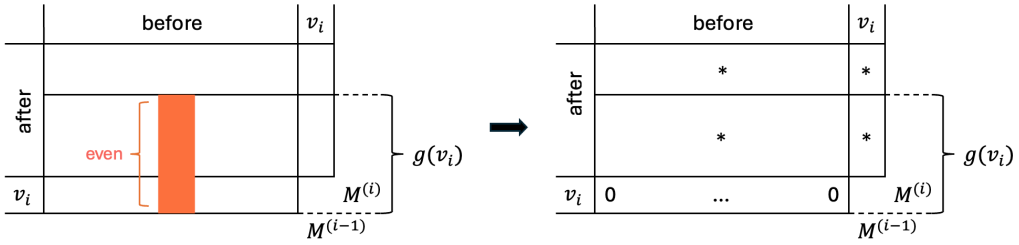


Figure 5.3: Matrix row reduction in the XZ and YZ cases.

5.3 Improving rank-decomposition

In this section, we present our method for improving rank-decomposition as in [16]. We outline the simulated annealing approach in general, as well as discuss its problem-specific features, such as the score function, operator selection, and cooling schedule. We also discuss efficient algorithm implementation, in particular, cut-rank caching and update techniques.

5.3.1 Overview of simulated annealing

Simulated annealing is a type of local search algorithm inspired by the physical process of cooling materials, particularly the annealing technique used in metallurgy. The method relies on a temperature (or energy) parameter that gradually decreases over time and determines the likelihood of accepting solutions that are worse than

the current one. Allowing such temporary degradations enables the algorithm to escape local minima. At higher temperatures, the algorithm explores more randomly, while a gradual reduction in temperature steers the search toward a stable solution. The schedule by which the temperature is lowered is known as the cooling schedule. Algorithm 5.1 illustrates a simplified outline of simulated annealing, where a fixed number of iterations is performed and a single score function is used for optimisation.

Algorithm 5.1 Simulated annealing general routine

Require: $T_0 > 0$, $Q > 0$, S_0 ▷ initial temperature, iteration count, initial solution
  $curSol \leftarrow S_0$
  $bestSol \leftarrow curSol$ ▷ initialise current and best solutions
  $curScore \leftarrow score(S_0)$
  $bestScore \leftarrow curScore$ ▷ initialise curScore and bestScore
  $T \leftarrow T_0$ ▷ initialise temperature
  for $i = 0 \dots Q - 1$ **do**
    $op \leftarrow random\_operator()$ ▷ randomly select operation
    $newSol \leftarrow op(curSol)$
    $newScore \leftarrow score(newSol)$ ▷ perform operation and calculate score
    if $newScore \leq curScore \vee random() < e^{(curScore - newScore)/T}$ **then**
      $curSol \leftarrow newSol$
      $curScore \leftarrow newScore$ ▷ update current solution and score
      if $curScore < bestScore$ **then**
        $bestScore \leftarrow curScore$
        $bestSol \leftarrow curSol$ ▷ update best solution and score
      end if
    end if
    $T \leftarrow next(T)$ ▷ compute new temperature based on cooling schedule
  end for

Our search starts from the linear rank-decomposition constructed in Section 5.2 and gradually improves it throughout the search. Instead of the fixed iteration loop, the stopping condition could be that the temperature is lower than a certain threshold or that the score is sufficiently low. It is also possible to stop the search after reaching a certain time limit.

Additionally, we can explore multiple operations in a single iteration and select the best candidate with respect to the score function. This potentially increases the quality of the solution found by our algorithm, but increases the overall complexity of the search.

5.3.2 Score function

Let $T = (V_T, E_T)$ be the rank-decomposition tree, and denote by r_e the cut-rank of an edge $e \in E_T$. Furthermore, for a non-leaf vertex $v \in V_T$ set $w_v := r_{e_1} + r_{e_2} + r_{e_3} - \max(r_{e_1}, r_{e_2}, r_{e_3})$, where e_1, e_2, e_3 are the incident edges to v. We have considered the following score functions:

- $score\_square(T) := \sum_{e \in E_T} r_e^2$,
- $score\_flops(T) := \log_2 \left(\sum_{v \in V_T} 2^{w_v} \right)$.

The latter score function is motivated by the complexity of our main simulation routine: its exponent denotes the computational cost of the next step, and this is the target function we wish to minimise. However, minimising it directly leads to poorer results rather than minimising *score_square* and maintaining the best candidate with respect to $score\_flops$. For experiments with the score functions, see Chapter 6.

5.3.3 Operator selection

We stick to [16] and implement three types of operators acting on the rank-decomposition:

1. Leaf swap. It takes two different random leaves v_1, v_2 of the decomposition and swaps them – see Figure 5.4.

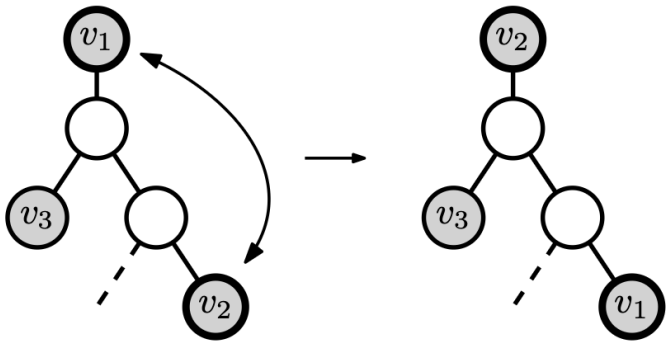


Figure 5.4: [16, Figure 6.2] Example of the leaf swap operator swapping v_1 and v_2.

2. Local swap. It starts by taking a random internal node c of the decomposition. Then, two neighbours of c are randomly selected, such that at least one of them is internal. Let b be the internal node and a be the other node. We choose another neighbour of b, call it d, and swap the nodes a and d, so that a becomes a neighbour of b and d becomes a neighbour of c. See Figure 5.5 for illustration.

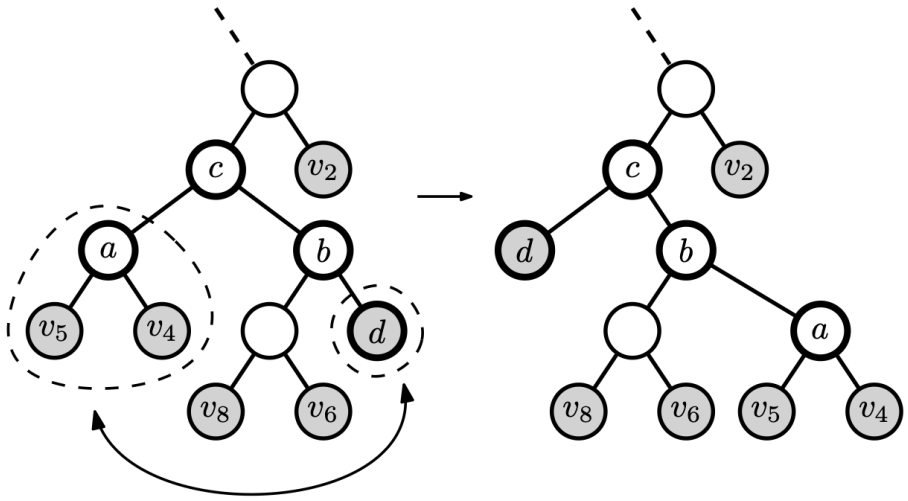


Figure 5.5: [16, Figure 6.3] Example of the local swap operator.

3. Move subtree. Two random nodes a and b are chosen such that they are not adjacent and have no common neighbour. The path P between a and b is then computed to find a' and b', which are the neighbours of respectively a and b on P. Finally, the two neighbors of a' other than a are joined together, and the edge between b and b' is split into two parts that get connected to a'. An example of the move subtree operator is given in Figure 5.6.

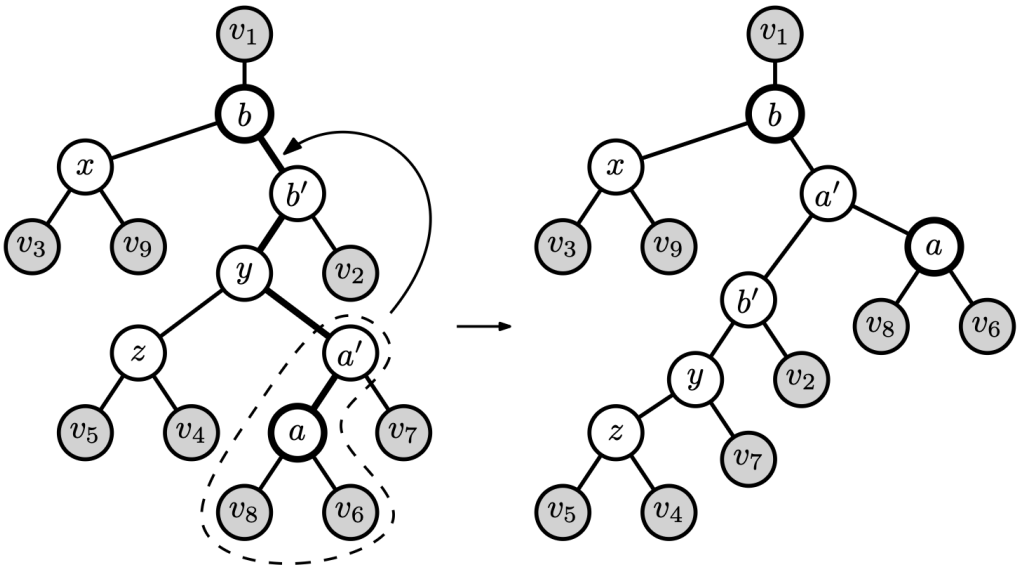


Figure 5.6: [16, Figure 6.1] Example of the move subtree operator.

