Computational Complexity Perspective on Graphical Calculi for Quantum Computation

by

PIOTR B. MITOSEK

A thesis submitted to the University of Birmingham for the degree of

DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham

May 2025

Abstract

Diagrammatic reasoning via languages such as ZX Calculus is a powerful tool in theoretical quantum computing. However, physical quantum computers do not understand diagrams and require explicit quantum circuits as inputs. Finding a circuit equivalent to the given diagram is known as circuit extraction. Such problems in graphical calculi form a limitation preventing the wider adoption of diagrammatic reasoning.

Computational complexity theory formalises the difficulty of problems. For ZX diagrams, the circuit extraction problem is $\#\boldsymbol{P}$-hard and unlikely to have an efficient algorithm. Therefore, partial solutions are critical. Currently, efficient circuit extraction is only possible for diagrams containing a structure of so-called Pauli flow. This structure is necessary and sufficient for robust determinism in measurement-based quantum computation (MBQC) – a model of quantum computation, an alternative to circuits, where adaptive single-qubit measurements drive computation. Thus, Pauli flow links MBQC, ZX Calculus, and the circuit model. It is known how to find flow structures in polynomial time when they exist; nevertheless, their lengthy and complex definitions often hinder working with them.

In this thesis, I present contributions to two connected topics: the hardness of problems in graphical calculi and Pauli flow as a method for overcoming circuit extraction. First, I show that circuit extraction can be $\#\boldsymbol{P}$-hard for phase-free ZH diagrams, extending the previous result for ZX diagrams. I also establish a relation between phase-free ZH and $\boldsymbol{NP}^{\#\boldsymbol{P}}$ class. This involves weakening the oracle, crafting a complete problem, and encoding the problem into phase-free ZH.

My next contribution concerns Pauli flow. I simplify the Pauli flow definition by providing a new algebraic interpretation. This involves defining two matrices arising from the adjacency matrix of the underlying graph. Then, Pauli flow corresponds to the existence of a right-inverse of the first matrix, which, when multiplied by the second matrix, results in a matrix of a directed acyclic graph. Based on the newly defined algebraic interpretation, I obtain $O(n^3)$ algorithms for finding Pauli flow, improving on the previous $O(n^4)$ bounds. I also introduce a lower bound for Pauli flow-finding, by linking it to linear algebra problems over $\mathbb{F}_2$. Finally, when given an unlabelled open graph, I show how to find measurement labelling resulting in Pauli flow.

Acknowledgements

I am incredibly grateful to everyone who helped me during PhD studies.

First and foremost, I thank my supervisor, Miriam Backens. When I approached them in early 2021, I did not know what a PhD would entail. They have guided me as an expert in the field. They helped me polish my ideas, as well as in the process of writing and preparing talks. They also aided me with other aspects of the PhD, such as conference applications and looking for future positions. Finally, they co-wrote a paper with me. Without the fantastic supervisor, I doubt I would have completed my degree.

I extend my sincere thanks to my second supervisor, Paul Levy. Paul was invaluable, especially during my third year, when my contact with Miriam was more limited. He was always available to meet and discuss my progress. He was also of great help when I got lost in university regulations while preparing for thesis submission.

Next, I would like to thank all my friends for helping me through the struggles of my studies. Double thanks to Michał for being a great friend and withstanding my endless questions about linear algebra. I thank Tommy for all our discussions. I also thank my friends from Ohana and Stacja – going through a PhD would be less fun without you. Last but not least, I am grateful to all flatmates with whom I lived in Birmingham.

I appreciate my family, especially my Mum, for supporting my life choices. I could not have undertaken this journey without them.

Finally, I want to thank everyone in the School of Computer Science. I am especially grateful for the financial support, equipment, and office space.

Contents

Chapter 1

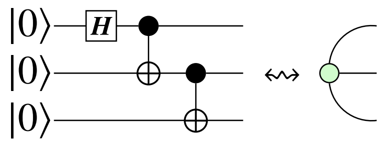
Introduction

Quantum computing offers capabilities beyond those of classical computers. Some problems are best suited for quantum computers (e.g. [83, 152]), and protocols that are impossible on classical machines exist (e.g. [21]). From a practical point of view, some results suggest we are crawling into the quantum supremacy era [141, 92]. Nonetheless, quantum computation is extremely difficult to realise in practice and is an ongoing topic of multi-disciplinary research spanning physics, mathematics, computer science, and engineering. Quantum physics is notorious for being unintuitive due to the seeming randomness of processes on the particle scale. Understanding these processes was and remains a critical challenge encountered by physicists.

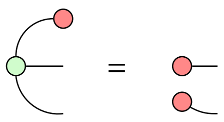
For a long time, quantum physics and computer science were disjoint fields. Only in the 80s did people consider whether quantum phenomena could be applied for computational purposes [19, 20]. Nowadays, Quantum computing is rapidly developing. However, its relatively young age means that jargon and notation constantly evolve. Sometimes, quantum computation borrows directly from physics, for example, adopting Dirac's notation. To perform quantum computation in practice, one must consider the model of quantum computation implemented by physical quantum computers. Usually, this is a quantum circuit model, see for example [71, 97]. For that reason, quantum circuits form the standard presentation of quantum computation in computer science. While circuits can work great for machines, they do not create a language ideal for humans. Recently, alternative representations of quantum computing are becoming more and more popular. In this thesis, I work with graphical calculi for quantum computation that arise from categorical quantum mechanics [88]. The most essential among these is the ZX Calculus [45, 171], which has applications in nearly all areas of theoretical quantum computing.

In graphical calculi, quantum processes are represented with string diagrams. Transformations between processes can be derived via simple rewrite rules. Graphical representation of quantum computation simplifies many of the issues with quantum circuits.

Firstly, they are often more efficient. For instance, consider Greenberger-Horne-Zeilinger state $|GHZ\rangle$ [81] preparation in circuit model and the representation of the same state in ZX Calculus:



The quantum processes are also significantly simplified. For instance, suppose that the first qubit of the $|GHZ\rangle$ state is measured in the computational basis and the outcome $\langle 0|$ is observed. This brings the remaining qubits into state $|00\rangle$. Graphically, in ZX Calculus, ignoring the scalars, this fact can be expressed as follows:



ZX Calculus is universal, sound, and complete (for an overview, see [171]). This means that any quantum process can be represented in ZX Calculus, rewrite rules correspond to equalities between quantum processes, and a sequence of rewrite rules can realise any transformation between processes. Similar results hold for other graphical calculi. For these reasons, graphical calculi offer no disadvantage from the theoretical perspective over circuit representation. More importantly, however, graphical reasoning transforms computation in unachievable or extremely tedious ways when attempted purely with circuits. For example, quantum circuits are suboptimal for reasoning about non-unitaries. This is especially evident when considering state preparation via circuits or the measurement effects. Diagrammatic reasoning does not face such obstacles. This is a double-edged sword, though, as sometimes graphical reasoning can result in a diagram that is hard to bring into language understood by a quantum computer, i.e. into a circuit. The problem of transforming a diagram to a circuit is called circuit extraction. While circuit extraction can be solved algorithmically, it can be costly from a computational perspective [62] and challenges the broader adoption of graphical calculi for quantum computation.

The study of resources necessary to solve a computational task is computational complexity theory. This subfield of computer science makes it possible to formalise the difficulty of the tasks, classify the problems based on the resources needed to solve them, and study the relations between problems. In the case of circuit extraction in the previous paragraph, it was shown that the problem is $\#\boldsymbol{P}$-hard [62]. While the exact meaning of this result will be clarified later on, the problem is likely impossible to solve efficiently with a classical (or even quantum) computer [4]. Similarly, other problems in graphical calculi can be shown to be computationally challenging. For instance, while completeness guarantees that any equality between quantum processes can be proven with graphical rewrite rules, we do not have an efficient algorithm verifying that two diagrams represent the same process. Most completeness proofs must go through normal forms that are exponential in the size of the initial diagrams, for example, see [98, 99]. Consequently, exponentially many applications of rewrite rules may be necessary to rewrite one diagram into another.

Due to circuit extraction's criticality, researchers have looked for efficient algorithms that can extract a circuit in at least some cases. While we cannot expect a general and efficient solution, partial solutions exist [64, 15, 153]. The broadest class of ZX diagrams for which such algorithms exist are the diagrams containing a structure of the so-called Pauli flow [29].

While the circuit model is the most common realisation of quantum computation, other models have been successfully defined. Measurement-based Quantum Computation (MBQC) is an example of an alternative that is at least as powerful as the circuit model [143, 144, 145]. In this model, quantum computation is driven forward entirely by adaptive single-qubit measurements of a highly entangled resource state. Due to the non-deterministic nature of quantum measurements, the central question in MBQC is whether the intended computation can be performed deterministically, see for example [52, 29, 125, 128]. The answer is yes if and only if a description of the desired state, given by what we call a labelled open graph, contains Pauli flow [29, 128]. This way, Pauli flow links MBQC, ZX Calculus, and extraction to the circuit model. It is known how to find Pauli flow and similar structures in deterministic polynomial time, when they exist [55, 56, 126, 15, 153].

While Pauli flow allows one to overcome the problem of circuit extraction from ZX Calculus, it comes with its issues. Firstly, the standard definition of Pauli flow is very long and complex. It hinders Pauli flow usability: some previous publications encountering circuit extraction problems looked for solutions involving inferior flow structures, such as generalised flow (gflow) or causal flow, for example [157]. Secondly, existing algorithms involving Pauli flow, while more general than those for other flow variants, are slow.

1.1 Contributions

In this thesis, I research graphical calculi for quantum computation from a computational complexity perspective. I formally state the limitations of graphical calculi by deriving the complexity hardness results of various problems appearing in graphical calculi [131]. I also contribute to methods for overcoming these limitations by improving the presentation of the Pauli flow structure [133] and developing new fastest algorithms for finding it

[132, 133].

The family of ZX-like graphical calculi for quantum computation is constantly growing. One of the simplest of them that is still (approximately) universal for quantum computation is the phase-free ZH Calculus [14]. This way, phase-free ZH provides an excellent case study for computational complexity approaches: Hardness results for phase-free ZH often extend to other graphical calculi, like ZX, (full) ZH, or ZW. I show that specific problems arising in phase-free ZH are $\boldsymbol{NP}^{\#\boldsymbol{P}}$-hard. These problems connect to the problem of comparing diagrams, i.e. checking whether linear processes represented by two diagrams match. While comparing diagrams asks for a universal property of matrix representations of diagrams matching in all positions, my results consider problems where the universal property is changed to an existential property. To prove $\boldsymbol{NP}^{\#\boldsymbol{P}}$-hardness, I craft one of the first examples of complete problems for this complexity class. Further, I extend the previously known $\#\boldsymbol{P}$-hardness of circuit extraction in ZX Calculus [62] to work for phase-free ZH.

I simplify the presentation of Pauli flow conditions by constructing an algebraic interpretation of flow. The new algebraic interpretation simplifies the flow condition to simple matrix operations, like finding the inverse and multiplication. This streamlines the process of verifying flow. Since Pauli flow is a generalisation of other flow structures, all of my results also apply to those weaker flow variants, like gflow. In the case of gflow, a previous algebraic interpretation existed, but it was limited only to a special case where all qubits are measured in the same plane of the Bloch sphere [125]. I also prove various properties of flow. Notably, this includes showing when and how flow can be reversed.

Finally, building on the new algebraic interpretation of flow, I created new algorithms for the Pauli flow-finding problem. In particular, I reduced the problem complexity to match the complexity of Gaussian elimination. My algorithms not only provide a speed-up over previous algorithms, but I also argue that further improvements must lead to, or more likely come from, new developments in algorithmics for linear algebra. Finally, I show that given a resource state, in the form of a graph state, for MBQC, it is always possible to determine the existence of a measurement choice that results in Pauli flow and, thus, robustly deterministic computation.

1.2 Structure

The rest of the thesis is structured as follows.

In Chapter 2, I provide computational background material and an overview of relevant literature. In terms of length, it is the longest chapter because it combines introductions to both computational complexity theory and quantum computation. From the quantum side, I focus mainly on MBQC and flow structures in this model.

Next, in Chapter 3, I discuss graphical calculi for quantum computation. This includes a general explanation of string diagrams and a more detailed look at ZX and ZH calculi. Together with the previous chapter, these chapters lay the ground for my novel results in the next three chapters.

The topics of Chapter 4 are the limitations of graphical calculi from a computational complexity perspective. Here, I show hardness results for various problems in graphical calculi. I adapt the Cook-Levin approach to construct a complete problem for the obscure complexity class $\boldsymbol{NP^{\#P}}$. Next, I explain how this problem can be encoded in phase-free ZH Calculus, and hence in most universal ZX-like graphical calculi. I also extended the previous result of circuit extraction to work for phase-free ZH.

After that, I switched to the analysis of flow structures in MBQC. In Chapter 5, I derive a new algebraic interpretation of Pauli flow, simplifying the previous definitions. I also prove various properties of flow implied by the new algebraic interpretation.

The last main-body part of my thesis is in Chapter 6. Here, I work out new flow algorithms, providing examples, pseudocode, and detailed analysis of their complexity. These include two algorithms for Pauli flow-finding, and an algorithm for finding measurement labelling resulting in Pauli flow.

Finally, I close this work in Chapter 7, by providing a summary, conclusions, and directions for further work. Afterwards, the bibliography follows.

1.3 Authorship

This thesis documents my results as a PhD student and is mainly constructed based on my three articles [131, 132, 133]. I am the first author of all of these papers, and the single author of the first two of them. The last one was co-written with my supervisor Miriam Backens. In particular, I found all the main proofs and wrote the initial version of the paper. Miriam wrote most of the paper's introduction, simplified the proof of flow reversibility, and improved the presentation of the results overall.

Chapter 2

Computational Background

In this chapter, we provide a background of the computational theory. By computation we mean any well-defined form of calculation. For us, the formal description of computation is via algorithms, i.e. we are interested in whether a problem can be solved in a rigorous way by following a sequence of instructions. Or, in other words, we are studying processes that can be achieved on Turing machines. We start with an example of a problem. Suppose that given a natural number we want to know whether the number is prime. We can formalise the task by defining the following *problem*:

> **PrimeTest**
> **Input:** a natural number $n \in \mathbb{N}$.
> **Output:** *True* when n is prime and *False* otherwise.

One could *solve* this problem algorithmically in the following way: first, if $n = 1$, output $False$ since 1 is not prime. Next, check whether n is divisible by any of the numbers in the range from 2 to $n - 1$. If so, then n is not prime. Otherwise, n is prime. Such a method shows that the problem **PrimeTest** is decidable. However, this method lacks efficiency: it is possible to solve the problem in a faster manner. For example, we can stop on the first found divisor and only have to test divisors up to $\sqrt{n}$. We can also use number theory results to test whether the number is prime in an even faster manner, for example, by utilising the AKS test [5], or the probabilistic Rabin-Miller test [142].

We consider computational complexity theory, an area of computer science that groups problems into classes and studies relations between them. Thus, computational complexity provides a framework formalising the meaning of terms such as ‘faster’, ‘more efficient’ etc. Many of the classes we look at are defined with Turing machines which capture the power of computation that can be performed in algorithmically rigorous way. Sometimes an algorithm for solving a problem requires an algorithm for another underlying problem: in the **PrimeTest** above, we assumed one can check whether n is divisible by another number. The approach of using a method for one problem to solve another problem can be formalised by oracle machines, a special subtype of Turing machines.

As this thesis is about quantum computation, we also provide an overview of the most essential topics in quantum computation. From the quantum side, we focus mainly on a general description of the circuit model and provide a more detailed dive into measurement-based quantum computation.

The main purposes of this chapter are to disambiguate between various equivalent yet slightly different definitions appearing in the literature and to provide general motivation for researching topics covered in the later chapters.

Structure: In Section 2.1, we introduce basic definitions from computational complexity. Next, in Section 2.2 we define Turing machines. We formalise and study the relation of various complexity classes in Section 2.3. After that, we switch to the quantum side of computation. We will consider two computational models arising from the mathematical model of quantum mechanics: the circuit model in Section 2.4 and the measurement-based quantum computation model in Section 2.5. We end the chapter by looking at the so-called flow structures arising in the measurement-based model in Section 2.6.

2.1 Computational Complexity

Computational complexity is an area of science that studies the resources necessary to solve a given problem, grouping problems that require similar resources into classes, and studies relations between classes.

We provide only a minimal description of basic definitions. A reader looking for a comprehensive introduction to the subject should consult any computational complexity textbook, for example [9].

The problems are defined as subsets of words over a given alphabet:

Definition 2.1.1. An *alphabet* Σ is any finite set with at least two elements. The set of finite words over alphabet Σ is denoted Σ^*.

Definition 2.1.2. A *language* or *decision problem* $\mathbf{Q}$ over alphabet Σ is a subset of Σ^*. An *input* i to problem $\mathbf{Q}$ is an element of Σ^*, i.e. any finite word over Σ. To *answer the problem* for a given input i, it is to determine whether $i \in \mathbf{Q}$.

While the above definition formalises the meaning of a problem, it is hard to work with. Typically, we do not think of problems as sets, but as questions with a ‘yes’ or ‘no’ answer:

Definition 2.1.3. Informally, a *decision problem* **Q** consists of a set of *inputs* I and *accepted inputs* A where given input i a problem is to determine whether $i \in A$.

The translations from informal descriptions to set-based definitions is possible and follows approach of defining suitable encoder and decoder functions. For instance, we return to the problem **PrimeTest**:

Example 2.1.4. It is possible to present **PrimeTest** as a language over $\Sigma = \{0, 1\}$. We can define a binary encoding function $f : \mathbb{N} \to \Sigma^*$ as follows: given $n \in \mathbb{N}$, let $f(n)$ be n written in binary. Then define **PrimeTest** as a language over $\Sigma = \{0, 1\}$ given by a subset $\{w \in \Sigma^* \mid \exists n \in \mathbb{N}. f(n) = w \wedge n \text{ is prime}\}$. To interpret such language back as a 'yes' or 'no' question, one must define a decoder $d : \mathbb{N} \to \Sigma^*$ such that $d \circ f = Id$, which is possible as f is clearly injective.

In the above example, we used binary representation of number n. This is a common practice which we will assume whenever a problem is given an integer in the input.

It should be clear that, in general, we want to avoid such low level encodings as much as possible. As a rule of thumb, all problems we look at can always be translated to languages. It requires the sets of inputs to always be countable and it must possible to formalise encoder and decoder functions. However, sometimes set definitions are more convenient, for instance when formalising the complement problem:

Definition 2.1.5. Given a problem **Q** over Σ, its *complement* **coQ** is defined as $\Sigma^* \setminus \mathbf{Q}$.

We also define function problems:

Definition 2.1.6. A *function problem* **R** over alphabet Σ is a relation over Σ^*. An *input* i to problem **R** is an element of Σ^*, i.e. any finite word over Σ. To *answer the problem* for a given input i is to determine any $j \in \Sigma^*$ such that $(i, j) \in \mathbf{R}$.

This definition is even harder to work with than that of decision problem. Instead, we think of function problems as a search problem: given an input, find an output that satisfies a required property. An example of function problem is as follows:

FindDivisor
Input: a natural number $n \in \mathbb{N}$.
Output: a natural number k such that $k \mid n$ and $2 \leq k \leq n - 1$, or a message that no such number exists.

Again, it is possible to present the above problem as a relation, but this time, we would need an encoder for both inputs (natural numbers) and possible answers (natural numbers and a message that no required number exists). We skip the details.

We can now define complexity classes:

Definition 2.1.7. A *class of decision problems* is a set of decision problems, and a *class of function problems* is a set of function problems.

Notation. We use bold font to denote problems and bold italic font to denote complexity classes.

For decision problems, we also define the complement class:

Definition 2.1.8. For a class of decision problems $\boldsymbol{C}$, we define its *complement* $\boldsymbol{coC}$ as the set of all decision problems $\mathbf{Q}$ such that $\mathbf{coQ} \in \boldsymbol{C}$.

The complement of the complexity class is not the same as the set of all problems not in the complexity class:

Example 2.1.9. The class $\boldsymbol{NONE}$ consists of no problems. The class $\boldsymbol{ALL}$ consists of all problems. For any problem $\mathbf{Q}$, we have $\mathbf{coQ} \in \boldsymbol{ALL}$ and $\mathbf{coQ} \notin \boldsymbol{NONE}$, as $\mathbf{coQ}$ is a problem so it must be in $\boldsymbol{ALL}$ and cannot be in $\boldsymbol{NONE}$. In fact the following holds:

$$\boldsymbol{coNONE} = \boldsymbol{NONE} \neq \boldsymbol{coALL} = \boldsymbol{ALL}.$$

To define actually meaningful classes, we need the quintessential tool in computational complexity: Turing machines.

2.2 Turing machines

We assume familiarity with basic Turing machines. The primary purpose of this section is to disambiguate between various definitions of Turing machines (TMs) encountered in the literature. A reader acquainted with TMs can skip this section. The exact details will only be necessary when proving the completeness of problems from the TM definition for obscure complexity classes.

TMs form a mathematical formalism of algorithmic computation [42, 167]: Any computation that can be performed algorithmically can be accomplished on some Turing machine.

The machines can be defined in many different ways. Sometimes, their heads must move, and other times, they must remain immobile when overwriting a cell. Some machines operate on multiple tapes, while at other times the tape is one-sided. The working alphabet can vary. Sometimes, TMs must leave the answer on the tape in a specific way. For most purposes, all these variations are equivalent. To avoid ambiguity, we provide a definition (adapted version of the one in [9]).

Definition 2.2.1. A *k-tape non-deterministic Turing machine (NDTM) $\mathcal{M}$* over alphabet Σ is a tuple (Γ, Q, δ) where:

- Γ is the *working alphabet* of $\mathcal{M}$ such that $\Gamma = \Sigma \cup \{\square\}$ where $\square \notin \Sigma$. We call $\square$ the *blank symbol* of $\mathcal{M}$,
- Q is a finite set of *states* containing q_{START}, q_{HALT}: the *starting* and *halting* states respectively,
- $\delta\colon Q \times \Gamma^k \to \mathcal{P}(Q \times \Gamma^{k-1} \times \{-1, 0, +1\}^k) \setminus \{\emptyset\}$ is the *transition function* describing the operation of the Turing machine. For any $\sigma_1, \ldots, \sigma_k \in \Gamma$, it satisfies:

$$\delta(q_{HALT}, (\sigma_1, \ldots, \sigma_k)) = \{(q_{HALT}, (\sigma_2, \ldots, \sigma_k), (0, \ldots, 0))\}.$$

We are defining non-deterministic Turing machines first and we also upfront allow the machine to be multi-tape. These choices are made to avoid repeating very similar definitions later on. We disallow $\emptyset$ in the image of δ so that a machine can always take a step. A 'dead end' allowed in many definitions of TMs can still be simulated.

We also define a deterministic Turing machine (DTM):

Definition 2.2.2. A deterministic Turing machine is a tuple (Γ, Q, δ) where Γ and Q are as in a non-deterministic Turing machine and $\delta\colon Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{-1, 0, 1\}^k$ is the transition function, again satisfying for any $\sigma_1, \ldots, \sigma_k \in \Gamma$:

$$\delta(q_{HALT}, (\sigma_1, \ldots, \sigma_k)) = (q_{HALT}, (\sigma_2, \ldots, \sigma_k), (0, \ldots, 0)).$$

All further definitions are written for NDTMs, but extend to DTMs by observing that a DTM corresponds to a NDTM in which the image of the transition function contains singletons only.

Next, we define configuration of a Turing machine $\mathcal{M}$:

Definition 2.2.3. At any time, the *configuration* of a Turing machine $\mathcal{M}$ is a tuple $(q, (T_1, \dots, T_k), (H_1, \dots, H_k))$ where q is the current state, $T_1, \dots, T_k$ are contents of the tapes and $H_1, \dots, H_k$ are the positions of the heads.

The meaning of the transition function is captured by the definition of a step:

Definition 2.2.4. If $\mathcal{M}$ is in configuration $(q, (T_1, \dots, T_k), (H_1, \dots, H_k))$, with $\sigma_i = T_i(H_i)$ for $i \in [1..k]$, and $\delta(q, (\sigma_1, \dots, \sigma_k)) \ni (q', (\sigma_2', \dots, \sigma_k'), (d_1, \dots, d_k))$, then $\mathcal{M}$ can move to configuration $(q', (T_1', \dots, T_k'), (H_1 + d_1, \dots, H_k + d_k))$ where $T_1' = T_1$, and for $i \geq 2$, T_i' differ from T_i possibly only in H_i position, where $T_i'(H_i) = \sigma_i'$. The act of changing the configuration in a valid way is called a *step*.
A *path* is an infinite sequence of steps.

A DTM can always take precisely one step from any configuration.
We can now formalise the run a Turing machine on a given input:

Definition 2.2.5. The *input* to the machine $\mathcal{M}$ is any word $w \in \Sigma^*$. The *initial configuration* of $\mathcal{M}$ is $(q_{START}, (T_1, T_2, \dots, T_k), (0, \dots, 0))$ where T_1 contains w on positions from 0 to $|w| - 1$ and $\square$ otherwise, while $T_2, \dots, T_k$ all contain $\square$ symbols only.

Definition 2.2.6. If a path p from starting configuration eventually reaches configuration c with state q_{HALT} after r steps, we say that p is a *halting path* with *terminal configuration* c and *length* r. We say that *memory usage* of path p is the sum of maximal absolute values of head positions across all tapes.
If all paths from starting configuration are halting, we say that $\mathcal{M}$ *halts* on input w. Otherwise, we say that $\mathcal{M}$ *hangs* on input w.
If $\mathcal{M}$ halts on all inputs, we call it *halting*.

In particular, a DTM has precisely one path for a fixed input.
Finally, we define the output, runtime and memory usage of Turing machine as follows:

Definition 2.2.7. If $\mathcal{M}$ halts on input w then:

- the *output* of $\mathcal{M}$ on w is the fragment of T_k on positions from 0 to H_k on any terminal configuration $(q_{HALT}, (T_1, \dots, T_k), (H_1, \dots, H_k))$,
- the *runtime* of $\mathcal{M}$ on w is the maximal runtime across all paths,
- the *memory usage* of $\mathcal{M}$ on w is the maximal memory usage across all paths.

The first tape of the machine never changes. Defining a machine this way allows one to easily define classes that operate on sub-linear memory. However, we will only be interested in polynomial bounds.

We also define a decision variant of a Turing machine that only outputs Boolean values:

Definition 2.2.8. A *decision* Turing machine is a Turing machine whose all possible outputs are 0 and 1. A path producing output 0 is called *rejecting* and a path producing output 1 is called *accepting*. A deterministic decision Turing machine *accepts* w if it halts and its unique path on w is accepting and it *rejects* w if it halts and its unique path on w is rejecting. Sometimes, instead of having a machine produce output when it reaches q_{HALT}, we say that the machine enters q_{ACC} or q_{REJ}, i.e. accepting or rejecting state at which the machine halts. The two formulations are equivalent.

When working with deterministic Turing machines, it is useful to formalise the outputs as a function:

Definition 2.2.9. Let $\mathcal{M}$ be a deterministic halting Turing machine. For each input w, we define $f_{\mathcal{M}}(w)$ as an output produced by $\mathcal{M}$ when run on w. We say, that $\mathcal{M}$ *computes* $f_{\mathcal{M}}$ and that $f_{\mathcal{M}}$ is *computed* by $\mathcal{M}$. When a function f equals $f_{\mathcal{M}}$ for some deterministic halting Turing machine, we say that f is *computable*.

To eliminate the need of repeats in the next section, we define one more property:

Definition 2.2.10. A *polynomial-time* Turing machine $\mathcal{M}$ is a halting Turing machine for which there exists polynomial p such that for all inputs w, the runtime of $\mathcal{M}$ on w is bounded above by $p(|w|)$.
We say that a function f is *polynomial-time computable* is there exists a polynomial-time deterministic Turing machine that computes f.

2.3 Complexity classes

Finally, we can define various complexity classes. We provide some known relations between them at the end of this section. In the greater part, this section is constructed using the Complexity Zoo, a great compendium displaying various species of complexity classes [2].

2.3.1 Standard classes

Definition 2.3.1. The *polynomial-time* (also known as *deterministic polynomial-time*) complexity class $\boldsymbol{P}$ is the complexity class of all decision problems **Q** solvable by a polynomial-time DTM.
A problem **Q** is solvable by a *polynomial-time DTM* when there exists a polynomial-time decision DTM machine $\mathcal{M}$ such that for all $w \in \Sigma^*$: $w \in \mathbf{Q}$ if and only if $\mathcal{M}$ accepts w.

As an example, **PrimeTest** is in $\boldsymbol{P}$, as given number n it is possible to check whether n is prime in time polynomial in $\log n$, which is the size of the input of **PrimeTest**:

Theorem 2.3.2 (Agrawal–Kayal–Saxena Primality Test [5])**.** **PrimeTest** $\in \boldsymbol{P}$.

Definition 2.3.3. The *non-deterministic polynomial-time* complexity class $\boldsymbol{NP}$ is the complexity class of all decision problems **Q** solvable by a polynomial-time NDTM.
A problem **Q** is solvable by a *polynomial-time NDTM* when there exists a polynomial-time decision NDTM machine $\mathcal{M}$ such that for all $w \in \Sigma^*$: $w \in \mathbf{Q}$ if and only if there exists an accepting path of $\mathcal{M}$ on w.

Definition 2.3.4. The *function polynomial-time* complexity class $\boldsymbol{FP}$ is the complexity class of all function problems **Q** such that there exists a polynomial-time computable function f such that for all inputs w we have $(w, f(w)) \in \mathbf{Q}$.

Definition 2.3.5. The *polynomial-space* complexity class $\boldsymbol{PSPACE}$ is the complexity class of all decision problems **Q** for which there exists a polynomial p and a deterministic decision Turing machine $\mathcal{M}$ such that for all $w \in \Sigma^*$: the memory usage of $\mathcal{M}$ on w is bounded above by $p(|w|)$ and $w \in \mathbf{Q}$ if and only if $\mathcal{M}$ accepts w.

When studying computational complexity, it is natural to ask what the ‘hardest’ problem is for a given class. Solving such a problem should be sufficient to solve all problems in the given class. To compare the hardness of problems, we use reductions. Generally, there are various types of reductions that depend on the underlying type of Turing machine. The polynomial-time reduction is the only type of reduction we use:

Definition 2.3.6. A problem $\mathbf{Q_1}$ is *polynomial-time reducible* to a problem $\mathbf{Q_2}$, denoted by $\mathbf{Q_1} \leq_p \mathbf{Q_2}$, if there exists a polynomial-time computable function f such that for all words w we have $w \in \mathbf{Q_1}$ if and only if $w \in \mathbf{Q_2}$.

Now, we are able to define hardness for the class $\boldsymbol{NP}$:

Definition 2.3.7. A problem $\mathbf{Q}$ is *$\boldsymbol{NP}$-hard* if for all problems $\mathbf{R} \in \boldsymbol{NP}$ we have $\mathbf{R} \leq_p \mathbf{Q}$. A problem is *$\boldsymbol{NP}$-complete* if it is $\boldsymbol{NP}$-hard and in $\boldsymbol{NP}$.

A natural question is whether there are $\boldsymbol{NP}$-complete problems. The answer is yes. The example of $\boldsymbol{NP}$-complete problem and many future problems will be defined on Boolean formulae.

Definition 2.3.8. *Boolean formulae* on variables $x_1, x_2, \ldots, x_n$ are inductively defined as follows:

- literals $True$ and $False$ are boolean formulae,
- all propositional variables $x_1, x_2, \ldots, x_n$ are boolean formulae,
- if ϕ is a boolean formula, then so is $\neg\phi$,
- if ϕ and ψ are boolean formulae, then so are $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, and $\phi \leftrightarrow \psi$.

We will sometimes denote boolean formula ϕ on variables $x_1, \ldots, x_n$ as $\phi(x_1, \ldots, x_n)$.

SAT
Input: a boolean formula $\phi(x_1, \ldots, x_n)$.
Output: $True$ when ϕ is satisfiable and $False$ otherwise.

The connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$ are the standard connectives from propositional logic. We say that a boolean formula $\phi(x_1, \ldots, x_n)$ is satisfiable if there exists a valuation $v_1, \ldots, v_n$ such that $\phi(v_1, \ldots, v_n) = True$. We sometimes refer to a formula whose only connective is $\neg$ as a literal.

To simplify certain constructions, we allow a formula on $x_1, \ldots, x_n$ to not contain all (or even any) of the variables $x_1, \ldots, x_n$. A formula could be always extended with, for instance, $\wedge(x_1 \vee \cdots \vee x_n \vee \neg x_1)$ to ensure it contains all variables without altering truth value on any valuation. The input to **SAT** can be any Boolean formula. In particular, we do not require the formula to be in CNF form.

Notation. We use T and F as shorthand for *True* and *False* literals when working with boolean formulae.

Example 2.3.9. A boolean formula $\phi = (x_1 \wedge x_2) \wedge (x_1 \wedge \neg x_3)$ is satisfied by a substitution of (x_1, x_2, x_3) with (T, T, F):

$$\phi(T,T,F) = (T \wedge T) \wedge (T \wedge \neg F) = T \wedge (T \wedge T) = T \wedge T = T$$

Thus, ϕ is satisfiable and $\mathbf{SAT}(\phi(x_1, x_2, x_3)) = True$.

Theorem 2.3.10 (Cook-Levin Theorem [49])**.** **SAT** is $\boldsymbol{NP}$-complete.

The proof of the Cook-Levin theorem reduces an arbitrary problem **Q** in $\boldsymbol{NP}$ to **SAT** by encoding a run of a polynomial-time NDTM $\mathcal{M}_{\mathbf{Q}}$ corresponding to **Q** on a given input w as a boolean formula $\phi_{\mathcal{M}_{\mathbf{Q}},w}$ such that $\phi_{\mathcal{M}_{\mathbf{Q}},w}$ is satisfiable if an only if there exists an accepting path when running $\mathcal{M}_{\mathbf{Q}}$ on w, i.e. when $w \in \mathbf{Q}$. Importantly, this approach can be made *parsimonious* (see for instance [9]), that is $\phi_{\mathcal{M}_{\mathbf{Q}},w}$ has precisely the same number of satisfying assignment as there are accepting paths when running $\mathcal{M}_{\mathbf{Q}}$ on w. Notably, parsimony only applies to the accepting paths, i.e. the number of unsatisfying assignments may greatly surpass the number of rejecting paths. The number of accepting paths is particularly relevant for some complexity classes we consider next.

2.3.2 Counting-based classes

Next, we define some more obscure classes used later on. For that, we must explicitly consider the number of accepting and rejecting paths that a non-deterministic Turing machine takes:

Definition 2.3.11. Let $\mathcal{M}$ be a polynomial-time non-deterministic Turing machine and w be any input. When running $\mathcal{M}$ on w, we use the following notation:

- $\textsc{AllPaths}(\mathcal{M}, w)$ for the *total number of paths*,
- $\textsc{AccPaths}(\mathcal{M}, w)$ for the *number of accepting paths*,
- $\textsc{RejPaths}(\mathcal{M}, w)$ for the *number of rejecting paths*.

The following class was defined in [170].

Definition 2.3.12. The *counting polynomial-time* complexity class $\#\boldsymbol{P}$ (read as *sharp-P*, *counting-P*, or *number-P*) is complexity class of all function problems of the form 'given a non-deterministic polynomial-time Turing machine $\mathcal{M}$ and an input w, compute $\textsc{AccPaths}(\mathcal{M}, w)$'.

Just like **SAT** is canonical for class $\boldsymbol{NP}$, the counting version **#SAT** of **SAT** is canonical for $\#\boldsymbol{P}$:

#SAT
Input: a boolean formula $\phi(x_1, \ldots, x_n)$.
Output: the number of satisfying assignments of ϕ.

Example 2.3.13. Consider the boolean formula $\phi = (x_1 \wedge x_2) \wedge (x_1 \wedge \neg x_3)$ from Example 2.3.9. The satisfying valuation (T, T, F) of (x_1, x_2, x_3) is the only satisfying assignment of ϕ. Thus, ϕ has exactly one satisfying assignment and $\textbf{\#SAT}(\phi(x_1, x_2, x_3)) = 1$.
Similarly, $\textbf{\#SAT}(\phi(x_1, x_2, x_3, x_4)) = 2$, as it is satisfied only by (T, T, F, F) and (T, T, F, T) valuations of (x_1, x_2, x_3, x_4).

In the above example, adding variable x_4 changes the answer to **#SAT**, even though the variable does not appear in the given boolean formula. Thus, the chosen list of variables is vital for the **#SAT** answer.

Theorem 2.3.14 ([170])**.** **#SAT** is $\#\boldsymbol{P}$-complete.

We also consider one decision class about counting accepting paths:

Definition 2.3.15. The *exact-counting polynomial-time* class $\boldsymbol{C_=P}$ is the complexity class that consists of all problems $\mathbf{Q}$ for which there exists a non-deterministic polynomial-time Turing machine $\mathcal{M}$ such that for all inputs w the following holds:

$$w \in \mathbf{Q} \leftrightarrow \textsc{AccPaths}(\mathcal{M}, w) = \textsc{RejPaths}(\mathcal{M}, w)$$

We provide an example of a complete problem for this class in Chapter 4.

Other classes directly concerning the number of accepting paths of a non-deterministic Turing machine are the probabilistic classes.

2.3.3 Probabilistic classes

The deterministic approach is not the limit of what we can achieve with classical computers. Sometimes, we consider classes with a sprinkle of randomness, for instance, by constructing an algorithm with access to a random number generator. We will be interested in polynomial-time algorithms with access to randomness as captured by complexity class ***RP*** and its derivates, initially considered in [80].

Definition 2.3.16. The *randomised polynomial-time* class ***RP*** is the complexity class that consists of all problems $\mathbf{Q}$ for which there exists a non-deterministic polynomial-time Turing machine $\mathcal{M}$ such that for all inputs w the following holds:

$$w \in \mathbf{Q} \rightarrow \textsc{AccPaths}(\mathcal{M}, w) > \frac{1}{2}\textsc{AllPaths}(\mathcal{M}, w)$$

and

$$w \notin \mathbf{Q} \rightarrow \textsc{AccPaths}(\mathcal{M}, w) = 0.$$

The relation to a classical computer is as follows: a classical algorithm can be viewed as simulating a non-deterministic Turing machine, where at each point where multiple transitions are possible, a random transition is picked. Then, the conditions state that if the desired answer is *True* (i.e. when $w \in \mathbf{Q}$), then the algorithm will output *True* with probability at least $\frac{1}{2}$. When the desired answer is *False* (i.e. when $w \notin \mathbf{Q}$), then the algorithm will always output *False*. In other words, the algorithm is allowed to make errors, but the error probability must be bounded by $\frac{1}{2}$ and may happen only on 'yes' instances. The exact value of the bound is not important: any fixed bound in the open interval $(0, 1)$ leads to an equal class.

The error allowed only for 'no' instances with an otherwise analogous definition leads to class ***coRP***.

Finally, we can allow error in both directions [80]:

Definition 2.3.17. The *bounded-error probabilistic polynomial-time* class ***BPP*** is the complexity class of problems $\mathbf{Q}$ for which there exists a non-deterministic polynomial-time Turing machine $\mathcal{M}$ such that for all inputs w the following holds:

$$w \in \mathbf{Q} \rightarrow \textsc{AccPaths}(\mathcal{M}, w) > \frac{2}{3}\textsc{AllPaths}(\mathcal{M}, w)$$

and

$$w \notin \mathbf{Q} \rightarrow \textsc{RejPaths}(\mathcal{M}, w) > \frac{2}{3}\textsc{AllPaths}(\mathcal{M}, w)$$

A classical computer can simulate a run of $\mathcal{M}$ again choosing transitions at random. In this case, we only require that the correct answer happens with probability at least $\frac{2}{3}$. By running the algorithm polynomially many times the error probability can be made exponentially small. For this reason, the class ***BPP*** is often considered the limit of efficient (as in possible in polynomial-time) computation on classical computers.

Again, the value $\frac{2}{3}$ is not essential: any value in the interval $(\frac{1}{2}, 1)$ would work. However, the value $\frac{1}{2}$ leads to a different class [80]:

Definition 2.3.18. The *probabilistic polynomial-time* class ***PP*** is the complexity class of problems **Q** for which there exists a non-deterministic polynomial-time Turing machine $\mathcal{M}$ such that for all inputs w the following holds:

$$w \in \mathbf{Q} \rightarrow \textsc{AccPaths}(\mathcal{M}, w) > \frac{1}{2}\textsc{AllPaths}(\mathcal{M}, w)$$

and

$$w \notin \mathbf{Q} \rightarrow \textsc{RejPaths}(\mathcal{M}, w) > \frac{1}{2}\textsc{AllPaths}(\mathcal{M}, w)$$

Such problems can still be run on classical computers. Once again, it is more likely for the computer to produce a correct answer than an incorrect answer, just like in the case of class ***BPP***. However, the error probability could be arbitrarily close to $\frac{1}{2}$ and in particular, it could depend on $|w|$. Hence, polynomially many runs may be insufficient to ensure that the probability becomes exponentially small.

2.3.4 Oracle machines

Sometimes we want to use an algorithm for one problem to solve another problem. Such an approach can lead to a reduction. For instance, one can show that a method for checking whether a graph is 3-colourable can efficiently solve **SAT** by proving that $\mathbf{SAT} \leq_p \mathbf{3-Colouring}$ [76, Theorem 2.1]. Suppose that we have a magical black box (or an *oracle*) that can answer instances of **SAT**. What problems could we solve then? What if we had an oracle for a likely harder problem **#SAT**? We can formalise the idea of using oracles with oracle machines.

Definition 2.3.19. An *oracle Turing machine* $\mathcal{M}$ with access to problem **Q** oracle is a Turing machine with two special states q_{ASK} and q_{ANS} and two special tapes called T_{ASK} and T_{ANS}. Whenever $\mathcal{M}$ enters state q_{ASK}, the content t_{ASK} of the T_{ASK} tape is sent as a *query* to the oracle for **Q**. Next, the answer $\mathbf{Q}(t_{ASK})$ is written on the T_{ANS} tape. Finally, the $\mathcal{M}$ transitions to state q_{ANS}. The entire use of the oracle counts as a single computational step.

The details, such as how exactly the content t_{ASK} is defined, how the answer is written, and where the heads are positioned, are unimportant here. In the later part of the thesis, we will need to be explicit when partially encoding the run of an NDTM with an oracle as a boolean formula.

What problems can we solve with polynomial-time DTM with **SAT** oracle? For instance, we can solve the following problem [175]:

LexMaxSAT
Input: a boolean formula $\phi(x_1, \ldots, x_n)$.
Output: *True* if the lexicographically last satisfying assignment of ϕ ends with *True*.

Lemma 2.3.20. **LexMaxSAT** can be solved by a polynomial-time DTM with **SAT** oracle.

Proof. We ask the oracle whether $\phi \wedge x_1$ is satisfiable. If yes, necessarily $x_1 = T$ in last satisfying assignment: We set $\phi'(x_2, \ldots, x_n) = \phi(T, x2, \ldots, x_n)$ and solve the problem for ϕ' (which has one less variable). If no, $x_1 = F$ in last satisfiable assignment (if any exists): We set $\phi'(x_2, \ldots, x_n) = \phi(F, x_2, \ldots, x_n)$ and again solve the problem for ϕ'. Eventually, we learn the necessary value of the last variable in the lexicographically last satisfying assignment. More precisely, the formula eventually collapses to a single literal T or F. If it is F, the initial formula was not satisfiable, so we output F. Otherwise, we output the assignment found for the last variable. Thus, we get the answer in at most n queries with polynomial-time computations between oracle calls. Thus, the total runtime is polynomial. □

We can define the classes resulting from the inclusion of an oracle:

Definition 2.3.21. The class $P^{\mathbf{Q}}$ consists of problems solvable by a polynomial-time DTM with **Q** oracle.

Definition 2.3.22. The class $NP^{\mathbf{Q}}$ consists of problems solvable by a polynomial-time NDTM with **Q** oracle.

The oracle can also be changed from a problem into complexity class in the following sense:

Definition 2.3.23. The class $\boldsymbol{P}^{\boldsymbol{C}}$ consists of all problems solvable by a polynomial-time DTM with **Q** oracle for some $\mathbf{Q} \in \boldsymbol{C}$.

Definition 2.3.24. The class $\boldsymbol{NP}^{\boldsymbol{C}}$ consists of all problems solvable by a polynomial-time NDTM with **Q** oracle for some $\mathbf{Q} \in \boldsymbol{C}$.

Since **SAT** is $\boldsymbol{NP}$-complete, it turns out that $\boldsymbol{P}^{\boldsymbol{NP}} = \boldsymbol{P}^{\mathbf{SAT}}$ and $\boldsymbol{NP}^{\boldsymbol{NP}} = \boldsymbol{NP}^{\mathbf{SAT}}$. Thus, Lemma 2.3.20 proves that $\mathbf{LexMaxSAT} \in \boldsymbol{P}^{\boldsymbol{NP}}$. In fact, **LexMaxSAT** is $\boldsymbol{P}^{\boldsymbol{NP}}$-complete [175, 111].

It is not known whether $\boldsymbol{P}^{\boldsymbol{NP}} = \boldsymbol{NP}$. In fact, $\boldsymbol{P}^{\boldsymbol{NP}}$ is one of the 'blocks' used to define the polynomial hierarchy [158] called $\boldsymbol{PH}$.

Definition 2.3.25. The *polynomial hierarchy* $\boldsymbol{PH}$ is the complexity class defined as follows. Start with $\Delta_0\boldsymbol{P} = \Sigma_0\boldsymbol{P} = \Pi_0\boldsymbol{P} := \boldsymbol{P}$, and then define for $n \in \mathbb{N}_+$:

$$\begin{aligned}\Delta_n\boldsymbol{P} &:= \boldsymbol{P}^{\Sigma_{n-1}\boldsymbol{P}}\\ \Sigma_n\boldsymbol{P} &:= \boldsymbol{NP}^{\Sigma_{n-1}\boldsymbol{P}}\\ \Pi_n\boldsymbol{P} &:= \boldsymbol{coNP}^{\Sigma_{n-1}\boldsymbol{P}}\end{aligned}$$

Finally, $\boldsymbol{PH}$ is the union of all above for all $n \in \mathbb{N}$.

For example, $\boldsymbol{NP} = \boldsymbol{NP}^{\boldsymbol{P}} = \Sigma_1\boldsymbol{P}$ and $\boldsymbol{P}^{\boldsymbol{NP}} = \boldsymbol{P}^{\Sigma_1\boldsymbol{P}} = \Delta_2\boldsymbol{P}$. There exists a formal way of creating a canonical complete problem for each of these classes by considering quantified formula problems:

QBF
Input: a quantified expression consisting of a sequence of expressions $\forall x_i$ or $\exists x_i$ for $i = 1, \ldots, n$, followed by a boolean formula $\phi(x_1, \ldots, x_n)$.
Output: *True* when the quantified expression from the input holds and *False* otherwise.

A special case of **QBF** where all quantifiers are $\exists$ is equivalent to **SAT**. Similarly, restricting the number of alternations of quantifiers in the initial part of **QBF** input to k, leads to canonical complete problems for Σ_k and Π_k, where for Σ_k the first quantifier must be $\exists$ and for Π_k the first quantifier must be $\forall$ [180].

Thus, $\boldsymbol{PH}$ contains all problems arising from **QBF** by bounding the allowed number of alternations of quantifiers in the input. However, the problem **QBF** itself, i.e. the 'unbounded' version, is $\boldsymbol{PH}$-hard.

It is not known whether $\boldsymbol{PH}$ collapses to $\boldsymbol{P}$ but a common belief is that it does not collapse.

To conclude this section, we look at other known relations between complexity classes.

2.3.5 Relations between complexity classes

Computational complexity theory is not only about defining complexity classes but also about showing relations between them.

Deterministic Turing machines correspond to the special case of non-deterministic Turing machines where the image of the transition function contains singletons only; thus, classes defined on NDTMs contain analogous classes defined for DTMs. For example: $\boldsymbol{P} \subseteq \boldsymbol{NP}$ and $\boldsymbol{P}^{\boldsymbol{NP}} \subseteq \boldsymbol{NP}^{\boldsymbol{NP}}$.

Granting a TM access to the oracle makes it more powerful; if the TM can represent inputs for the oracle, then such 'equipment' allows the TM to solve the problems solvable by the oracle (and potentially other, more complex problems). For example: $\boldsymbol{NP} \subseteq \boldsymbol{P}^{\boldsymbol{NP}}$.

Based on the definition of $\boldsymbol{PH}$, we also know that $\boldsymbol{NP}^{\boldsymbol{NP}} \subseteq \boldsymbol{PH}$.

The $\boldsymbol{PH}$-hard problem **QBF** is known to be in $\boldsymbol{PSPACE}$ [124], which together with the above statements leads to:

$$\boldsymbol{P} \subseteq \boldsymbol{NP} \subseteq \boldsymbol{P}^{\boldsymbol{NP}} \subseteq \boldsymbol{PH} \subseteq \boldsymbol{PSPACE}$$

Just as adding an oracle or changing DTM to NDTM allows more problems to be solved, allowing errors in the answers leads to containments between probabilistic classes. For instance, it is known that (all follow from [80]):

$$\boldsymbol{P} \subseteq \boldsymbol{RP} \subseteq \boldsymbol{BPP} \subseteq \boldsymbol{PP} \subseteq \boldsymbol{PSPACE}$$

Relations of the first group of problems, like classes $\boldsymbol{NP}$ and $\boldsymbol{PH}$, and probabilistic classes are less obvious. We know that $\boldsymbol{BPP} \subseteq \boldsymbol{PH}$ [116]. Toda's theorem [165] states that:

$$\boldsymbol{PH} \subseteq \boldsymbol{P}^{\boldsymbol{PP}}$$

Using similar approaches, one can also show that:

$$\boldsymbol{P}^{\boldsymbol{PP}} = \boldsymbol{P}^{\#\boldsymbol{P}}$$

which motivates the statement that $\#\boldsymbol{P}$ is the function class equivalent of the decision class $\boldsymbol{PP}$.

At the same time, computational complexity is notorious for a lack of separation results between most standard classes, i.e. proving that two classes are different. The best-known open problem is the question of whether $\boldsymbol{P} = \boldsymbol{NP}$. While it is widely expected that the classes are different, we don't know for sure, and the question is on the Millennium Problems List. It is known that a resolution would need to utilise sophisticated techniques beyond so-called natural proofs [147]. In fact, we do not even know whether $\boldsymbol{P} = \boldsymbol{PSPACE}$, and hence we should not expect to find a surprising separation between the above-defined classes if one is already known to be a subset of the other.

Later on, we will study class $\boldsymbol{NP}^{\#\boldsymbol{P}}$ and its relation to graphical languages for quantum computation. It could be the case that $\boldsymbol{P} = \boldsymbol{PH}$, but $\boldsymbol{PH} \neq \boldsymbol{P}^{\#\boldsymbol{P}}$: therefore, having **#SAT** oracle is likely greatly more powerful that having **SAT** oracle.

This marks the end of the computational background from a classical perspective – now we move to the quantum side.

2.4 Quantum circuit model

We assume familiarity with basic quantum computation and circuit model as it is presented in the textbook by Nielsen & Chaung [137]. This section includes only a minimal overview of the topic and is based on the textbook mentioned above and another textbook by Coecke & Kissinger [46].

Quantum computation harnesses the power of quantum mechanics at the fundamental level of physics. However, just as we do not study the implementation of electric circuits in classical computers, we also avoid purely physical interpretations of quantum computation. Instead, we work on a formalised mathematical model encapsulating what can be achieved in the quantum world.

The mathematical model arises from von Neumann's postulates:

1. Quantum systems are represented by Hilbert spaces. The state of a system corresponds to an equivalence class of normalised vectors equal up to a global phase. The composite systems are represented by the tensor product of the Hilbert spaces representing the subsystems.

2. Deterministic, reversible quantum processes are represented by unitaries on the Hilbert space.

3. Quantum measurements are represented by self-adjoint linear maps on the Hilbert space. When a measurement takes place, the state collapses according to the action of one of the projectors, with the probability of each projector given by the Born rule.

We exclusively work with systems constructed on qubits, meaning that the dimensions of the underlying Hilbert spaces are powers of two (though we might mention some results on qudits).

2.4.1 Components of the circuit model

In the circuit model, the quantum part of the computation consists of several steps:

1. Preparation of the qubits in simple single qubit states such as $|0\rangle$ or $|+\rangle$,
2. Application of the unitary gates to the qubits,
3. Measurements of the qubit in the computational basis.

The computation may also feature classical pre- and post-processing, and the steps can interweave; for instance, further parts of the computation may depend on earlier measurement outcomes. The measurements do not necessarily need to be performed in the computational basis. In principle, measurements along any computable axis are valid.

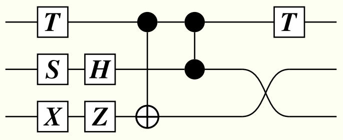
The proper part of the computation is the second step, i.e. application of quantum gates with standard gates presented in Figure 2.1. In particular, we define rotational gates so that the angles match the presentation in graphical calculi established in the next chapter. For example, the T gate is (up to a global phase that we ignore) a Z rotation gate with parameter $\frac{\pi}{4}$ rather than $\frac{\pi}{8}$, which is the case in some literature.

Notation. We write subscripts to indicate to which qubit the gate is applied. For example, on a system with qubits $1, 2, 3$, writing T_2 stands for the process $Id \otimes T \otimes Id$, i.e. the T gate is applied to the second qubit and an identity to the other qubits. For controlled gates, we first list the control and then the target qubits.

Example 2.4.1. An example of a quantum circuit:

$$(T_1 SWAP_{2,3})(CZ_{1,2})(CNOT_{1,3})(H_2 Z_3)(T_1 S_2 X_3)$$

and its graphical representation:



The Hadamard gate, both rotational single-qubit gates (for any choice of parameter in $[0, 2\pi)$), and any two-qubit entangling gate like CNOT are sufficient for universal

$$H = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

(a) Hadamard gate

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

(b) Pauli X gate

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

(c) Pauli Y gate

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

(d) Pauli Z gate

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

(e) S gate

$$T = \begin{pmatrix} 1 & 0 \\ 0 & \exp(\frac{i\pi}{4}) \end{pmatrix}$$

(f) T gate

$$SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(g) SWAP gate

$$R_X(\alpha) = \begin{pmatrix} \cos(\frac{\alpha}{2}) & -i\sin(\frac{\alpha}{2}) \\ -i\sin(\frac{\alpha}{2}) & \cos(\frac{\alpha}{2}) \end{pmatrix}$$

(h) X rotation gate

$$R_Z(\alpha) = \begin{pmatrix} \exp(-\frac{i\alpha}{2}) & 0 \\ 0 & \exp(\frac{i\alpha}{2}) \end{pmatrix}$$

(i) Z rotation gate

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

(j) Cotrolled-NOT gate

$$CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

(k) Cotrolled-Z gate

$$Toff = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

(l) Toffoli gate

Figure 2.1: Quantum gates

quantum computation, meaning that any quantum process can be constructed with these gates. Usually, we restrict ourselves to finite generator sets. For instance, we refer to a system given by Hadamard, S, and CNOT as the Clifford fragment of the quantum computation. The Clifford+T fragment extends the Clifford fragment by allowing the T gate to be used as a generator, leading to an approximately universal gate set. The Hadamard and Toffoli gates generate another important approximately universal gate set [6].

2.4.2 Power of quantum computers

Quantum computers are physical machines that can implement quantum computations given by quantum circuits. Quantum computers are more efficient than classical computers for particular problems, though it may depend on the chosen generator set of gates used to construct the circuits. Quantum circuits constructed with infinite generator sets are impossible to achieve in practice, and their computational power goes beyond the regime of computational complexity (for instance, due to Euler decomposition, any single qubit quantum process can be performed with arbitrary rotational gates in a circuit of constant depth). On the other hand, classical computers can efficiently simulate Clifford circuits; thus, Clifford computation offers no quantum advantage. However, approximately universal quantum circuits may lead to a quantum advantage.

Formally, the class of problems solvable by quantum computers in polynomial-time is called ***BQP***[22].

Definition 2.4.2. The *bounded-error quantum polynomial-time* class ***BQP*** is the complexity class of problems $\mathbf{Q}$ for which there exists a quantum Turing machine $\mathcal{M}$ such that for all inputs w the following holds:

$$w \in \mathbf{Q} \to \mathrm{Prob}(\mathcal{M}(w) = True) > \frac{2}{3}$$

and

$$w \notin \mathbf{Q} \to \mathrm{Prob}(\mathcal{M}(w) = True) < \frac{1}{3}$$

This initial formulation of the power of quantum computers uses quantum Turing machines (whose definition we skip here). Nowadays, the standard way to formalise ***BQP*** is via uniform circuits [39, 4], based on which we can conclude that polynomial-size circuits constructed with Clifford+T and H+Toffoli generator sets can precisely solve the problems in ***BQP***.

It is known that $\boldsymbol{BPP} \subseteq \boldsymbol{BQP} \subseteq \boldsymbol{PP}$ [22, 4]. As is often the case in computational complexity, we do not know whether $\boldsymbol{BQP} \supsetneq \boldsymbol{P}$ (or whether $\boldsymbol{BQP} \supsetneq \boldsymbol{BPP}$), and thus, it could be the case that $\boldsymbol{BQP} = \boldsymbol{P}$. However, some candidate problems are known to be in $\boldsymbol{BQP}$ and believed to not be in $\boldsymbol{P}$. For instance, Shor's algorithm [152] can solve **FindDivisor** and thus this problem is in $\boldsymbol{BQP}$[1]. At the same time, such problems are notorious for being difficult to solve with a classical computer and are the focal point of many cryptography protocols such as RSA [148].

The non-deterministic nature of measurements in the quantum world is essential when formalising problems in $\boldsymbol{BQP}$. If we were able to select which measurement outcome should happen (provided that the selection we make could happen with non-zero probability), then the polynomial-sized quantum circuits could solve problems in $\boldsymbol{PostBQP}$, i.e. $\boldsymbol{BQP}$ with post-selection on measurement outcomes [1]. It is known that $\boldsymbol{PostBQP} = \boldsymbol{PP}$ [1]. On the other hand, there are no known relations between problems solvable in polynomial time by quantum Turing machines and non-deterministic Turing machines: It is believed that $\boldsymbol{NP}$ and $\boldsymbol{BQP}$ are different, and neither is a subset of the other.

Quantum computers are not only expected to efficiently solve problems outside of $\boldsymbol{P}$; there are also problems in $\boldsymbol{P}$ which could be sped up with quantum computers, such as the search problem via Grover's algorithm [83].

2.5 Measurement-based Quantum Computation

While the standard model for quantum computation is the circuit model, other models have also been formalised. One of such models is the one-way quantum computation, or measurement-based quantum computation (MBQC). It is a quantum computation model alternative and equally powerful to the circuit model [143, 144, 145].

The computation consists of two steps: preparation of highly entangled resource state, and adaptive single qubit measurements on this resource state. Sometimes the two steps are intertwined: it is possible to re-use the same physical systems as different qubits over the course of a single computation, but for simplicity we will consider state preparation and measurement phase separately here.

In this section, we examine formal definitions in the MBQC model and explore the fundamental problem of determinism.

2.5.1 Basic definitions

Most resource states used in MBQC are graph states, arising from simple graphs. We use the following notations:

[1]More precisely, the decision variant is in $\boldsymbol{BQP}$; that is the problem that asks whether input number n has a divisor in some interval (a, b) which is also given in the input.

Notation. Unless specified otherwise, all graphs are assumed to be simple graphs (undirected graphs in which all edges are distinct and connect two distinct vertices). Given graph G, we always refer to its vertices with V and edges with E. Vertices are always written with small letters, avoiding letter c to eliminate confusion with correction functions. An edge is written by listing two vertices next to each other, e.g. $uv \in E$ means that there is an edge connecting u and v. N_u denotes neighbourhood of u, i.e. set of vertices connected to u.

We also define less standard notions of odd neighbourhood and closed odd neighbourhood:

Definition 2.5.1. Let $G = (V, E)$ be a simple graph and $\mathcal{A} \subseteq V$ be any subset of vertices. The *odd neighbourhood* of $\mathcal{A}$ is the set of vertices $v \in V$ which are adjacent to an odd number of elements of $\mathcal{A}$, i.e.:

$$\mathrm{Odd}(\mathcal{A}) := \{v \in V \colon |\{u \in \mathcal{A} \mid vu \in E\}| \text{ is odd}\}\,.$$

The *closed odd neighbourhood* of $\mathcal{A}$ is defined as follows, where Δ stands for the symmetric difference of sets, i.e. $\mathcal{D} \,\Delta\, \mathcal{D}' = (\mathcal{D} \cup \mathcal{D}') \setminus (\mathcal{D} \cap \mathcal{D}')$:

$$\mathrm{Odd}[\![\mathcal{A}]\!] := \mathrm{Odd}(\mathcal{A}) \,\Delta\, \mathcal{A}.$$

We are now ready to define graph states:

Definition 2.5.2. Given a simple graph $G = (V, E)$, we define a corresponding *graph state* $|\Phi_G\rangle$ as follows:

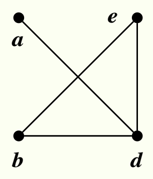
1. For each vertex $v \in V$, prepare a corresponding qubit in state $|+\rangle$,
2. For each edge $uv \in E$, apply a CZ gate between qubits corresponding to vertices u and v.

Since CZ gates commute, the order in which CZ gates are applied does not matter. This means a graph state is well-defined and unique for a given graph:

Remark 2.5.3. Given graph $G = (V, E)$, the corresponding graph state $|\Phi_G\rangle$ can be written as:

$$\prod_{uv \in E} CZ_{u,v} \bigotimes_{v \in V} |+\rangle_v$$

Example 2.5.4. Consider the following graph:



The corresponding graph state is:

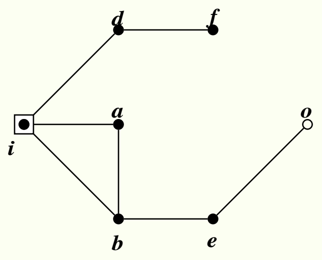
$$CZ_{ed}CZ_{bd}CZ_{be}CZ_{ad}\,|{+}{+}{+}{+}\rangle_{abcd}$$

We often work with a related structure of open graphs, which explicitly specify the input and output qubits:

Definition 2.5.5. An *open graph* is a triple (G, I, O), where $G = (V, E)$ is a simple graph, and $I, O \subseteq V$ are the *input* and *output* sets respectively.
We denote the set of *non-inputs* with $\bar{I} := V \setminus I$ and the set of *non-outputs* with $\bar{O} := V \setminus O$.

Notation. When drawing open graphs, inputs are denoted with a box around the vertex, and outputs are denoted with empty circles rather than filled ones.

Example 2.5.6. An open graph:



with one input: $I = \{i\}$ and one output $O = \{o\}$.

The correspondence to a quantum state is as follows: instead of preparing qubits in state $|+\rangle$ for all vertices, we only prepare them for $v \in V \setminus I$. For $v \in I$, we instead assume they are given as input of the computation. Finally, instead of measuring all qubits, we only measure qubits corresponding to $v \in V \setminus O$. Qubits corresponding to $v \in O$ are treated as outputs of the computation: Such output qubits could be used for further computation or measured afterwards to finish the computation.

In the MBQC model, progressive, adaptive measurements drive the computation forward. We allow three types of planar measurements corresponding to the three planes of the Bloch sphere:

Definition 2.5.7. The operators corresponding to *XY, YZ, and XZ planar measurements* are defined as follows for any $\alpha \in [0, 2\pi]$:

$$\langle +_{XY,\alpha}| = \frac{1}{\sqrt{2}} (\langle 0| + e^{-i\alpha} \langle 1|) \qquad \langle -_{XY,\alpha}| = \frac{1}{\sqrt{2}} (\langle 0| - e^{-i\alpha} \langle 1|)$$
$$\langle +_{YZ,\alpha}| = \frac{1}{\sqrt{2}} (\langle +| + e^{-i\alpha} \langle -|) \qquad \langle -_{YZ,\alpha}| = \frac{1}{\sqrt{2}} (\langle +| - e^{-i\alpha} \langle -|)$$
$$\langle +_{XZ,\alpha}| = \frac{1}{\sqrt{2}} (\langle +i| + e^{-i\alpha} \langle -i|) \qquad \langle -_{XZ,\alpha}| = \frac{1}{\sqrt{2}} (\langle +i| - e^{-i\alpha} \langle -i|)$$

where:

$$\langle +i| := \frac{1}{\sqrt{2}} (\langle 0| - i \langle 1|) \qquad \langle -i| := \frac{1}{\sqrt{2}} (\langle 0| + i \langle 1|)$$

We have defined operators by stating bras, i.e. conjugate transposes of more standard kets; hence, all $e^{i\alpha}$ terms had to change to $e^{-i\alpha}$.

The measurement in plane λ at an angle α can result in two outcomes: desired outcome $\langle +_{\lambda,\alpha}|$ and undesired outcome $\langle -_{\lambda,\alpha}|$. Note that these two outcomes indeed extend to measurements, as the two bras form an orthonormal basis, and hence the corresponding projectors satisfy:

$$|+_{\lambda,\alpha}\rangle \langle +_{\lambda,\alpha}| + |-_{\lambda,\alpha}\rangle \langle -_{\lambda,\alpha}| = Id.$$

When α is a multiple of $\frac{\pi}{2}$, the planar measurements collapse to Pauli X, Y, and Z measurements.

MBQC on graph states can be shown to be universal for quantum computing with just one type of planar measurements; see [145, 119] for XY, [127] for XZ, and [156] for YZ (with additional local rotation gates required in the last case). However, having access to all types of computation gives more flexibility, possibly leading to smaller requirements on the number of measured qubits. The necessity of having some planar measurements cannot be relaxed if we wish to maintain universality, as graph states with only Pauli measurements fall into the Clifford fragment. While we restrict ourselves to graph states, we note that Pauli measurements suffice for universality if the resource state is transformed from a graph state to a more complex structure. For instance, Pauli X and Z measurements are sufficient for universality in hypergraph MBQC [135, 163, 181], that is MBQC in which resource states are hypergraph states [149]. Similarly, these measurements also suffice for states where entangling is changed from the CZ gate to (up to local unitaries) the $\sqrt{\text{CZ}}$ gate [105].

Given an open graph, we can describe the desired computation in the MBQC model by specifying the desired outcome for each non-output, i.e., choosing the measurement plane and measurement angle.

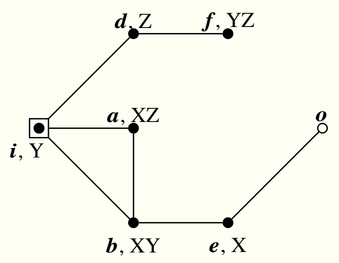
Definition 2.5.8. Given an open graph (G, I, O) where $G = (V, E)$, we define *measurement labelling* as any function $\lambda\colon V \setminus O \to \{X, XY, Y, YZ, Z, ZX\}$ such that $\lambda(v) \in \{X, XY, Y\}$ for all $v \in I \setminus O$.
We refer to the set of planar measured vertices as Λ_{planar}, and to the set of Pauli measured vertices as Λ_{Pauli}:

$$\Lambda_{planar} = \left\{v \in \bar{O} \mid \lambda(v) \in \{XY, YZ, XZ\}\right\}$$
$$\Lambda_{Pauli} = \left\{v \in \bar{O} \mid \lambda(v) \in \{X, Y, Z\}\right\}.$$

Definition 2.5.9. A *labelled open graph* is a quadruple (G, I, O, λ) such that (G, I, O) is an open graph, and λ is a corresponding measurement labelling.

Notation. When drawing labelled open graphs, the measurement labels for each non-output are specified near vertex names.

Example 2.5.10. A labelled open graph:



with the underlying open graph from Example 2.5.6.

To fully specify the intended outcome of each measurement, one would also need to choose the measurement angle:

Definition 2.5.11. Given a labelled open graph (G, I, O, λ), we define the *measurement choice* as any function $\alpha\colon V \setminus O \to [0, 2\pi]$ such that $\forall v \in V \setminus O . \lambda(v) \in$

$\{X, Y, Z\} \to \alpha(v) \in \{0, \pi\}$.

The last condition on the allowed angles ensures that the vertices measured with Pauli measurements have a compatible angle. We can now add measurement choices to labelled open graphs to obtain what we call MBQC schemes (to my best knowledge, this structure lacks a standardised name):

Definition 2.5.12. An *MBQC scheme* is a quintuple $(G, I, O, \lambda, \alpha)$ such that (G, I, O, λ) is a labelled open graph and α is a compatible measurement choice.

While an MBQC scheme fully describes the quantum process desired for the computation, we will primarily work with labelled open graphs later on. The exact angle will be meaningless for the structures for deterministic computation that we define later.

2.5.2 Corrections

Since quantum measurements are inherently non-deterministic, undesired outcomes must be accounted for by defining a suitable correction procedure for each.

For each measurement, only one outcome is desired. The undesired outcomes differ from the desired outcomes by a Pauli gate, for instance:

$$\left\langle -_{XY,\alpha}\right| = \left\langle +_{XY,\alpha}\right| Z$$

for any angle α. Thus, obtaining an undesired outcome for planar XY measurement results in a Pauli Z byproduct. Similarly, all other undesired outcomes can be viewed as Pauli byproducts. Correcting such byproducts by complementing them to stabilisers of the underlying graph state is standard. Stabilisers of graph states are well understood [86], making it possible to detect stabilisers that can be used for correction purposes. Such an approach is also sufficient for universal quantum computation. Finally, the MBQC model is supposed to function as an alternative to the circuit model; hence, we do not want the corrections to involve complicated quantum processes.

The theorem below follows from [86, Subsection II.B].

Theorem 2.5.13. The stabilizers of a graph state $|\Phi_G\rangle$ form a group with the following generator set:

$$\left\{Z_{N(u)}X_v \mid v \in V\right\}$$

where $N(u)$ stands for neighbourhood of v.

From these generators, we can construct all stabilisers. This requires choosing a set of applied generators, i.e. each stabiliser is uniquely induced by set $\mathcal{A}$ of vertices to which X gate is applied. The vertices to which Z is applied form precisely $\mathrm{Odd}(\mathcal{A})$. See the example:

Example 2.5.14. Consider the graph state $|\phi_G\rangle$ from Example 2.5.4. The generator stabilisers of $|\phi_G\rangle$ are:

$$X_a Z_d \quad X_b Z_e Z_d \quad Z_b X_e Z_d \quad Z_a Z_b Z_e X_d.$$

The stabiliser induced by the set of vertices $\{b, d\}$ is:

$$(X_b Z_e Z_d)(Z_a Z_b Z_e X_d) = Z_a (X_b Z_b)(Z_e Z_e)(Z_d X_d) = Z_a (iY_b) I_e (-iY_d) = Z_a Y_b Y_d.$$

2.5.3 Determinism

An MBQC scheme that can be performed independently of the observed measurement outcomes is called deterministic. In other words, deterministic schemes are those for which it is possible to correct all undesired outcomes. The formal definition of determinism requires first defining measurement patterns arising from measurement calculus.

Measurement calculus

The measurement calculus [53, 54] formalises transformations of computations in MBQC by defining measurement patterns: the structure describing the procedure necessary to perform the intended computation. This structure consists of an MBQC scheme, along with the order in which measurements should be applied and the correction procedure for undesired outcomes. The next two definitions are adapted from [153]:

Definition 2.5.15. A *measurement pattern* consists of n qubit register V, with distinguished sets $I, O \subseteq V$ of input and output qubits and a sequence of commands consisting of the following operators:

- *Preparations* N_v of qubit $v \in V \setminus I$ in state $|+\rangle$,
- *Entangling operators* E_{vw} applying CZ gate between qubits v and w,
- *Destructive measurements* $M_v^{\lambda,\alpha}$, projecting qubit $v \in V \setminus O$ onto either $\langle +_{\lambda,\alpha}|$ with outcome 0 or $\langle -_{\lambda,\alpha}|$ with outcome 1,
- Corrections $[X_u]^v$ or $[Z_u]^v$ conditionally applying an X or Z gate respectively to qubit $u \in V$ if the outcome of the measurement for qubit v is 1.

We also define runnable patterns to exclude patterns with measurements of non-existing qubits, applications of gates to previously measured qubits and similar contradictory operations:

Definition 2.5.16. A measurement pattern $\mathfrak{P}$ on register V with inputs and outputs I, O is *runnable* if:

- All non-input qubits are prepared exactly once, i.e. $\mathfrak{P}$ contains N_i for all $i \in V \setminus I$.
- A non-input qubit is not acted on by any other command before its preparation, i.e. the first command on any $v \in V \setminus I$ is N_v.
- All non-output qubits are measured exactly once, i.e. for all $v \in V \setminus O$, there is a command $M_v^{\lambda,\alpha}$ for some λ and α.
- A non-output qubit is not affected by any other command after its measurement, i.e. the last command affecting $v \in V \setminus O$ is $M_v^{\lambda,\alpha}$ for some λ and α.
- No correction depends on an outcome not yet measured, i.e. commands $[X_u]^v$ and $[Z_u]^v$ can appear in $\mathfrak{P}$ only after the command $M_v^{\lambda,\alpha}$.

Notions of determinism

When a computation given by an MBQC scheme can be performed in a way in which all undesired outcomes are corrected, we call it *deterministic*. However, there are multiple slightly different notions of determinism that people consider. To explore these notions, we first must define a branch of computation in the MBQC model. The following definition is adapted from [15] and [128]:

Definition 2.5.17. Let $\mathfrak{P}$ be a measurement pattern with n measurement commands. A *branch* of the pattern is a length-n binary sequence p together with a linear map $[\![\mathfrak{P}]\!]_p$, where each measurement is replaced with an outcome corresponding to the entries in p.

Definition 2.5.18. Let $\mathfrak{P}$ be a measurement pattern with n measurement commands and let $k \leq n$. An *intermediate pattern* $\langle \mathfrak{P} \rangle_k$ is a measurement pattern obtained as a truncation of $\mathfrak{P}$ that excludes all measurements and all corrections dependent on such measurements except for the first k measurements.

We are now able to formally define determinism notions [52, Section II]:

Definition 2.5.19. A runnable measurement pattern $\mathfrak{P}$ with n measurement commands is:

- *deterministic* if for any branches p and q we have $[\![\mathfrak{P}]\!]_p = c[\![\mathfrak{P}]\!]_q$ for some $c \in \mathbb{C}$,
- *strongly deterministic* if for any branches p and q we have $[\![\mathfrak{P}]\!]_p = c[\![\mathfrak{P}]\!]_q$ for some $c \in S^1$ (element of the unit circle in the complex plane),
- *stepwise deterministic* if for any $k \leq n$, the intermediate pattern $\langle\mathfrak{P}\rangle_k$ is deterministic,
- *uniformly deterministic* if it is deterministic for any choice of angles α appearing in the measurement commands in $\mathfrak{P}$,
- *robustly deterministic* if it is strongly, stepwise, and uniformly deterministic.

Some other notions of determinism, which we do not work with in this thesis, include uniform equiprobability and constant probability [125].

The notion of robust determinism is of most significant use for a few reasons. Firstly, strongly deterministic measurement patterns implement unitary embeddings [52, Lemma 1] and, thus, valid quantum processes. Secondly, stepwise deterministic measurement patterns can be performed single qubit at a time and the pattern can be implemented in parts by partitioning computation into smaller, still deterministic fragments. Thirdly, uniformly deterministic measurement patterns allow performance of different computations with the same underlying labelled open graph. We will restrict ourselves to only looking for robustly deterministic patterns. We will say that computation in MBQC is deterministic to signal that the implied measurement pattern is robustly deterministic.

2.5.4 Applications

We briefly outline most important applications of MBQC. The split of computation into two parts (resource state preparation and adaptive single qubit measurements) is particularly well suited to a client-server split: e.g. blind quantum computing scheme allows computations to be securely delegated to quantum server while requiring no or minimal quantum capabilities from the client [27, 73, 102, 100]. MBQC also links with quantum secret-sharing protocols [101] and forms the basis of approaches for verifying quantum computations [79, 100]. Scalable implementations of photonic quantum computation are expected to be based on the one-way model [162, 182, 69]. Finally, fault-tolerant quantum computation using surface codes lattice surgery is closely related to the one-way model [146, 59]. The one-way model is also of theoretical significance: the more flexible structure of graph states (as compared to quantum circuits) means it is often easier to optimise computations expressed in the one-way model, whether that is to trade ancillas against circuit depth [28], reducing the number of T gates [64], or other metrics [157].

2.6 Flow structures

Given the MBQC scheme, can the computation be performed deterministically? This question can be answered by examining the so-called flow structures that capture sufficient (and sometimes necessary) conditions for the robust determinism of the implied measurement patterns.

Since robust determinism is a special case of uniform determinism, the exact measurement angles need not be considered in the MBQC scheme. Therefore, we can consider labelled open graphs as the notion of the intended quantum process. For this reason, the flow structures are defined on labelled open graphs, as opposed to MBQC schemes.

The general idea behind flow structures is to provide an algebraic set of conditions on a labelled open graph that, when satisfied, implies the existence of a robustly deterministic measurement pattern.

In the later chapters, we discuss Pauli flow. However, as our results apply to Pauli flow, they also apply to more restricted flow structures such as causal flow and gflow. For this reason, we provide a detailed overview of all of these flow types.

2.6.1 Causal flow

The oldest and simplest of the flow structures is the causal flow that works only for planar XY measurements. It was defined in [52, Definition 2] (were it was called simply *flow*):

Definition 2.6.1. Consider a labelled open graph (G, I, O, λ) with $\lambda(v) = XY$ for all $v \in \bar{O}$. A *causal flow* is a pair $(f, \prec)$ where $f : \bar{O} \to \bar{I}$ is a *correction function* and $\prec$ is a strict partial order on V such that for all $v \in \bar{O}$:

- $v f(v) \in E$,
- $v \prec f(v)$, and
- $\forall w \in N(f(v)). w = v \vee v \prec w$.