The probabilities with which the operations are chosen are set to 0.1 for the leaf swap, 0.4 for the local swap, and 0.5 for the move subtree operator.

5.3.4 Cooling schedule

Various cooling schedules are used for simulated annealing, such as:

- Linear $T_i = T_{i-1} - \beta$ for constant β,
- Exponential $T_i = \alpha \cdot T_{i-1}$, with constant $\alpha \in [0; 1]$, and
- Logarithmic $T_i = T_0 / \log(i + 10)$,

where T_i denotes the temperature on the i-th iteration. In practice, the exponential schedule is the most widely used. In Chapter 6, we fine-tune the parameters of the cooling schedule, such as the initial temperature and the cooling rate.

Additionally, we employ the strategy called *adaptive cooling*. This technique adjusts the cooling rate based on the success of the search process. The idea behind this is to allow the algorithm to make bigger changes to the solution when it has diverged significantly from the best found so far. The modified temperature T_i' is computed as follows:

$$T_i' = T_i \cdot \left(1 + \frac{curScore - bestScore}{curScore}\right).$$

5.3.5 Efficient cut-rank calculation

After changing the decomposition tree structure, the cut-ranks of the edges need to be recalculated. Recall that the cut-rank is defined as the rank over $\mathbb{F}_2$ of the adjacency matrix of the partition corresponding to that edge. Thus, naively, each update of the tree can be handled in $O(N^4)$ operations (for the tree size N) by recomputing all the cut-ranks using Gaussian elimination.

Let us discuss a few optimisations. First, note that not all cut-ranks need to be recomputed. For the leaf swap, only the edges on the path between v_1 and v_2 need to be considered; for all other edges, the partitions are unchanged. For the local swap, only one edge (bc) requires update – see Figure 5.5. For the move subtree operator, $O(|P|)$ new ranks need to be computed.

Another observation is that the rows of the adjacency matrix can be stored as a bit vector, that is, a vector of unsigned 64-bit integers. Thus, each primitive row operation throughout Gaussian elimination costs $N/64$ instead of N processor instructions, resulting in the $\times 64$ speedup for the algorithm.

Let us introduce another optimisation (which is not yet implemented) that brings the cut-rank calculation cost down to $O(N^3 \log N)$ per iteration. The idea is that

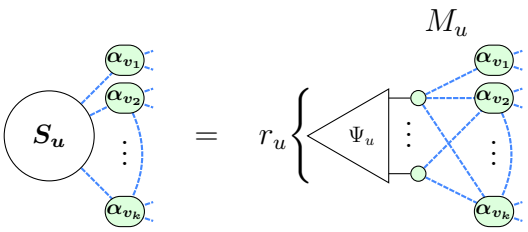
while recursively traversing the decomposition tree, we can reuse the cut-ranks of the children to calculate the cut-rank of the parent. For each edge, we store the row-echelon form of the adjacency matrix, whose rows correspond to the leaves belonging to the subtree and whose columns correspond to the remaining leaves. After computing the cut-ranks of the children, we first take the child with the largest subtree size and copy its row-echelon form to the parent. Then, we account for the leaves in the smaller subtree by iteratively deleting columns and appending rows to the row-echelon form. Note that deleting a single column, as well as appending a single row, might break the row-echelon form. However, it is possible to recompute it in $O(N^2)$ time, which we leave as an exercise to the reader. It can be shown that, due to 'merging' the smallest subtree to the largest, we perform $O(N \log N)$ such updates in total. Hence, the overall complexity is $O(N^3 \log N)$. Note that in the case of the linear rank-decomposition, the complexity is actually $O(N^3)$.

5.4 Main routine

In this section, we describe our approach to simulating graph-like diagrams using the rank-decomposition obtained from the previous step. We assume that the ZX diagram is given as a simple undirected graph $G = (V, E)$ along with the phases $\alpha_v \in [0; 2\pi)$ for each $v \in V$. We also assume that the decomposition tree $T = (V_T, E_T)$ is *rooted* by choosing an arbitrary edge $(v, w) \in E_T$, orienting the edges from the subtrees towards v and w, and adding a dummy vertex 0 as a parent of v and w. Each vertex u in the rooted decomposition tree corresponds to a set of leaves from its subtree, that is, a subset of the vertices of the ZX diagram: $S_u \subset V$. The *rank* r_u of a vertex u is defined as the cut-rank of (u, p) for the parent p of u if $u \neq 0$, and 0 otherwise. Note that r_u equals the cut-rank of the partition $(S_u, V \setminus S_u)$.

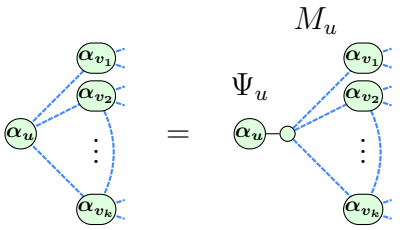
5.4.1 Recursive subroutine

Our main simulation routine is based on the depth-first search over the rooted rank-decomposition tree. The primary recursive subroutine simulate_recursive(u) returns a simulated version of the subdiagram induced by S_u:



In other words, it returns an r_u-dimensional tensor Ψ_u and a parity matrix M_u of size $r_u \times |V|$ to satisfy the equivalence above. Hereafter, we assume that a parity map $\mathbb{F}_2^n \to \mathbb{F}_2^m$ is given by a parity matrix of size $n \times m$, acting on a row-vector of length n – see Section 2.4 for the background regarding parity maps.

The function simulate_recursive (u) works as follows. For a leaf u, it returns $\Psi_u = |0\rangle + e^{i\alpha_u} |1\rangle$ and M_u being the neighbourhood of u:

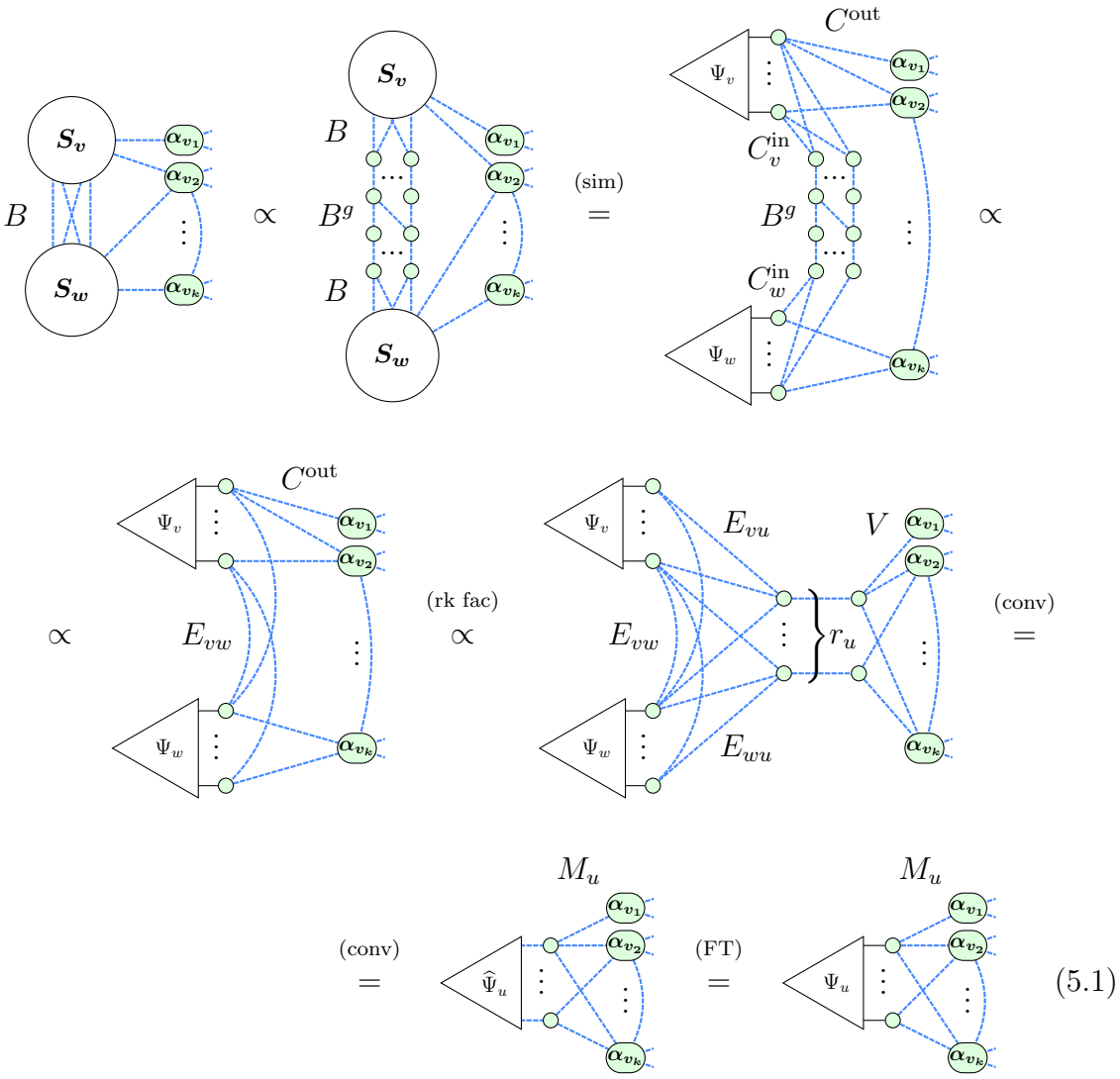


For a non-leaf vertex u, we follow the course of reasoning given by Eq. (5.1). First, we evaluate (Ψ_v, M_v), (Ψ_w, M_w) for the children v, w of u. Next, we take the biadjacency matrix B between S_v and S_w and compute its generalised inverse B^g, that is, a binary matrix B^g satisfying $BB^gB = B$. Let C^{out} be the submatrix of $M_v \sqcup M_w$ representing the outgoing edges from Ψ_v, Ψ_w to $V \setminus (S_v \cup S_w)$, and let C_v^{in}, C_w^{in} represent the remaining edges, going from Ψ_v, Ψ_w to B^g. We compute the matrix $E_{vw} := C_v^{\text{in}} B^g (C_w^{\text{in}})^T$, which represents the parity map between Ψ_v and Ψ_w. Also, we compute the rank-factorisation $C^{\text{out}} = UV$ for U, V being the binary matrices of size $(r_v + r_w) \times r_u$ and $r_u \times |V|$, respectively. Denote $U = \begin{pmatrix} E_{vu} \\ E_{wu} \end{pmatrix}$, and set $M_u := V$. We now perform the convolution $\text{conv}(\Psi_v, \Psi_w, E_{vu}, E_{wu}, E_{vw})$ to obtain $\widehat{\Psi}_u$ as explained in the next subsection. Finally, we apply the Fourier transform to $\widehat{\Psi}_u$ to compute Ψ_u.

This approach allows us to calculate the correct Ψ_u up to a multiplicative scalar. We keep track of the scalar as in Section 2.4.

When the call of simulate_recursive (0) terminates, it returns (Ψ_0, M_0) corresponding to the root of the decomposition tree. Note that Ψ_0 is a scalar, and we return Ψ_0 as the result of the simulation.

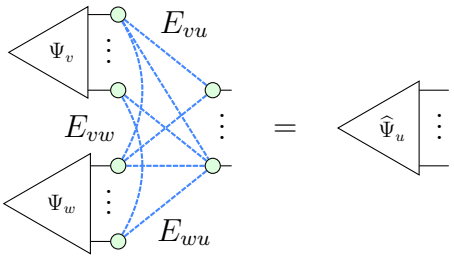
Let us discuss the complexity of our routine. By w_u we denote the convolution complexity logarithm for the vertex u, which we establish in the next subsection. The second most expensive part is the rank factorisation, which takes roughly $O(|V|^3)$ operations per vertex – see Section 4.1. Hence, the complexity of the last step of our simulation strategy is $O(|V|^4) + \tilde{O}(\sum_u 2^{w_u})$.



(5.1)

5.4.2 Convolution

The most expensive part of simulate_recursive(u) is the convolution, which involves evaluating $\widehat{\Psi}_u$ for the given Ψ_v, Ψ_w such that



(5.2)

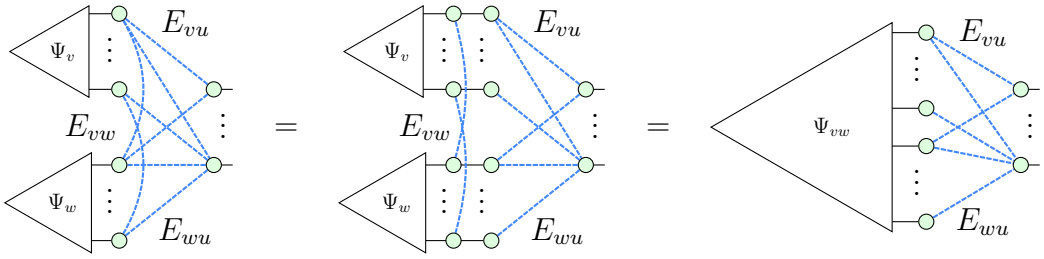
The methods we employ may turn out to be asymptotically suboptimal, yet we have been unable to improve their efficiency for now. This could be a good direction for future work.

Recall that the tensors $\widehat{\Psi}_u$, Ψ_v, and Ψ_w have r_u, r_v, and r_w legs, respectively. According to [9, Eq. 2], these cut-ranks satisfy the *submodular inequalities*:

$$r_u \le r_v + r_w, \qquad r_v \le r_u + r_w, \qquad r_w \le r_u + r_v.$$

We will assume these inequalities for the purposes of our complexity analysis. Let us present three approaches with complexities $\tilde{O}(2^{r_v+r_w})$, $\tilde{O}(2^{r_u+r_v})$, and $\tilde{O}(2^{r_u+r_w})$.

In the first approach, we unfuse the left part of the diagram and compute the entangled state separately:

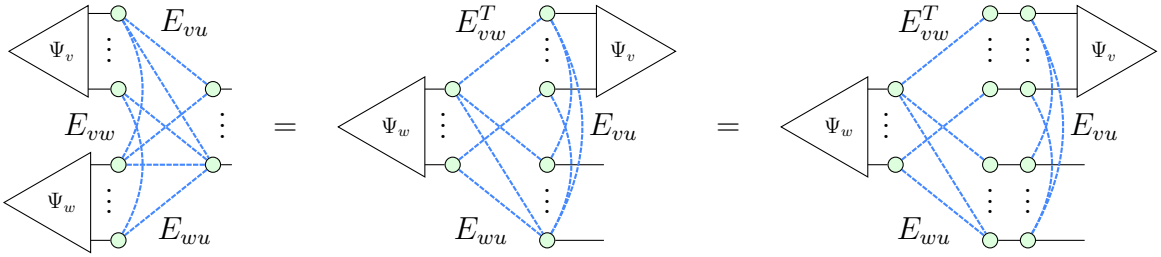


where $(\Psi_{vw})_{ab} = \left(\frac{1}{\sqrt{2}}\right)^{|E_{vw}|} (\Psi_v)_a (\Psi_w)_b (-1)^{\langle a, E_{vw} b\rangle}$. We see that the remaining part is a parity map, thus, expanding Ψ_{vw} into the basis states gives

$$\Psi_u \propto \sum_{a,b} (\Psi_{vw})_{ab} \left| E_{vu}^T a + E_{wu}^T b \right\rangle .$$

The complexity of this method is $\tilde{O}(2^{r_v+r_w})$, which is the cost of computing Ψ_{vw}.

In the second approach, we first rotate the state Ψ_v so that it becomes a post-selection, and then unfuse the rightmost part:



Again, we see that the left part is a parity map applied to Ψ_w, whose output dimension is $r_u + r_v$; thus, we can apply this map in $\tilde{O}(2^{r_u+r_v})$ time. The vertical Hadamard edges on the right introduce a phase $(-1)^{\langle a, E_{vu} b\rangle}$. Finally, the post-selection with Ψ_v is a tensor contraction, which costs $\tilde{O}(2^{r_u+r_v})$ as well.

The third approach with complexity $\tilde{O}(2^{r_u+r_w})$ is the symmetric version of the second one, where we rotate the state Ψ_w instead of Ψ_v.

The three methods described above result in the convolution complexity logarithm $w_u = \min(r_u + r_v, r_u + r_w, r_v + r_w) = r_u + r_v + r_w - \max(r_u, r_v, r_w)$. Hence, the total simulation complexity of the main routine is indeed $\tilde{O}\left(2^{score\_flops(T)}\right)$, with $score\_flops(T)$ defined as in Section 5.3.