The vertex $f(v)$ is called the *corrector* for v.

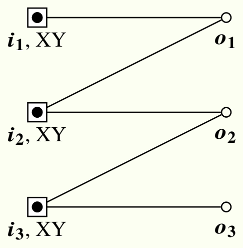
If an undesired outcome is observed when measuring qubit $v \in \bar{O}$, then the Pauli byproduct on v is complemented to a stabiliser by applying X gate to $f(v)$ and Z gate to neighbours of $f(v)$ other than v. The flow conditions ensure that such application of gates is causal, meaning that the gates can indeed be applied.

A causal flow is a sufficient condition for robust determinism, as implied by the following theorem [52, Theorem 1]:

Theorem 2.6.2. Let (G, I, O, λ) be a labelled open graph with $\lambda(v) = XY$ for all $v \in \bar{O}$ and $(f, \prec)$ be a causal flow on it. Then, there exists a robustly deterministic measurement pattern $\mathfrak{P}$ implementing $(G, I, O, \lambda, \alpha)$ for any α.

We illustrate causal flow with an example:

Example 2.6.3. Consider the following labelled open graph from [29, Figure 3]:



It does have causal flow:

$$f(i_k) = o_k \text{ for } k \in \{1, 2, 3\}$$
$$i_1 \prec i_2 \prec i_3 \prec o_1, o_2, o_3$$

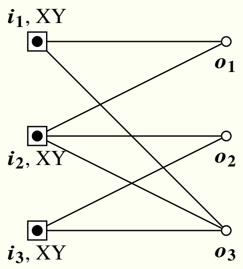
Therefore, there exists a robustly deterministic pattern implementing the given computation:

$$[X_{o_3}]^{i_3} M_{i_3}^{XY,\alpha(i_3)} [Z_{i_3}]^{i_2} [X_{o_2}]^{i_2} M_{i_2}^{XY,\alpha(i_2)} [Z_{i_2}]^{i_1} [X_{o_1}]^{i_1} M_{i_1}^{XY,\alpha(i_1)}$$
$$E_{i_3 o_3} E_{i_3 o_2} E_{i_2 o_2} E_{i_2 o_1} E_{i_1 o_1} N_{i_3} N_{i_2} N_{i_1}$$

The above example captures a rather small graph, yet the measurement pattern is already quite lengthy; in this case, it contains sixteen commands. We refrain from writing full measurement patterns for future examples: the important part is that they can always be achieved.

We know that causal flow is sufficient for robust determinism. However, is it necessary? Consider the following example without causal flow:

Example 2.6.4. Consider the following labelled open graph (G, I, O, λ) from [29, Figure 2]:



Suppose it had causal flow $(c, \prec)$. Let i_k be the last input in $\prec$. Since $i_k f(i_k) \in E$, necessarily $f(i_k) \in O$. However, all outputs have at least two neighbours. Thus, there is $i_\ell \neq i_k$ such that $i_\ell \in N(f(i_k))$. However, then we would need $i_k \prec i_\ell$, which contradicts the assumption that i_k is last in $\prec$.

The proof above can be adapted for any labelled open graph where $\delta(G) \geq 2$: instead of considering the last input in the order, consider the last non-output in the order. On the other hand, the labelled open graph above can be implemented in a robustly deterministic way, and thus, causal flow is not necessary for robust determinism. The necessary (and sufficient) condition that covers XY planar measurements (as well as YZ and XZ planar measurements) is the generalised flow, which we explore in the next subsection.

2.6.2 Generalised flow

The correction mechanism defined in causal flow is limited. When an undesired outcome is observed, the corresponding Pauli byproduct is completed to some generator stabiliser. Therefore, causal flow does not utilise the entire stabiliser group of the resource state in the correction procedure. Further, causal flow only works where all measurements are planar XY measurements. These restrictions are relieved in generalised flow, initially defined in [29, Definition 3]:

Definition 2.6.5. Consider a labelled open graph (G, I, O, λ) with $\lambda(v) \in \{XY, YZ, XZ\}$ for all $v \in \bar{O}$, i.e. $\Lambda_{Pauli} = \emptyset$ and $\Lambda_{planar} = \bar{O}$. A *generalised flow* (shortened to *gflow*) is a pair $(c, \prec)$ where $c\colon \bar{O} \to \mathcal{P}(\bar{I})$ is a *correction function* and $\prec$ is a strict partial order on $\bar{O}$ such that for all $v \in \bar{O}$:

- $\forall w \in c(v). v \neq w \to v \prec w$,
- $\forall w \in \mathrm{Odd}(c(v)). v \neq w \to v \prec w$,
- $\lambda(v) = XY \to v \notin c(v) \wedge v \in \mathrm{Odd}(c(v))$,

- $\lambda(v) = YZ \rightarrow v \in c(v) \wedge v \notin \mathrm{Odd}(c(v))$,
- $\lambda(v) = XZ \rightarrow v \in c(v) \wedge v \in \mathrm{Odd}(c(v))$.

The set $c(v)$ is called the *correction set* for v, and its elements are called *correctors* for v.

The gflow is called *generalised* as it is a strict generalisation of causal flow:

Observation 2.6.6. Let $(f, \prec)$ be a causal flow on some labelled open graph. Then $(c, \prec)$ is a gflow on the same labelled open graph where $c(v) := \{f(v)\}$ for all $v \in \bar{O}$.

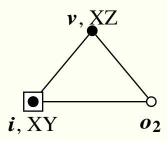
In gflow, the correction function specifies not just the stabiliser generator but a set of vertices inducing a stabiliser to which potential byproduct is complemented. This way, gflow allows more sophisticated correction mechanisms and can be used for all three measurement planes.

Example 2.6.7. Consider the labelled open graph from Example 2.6.4. It does have gflow:

$$
\begin{aligned}
c(i_1) &= \{o_2, o_3\} \\
c(i_2) &= \{o_1, o_2, o_3\} \\
c(i_3) &= \{o_3\} \\
i_3 \prec i_1, i_2 &\prec o_1, o_2, o_3
\end{aligned}
$$

Since the XY plane suffices for universality, the literature most commonly considers labelled open graphs with only planar XY measurements. This leads to a naming conflict, where sometimes gflow refers to cases with XY planar measurement only, and instead, the variant compatible with all three planes is referred to as *extended gflow*. However, the word *extended* also has other meanings in MBQC theory. We will avoid the term extended gflow and assume that gflow covers all three planes. We will use term *XY-only gflow* to refer to gflow variant when all vertices are measured in XY plane.

Example 2.6.8. Consider the following labelled open graph:



It does have gflow:

$$c(i) = \{o\}$$
$$c(v) = \{v, o\}$$
$$i \prec v \prec o$$

Like causal flow, gflow is also a sufficient condition for robustly deterministic computation. Unlike causal flow, gflow is also a necessary condition, and thus gflow may be viewed as equivalent to robust determinism [29, Theorems 2 and 3]:

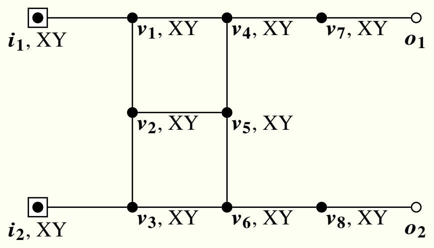
Theorem 2.6.9. Let (G, I, O, λ) be a labelled open graph with planar measurements only. The following statements are equivalent:

- There exists a robustly deterministic measurement pattern implementing $(G, I, O, \lambda, \alpha)$ for any α,
- The labelled open graph (G, I, O, λ) has gflow.

Example 2.6.10. The labelled open graphs from Examples 2.6.4 and 2.6.8 have gflow. Therefore, there exist robustly deterministic measurement patterns implementing the given computations.

Based on Theorem 2.6.9, for labelled open graphs without gflow, we can conclude there is no corresponding robustly deterministic measurement pattern:

Example 2.6.11. Consider the following labelled open graph from [29, Figure 7]:



It can be shown that it does not have gflow. Therefore, there is no robustly deterministic measurement pattern implementing it.

A full proof that the labelled open graph in Example 2.6.11 has no gflow is quite tedious: one must verify that no pair $(c, \prec)$ of correction function and partial order satisfies all conditions of gflow. The procedure can be somewhat simplified by considering a focused gflow instead. Initially defined in [125, Definition 5] for XY only case. Later adapted to all three planes in [15, Subsection 3.3] together with procedure transforming gflow into focused gflow:

Definition 2.6.12. A gflow $(c, \prec)$ on a labelled open graph (G, I, O, λ) is called *focused* if for all $v \in \bar{O}$:

- $\forall w \in \bar{O} \cap c(v).\lambda(v) \neq XY \to w = v$ and
- $\forall w \in \bar{O} \cap \text{Odd}(c(v)).\lambda(v) = XY \to w = v$.

Theorem 2.6.13. If a labelled open graph has gflow, it also has a focused gflow.

The conditions of a focused gflow can be interpreted as follows: Pauli X corrections can only be applied to XY measured vertices and Pauli Z corrections can only be applied to YZ or XZ measured vertices. Focussing conditions streamline the correction procedure by allowing only one type of correction to be applied to each vertex, which could be beneficial when constructing real-world architecture for MBQC. A particularly nice result about focused flow is known when $|I| = |O|$ ([125, Subsection 3.1] for XY only case, extention to all planes follows from Pauli flow version stated in Theorem 2.6.24):

Theorem 2.6.14. If a labelled open graph has gflow and $|I| = |O|$, then it has a unique (up to a weakening of the partial order) focused gflow.

Due to the above theorem, focused flow can be considered canonical or normal form flow whenever $|I| = |O|$. While this restriction on inputs and outputs may seem very strong, the cases with $|I| = |O|$ are, in fact, the most common, as then the corresponding robustly deterministic measurement patterns correspond to unitaries rather than just unitary embeddings.

We can now prove that the labelled open graph from Example 2.6.11 has no gflow:

Example 2.6.15. Consider labelled open graph from Example 2.6.11. Suppose it has gflow. Then, by Theorem 2.6.13, it has a focused gflow $(c, \prec)$. Suppose that $v_2 \notin c(v_5)$. Necessarily, we must have $v_5 \in \mathrm{Odd}(c(v_5))$, necessarily $v_4 \in c(v_5)$ or $v_6 \in c(v_5)$. Since $i_1, i_2, v_2 \notin c(v_5)$, necessarily v_1 or v_3 must be in $\mathrm{Odd}(c(v_5))$, contradiction with focussing condition. Thus, $v_2 \in c(v_5)$. Now, if $v_1 \in c(v_2)$, then $i_1 \in \mathrm{Odd}(c(v_2))$, contradicting focussing conditions, so $v_1 \notin c(v_2)$. Analogously, $v_3 \notin c(v_2)$ and thus necessarily $v_5 \in c(v_2)$. However then $v_5 \in c(v_2)$ and $v_2 \in c(v_5)$, which implies $v_2 \prec v_5$ and $v_5 \prec v_2$ contradicting $\prec$ being a strict partial order. Therefore the above labelled open graph does not have a focused gflow, and hence no gflow at all.

By considering a focused gflow instead of any gflow, we were able to construct a proof that a particular labelled open graph does not have gflow and therefore cannot be implemented in a robustly deterministic way. However, to provide the exact proof we had to consider multiple different cases, which is nonideal. This is a common problem when working with standard definitions of flows: proving even a simple statement often requires detailed consideration of which of the condition breaks. Similarly, verification that a pair is a flow is also tedious and likewise requires checking every single condition. These processes can be automated and in fact the linearity of the flow conditions allows construction of polynomial-time algorithms for finding gflow. We will study and improve such algorithms in great detail in later chapters.

2.6.3 Pauli flow

The generalised flow provides necessary and sufficient condition for robust determinism when all vertices are measured in the planes of Bloch sphere. Yet, a more general variant has been defined: Pauli flow. To motivate Pauli flow, we return to the Example 2.6.11. The labelled open graph presented there cannot be implemented in a robustly deterministic way as there is no causaly sound way to correct vertices v_2 and v_5. However, if we choose the angle at which vertex v_5 is measured to be 0 or π, then vertex v_5 is measured in Pauli X basis and the causality issue vanishes: we could measure first vertex v_5, and later still correct v_2 utilising the v_5 as a corrector. The reason this approach works is because application of the Pauli X gate to an X measured vertex does not affect the measurement outcome probabilities, and thus we can 'allow' application of the X gate to an already measured vertex. In other words, Pauli measurements are more flexible for corrections. The structure which allows full utilisation of this flexibility is the Pauli flow [29, Definition 5]:

Definition 2.6.16 (Pauli flow)**.** A *Pauli flow* for a labelled open graph (G, I, O, λ) is a pair $(c, \prec)$, where c is a correction function for (G, I, O, λ) and $\prec$ is a strict partial order on $\bar{O}$, such that for all $u \in \bar{O}$:

(P1) $\forall v \in c(u). u \neq v \wedge \lambda(v) \notin \{X, Y\} \Rightarrow u \prec v$

(P2) $\forall v \in \mathrm{Odd}(c(u)). u \neq v \wedge \lambda(v) \notin \{Y, Z\} \Rightarrow u \prec v$

(P3) $\forall v \in \bar{O}. \neg(u \prec v) \wedge u \neq v \wedge \lambda(v) = Y \Rightarrow v \notin \mathrm{Odd}⟦c(u)⟧$

(P4) $\lambda(u) = XY \Rightarrow u \notin c(u) \wedge u \in \mathrm{Odd}(c(u))$

(P5) $\lambda(u) = XZ \Rightarrow u \in c(u) \wedge u \in \mathrm{Odd}(c(u))$

(P6) $\lambda(u) = YZ \Rightarrow u \in c(u) \wedge u \notin \mathrm{Odd}(c(u))$

(P7) $\lambda(u) = X \Rightarrow u \in \mathrm{Odd}(c(u))$

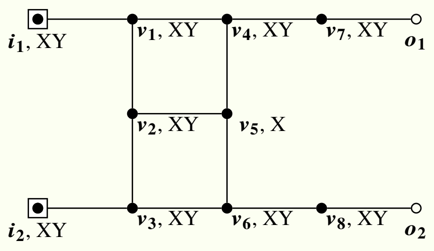
(P8) $\lambda(u) = Z \Rightarrow u \in c(u)$

(P9) $\lambda(u) = Y \Rightarrow u \in \mathrm{Odd}⟦c(u)⟧$.

The sets $c(v)$ for $v \in \bar{O}$ are called the *correction sets*.

We can now verify that a tranformed variant of a labelled open graph in Example 2.6.11 does have Pauli flow:

Example 2.6.17. Consider the following labelled open graph:



It does have Pauli flow with the correction function and relevant odd neighbourhood given in a table:

	$c(v)$	$\mathrm{Odd}(c(v))$
i_1	v_1, v_5, v_8	i_1, o_2
i_2	v_3, v_5, v_7	i_2, o_1
v_1	v_2, v_6, o_2	v_1
v_2	v_5, v_7, v_8	v_2, o_1, o_2
v_3	v_2, v_4, o_1	v_3
v_4	v_7	v_4, o_1
v_5	v_2, v_4, v_6, o_1, o_2	v_5
v_6	v_8	v_6, o_2
v_7	o_1	v_7
v_8	o_2	v_8

and the partial order is, for instance:

$$v_5, i_1, i_2 \prec v_1, v_3 \prec v_2, v_4, v_6 \prec v_7, v_8 \prec o_1, o_2$$

and another example, featuring all measurement labels:

Example 2.6.18. Consider the labelled open graph from Example 2.5.10. It does have Pauli flow with the correction function and relevant odd neighbourhood given in a table:

	$c(v)$	$\mathrm{Odd}(c(v))$
i	b, o	i, a
a	a, b, e, o	a, o
b	e	b, o
e	o	e
d	b, d, o	a, f
f	f	d

and the partial order is, for instance:

$$i, d, e \prec a \prec b, f \prec o$$

Just like gflow generalises causal flow, Pauli flow generalised gflow:

Observation 2.6.19. A gflow is also a Pauli flow.

Furthermore, Pauli flow is once again a sufficient [29, Theorem 4] and necessary [128, Theorem 3] condition for robust determinism on resource states given as labelled open graphs:

Theorem 2.6.20. Let (G, I, O, λ) be a labelled open graph. The following statement are equivalent:

- There exists a robustly deterministnic measurement pattern implementing $(G, I, O, \lambda, \alpha)$ for any α,
- The labelled open graph (G, I, O, λ) has Pauli flow.

The above statement is true when considering existence of any robustly deterministic measurement pattern. However, if the order in which vertices should be measured is fixed, then Pauli flow is no longer necessary, and instead equivalence is captured by *shadow Pauli flow* which we do not explore in this chapter [128, Theorem 1].

Example 2.6.21. The labelled open graph from Example 2.6.17 has Pauli flow. Therefore, there exists robustly deterministic measurement pattern implementing the given computation.

The notion of focussing can also be extended to Pauli flow [153, part of Definition 4.3] (throughout that paper, 'focused' is spelled 'focussed' instead):

Definition 2.6.22 (Focused Pauli flow)**.** The Pauli flow $(c, \prec)$ is *focused* when for all $v \in \bar{O}$ the following hold:

(F1) $\forall w \in (\bar{O} \setminus \{v\}) \cap c(v).\lambda(w) \in \{XY, X, Y\}$

(F2) $\forall w \in (\bar{O} \setminus \{v\}) \cap \mathrm{Odd}(c(v)).\lambda(w) \in \{XZ, YZ, Y, Z\}$

(F3) $\forall w \in (\bar{O} \setminus \{v\}).\lambda(w) = Y \Rightarrow w \notin \mathrm{Odd}⟦c(v)⟧$, where $w \notin \mathrm{Odd}⟦c(v)⟧$ is equivalent to $w \in c(v) \Leftrightarrow w \in \mathrm{Odd}(c(v))$.

Focused Pauli flow has similar properties as focused gflow: existence is stated in [153, Lemma 4.6], while uniqueness follows from [153, consequence of Theorem 5.4]:

Theorem 2.6.23. For any open labelled graph, if a Pauli flow exists, then there also exists a focused Pauli flow.

Theorem 2.6.24. If a labelled open graph has Pauli flow and $|I| = |O|$, then it has a unique (up to a weakening of the partial order) focused Pauli flow.

Example 2.6.25. The Pauli flow in Example 2.6.17 is focused.

While the presented gflow definition is already difficult to work with due to the necessity of checking multiple conditions, the Pauli flow definition is borderline impossible to use in the outlined form. Developing a more intuitive, human-friendly algebraic interpretation of flow is the subject of my work in Chapter 5.

2.6.4 Other flows

While Pauli flow is equivalent to robust determinism on labelled open graphs with all six types of labels, a number of other flow structures are defined for other variants of quantum computation. As mentioned earlier, in [128], a shadow Pauli flow is defined, equivalent to robust determinism when the labelled open graph comes with a forced partial order. Two more flow variants have been constructed in MBQC settings other than labelled open graphs on qubits. In [173], a gflow equivalent to hypergraph states was defined. Finally, in [24], a Zd flow was introduced, which relates to determinism on qudit graph states. Yet another structure related to MBQC flows was defined in [59], the so-called Pauli fusion flow, which guarantees determinism in Pauli fusion quantum computation, an abstract model related to ZX Calculus.

2.6.5 Finding flows

We argued that robust determinism is a desirable property which guarantees that the computation can be performed in the MBQC model independently of the observed measurement outcomes. Robust determinism corresponds to flow structures, and thus, asking whether computation can be performed in a robustly deterministic way is equivalent to asking whether the underlying labelled open graph has Pauli flow. Thus, flow-finding algorithms are critical and have been researched extensively. Causal flow was the first flow for which a polynomial-time algorithm was found [55, 56]. The algorithm was improved to $O(m)$ in [126], where m is the number of edges. Also in [126], an $O(n^4)$ method for finding XY-only gflow was found, where n is the number of vertices. This result was corrected and extended to all three planes in [15]. These gflow finding algorithms always return a focused gflow. For Pauli flow, an $O(n^5)$ was found in [153]. While the algorithm is not guaranteed to find focused flow, a focussing algorithm transforms arbitrary Pauli flow into focused one in $O(n^3)$ time. Later, Simmons improved his result of finding Pauli flow to $O(n^4)$ in a pull request to the quantum compiler t $|\text{ket}\rangle$ [154] and the improved algorithm is included in his (yet unpublished) PhD thesis. All gflow and Pauli flow algorithms utilise a layer-by-layer approach: the algorithm loops through vertices, detecting which vertices can be corrected without imposing order restrictions between two non-yet-solved vertices. Correctable vertices are detected by solving a linear system, which requires $O(n^3)$ steps by utilising Gaussian elimination. The process must be repeated at most $O(n)$ times, leading to overall complexity of $O(n^4)$. I improve on gflow and Pauli flow finding methods in Chapter 6. Other flow algorithms for their corresponding flow structures are included in [59, 24, 128].

2.6.6 Applications

Due to the flow correspondence to robust determinism, most of the MBQC applications discussed in Subsection 2.5.4 are also flow applications. Flow is also used in programming tools for MBQC, for example, see [160]. Flow structures can also be used for the optimisation of robustly deterministic MBQC schemes [15, 172] by transforming schemes only in ways that preserve flow. While translating a circuit to the MBQC model is straightforward, the opposite direction is far more challenging. Flow structures guarantee that efficient circuit extraction is possible [64, 15, 153, 157, 172]. The circuit extraction itself runs in time $O(n^3)$ when using gflow-based methods [64, 15], which is faster than corresponding algorithm for finding flow: in this cases, circuit extraction method requires flow to exist, but it does not need to be known to perform circuit extraction. On the other hand, the Pauli flow-based method of circuit extraction in [153] relies on knowing the flow: i.e. the first part of the circuit extraction method is to find Pauli, and the remaining part runs in $O(n^2)$ steps. The circuit extraction application is essential for ZX Calculus, and we discuss it in Subsection 3.3.2. Flow has also been considered for strategies of circuit extraction that go beyond robust determinism, for instance, by computing how far a given MBQC scheme (or ZX diagram) is from having flow [67].

Chapter 3

Graphical Calculi

In this chapter, we motivate the use of string diagrams and rewrite rules to alleviate some issues encountered in other representations of quantum computation. We focus on phase-free ZH Calculus and other related calculi, such as ZX Calculus.

In the previous chapter, we examined the mathematical model of quantum computation and two models for its realisation: the circuit model and MBQC. Representing computation via a circuit comes with many issues. Circuits often become very large (and hence less readable) even when representing a simple quantum process. Another problem with circuits is that they are challenging to transform – while rewriting rules for circuits exist, they tend to become very complex, for example, [43]. Finally, circuits are best suited to reason about unitaries which is problematic when studying arbitrary linear processes.

Similar issues arise when representing MBQC in the usual way: the graph representation of computation is again challenging to modify, while the expressions in the measurement calculus do not offer visualisation of the computation.

Although the circuit model and standard MBQC presentations function well as machine languages, humans need different representations to reason effectively about quantum computation. One solution is to use graphical calculi built with string diagrams.

Structure: In Section 3.1, we discuss string diagrams in general and briefly consider their categorical background. Next, in Section 3.2, we define ZH Calculus and study some of its properties and applications. After that, in Section 3.3, we explore a more well-known ZX Calculus and its relation to flow-structures and MBQC. Lastly, in Section 3.4, we look at other graphical calculi for quantum computation, ending this chapter.

3.1 String diagrams

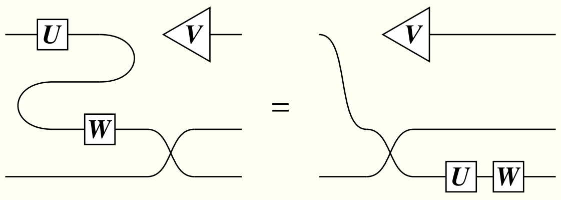
We start with a brief introduction to string diagrams. String diagrams form a graphical language with applications in many areas of science, including quantum computation. The diagrams consist of boxes representing maps and wires connecting boxes representing identities.

A string diagram is a graphical representation of a morphism in a monoidal category, i.e. a category equipped with a tensor product. All diagrams are read from left to right. The wires represent identity morphisms on particular systems, i.e. objects of the category. Combining two diagrams horizontally (plugging one diagram into another) corresponds to composition, while combining two diagrams vertically (stacking one diagram on top of another) corresponds to tensor product.

On the premises, boxes are rectangles, but modifying them to have non-rectangular shapes is often more convenient. For instance, states and effects are commonly denoted with triangles, while some maps are represented with circles.

The structure of a monoidal category makes it possible to formalise intuitive transformations of string diagrams:

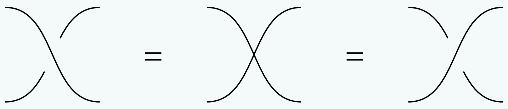
Example 3.1.1. Consider the following diagrammatic equation:



In this example, a *snake* is expanded, and next W and U are both pushed through a crossing.

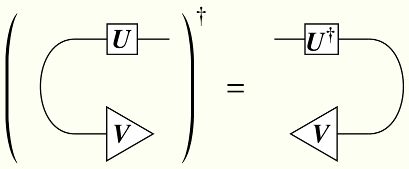
In this work, we use string diagrams to represent quantum computation. The objects of the category are Hilbert spaces, while the boxes represent linear maps. From the category theory perspective, we work with *dagger symmetric monoidal category $FdHilb$* of finite-dimensional Hilbert spaces. Since the carriers of quantum information throughout this thesis are qubits (and specifically not qudits), we are restricted to Hilbert spaces with powers of two degrees. The fact that the monoidal category is *symmetric* means that it is a *braided* monoidal category and the underlying braided structure is symmetric:

Proposition 3.1.2. Wires can be crossed, and when crossing two wires, there is no distinction between the top and bottom wires.



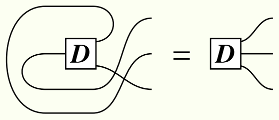
The *dagger* in dagger symmetric monoidal category [151] means that each process has a dual process: Each diagram can be mirrored, with boxes being mapped to their adjoints. For quantum computations, the dagger corresponds to the adjoint of a map with respect to an inner product structure, typically simplifying to the conjugate transpose operation.

Example 3.1.3. An example of the dagger operator:



The categorical structure also motivates some rewrite rules: The boxes can be moved, and wires can be stretched and crossed arbitrarily as long as the connections between boxes and dangling wires are preserved. The edges of a box are also not ordered; however, the order of the open ends of the edges does matter. In particular, the generator boxes of the calculi we work with satisfy the property of *flexsymmetry* [32], meaning that we do not even have to distinguish between input and output wires of individual boxes (example taken from [32]):

Example 3.1.4.

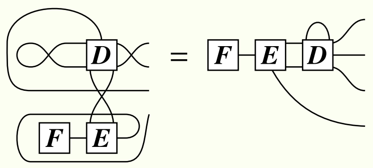


Hence, even though technically each edge is either an input or output edge, we can draw them as if they were starting in an arbitrary part of the box.

Collectively, the transformations of the diagrams arising from the categorical structure and flexsymmetry are known as *only topology matters*[2] (OTM). For example, OTM allows us to derive the following equality between diagrams:

[2]In literature, it is also known as *only connectivity matters*.

Example 3.1.5. Suppose that D and E are flexsymmetric. Then:



I do not study the category theory background of string diagrams in this thesis. Instead, string diagrams are applied to represent quantum systems via graphical calculi for quantum computation. Thus, I will not mention category theory later on. Readers interested in this topic should consult one of the following:

- [88] for a detailed categorical background;
- [45] for a detailed categorical background concerning ZX Calculus;
- [46] or [107] for a more general categorical overview concerning ZX Calculus; and
- [171] for a shorter overview of ZX Calculus linking it to other subfields of quantum computation.

3.2 ZH Calculus

ZH Calculus [13] and its phase-free variant [14] are graphical calculi representing quantum computation and were designed to efficiently reason about Toffoli+H universal gate set.

While ZH Calculus is less well-known than ZX Calculus which we discuss in Section 3.3, ZH and particularly phase-free ZH offers a strong toolkit to reason about counting complexity which will be essential in Chapter 4. Furthermore, phase-free ZH Calculus is in some sense 'minimal' among Z-like graphical calculi and thus complexity results about phase-free ZH typically extend to other Z-like calculi, while the reverse is not necessarily the case.

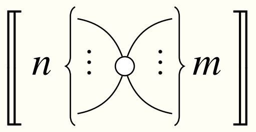
Firstly, we define generators of ZH Calculus in Subsection 3.2.1. We argue these generators suffice for universal quantum computation in Subsection 3.2.2. Next, in Subsection 3.2.3, we show how the diagrams can be transformed with rewrite rules and explain the concept of completeness. Finally, we look at applications of ZH Calculus in Subsection 3.2.4.

3.2.1 Generators

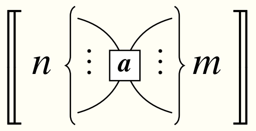
We start by formally defining ZH diagrams.

Notation. We use double brackets $\llbracket D \rrbracket$ for the algebraic interpretation of a diagram D.

Definition 3.2.1. *ZH diagrams* are string diagrams constructed with two generators: *white Z spiders* (○) and *H-boxes* (□):



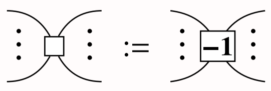
$$:= |0\rangle^{\otimes m} \langle 0|^{\otimes n} + |1\rangle^{\otimes m} \langle 1|^{\otimes n}$$



$$:= \sum_{i_1,\dots,i_m,j_1,\dots,j_n \in \{0,1\}} a^{i_1\dots i_m j_1 \dots j_n} |i_1 \dots i_m\rangle \langle j_1 \dots j_n|$$

The exponent $i_1 \dots i_m j_1 \dots j_n$ in the H-box's algebraic interpretation equals 0 unless all variables are equal 1, in which case the exponent itself also equals 1. Hence, the matrix representation of an H-box contains 1 in all entries except for the bottom right corner, where we have a instead.

Notation. When the number inside an H-box satisfies $a = -1$, we do not write it:



An H-box with -1 that has precisely one left and one right leg is a rescaled Hadamard. Thus, the H-box can be viewed as a generalisation of the Hadamard gate.

We will mainly focus on the phase-free variant of ZH, in which we only allow $a = -1$ as a generator:

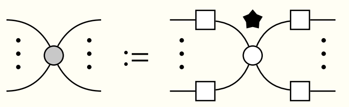
Definition 3.2.2. *Phase-free ZH diagrams* are string diagrams constructed with three generators: white Z spiders, H-boxes with number -1, and a *star* generator:

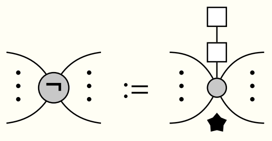
$$\llbracket \bigstar \rrbracket = \frac{1}{2}$$

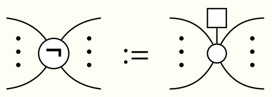
Even though we only include H-boxes with -1 as generators, H-boxes with arbitrary dyadic rational numbers can still be derived within phase-free ZH [14]. The star generator has no legs and represents a number. It is added to ensure proper scaling of the diagrams.

We also define three derived generators in phase-free ZH:

Definition 3.2.3. *dark Z spider*, (◉), *white Z* NOT (¬), and *dark X* NOT (¬) are three derived generators of phase-free ZH Calculus and are defined as follows:


(X)


(NOT)


(Z)

Spiders can have any number of legs, including zero. For some numbers of legs additional names are used:

Definition 3.2.4. We call spiders with precisely zero legs *scalars*. We also refer to diagrams without any dangling edges as *scalars*.

3.2.2 Universality

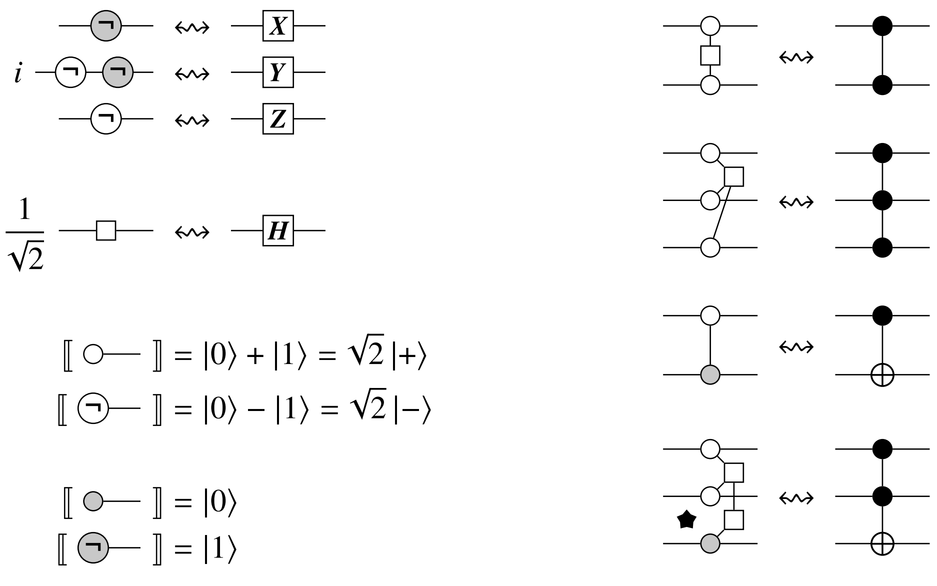
ZH Calculus is *universal* for quantum computation, meaning every linear process of a quantum system on qubits can be represented in ZH Calculus.

Further, the phase-free ZH is universal for quantum computation corresponding to the Toffoli+H universal gate set. We present encodings of some basic states and gates within phase-free ZH in Figure 3.1.

In ZH, for any complex number k, it is always possible to construct a scalar diagram with algebraic interpretation equal k; it is a scalar H-box generator (i.e. an H-box with zero legs) with number k inside. However, in phase-free ZH, this is not always possible. In phase-free ZH, we can only construct diagrams whose matrix interpretations contain dyadic rationals: We only have integers in matrix interpretations of spider and H-box generators, and an extra $\frac{1}{2}$ scalar from the star.

For the above reason saying phase-free ZH corresponds to Toffoli+H can be a bit misleading: It corresponds to Toffoli+H up to scaling. For example, $\frac{1}{\sqrt{2}}$ required for exact representation of the Hadamard gate is impossible to obtain in phase-free ZH. There are ways around this issue, for instance, one could redefine the star generator as $\frac{1}{\sqrt{2}}$ instead of $\frac{1}{2}$. However, we will keep using the standard $\frac{1}{2}$ definition so that matrix representations of all phase-free diagrams contain dyadic rationals only.

In Figure 3.1, the ZH representations of the CZ gate and CCZ gate are very similar: They both contain wires with white spiders connected to an H-box. As it turns out, an

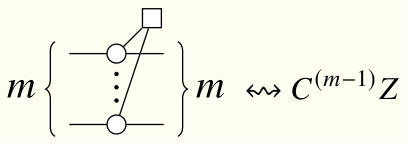


X
i
Y
Z
$\frac{1}{\sqrt{2}}$
H
$[\![\ ]\!] = |0\rangle + |1\rangle = \sqrt{2}\,|+\rangle$
$[\![\ ]\!] = |0\rangle - |1\rangle = \sqrt{2}\,|-\rangle$
$[\![\ ]\!] = |0\rangle$
$[\![\ ]\!] = |1\rangle$

Figure 3.1: Basic states and gates in ZH Calculus.

H-box connecting more wires also corresponds to a generalised controlled Z gate:

Example 3.2.5. For all $m \in \mathbb{N}$:

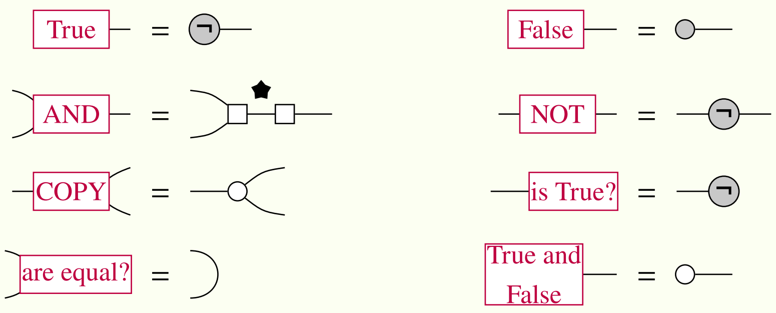


where by $C^{(m-1)}Z$ we mean a generalised controlled Z gate on m qubits out of which $m-1$ are control. In particular, when $m = 2$, we obtain the representation of CZ gate, i.e. the first correspondence in the right column of Figure 3.1; when $m = 3$ we get CCZ gate, i.e. the second correspondence in the right column of Figure 3.1. When $m = 1$, we get just Z gate (cf. with the definition of a white NOT). Finally, this notation makes sense in ZH even for $m = 0$, where the presented diagram becomes just an H-box with no legs, i.e. a global phase -1.

Such a simple representation of generalised controlled Z gates is convenient when working with hypergraph states [113, 38, 173].

ZH is also a valuable tool for reasoning about classical logic circuits [13, 14, 61, 114, 115]:

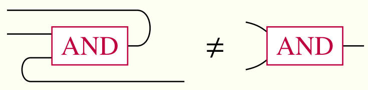
Example 3.2.6. We treat $|1\rangle$ as *True* and $|0\rangle$ as *False*. From these we can easily construct basic logic operations on these values as phase-free ZH diagrams:



Notation. To avoid confusion with an H-box, we represent derived constructions with purple boxes with plain text inside.

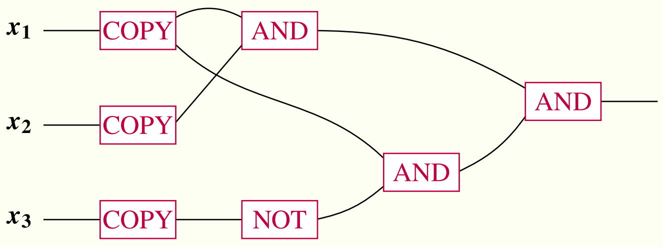
It may matter which wires are input and output in the derived constructions. For instance:

Example 3.2.7. The derived AND gate is not flexsymmetric:

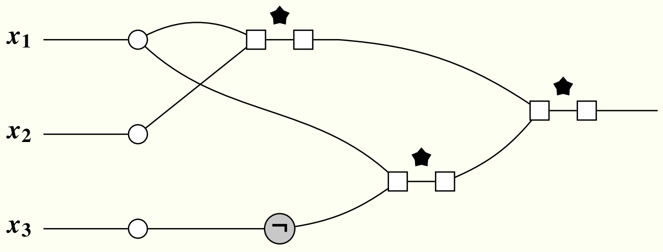


Using constructions from Example 3.2.6, we can represent arbitrary boolean formulae:

Example 3.2.8. Recall the formula $(x_1 \wedge x_2) \wedge (x_1 \wedge \neg x_3)$ from Example 2.3.9. Its ZH representation is as follows:



When translated directly into generators of phase-free ZH, we obtain:



The efficient representation of Boolean formulae in ZH is a powerful tool for reasoning about Boolean circuits: The dangling edges corresponding to the variables can be closed with the superposition of all possible valuations, leading to efficient representation of **#SAT** instances:

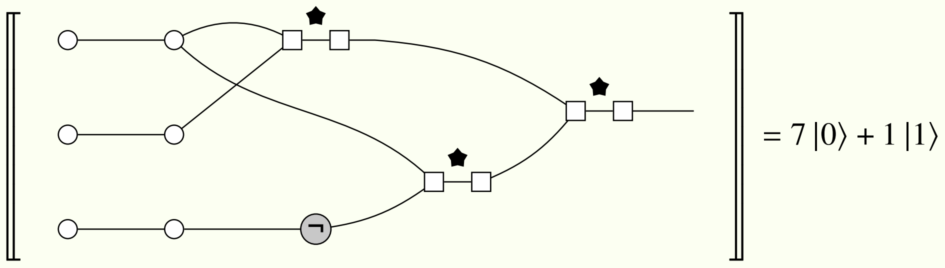
Lemma 3.2.9. Let ϕ be a boolean formula on variables $X = x_1, \ldots, x_n$ and D_ϕ be the phase-free ZH encoding of ϕ. The diagram obtained by attaching white spiders to the dangling edges corresponding to the variables has matrix representation $\textbf{\#SAT}(\phi, X)\,|1\rangle + (2^n - \textbf{\#SAT}(\phi, X))\,|0\rangle$.

Proof. We have:

$$\begin{aligned}
[\![D_\phi]\!]\,(\sqrt{2}^n\,|+\rangle^{\otimes n}) &= \sum_{v_1,\ldots,v_n \in \{0,1\}} [\![D_\phi]\!]\,|v_1 \ldots v_n\rangle \\
&= \sum_{v_1,\ldots,v_n \in \{0,1\}} |v(\phi)\rangle \\
&= \textbf{\#SAT}(\phi, X)\,|1\rangle + (2^n - \textbf{\#SAT}(\phi, X))\,|0\rangle
\end{aligned}$$

where v stands for the valuation of X mapping x_i to *True* when $v_i = 1$ and to *False* when $v_i = 0$. □

Example 3.2.10. Recall the Boolean formula from Examples 2.3.9 and 3.2.8. We can evaluate it simultaneously on all valuations:



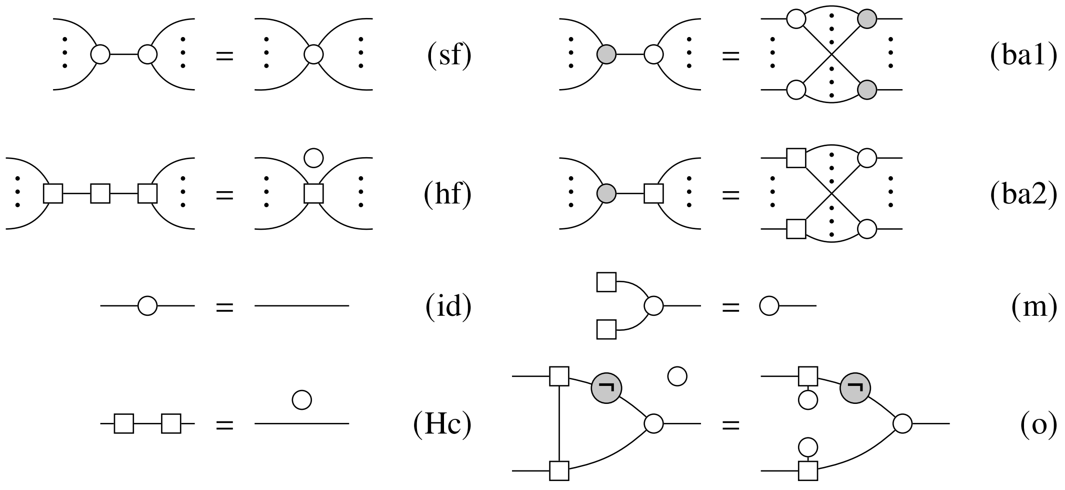


Figure 3.2: A complete set of rewrite rules for phase-free ZH Calculus [14, Figure 1].

The resulting diagram representation implied by Lemma 3.2.9 follows from Example 2.3.13.

ZH encodings of boolean formulae will be essential when we discuss the hardness of specific problems arising in graphical calculi, i.e. the main subject of Chapter 4.

3.2.3 Rewrite rules and completeness

Next, we look at the rewrite rules of ZH Calculus; see Figure 3.2. These rules are exact, i.e. all necessary scalar diagrams are included. While the star generator does not explicitly appear in any rules, it is a part of the dark spider and the dark NOT definitions. We also present some derived rules in Figure 3.3: these rules will be used in Chapter 4.

Going by columns, the rules are commonly referred to as spider and H-box fusions (sf and hf), identity rules (id and Hc), and bialgebra rules (ba1 and ba2). The last two rules are called multiplication (m) and ortho (o).

All phase-free ZH rewrite rules can be motivated by transformations between logical circuits [14]. As such, the rewrite rules are also *sound*, meaning that if one diagram can be rewritten to another, then the two diagrams represent the same linear map.

Furthermore, phase-free ZH is also *complete* [14], which means that if two diagrams represent the same linear map, one can be rewritten to another by a series of rewrite rules. The same result holds for the 'full' ZH with appropriate rewrite rules. The completeness proof follows the idea of 'normal form': given a diagram, a canonical normal form is defined based on the algebraic interpretation of the given diagram. Then, one shows that a diagram can be rewritten to corresponding normal form. Thus, if two diagrams represent

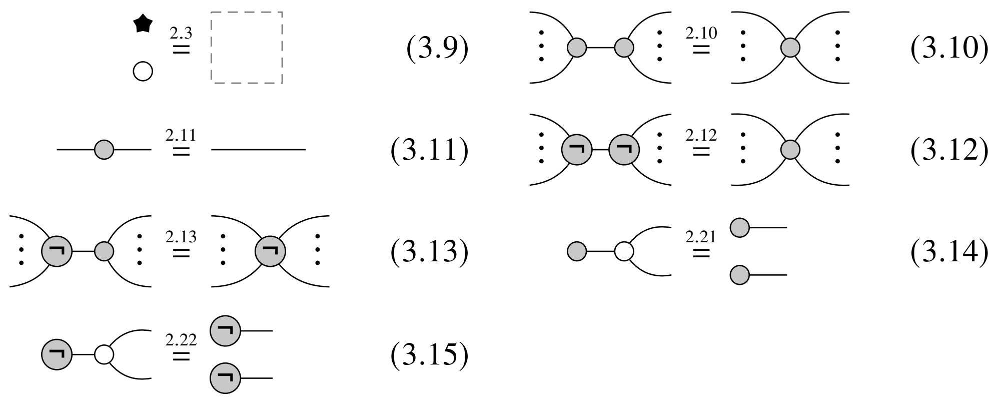


Figure 3.3: Some derived rewrites of phase-free ZH Calculus. The numbers above the equalities specify the corresponding Lemmata from [14]. The RHS of equation (3.9) contains an empty diagram.

the same map, they can be rewritten into the same normal form, and hence each other.

Normal form approach gives rise to a simple method for testing whether two diagrams are equal: simply compare their normal forms. This method fails for anything (approximately) universal in quantum computation: the normal forms are exponential in the size of initial diagrams, making such an approach for comparing diagrams extremely inefficient: not only exponentially many steps would be requires, one would also need exponential memory to store the diagrams. As we will explain later, exponential memory is likely not necessary to compare diagrams. Though our explanation does not conjure alternative to normal form reasoning for completeness results, it could motivate looking for alternatives that do not require exponential in size normal forms.

3.2.4 Applications

ZH Calculus is of theoretical significance as it complements algebraic structure of more popular ZX Calculus (which we consider in the next section) in a non-trivial way [34]. Due to its design, it also suits considerations of quantum computation that feature a high number of multi-controlled gates [168]. This paper does not use pure ZH Calculus but ZX Calculus with the addition of a ZH H-box. A similar approach can often simplify complex mathematical proofs that require multi-qubit entanglement; for example in [172]. Another instance with many multi-controlled gates is the hypergraph MBQC. ZH offers a clean presentation of measurement byproduct mitigation in universal hypergraph states [113, 38]. Finally, ZH was used to find the first extension of MBQC flows to hypergraph settings [173].

Besides MBQC, ZH has also been applied when reasoning about counting problems.

In [61], the authors showed how to efficiently represent counting problems in ZH and how rewrite rules allow fast solving of diagrams corresponding to tractable problems. Building on these ideas, in [115], ZH is used to rediscover the hardness of various problems from computational complexity, for example, showing that **#2SAT** is #$\boldsymbol{P}$-complete, reducing the length of the classical proof to a few diagrams. In [114], ZH was used to develop a graphical **#SAT** algorithm for a particular subclass of Boolean formulae. These results show that ZH can have applications not only in quantum computation but also in computational complexity. We will follow a similar path in Chapter 4, where we show how encoding #$\boldsymbol{SAT}$ instances in ZH can lead to the complexity results for various problems arising in graphical calculi.

3.3 ZX Calculus

While in the later parts of the thesis we use phase-free ZH Calculus, a more well-known ZX Calculus [45, 171] forms important motivation for studying flow structures outside of purely MBQC settings. ZX was originally formulated in 2007 [44] and is the oldest and most commonly used calculus from the family of Z-like calculi.

The essence of ZX Calculus is captured by the following sentence from ZX Calculus website [130]:

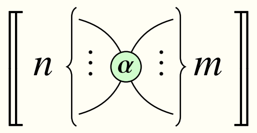
> The idea behind ZX is to 'split the atom' of well-known quantum logic gates to reveal the compositional structure inside.

In this section, we restrict to the definition of ZX diagrams and motivations for the later chapters: the problem of circuit extraction and heuristic solutions involving flow structures from MBQC.

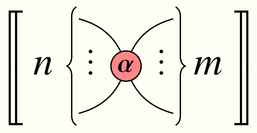
3.3.1 Generators

We again start by formally defining ZX diagrams.

Definition 3.3.1. *ZX diagrams* are string diagrams constructed with two generator *spiders*: *green Z spiders* (🟢) and *red X spiders* (🔴):



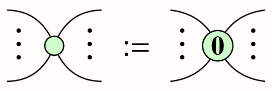
$$:= |0\rangle^{\otimes m}\langle 0|^{\otimes n} + e^{i\alpha}|1\rangle^{\otimes m}\langle 1|^{\otimes n}$$



$$:= |+\rangle^{\otimes m}\langle +|^{\otimes n} + e^{i\alpha}|-\rangle^{\otimes m}\langle -|^{\otimes n}$$

The number α inside a spider is called its *phase* and satisfies $\alpha \in \mathbb{R}$.

Notation. When the phase inside a spider satisfies $\alpha = 0$, we do not write it:



and similar for the red spiders.

We consider phases modulo 2π as spiders with phases differing by multiple of 2π have identical interpretations.

When using ZX Calculus, we often present a diagram *up to a scalar*, as motivated by the following definition:

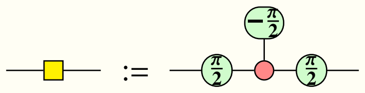
Definition 3.3.2. For a diagram D, we say it equals to a linear map M *up to a scalar* if there exists a number $k \in \mathbb{C} \setminus \{0\}$ such that $[\![D]\!] = kM$. We write $[\![D]\!] \approx M$.
For a pair of diagrams D_1 and D_2 we say they are equal *up to a scalar* if there exists a number $k \in \mathbb{C} \setminus \{0\}$ such that $[\![D_1]\!] = k\,[\![D_2]\!]$. We write $D_1 \approx D_2$.
In both cases, we refer to the number k as the *scalar*.

We have just defined five different meanings for the word *scalar*: generator with zero legs, diagram with zero legs, two meanings of *up to a scalar*, and as number from *up to a scalar* definition. It should be clear from context what we mean by a *scalar*, though at times we will specify exactly which *scalar* we mean.

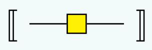
Like in ZH, any complex number can be represented via a scalar ZX diagram, but the construction is more involved then in the ZH case [171, Subsection 3.4].

There is also one derived generator in ZX Calculus: a yellow box which corresponds to the Hadamard gate.

Definition 3.3.3. A *yellow box* (🟨) is a derived generator defined as follows:



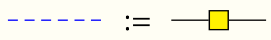
Observation 3.3.4.



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

We sometimes represented Hadamard wire with a dashed blue line:

Definition 3.3.5. A blue dashed edge in a diagram is defined as a wire with a Hadamard:



We will only use blue dashed wires when connecting two generators (i.e. not dangling wires).

Just like ZH, ZX is also universal [171, Subsection 3.7]. When equipped with appropriate rewrite rules, it is also complete. The completeness of Clifford fragment where all phases are multiples of $\frac{\pi}{2}$ was derived in [11]. Clifford+T fragment where all phases are multiples of $\frac{\pi}{4}$ is universal based on [98] with partial results in [10, 47]. ZX without restrictions on phases is also complete [136, 99]. Unlike phase-free ZH which only requires a few simple rewrite rules, the completeness of ZX Calculus outside Clifford fragment usually requires complex rewrite rules. For example, in [98], a rewrite rule involving twenty-one generators with only four dangling edges was necessary, while results for universality of the entire ZX feature Euler decompositions where phases undergo discontinuous transformation. Some of these results follow ZH strategy of 'normal forms', while other achieve completeness by translating to other graphical calculus (usually ZW Calculus) and arguing that the other calculus is complete.

Phase-free ZH can be translated to Clifford+T ZX by adopting construction of three-legged H-box in ZX [112], and noticing that interpretations of phase-free ZH spiders and ZX spiders with phases being multiples of π only differ by (derivable) scalars. Yet, translation from Clifford+T to phase-free ZH is impossible; again phase-free ZH diagrams have interpretations restricted to real-valued matrices. This makes phase-free ZH strictly 'smaller' than Clifford+T ZX Calculus, yet still approximately universal. At the same time, it is possible to translate between universal ZX and ZH calculi. Yet, since ZX and ZH have been designed with slightly different goals in mind, they are considered distinct calculi.

3.3.2 Circuit optimisation in ZX Calculus

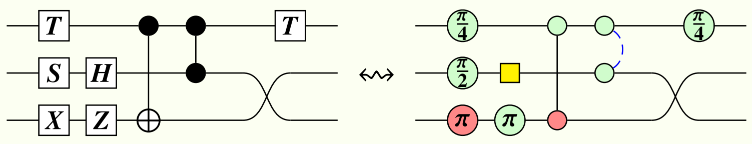
One of the main applications of ZX is circuit optimisation. Given a quantum circuit, can we find a more efficient one? A more efficient circuit may refer to a circuit with fewer T-gates, 2-qubit gates, or other metrics. While standard circuit diagrams are difficult to modify and optimise via existing rewrite rules for circuit, ZX overcomes this problems by instead operating on ZX diagrams whose transformations go beyond circuit diagrams.

A detailed review of circuit optimisation techniques utilising ZX Calculus is the subject of [72]. Circuit optimisation strategy based on ZX Calculus can be summarised as follows:

- Given an unoptimised circuit, translate it to ZX Calculus.
- Rewrite the resulting diagram to a graph-like form.
- Optimise the graph-like diagram by applying ZX rewrite rules.
- Extract the optimised quantum circuit from the final ZX diagram.

The first step follows from encodings of quantum gates in ZX Calculus, which for the most part is similar to ZH encodings from Figure 3.3. The exact presentation of ZX encodings can be found for instance in [171]. ZX is particularly well suited towards Clifford+T gate set:

Example 3.3.6. A circuit from Example 2.4.1 corresponds up to scalar to the following ZX diagram:



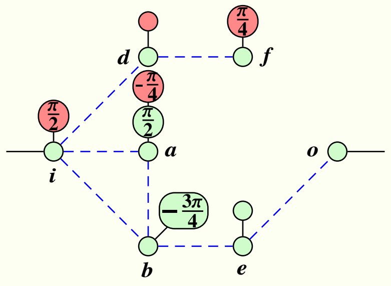
Omitting the next two steps from circuit optimisation strategy for the moment, let's focus on the last step: transforming optimised diagram back into a circuit, i.e. circuit extraction. In general, circuit extraction is #$\boldsymbol{P}$-hard [62] (we will provide a formal description of the problem in Chapter 4). For that reason, ZX-based circuit optimisation is doomed to fail unless we can ensure that circuit optimisation is possible.

Given any ZX diagram, it can be brought to a so-called *graph-like form* [64]. Such diagrams are effectively encodings of MBQC schemes from Definition 2.5.12: initial formulation encoded schemes with XY planar measurements only, but a natural extension to all three standard planes was considered in [15, 121, 122, 16]. If the labelled open graph underlying a diagram in graph-like form has Pauli flow, then circuit extraction is possible in deterministic polynomial time [64, 15, 153].

Example 3.3.7. Consider the labelled open graph from Example 2.5.10. Consider the measurement angle α given in the following table:

$v:$	i	a	b	e	d	f
$\alpha(v):$	$\frac{\pi}{2}$	$-\frac{\pi}{4}$	$-\frac{3\pi}{4}$	0	0	$\frac{\pi}{4}$

The ZX representation of the corresponding MBQC scheme is:



We only indicate the desired outcomes in the diagram. If the undesired outcome is observed, the corresponding process can be visualised in ZX by closing the dangling wire with effect with phase altered by π.

As of now, diagrams corresponding to labelled open graphs with Pauli flow are the largest subclass of diagrams for which circuit extraction is tractable. As explained in Subsection 2.6.6, the extraction method based on Pauli flow is dominated by finding Pauli flow. On the other hand, the gflow-based approach runs in $O(n^3)$ time where n is the number of vertices in the corresponding labelled open graph. The slower circuit extraction method for labelled open graphs with Pauli flow rather than gflow was an obstacle preventing wider application of Pauli flow in circuit optimisation methods. As mentioned, in Chapter 6, I improve flow finding algorithms: this way, the gap between gflow-based and Pauli flow-based circuit extraction complexity is resolved, opening potential for a wider application of Pauli flow-based circuit extraction.

Returning to the circuit optimisation overview, due to the limitation of cases in which circuit extraction is possible, existing methods must ensure that the final diagram, right before circuit extraction, has Pauli flow. The first two steps of the optimisation strategy result not in any graph-like diagram but, in fact, one with gflow. Thus, the challenge of the third step of the optimisation is to ensure that the final diagram has Pauli flow, given that the initial one does.

Theoretically, any technique ending with the underlying labelled open graph with Pauli flow could be incorporated into the optimisation process. Yet, if a diagram loses flow at some point, it can be challenging to regain it later without undoing the progress.

Therefore, existing optimisation results that follow this strategy preserve the flow on the way: all diagrams obtained during the third step correspond to a labelled open graph with flow.

The number of known rewrite rules that preserve flow is limited, but it suffices for powerful optimisation strategies. In [64], the authors used the local complementation rule (and its derivative: pivot) to maximally simplify Clifford circuits and to detect and optimise Clifford fragments of larger circuits. The method has since been refined, leading to T-count reduction methods [106][3], or two-qubit gate reduction [157].

These applications make the flow-preserving rewrites of extraordinary interest, and they have been a subject of multiple works [121, 122, 16]. Optimisation based on causal flow preservation has also been considered [89].

The optimisation techniques using flow and ZX Calculus were especially successful when considering the computation in the one-way model. In [15], measurement patterns were optimised, reducing the number of qubits in a flow-preserving manner. These results were later found to in fact produce an optimal number of qubits when nothing is assumed about the angles, i.e. when all measurement angles are treated as independent parameters [172].

The T-count reductions methods that do not directly invoke flow, but still arise from related concepts in MBQC we considered in [58, 57]. Some other circuit optimisation results used Pauli fusion flow [59, 69]. This structure is similar to MBQC flows in many ways, but a full connection has yet to be established. The techniques based on Pauli-fusion flow were used for optimisation results, for example, in [174].

3.3.3 Applications

Apart from circuit optimisation via MBQC techniques discussed in Subsections 3.3.2, ZX Calculus has found applications in many other subfields of quantum computation. A very detailed overview can be found in [171]. The ZX-calculus website [130] contains a maintained list of publications utilising ZX and similar graphical calculi.

Multiple software tools apply ZX Calculus, with PyZX [108] being the most prominent example. This tool implements some of the circuit optimisation methods that we discussed earlier. PyZX, and thus ZX, is also used in state-of-the-art compilers, like $\mathrm{t}\,|\mathrm{ket}\rangle$ [155]. Some older graphical tools utilising ZX Calculus include Quantomatic [109].

ZX Calculus has found many applications in Quantum Error Correction codes, ranging from explicit correspondence between variants of ZX and correction codes [104, 48, 103], through graphical analysis and design of new codes [94, 51, 74], to code verification [37]. Some older verification results utilised ZX in Quantomatic [65, 77].

[3]In this case, flow was not essential, but the optimisation techniques were based on the MBQC model.

Decomposition methods for ZX diagrams led to state-of-the-art results in quantum simulation; for example in [161].

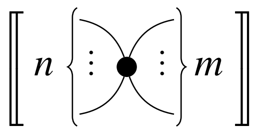
ZX Calculus has also been adapted to present quantum computation for teaching purposes. There are now two big textbooks [46, 107] about ZX Calculus and quantum computation and another one describing categorical background [88]. Multiple universities now use these textbooks; however, the diagrammatic reasoning in simpler terms has been tested even on high school level students [66].

Finally, the transformations between diagrams also reach outside of Quantum Computation – for instance, diagrammatic reasoning based on ZX Calculus has been used in Natural Language Processing [123].

The most important takeaway for us, is that when utilising ZX in a way that eventually needs translation to language of quantum computers, i.e. quantum circuit, then flow-based techniques are necessary to ensure existence of efficient circuit extraction procedure. Otherwise, we would run into $\#\boldsymbol{P}$-hard problem.

3.4 Other graphical calculi

There are other Z-like graphical calculi for quantum computation. For example, ZW is a graphical Calculus that contains Z and W spiders as generators. The W spider (●) is defined as follows:



$$:= \sum_{i_1+\cdots+i_m+j_1+\cdots+j_n=1} |i_1 \ldots i_m\rangle \langle j_1 \ldots j_n|$$

This spider is the sum of $|i_1 \ldots i_m\rangle \langle j_1 \ldots j_n|$ where the Hamming weight of variables equals 1.

Similar to the ZH H-box defined to efficiently reason about Toffoli circuits, the W spider is motivated by the need to capture three qubits entanglement that otherwise does not have a simple presentation in ZX or ZH, in this case: the $|W\rangle$ state. Since a three-legged Z spider can represent $|GHZ\rangle$, the ZW Calculus can efficiently represent all entanglement classes on three qubits [3].

ZW is universal and can be made complete by adding appropriate rewrite rules. The first completeness result for the entire ZX Calculus in [136] works by adapting the completeness result for ZW.

Aside from aiding in achieving complete results for other calculi, ZW Calculus has other applications. For instance, ZW combined with ZX can be used in machine learning [110]. Similar to ZH, ZW Calculus is also interesting from the perspective of counting problems [35].

There exists some limit [34] based on which new Z-like languages will eventually repeat the algebraic structure of other calculi. However, new graphical languages are still considered for more tailored applications. For example, SZX, scalable ZX Calculus, was defined for compact representation of multi-qubit processes with applications in formal verification and error correction [33], or ZQ, where Q stands for Quaternions, is defined for efficiently considering hardware-specific rotations [129]. Finally, there are extensions of ZX, ZH, and ZW to higher dimensions [176, 140, 60] or mixed dimensions [177]. The languages can also be combined by mixing generators from different calculi.

Critically, all universal graphical calculi contain linear processes that are representable in phase-free ZH. Based on that, we can argue that our complexity results about problem hardness for phase-free ZH extend to all other universal Z-like graphical calculi.

Chapter 4

Complexity Problems arising in Graphical Calculi

In the previous chapter, we explored a graphical language for quantum computation reasoning: ZH Calculus. The phase-free variant offers a simple set of generators that guarantee universality. The calculus is effective in MBQC and analysis of quantum circuits constructed with the universal gate set Toffoli+H. While circuits naturally translate to ZH diagrams, finding an ancilla-free circuit equivalent to a given diagram is hard. Here, we show that circuit extraction for phase-free ZH Calculus is $\#\boldsymbol{P}$-hard, extending the existing result for ZX Calculus. Another problem believed to be hard is comparing whether two diagrams represent the same process. We show that two closely related problems are $\boldsymbol{NP}^{\#\boldsymbol{P}}$-complete. The first problem is: given two processes represented as diagrams, determine the existence of a computational basis state on which they equalize. The second problem is checking whether the matrix representation of a given diagram contains an entry equal to a given number. The proof adapts the proof of Cook-Levin theorem to a reduction from a non-deterministic Turing Machine with access to $\#\boldsymbol{P}$ oracle.

Corresponding paper. Most of the novel results in this chapter have appeared in my paper [131] (single author).

4.1 Introduction

The central problem to ask about any model of computation is whether two given computations are equivalent. For graphical calculi, this problem is comparing diagrams:

CompareDiagrams
Input: two phase-free ZH diagrams D_1 and D_2.
Output: *True* if and only if $[\![D_1]\!] = [\![D_2]\!]$.

One of the ways to check whether two diagrams represent the same quantum process is to attempt rewriting one to the other. For graphical languages complete for quantum computation, the proof goes through exponential in size normal forms. This means that, concerning current knowledge, the number of rewrites transforming one diagram to another may be exponential, and the diagrams on the way can be exponential in size as well. The best-known upper bound for comparing diagrams is $\boldsymbol{coNP^{\#P}}$ [62].

The goal of the research is to establish a tight bound for comparing diagrams. The main motivations come from [62], where an upper bound for comparing diagrams is used to bound circuit extraction from above with $\boldsymbol{NP^{NP^{\#P}}}$. Further, the hardness of comparing diagrams would extend to the comparison of circuits with measurement post-selection and thus, it would connect to $\boldsymbol{PostBQP}$ (Bounded-error Quantum Polynomial-time with Post-selection [1]) and $\boldsymbol{PP}$ (Probabilistic Polynomial-time [80]) computation schemes. Another motivation was exploring the encoding of $\#\boldsymbol{P}$-hard problems as ZH diagrams, and to look for interesting completeness results.

In this chapter, we make progress towards the above goal. We prove two closely related problems arising in phase-free ZH Calculus are $\boldsymbol{NP^{\#P}}$-complete. These are some of the first examples of $\boldsymbol{NP^{\#P}}$-complete problems (other examples that do not directly relate to quantum can be found in [166, 134]). Comparing diagrams asks that a certain property holds for all entries in the matrix representation of two diagrams: they are equal if and only if they agree on all entries. The presented problems check whether there exists an entry satisfying the property. In other words, comparing diagrams concerns *universal* property, while the presented problems concern *existential* properties.

The first of the problems:

StateEq
Input: a pair of phase-free ZH diagrams D_1, D_2 with matching number of dangling edges.
Output: *True* if and only if there exists a computational basis state $|v\rangle$ such that $[\![D_1]\!]\,|v\rangle = [\![D_2]\!]\,|v\rangle$.

By interpreting a diagram with dangling edges as a quantum state, this problem can be viewed as checking whether there exists a choice of measurement outcomes (on the computational basis) that appears with the same probability for both quantum states given as diagrams. Another way to understand this problem is by checking whether matrix interpretations of given diagrams are equal in some positions.

The second problem is as follows, for $k \in \mathbb{Z}[\frac{1}{2}]$ (dyadic rationals):

> **ContainsEntry**
> **Input:** a phase-free ZH diagram D.
> **Output:** *True* if and only if k appears in the matrix interpretation of D.

In particular, when $k = 0$, the problem can be interpreted as: given a diagram representing a quantum state, does there exist a measurement choice (on the computational basis) that happens with probability 0? Our result also works when the number k is part of the input, rather than a fixed number on which the problem depends. Our main contribution is proving the following:

> **Theorem 4.1.1. StateEq** and **ContainsEntry** are both $\boldsymbol{NP^{\#P}}$-complete.

We also give a construction showing $\#\boldsymbol{P}$-hardness of circuit extraction from phase-free ZH – the original paper [62] showing $\#\boldsymbol{P}$-hardness of circuit extraction did not work for the phase-free variant – the construction used in the proof required $\frac{\pi}{2}$ phase spider, which does not exist in phase-free ZH.

> **CircuitExtraction**
> **Input:** a phase-free ZH diagram D_1 proportional to a unitary and a (finitely presented) set of quantum gates $\mathcal{G}$.
> **Output:** a polynomial in size of D_1 quantum circuit constructed with gates from $\mathcal{G}$ representing a process proportional to $[\![D_1]\!]$ or a message that no such circuit exists.

> **Theorem 4.1.2. CircuitExtraction** is $\#\boldsymbol{P}$-hard.

The trouble of proving $\boldsymbol{NP^{\#P}}$-hardness of **StateEq** and **ContainsEntry** comes from the lack of a simple $\boldsymbol{NP^{\#P}}$-complete problem that can be used in reductions. Crafting such a problem is the most challenging part; we will dedicate multiple sections to it.

Our proofs work for phase-free ZH Calculus. These results are also generalised to other graphical calculi, like ZX, ZH, and ZW. This is because the phase-free ZH diagrams can always be represented in those other graphical calculi, and the translation from phase-free ZH to other calculi is polynomial in size.

Structure: The rest of this chapter is dedicated to proofs of the outlined results and discussion of their limitations. The main proof is split into multiple sections. In Section 4.2, we give upper bounds for the seen problems. The hardness proof has three parts. Firstly, in Section 4.3, we show that $\boldsymbol{NP}^{\#\boldsymbol{P}} = \boldsymbol{NP}^{\boldsymbol{C}_=\boldsymbol{P}[1]}$, so we only need to prove $\boldsymbol{NP}^{\boldsymbol{C}_=\boldsymbol{P}[1]}$-hardness (see below for notation explanation). Secondly, in Section 4.4, we craft an $\boldsymbol{NP}^{\#\boldsymbol{P}}$-complete problem for reduction. Thirdly and finally, in Section 4.5, we encode the problem in phase-free ZH, obtaining the reductions proving $\boldsymbol{NP}^{\#\boldsymbol{P}}$-hardness. After the main proof, in Section 4.6, we provide outlined results for the circuit extraction hardness. Lastly, in Section 4.7, we discuss limitations of the work in this chapter: in particular, we explain why the results do not extend to the hardness of the comparing diagrams problem.

Notation. We write $[1]$ next to the oracle to indicate that the oracle is used precisely once, and the answer to the oracle query is returned as the output of the entire computation.

4.2 Upper bounds for problems in graphical calculi

Firstly, we provide upper bounds for various problems arising in graphical calculi. The oracle bounds are based on [62] and the utilisation of the following problem:

ScalarDiagram
Input: a phase-free ZH diagram D with no dangling edges (i.e. a scalar diagram).
Output: the number $[\![D]\!]$.

It is known that this problem is in $\boldsymbol{FP}^{\#\boldsymbol{P}}$ [115]. A version that works for some tensor networks other than only phase-free ZH follows from the Holant framework [31, page 212] and [12, Lemma 54]. Yet another explanation was given in [62], where authors used the method introduced in [4] to obtain a similar bound for circuits rather than a diagram, though, as the authors point out, the method works for diagrams as well. By pushing the deterministic polynomial time operations to instead be performed by the NDTM: $\boldsymbol{NP}^{\boldsymbol{FP}^{\#\boldsymbol{P}}} \subseteq \boldsymbol{NP}^{\#\boldsymbol{P}}$. Hence, for the containment of **StateEq** and **ContainsEntry** in $\boldsymbol{NP}^{\#\boldsymbol{P}}$, it suffices to show containment in $\boldsymbol{NP}^{\textbf{ScalarDiagram}}$.

Theorem 4.2.1. The problem **StateEq** is in $\boldsymbol{NP}^{\#\boldsymbol{P}}$.

Proof. We construct a polytime NDTM $\mathcal{M}$ with **ScalarDiagram** oracle that takes input (D_1, D_2) and non-deterministically chooses a computation basis state $|v\rangle$. Next, using the oracle, $\mathcal{M}$ computes both $[\![D_1]\!]\,|v\rangle$ and $[\![D_2]\!]\,|v\rangle$. Finally, $\mathcal{M}$ accepts if the two values are equal. $\square$

For the second problem, we proceed similarly:

Theorem 4.2.2. The problem **ContainsEntry** is in $\boldsymbol{NP^{\#P}}$.

Proof. We construct a polytime NDTM $\mathcal{M}$ with **ScalarDiagram** oracle that takes input D and non-deterministically chooses a computational basis state $|v\rangle$. Next, using the oracle, $\mathcal{M}$ computes $[\![D]\!]\,|v\rangle$. Finally, $\mathcal{M}$ accepts if the obtained value equals k. $\square$

From [62], we know that the decision variant of **CircuitExtraction** is in $\boldsymbol{NP^{NP^{\#P}}}$: decide the existence of a required circuit without returning it as an output. The overview is as follows: non-deterministically choose a candidate circuit C and, using $\boldsymbol{NP^{\#P}}$ oracle, verify that the input diagram and C are proportional. If they are, return C; otherwise, return that no circuit exists. Using such approach for (functional) **CircuitExtraction** problem leads to upper bound of $\boldsymbol{NPMV^{NP^{\#P}}}$, where $\boldsymbol{NPMV}$ ($\boldsymbol{NP}$ with Multiple Value) is the class from [23].

Finally, for the problem **CompareDiagrams**, a similar approaches lead to bound $\boldsymbol{coNP^{\#P}}$: non-deterministically choose state $|v\rangle$ and verify that two scalar diagrams obtained by closing input diagrams with $|v\rangle$ represent the same scalar. Since the property needs to hold universally for answer $True$, the bound has $\boldsymbol{coNP}$ class rather $\boldsymbol{NP}$ in the base.

4.3 Oracle change in $\boldsymbol{NP^{\#P}}$

The main trick used to prove $\boldsymbol{NP^{\#P}}$-hardness of **StateEq** and **ContainsEntry** is captured by the following theorem:

Theorem 4.3.1. $\boldsymbol{NP^{\#P}} = \boldsymbol{NP^{C_=P[1]}}$.

We dedicate the rest of this section to the proof of the above theorem. The idea is that instead of calling $\boldsymbol{\#P}$ oracle, an NDTM can non-deterministically choose the answer. Then, at the end of the computation, it can verify all answer choices with a single call to the $\boldsymbol{C_=P}$ oracle. The same idea was used, for instance, in [134], though for a different class.

For $\boldsymbol{C_=P}$ oracle, we use the following problem:

Compare#SAT

Input: Two boolean formulae defined on n variables each:

- ϕ on variables $X := x_1, \ldots, x_n$,
- ψ on variables $Y := y_1, \ldots, y_n$.

Output: *True* if and only if $\textbf{\#SAT}(\phi, X) = \textbf{\#SAT}(\psi, Y)$.

This problem is $\boldsymbol{C_=P}$-complete. While the proof follows immediately from the definitions of $\boldsymbol{C_=P}$ and #$\boldsymbol{P}$, we present it below, as we could not find any publications with the proof (or this problem definition).

Theorem 4.3.2. **Compare#SAT** is $\boldsymbol{C_=P}$-complete.

Proof. **Containment:**
Define an NDTM $\mathcal{M}$ that given an input (ϕ, ψ), where the formulae are on n variables each, runs 2^{n+1} paths. $\mathcal{M}$ uses 2^n paths to evaluate all possible assignments of ϕ and returns $True$ precisely for satisfying assignments of ϕ. Similarly, on the other 2^n paths, $\mathcal{M}$ evaluates and returns $True$ precisely for unsatisfying assignments of ψ. Thus, in total $\mathcal{M}$ accepts input on $\textbf{\#SAT}(\phi) + \textbf{\#SAT}(\neg\psi) = \textbf{\#SAT}(\phi) + 2^n - \textbf{\#SAT}(\psi)$ paths and rejects on $2^n - \textbf{\#SAT}(\phi) + \textbf{\#SAT}(\psi)$ paths. The two numbers equal precisely when $\textbf{\#SAT}(\phi) = \textbf{\#SAT}(\psi)$, as required.
Hardness:
Given a problem $A \in \boldsymbol{C_=P}$, let $\mathcal{M}$ be the corresponding NDTM. For any instance w for A, we use the Cook-Levin method [49] to construct a Boolean formula ϕ corresponding to running $\mathcal{M}$ on w. This construction is parsimonious (or at least it can be made parsimonious based on [9]), thus $\textbf{\#SAT}(\phi) = \textsc{AccPaths}(\mathcal{M}, w)$. Similarly, we construct a Boolean formula ψ corresponding to running $\mathcal{M}'$ on w, where $\mathcal{M}'$ is $\mathcal{M}$ with flipped answer (i.e. returning 0 in place of 1 and 1 in place of 0). Then, again by parsimony of the Cook-Levin method, $\textbf{\#SAT}(\psi) = \textsc{RejPaths}(\mathcal{M}, w)$. We extend both formulae to ϕ', ψ' that operate on the same number of variables, without changing the number of satisfying assignments (for instance, by appending a clause $(z_1 \wedge z_2 \wedge \cdots \wedge z_\ell)$ where $z_1, \ldots, z_\ell$ are newly introduced variables). Finally, the instance w of A is equivalent to (ψ', ϕ') instance of **Compare#SAT**, as $\textbf{\#SAT}(\phi') = \textbf{\#SAT}(\psi') \Leftrightarrow \textsc{AccPaths}(\mathcal{M}, w) = \textsc{RejPaths}(\mathcal{M}, w)$. □

We need a few additional lemmata to prove Theorem 4.3.1.

Lemma 4.3.3 (Concat formulae)**.** Given two boolean formulae ϕ and ψ, it is possible to deterministically construct a boolean formula ρ, such that:

- ρ can be constructed in time polynomial in the sizes of ϕ and ψ,
- **#SAT**(ρ) written in binary is a concatenation of **#SAT**(ϕ) and **#SAT**(ψ) written in binary, possibly with some additional leading zeros.

Proof. Let $x_1, \dots, x_n$ be variables of ϕ and $y_1, \dots, y_m$ be variables of ψ. By substituting variables in one formula with new ones, we can assume that $x_i \neq y_j$ for all i, j. We define ρ as a boolean formula on $x_1, \dots, x_n, y_1, \dots, y_m, z, z'$ as follows:

$$\rho = (\phi \wedge z) \vee (x_1 \wedge \cdots \wedge x_n \wedge \psi \wedge \neg z \wedge z')$$

Let:

$$\begin{aligned} X &= x_1, \dots, x_n \\ Y &= y_1, \dots, y_m \\ Z &= z, z' \\ A &= X, Y, Z \end{aligned}$$

Then, we have:

$$\begin{aligned} \mathbf{\#SAT}(\rho, A) &= \mathbf{\#SAT}(\phi \wedge z, A) + \mathbf{\#SAT}(x_1 \wedge \cdots \wedge x_n \wedge \psi \wedge \neg z \wedge z', A) \\ &= \mathbf{\#SAT}(\phi, X) \cdot 2^{m+1} + \mathbf{\#SAT}(\psi, Y) \end{aligned}$$

The first equality holds as ρ is a conjunction of two formulae that cannot be simultaneously satisfied due to the requirement of different valuations of z. The second equality follows as $\phi \wedge z$ is independent of valuations of Y, z'. Thus, the last $m + 1$ digits of **#SAT**(ρ, A) in binary represent **#SAT**(ψ, Y) possibly with leading zeros, and the other first digits of **#SAT**(ρ, A) represent **#SAT**(ϕ, X). □

Lemma 4.3.4 (Concat **Compare#SAT** instances)**.** Let (ϕ, ψ) and (ζ, ξ) be two instances for problem **Compare#SAT**. They can be combined to a single instance for which the answer is $True$ if and only if the answers are $True$ for the two given instances.