Chapter 6

Benchmarks

We evaluate our methods on various quantum circuits, including a few small ones from the PyZX repository and random CNOT + Hadamard + T circuits of different sizes. We begin by measuring the quality of the rank-decomposition obtained after steps 3 and 4 of our strategy, including fine-tuning the parameters of the simulated annealer. Then, we compare our main routine to other simulation strategies, such as the ones in Quimb. We use the number of scalar operations (flops) to assess the complexity of the methods, without actually performing the contractions.

6.1 Circuit rank-width

In this section, we run the first four steps of our simulation pipeline. We record the rank-width of the decomposition obtained after steps 3 and 4, as well as the $score\_flops$ function, defined in Section 5.3, which serves as an approximation to the total number of scalar operations throughout the main routine. For the simulated annealer, we consider the score functions discussed in Section 5.3 and compare the performance on several special circuits. We use exponential scheduling and adjust the annealer's cooling rate α, as well as its initial and stopping temperatures T_0, T_{end}.

6.1.1 Special circuits

In the case of special circuits, the optimal parameter configuration for the annealer turns out to be $\alpha = 0.99$, $T_0 = 10$, and $T_{end} = 0.001$; see Appendix A for comparison.

Tables 6.1, 6.2 illustrate the performace of the annealer with respect to the $score\_flops$ function, using either $score\_flops(T)$ or $score\_square(T)$ as the score function. Although the score function may differ from the one we wish to minimise, we always maintain the best decomposition in terms of $score\_flops$ throughout the

optimisation. Hereafter Q, G are the qubit and gate counts of the circuit, R_{init}, R_{ann} are the widths of the initial and improved rank-decompositions, and ScFl_{init}, ScFl_{ann} are the values of $score\_flops$ before and after simulated annealing. We set $\Delta\mathrm{ScFl} := \mathrm{ScFl}_{init} - \mathrm{ScFl}_{ann}$, measuring the score improvement achieved by the annealer.

As seen from the Tables 6.1, 6.2 below, the $score\_square$ function leads to better improvements of $score\_flops$, rather than optimising $score\_flops$ directly. This is probably due to $score\_square$ having a smoother landscape and fewer local minima. Indeed, $score\_flops$ is very sensitive to larger cut-ranks (aka hills), which need to be traversed before falling into a deeper valley. Overall, the annealer with the square score function performs quite modestly, typically resulting in $\approx \times 2$ speed-up to the main routine, but in the luckiest case, it can give up to $\times 30$ speed-up.

Note that in rare cases, the rank-width of the resulting decomposition is larger than the initial one. This inconsistency showcases the gap between the rank-width and the complexity of our method, which can potentially be reduced in future work.

Circuit	**Q**	**G**	$\mathbf{R_{init}}$	$\mathbf{R_{ann}}$	$\mathbf{ScFl_{init}}$	$\mathbf{ScFl_{ann}}$	**ΔScFl**
Adder8	23	105	22	22	27.432	27.432	0.000
QFT8	8	148	8	8	13.738	13.738	0.000
barenco_tof_3.qasm	5	20	5	5	9.077	7.833	1.244
barenco_tof_4.qasm	7	34	7	7	11.355	11.285	0.070
barenco_tof_5.qasm	9	50	9	9	13.752	13.000	0.752
csla_mux_3_original	15	50	12	12	17.698	17.698	0.000
csum_mux_9_corrected	30	56	29	29	33.250	33.250	0.000
gf2^4_mult	12	51	12	12	18.342	18.342	0.000
ham15-low.qc	17	167	17	17	24.358	24.358	0.000
hwb6.qc	7	79	7	7	14.176	14.176	0.000
mod_mult_55	9	35	9	9	15.203	15.203	0.000
mod_red_21	11	74	11	11	17.722	17.722	0.000
qcla_com_7	24	95	24	24	30.174	30.169	0.005
qft_4.qasm	5	159	5	5	11.811	11.811	0.000
rc_adder_6	14	68	14	14	19.871	19.871	0.000
tof_10	19	51	19	19	24.778	24.774	0.004
tof_3	5	9	5	4	9.209	7.755	1.454
tof_4	7	15	7	6	11.980	10.488	1.492
tof_5	9	21	9	8	14.076	12.181	1.895
vbe_adder_3	10	30	10	8	14.015	12.224	1.791

Table 6.1: Annealer benchmark on special circuits with the $score\_flops$ function.

Circuit	**Q**	**G**	$\mathbf{R_{init}}$	$\mathbf{R_{ann}}$	$\mathbf{ScFl_{init}}$	$\mathbf{ScFl_{ann}}$	**ΔScFl**
Adder8	23	105	22	22	27.432	27.432	0.000
QFT8	8	148	8	8	13.738	13.711	0.027
barenco_tof_3.qasm	5	20	5	3	9.077	7.358	1.719
barenco_tof_4.qasm	7	34	7	6	11.355	9.577	1.778
barenco_tof_5.qasm	9	50	9	6	13.752	11.649	2.103
csla_mux_3_original	15	50	12	12	17.698	17.698	0.000
csum_mux_9_corrected	30	56	29	29	33.250	33.250	0.000
gf2^4_mult	12	51	12	12	18.342	18.342	0.000
ham15-low.qc	17	167	17	17	24.358	24.358	0.000
hwb6.qc	7	79	7	7	14.176	14.176	0.000
mod_mult_55	9	35	9	8	15.203	14.466	0.737
mod_red_21	11	74	11	11	17.722	17.721	0.001
qcla_com_7	24	95	24	24	30.174	30.174	0.000
qft_4.qasm	5	159	5	6	11.811	11.547	0.264
rc_adder_6	14	68	14	14	19.871	19.858	0.012
tof_10	19	51	19	19	24.778	24.761	0.017
tof_3	5	9	5	3	9.209	7.129	2.080
tof_4	7	15	7	5	11.980	8.755	3.225
tof_5	9	21	9	6	14.076	11.177	2.899
vbe_adder_3	10	30	10	3	14.015	8.768	**5.247**

Table 6.2: Annealer benchmark on special circuits with the *score_square* function.

6.1.2 Random circuits

Table 6.3 shows the annealer performance on a series of random CNOT + Hadamard + T circuits of various sizes. Namely, we vary the number of qubits and the gate count of the circuit, with the gate type probabilities set to 0.6 for CNOT, 0.2 for Hadamard, and 0.2 for the T gate. After sampling the gate type, we randomly choose its input qubits and append the gate to the diagram. This circuit sampling is implemented in PyZX as pyzx.generate.CNOT_HAD_PHASE_circuit(Q, G).

For a fixed qubit count, as the number of gates increases, the structure of the corresponding ZX diagram becomes more random. It is no surprise that the annealer fails to find much improvement for the larger gate counts: we conjecture that the rank-width of a dense random circuit is close to the number of qubits. For the smaller gate counts, as the circuit gets sparser, it finds $\approx \times 5$ improvement on average, better than what we have seen for the special circuits. To achieve even better results, we plan to increase the number of iterations for the annealer by implementing efficient cut-rank updates as described in Section 5.3.

Q	G	R_{init}	R_{ann}	$ScFl_{init}$	$ScFl_{ann}$	ΔScFl
10	50	10	5	14.521	9.257	5.264
10	100	9	7	12.808	12.280	0.529
10	200	10	10	14.601	14.525	0.076
10	300	10	10	15.525	15.425	0.099
15	75	15	8	18.248	15.122	3.126
15	150	15	14	18.460	17.757	0.703
15	250	15	15	20.295	20.295	0.000
15	350	15	15	21.248	21.228	0.020
20	100	17	12	20.087	16.620	3.468
20	200	19	19	22.954	22.954	0.000
20	300	20	20	24.775	24.775	0.000
20	400	20	20	26.338	26.338	0.000
25	125	22	12	25.558	21.079	4.480
25	250	25	25	29.276	29.276	0.000
25	350	25	25	30.738	30.737	0.001
25	450	25	25	31.273	31.273	0.000

Table 6.3: Annealer benchmark on random CNOT + H + T circuits.

6.2 Circuit simulation

In this section, we estimate the cost of tensor contractions performed by the main routine after steps 3 and 4 have taken place. As a baseline, we use the library Quimb, which contains several advanced strategies for tensor network contraction. Similar to the ZX-based approaches, Quimb first transforms the given quantum circuit to a simplified tensor network. Then, it finds an efficient contraction tree, that is, a sequence of tensor contractions required for diagram evaluation.

We compare our strategy against three Quimb optimisers: 'auto', 'auto-hq', and 'greedy' – these configurations work efficiently on our examples, as opposed to more expensive optimisers such as 'optimal'. For each optimiser, we take the produced contraction tree and evaluate the flops by calling its contraction_cost() method.

The Quimb optimisers are called in two different scenarios. In the first scenario, we feed the reduced ZX diagram into the optimiser, which is the exact input to our main contraction routine. In the second scenario, we run Quimb on the initial quantum circuit and let Quimb do the simplifications. Generally, we observe that the second scenario results in a better simulation cost. The reason might be that the ZX simplifications provide denser diagrams and are less suited for the Quimb's contraction routines.

6.2.1 Special circuits

Tables 6.4, 6.5 compare our simulation cost on special circuits against Quimb in the two scenarios specified above. To produce meaningful outcomes, the inputs and outputs for our simulation are set to the T-states.

When we run the Quimb optimisers on the reduced ZX diagram, in the majority of the examples, we see that our contraction cost is similar to the baseline. Note that on the 'vbe_adder_3' circuit, although the annealer gave the $\times 32$ speed-up, the performance is still close to the baseline. Also, the best result is achieved on the 'ham15-low' circuit, where the annealer hasn't improved at all. Hence, the greatest contribution to the supremacy of our strategy comes primarily from the initial rank-decomposition.

When running Quimb optimisers on the original circuit, our strategy is totally outperformed by Quimb. In many cases, we see that the Quimb optimisers reduce the input circuit to an empty tensor diagram. This outcome suggests that some future work is necessary to enhance the efficiency of ZX-based simplifications on special circuits.

Circuit	**Q**	**G**	$\mathbf{Fl_{auto}}$	$\mathbf{Fl_{hq}}$	$\mathbf{Fl_{greedy}}$	$\mathbf{Fl_{rw}}$
Adder8	23	105	$9.32 \cdot 10^3$	$3.35 \cdot 10^3$	$8.80 \cdot 10^3$	$\mathbf{5.60 \cdot 10^8}$
QFT8	8	148	$2.45 \cdot 10^3$	$1.98 \cdot 10^3$	$1.46 \cdot 10^4$	$4.18 \cdot 10^4$
barenco_tof_3.qasm	5	20	$2.14 \cdot 10^3$	$6.38 \cdot 10^2$	$9.68 \cdot 10^2$	$5.58 \cdot 10^2$
barenco_tof_4.qasm	7	34	$1.56 \cdot 10^3$	$3.11 \cdot 10^3$	$4.75 \cdot 10^3$	$3.59 \cdot 10^3$
barenco_tof_5.qasm	9	50	$5.76 \cdot 10^3$	$2.51 \cdot 10^3$	$7.72 \cdot 10^3$	$1.53 \cdot 10^4$
csla_mux_3_original	15	50	$1.46 \cdot 10^6$	$1.29 \cdot 10^4$	$9.86 \cdot 10^5$	$6.73 \cdot 10^5$
csum_mux_9_corrected	30	56	$5.67 \cdot 10^6$	$6.75 \cdot 10^3$	$3.24 \cdot 10^4$	$\mathbf{3.33 \cdot 10^{10}}$
gf2^4_mult	12	51	$2.98 \cdot 10^8$	$3.74 \cdot 10^6$	$4.06 \cdot 10^{16}$	$\mathbf{1.01 \cdot 10^6}$
ham15-low.qc	17	167	$8.99 \cdot 10^{14}$	$2.13 \cdot 10^{10}$	$6.43 \cdot 10^{23}$	$\mathbf{6.50 \cdot 10^7}$
hwb6.qc	7	79	$1.20 \cdot 10^7$	$2.97 \cdot 10^6$	$2.27 \cdot 10^9$	$\mathbf{5.60 \cdot 10^4}$
mod_mult_55	9	35	$1.53 \cdot 10^6$	$3.75 \cdot 10^4$	$4.61 \cdot 10^6$	$2.15 \cdot 10^5$
mod_red_21	11	74	$3.17 \cdot 10^8$	$1.44 \cdot 10^7$	$5.45 \cdot 10^7$	$\mathbf{6.57 \cdot 10^5}$
qcla_com_7	24	95	$8.07 \cdot 10^4$	$7.31 \cdot 10^3$	$1.02 \cdot 10^5$	$\mathbf{3.70 \cdot 10^9}$
qft_4.qasm	5	159	$1.93 \cdot 10^5$	$1.17 \cdot 10^4$	$1.46 \cdot 10^5$	$1.57 \cdot 10^4$
rc_adder_6	14	68	$1.15 \cdot 10^7$	$4.98 \cdot 10^3$	$2.14 \cdot 10^4$	$2.92 \cdot 10^6$
tof_10	19	51	$8.61 \cdot 10^4$	$4.51 \cdot 10^3$	$6.80 \cdot 10^3$	$\mathbf{8.85 \cdot 10^7}$
tof_3	5	9	$6.82 \cdot 10^2$	$4.50 \cdot 10^2$	$6.64 \cdot 10^2$	$4.78 \cdot 10^2$
tof_4	7	15	$1.01 \cdot 10^3$	$9.42 \cdot 10^2$	$1.49 \cdot 10^3$	$1.67 \cdot 10^3$
tof_5	9	21	$2.65 \cdot 10^3$	$1.08 \cdot 10^3$	$1.72 \cdot 10^3$	$\mathbf{1.33 \cdot 10^4}$
vbe_adder_3	10	30	$1.38 \cdot 10^4$	$1.92 \cdot 10^3$	$3.16 \cdot 10^3$	$1.43 \cdot 10^3$

Table 6.4: Circuit simulation benchmark on special circuits against Quimb.

Circuit	Q	G	$\mathbf{Fl_{auto}}$	$\mathbf{Fl_{hq}}$	$\mathbf{Fl_{greedy}}$	$\mathbf{Fl_{rw}}$
Adder8	23	105	0	0	0	$5.60 \cdot 10^8$
QFT8	8	148	$8.36 \cdot 10^2$	$7.00 \cdot 10^2$	$7.68 \cdot 10^2$	$4.18 \cdot 10^4$
barenco_tof_3.qasm	5	20	0	0	0	$5.58 \cdot 10^2$
barenco_tof_4.qasm	7	34	$2.04 \cdot 10^2$	$2.40 \cdot 10^2$	$2.40 \cdot 10^2$	$3.59 \cdot 10^3$
barenco_tof_5.qasm	9	50	$6.16 \cdot 10^2$	$6.64 \cdot 10^2$	$7.44 \cdot 10^2$	$1.53 \cdot 10^4$
csla_mux_3_original	15	50	0	0	0	$6.73 \cdot 10^5$
csum_mux_9_corrected	30	56	0	0	0	$3.33 \cdot 10^{10}$
gf2^4_mult	12	51	$6.48 \cdot 10^3$	$3.28 \cdot 10^3$	$8.05 \cdot 10^3$	$1.01 \cdot 10^6$
ham15-low.qc	17	167	$3.03 \cdot 10^8$	$2.19 \cdot 10^7$	$3.57 \cdot 10^8$	$6.50 \cdot 10^7$
hwb6.qc	7	79	$3.19 \cdot 10^3$	$3.08 \cdot 10^3$	$4.62 \cdot 10^3$	$5.60 \cdot 10^4$
mod_mult_55	9	35	$3.32 \cdot 10^2$	$3.32 \cdot 10^2$	$3.60 \cdot 10^2$	$2.15 \cdot 10^5$
mod_red_21	11	74	$1.46 \cdot 10^3$	$1.42 \cdot 10^3$	$1.50 \cdot 10^3$	$6.57 \cdot 10^5$
qcla_com_7	24	95	0	0	0	$3.70 \cdot 10^9$
qft_4.qasm	5	159	$9.84 \cdot 10^2$	$9.80 \cdot 10^2$	$9.80 \cdot 10^2$	$1.57 \cdot 10^4$
rc_adder_6	14	68	$1.40 \cdot 10^3$	$1.08 \cdot 10^3$	$1.08 \cdot 10^3$	$2.92 \cdot 10^6$
tof_10	19	51	0	0	0	$8.85 \cdot 10^7$
tof_3	5	9	0	0	0	$4.78 \cdot 10^2$
tof_4	7	15	0	0	0	$1.67 \cdot 10^3$
tof_5	9	21	0	0	0	$1.33 \cdot 10^4$
vbe_adder_3	10	30	$2.48 \cdot 10^2$	$2.46 \cdot 10^2$	$2.52 \cdot 10^2$	$1.43 \cdot 10^3$

Table 6.5: Circuit simulation benchmark on special circuits against Quimb without ZX simplifications.

6.2.2 Random circuits

We have also run the experiments on the random CNOT + Hadamard + T circuits, where our strategy demonstrates a lot better performance – see Tables 6.6, 6.7.

For the sparser circuits, we do not observe a considerable difference: our performance is similar or slightly better than Quimb in both scenarios. However, in the dense case, our strategy performs a lot better than Quimb in both scenarios. Again, referring to Table 6.3, we see that the major contribution comes from the initial decomposition obtained from the extended gflow.