Proof. Let ρ be a formula constructed from ϕ and ζ in Lemma 4.3.3 and τ be a formula constructed from ψ and ξ. Then the instance (ρ, τ) can be constructed in polytime and by construction: $\textbf{\#SAT}(\rho) = \textbf{\#SAT}(\phi) \cdot 2^{m+1} + \textbf{\#SAT}(\zeta)$ and $\textbf{\#SAT}(\tau) = \textbf{\#SAT}(\psi) \cdot 2^{m+1} + \textbf{\#SAT}(\xi)$, where m is the number of variables of ζ (and, thus, also of ξ). Since $\textbf{\#SAT}(\zeta)$ and $\textbf{\#SAT}(\xi)$ are both bounded above by 2^m, we have:

$$\textbf{\#SAT}(\rho) = \textbf{\#SAT}(\tau) \Leftrightarrow (\textbf{\#SAT}(\xi) = \textbf{\#SAT}(\zeta)) \wedge (\textbf{\#SAT}(\phi) = \textbf{\#SAT}(\psi))$$

as required. □

Another necessary construction is creating a Boolean formula with a given number of satisfying assignments.

Lemma 4.3.5. Let $x_1, \ldots, x_n$ be variables and k a number in $[0..2^n]$. Then, it is possible to construct a boolean formula ψ on $x_1, \ldots, x_n$ with $\textbf{\#SAT}(\psi, (x_1, \ldots, x_n)) = k$.

Proof. Let $a_0, \ldots, a_n$ be the digits of k written in binary, possibly with leading zeros. Thus, we have $k = \sum_{i \in [0..n]} 2^{n-i} \cdot a_i = \overline{(a_0 \ldots a_n)}_2$. Now, consider the following formula on $x_0, x_1, \ldots, x_n$:

$$LT_{a_0,\ldots,a_n}(x_0, \ldots, x_n) := (x_0 < \ell(a_0)) \vee ((x_0 = \ell(a_0)) \wedge LT_{a_1,\ldots,a_n}(x_1, \ldots, x_n))$$

with the base case $LT(x_n)_{a_n} = (x_n < \ell(a_n))$; where $\ell(0) = False$ and $\ell(1) = True$. If we treat variables as digits 0 or 1, then $LT_{a_0,\ldots,a_n}(x_0, \ldots, x_n)$ encodes the predicate $\overline{(x_0 \ldots x_n)}_2 < \overline{(a_0 \ldots a_n)}_2$. Therefore, we get $\textbf{\#SAT}(LT_{a_0,\ldots,a_n}(x_0, \ldots, x_n)) = \overline{(a_0 \ldots a_n)}_2 = k$. The connectives $<$ and $=$ are realised in predicate logic via $\rightarrow$ and $\leftrightarrow$ respectively. Thus, the above formula almost works, except that it is defined on $n+1$ variables rather than n variables.
We can alleviate the need for the x_0 variable as follows. Consider the following boolean formula:

$$\psi(x_1, \ldots, x_n) := \ell(a_0) \vee LT_{a_1,\ldots,x_n}(x_1, \ldots, x_n)$$

for $n \geq 1$ and $\psi := \ell(a_0)$ for $n = 0$. When $n = 0$, clearly $\textbf{\#SAT}(\psi) = a_0 = k$. Thus, assume $n \geq 1$ from now. When $a_0 = 1$, then $k \geq 2^n$, and since $k \in [0..2^n]$, necessarily $k = 2^n$. At the same time, when $a_0 = 1$, ψ simplifies to $True$, and as a formula on $x_1, \ldots, x_n$, it has 2^n satisfying assignments, as required. Finally, when $a_0 = 0$, then ψ simplifies to $LT_{a_1,\ldots,x_n}(x_1, \ldots, x_n)$ and has $\overline{(a_1 \ldots a_n)}_2 = \overline{(a_0 a_1 \ldots a_n)}_2 = k$ satisfying assignments, as required. □

Finally, we can prove Theorem 4.3.1.

Proof of Theorem 4.3.1. We start by proving the right containment $\boldsymbol{NP}^{\#\boldsymbol{P}} \subseteq \boldsymbol{NP}^{\boldsymbol{C_=P}[1]}$.

#SAT is complete for class #$\boldsymbol{P}$, thus $\boldsymbol{NP}^{\#\boldsymbol{P}} = \boldsymbol{NP}^{\#\mathbf{SAT}}$. Let A be a problem in $\boldsymbol{NP}^{\#\mathbf{SAT}}$. It suffices to show that $A \in \boldsymbol{NP}^{\boldsymbol{C_=P}[1]}$. Let $\mathcal{M}$ be a polytime NDTM with access to the **#SAT** oracle that recognizes A. Define NDTM $\mathcal{M}'$ as follows. $\mathcal{M}'$ contains one extra tape over $\mathcal{M}$ and for all inputs w, the transition function of $\mathcal{M}'$ matches that of $\mathcal{M}$ (ignoring the extra tape of $\mathcal{M}'$) for all configurations except for the three states: the oracle ask q_{ASK}, the accepting state q_{ACC} and the rejecting state q_{REJ}. Further, $\mathcal{M}'$ has access **Compare#SAT** oracle rather than **#SAT** oracle.

When $\mathcal{M}$ enters the oracle query state q_{ASK} and sends $\phi, (x_1, \ldots, x_n)$ to the **#SAT** oracle, then $\mathcal{M}'$ instead non-deterministically chooses an answer $k \in [0..2^n]$ to the oracle query. Next, it creates a Boolean formula ψ on $x_1, \ldots, x_n$ such that $\mathbf{\#SAT}(\psi) = k$ by using Lemma 4.3.5. Then, $\mathcal{M}'$ writes the **Compare#SAT** instance (ϕ, ψ) on its extra tape. Finally, $\mathcal{M}$ moves to the configuration that $\mathcal{M}$ would be in if it received the answer k from the oracle (except for the extra tape).

When $\mathcal{M}$ enters the accepting state q_{ACC}, then $\mathcal{M}'$ instead constructs two boolean formulae (Ψ, Φ) such that $\mathbf{\#SAT}(\Psi) = \mathbf{\#SAT}(\Phi)$ if and only if $\mathbf{\#SAT}(\phi) = \mathbf{\#SAT}(\psi)$ for all pairs (ϕ, ψ) stored in the extra tape of $\mathcal{M}'$. Finally, $\mathcal{M}'$ sends (Φ, Ψ) to its oracle and returns the oracle's answer. The construction of (Φ, Ψ) is always possible via a series of concatenations presented in Lemma 4.3.4.

When $\mathcal{M}$ enters the rejecting state q_{REJ}, $\mathcal{M}'$ calls its oracle with a $False$ instance and returns its answer.

The NDTM $\mathcal{M}'$ runs in a polytime since $\mathcal{M}$ does, and all extra operations of $\mathcal{M}'$ can be performed in time polynomial in the input size. Further, $\mathcal{M}$ recognizes the same language. To see this, consider $w \in A = \mathrm{L}(\mathcal{M})$. Then, $\mathcal{M}$ accepts w in one of its non-deterministic paths p. Then, in one of its non-deterministic paths, $\mathcal{M}'$ picks the correct answer to the first oracle query that $\mathcal{M}$ asks for on the path p. In that path, $\mathcal{M}'$ can further non-deterministically choose a path that also picks the correct answer to the second oracle query $\mathcal{M}$ asks for on the path p, then the third, etc. In the end, $\mathcal{M}$ enters q_{ACC}, so $\mathcal{M}'$ constructs a single instance of **Compare#SAT** that is used to verify that all non-deterministically chosen answers to the $\mathcal{M}$ oracle queries were correct, so, in the end, $\mathcal{M}'$ accepts w. When $\mathcal{M}$ does not accept w, then on all its paths, $\mathcal{M}$ eventually enters the rejecting state. Thus $\mathcal{M}'$ either ends with an unsatisfiable instance for the query (when $\mathcal{M}'$ enters equivalent of q_{REJ} state of $\mathcal{M}$), or it had to choose an incorrect answer for at least one of the queries, which is captured in the end with the **Compare#SAT** oracle.

The left containment $\boldsymbol{NP}^{\boldsymbol{\#P}} \supseteq \boldsymbol{NP}^{\boldsymbol{C_=P}[1]}$ is immediate. Instead of sending **Compare#SAT** query (ϕ, ψ), a Turing Machine can ask for **#SAT**(ϕ) and **#SAT**(ψ), and then compare the answers to effectively simulate **Compare#SAT** oracle with two calls to the **#SAT** oracle. $\square$

We could not find a shorter proof of the above fact. However, it is worth mentioning how a shorter proof could proceed. By corollary of Toda's theorem [165]: $\boldsymbol{P}^{\boldsymbol{\#P}} = \boldsymbol{P}^{\boldsymbol{PP}}$ and hence $\boldsymbol{NP}^{\boldsymbol{\#P}} = \boldsymbol{NP}^{\boldsymbol{PP}}$. Then it may be possible to adjust Torán's work [166, Corollary 3.13 and Theorem 4.1] to obtain $\boldsymbol{NP}^{\boldsymbol{PP}} = \boldsymbol{NP}^{\boldsymbol{C_=P}}$. Finally, $\boldsymbol{NP}^{\boldsymbol{C_=P}} = \boldsymbol{NP}^{\boldsymbol{C_=P}[1]}$ by some clever adjustment of Lemma 4.3.4. We would skip the Turing Machines here. However, they are necessary in the remaining parts of the proof anyway.

By the above theorem, we can show $\boldsymbol{NP}^{\boldsymbol{\#P}}$-hardness instead of $\boldsymbol{NP}^{\boldsymbol{C_=P}[1]}$-hardness.

4.4 Crafting $\boldsymbol{NP}^{\boldsymbol{\#P}}$-complete problem

We want to show $\boldsymbol{NP}^{\boldsymbol{\#P}}$-hardness. By Theorem 4.3.1, it is equivalent to showing $\boldsymbol{NP}^{\boldsymbol{C_=P}[1]}$-hardness. We craft an artificial $\boldsymbol{NP}^{\boldsymbol{C_=P}[1]}$-complete problem **∃Compare#SAT** to use in a reduction.

∃Compare#SAT

Input: Natural numbers n, m and two boolean formulae:

- ψ, defined on $x_1, \ldots, x_n, y_1, \ldots, y_m$,
- ρ, defined on $x_1, \ldots, x_n, z_1, \ldots, z_m$

Output: $True$ if and only if there exists a valuation $v\colon \{x_1, \ldots, x_n\} \to \{True, False\}$ such that:

$$\textbf{\#SAT}(v(\psi), y_1, \ldots, y_m) = \textbf{\#SAT}(v(\rho), z_1, \ldots, z_m)$$

In the problem definition, $v(\psi)$ and $v(\rho)$ are formulae on variables $y_1, \ldots, y_m$ and $z_1, \ldots, z_m$ respectively.

While the definition of the problem is lengthy, this problem is, in some sense, the most natural problem for $\boldsymbol{NP}^{\boldsymbol{C_=P}[1]}$: It combines **SAT** and **Compare#SAT**, the natural problems for $\boldsymbol{NP}$ and $\boldsymbol{C_{=}P}$.

The other existing complete problems for $\boldsymbol{NP^{\#P}}$ [166, 134] are based on checking which answer to two **#SAT** instances is greater or how to maximize an answer rather than asking for exact equality. Unlike for ∃**Compare#SAT**, we have not found a way of translating those problems to ZH Calculus directly.

Example 4.4.1. Let $n = 1, m = 2$ and:

- $\psi = x_1 \lor y_1 \lor \neg y_2$,
- $\rho = \neg(z_1 \land (z_2 \lor x_1))$.

Then, under valuation v mapping x_1 to *False*, we have:

$$v(\psi) = \mathit{False} \lor y_1 \lor \neg y_2 = y_1 \lor \neg y_2$$
$$v(\rho) = \neg(z_1 \land (z_2 \lor \mathit{False})) = \neg(z_1 \land z_2) = \neg z_1 \lor \neg z_2$$

Since:

$$\textbf{\#SAT}(v(\psi)) = \textbf{\#SAT}(y_1 \lor \neg y_2) = 3 = \textbf{\#SAT}(\neg z_1 \lor \neg z_2) = \textbf{\#SAT}(v(\rho))$$

the answer to such an instance of ∃**Compare#SAT** is *True*. For valuation v' mapping x_1 to *True*, the two values do not equalise:

$$v'(\psi) = \mathit{True} \lor y_1 \lor \neg y_2 = \mathit{True}$$
$$v'(\rho) = \neg(z_1 \land (z_2 \lor \mathit{True})) = \neg(z_1 \land \mathit{True}) = \neg z_1$$

and:

$$\begin{aligned}\textbf{\#SAT}(v'(\phi), y_1, y_2) &= \textbf{\#SAT}(\mathit{True}, y_1, y_2) = 4 \\ &\neq 2 = \textbf{\#SAT}((\neg z_1), z_1, z_2) = \textbf{\#SAT}(v'(\rho), z_1, z_2).\end{aligned}$$

The rest of this section shows the $\boldsymbol{NP^{C_{=}P[1]}}$-completeness of the above problem. The proof involves typical yet tedious transformations of Turing machines.

We put a plethora of constraints on TMs for problems in $\boldsymbol{NP^{C_{=}P[1]}}$. One of these constraints describes how a TM should represent a Boolean formula used in the query. We want TMs to work solely with 0 and 1 symbols and pad any possible formula that it could reach as an oracle query to a formula with a fixed length in its binary representation. This fixed length is supposed to depend on the size of the TM's input, but not on the exact content of the input. The precise representation of the Boolean formula in binary that we introduce here is used only to facilitate such padding.

Definition 4.4.2 (CNF)**.** A boolean formula on $x_1, \dots, x_n$ is in *conjunctive normal form* (*CNF*) if it is a conjunction of clauses, where each clause is a disjunction of some of the literals $x_1, \neg x_1, \dots, x_n, \neg x_n$.

Definition 4.4.3 (0 – 1 encoding of CNFs)**.** Let $\phi = c_1 \wedge c_2 \wedge \cdots \wedge c_m$ be a boolean formula in CNF on variables $x_1, \dots, x_n$. We define $0 - 1$ encoding of ϕ as a word e_ϕ, defined as follows, where $k = \max(m, n)$:

- Change the set of allowed variables by adding $k - n$ variables $x_{n+1}, \dots, x_k$,
- Add $k - n$ clauses: $x_{n+1}, \dots, x_k$, each with a single literal,
- Extend ϕ with $2k - (m + n)$ clauses containing literals x_1 and $\neg x_1$,
- Translate each clause c_j of ϕ to $2k$ symbols from $\{0, 1\}$, where symbols at the positions $2i - 1$ and $2i$ correspond to variable x_i for $i \in [1..k]$ as follows:
  - The first symbol is 0 when c_j does not contain literal x_i and 1 otherwise,
  - The second symbol is 0 when c_j does not contain literal $\neg x_i$ and 1 otherwise.

Example 4.4.4. Consider the CNF formula $x_1 \wedge (x_2 \vee \neg x_3)$ with $m = 2$ clauses and $n = 3$ variables. Let $k = \max(m, n) = 3$. Following construction from Definition 4.4.3, the formula is extended to a formula with $2k = 6$ clauses and $k = 3$ variables:

$$x_1 \wedge (x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_1) \wedge (x_1 \vee \neg x_1) \wedge (x_1 \vee \neg x_1) \wedge (x_1 \vee \neg x_1)$$

Then, the word $e_{x_1 \wedge (x_2 \vee \neg x_3)}$ is constructed by translating the extended formula (spaces are added for visibility only):

$$10\,00\,00 \quad 00\,10\,01 \quad 11\,00\,00 \quad 11\,00\,00 \quad 11\,00\,00 \quad 11\,00\,00$$

with blocks of six symbols corresponding to each of the six clauses.

Definition 4.4.5. Let w be any word over $\{0, 1\}$ with $4k^2$ symbols for some k. We define formula $\textsc{Formula}(w)$ as follows:

- $\textsc{Formula}(w)$ has $2k$ clauses and is defined on k variables $x_1, \ldots, x_k$,
- Each clause corresponds to a block of $2k$ symbols, where the next 2 symbols correspond to literals describing variable x_i, same as in the definition of $0-1$ encoding:
  - If the first symbol is 1, add literal x_i, otherwise add literal *False*,
  - If the second symbol is 1, add literal $\neg x_i$, otherwise add literal *False*.

Example 4.4.6. Consider the previous word:

10 00 00 00 10 01 11 00 00 11 00 00 11 00 00 11 00 00

It corresponds to the formula below, where 0 is used as shorthand for *False*:

$$(x_1 \vee 0 \vee 0 \vee 0 \vee 0 \vee 0) \wedge (x_2 \vee 0 \vee 0 \vee 0 \vee 0 \vee \neg x_3) \wedge (x_1 \vee \neg x_1 \vee 0 \vee 0 \vee 0 \vee 0)$$
$$\wedge (x_1 \vee \neg x_1 \vee 0 \vee 0 \vee 0 \vee 0) \wedge (x_1 \vee \neg x_1 \vee 0 \vee 0 \vee 0 \vee 0) \wedge (x_1 \vee \neg x_1 \vee 0 \vee 0 \vee 0 \vee 0)$$

By removing *False* literals from clauses, we get:

$$x_1 \wedge (x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_1) \wedge (x_1 \vee \neg x_1) \wedge (x_1 \vee \neg x_1) \wedge (x_1 \vee \neg x_1)$$

By removing the last four clauses, which are satisfied by any assignment, we get the original formula:

$$x_1 \wedge (x_2 \vee \neg x_3)$$

Theorem 4.4.7. Let Φ be a boolean formula in CNF with m clauses, and let $x_1, \ldots, x_n$ be variables on which it is considered. Then:

$$\#\mathbf{SAT}(\Phi, (x_1, \ldots, x_n)) = \#\mathbf{SAT}(\textsc{Formula}(e_\Phi), (x_1, \ldots, x_{\max(n,m)}))$$

Proof. Let $k = \max(m, n)$. Consider steps in the construction of e_Φ in Definition 4.4.3:

- Extension of Φ by the clauses containing literals x_1 and $\neg x_1$ does not change satisfiable assignments – any assignment satisfies such clauses,

- Extension of set of variables to $x_1, \ldots, x_k$ while adding clauses containing each of these new variables as only literal does not change number of satisfiable assignments – the only satisfiable assignments after the extension are those with $x_{n+1}, \ldots, x_k$ set to $True$ and $x_1, \ldots, x_n$ set to satisfiable assignment of the original Φ.

Let Ψ be the formula Φ after these steps, therefore:

$$\textbf{\#SAT}(\Phi, (x_1, \ldots, x_n)) = \textbf{\#SAT}(\Psi, (x_1, \ldots, x_k))$$

The encoding of literals preserves the satisfiable assignments when the number of clauses is twice the number of variables. Therefore: $\textsc{Formula}(e_\Psi) = \Psi$. By construction $e_\Psi = e_\Phi$ and thus the thesis follows:

$$\begin{aligned}\textbf{\#SAT}(\textsc{Formula}(e_\Phi), (x_1, \ldots, x_k)) &= \textbf{\#SAT}(\textsc{Formula}(e_\Psi), (x_1, \ldots, x_k))\\ &= \textbf{\#SAT}(\Psi, (x_1, \ldots, x_k))\\ &= \textbf{\#SAT}(\Phi, (x_1, \ldots, x_n))\end{aligned}$$

□

We can now constrain TMs for problems in $\boldsymbol{NP}^{\boldsymbol{C_=P}[1]}$.

Theorem 4.4.8. Let $A \in \boldsymbol{NP}^{\boldsymbol{C_=P}[1]}$. Then, there exists a polytime NDTM $\mathcal{M}$ with access to **Compare#SAT** oracle, that recognized A and satisfies the following conditions:

1. $\mathcal{M}$ calls the oracle A precisely once, at the very end of the computation, and $\mathcal{M}$ returns the oracle answer,
2. The boolean formulae that $\mathcal{M}$ sends to the oracle are all in CNF,
3. $\mathcal{M}$ has two dedicated tapes used for communication with the oracle. When $\mathcal{M}$ enters its q_{ASK} state to ask an **Compare#SAT** query (ζ, ξ), then on the first tape it contains a representation of the first boolean formula ζ as a sequence of zeros and ones e_ζ, and on the second tape it contains a representation e_ξ of ξ,
4. There exists a polynomial p such that for any input w, when $\mathcal{M}$ calls the

oracle with (Φ, Ψ) query then $|e_\Phi| = |e_\Psi| = p(|w|)$. In other words, the size of the query depends only on the size of the input, not on the input itself.

5. There exist a polynomial q such that for any input w, $\mathcal{M}$ runs in precisely $q(|w|)$ steps.

Proof. By Theorem 4.3.1, there exists polytime NDTM $\mathcal{M}_0$ with access to **Compare#SAT** oracle satisfying the first condition.
Since every Boolean formula can be transformed into CNF in polynomial time, we can assume that $\mathcal{M}_0$ satisfies the second condition.
The third condition formally describes how the machine should communicate with the oracle. By Theorem 4.4.7, the encoding of the formula does not alter the number of satisfying assignments, and thus, we can assume that $\mathcal{M}_0$ further satisfies the third condition.
The fourth condition is more involved. Let $p_{\mathcal{M}_0}$ be the polynomial bounding number of steps of $\mathcal{M}_0$, i.e. for input w, $\mathcal{M}_0$ terminates in at most $p_{\mathcal{M}_0}(|w|)$ steps. Define $\mathcal{M}_1$ as a TM that follows $\mathcal{M}_0$, however, when $\mathcal{M}_0$ enters q_{ASK} state, $\mathcal{M}_1$ instead changes the contents of oracle tapes: the boolean formulae are transformed to have encoding sizes $4p_{\mathcal{M}_0}(|w|)^2$ each – via the same procedure as in the construction of the encoding in Definition 4.4.3, we add clauses and variables to have $p_{\mathcal{M}_0}(|w|)$ of variables and $2p_{\mathcal{M}_0}(|w|)$ clauses.
Therefore, $\mathcal{M}_1$ satisfies the first three conditions, as $\mathcal{M}_0$ does, and it satisfies the fourth condition with $p = 4p^2_{\mathcal{M}_0}$. Further, $\mathcal{M}_1$ is polytime, as $\mathcal{M}_0$ is, and extra steps of $\mathcal{M}_1$ take at most $O(p)$ steps. Let $p_{\mathcal{M}_1}$ be the polynomial bounding number of steps of $\mathcal{M}_1$, i.e. for input w, $\mathcal{M}_1$ terminates in at most $p_{\mathcal{M}_1}(|w|)$ steps.
Finally, we define $\mathcal{M}$ as a TM that follows $\mathcal{M}_1$. However, if $\mathcal{M}_1$ enters q_{ASK} state, then $\mathcal{M}$ stalls for some number of steps so that it always calls the oracle in the $q(|w|)$ step, where q is some polynomial. This can be achieved, for example, by equipping $\mathcal{M}$ with an extra 'counter' tape on which it initially fills $p_{\mathcal{M}_1}(|w|)$ cells with 1. Then, $\mathcal{M}$ starts computation, following $\mathcal{M}_1$ and removing 1 cell from the counter during each step. When the oracle should be called, $\mathcal{M}$ first continues to clear the counter and only then calls the oracle. The value $p_{\mathcal{M}_1}(|w|)$ can be computed based on the input size but not the input itself. Thus, the creation and clearance of the counter requires several steps, depending on the length of w, but not on the contents of w. Therefore, the total number of steps on input w is indeed given by some $q(|w|)$. □

We can now prove that the crafted problem is $\boldsymbol{NP}^{\boldsymbol{C_=P}[1]}$-complete.

Theorem 4.4.9. **∃Compare#SAT** is $\boldsymbol{NP}^{\boldsymbol{C_=P}[1]}$-complete.

Proof.
Containment: given an instance (n, m, ψ, ρ) of **∃Compare#SAT**, a NDTM with **Compare#SAT** oracle can do the following:

- Non-deterministically choose a valuation of $x_1, \dots, x_n$,
- Call the oracle on $(v(\psi), v(\rho))$ and returns its answer.

Hardness: Let $A \in \boldsymbol{NP}^{\boldsymbol{C_=P}[1]}$. We must show that problem A can be reduced to **∃Compare#SAT**. Let w be any instance of A. Let $\mathcal{M}, p, q$ be a TM and polynomials guaranteed to exist by Theorem 4.4.8.
Let $\Phi_{\mathcal{M},w}$ be a Boolean formula constructed by Cook-Levin reduction that describes whether M reaches q_{ASK}, and let $X := x_1, x_2, \dots, x_n$ be the variables from the Cook-Levin construction. These include variables of the form $y_{t,i,s,k}$ corresponding to statement "on k^{th} step of the computation the i^{th} cell of t^{th} tape contains symbol s" and similar variables describing the state of TM in the given step of computation and the positions of heads.
Let $o1$ and $o2$ be the two oracle tapes used to communicate with the oracle, i.e. $o1$ stores the first Boolean formula as a sequence, and $o2$ stores the second Boolean formula, both as sequences of zeros and ones. By construction, $w \in A$ if and only if the answer to the final (and only) oracle call is *True*. This oracle call is entirely encoded on the two tapes and can thus be expressed using the same variables as $\Phi_{\mathcal{M},w}$.
Let v be a valuation of the variables X satisfying $\Phi_{\mathcal{M},w}$, so a description of the path taken to reach q_{ASK}. Let $\tilde{v} \colon X \longrightarrow \{0, 1\}$ with $\tilde{v}(x_i) = 1$ when $v(x_i) = \mathit{True}$ and $\tilde{v}(x_i) = 0$ when $v(x_i) = \mathit{False}$. The first stored Boolean formula is:

$$\zeta_{\mathcal{M},w,v} = \textsc{Formula}(\tilde{v}(y_{o1,1,1,q(|w|)-1})\tilde{v}(y_{o1,2,1,q(|w|)-1}) \dots \tilde{v}(y_{o1,p(|w|),1,q(|w|)-1}))$$

By theorem 4.4.8, only the above variables matter – the oracle call is always precisely on positions 1 to $p(|w|)$ of $o1$ tape in the $q(|w|) - 1^{\text{th}}$ step of computation (as the TM in total has $q(|w|)$ steps where the last step is the oracle call). By theorem 4.4.7, the only two possible symbols are 0 and 1, and $\tilde{v}(x_{o1,i,1,q(|w|)-1})$ is 1 when the corresponding symbol is 1, and 0 when the corresponding symbol is 0. We have $p(|w|) = 4k^2$ for some $k \in \mathbb{Z}$, as the $(0-1)$ representation of CNF formulae must be of such length. Let $u_1, \ldots, u_k$ be the variables of $\zeta_{\mathcal{M},w,v}$. Let y'_l be a shorthand for $y_{o1,l,1,q(|w|)-1}$. By construction of Formula, the value $\tilde{v}(y'_{2k(i-1)+(2j-1)})$ for $i \in [1..2k]$ and $j \in [1..k]$ describes whether in the i^{th} clause we put literal u_j or *False*. Thus, such literal is equal to the expression $u_j \wedge y'_{2k(i-1)+(2j-1)}$ under the valuation v. Similarly, the value $\tilde{v}(y'_{2k(i-1)+2j})$ describes whether in the i^{th} clause we put literal $\neg u_j$ or $False$. Thus, such literal is equal to the expression $\neg u_j \wedge y'_{2k(i-1)+2j}$ under the valuation v. Now, define $\Psi_{\mathcal{M},w}$ as follows:

$$\Psi_{\mathcal{M},w} := \psi_1 \wedge \cdots \wedge \psi_{2k}, \quad \text{where:}$$
$$\psi_i := ((u_1 \wedge y'_{2k(i-1)+1}) \vee (\neg u_1 \wedge y'_{2k(i-1)+2}) \vee \cdots \vee (u_k \wedge y'_{2ki-1}) \vee (\neg u_k \wedge y'_{2ki}))$$

Then, $v(\psi_i)$ is equal to the i^{th} clause of $\zeta_{\mathcal{M},w,v}$, and hence $v(\Psi_{\mathcal{M},w})$ is equal $\zeta_{\mathcal{M},w,v}$. Similarly, the second stored formula is:

$$\xi_{\mathcal{M},w,v} = \textsc{Formula}(\tilde{v}(x_{o2,1,1,q(|w|)-1})\tilde{v}(x_{o2,2,1,q(|w|)-1}) \ldots \tilde{v}(x_{o2,p(|w|),1,q(|w|)-1}))$$

Let $u'_1, \ldots, u'_k$ be the variables of it. Via a similar procedure, we can produce $\mathrm{P}_{\mathcal{M},w}$ such that $v(\mathrm{P}_{\mathcal{M},w}) = \xi_{\mathcal{M},w,v}$.
By construction, $w \in A$ if and only if there exists a valuation v, such that $\Phi_{\mathcal{M},w}$ is satisfied and **#SAT**$(\zeta_{\mathcal{M},w,v})$ = **#SAT**$(\xi_{\mathcal{M},w,v})$. Equivalently, $v(\Phi_{\mathcal{M},w}) = \mathit{True}$ and **#SAT**$(v(\Psi_{\mathcal{M},w}))$ = **#SAT**$(v(\mathrm{P}_{\mathcal{M},w}))$.
We can now define an equivalent instance of ∃**Compare#SAT**:

- Numbers $n, k+1$, where n is the number of variables of $\Phi_{\mathcal{M},w}$ and k is the number of variables of $v(\Phi_{\mathcal{M},w})$ and $v(\mathrm{P}_{\mathcal{M},w})$,
- Boolean formula Ψ defined as $\Phi_{\mathcal{M},w} \wedge (\Psi_{\mathcal{M},w} \vee y)$ on $x_1, \ldots, x_n, u_1, \ldots, u_k, y$,
- Boolean formula P defined as $\mathrm{P}_{\mathcal{M},w} \vee y'$ on $x_1, \ldots, x_n, u'_1 \ldots, u'_k, y'$.

Then **#SAT**$(v(\Psi))$ = **#SAT**$(v(\mathrm{P}))$ under valuation v of $x_1, \ldots, x_n$ happens if and only if the following hold, where $U = u_1, \ldots, u_k$ and $U' = u'_1, \ldots, u'_k$:

$$\textbf{\#SAT}(v(\Phi_{\mathcal{M},w} \wedge (\Psi_{\mathcal{M},w} \vee y)), U, y) = \textbf{\#SAT}(v(\mathrm{P}_{\mathcal{M},w} \vee y'), U', y')$$

Since $v(\Phi_{\mathcal{M},w})$ is either *True* or *False*, the number **#SAT**$(v(\Phi_{\mathcal{M},w}), \emptyset)$ equals 0 or 1. y appears in the formula once, used in disjunction with $\Psi_{\mathcal{M},w}$. Hence, we get the following:

$$\begin{aligned}
&\textbf{\#SAT}(v(\Phi_{\mathcal{M},w} \wedge (\Psi_{\mathcal{M},w} \vee y)), U, y) \\
&\qquad = \textbf{\#SAT}(v(\Phi_{\mathcal{M},w}), \emptyset) \cdot \textbf{\#SAT}(v(\Psi_{\mathcal{M},w} \vee y), U, y) \\
&\qquad = \textbf{\#SAT}(v(\Phi_{\mathcal{M},w}), \emptyset) \cdot (\textbf{\#SAT}(v(\Psi_{\mathcal{M},w}), U) + 2^k) \\
&\qquad = \begin{cases} 0, & \text{when } \Phi_{\mathcal{M},w} \text{ is not satisfied under } v \\ \textbf{\#SAT}(v(\Psi_{\mathcal{M},w}, U)) + 2^k, & \text{when } \Phi_{\mathcal{M},w} \text{ is satisfied under } v \end{cases}
\end{aligned}$$

Similarly:

$$\textbf{\#SAT}(v(\mathrm{P}_{\mathcal{M},w} \vee y')) = \textbf{\#SAT}(v(\Psi_{\mathcal{M},w}, U)) + 2^k$$

Since $2^k > 0$, the two expressions equalize precisely when $\Phi_{\mathcal{M},w}$ is satisfied under v and $\textbf{\#SAT}(v(\Psi_{\mathcal{M},w}, U)) = \textbf{\#SAT}(v(\mathrm{P}_{\mathcal{M},w}, U))$. By the above, this is equivalent to $\mathcal{M}$ accepting w. The above $\exists$**Compare#SAT** instance is constructible in polytime in $|w|$, which ends the reduction and the entire proof. □

Now that we have crafted the $\boldsymbol{NP^{C_=P[1]}}$-complete problem, which is also $\boldsymbol{NP^{\#P}}$-complete, we finally have a problem to reduce from to finish the main proof.

4.5 ZH encoding

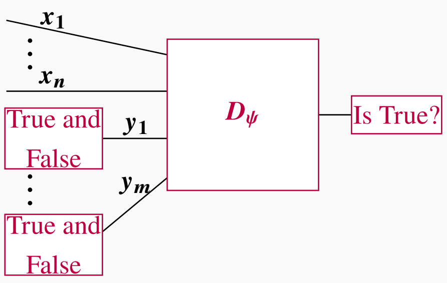
The goal is to show that **StateEq** and **ContainsEntry** are both $\boldsymbol{NP^{C_=P[1]}}$-hard. It suffices to reduce from $\exists$**Compare#SAT**. We already know how to represent both a boolean formula and the number of satisfying assignments of a boolean formula in phase-free ZH. Hence, the ZH representation is almost immediate.

Theorem 4.5.1. StateEq is $\boldsymbol{NP^{C_=P[1]}}$-hard.

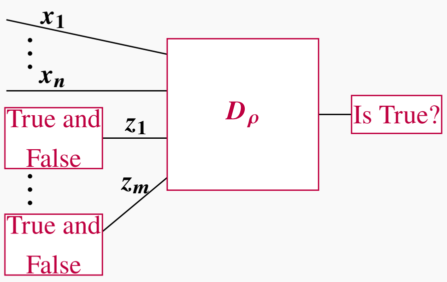
Proof. We perform a reduction from $\exists$**Compare#SAT**. Let (m, n, ψ, ρ) be any $\exists$**Compare#SAT** instance, where the common variables are $x_1, \ldots, x_n$ and extra variables are $y_1, \ldots, y_m$ for ψ and $z_1, \ldots, z_m$ for ρ. Now, define diagrams D_1, D_2

as follows:

$D_1 :=$



$D_2 :=$



where D_ψ and D_ρ are phase-free ZH encodings of formulae ψ and ρ respectively. The variables are not part of the diagram and are written only to help indicate the logical expressions corresponding to the wires.

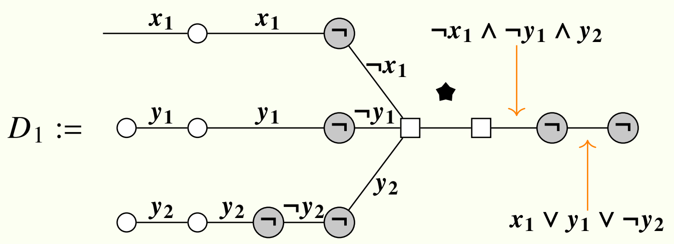
Let $|v\rangle = |v_1 \dots v_n\rangle$ be any state from the n qubits computational basis. Then, v gives a valuation of $x_1, \dots, x_n$, by taking $v(x_i) = \mathit{False}$ when $v_i = 0$ and $v(x_i) = \mathit{True}$ when $v_i = 1$. Let $Y = y_1, \dots, y_m$. We get:

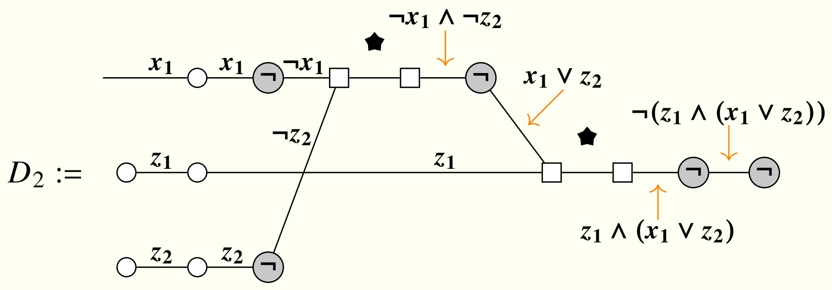
$$\begin{aligned}
[\![D_1]\!]\,|v\rangle &= \langle 1|\,[\![D_\psi]\!]\,(|v\rangle \otimes (\sqrt{2}^m\,|+\rangle^{\otimes m}))\\
&= \sum_{v'\colon Y\to\{0,1\}} \langle 1|\,[\![D_\psi]\!]\,(|v\rangle \otimes (|v'\rangle)\\
&= \sum_{v'\colon Y\to\{0,1\}} \langle 1|\,|(v'(v(\psi))\rangle\\
&= \langle 1|\,(\#\mathbf{SAT}(v(\psi),Y)\,|1\rangle + (2^m - \#\mathbf{SAT}(v(\psi),Y))\,|0\rangle)\\
&= \#\mathbf{SAT}(v(\psi),Y)
\end{aligned}$$

where $|v'\rangle = |v'(y_1)\dots v'(y_m)\rangle$. The first equality comes from the definition of 'Is True?' block as $\langle 1|$, the second equality from expanding $|+\cdots+\rangle$ as the sum of computational basis states, the third equality from combining valuations v of $x_1, \dots, x_n$ and v' of $y_1, \dots, y_m$ to a single valuation of $x_1, \dots, x_n, y_1, \dots, y_m$, and the fourth equality follows from Lemma 3.2.9.

Similarly, $[\![D_2]\!]\,|v\rangle = \#\mathbf{SAT}(v(\rho), z_1, \dots, z_m)$. Thus, the diagrams equalize on state $|v\rangle$ precisely when $\#\mathbf{SAT}(v(\psi), y_1, \dots, y_m) = \#\mathbf{SAT}(v(\rho), z_1, \dots, z_m)$, which ends the reduction and the entire proof. □

Example 4.5.2. Consider ∃**Compare#SAT** instance from Example 4.4.1. The two diagrams D_1 and D_2 constructed in the proof of Theorem 4.5.1 are presented below. All blocks are translated to the ZH generators. The orange lines are not part of the diagram and only indicate logical expressions corresponding to the wires.





$D_2 :=$
$\neg x_1 \wedge \neg z_2$
x_1
x_1
$\neg x_1$
$x_1 \vee z_2$
$\neg(z_1 \wedge (x_1 \vee z_2))$
$\neg z_2$
z_1
z_1
z_2
z_2
$z_1 \wedge (x_1 \vee z_2)$

The matrix representations of the diagrams correspond to the values found for different assignments of x_1 in Example 4.4.1:

$$[\![D_1]\!] = \begin{pmatrix} 3 & 4 \end{pmatrix}, \qquad [\![D_2]\!] = \begin{pmatrix} 3 & 2 \end{pmatrix}$$

and they indeed agree on the first entry.

The above constructions require only the blocks representing *True* and *False* simultaneously, COPY, NOT, AND gates, and Is True? check, i.e. most of the constructions in Example 3.2.6. It means, that **StateEq** problem is $NP^{\#P}$-hard not only over phase-free ZH but also over any language in which these blocks can be represented like ZX, ZW, etc. In particular, the version of **StateEq** problem where the input consists of two tensors constructed from the mentioned blocks is also $NP^{\#P}$-complete.

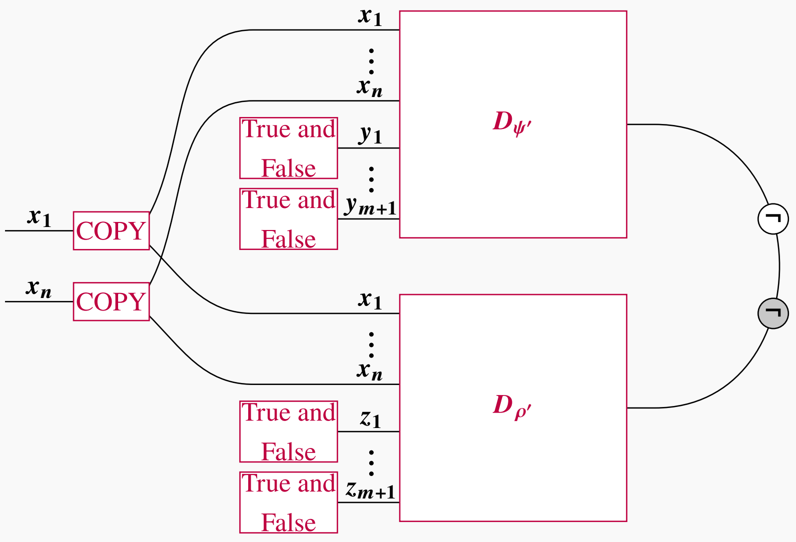
Corollary 4.5.3. The problem **ContainsEntry** is $NP^{\#P}$-hard for all generator sets in which the following tensors can be achieved:

$$\begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}1&0\\0&0\\0&0\\0&1\end{pmatrix}, \begin{pmatrix}0&1\\1&0\end{pmatrix}, \begin{pmatrix}1&1&1&0\\0&0&0&1\end{pmatrix}, \text{ and } \begin{pmatrix}0&1\end{pmatrix}.$$

All entries in these matrices are 0 or 1, so any tensor constructed from these has a matrix representation with all entries being natural numbers. This is even stricter than phase-free ZH diagrams, where entries can be dyadic rationals.

Theorem 4.5.4. ContainsEntry is $NP^{C_=P[1]}$-hard.

Proof. We reduce from **∃Compare#SAT**. First, we focus on the cases $k \in \{0, 1\}$. Let (m, n, ψ, ρ) be any instance. The goal is to construct a diagram that when evaluated on a state corresponding to valuation v of $x_1, \ldots, x_n$ results in the number $\textbf{\#SAT}(v(\rho), y_1, \ldots, y_m) - \textbf{\#SAT}(v(\psi), z_1, \ldots, z_m) + k$. I.e. the resulting number equals k precisely when v is a witness for the initial **∃Compare#SAT** instance.
In the previous proof, we showed how to get the numbers $\textbf{\#SAT}(v(\rho))$ and $\textbf{\#SAT}(v(\psi))$ in phase-free ZH. The difference can be achieved by utilizing white and dark NOTs. The addition of k can be achieved by modifying the pair of initial boolean formulae. Consider the following diagram:



Where ψ' and ρ' are defined as follows:

$$\psi' = (\psi \wedge \neg y_{m+1})$$
$$\rho' = \begin{cases} (\rho \wedge \neg z_{m+1}), & \text{when } k = 0 \\ (\rho \wedge \neg z_{m+1}) \vee (z_1 \wedge \cdots \wedge z_m \wedge z_{m+1}), & \text{when } k = 1 \end{cases}$$

Again, let $|v\rangle = |v_1 \dots v_n\rangle$ be any state from the n qubits computational basis. Then, v gives a valuation of $x_1, \dots, x_n$, by taking $v(x_i) = \mathit{False}$ when $v_i = 0$ and $v(x_i) = \mathit{True}$ when $v_i = 1$. Let $Y' = y_1, \dots, y_m, y_{m+1}$ and $Z' = z_1, \dots, z_m, z_{m+1}$.
By considering requirements on variables y_{m+1} and z_{m+1}, we obtain:

$$\begin{aligned}\#\mathbf{SAT}(v(\psi'), y_1, \dots, y_{m+1}) &= \#\mathbf{SAT}(v(\psi), y_1, \dots, y_m) \\ \#\mathbf{SAT}(v(\rho'), z_1, \dots, z_{m+1}) &= \left\{\begin{matrix} \#\mathbf{SAT}(v(\rho), z_1, \dots, z_m), & \text{when } k = 0 \\ \#\mathbf{SAT}(v(\rho), z_1, \dots, z_m) + 1, & \text{when } k = 1 \end{matrix}\right\} \\ &= \#\mathbf{SAT}(v(\rho), z_1, \dots, z_m) + k\end{aligned}$$

Hence:

$$\begin{aligned}&\#\mathbf{SAT}(v(\rho'), z_1, \dots, z_{m+1}) - \#\mathbf{SAT}(v(\psi'), y_1, \dots, y_{m+1}) \\ &\qquad = \#\mathbf{SAT}(v(\rho), z_1, \dots, z_m) - \#\mathbf{SAT}(v(\psi), y_1, \dots, y_m) + k\end{aligned}$$

The rightmost part of the diagram D (the two nots on a cap) has matrix representation $\langle 01| - \langle 10|$. It is the only part of the proof where white not is used.
Similarly to the previous proof, we get:

$$\begin{aligned}[\![D]\!]\,|v\rangle &= \frac{1}{2^{m+1}}(\langle 01| - \langle 10|)([\![D_{\psi'}]\!] \otimes [\![D_{\rho'}]\!])(|v\rangle \otimes \sqrt{2}^{m+1}\,|+\rangle^{\otimes(m+1)})^{\otimes 2} \\ &= \frac{1}{2^{m+1}}(\langle 01| - \langle 10|)(a_1\,|1\rangle + a_0\,|0\rangle) \otimes (b_1\,|1\rangle + b_0\,|0\rangle)\end{aligned}$$

where:

$$\begin{aligned}a_1 &= (\#\mathbf{SAT}(v(\phi'), Y') \\ a_0 &= (2^{m+1} - \#\mathbf{SAT}(v(\phi'), Y')) \\ b_1 &= (\#\mathbf{SAT}(v(\rho'), Z') \\ b_0 &= (2^{m+1} - \#\mathbf{SAT}(v(\rho'), Z')).\end{aligned}$$

Therefore:

$$\begin{aligned}
[\![D]\!] |v\rangle &= \frac{1}{2^{m+1}}(a_0 b_1 - a_1 b_0) \\
&= \frac{1}{2^{m+1}}((2^{m+1} - a_1)b_1 - a_1(2^{m+1} - b_1)) \\
&= \frac{1}{2^{m+1}}(2^{m+1}b_1 - a_1 b_1 - 2^{m+1}a_1 + a_1 b_1) \\
&= b_1 - a_1 \\
&= (\#\mathbf{SAT}(v(\rho'), Z') - (\#\mathbf{SAT}(v(\phi'), Y') \\
&= \#\mathbf{SAT}(v(\rho), z_1, \ldots, z_m) - \#\mathbf{SAT}(v(\psi), y_1, \ldots, y_m) + k
\end{aligned}$$

The last number equals k precisely when $\#\mathbf{SAT}(v(\psi), y_1, \ldots, y_m) = \#\mathbf{SAT}(v(\rho), z_1, \ldots, z_m)$. Therefore instance of **ContainsEntry** consisting of diagram D is equivalent to the initial instance (m, n, ψ, ρ) of $\exists$**Compare#SAT**.
This ends the proof for $k \in \{0, 1\}$. For any other dyadic rational $k = \frac{c}{2^d}$, we can modify diagram D for $k = 1$ to also contain a scalar representing the new k in phase-free ZH, effectively rescaling the diagram by a factor of k. For instance, this scalar can be obtained by adding d star generators and evaluating the formula with c satisfying assignments on all of its assignments. Such a formula can be constructed by Lemma 4.3.5. Hence, the problem **ContainsEntry** is $\boldsymbol{NP^{\#P}}$-hard for all dyadic rationals k. □

When k is not a dyadic rational, then k cannot appear in the matrix representation of any phase-free ZH diagram. Then, **ContainsEntry** problem is trivial – any instance can be immediately answered with *False*.

In the above proof, we had to obtain a difference of two numbers in phase-free ZH and rescale such difference by a negative power of two. We used white not and star generators. It extends the necessary tensors over those for **StateEq** by tensors with matrix representation:

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad \text{and} \quad \frac{1}{2}.$$

The second of these is not necessary when $k = 0$. Thus, the problem $\mathbf{ContainsEntry}_0$ is $\boldsymbol{NP^{\#P}}$-hard even for tensors with matrix representations containing only integers.

Combining everything, we obtain the proof of the main result from this chapter:

Theorem 4.1.1 (Repeated)**. StateEq** and **ContainsEntry** are both $\boldsymbol{NP^{\#P}}$-complete.

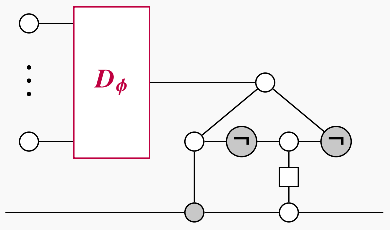
Proof. Completeness of **StateEq** follows from Theorems 4.2.1, 4.5.1, and 4.3.1; and completeness of **ContainsEntry** follows from Theorems 4.2.2, 4.5.4, and 4.3.1. □

4.6 Circuit extraction

As an additional result, we present how to adapt proof form [62] for phase-free ZH, i.e. we show that circuit extraction remains #$\boldsymbol{P}$-hard for phase-free ZH diagrams. The original proof required a $\frac{\pi}{2}$ phase spider that cannot be constructed in phase-free ZH.

Theorem 4.1.2 (Repeated)**.** **CircuitExtraction** is #$\boldsymbol{P}$-hard.

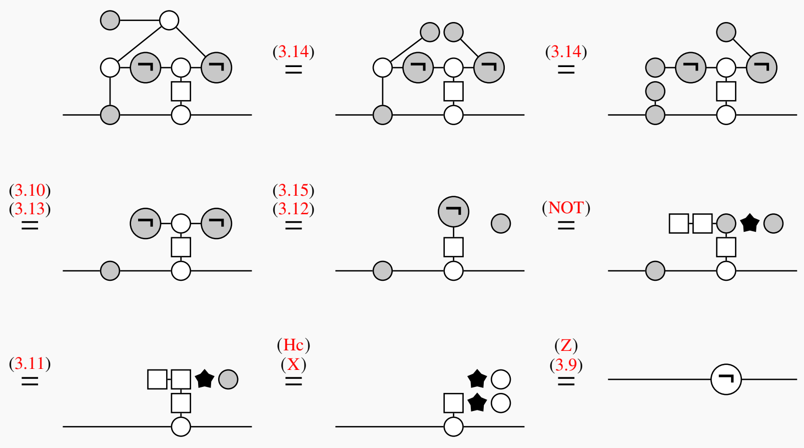
Proof. We reduce from **#SAT**. Let ϕ be any boolean formula and $x_1, \ldots, x_n$ be the variables on which it is considered. Consider the following diagram:

$D =$ 

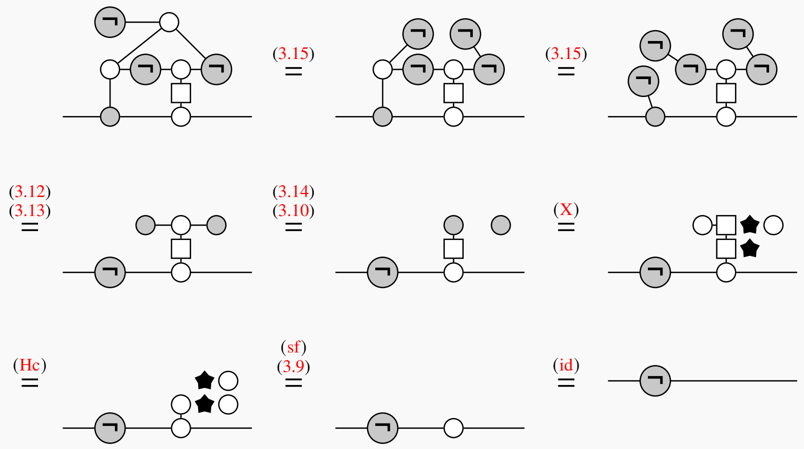
By Lemma 3.2.9, $[\![D_\phi]\!]\,\sqrt{2}^n\,|+\rangle^{\otimes n} = a_0\,|0\rangle + a_1\,|1\rangle$, where:

$$a_1 = \textbf{\#SAT}(\phi)$$
$$a_0 = 2^n - \textbf{\#SAT}(\phi)$$

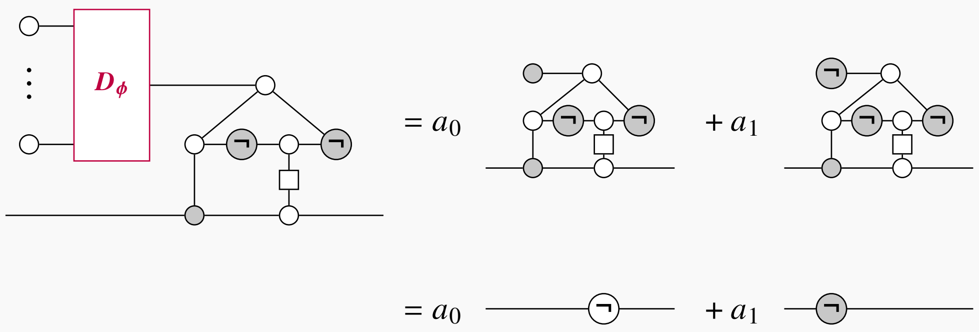
The construction to the right of D_ϕ works as follows: if in place of D_ϕ, we put $|0\rangle$, then the diagram simplifies to a Z gate. The rewrites are labelled by rules and lemmata from [14]:



On the other hand, if we put $|1\rangle$, then the diagram simplifies to an X gate:



Thus, we obtain the following:



By considering the matrix representation of the remaining part of the diagram, we find:

$$[\![D]\!] = \begin{bmatrix} a_0 & a_1 \\ a_1 & -a_0 \end{bmatrix} =: M$$

which is proportional to a unitary:

$$MM^\dagger = \begin{bmatrix} a_0^2 + a_1^2 & 0 \\ 0 & a_0^2 + a_1^2 \end{bmatrix}$$

The finish of the proof is the same as in [62]: Given a polynomial-size ancilla-free circuit equivalent to the presented diagram, we can compute the matrix representation of the circuit in polynomial time. The obtained matrix representation is proportional to the matrix M above. Hence, it is possible to deduce **#SAT**(ϕ). Therefore **#SAT** $\in$ $\boldsymbol{FP}^{\textbf{CircuitExtraction}}$, which means precisely that **CircuitExtraction** is #$\boldsymbol{P}$-hard. □

Our construction is a bit more complex than the one from the proof for ZX Calculus – rather than applying the controlled iX gate, we apply the variant of the controlled iY gate that can be achieved in phase-free ZH. It has the advantage of working in every graphical calculus in which one can represent $[\![D_\phi]\!]$, Hadamard gate, and equivalents of π phase spiders from ZX. Similarly to [62], the hardness of different variants of circuit extraction follows from the same construction as above.

4.7 Limitations

We showed $\boldsymbol{NP}^{\#\boldsymbol{P}}$-completeness of **StateEq** and **ContainsEntry**. The next question is whether similar results can be obtained for other problems. **CircuitExtraction** was shown to be #$\boldsymbol{P}$-hard and the decision variant is in $\boldsymbol{NP}^{\boldsymbol{NP}^{\#\boldsymbol{P}}}$, leaving a lot of room for improvement. This section briefly outlines why the results cannot be easily extended to problems such as **CompareDiagrams**, or improve the hardness of **CircuitExtraction**.

4.7.1 Complexity of comparing diagrams

As I alluded to at page 66, **CompareDiagrams** asks for a universal property of matching matrix interpretations, while **StateEq** asks for the existential property of matching matrix interpretations. Now, consider the following problem, a modification of ∃**Compare#SAT**, where the existential property is switched to a universal one:

∀Compare#SAT

Input: Natural numbers n, m and two boolean formulae:

- ψ, defined on $x_1, \dots, x_n, y_1, \dots, y_m$,
- ρ, defined on $x_1, \dots, x_n, z_1, \dots, z_m$

Output: $True$ if and only if *for all* valuations $v\colon \{x_1, \dots, x_n\} \to \{True, False\}$:

$$\textbf{\#SAT}(v(\psi), y_1, \dots, y_m) = \textbf{\#SAT}(v(\rho), z_1, \dots, z_m)$$

By ∀ in the problem name, we mean correspondence to the tautology problem, i.e. checking whether a given formula is satisfied by all assignments. The construction as in the proof of Theorem 4.5.1 exhibits reduction of **∀Compare#SAT** to **CompareDiagrams**:

Observation 4.7.1. **∀Compare#SAT** reduces to **CompareDiagrams**.

The required construction does not just resemble that from proof of Theorem 4.5.1: It is *exactly* the same construction.

Earlier on, we mentioned that **CompareDiagrams** $\in \boldsymbol{coNP}^{\boldsymbol{\#P}}$. To obtain completeness, we would need **∀Compare#SAT** to be $\boldsymbol{coNP}^{\boldsymbol{\#P}}$-hard. I did not manage to prove this fact, and I expect doing so to be quite difficult.

At first, it might look as **∀Compare#SAT** is the dual of **∃Compare#SAT**. However, it is not the case. The problem dual to **∃Compare#SAT** takes the same input as **∃Compare#SAT** and **∀Compare#SAT**, but instead asks whether for all valuations $v\colon \{x_1, \dots, x_n\} \to \{True, False\}$:

$$\textbf{\#SAT}(v(\psi), y_1, \dots, y_m) \neq \textbf{\#SAT}(v(\rho), z_1, \dots, z_m)$$

It is important to point the inequality sign instead of equality. This variant with inequality could be shown to be $\boldsymbol{coNP}^{\boldsymbol{\#P}}$-complete. However, it does not reduce to **CompareDiagrams** via a simple construction.

Thus, the question is how hard is **∀Compare#SAT**, and **CompareDiagrams** by consequence. Again, I do not have the answer. When $n = 0$, then the problem **∀Compare#SAT** collapses to $\boldsymbol{C_{=}P}$, so **CompareDiagrams** is at least $\boldsymbol{C_{=}P}$-hard. In the proof of **∃Compare#SAT** hardness (Theorem 4.4.9), I adapted Cook-Levin method. This approach does not lead to anything in case of **∀Compare#SAT**. The key part of Cook-Levin construction for $\boldsymbol{NP}$ machines is that while it can be made parsimonious, the number of unsatisfying assignments of the formula may vastly outnumber rejecting paths of the corresponding Turing machine. This is not an issue when adapting construction to $\boldsymbol{NP}$ with an oracle. However, the approach fails for $\boldsymbol{coNP}$ with an oracle.

I expect that **CompareDiagrams** is $\boldsymbol{coNP}^{\#\boldsymbol{P}}$-complete, and thus it could be strictly harder than comparing circuits: determining whether circuit is almost equivalent to identity is $\boldsymbol{QMA}$-complete [96] (Quantum Merlin-Arthur [178]), while an exact non-identity check is $\boldsymbol{NQP}$-complete [164] (Non-deterministic Quantum Polynomial-time [4]). Both these classes are within $\boldsymbol{PP}$, and thus within $\boldsymbol{P}^{\#\boldsymbol{P}}$. This way, graphical calculi like phase-free ZH can be understood as more expressive than quantum circuits.

4.7.2 Improving circuit extraction bounds

Improving the hardness of circuit extraction faces the same challenges as explained in [62]. In the hardness proof, we only used single-qubit circuits arising from diagrams with one input and output edge. Following [62], circuits on logarithmically many qubits (in the size of diagrams) would also work. Given circuit of such form we can recover answer to encoded **#SAT** instance by finding (or approximating) explicit matrix representation of the circuit. In circuits with a higher number of qubits, this approach no longer works, as the matrix representation becomes exponential in the size of the input diagram. Thus, it is no longer possible to recover the answer to the encoded problem in polynomial time. At the same time, circuit extraction for diagrams with one input and one output edge is in $\boldsymbol{FP}^{\#\boldsymbol{P}}$: by closing the dangling edges with each element of the computational basis and using #$\boldsymbol{P}$ oracle, we could find exact matrix representation of given diagram, and then construct equivalent circuit (if one exists) by following methods showcasing approximate universality of various gate sets. For that reason, improving the result of #$\boldsymbol{P}$-hardness of **CircuitExtraction** can again be challenging.

The current upper bound of **CircuitExtraction** uses **CompareDiagrams** under the hood: given a diagram, non-deterministically choose a circuit, encode the circuit as a diagram and verify with (multiple calls to) **CompareDiagrams** oracle that the input diagram and the diagram representing the chosen circuit represent proportional linear processes (see [62] for details). Therefore, any better bound of **CompareDiagrams** would also imply a better **CircuitExtraction** bound.

As a final connection, the problems **CompareDiagrams** and **CircuitExtraction** are related to the minimal circuit size problem **MCSP**. Classically, given a boolean function, the problem asks for the smallest circuit implementing it. It is known that classical version of the problem is in $\boldsymbol{NP}$, but it remains open whether the problem is $\boldsymbol{NP}$-complete and it forms a good candidate for an $\boldsymbol{NP}$-intermediate problem, i.e. a problem beyond $\boldsymbol{P}$ but not $\boldsymbol{NP}$-complete. The quantum variants of such problem has also been studied [41, 40]. The problem relates to circuit optimisation, where if we know that one circuit implements input function, then the problem is to find minimal equivalent circuit. Graphical calculi provide a framework for circuit optimisation and thus the study of **MCSP**. A description of a function (or unitary which is a more natural input for quantum versions of **MCSP**) can be interpreted as a diagram D_1 and then the quest is to find a smallest other diagram D_2 which agrees with given one (i.e. forming satisfying input (D_1, D_2) of **CompareDiagrams**) such that the diagram corresponds to circuit (meaning that **CircuitExtraction** on such D_2 is a trivial). In some cases, circuit optimisation via graphical reasoning is optimal [172], suggesting that Z-like graphical calculi are ideal for heuristic approach to cases of quantum **MCSP**. On the other hand, some bounds from [41, 40] fall into $\boldsymbol{QCMA} \subseteq \boldsymbol{QMA}$ and thus showcase that induced circuit optimisation is of similar hardness as comparing circuits. Therefore it is interesting to investigate whether tighter bounds to **CircuitExtraction** and **CompareDiagrams** can be obtained by considering results for quantum **MCSP** when adapted to ask about minimal diagram instead of minimal circuit.

In the following chapters, I work on flow structures. As explained in Subsection 3.3.2, these structures aid in solving the **CircuitExtraction** problem for practical purposes. In light of the hardness results in this chapter, techniques for circuit extraction such as in [153] are the best we can hope for.

Chapter 5

Algebraic Interpretation of Flow

In the previous chapters, we saw that **CircuitExtraction** is a crucial obstacle when working with graphical calculi for quantum computation. We saw that this problem is #***P***-hard for any language of tensor networks which allows representation of #**SAT** instances. For instance, this problem is #***P***-hard for phase-free ZH diagrams, which form far more restrictive calculus than ZX Calculus.

On the other hand, circuit extraction can be performed in deterministic polynomial time when the diagram corresponds to a labelled open graph with Pauli flow. However, Pauli flow is difficult to work with due to exceptionally dense definitions that hinder flow applications.

This chapter simplifies these definitions by providing a new algebraic interpretation of Pauli flow. This involves defining two matrices arising from the adjacency matrix of the underlying graph: the flow-demand matrix M and the order-demand matrix N. We show that Pauli flow exists if and only if there is a right inverse C of M such that the product NC forms the adjacency matrix of a directed acyclic graph.

Since our results work for Pauli flow, they also apply to gflow and XY-only gflow. Furthermore, we show multiple examples of how proofs can be carried out with the new interpretation, extending previous results about flow reversibility and understanding focused set structure. Finally, this interpretation provides groundwork for algorithms for Pauli flow, which will be the focus of Chapter 6.

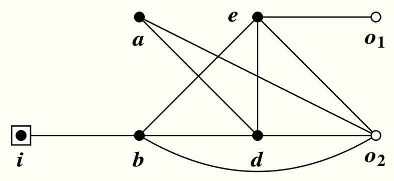
Corresponding papers. The novel results in this chapter have appeared in my papers: [132] (single author) and [133] (first author; co-written with my supervisor Miriam Backens).

Structure: In Section 5.1, we provide preliminary notions regarding flow, expanding on what we saw in Section 2.6. Next, in Section 5.2, we define matrices appearing in algebraic interpretation and state the corresponding theorem. The proof of the theorem and most essential properties of the defined matrices follow in Section 5.3. Later, in Section 5.4, we look at the flow reversibility result. Next, in Section 5.5, we explore the structure of focused sets. Finally, in Section 5.6, we provide one final non-algorithmic result arising from algebraic interpretation about modifications to the number of inputs and outputs in the labelled open graph.

5.1 Preliminary notions

We start by slightly extending the standard definitions of Pauli flow to suit our needs. We use the following running examples to aid the explanation here and throughout the chapter. The labelled open graph is based on [153, Example D.18], with the measurement of vertex b changed from XY to Y.

Example 5.1.1. An open graph with one input $\{i\}$ and two outputs $\{o_1, o_2\}$:

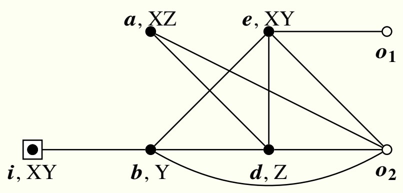


The neighbourhoods and measurement labelling are also presented in the table:

v	$\lambda(v)$	Neighbourhood of v
i	XY	b
a	XZ	d, o_2
b	Y	i, e, d, o_2
e	XY	b, d, o_1, o_2
d	Z	a, b, c, o_2
o_1		e
o_2		a, b, e, d

Example 5.1.2. The following labelled open graph with underlying open graph

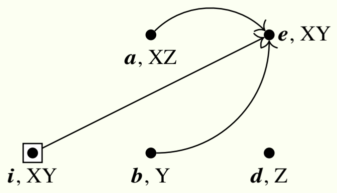
from Example 5.1.1:



has Pauli flow $(c, \prec)$, where c is given in the following table:

	λ	$c(v)$	$\mathrm{Odd}(c(v))$
i	XY	b, e, o_1	i, b, o_1
a	XZ	a, e, o_1, o_2	a, d, o_1
b	Y	e	b, d, o_1, o_2
e	XY	o_1	e
d	Z	d, o_2	d, o_2

and the partial order is given as a directed acyclic graph:



i.e. the order is: $a, i, b \prec e$. This flow is focused.

Firstly, we redefine the correction function to any function of the required type, not just a part of Pauli flow:

Definition 5.1.3 (Correction function)**.** Let (G, I, O, λ) be a labelled open graph. A *correction function* for (G, I, O, λ) is a function $c\colon \bar{O} \to \mathcal{P}(\bar{I})$.

The focusing conditions, as defined in Definition 2.6.22, only refer to the function part of the flow, thus, it makes sense to define focused correction function:

Definition 5.1.4 (Focused correction function)**.** Let (G, I, O, λ) be a labelled open graph and c a correction function. We say that c is *focused* when it satisfies (F1), (F2), and (F3).

For the partial order, we introduce minimal order as the coarsest relation that can appear in a flow:

Definition 5.1.5 (Minimal order)**.** Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph and let $(c, \prec)$ be a Pauli flow on Γ. We call $\prec$ *minimal* when no proper subset of $\prec$ forms a Pauli flow with c.

Next, we define extensive correction function as a correction function that appears in some Pauli flow:

Definition 5.1.6 (Extensive correction function)**.** Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph and let c be a correction function on Γ. We call c *extensive* when there exists $\prec$ such that $(c, \prec)$ is a Pauli flow on Γ. We call c *focused extensive* when there exists $\prec$ such that $(c, \prec)$ is a focused Pauli flow on Γ.

I am unaware of the above definitions for Pauli flow appearing in the literature (outside my joint work with Miriam Backens [133]). However, an analogous version for the case of XY-only gflow was considered in [125].

The point of these definitions is to motivate the canonical relation induced by the correction function:

Definition 5.1.7 (Induced relation)**.** Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph and let c be a correction function on Γ. The *induced relation* $\lessdot_c$ is the minimal relation on $\bar{O}$ implied by (P1), (P2), (P3) from Definition 2.6.16. That is, $u \lessdot_c v$ if and only if at least one of the following holds:

- $v \in c(u) \wedge u \neq v \wedge \lambda(v) \notin \{X, Y\}$ (corresponding to (P1)),
- $v \in \mathrm{Odd}(c(u)) \wedge u \neq v \wedge \lambda(v) \notin \{Y, Z\}$ (corresponding to (P2)),
- $v \in \mathrm{Odd}⟦c(u)⟧ \wedge u \neq v \wedge \lambda(v) = Y$ (corresponding to (P3)).

We denote the transitive closure of $\lessdot_c$ as $\prec_c$.

In the following, we prove some useful properties of $\lessdot_c$: Firstly, the transitive closure $\prec_c$ of the relation $\lessdot_c$ induced by the correction function c is minimal. Further, if c forms Pauli flow with any partial order, it also does so with $\prec_c$.

Lemma 5.1.8 (Minimal order containment)**.** Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph and let $(c, \prec)$ be a Pauli flow. Then $\prec_c \subseteq \prec$.

Proof. By construction, $\triangleleft_c$ is the minimal relation implied by (P1), (P2), (P3). Thus, any relation satisfying these conditions must contain $\triangleleft_c$, thus $\triangleleft_c \subseteq \prec$. Further, $\prec$ is a strict partial order, and hence it is transitive. Thus, it also contains the transitive closure of $\triangleleft_c$, i.e. $\prec_c$, ending the proof. □

Theorem 5.1.9 (Extending a correction function to a Pauli flow)**.** Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph and let c be a correction function on Γ. Then c is (focused) extensive if and only if $(c, \prec_c)$ is a minimal (focused) Pauli flow.

Proof. ($\Rightarrow$): Suppose that c is (focused) extensive. Then there exists $\prec$ on $\bar{O}$ such that $(c, \prec)$ is a (focused) Pauli flow. Thus, $(c, \prec_c)$ satisfies all of (P4)-(P9) from Definition 2.6.16 (and (F1)-(F3)), as those conditions only ask about c. Further, $\prec_c$ contains $\triangleleft_c$, and so $(c, \prec_c)$ satisfies (P1)-(P3). Finally, $\prec$ is a strict partial order; hence $\prec_c$ is also a strict partial order, as it is a transitive subset of $\prec$ by 5.1.8. Therefore $(c, \prec_c)$ is a (focused) Pauli flow. It is minimal by Lemma 5.1.8.
($\Leftarrow$): Immediate from the definitions. □

The above theorem tells us that, to verify whether a correction function c is extensive, we only need to test one order induced by c. In other words, we only need to search for an extensive correction function rather than searching for a correction function and a partial order forming a flow together. The induced order is particularly well-behaved in the case of the focused correction function. Then, all Pauli-measured vertices are initial in the partial order, as explained in the following lemma.

Lemma 5.1.10 (All Paulis are initial)**.** Let (G, I, O, λ) be a labelled open graph and c be a focused correction function. Then $\forall u \in \Lambda_{Pauli}. \forall v \in \bar{O}. \neg v \prec_c u$.

Proof. Implied by [153, Lemma B.11]. □

5.2 Matrix construction

As may be seen from the previous section, Pauli flow can be hard to work with, as the relevant definitions are long and complicated. For instance, to check if a given pair

$(c, \prec)$ forms a focused Pauli flow, one needs to verify the twelve conditions (P1)-(P9) and (F1)-(F3) for each vertex, which is a tedious procedure.

We now provide an alternative interpretation of Pauli flow: Given a labelled open graph Γ, we define two matrices constructed from the adjacency matrix – a 'flow-demand matrix' M_Γ (formally defined in Definition 5.2.7) and an 'order-demand matrix' N_Γ (formally defined in Definition 5.2.9). We prove that Γ has Pauli flow if and only if there exists a matrix C (encoding the correction function) such that $M_\Gamma C$ is the identity matrix and $N_\Gamma C$ forms a DAG:

Theorem 5.2.1. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph, c be a correction function, M_Γ be the flow-demand matrix of Γ, and N_Γ be the order-demand matrix of Γ. Then c is focused extensive if and only if the following two facts hold for the corresponding correction matrix C:

- $M_\Gamma C = Id_{\bar{O}}$,
- $N_\Gamma C$ is the adjacency matrix of a DAG.

This section defines various matrices ending with the flow-demand and order-demand matrices from the above theorem. The proof of the theorem follows in the subsequent section.

Notation. All matrices are over the field $\mathbb{F}_2$. We drop the subscripts specifying the ((labelled) open) graph when it is clear from context.
It will sometimes be helpful to refer to individual rows or columns of a matrix, we will use the notation $M_{v,*}$ to denote the v-th row of M and the notation $C_{*,u}$ to denote the u-th column of C.

We start with the standard definition of the adjacency matrix.

Definition 5.2.2 (Adjacency matrix)**.** Let $G = (V, E)$ be a simple graph. Its *adjacency matrix* is the $n \times n$ matrix Adj_G with rows and columns corresponding to elements of v, where:

$$(Adj_G)_{u,v} = \begin{cases} 1 & \text{if } uv \in E \\ 0 & \text{otherwise} \end{cases}$$

A particularly important submatrix of the adjacency matrix of an open graph is the so-called 'reduced adjacency matrix'.

Definition 5.2.3 (Reduced adjacency matrix [125, Definition 6] and [132, Definition 3.11]). Let (G, I, O) be an open graph. Its *reduced adjacency matrix* is the $(n - n_O) \times (n - n_I)$ submatrix $Adj_G\ |_{\bar{O}}^{\bar{I}}$ of the adjacency matrix Adj_G, keeping only the rows corresponding to the non-outputs $\bar{O}$ and the columns corresponding to the non-inputs $\bar{I}$.

The above matrix was previously used in characterising focused gflow on labelled open graphs with only *XY* planar measurements [125].

Instead of considering the correction function as a function, we can transform it into a matrix.

Definition 5.2.4 (Correction matrix). Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph and let c be a correction function on Γ. We define the *correction matrix* C encoding the function c as the $(n - n_I) \times (n - n_O)$ matrix with rows corresponding to non-inputs $\bar{I}$ and columns corresponding to non-outputs $\bar{O}$, where $C_{u,v} = 1$ if and only if $u \in c(v)$.

Observation 5.2.5. Given a correction matrix C encoding an unknown correction function c, the correction function can be recovered as $c(v) = \{u \in \bar{I} \mid C_{u,v} = 1\}$.

In other words, the v column in C encodes the characteristic function of set $c(v)$. That is $C_{*,v} = \mathbb{1}_{c(v)}^{\bar{I}}$ and $\operatorname{supp}\left(C_{*,v}\right) = c(v)$. Here, $\mathbb{1}_{\mathcal{A}}^{\mathcal{D}}$ stands for the indicator function of the subset $\mathcal{A}$ of $\mathcal{D}$ (as well as for the corresponding vector over $\mathbb{F}_2$), and supp stands for the support. See the following example:

Example 5.2.6. Consider the labelled open graph and it Pauli flow from Example 5.1.2. The correction matrix corresponding to the stated correction function is:

C	i	a	b	e	d
a	0	1	0	0	0
b	1	0	0	0	0
e	1	1	1	0	0
d	0	0	0	0	1
o_1	1	1	0	1	0
o_2	0	1	0	0	1

Based on Observation 5.2.5, the correction sets can be read from columns: for example, $c(a) = \{a, e, o_1, o_2\}$ as a column has 1 precisely at intersections with a, e, o_1, and o_2 rows.

In [125], it was shown that a labelled open graph with all-XY measurements has Pauli flow if and only if its reduced adjacency matrix is right-invertible, and one of the right inverses is the adjacency matrix of a DAG. In that case, the right inverse corresponds precisely to the correction matrix. I included generalisation of this algebraic result to also allow Pauli-X measurements in [132] (with part of the proof appearing in [133]).

In [132], I have used algebraic interpretation to deal with Pauli X and Z measurements by showing that a particular matrix called 'flow matrix' must be right-invertible [132]. I combined and indeed improved on those results in [133] (co-written with my supervisor Miriam Backens) by defining what we now call a 'flow-demand matrix', which works for all six measurement types.

Definition 5.2.7 (Flow-demand matrix)**.** Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph. We define the *flow-demand matrix* M_Γ as the $(n - n_O) \times (n - n_I)$ matrix with rows corresponding to non-outputs $\bar{O}$ and columns corresponding to non-inputs $\bar{I}$, where the row $M_{v,*}$ corresponding to the vertex $v \in \bar{O}$ satisfies the following for any $w \in \bar{I} \setminus \{v\}$:

- if $\lambda(v) \in \{X, XY\}$, then $M_{v,v} = 0$ and $M_{v,w} = Adj_{v,w}$ i.e. the v row encodes the neighbourhood of v,
- if $\lambda(v) \in \{Z, YZ, XZ\}$, then $M_{v,v} = 1$ and $M_{v,w} = 0$, i.e. the v row contains a 1 at the intersection with the v column and is identically 0 otherwise, and
- if $\lambda(v) \in \{Y\}$, then $M_{v,v} = 1$ (provided that v column exists) and $M_{v,w} = Adj_{v,w}$, i.e. the v row encodes the neighbourhood of v and also has a 1 at the intersection with the v column.

Example 5.2.8. The flow-demand matrix of labelled open graph in Example 5.1.2

is:

M	a	b	e	d	o_1	o_2
i	0	1	0	0	0	0
a	1	0	0	0	0	0
b	0	1	1	1	0	1
e	0	1	0	1	1	1
d	0	0	0	1	0	0

The correction matrix from Example 5.2.6 is the right inverse of this flow demand matrix, which verifies the condition from Theorem 5.2.1 for this particular labelled open graph.

As we will soon prove, the right invertibility of the flow-demand matrix relates to most, but not all, of the conditions of the focused Pauli flow. For the remaining conditions, we need the following construction, which we call the 'order-demand matrix'.