Finally, we have measured the running time of our entire simulation routine on random CNOT + H + T circuits. Figures 6.1, 6.2 illustrate the performance of our approach without the annealer against the default 'auto-hq' Quimb simulator, run on our MacBook Air M1 laptop. Each experiment was repeated 5 times. We observe a consistent 10x speedup compared to Quimb on all instances. Such success may be attributed to the power of ZX simplifications in the case of random circuits and the gflow technique, which works well in combination with the ZX-based transformations.

Q	G	$\mathbf{Fl_{auto}}$	$\mathbf{Fl_{hq}}$	$\mathbf{Fl_{greedy}}$	$\mathbf{Fl_{rw}}$
10	50	$6.80 \cdot 10^1$	$6.60 \cdot 10^1$	$7.00 \cdot 10^1$	$7.80 \cdot 10^1$
10	100	$7.60 \cdot 10^1$	$7.60 \cdot 10^1$	$8.00 \cdot 10^1$	$7.20 \cdot 10^1$
10	200	$2.63 \cdot 10^3$	$1.54 \cdot 10^3$	$6.42 \cdot 10^3$	$9.38 \cdot 10^2$
10	300	$3.92 \cdot 10^7$	$3.12 \cdot 10^5$	$3.41 \cdot 10^8$	$\mathbf{3.91 \cdot 10^4}$
15	75	$6.40 \cdot 10^1$	$6.40 \cdot 10^1$	$6.80 \cdot 10^1$	$5.40 \cdot 10^1$
15	150	$1.31 \cdot 10^3$	$9.04 \cdot 10^2$	$1.51 \cdot 10^3$	$7.26 \cdot 10^2$
15	250	$5.91 \cdot 10^5$	$3.82 \cdot 10^5$	$7.69 \cdot 10^5$	$\mathbf{4.59 \cdot 10^4}$
15	350	$2.90 \cdot 10^7$	$2.25 \cdot 10^6$	$5.82 \cdot 10^8$	$\mathbf{8.24 \cdot 10^5}$
20	100	$5.00 \cdot 10^1$	$5.00 \cdot 10^1$	$5.20 \cdot 10^1$	$9.00 \cdot 10^1$
20	200	$3.01 \cdot 10^3$	$1.53 \cdot 10^3$	$2.74 \cdot 10^3$	$1.10 \cdot 10^3$
20	300	$3.21 \cdot 10^9$	$6.48 \cdot 10^8$	$1.59 \cdot 10^{11}$	$\mathbf{2.44 \cdot 10^6}$
20	400	$4.36 \cdot 10^{12}$	$3.81 \cdot 10^{11}$	$8.06 \cdot 10^{13}$	$\mathbf{4.75 \cdot 10^7}$
25	125	$1.20 \cdot 10^1$	$1.20 \cdot 10^1$	$1.20 \cdot 10^1$	$2.20 \cdot 10^1$
25	250	$2.08 \cdot 10^4$	$6.34 \cdot 10^3$	$1.14 \cdot 10^4$	$3.35 \cdot 10^3$
25	350	$3.68 \cdot 10^8$	$6.10 \cdot 10^8$	$5.25 \cdot 10^{10}$	$\mathbf{3.92 \cdot 10^6}$
25	450	$1.81 \cdot 10^{14}$	$8.97 \cdot 10^{12}$	$3.23 \cdot 10^{14}$	$\mathbf{1.58 \cdot 10^8}$

Table 6.6: Circuit simulation benchmark on random CNOT + H + T circuits against Quimb.

Q	G	$\mathbf{Fl_{auto}}$	$\mathbf{Fl_{hq}}$	$\mathbf{Fl_{greedy}}$	$\mathbf{Fl_{rw}}$
10	50	0	0	0	0
10	100	$2.44 \cdot 10^2$	$2.44 \cdot 10^2$	$2.44 \cdot 10^2$	$7.80 \cdot 10^1$
10	200	$1.32 \cdot 10^5$	$1.02 \cdot 10^5$	$2.29 \cdot 10^5$	$\mathbf{5.55 \cdot 10^3}$
10	300	$5.92 \cdot 10^5$	$5.40 \cdot 10^5$	$1.13 \cdot 10^6$	$\mathbf{1.91 \cdot 10^5}$
15	75	0	0	0	$5.80 \cdot 10^1$
15	150	$6.86 \cdot 10^4$	$7.17 \cdot 10^4$	$5.23 \cdot 10^4$	$\mathbf{2.47 \cdot 10^3}$
15	250	$6.64 \cdot 10^6$	$4.95 \cdot 10^6$	$1.30 \cdot 10^{10}$	$\mathbf{2.07 \cdot 10^5}$
15	350	$1.04 \cdot 10^7$	$6.74 \cdot 10^6$	$1.29 \cdot 10^9$	$\mathbf{1.88 \cdot 10^6}$
20	100	0	0	0	$9.00 \cdot 10^1$
20	200	$1.58 \cdot 10^5$	$6.78 \cdot 10^4$	$3.78 \cdot 10^5$	$\mathbf{6.84 \cdot 10^3}$
20	300	$4.12 \cdot 10^7$	$6.09 \cdot 10^6$	$6.14 \cdot 10^8$	$\mathbf{8.99 \cdot 10^4}$
20	400	$6.68 \cdot 10^8$	$1.09 \cdot 10^8$	$1.03 \cdot 10^{11}$	$\mathbf{2.34 \cdot 10^6}$
25	125	0	0	0	$1.04 \cdot 10^2$
25	250	$4.30 \cdot 10^5$	$1.64 \cdot 10^5$	$4.92 \cdot 10^5$	$\mathbf{3.69 \cdot 10^3}$
25	350	$1.58 \cdot 10^9$	$2.89 \cdot 10^8$	$1.75 \cdot 10^{11}$	$\mathbf{1.51 \cdot 10^5}$
25	450	$4.19 \cdot 10^{10}$	$5.55 \cdot 10^9$	$7.59 \cdot 10^{11}$	$\mathbf{2.38 \cdot 10^8}$

Table 6.7: Circuit simulation benchmark on random CNOT + H + T circuits against Quimb without ZX simplifications.

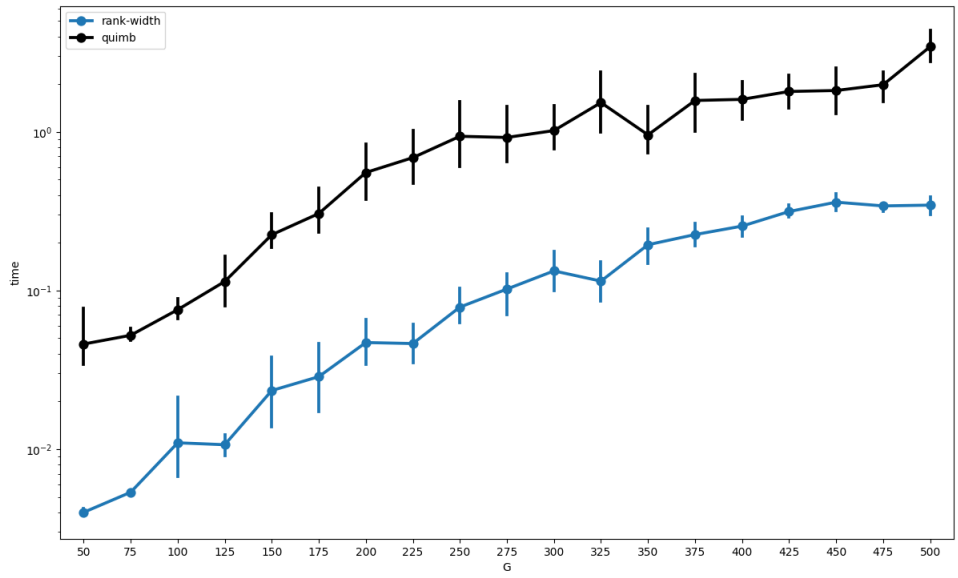


Figure 6.1: Running time (s) of our routine vs Quimb on random CNOT + H + T circuits on 10 qubits.