Definition 5.2.9 (Order-demand matrix). Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph. We define the *order-demand matrix* N_Γ as the $(n - n_O) \times (n - n_I)$ matrix with rows corresponding to non-outputs $\bar{O}$ and columns corresponding to non-inputs $\bar{I}$, where the row $N_{v,*}$ corresponding to the vertex $v \in \bar{O}$ satisfies the following for any $w \in \bar{I}$:

- if $\lambda(v) \in \{X, Y, Z\}$, then $N_{v,*} = \mathbf{0}$, i.e. the v row is identically 0,
- if $\lambda(v) = YZ$, then $N_{v,v} = 0$ and $N_{v,w} = Adj_{v,w}$, i.e. the v row encodes the neighbourhood of v,
- if $\lambda(v) = XZ$, then $N_{v,v} = 1$ and $N_{v,w} = Adj_{v,w}$, i.e. the v row encodes the neighbourhood of v and also has a 1 at the intersection with the v column, and
- if $\lambda(v) = XY$, then $N_{v,v} = 1$ (provided that the v column exists) and $N_{v,w} = 0$, i.e. the v row contains a 1 at the intersection with the v column and is identically 0 otherwise.

In order-demand matrices, the rows of XY measured vertices are encoded as rows of Z measured vertices would be in the flow-demand matrix. Similar correspondence holds between YZ and X measurements and between XZ and Y measurements. In the definitions, when setting the intersection of v row and v column to 1, we must exclude $v \in I$, as for such v, there exists a row but not a column. The rows of Pauli measurements and XY measured inputs are identically 0 in the order-demand matrix.

Example 5.2.10. The order-demand matrix of the labelled open graph from Example 5.1.2 is:

N	a	b	e	d	o_1	o_2
i	0	0	0	0	0	0
a	1	0	0	1	0	1
b	0	0	0	0	0	0
e	0	0	1	0	0	0
d	0	0	0	0	0	0

The product of this order-demand matrix with the correction matrix from Example 5.2.6 is:

NC	i	a	b	e	d
i	0	0	0	0	0
a	0	0	0	0	0
b	0	0	0	0	0
e	1	1	1	0	0
d	0	0	0	0	0

which corresponds precisely to the directed acyclic graph from Example 5.1.2, verifying the second condition from Theorem 5.2.1 for this particular labelled open graph.

5.3 Properties of the flow-related matrices

In this section, we prove useful properties of the newly defined flow-demand and order-demand matrices, culminating in the proof of Theorem 5.2.1. Some of these properties require us to split the conditions (P4), (P5), and (P6) of the Pauli flow definition: They effectively have two parts each, and it will be easier to consider each part on its own.

Observation 5.3.1. Let (G, I, O, λ) be a labelled open graph and let c be a correction function. For all $u \in \bar{O}$, define the following conditions:

(P4a) $\lambda(u) = XY \Rightarrow u \in \mathrm{Odd}(c(u))$

(P4b) $\lambda(u) = XY \Rightarrow u \notin c(u)$

(P5a) $\lambda(u) = XZ \Rightarrow u \in c(u)$

(P5b) $\lambda(u) = XZ \Rightarrow u \notin \mathrm{Odd}[\![c(u)]\!]$

(P6a) $\lambda(u) = YZ \Rightarrow u \in c(u)$

(P6b) $\lambda(u) = YZ \Rightarrow u \notin \mathrm{Odd}(c(u))$.

Then, (P4) is equivalent to the conjunction of (P4a) and (P4b); (P5) is equivalent to the conjunction of (P5a) and (P5b); and (P6) is equivalent to the conjunction of (P6a) and (P6b).
In particular, when c satisfies (P5), then $u \in c(u)$ and $u \in \mathrm{Odd}(c(u))$, so that $u \notin c(u) \,\Delta\, \mathrm{Odd}(c(u)) = \mathrm{Odd}[\![u]\!]$ and thus (P5a) and (P5b) follow. When c satisfies (P5a) and (P5b), then $u \in c(u)$ and $u \notin \mathrm{Odd}[\![c(u)]\!] = c(u) \,\Delta\, \mathrm{Odd}(c(u))$ and hence $u \in \mathrm{Odd}(c(u))$, i.e. (P5) follows.

Now, we prove different facts regarding matrix products involving flow-demand or order-demand matrices. First, we look at the product of the flow-demand matrix and a column vector.

Lemma 5.3.2. Let (G, I, O, λ) be a labelled open graph. Let $\mathcal{A} \subseteq \bar{I}$. Let $u \in \bar{O}$. Then the product of the u-th row of M with the indicator vector of $\mathcal{A}$ satisfies $M_{u,*}\mathbb{1}^{\bar{I}}_{\mathcal{A}} = \left(M\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right)_u = 1$ if and only if:

$$\begin{aligned} \lambda(u) \in \{X, XY\} \quad &\text{and} \quad u \in \mathrm{Odd}(\mathcal{A}), \text{or} \\ \lambda(u) \in \{XZ, YZ, Z\} \quad &\text{and} \quad u \in \mathcal{A}, \text{or} \\ \lambda(u) = Y \quad &\text{and} \quad u \in \mathrm{Odd}[\![\mathcal{A}]\!]. \end{aligned}$$

Proof. The fact $M_{u,*}\mathbb{1}^{\bar{I}}_{\mathcal{A}} = \left(M\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right)_u$ follows immediately from the definition of matrix multiplication. Now, we consider different cases depending on $\lambda(u)$. Here, $w \in \bar{I}$ is any non-input.

1. Suppose $\lambda(u) \in \{X, XY\}$. Then $M_{u,*}$ encodes $Adj(u)$ and thus:

$$M_{u,w}\left(\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right)_w = \begin{cases} 1 & \text{if } uw \in E \wedge w \in \mathcal{A} \\ 0 & \text{otherwise} \end{cases}$$

Recall that the matrices and computations are over $\mathbb{F}_2$. Then, we have:

$$\left(M\mathbb{1}_{\mathcal{A}}^{\bar{I}}\right)_u = \sum_{w\in\bar{I}} M_{u,w}\left(\mathbb{1}_{\mathcal{A}}^{\bar{I}}\right)_w = \left|\left\{w \in \bar{I} : uw \in E \wedge w \in \mathcal{A}\right\}\right| \bmod 2$$

$$= |\{w \in \mathcal{A} : uw \in E\}| \bmod 2$$

$$= \begin{cases} 1 & \text{if } u \in \mathrm{Odd}(\mathcal{A}) \\ 0 & \text{otherwise} \end{cases}$$

2. Suppose $\lambda(u) \in \{XZ, YZ, Z\}$. Then $M_{u,*}$ equals 0 except for $M_{u,u} = 1$ and thus:

$$M_{u,w}\left(\mathbb{1}_{\mathcal{A}}^{\bar{I}}\right)_w = \begin{cases} 1 & \text{if } u = w \wedge w \in \mathcal{A} \\ 0 & \text{otherwise} \end{cases}$$

We have $u \in \bar{I}$ because inputs cannot be measured in XZ, YZ, Z (cf. Definition 2.5.8). Therefore:

$$\left(M\mathbb{1}_{\mathcal{A}}^{\bar{I}}\right)_u = \sum_{w\in\bar{I}} M_{u,w}\left(\mathbb{1}_{\mathcal{A}}^{\bar{I}}\right)_w = M_{u,u}\left(\mathbb{1}_{\mathcal{A}}^{\bar{I}}\right)_u = \begin{cases} 1 & \text{if } u \in \mathcal{A} \\ 0 & \text{otherwise} \end{cases}$$

3. Suppose $\lambda(u) = Y$. Then $M_{u,*}$ encodes $Adj(u)$ except for $M_{u,u} = 1$ and thus:

$$M_{u,w}\left(\mathbb{1}_{\mathcal{A}}^{\bar{I}}\right)_w = \begin{cases} 1 & \text{if } uw \in E \wedge w \in \mathcal{A} \\ 1 & \text{if } u = w \wedge w \in \mathcal{A} \\ 0 & \text{otherwise} \end{cases}$$

The first two options are disjoint as the graph G is simple, so it does not contain the edge uu. Therefore, the expression for $\left(M\mathbb{1}_{\mathcal{A}}^{\bar{I}}\right)_u$ is the sum of the expressions obtained in the previous two cases. Hence there are the following cases for $\left(M\mathbb{1}_{\mathcal{A}}^{\bar{I}}\right)_u = \sum_{w\in\bar{I}} M_{u,w}\left(\mathbb{1}_{\mathcal{A}}^{\bar{I}}\right)_w$:

- When $u \in \mathrm{Odd}(\mathcal{A})$ and $u \in \mathcal{A}$, we get $\left(M\mathbb{1}_{\mathcal{A}}^{\bar{I}}\right)_u = 1 + 1 = 0$ (recall that we work over $\mathbb{F}_2$),
- When $u \notin \mathrm{Odd}(\mathcal{A})$ and $u \in \mathcal{A}$, we get $\left(M\mathbb{1}_{\mathcal{A}}^{\bar{I}}\right)_u = 0 + 1 = 1$,
- When $u \in \mathrm{Odd}(\mathcal{A})$ and $u \notin \mathcal{A}$, we get $\left(M\mathbb{1}_{\mathcal{A}}^{\bar{I}}\right)_u = 1 + 0 = 1$,

- When $u \notin \mathrm{Odd}(\mathcal{A})$ and $u \notin \mathcal{A}$, we get $\left(M\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right)_u = 0 + 0 = 0$.

Combining the conditions, we obtain:

$$\left(M\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right)_u = \begin{cases} 1 & \text{if } u \in \mathrm{Odd}[\![\mathcal{A}]\!] \\ 0 & \text{otherwise} \end{cases}$$

which ends the proof. □

Using the above lemma, we can provide an interpretation of the product of the flow-demand matrix and the correction matrix.

Lemma 5.3.3. Let (G, I, O, λ) be a labelled open graph and let c be a correction function. Let $u, v \in \bar{O}$. Then, $(MC)_{u,v} = 1$ if and only if:

$$\begin{aligned} \lambda(u) \in \{X, XY\} &\quad \text{and} \quad u \in \mathrm{Odd}(c(v)), \text{or} \\ \lambda(u) \in \{XZ, YZ, Z\} &\quad \text{and} \quad u \in c(v), \text{or} \\ \lambda(u) = Y &\quad \text{and} \quad u \in \mathrm{Odd}[\![c(v)]\!]. \end{aligned}$$

Further, $(NC)_{u,v} = 1$ if and only if:

$$\begin{aligned} \lambda(u) = YZ &\quad \text{and} \quad u \in \mathrm{Odd}(c(v)), \text{or} \\ \lambda(u) = XY &\quad \text{and} \quad u \in c(v), \text{or} \\ \lambda(u) = XZ &\quad \text{and} \quad u \in \mathrm{Odd}[\![c(v)]\!]. \end{aligned}$$

Proof. We start with $(MC)_{u,v} = M_{u,*}C_{*,v}$. Columns of C encode $c(v)$, i.e. $C_{*,v} = \mathbb{1}^{\bar{I}}_{c(v)}$. Hence, from Lemma 5.3.2, we have that $M_{u,*}C_{*,v}$ equals 1 if and only if:

$$\begin{aligned} \lambda(v) \in \{X, XY\} &\quad \text{and} \quad v \in \mathrm{Odd}(c(v)), \text{or} \\ \lambda(v) \in \{XZ, YZ, Z\} &\quad \text{and} \quad v \in c(v), \text{or} \\ \lambda(v) = Y &\quad \text{and} \quad v \in \mathrm{Odd}[\![c(v)]\!]. \end{aligned}$$

The proof for $(NC)_{u,v}$ is analogous, taking into account the construction of each row of N as given in Definition 5.2.9. □

We can now relate several of the Pauli flow conditions from Definition 2.6.16 and Observation 5.3.1, as well as the focusing conditions from Definition 2.6.22, to the matrix product MC.

Theorem 5.3.4. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph and let c be a correction function on Γ. The following two statements are equivalent:

- $MC = Id_{\bar{O}}$, where C is the correction matrix encoding c according to Definition 5.2.4 and M is the flow-demand matrix of Γ,
- c satisfies (P4a), (P5a), (P6a), (P7), (P8), (P9), (F1), (F2), and (F3).

Proof. ($\Leftarrow$): We must show that:

$$(MC)_{u,v} = \begin{cases} 1 & \text{if } u = v \\ 0 & \text{otherwise} \end{cases}$$

We consider three cases depending on $\lambda(u)$.

1. $\lambda(u) \in \{X, XY\}$. By Lemma 5.3.3:

$$(MC)_{u,v} = \begin{cases} 1 & \text{if } u \in \mathrm{Odd}(c(v)) \\ 0 & \text{otherwise} \end{cases}.$$

   Now, we have $u \in \mathrm{Odd}(c(u))$ by (P4a) for $\lambda(u) = XY$ and by (P7) for $\lambda(u) = X$. Furthermore, $u \notin \mathrm{Odd}(c(v))$ for $u \neq v$ by (F2) for either label. Thus, $(MC)_{u,v} = 1$ if and only if $u = v$.

2. $\lambda(u) \in \{XZ, YZ, Z\}$. By Lemma 5.3.3:

$$(MC)_{u,v} = \begin{cases} 1 & \text{if } u \in c(v) \\ 0 & \text{otherwise} \end{cases}.$$

   Now, we have $u \in c(u)$ by (P5a) for $\lambda(u) = XZ$, by (P6a) for $\lambda(u) = YZ$, and by (P8) for $\lambda(u) = X$. Furthermore, $u \notin c(v)$ for $u \neq v$ by (F1). Thus, $(MC)_{u,v} = 1$ if and only if $u = v$.

3. $\lambda(u) = Y$. By Lemma 5.3.3:

$$(MC)_{u,v} = \begin{cases} 1 & \text{if } u \in \mathrm{Odd}[\![c(v)]\!] \\ 0 & \text{otherwise} \end{cases}.$$

   Now, we have $u \in \mathrm{Odd}[\![c(u)]\!]$ by (P9). Finally, $u \notin \mathrm{Odd}[\![c(v)]\!]$ for $v \neq u$ by (F3). Thus, again, $(MC)_{u,v} = 1$ if and only if $u = v$.

Hence, indeed we have

$$(MC)_{u,v} = \begin{cases} 1 & \text{if } u = v \\ 0 & \text{otherwise} \end{cases}$$

for any $u, v \in \bar{O}$, which ends the proof.
$(\Rightarrow)$: This direction is very similar. By assumption, $MC = Id_{\bar{O}}$, so:

$$(MC)_{u,v} = \begin{cases} 1 & \text{if } u = v \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

Again, we consider three cases depending on $\lambda(u)$.

1. $\lambda(u) \in \{X, XY\}$. By Lemma 5.3.3, we get:

   $$(MC)_{u,v} = \begin{cases} 1 & \text{if } u \in \text{Odd}(c(v)) \\ 0 & \text{otherwise} \end{cases}$$

   Combining it with Equation (5.1), we get $u \in \text{Odd}(c(u))$ for any u measured in X, XY and thus (P4a), (P7) hold. Further, we get that $u \notin \text{Odd}(c(v))$ when $u \neq v$ for any u measured in X, XY and thus $\text{Odd}(c(v))$ may only contain v and vertices measured in $\{XZ, YZ, Y, Z\}$, i.e. (F2) holds.

2. $\lambda(u) \in \{XZ, YZ, Z\}$. By Lemma 5.3.3, we get:

   $$(MC)_{u,v} = \begin{cases} 1 & \text{if } u \in c(v) \\ 0 & \text{otherwise} \end{cases}$$

   Combining it with Equation (5.1), we get $u \in c(u)$ for any u measured in XZ, YZ, Z and thus (P5a), (P6a), (P8) hold. Further, we get that $u \notin c(v)$ when $u \neq v$ for any u measured in XZ, YZ, Z and thus $c(v)$ may only contain v and vertices measured in $\{X, XY, Y\}$, i.e. (F1) holds.

3. $\lambda(u) = Y$. By Lemma 5.3.3, we get:

   $$(MC)_{u,v} = \begin{cases} 1 & \text{if } u \in \text{Odd}[\![c(v)]\!] \\ 0 & \text{otherwise} \end{cases}$$

   Combining it with Equation (5.1), we get $u \in \text{Odd}[\![c(u)]\!]$ for all u measured in Y and thus (P9) holds. Further, we get that $u \notin \mathrm{Odd}\llbracket c(v) \rrbracket$ when $u \neq v$ for all u measured in Y, i.e. (F3) holds. □

Similarly, the product NC can be related to the remaining conditions from the flow definition. First, we show that NC without its main diagonal encodes precisely the induced relation $\lhd_c$.

Lemma 5.3.5. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph and let c be a focused correction function on Γ. Then, for any $u, v \in \bar{O}$ such that $u \neq v$, we have $(NC)_{u,v} = 1$ if and only if $v \lhd_c u$. In other words, the off-diagonal part of the matrix NC encodes the relation $\lhd_c$.

Proof. Let $u, v \in \bar{O}$ such that $u \neq v$. By Lemma 5.1.10, if $v \lhd_c u$ then $\lambda(u) \in \{XY, YZ, XZ\}$. Now, we consider three cases on $\lambda(u)$.

1. $\lambda(u) = XY$. Then, by Lemma 5.3.3:

$$(NC)_{u,v} = \begin{cases} 1 & \text{if } u \in c(v) \\ 0 & \text{otherwise} \end{cases}$$

   Since $u \neq v$ and $\lambda(u) \notin \{X, Y\}$, by (P1) we get $v \lhd_c u$ if and only if $u \in c(v)$, as desired.

2. $\lambda(u) = YZ$. Then, by Lemma 5.3.3:

$$(NC)_{u,v} = \begin{cases} 1 & \text{if } u \in \mathrm{Odd}(c(v)) \\ 0 & \text{otherwise} \end{cases}$$

   Since $u \neq v$ and $\lambda(u) \notin \{Y, Z\}$, by (P2) we get $v \lhd_c u$ if and only if $u \in \mathrm{Odd}(c(v))$, as desired.

3. $\lambda(u) = XZ$. Then, by Lemma 5.3.3:

$$(NC)_{u,v} = \begin{cases} 1 & \text{if } u \in \mathrm{Odd}\llbracket c(v) \rrbracket \\ 0 & \text{otherwise} \end{cases}$$

   Since $u \neq v$ and $\lambda(u) \notin \{Y, Z\}$, by (P2) we get $v \lhd_c u$ if and only if $u \in \mathrm{Odd}(c(v))$. As c is focused, it satisfies (F1) and thus $u \notin c(v)$. Hence $u \in \mathrm{Odd}\llbracket c(v) \rrbracket \Leftrightarrow u \in \mathrm{Odd}(c(v))$. Combining with the above, we again get the desired equivalence. □

Now, we can relate the matrix product NC to the Pauli flow conditions.

Theorem 5.3.6. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph and let c be a focused correction function on Γ. The following two statements are equivalent:

- NC is the adjacency matrix of a DAG, where C is the correction matrix encoding c according to Definition 5.2.4 and N is the order-demand matrix of Γ,
- $\prec_c$ is a partial order on $\bar{O}$ and the tuple $(c, \prec_c)$ satisfies (P1), (P2), (P3), (P4b), (P5b), and (P6b).

Proof. ($\Rightarrow$): By Lemma 5.3.5, $\lessdot_c$ is encoded entirely within NC. Since NC is a DAG, $\lessdot_c$ is an acyclic relation and thus its transitive closure is a partial order. Hence, $\prec_c$ satisfies (P1), (P2), (P3) and is a partial order. What remains is to prove (P4b), (P5b), (P6b). Let $u \in \bar{O}$ be any planar measured vertex. Since NC is a DAG, it cannot contain a one-cycle (i.e. a loop) for any vertex u, and thus:

$$(NC)_{u,u} = 0 \tag{5.2}$$

We consider three cases depending on $\lambda(u)$.

1. $\lambda(u) = XY$. By Lemma 5.3.3:

$$(NC)_{u,u} = \begin{cases} 1 & \text{if } u \in c(u) \\ 0 & \text{otherwise} \end{cases}$$

   Combining this with (5.2), we find $u \notin c(u)$ for any u measured in XY, i.e. (P4b) holds.

2. $\lambda(u) = XZ$. By Lemma 5.3.3:

$$(NC)_{u,u} = \begin{cases} 1 & \text{if } u \in \mathrm{Odd}[\![c(u)]\!] \\ 0 & \text{otherwise} \end{cases}$$

   Combining this with (5.2), we find $u \notin \mathrm{Odd}[\![c(u)]\!]$ for any u measured in XZ, i.e. (P5b) holds.

3. $\lambda(u) = YZ$. By Lemma 5.3.3:

$$(NC)_{u,u} = \begin{cases} 1 & \text{if } u \in \mathrm{Odd}(c(u)) \\ 0 & \text{otherwise} \end{cases}$$

Combining this with (5.2), we find $u \notin \mathrm{Odd}(c(u))$ for any u measured in YZ, i.e. (P6b) holds.

($\Leftarrow$): Similarly to the other direction, from (P4b), (P5b), and (P6b), we get $(NC)_{u,u} = 0$ for all planar measured u. Further, the off-diagonal entries of NC encode $\lhd_c$ by Lemma 5.3.5. Hence, as $\prec_c$ is a partial order, $\lhd_c$ is acyclic and thus the off-diagonal part of NC is acyclic. Combining the two facts, we indeed find that NC is the adjacency matrix of a DAG. □

Combining the previous theorems, we obtain the desired algebraic interpretation of Pauli flow: the main result of this chapter.

Theorem 5.2.1 (Repeated)**.** Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph, c be a correction function, M_Γ be the flow-demand matrix of Γ, and N_Γ be the order-demand matrix of Γ. Then c is focused extensive if and only if the following two facts hold for the corresponding correction matrix C:

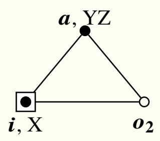
- $M_\Gamma C = Id_{\bar{O}}$,
- $N_\Gamma C$ is the adjacency matrix of a DAG.

Proof. ($\Rightarrow$): Let c be focused extensive. By Theorem 5.1.9, this means $(c, \prec_c)$ is a minimal focused Pauli flow. Then by Theorem 5.3.4, $MC = Id_{\bar{O}}$. Moreover, by Theorem 5.3.6, NC is the adjacency matrix of a DAG.
($\Leftarrow$): This follows immediately from Theorems 5.3.4 and 5.3.6, as well as Observation 5.3.1. □

For labelled open graph from Example 5.1.2, in Examples 5.2.8 and 5.2.10, we have verified Theorem 5.2.1.

When the numbers of inputs and outputs are equal, i.e. $n_I = n_O$, the flow-demand matrix is square. Thus, it has a unique right inverse (if any), which is the inverse. Thus, Theorem 5.2.1 provides an alternative proof of Theorem 2.6.24. Also by Theorem 5.2.1, to verify the existence of flow, we only need to check that such an inverse exists and that the product of the order-demand matrix and the inverse of the flow-demand matrix forms a DAG. For an example of this $n_I = n_O$ case, see the following:

Example 5.3.7. Condiser the following labelled open graph:



Its flow-demand matrix M and order-demand matrix N are:

M	a	o
i	1	1
a	1	0

N	a	o
i	0	0
a	0	1

The flow-demand M is square, and hence, it can have precisely one right-inverse (which is just the inverse). Thus, the only correction matrix C that can satisfy condition $MC = Id$ from Theorem 5.2.1 is:

C	i	a
a	0	1
o	1	1

However, the product NC contains a loop at a and it is therefore not a DAG, so the labelled open graph has no flow:

NC	i	a
i	0	0
a	1	1

The case of equal numbers of inputs and outputs has another interesting aspect – it is possible to switch the sets of inputs and outputs, obtaining another labelled open graph which may have Pauli flow (if, on the other hand, the numbers of inputs and outputs are different, then it is immediate that at most one of the pair of labelled open graphs could have flow). Indeed, this reversal transformation preserves Pauli flow, as shown in the next section.

5.4 Reversibility of Pauli flow

For labelled open graphs where the number of inputs is equal to the number of outputs and all measurements are of XY type, the previous algebraic interpretation was used to show two properties of focused gflow: it is unique, and it is reversible [125, Theorem 4]. Here, reversing a labelled open graph means swapping the roles of inputs and outputs. Reversibility of the flow then means that the reverse labelled open graph has gflow if and only if the original labelled open graph has gflow. Moreover, the correction matrix of the gflow on the reverse labelled open graph was shown to be C^T, where C is the correction matrix of the original gflow.

Using the new algebraic interpretation for all measurement types, it has been straightforward to show the uniqueness of focused Pauli flow on labelled open graphs with equal numbers of inputs and outputs. We now prove that under these conditions, focused Pauli flow can also be reversed, although the relationship between the reversed and the original flow is less straightforward if there are measurements of type XZ, YZ, or Z.

Lemma 5.4.1. Let (G, I, O, λ) be a labelled open graph. Then (G, I, O, λ) has Pauli flow if and only if $(G', I \setminus O, O \setminus I, \lambda)$ does, where G' is G with vertices in $I \cap O$ removed.

Proof. Suppose $v \in I \cap O$. A correction function c in Pauli flow has codomain $\mathcal{P}(\bar{I})$ and $v \in I$ so v cannot be used in any correction set. Further, as $v \in O$, v is not measured and so $c(v)$ is undefined and the partial order does not operate on v. Therefore v does not impact any of the nine flow conditions in any way. □

As an alternative, we could observe that if $v \in I \cap O$, then there is no v row or column in either of the flow-demand, order-demand, and correction matrices.

We will assume throughout this section that all labelled open graphs satisfy $I \cap O = \emptyset$. This is without loss of generality regarding the existence of Pauli flow, as shown by the above lemma. The lemma is an if and only if statement, so vertices that are simultaneously inputs and outputs can always be reintroduced without affecting the existence of flow.

Definition 5.4.2. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph. Define the following subsets of vertices:

- $\mathcal{B} := V \setminus (I \cup O)$, the set of internal vertices,
- $\mathcal{X} := \{v \in \mathcal{B} \mid \lambda(v) \in \{XY, X, Y\}\}$, the set of '$X$-like' measured internal vertices,

- $\mathcal{Z} := \{v \in \mathcal{B} \mid \lambda(v) \in \{XZ, YZ, Z\}\}$, the set of 'Z-like' measured internal vertices,
- $\mathcal{P} := \{v \in \mathcal{B} \mid \lambda(v) \in \{X, Y, Z\}\}$ the set of Pauli measured internal vertices, and
- $\mathcal{L} := \{v \in \mathcal{B} \mid \lambda(v) \in \{XY, XZ, YZ\}\}$ the set of planar measured internal vertices.

Definition 5.4.3. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph which satisfies $|I| = |O|$. Then its *reverse labelled open graph* is $\Gamma' = (G, O, I, \lambda')$ where the role of inputs and outputs is swapped, and

$$\lambda'(v) = \begin{cases} \lambda(v) & \text{if } v \in \mathcal{B} \\ XY & \text{if } v \in O. \end{cases}$$

Definition 5.4.4. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph. Define the *extended adjacency matrix* A of Γ as

$$A_{vw} := \begin{cases} 1 & \text{if } \{v, w\} \in E \vee (v = w \wedge \lambda(v) \in \{Y, XZ\}) \\ 0 & \text{otherwise.} \end{cases}$$

We will need four disjoint submatrices of the extended adjacency matrix (which do not quite cover the full matrix):

- Let A_{00} be the submatrix of A whose rows are given by $I \cup \mathcal{X}$ and whose columns are given by $\mathcal{X} \cup O$.
- Let A_{01} be the submatrix of A whose rows are given by $I \cup \mathcal{X}$ and whose columns are given by $\mathcal{Z}$.
- Let A_{10} be the submatrix of A whose rows are given by $\mathcal{Z}$ and whose columns are given by $\mathcal{X} \cup O$.
- Let A_{11} be the submatrix of A whose rows and columns are both given by $\mathcal{Z}$.

In summary, where $*$ indicates unnamed blocks of the extended adjacency matrix:

$$A = \begin{array}{c} \\ \\ \\ \end{array}\begin{array}{ccc|c} \mathcal{I} & \mathcal{X}\cup\mathcal{O} & \mathcal{Z} & \\ * & A_{00} & A_{01} & \mathcal{I}\cup\mathcal{X} \\ * & * & * & \mathcal{O} \\ * & A_{10} & A_{11} & \mathcal{Z} \end{array} \tag{5.3}$$

Observation 5.4.5. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph which satisfies $|I| = |O|$, and let Γ' be its reverse. Then the flow demand matrices M of Γ and M' of Γ' are given by

$$M = \begin{array}{cc|c} \mathcal{X}\cup\mathcal{O} & \mathcal{Z} & \\ A_{00} & A_{01} & \mathcal{I}\cup\mathcal{X} \\ 0 & Id & \mathcal{Z} \end{array} \qquad \text{and} \qquad M' = \begin{array}{cc|c} \mathcal{I}\cup\mathcal{X} & \mathcal{Z} & \\ (A_{00})^T & (A_{10})^T & \mathcal{X}\cup\mathcal{O} \\ 0 & Id & \mathcal{Z} \end{array} = \begin{pmatrix} A_{00} & 0 \\ A_{10} & Id \end{pmatrix}^T$$

Observation 5.4.6. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph which satisfies $|I| = |O|$, and let Γ' be its reverse. The flow demand matrix N of Γ is a matrix whose rows are labelled by non-outputs, whose columns are labelled by non-inputs, and where the rows corresponding to inputs or Pauli-measured vertices are identically 0. We can thus write $N = PS$, where P is a diagonal matrix whose rows and columns are both labelled by non-outputs, such that:

$$P_{vw} = \begin{cases} \delta_{vw} & \text{if } v \in \mathcal{L} \\ 0 & \text{otherwise,} \end{cases} \qquad \text{and} \qquad S = \begin{array}{cc|c} \mathcal{X}\cup\mathcal{O} & \mathcal{Z} & \\ J & 0 & \mathcal{I}\cup\mathcal{X} \\ A_{10} & A_{11} & \mathcal{Z} \end{array}$$

with $J_{vw} = \delta_{vw}$. While J is square and takes values in $\{0, 1\}$, it is not an identity matrix because the set of row labels is not the same as the set of column labels. Similarly, $N' = P'S'$, where P' is a diagonal matrix whose rows and columns are both labelled by non-inputs, such that:

$$P'_{vw} = \begin{cases} \delta_{vw} & \text{if } v \in \mathcal{L} \\ 0 & \text{otherwise,} \end{cases} \qquad \text{and} \qquad S' = \begin{array}{cc|c} \mathcal{I}\cup\mathcal{X} & \mathcal{Z} & \\ J^T & 0 & \mathcal{X}\cup\mathcal{O} \\ (A_{01})^T & A_{11} & \mathcal{Z} \end{array} = \begin{pmatrix} J & A_{01} \\ 0 & A_{11} \end{pmatrix}^T$$

because A_{11} is symmetric.

The decomposition of the order-demand matrix in the above Observation highlights the symmetry between flow-demand and order-demand matrix for the same labelled open graph: In each case, one row of the block form is taken directly from the extended adjacency matrix, while the other row has an identity-like block on the diagonal and an all-zero block on the off-diagonal. The big difference is that some of the rows in the order-demand matrix are then set to zero via multiplication by the diagonal matrix P.

By comparing with the extended adjacency matrix A in (5.3) and recalling that A is symmetric by construction, we note furthermore that the matrices M and S encode almost all the information of the extended adjacency matrix: the only parts missing are edges among inputs and edges among outputs (which can straightforwardly be seen to be irrelevant to the existence of flow).

Theorem 5.4.7. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph which satisfies $|I| = |O|$. Let Γ' be the reverse of Γ. Then Γ has Pauli flow if and only if Γ' has Pauli flow.

Proof. Without loss of generality, assume $I \cap O = \emptyset$.
($\Rightarrow$): First, suppose Γ has Pauli flow. By Theorem 5.2.1, the inverse C of its flow demand matrix M exists and the product NC with the order-demand matrix N forms a DAG.
Now by Observation 5.4.5, the flow demand matrix for Γ is

$$M = \begin{array}{c} \\ \\ \\ \end{array}\begin{array}{cc|c} \mathcal{X}\cup O & \mathcal{Z} & \\ \hline A_{00} & A_{01} & I\cup\mathcal{X} \\ 0 & Id & \mathcal{Z} \end{array}$$

Break its inverse $C = M^{-1}$ into analogous blocks, i.e. let

$$C = \begin{array}{cc|c} I\cup\mathcal{X} & \mathcal{Z} & \\ \hline C_{00} & C_{01} & \mathcal{X}\cup O \\ C_{10} & C_{11} & \mathcal{Z} \end{array}$$

such that

$$Id = MC = \begin{array}{cc|c} I\cup\mathcal{X} & \mathcal{Z} & \\ \hline A_{00}C_{00} + A_{01}C_{10} & A_{00}C_{01} + A_{01}C_{11} & I\cup\mathcal{X} \\ C_{10} & C_{11} & \mathcal{Z} \end{array}$$

Then we have $C_{10} = 0$ and $C_{11} = Id$ by inspection. Thus the top left block implies $Id = A_{00}C_{00}$, i.e. $C_{00} = (A_{00})^{-1}$. The top right block becomes $0 = A_{00}C_{01} + A_{01}$. Hence $A_{00}C_{01} = A_{01}$ (as we are working over $\mathbb{F}_2$) and finally $C_{01} = (A_{00})^{-1}A_{01} =$

$C_{00}A_{01}$. To summarise,

$$C = \begin{array}{cc} \begin{array}{cc} I\cup X & \mathcal{Z} \end{array} & \\ \begin{pmatrix} C_{00} & C_{00}A_{01} \\ 0 & Id \end{pmatrix} & \begin{array}{c} X\cup O \\ \mathcal{Z} \end{array} \end{array}$$

Again by Observation 5.4.5, the flow demand matrix for Γ' is

$$M' = \begin{pmatrix} A_{00} & 0 \\ A_{10} & Id \end{pmatrix}^T$$

As A_{00} is invertible (we found $C_{00} = (A_{00})^{-1}$) and because of its block structure, M' is also invertible. By an argument analogous to the above, we find that

$$C' = (M')^{-1} = \begin{pmatrix} C_{00} & 0 \\ A_{10}C_{00} & Id \end{pmatrix}^T$$

with C_{00} the same matrix as above.
It remains to show that $N'C'$ is a DAG, knowing that NC is a DAG. First, by Observation 5.4.6:

$$NC = PSC = P\begin{pmatrix} J & 0 \\ A_{10} & A_{11} \end{pmatrix}\begin{pmatrix} C_{00} & C_{00}A_{01} \\ 0 & Id \end{pmatrix} = P\begin{pmatrix} JC_{00} & JC_{00}A_{01} \\ A_{10}C_{00} & A_{10}C_{00}A_{01} + A_{11} \end{pmatrix}$$

and similarly,

$$\begin{aligned} N'C' = P'S'C' &= P'\begin{pmatrix} J & A_{01} \\ 0 & A_{11} \end{pmatrix}^T\begin{pmatrix} C_{00} & 0 \\ A_{10}C_{00} & Id \end{pmatrix}^T \\ &= P'\left(\begin{pmatrix} C_{00} & 0 \\ A_{10}C_{00} & Id \end{pmatrix}\begin{pmatrix} J & A_{01} \\ 0 & A_{11} \end{pmatrix}\right)^T \\ &= P'\begin{pmatrix} C_{00}J & C_{00}A_{01} \\ A_{10}C_{00}J & A_{10}C_{00}A_{01} + A_{11} \end{pmatrix}^T \end{aligned}$$

We have $(NC)_{vw} = 0$ whenever $v \in I \cup \mathcal{P}$: inputs and Pauli measured vertices are always initial in the partial order. This implies that the values in the *rows* labelled by $I \cup \mathcal{P}$ are irrelevant to determining whether NC is a DAG: it suffices to consider only the submatrix of NC whose rows and columns are both labelled by $\mathcal{L}$. The same holds for $N'C'$.
Now, the three matrices P, P', and J are simply the identity matrix when restricted to rows and columns that are labelled as $\mathcal{L}$. Thus the $\mathcal{L}$ times $\mathcal{L}$ submatrices of the

two products satisfy:

$$(N'C')_{\mathcal{L}}^{\mathcal{L}} = \left(\begin{pmatrix} C_{00}J & C_{00}A_{01} \\ A_{10}C_{00}J & A_{10}C_{00}A_{01} + A_{11} \end{pmatrix}^T\right)_{\mathcal{L}}^{\mathcal{L}}$$
$$= \left(\begin{pmatrix} C_{00} & C_{00}A_{01} \\ A_{10}C_{00} & A_{10}C_{00}A_{01} + A_{11} \end{pmatrix}_{\mathcal{L}}^{\mathcal{L}}\right)^T = \left((NC)_{\mathcal{L}}^{\mathcal{L}}\right)^T$$

since taking submatrices commutes with taking the transpose. By assumption of flow existence for Γ, NC is a DAG. Hence $N'C'$ is a DAG and by Theorem 5.2.1, Γ' has Pauli flow, ending the proof.
($\Leftarrow$): Different measurement labels of the inputs do not change the flow-demand and order-demand matrices of Γ, and hence by Theorem 5.2.1 they do not change the existence of flow. Thus without loss of generality, we can assume that $\lambda(v) = XY$ for all $v \in I$. Using the previous part of the proof, if Γ' has Pauli flow, then so does Γ''. However, $\Gamma'' = \Gamma$ (up to a potential and irrelevant difference in measurement labels of inputs), so Γ has Pauli flow, ending the proof. □

Corollary 5.4.8. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph which satisfies $|I| = |O|$. Suppose Γ has focused Pauli flow $(c, \prec_c)$ and its reverse Γ' has focused Pauli flow $(c', \prec_{c'})$, then:

- For all $u \in I \cup \mathcal{X}$ and $w \in \mathcal{X} \cup O$, we have $w \in c(u) \Leftrightarrow u \in c'(w)$.
- For all $u, w \in \mathcal{L}$, we have $u \prec_c w \Leftrightarrow w \prec_{c'} u$.

The complexity of finding a reverse flow requires matrix multiplication, so it is not fundamentally easier than finding a flow from scratch (as we will see in Chapter 6). Yet if a labelled open graph with equal numbers of inputs and outputs is dominated by vertices in $\mathcal{X}$, then it will be more efficient to find the flow of the reverse graph by taking the matrix C_{00} (which describes the flow on the X-like part of the original labelled open graph) and multiplying it with the (smaller) submatrices of the extended adjacency matrix.

When the number of inputs is smaller than the number of outputs, the flow cannot be reversed – the flow-demand matrix of the reverse labelled open graph must have more rows than columns and thus cannot be right-reversible. For example, the flow from Example 5.1.2 cannot be reversed. Similarly, the flow in such a case does not need to be unique as the flow-demand matrix may have many different right inverses. In the future algorithm, we parametrise all such right inverses by relating them via focused sets, which we study next.

5.5 Focused sets

The flow-demand matrix has another connection to Pauli flow that we explore in this section, this time not via its inverse but via its kernel. First, we define focused sets: sets with trivial net effect when used for corrections.

Definition 5.5.1 (Focused set [153, part of Definition 4.3])**.** Given the labelled open graph $\Gamma = (G, I, O, \lambda)$, a set $\mathcal{A} \subseteq \bar{I}$ is *focused over* $S \subseteq \bar{O}$ if:

(Fs1) $\forall w \in S \cap \mathcal{A}. \lambda(w) \in \{XY, X, Y\}$

(Fs2) $\forall w \in S \cap \mathrm{Odd}(\mathcal{A}). \lambda(w) \in \{XZ, YZ, Y, Z\}$

(Fs3) $\forall w \in S. \lambda(w) = Y \Rightarrow w \notin \mathrm{Odd}[\![\mathcal{A}]\!]$, where $w \notin \mathrm{Odd}[\![\mathcal{A}]\!]$ is equivalent to $w \in \mathcal{A} \Leftrightarrow w \in \mathrm{Odd}(\mathcal{A})$.