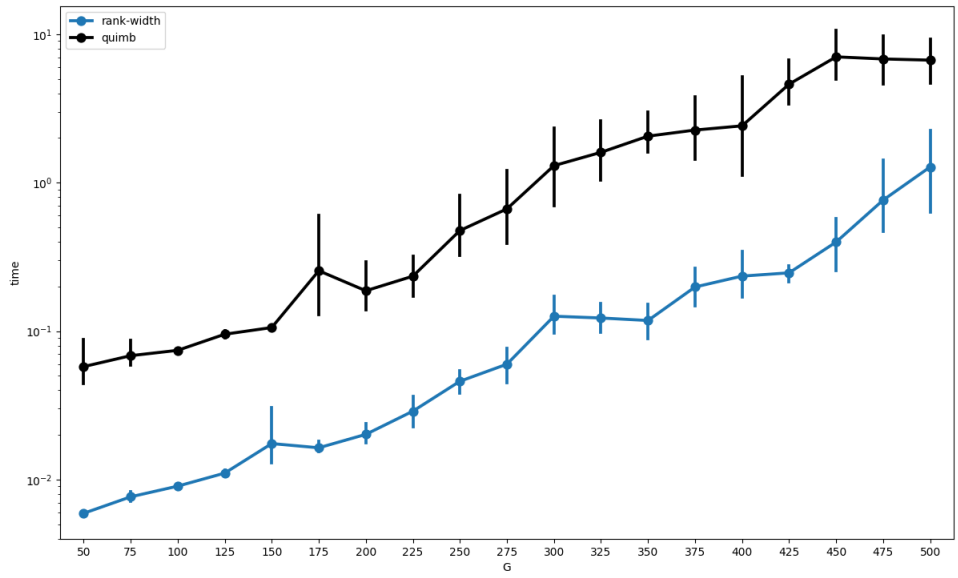


Figure 6.2: Running time (s) of our routine vs Quimb on random CNOT + H + T circuits on 20 qubits.

Chapter 7

Conclusions and future work

The results presented in this dissertation demonstrate that rank-width offers a powerful tool for guiding classical simulation of quantum circuits within the ZX-calculus framework. By combining extended gflow, simulated annealing, and a recursive contraction strategy, we established a new simulation pipeline that competes with existing approaches in both conceptual clarity and computational efficiency. Nevertheless, several directions remain open for extending and refining our method.

A natural first avenue for future work concerns the optimisation of the convolution subroutine in the main simulation routine. Currently, the computational cost is governed by the quantity $w_u = \min(r_u + r_v, r_u + r_w, r_v + r_w)$ at each decomposition node, with asymptotic complexity $\tilde{O}(2^{w_u})$. While this already provides a non-trivial improvement over naive contraction, the routine is still the computational bottleneck of our method. Investigating algebraic shortcuts for parity maps or exploiting additional structure of the ZX diagram (such as repeated subpatterns or phase gadget symmetries) could reduce this cost. Techniques from fast matrix multiplication over $\mathbb{F}_2$, or alternative tensor convolution algorithms, may yield further asymptotic gains.

A second direction involves the improvement of rank-decomposition heuristics. Our approach currently depends on extended gflow to generate an initial decomposition, which is then optimised using simulated annealing. Our benchmarks show that while the annealer typically yields modest improvements, the initial rank-decomposition already provides competitive simulation costs compared to existing strategies. Alternative heuristics such as those based on reinforcement learning or hybrid combinations of graph partitioning with flow-based orderings may generate more balanced decompositions and avoid the limitations of purely annealing-based refinements. In addition, developing efficient update techniques for cut-rank calculations would allow significantly longer annealing runs and deeper exploration of the search space, potentially unlocking decompositions of substantially lower width.

Another promising line of investigation is the integration of our method with stabiliser decompositions. While our current strategy treats Clifford+T circuits uniformly, there are many instances where decomposing collections of T gates into Clifford components offers exponential savings. A hybrid method could apply our rank-width guided contraction on parts of the diagram with favourable structure, while simultaneously employing stabiliser decompositions on subdiagrams rich in non-Clifford spiders. Such an approach would combine the strengths of graph-theoretic and algebraic methods, and could bridge the gap between theoretical efficiency and practical runtime.

Finally, our work leaves open the question of whether rank-width can be linked directly to classical hardness results in quantum simulation. While it is known that computing rank-width exactly is NP-hard, its precise relationship with measures such as treewidth, entanglement entropy, or stabiliser rank remains underexplored in the context of ZX diagrams. Establishing such connections would not only clarify the theoretical limits of our method but could also identify precise thresholds where classical simulation is expected to fail. This line of inquiry lies at the interface of graph theory, complexity theory, and quantum information, and may shed light on the fundamental boundary between classical and quantum computation.

In conclusion, we have presented a simulation algorithm for Clifford+T quantum circuits based on rank-width. The method proceeds by reducing the input circuit to a graph-like ZX diagram, computing a rank-decomposition using extended gflow, optimising this decomposition with simulated annealing, and finally performing recursive contractions guided by the decomposition. Our benchmarks demonstrate that this strategy produces non-trivial improvements in contraction cost and achieves competitive performance with existing tensor network simulators. The central technical contribution lies in showing that extended gflow yields a linear rank-decomposition of bounded width, and in designing an efficient simulation routine that leverages the resulting structure.

Beyond the immediate algorithmic contributions, the broader significance of this work lies in establishing rank-width as a meaningful complexity parameter for quantum circuit simulation. Just as treewidth has become a cornerstone in both theoretical computer science and practical algorithms, rank-width may play a similar role in quantum information, offering a principled way to interpolate between efficiently simulable instances and classically intractable ones.

Appendix A

Annealer parameter fine-tuning

Below are the benchmarks of the simulated annealer corresponding to the following suboptimal parameter configurations:

1. $\alpha = 0.95$, $T_0 = 10$, and $T_{end} = 0.001$ (Table A.1),
2. $\alpha = 0.99$, $T_0 = 1$, and $T_{end} = 0.001$ (Table A.2),
3. $\alpha = 0.99$, $T_0 = 2$, and $T_{end} = 0.001$ (Table A.3),
4. $\alpha = 0.99$, $T_0 = 5$, and $T_{end} = 0.001$ (Table A.4),
5. $\alpha = 0.99$, $T_0 = 20$, and $T_{end} = 0.001$ (Table A.5).

The score function is set to $score\_square(T)$ everywhere. For the optimal configuration $\alpha = 0.99$, $T_0 = 10$, and $T_{end} = 0.001$, refer to Table 6.2.

Circuit	**Q**	**G**	$\mathbf{R_{init}}$	$\mathbf{R_{ann}}$	$\mathbf{ScFl_{init}}$	$\mathbf{ScFl_{ann}}$	**ΔScFl**
Adder8	23	105	22	22	27.432	27.432	0.000
QFT8	8	148	8	8	13.738	13.711	0.027
barenco_tof_3.qasm	5	20	5	6	9.077	8.966	0.111
barenco_tof_4.qasm	7	34	7	8	11.355	11.219	0.136
barenco_tof_5.qasm	9	50	9	9	13.752	13.752	0.000
csla_mux_3_original	15	50	12	12	17.698	17.698	0.000
csum_mux_9_corrected	30	56	29	29	33.250	33.250	0.000
gf2^4_mult	12	51	12	12	18.342	18.342	0.000
ham15-low.qc	17	167	17	17	24.358	24.358	0.000
hwb6.qc	7	79	7	7	14.176	14.176	0.000
mod_mult_55	9	35	9	9	15.203	14.809	0.394
mod_red_21	11	74	11	11	17.722	17.721	0.001
qcla_com_7	24	95	24	24	30.174	30.174	0.000
qft_4.qasm	5	159	5	6	11.811	11.587	0.224
rc_adder_6	14	68	14	14	19.871	19.858	0.012
tof_10	19	51	19	19	24.778	24.761	0.017
tof_3	5	9	5	4	9.209	8.087	1.122
tof_4	7	15	7	7	11.980	11.980	0.000
tof_5	9	21	9	9	14.076	13.575	0.502
vbe_adder_3	10	30	10	8	14.015	12.109	1.906

Table A.1: Annealer benchmark for $\alpha = 0.95$, $T_0 = 10$, and $T_{end} = 0.001$.

Circuit	**Q**	**G**	$\mathbf{R_{init}}$	$\mathbf{R_{ann}}$	$\mathbf{ScFl_{init}}$	$\mathbf{ScFl_{ann}}$	**ΔScFl**
Adder8	23	105	22	22	27.432	27.432	0.000
QFT8	8	148	8	8	13.738	13.711	0.027
barenco_tof_3.qasm	5	20	5	3	9.077	7.492	1.585
barenco_tof_4.qasm	7	34	7	6	11.355	10.600	0.755
barenco_tof_5.qasm	9	50	9	7	13.752	12.747	1.005
csla_mux_3_original	15	50	12	12	17.698	17.698	0.000
csum_mux_9_corrected	30	56	29	29	33.250	33.250	0.000
gf2^4_mult	12	51	12	12	18.342	18.342	0.000
ham15-low.qc	17	167	17	17	24.358	24.358	0.000
hwb6.qc	7	79	7	7	14.176	14.176	0.000
mod_mult_55	9	35	9	9	15.203	14.807	0.396
mod_red_21	11	74	11	11	17.722	17.721	0.001
qcla_com_7	24	95	24	24	30.174	30.174	0.000
qft_4.qasm	5	159	5	6	11.811	11.591	0.220
rc_adder_6	14	68	14	14	19.871	19.858	0.013
tof_10	19	51	19	19	24.778	24.761	0.017
tof_3	5	9	5	3	9.209	7.322	1.887
tof_4	7	15	7	6	11.980	9.401	2.579
tof_5	9	21	9	6	14.076	11.236	2.840
vbe_adder_3	10	30	10	5	14.015	9.401	4.614

Table A.2: Annealer benchmark for $\alpha = 0.99$, $T_0 = 1$, and $T_{end} = 0.001$.

Circuit	**Q**	**G**	$\mathbf{R_{init}}$	$\mathbf{R_{ann}}$	$\mathbf{ScFl_{init}}$	$\mathbf{ScFl_{ann}}$	**ΔScFl**
Adder8	23	105	22	16	27.432	27.105	0.327
QFT8	8	148	8	8	13.738	13.711	0.027
barenco_tof_3.qasm	5	20	5	3	9.077	7.392	1.684
barenco_tof_4.qasm	7	34	7	4	11.355	8.907	2.448
barenco_tof_5.qasm	9	50	9	7	13.752	12.747	1.005
csla_mux_3_original	15	50	12	12	17.698	17.698	0.000
csum_mux_9_corrected	30	56	29	29	33.250	33.250	0.000
gf2^4_mult	12	51	12	12	18.342	18.342	0.000
ham15-low.qc	17	167	17	17	24.358	24.358	0.000
hwb6.qc	7	79	7	7	14.176	14.176	0.000
mod_mult_55	9	35	9	8	15.203	14.577	0.626
mod_red_21	11	74	11	11	17.722	17.721	0.001
qcla_com_7	24	95	24	24	30.174	30.174	0.000
qft_4.qasm	5	159	5	6	11.811	11.591	0.220
rc_adder_6	14	68	14	14	19.871	19.858	0.012
tof_10	19	51	19	19	24.778	24.761	0.017
tof_3	5	9	5	3	9.209	7.322	1.888
tof_4	7	15	7	5	11.980	9.925	2.055
tof_5	9	21	9	6	14.076	10.512	3.564
vbe_adder_3	10	30	10	5	14.015	9.989	4.026

Table A.3: Annealer benchmark for $\alpha = 0.99$, $T_0 = 2$, and $T_{end} = 0.001$.

Circuit	**Q**	**G**	$\mathbf{R_{init}}$	$\mathbf{R_{ann}}$	$\mathbf{ScFl_{init}}$	$\mathbf{ScFl_{ann}}$	**ΔScFl**
Adder8	23	105	22	22	27.432	27.432	0.000
QFT8	8	148	8	8	13.738	13.711	0.027
barenco_tof_3.qasm	5	20	5	3	9.077	7.392	1.684
barenco_tof_4.qasm	7	34	7	8	11.355	11.219	0.136
barenco_tof_5.qasm	9	50	9	7	13.752	11.750	2.002
csla_mux_3_original	15	50	12	12	17.698	17.698	0.000
csum_mux_9_corrected	30	56	29	29	33.250	33.250	0.000
gf2^4_mult	12	51	12	12	18.342	18.342	0.000
ham15-low.qc	17	167	17	17	24.358	24.358	0.000
hwb6.qc	7	79	7	7	14.176	14.176	0.000
mod_mult_55	9	35	9	9	15.203	14.809	0.395
mod_red_21	11	74	11	11	17.722	17.721	0.001
qcla_com_7	24	95	24	24	30.174	30.174	0.000
qft_4.qasm	5	159	5	6	11.811	10.989	0.822
rc_adder_6	14	68	14	14	19.871	19.858	0.013
tof_10	19	51	19	12	24.778	20.468	4.310
tof_3	5	9	5	3	9.209	7.170	2.039
tof_4	7	15	7	4	11.980	8.714	3.266
tof_5	9	21	9	4	14.076	10.028	4.048
vbe_adder_3	10	30	10	5	14.015	9.721	4.294

Table A.4: Annealer benchmark for $\alpha = 0.99$, $T_0 = 5$, and $T_{end} = 0.001$.

Circuit	**Q**	**G**	$\mathbf{R_{init}}$	$\mathbf{R_{ann}}$	$\mathbf{ScFl_{init}}$	$\mathbf{ScFl_{ann}}$	**ΔScFl**
Adder8	23	105	22	15	27.432	27.051	0.381
QFT8	8	148	8	8	13.738	13.711	0.027
barenco_tof_3.qasm	5	20	5	3	9.077	7.585	1.492
barenco_tof_4.qasm	7	34	7	5	11.355	9.435	1.921
barenco_tof_5.qasm	9	50	9	8	13.752	12.137	1.615
csla_mux_3_original	15	50	12	12	17.698	17.698	0.000
csum_mux_9_corrected	30	56	29	29	33.250	33.250	0.000
gf2$\hat{4}$_mult	12	51	12	12	18.342	18.342	0.000
ham15-low.qc	17	167	17	17	24.358	24.358	0.000
hwb6.qc	7	79	7	7	14.176	14.176	0.000
mod_mult_55	9	35	9	8	15.203	14.114	1.089
mod_red_21	11	74	11	11	17.722	17.721	0.001
qcla_com_7	24	95	24	24	30.174	30.174	0.000
qft_4.qasm	5	159	5	6	11.811	11.605	0.205
rc_adder_6	14	68	14	14	19.871	19.858	0.012
tof_10	19	51	19	19	24.778	24.761	0.017
tof_3	5	9	5	3	9.209	7.129	2.080
tof_4	7	15	7	5	11.980	9.531	2.449
tof_5	9	21	9	6	14.076	12.420	1.656
vbe_adder_3	10	30	10	4	14.015	8.954	5.061

Table A.5: Annealer benchmark for $\alpha = 0.99$, $T_0 = 20$, and $T_{end} = 0.001$.

Bibliography

[1] Miriam Backens, Hector Miller-Bakewell, Giovanni de Felice, Leo Lobski, and John van de Wetering. There and back again: A circuit extraction tale. *Quantum*, 5:421, March 2021.

[2] Martin Beyß. Fast algorithm for rank-width. In Antonín Kučera, Thomas A. Henzinger, Jaroslav Nešetřil, Tomáš Vojnar, and David Antoš, editors, *Mathematical and Engineering Methods in Computer Science*, pages 82–93, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] Sergey Bravyi, Dan Browne, Padraic Calpin, Earl Campbell, David Gosset, and Mark Howard. Simulation of quantum circuits by low-rank stabilizer decompositions. *Quantum*, 3:181, September 2019.

[4] Sergey Bravyi, Graeme Smith, and John A. Smolin. Trading classical and quantum computational resources. *Physical Review X*, 6(2), June 2016.

[5] Julien Codsi. Cutting-edge graphical stabiliser decompositions for classical simulation of quantum circuits, 2022.

[6] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017.

[7] Luke Heyfron and Earl T. Campbell. An efficient quantum compiler that reduces t count, 2018.

[8] Sang il Oum. Approximation algorithm for the clique-width. 2003. https://mathsci.kaist.ac.kr/ sangil/pdf/rwdslide.pdf.

[9] Sang il Oum. Rank-width: Algorithmic and structural results. *Discrete Applied Mathematics*, 231:15–24, 2017. Algorithmic Graph Theory on the Adriatic Coast.

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

[11] 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.

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

[13] Aleks Kissinger, John van de Wetering, and Renaud Vilmart. Classical simulation of quantum circuits with partial and graphical stabiliser decompositions. Schloss Dagstuhl Leibniz-Zentrum fr Informatik, 2022.

[14] Tuomas Laakkonen. Graphical stabilizer decompositions for counting problems, 2022.

[15] Mehdi Mhalla and Simon Perdrix. *Finding Optimal Flows Efficiently*, pages 857–868. Springer Berlin Heidelberg, 2008.

[16] Florian Nouwt. A simulated annealing method for computing rank-width, 2022.

[17] Snehal Raj. Graphical calculus for tensor network contractions, 2022.

[18] Francisco J. R. Ruiz, Tuomas Laakkonen, Johannes Bausch, Matej Balog, Mohammadamin Barekatain, Francisco J. H. Heras, Alexander Novikov, Nathan Fitzpatrick, Bernardino Romera-Paredes, John van de Wetering, Alhussein Fawzi, Konstantinos Meichanetzidis, and Pushmeet Kohli. Quantum circuit optimization with alphatensor. *Nature Machine Intelligence*, 7(3):374–385, Mar 2025.

[19] Will Simmons. Relating measurement patterns to circuits via pauli flow. *Electronic Proceedings in Theoretical Computer Science*, 343:50101, September 2021.

[20] Matthew Sutcliffe. Smarter k-partitioning of zx-diagrams for improved quantum circuit simulation, 2024.

[21] Matthew Sutcliffe and Aleks Kissinger. Procedurally optimised zx-diagram cutting for efficient t-decomposition in classical simulation. *Electronic Proceedings in Theoretical Computer Science*, 406:6378, August 2024.

[22] Matthew Sutcliffe and Aleks Kissinger. Fast classical simulation of quantum circuits via parametric rewriting in the zx-calculus, 2025.