A set is simply called *focused* if it is focused over $\bar{O}$. The set of all focused sets for Γ is denoted $\mathfrak{F}_\Gamma$.

The focusing conditions for the Pauli flow (cf. Definition 2.6.22) equivalently say that the correction set $c(v)$ is focused over $\bar{O} \setminus \{v\}$ for all $v \in \bar{O}$. The above definition was introduced by Simmons, who also proved that the focused sets for a given labelled open graph form a group [153, Lemma B.8]. Here, we provide an alternative interpretation for the collection of focused sets, by providing an isomorphism with the kernel of the flow-demand matrix, so that the focused sets form a vector space. First, we show that given some set $\mathcal{A} \subseteq \bar{I}$ one can find a maximal set over which $\mathcal{A}$ is focused by considering the product $M\mathbb{1}^{\bar{I}}_{\mathcal{A}}$.

Lemma 5.5.2. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph and $\mathcal{A} \subseteq \bar{I}$ be any subset of non-inputs. Then, $\mathcal{A}$ is focused over $\mathrm{supp}\left(\mathbb{1}^{\bar{O}}_{\bar{O}} + M\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right)$ and $\mathcal{A}$ is not focused over any larger subset of the non-outputs.

Proof. Before beginning the proof, we will explain the notation $\mathrm{supp}\left(\mathbb{1}^{\bar{O}}_{\bar{O}} + M\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right)$, which can be unpacked as follows: $M\mathbb{1}^{\bar{I}}_{\mathcal{A}}$ is the result of multiplying the flow-demand matrix by the vector corresponding to the set $\mathcal{A}$. This vector is in the space $\mathbb{F}_2^{\bar{O}}$. Next, we add to it $\mathbb{1}^{\bar{O}}_{\bar{O}}$, i.e. the vector of all-1s in $\mathbb{F}_2^{\bar{O}}$. Hence, we are flipping all values in $M\mathbb{1}^{\bar{I}}_{\mathcal{A}}$. Next, we take the support of this vector: these are the vertices v such that the flipped vector has a 1 for its v-th entry. This means, $M\mathbb{1}^{\bar{I}}_{\mathcal{A}}$ has a 0 for its v-th entry. Thus, the lemma claims that $\mathcal{A}$ is focused over the set of vertices corresponding to the 0-valued entries of $M\mathbb{1}^{\bar{I}}_{\mathcal{A}}$. Considering a vector as a linear map via the inner product, these entries correspond to the *kernel of* $M\mathbb{1}^{\bar{I}}_{\mathcal{A}}$. We are now ready to proceed to the proof itself.

Let $F_{\mathcal{A}} = \operatorname{supp}\left(\mathbb{1}^{\bar{O}}_{\bar{O}} + M\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right)$. By inspecting conditions (Fs1)-(Fs3), for any sets $S, S' \subseteq \bar{O}$, $\mathcal{A}$ is simultaneously focused over S and focused over S' if and only if it is focused over $S \cup S'$. Therefore, it is sufficient to show that for all $v \in \bar{O}$: $v \in F_{\mathcal{A}}$ if and only if $\mathcal{A}$ is focused over $\{v\}$.

($\Rightarrow$): Let $v \in F_{\mathcal{A}}$. Then $1 = \left(\mathbb{1}^{\bar{O}}_{\bar{O}} + M\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right)_v = 1 + \left(M\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right)_v$. Hence, $0 = \left(M\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right)_v$. Therefore $M_{v,*}\mathbb{1}^{\bar{I}}_{\mathcal{A}} = 0$ and by Lemma 5.3.2, we get:

$$\begin{aligned} \lambda(v) \in \{X, XY\} &\quad \text{and} \quad v \notin \operatorname{Odd}(\mathcal{A}), \text{or} \\ \lambda(v) \in \{XZ, YZ, Z\} &\quad \text{and} \quad v \notin \mathcal{A}, \text{or} \\ \lambda(v) = Y &\quad \text{and} \quad v \notin \operatorname{Odd}[\![\mathcal{A}]\!]. \end{aligned}$$

Hence $\mathcal{A}$ satisfies all three conditions (Fs1), (Fs2), and (Fs3) over the set $\{v\}$, as required.

($\Leftarrow$): Let $\mathcal{A}$ be focused over $\{v\}$. Then, by (Fs2):

$$v \notin \operatorname{Odd}(\mathcal{A}) \quad \text{when} \quad \lambda(v) \in \{X, XY\}.$$

Similarly, by (Fs1):

$$v \notin \mathcal{A} \quad \text{when} \quad \lambda(v) \in \{XZ, YZ, Z\},$$

and by (Fs3):

$$v \notin \operatorname{Odd}[\![\mathcal{A}]\!] \quad \text{when} \quad \lambda(v) = Y.$$

Hence, again using Lemma 5.3.2, we get $M_{v,*}\mathbb{1}^{\bar{I}}_{\mathcal{A}} = 0$ and thus $v \in F_{\mathcal{A}}$. □

Using the above lemma, we find an isomorphism between the kernel of the flow-demand matrix and the collection of focused sets.

Theorem 5.5.3. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph and let M be its flow-demand matrix. Then $\mathfrak{F}_\Gamma \cong \ker M_\Gamma$.

Proof. Suppose $\mathcal{A} \in \mathfrak{F}$. By Lemma 5.5.2, any set $\mathcal{A}$ is focused over $\operatorname{supp}\left(\mathbb{1}^{\bar{O}}_{O} + M\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right)$. Combining these, we find $\operatorname{supp}\left(\mathbb{1}^{\bar{O}}_{O} + M\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right) = \bar{O}$, which implies $\left(M\mathbb{1}^{\bar{I}}_{\mathcal{A}}\right)_v = 0$ for all $v \in \bar{O}$. In other words, $\mathbb{1}^{\bar{I}}_{\mathcal{A}} \in \ker M$.
Now suppose $w \in \ker M$. Again by Lemma 5.5.2, this implies that $\operatorname{supp}(w)$ is focused over $\bar{O}$, i.e. $\operatorname{supp}(w)$ is a focused set.
Therefore, an isomorphism between $\mathfrak{F}$ and $\ker M$ is given by the indicator function $\mathbb{1}^{\bar{I}}_{\bullet}$ and its inverse is given by the support $\operatorname{supp}(\bullet)$. □

For example, the kernel of the flow-demand matrix from Example 5.2.8 consists of two vectors:

$$\begin{pmatrix}0 & 0 & 0 & 0 & 0 & 0\end{pmatrix}^T \text{ and } \begin{pmatrix}0 & 0 & 1 & 0 & 1 & 1\end{pmatrix}^T$$

corresponding to focused sets $\emptyset$ and $\{e, o_1, o_2\}$, respectively.

5.6 Number of inputs and outputs

Throughout the thesis, many statements required $|I| = |O|$. Most importantly, the uniqueness of focused flow works only when $|I| = |O|$; otherwise, focused sets arise, potentially leading to an exponential number of focused flows. In the next chapter, we will also see that finding flow is significantly easier when $|I| = |O|$. This assumption naturally occurs when using flow within circuit optimisation, as circuits are unitary. Hence, labelled open graphs obtained from translation to MBQC must have an equal number of inputs and outputs. Similarly, strong deterministic measurement patterns correspond to unitaries when $|I| = |O|$, making cases satisfying this condition most important. Finally, in [125], it is shown that in the case of planar XY measurements only, the existence of gflow corresponds to not only robust determinism but also another notion called equiprobability. However, when $|I| \neq |O|$, gflow no longer guarantees equiprobability.

Thus, a question arises: if the sizes of the input and output sets do not match, can we modify the labelled open graph to one with $|I| = |O|$? In this section, we show that given a labelled open graph with Pauli flow, it is always possible to reduce the number of outputs to match the number of inputs while preserving the existence of Pauli flow for some measurement labelling:

Theorem 5.6.1. Suppose (G, I, O, λ) has Pauli flow and $|O| > |I|$. Then there exists a subset $O' \subseteq O$ such that $|O'| = |I|$ and a labelling λ' such that (G, I, O', λ') has Pauli flow.

Initially, I included these results in [132]. I slightly modified them to take into account improved algebraic interpretation. Before moving to the proof of this theorem, we note this theorem says nothing about measurement labelling λ' other than its existence. In particular, λ' can significantly differ from λ. In fact, λ' will always consist of Pauli measurements only, as motivated by the following theorem:

Theorem 5.6.2. Suppose that a labelled open graph (G, I, O, λ) has Pauli flow. Then, there exists $\lambda' \colon \bar{O} \to \{X, Y, Z\}$ such that (G, I, O, λ') has Pauli flow.

Proof. We start by fixing a Pauli flow $(c, \prec)$ on (G, I, O, λ). The conditions for a planar measurement XY combine the requirements for the two Pauli measurements X and Y. Hence, swapping XY measurements to X preserves $(c, \prec)$ as the Pauli flow. Similarly, we can swap XZ and YZ mesaurements to Z. □

Based on the above, in the proof of Theorem 5.6.1, we can restrict ourselves to all Pauli measurement labels. The procedure above does not change the flow-demand matrix, as the X and XY measurements are encoded similarly. Analogous property holds for Z, YZ, and XZ measurements. However, the order-demand matrix becomes identically zero, as the following observation points out:

Observation 5.6.3. Let (G, I, O, λ) be a labelled open graph with $\lambda(v) \in \{X, Y, Z\}$ for all $v \in \bar{O}$, i.e. $\Lambda_{Pauli} = \bar{O}$. Then the order-demand matrix of (G, I, O, λ) is identically zero.

Out of the three Pauli measurements, Z measured vertices have additional useful properties: they can be removed or introduced without affecting flow existence.

Lemma 5.6.4 (Introduction of Z measured vertex [121, Proposition 4.1])**.** Let (G, I, O, λ) have Pauli flow. Then, a new Z measured vertex $v \notin V$ can be added to G, with any edges from v, without affecting flow existence. In other words, any labelled open graph (G', I, O, λ') has Pauli flow, where:

$$\begin{aligned} V(G') &= V \cup \{v\} \\ G'[V] &= G \\ \lambda'|_{V \setminus O} &= \lambda \\ \lambda'(v) &= Z. \end{aligned}$$

We are almost ready to prove Theorem 5.6.1. The proof will be based on the following simple fact:

Lemma 5.6.5. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph satisfying $|O| > |I|$ with $\lambda(v) \in \{X, Y, Z\}$ for all $v \in \bar{O}$, i.e. $\Lambda_{Pauli} = \bar{O}$. Further, assume that Γ has Pauli flow. Then there exists $v \in V$ that can be changed to be Z measured while preserving Pauli flow, where initially either:

- v is an output, i.e. $v \in O$,
- or v is not measured in Z basis, i.e. $\lambda(v) \in \{X, Y\}$.

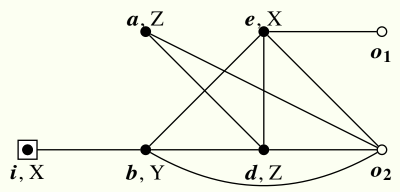
Proof. Let c be a focused extensive correction function for Γ. Let M be the flow-demand matrix of Γ and C the correction matrix of function c.
By Theorem 5.2.1, $MC = Id$ and therefore rank $M = \bar{O}$. Hence, there exists invertible $|\bar{O}| \times |\bar{O}|$ submatrix M' of M. Since $|O| > |I|$, there are more columns than rows in M.
Thus, $v \in \bar{I}$ exists such that the v column $M_{*,v}$ is not included in M'. Suppose $v \in \bar{O}$ and $\lambda(v) = Z$, then the v row $M_{v,*}$ exists in M and is identically zero except for its intersection with the column $M_{*,v}$. Thus, the v column must be included in M'; otherwise, the v row of M' would be identically zero, contradicting the invertability of M'. Therefore $v \in O$ or $\lambda(v) = \{X, Y\}$ satisfying last part of the lemma.
It remains to show that v can be changed to be Z measured. Let M_v be the matrix M with the column v removed, and Γ_v be the labelled open graph Γ with vertex v removed. We know that M is right-invertible, and since the column of v is not included in M', M_v is also right-invertible, as it still contains the invertible submatrix M'. Then M_v is the flow-demand matrix for the labelled open graph Γ_v.
By Observation 5.6.3, Γ with vertex v has an order-demand matrix N_v equal to identically zero, and by the above, its flow-demand matrix M_v is right-invertible. Hence, by Theorem 5.2.1, Γ_v has Pauli flow (since $N_v C' = 0$ for any C'). Therefore, by Theorem 5.6.4, v can be reintroduced to Γ_v with same connections as v has in Γ, while making $\lambda(v) = Z$, ending the proof. □

With the above Lemma, the proof of Theorem 5.6.1 becomes very simple:

Proof of Theorem 5.6.1. Let (G, I, O, λ) be a labelled open graph with Pauli flow. By Theorem 5.6.2, there is λ' such that (G, I, O, λ') has Pauli flow and $\lambda'(v) \in \{X, Y, Z\}$. Application of Lemma 5.6.5 to (G, I, O, λ') strictly increases the number of Z measured vertices and is possible whenever $|O| > |I|$. Since the number of Z measured vertices cannot go beyond $|V|$, repeated application of Lemma 5.6.5 to

(G, I, O, λ') eventually results in some labelled open graph (G, I, O', λ'') with an equal number of inputs and outputs that has Pauli flow and in which $O' \subseteq O$, as required. □

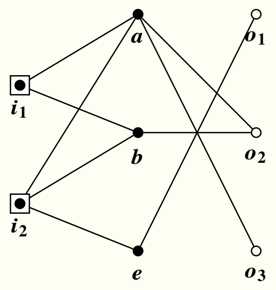
Example 5.6.6. Consider the labelled open graph from Example 5.1.2. It has more outputs than inputs and has Pauli flow so Theorem 5.6.1 applies. Firstly, according to Theorem 5.6.2, the labelled open graph is transformed into the following:



Its flow-demand matrix M is equal to that of the original labelled open graph, i.e. it is the matrix from Example 5.2.8. We have $M_{*,o_1} = M_{*,e} + M_{*,o_2}$ meaning the column of o_1 is linearly dependant with other columns. Hence, it is possible to remove o_1 (equivalently: change it to be Z measured). The output o_1 cannot be removed in a flow-preserving way in the initial labelled open graph, i.e. the one from Example 5.1.2 – skipping the details: while the only change to the flow-demand matrix is column removal, the inverse changes in a more complex way than just removing one row. In this case, removal of o_2 results in a correction matrix such that the product with the order-demand matrix is no longer a DAG, and thus, by Theorem 5.2.1, removal of o_2 also removes flow.

We also point out that Theorem 5.6.1 does not always instantly remove outputs for labelled open graphs with only Pauli measurements. Sometimes, it may be necessary to first change some of the measurements to Z measurements, and only afterwards may some outputs become possible to remove while preserving flow. In other words, the second possibility from Lemma 5.6.5 may be necessary to apply in the process:

Example 5.6.7. Consider the following open graph:



With all non-outputs measured in X, it does have Pauli flow. However, it is impossible to remove any of the outputs outright. It becomes possible to remove output o_2 (or o_3) after changing the measurement label of b to Z.

The step in Lemma 5.6.5 can be viewed as basis finding: in each step, we take the basis of the column space of the flow-demand matrix and remove a vertex whose column does not appear in the flow-demand matrix. Such a basis finding approach has other applications. In the case of X labelling only, suppose we are given O but not I. We can then find I with $|I| = |O|$ resulting in flow, or determine that no flow exists for any set of inputs.

Lemma 5.6.8. Let G be a graph and $O \subseteq V$. Let $\lambda\colon \bar{O} \to \{X\}$. Then either (G, I, O, λ) does not have Pauli flow for any I, or (G, I, O, λ) has Pauli flow for some I with $|I| = |O|$.

Proof. Consider $(G, \emptyset, O, \lambda)$. Suppose that it does not have Pauli flow. Since $\lambda(v) = X$ for all $v \in \bar{O}$, the flow-demand matrix M is equal to reduced adjacency matrix $A := Adj_G\,|^{\emptyset}_{\bar{O}}$ and the order-demand matrix is a zero matrix. By Theorem 5.2.1, lack of flow translates to A not being right-invertible. Changing inputs to a different set than $\emptyset$ corresponds to the removal of columns but not rows from the reduced adjacency matrix A – it cannot make the matrix right-invertible. Hence, there is no I for which (G, I, O, λ) has Pauli flow. Conversely, if $(G, \emptyset, O, \lambda)$ has Pauli flow, then we can choose $|\bar{O}| \times |\bar{O}|$ submatrix from the reduced adjacency matrix A. The not chosen columns can be removed without breaking the flow, i.e. the corresponding vertices can be changed to be inputs. We note that some output could be changed to be an input. This process always turns $|V| - |\bar{O}| = |O|$ vertices into inputs, which ends the proof. □

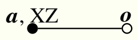
Finally, again in the case of X labelling only, suppose we are given inputs. Can we find a minimal (smallest) set of outputs resulting in Pauli flow? The answer is yes.

Lemma 5.6.9. Let G be a graph and $I \subseteq V$. Let $\lambda(v) = X$ for $v \in V$. Then a minimal O resulting in $(G, I, O, \lambda\mid_{\bar{O}})$ having Pauli flow can be efficiently found.

Proof. Consider the reduced adjacency matrix $Adj_G\mid_{\emptyset}^{\bar{I}}$ of $(G, I, \emptyset, \lambda)$. Let D be a set of rows forming the basis of the space given by all rows. Let R be the set of vertices whose rows are not in D. Then $O := R$ is the required minimal set: $(G, I, O, \lambda\mid_{\bar{O}})$ has Pauli flow – in its reduced adjacency matrix $Adj_G\mid_{O}^{\bar{I}}$ the rows are linearly independent, so the matrix is right-invertible which suffices for flow by the same argument as in the proof of Lemma 5.6.8. Also, O is minimal – any smaller set O' of outputs cannot result in a right-invertible reduced adjacency matrix, as such a matrix would have more rows than $Adj_G\mid_{O}^{\bar{I}}$, and all of its rows would be from $Adj_G\mid_{\emptyset}^{\bar{I}}$. But $Adj_G\mid_{O}^{\bar{I}}$ has the maximal number of linearly independent rows, as those rows form a basis of the space spanned by the rows of $Adj_G\mid_{\emptyset}^{\bar{I}}$. We note that some inputs could be changed to be outputs. □

While the above methods can help equalise the number of inputs and outputs, they have a critical limitation: they only concern flow-demand matrices and not order-demand matrices. That is not a problem in the case of Pauli measurements, as the partial order is trivial in the focused flow. However, planar measurements impose non-trivial order, and the above methods may break. To illustrate this restriction, see the following example.

Example 5.6.10. Consider the following labelled open graph with one output and no inputs:



It does have Pauli flow with $c(a) = \{a, o\}$. However, it is impossible to turn vertex a into an input or remove the output o in a way that preserves Pauli flow. Any transformations must change the measurement basis of a.

Chapter 6

Flow Algorithms

While the algebraic interpretation from Chapter 5 can aid various proofs extending the understanding of Pauli flow, the most important application is in the algorithms for finding Pauli flow. As explained in Subsection 2.6.5, the problem of finding flow is fundamental and was extensively researched.

In this chapter, we present new algorithms, improving the complexity of the previously known methods. Furthermore, we establish the first lower bounds for the problem of finding Pauli flow. Whenever considering a graph-based problem, we use n to denote the number of vertices.

Firstly, let's formally define the flow finding problem:

FlowFinding
Input: A labelled open graph $\Gamma = (G, I, O, \lambda)$.
Output: Pauli flow $(c, \prec)$ for Γ, or a message that no such Pauli flow exists.

As explained earlier, this problem was already known to be in $\boldsymbol{FP}$, yet existing algorithms were rather slow. Using our algebraic interpretation, we provide a new $O\left(n^3\right)$ method. We treat the special case $|I| = |O|$ separately. While it would be sufficient in principle to consider only the general case, it is nevertheless worth distinguishing this special case due to the remarkably simple algorithm. Furthermore, we also explain potential further speed-up for this special case where $|I| = |O|$.

Finally, we also explore the problem of finding flow in a slightly different setting: looking for flow on open graphs (we specifically mean unlabelled open graphs). Given an open graph, we show that it is possible to decide the existence of measurement labelling that results in a labelled open graph with Pauli flow.

Corresponding papers. The novel results in this chapter have appeared in my papers: [132] (single author) and [133] (first author; co-written with my supervisor Miriam Backens).

Structure: In Section 6.1, we slightly adapt main results from Chapter 5, provide a lower bound for the flow-finding problem, and state explicit constructions of the relevant matrices. In Section 6.2, we focus on algorithm for the case $|I| = |O|$. The general case follows in Section 6.3. In Section 6.4, we explore flow on unlabelled open graphs. Lastly, the commentary on efficiency of presented algorithms is included in Section 6.5.

6.1 Preliminary constructions

Theorem 5.2.1 is the backbone of our algorithms. However, we will use a slightly restated version:

Corollary 6.1.1. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph. Let M be the flow-demand matrix of Γ and let N be the order-demand matrix of Γ. Then Γ has Pauli flow if and only if there exists a correction matrix C such that:

- C has shape $(n - n_I) \times (n - n_O)$ with rows corresponding to $\bar{I}$ and columns corresponding to $\bar{O}$,
- $MC = Id_{\bar{O}}$, and
- NC is the adjacency matrix of some directed acyclic graph.

Furthermore, for any such matrix C, the corresponding minimal focused Pauli flow $(c, \prec_c)$ is given as follows, where $v, w \in \bar{O}$:

- $c(v) := \{u \in \bar{I} \mid C_{u,v} = 1\}$,
- $v \lessdot_c w \Leftrightarrow (NC)_{w,v} = 1$, and
- $\prec_c$ is the transitive closure of $\lessdot_c$.

Proof. By inspection, it follows from the theorems in Chapter 5. □

Based on this corollary, we can immediately obtain the lower bound for **FindFlow**:

Theorem 6.1.2. The problem of finding Pauli flow is at least as computationally expensive as the problem of finding the inverse of a matrix over $\mathbb{F}_2$.

Proof. Let M be an arbitrary $n \times n$ matrix over $\mathbb{F}_2$. Let $\Gamma = (G, I, O, \lambda)$ be defined as follows:

$$\begin{aligned}
I &= \{i_1, \ldots, i_n\} \\
O &= \{o_1, \ldots, o_n\} \\
V &= I \cup O \\
E &= \{(i_k, o_\ell) \mid M_{k,\ell} = 1 \text{ for } k, \ell \in \{1, \ldots, n\}\} \\
G &= (V, E) \\
\lambda(i_k) &= X \quad (\forall k \in \{1, \ldots, n\})
\end{aligned}$$

Then, the flow-demand matrix M_Γ equals M and the order-demand matrix N_Γ is identically 0. Hence, by Theorem 6.1.1, Γ has Pauli flow if and only if M is invertible and the correction function is encoded by the inverse of M. □

For the moment, no better lower bound for finding the inverse of a matrix than $\Omega(n^2)$ is known[4]. Therefore, finding Pauli flow must take at least $\Omega(n^2)$ operations. In particular, it can be no better than the best-known method for finding causal flow from [126] with a runtime of $O(m)$ where m is the number of edges.

Before we move to the algorithms' main part, we examine the explicit construction of flow-demand and order-demand matrices for the input labelled open graph. This pseudocode relates to Definitions 5.2.7 and 5.2.9.

Algorithm 1 Construction of flow and order-demand matrices

```
    Returns the flow-demand matrix of a given labelled open graph
 1: procedure FLOW-DEMANDMATRIX($G, I, O, \lambda$)
 2:     $M = Adj_G |_{\bar{I}}^{\bar{O}}$                ▷ Construction of reduced adjacency matrix
 3:     for $v \in \bar{O}$                              ▷ For all measured vertices
 4:         if $\lambda(v) \in \{Z, YZ, XZ\}$
 5:             $M_{v,*}$ *= 0              ▷ Multiply rows of $Z, YZ, XZ$ measurements by 0
 6:         if $\lambda(v) \in \{Y, Z, YZ, XZ\} \wedge v \notin I$
```

[4]Over real or complex numbers, there exists a bound $\Omega\left(n^2 \log_2 n\right)$, but it does not apply to finite fields (see [50, Section 28.2]).

```
7:              $M_{v,v} = 1$ ▷ Set intersections of rows and columns for $Y, Z, YZ, XZ$ measurements to 1
8:      return $M$

    Returns the order-demand matrix of a given labelled open graph
9:  procedure ORDER-DEMANDMATRIX($G, I, O, \lambda$)
10:     $N = Adj_G\,|_{\bar{I}}^{\bar{O}}$          ▷ Construction of reduced adjacency matrix
11:     for $v \in \bar{O}$
12:         if $\lambda(v) \in \{X, Y, Z, XY\}$
13:             $N_{v,*} \mathrel{*}= 0$        ▷ Multiply rows of $XY$ and Pauli measurements by 0
14:         if $\lambda(v) \in \{XY, XZ\} \wedge v \notin I$
15:             $N_{v,v} = 1$ ▷ Set intersections of rows and columns for $XY, XZ$ measurements to 1
16:     return $N$
```

Of course, given a labelled open graph, these matrices take only $O(n^2)$ time to produce.

6.2 Equal number of inputs and outputs

Suppose, that we are given labelled open graph $\Gamma = (G, I, O, \lambda)$ with $n_I = n_O$, i.e. $|I| = |O|$. Then, the flow-demand matrix is square, and Corollary 6.1.1 gives rise to a simple and efficient algorithm for Pauli flow detection.

Theorem 6.2.1. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph with $n_I = n_O$. Then, there exists an $O(n^3)$ algorithm which either finds a Pauli flow or determines that no Pauli flow exists.

Proof. Proceed as follows:

1. Construct the flow-demand matrix M according to Definition 5.2.7.
2. Construct the order-demand matrix N according to Definition 5.2.9.
3. Check if M is invertible.
    - If not, return that there is no flow.
4. Otherwise, compute the unique inverse C of M.

5. Compute the matrix product NC.

6. Check if NC is a DAG.

   - If not, return that there is no flow.

7. Otherwise, return the tuple (C, NC): these form the matrix encodings of the focused extensive correction function c and the corresponding induced relation $\lhd_c$.

The above procedure correctly finds flow: by Corollary 6.1.1, any focused extensive correction matrix C must be a right inverse of M. However, since M is square, any right inverse is a unique two-sided inverse. Then, the only other condition that must be checked is whether NC forms a DAG, as in the procedure above.
To see that the complexity of the above procedure is $O(n^3)$, consider:

- Steps 1–2 can be implemented in $O(n^2)$, i.e. in time linear in the size of the constructed matrices.

- Steps 3–4 can be achieved by performing Gaussian elimination, which runs in $O(n^3)$ time.

- Step 5 requires matrix multiplication, with a naïve implementation again taking $O(n^3)$ time.

- Checking if a graph is a DAG can be done in $O(n^2)$ by running depth-first search algorithm or similar (see [50, Section 20.4]).

Thus, the total runtime is dominated by Gaussian elimination and matrix multiplication running in $O(n^3)$, with the remaining steps running in $O(n^2)$. If one requires the full partial order $\prec_c$ rather than only $\lhd_c$, the transitive closure of $\lhd_c$ must be computed. This again can be done in $O(n^2)$ time, so it does not affect the overall complexity. □

We provide pseudocode that corresponds to this theorem:

Algorithm 2 Finding flow when $n_I = n_O$

```
    Checks if a labelled open graph $\Gamma = (G, I, O, \lambda)$ with $n_I = n_O$ has Pauli flow
    and, if it exists, returns the focused extensive correction function $c$ (in the form
    of a matrix) and the corresponding $\lhd_c$ (also in the form of a matrix)
1:  procedure FindFlowSimple($G, I, O, \lambda$)
```

```
2:    $M$ = FLOW-DEMANDMATRIX$(G, I, O, \lambda)$
3:    $N$ = ORDER-DEMANDMATRIX$(G, I, O, \lambda)$
4:    $C$ = INVERSE$(M)$
5:    **if** $C$ is **None**
6:        **return** ‘NO FLOW EXISTS’
7:    $R = NC$
8:    **if** $R$ is not a DAG
9:        **return** ‘NO FLOW EXISTS’
10:   **return** $(C, R)$
```

The above theorem already gives us a faster method for finding extended gflow and Pauli flow than previously known procedures: $O(n^4)$ from [15] for extended gflow and $O(n^5)$ from [153] for Pauli flow. Yet, it may still be possible to further speed up our algorithm. We discuss it further in Section 6.5.

6.3 Different number of inputs and outputs

If $n_O < n_I$, then the flow-demand matrix has more rows than columns and hence it never has a right inverse, so there is no flow. Thus, we only need to consider the case $n_I < n_O$, where the situation is more complex. The lower bound proved in Theorem 6.1.2 applies. We also still obtain a $O(n^3)$ algorithm for flow detection and finding. However, the algorithm is no longer as simple as computing the inverse and verifying whether one matrix product is a DAG.

While the focused correction matrix must be a right inverse of the flow-demand matrix, this right inverse is not unique. We modify the flow-finding algorithm from the previous section to identify a right inverse that yields a DAG when multiplied by the order-demand matrix or conclude that no such right inverse exists. Firstly, we perform a change of basis that gives a more convenient parametrisation of the right inverses. Secondly, we use an approach similar to the one from [126], which was subsequently extended in [15, 153] – we define the correction function in layers of the partial order, starting with the last layer and moving towards the first vertices in the order. In the first step, we define correction sets for all vertices that are final in the order, i.e. vertices v such that for all vertices u we have $\neg v \prec u$. Next, we find correction sets for vertices that can only be succeeded by vertices from the first layer et cetera. If, at any point, we cannot correct any vertices, there is no flow. The methods used in [126, 15] required Gaussian elimination for each layer of vertices, while in [153] up to $O(n)$ Gaussian eliminations per layer were necessary. Since the number of layers could be $O(n)$, it leads to upper bounds of, at best, $O(n^4)$. We improve on this approach by reducing the number of operations performed at each layer to $O(n^2 s)$ where s is the number of vertices in the layer. This leads to a total cost of $O(n^3)$ operations.

6.3.1 Algorithm and correctness proof

The following theorem and its proof describe the Pauli flow-finding algorithm in prose and explain its functioning as well as its complexity. A pseudocode presentation of the same algorithm may be found in Subsetion 6.3.2.

Theorem 6.3.1. Let $\Gamma = (G, I, O, \lambda)$ be a labelled open graph with $n_I \leq n_O$, i.e. $|I| \leq |O|$. Then, there exists an $O(n^3)$ algorithm which either finds a Pauli flow or determines that no Pauli flow exists.

Proof. The proof consists of providing an algorithm, showing its correctness, and verifying the complexity bound. We interleave the three parts: for each step of the algorithm, we first provide the procedure and then explain it and analyse its complexity.

1. Construct the flow-demand matrix M according to Definition 5.2.7. Recall this matrix has size $(n - n_O) \times (n - n_I)$.

2. Construct the order-demand matrix N according to Definition 5.2.9.

3. Check whether M is right-invertible, i.e. whether rank $M = n - n_O$.

   - If M is not right-invertible, return that there is no flow.

   Explanation of the step: By Corollary 6.1.1, flow existence necessarily means that M is right-invertible. M is right-invertible if all of its rows are linearly independent, i.e. if rank M equals the number of rows of M, that is $n - n_O$. The check for right invertibility can be achieved by performing Gaussian elimination, which runs in time $O(n^3)$.

4. Find any right inverse of M and call it C_0.

   Explanation of the step: By the previous step, a right inverse exists. It can be found for instance by Gaussian elimination and backtracking, again in $O(n^3)$ and has size $(n - n_I) \times (n - n_O)$.

   Even if there is a Pauli flow, the matrix C_0 will not necessarily be such that NC_0 is a DAG. Yet by combining it with information about the kernel of M, we will be able to parametrise all the right inverses of M: Any right inverse of M must necessarily be of the form C_0+F' where F' is a matrix whose columns are vectors in $\ker M$. Such matrices work as then $M(C_0+F') = Id_{\bar{O}}+\mathbf{0} = Id_{\bar{O}}$. They are also necessary, as if $MC = Id_{\bar{O}}$, then $F' = C - C_0$ must satisfy both $C = C_0 + F'$ and $\mathbf{0} = MC - MC_0 = M(C - C_0) = MF'$.

In the next three steps, we will first find the kernel of M and then use it to perform a change of basis, that gives a simpler form to the possible right inverses. This simplifies the search for a right inverse that satisfies all the Pauli flow conditions.

5. Find a matrix F whose columns form a basis of $\ker M$.

   Explanation of the step: This can again be found with Gaussian elimination. By the rank-nullity theorem, the dimension of $\ker M$ is $n_O - n_I$, hence F has size $(n - n_I) \times (n_O - n_I)$.

6. Let $C' = [C_0 \mid F]$.

   Explanation of the step: C' is a square matrix of size $(n - n_I) \times (n - n_I)$. We know $MC_0 = Id$, so the columns of C_0 form a basis of $\text{supp}\,(M)$. Similarly, the columns of F form a basis of $\ker M$. Thus, the union of the columns of both forms a basis of $\mathbb{F}^{n-n_I}$, i.e. the columns of C' are linearly independent. Therefore, $\text{rank}\, C' = n - n_I$ and C' is invertible. From now on, we will use diagrams to illustrate the sizes of the matrices we work with. When the rows or columns correspond to sets of vertices, we also indicate them on the diagrams.

$$M_{\mathcal{B}} = \begin{array}{|c|c|l} \multicolumn{1}{c}{\bar{O}} & \multicolumn{1}{c}{n_O - n_I} & \\ \hline Id_{\bar{O}} & \mathbf{0} & \bar{O} \\ \hline \end{array} \qquad\qquad C' = \begin{array}{|c|c|l} \multicolumn{1}{c}{\bar{O}} & \multicolumn{1}{c}{n_O - n_I} & \\ \hline C_0 & F & \bar{I} \\ \hline \end{array}$$

7. Compute $N_{\mathcal{B}} = NC'$.

   Explanation of the step: As explained in Step 4, the right inverse C_0 is not necessarily unique. However, we know the form of all possible right inverses, which can be parametrised in terms of C_0 and the columns of F.

The problem is to find some right inverse C such that NC is a DAG. Brute force checking all possible right inverses cannot be performed, as the number of right inverses grows exponentially in $n_O - n_I$. Instead, we simplify the form of right inverses by the basis change just performed.

To see this, we note that the equivalent basis change on M yields $M_{\mathcal{B}} = MC' = [MC_0 \mid MF] = \left[Id_{\bar{O}} \mid \mathbf{0}\right]$, since C_0 is the right inverse of M and F is the basis of $\ker M$. This makes all right inverses of $M_{\mathcal{B}}$ very simple: their first $n - n_O$ rows form $Id_{\bar{O}}$ and the remaining rows can contain any values. In the following picture, the part of the matrix that can have any values is denoted P.

$$M_{\mathcal{B}} = \begin{array}{c|c|c|l} \multicolumn{1}{c}{\bar{O}} & \multicolumn{1}{c}{n_O - n_I} & \\ \hline Id_{\bar{O}} & \mathbf{0} & \}\,\bar{O} \\ \hline \end{array} \qquad\qquad C^{\mathcal{B}} = \begin{array}{|c|l} \multicolumn{1}{c}{\bar{O}} & \\ \hline Id_{\bar{O}} & \}\,\bar{O} \\ \hline P & \}\,n_O - n_I \\ \hline \end{array} \tag{6.1}$$

Now, suppose that we find $C^{\mathcal{B}}$ such that $M_{\mathcal{B}}C^{\mathcal{B}} = Id_{\bar{O}}$ and $N_{\mathcal{B}}C^{\mathcal{B}}$ is a DAG. Then, we can find the desired correction matrix C encoding a focused extensive correction function c by computing $C'C^{\mathcal{B}}$, since:

$$\begin{aligned} M_{\mathcal{B}}C^{\mathcal{B}} &= (MC')C^{\mathcal{B}} = M(C'C^{\mathcal{B}}) \\ N_{\mathcal{B}}C^{\mathcal{B}} &= (NC')C^{\mathcal{B}} = N(C'C^{\mathcal{B}}) \end{aligned}$$

and so $M(C'C^{\mathcal{B}}) = Id_{\bar{O}}$ and $N(C'C^{\mathcal{B}})$ is a DAG.

This approach also captures any possible solution as given a working C one can find a working $C^{\mathcal{B}}$ by computing $(C')^{-1}\,C$. This is to say, that instead of solving the problem given M and N, we can solve the problem given $M_{\mathcal{B}}$ and $N_{\mathcal{B}}$.

8. Define N_L and N_R as the submatrices of $N_{\mathcal{B}}$ given by first $n - n_O$ columns

and the remaining $n_O - n_I$ columns:

$$N_{\mathcal{B}} = \left.\begin{array}{|c|c|} \multicolumn{1}{c}{\overbrace{\hphantom{N_L}}^{\bar{O}}} & \multicolumn{1}{c}{\overbrace{\hphantom{N_R}}^{n_O - n_I}} \\ \hline N_L & N_R \\ \hline \end{array}\right\}\bar{O}$$

Explanation of the step: From the condition $M_{\mathcal{B}}C^{\mathcal{B}} = Id_{\bar{O}}$, we already know the first $n - n_O$ rows of $C^{\mathcal{B}}$. This means, that we are able to express the product $N_{\mathcal{B}}C^{\mathcal{B}}$ as follows:

$$N_{\mathcal{B}}C^{\mathcal{B}} = N_L Id_{\bar{O}} + N_R P = N_L + N_R P$$

Thus, the problem becomes to find a $(n_O - n_I) \times (n - n_O)$ matrix P such that $N_L + N_R P$ is a DAG:

$$\boxed{N_L} + \boxed{N_R}\ \boxed{P}$$

To find P, we will use a layer-by-layer approach, as in many existing flow-finding algorithms. This means we first identify vertices that are maximal in the partial order, and then work 'down the order' from there. A naïve version of the layer-by-layer approach would be as follows.

- Take the system of linear equations $[N_R \mid N_L]$, where N_R are the coefficients and N_L are the attached vectors, i.e. the desired values. Recall the columns of N_L are labelled by non-output vertices, as are the columns of P.

  This means we can alternatively consider $[N_R \mid N_L]$ as a collection of $n - n_O$ independent linear systems, one for each $v \in \bar{O}$, where each system has the same coefficients for the unknowns but generally different constants (and thus different solutions). Each of these systems consists of $n - n_O$ equations in $n_O - n_I$ unknowns. For most interesting computations, we expect the labelled open graph to be dominated by internal vertices, i.e. $V \setminus (I \cup O)$ is larger than $I \cup O$. In that case, the linear systems are generally overdetermined.

Indeed, if all linear systems are solvable, then $N_L + N_R P$ is the all-zero matrix: a trivial DAG. Yet, if a flow exists, there must be vertices that are maximal in the associated partial order. The columns corresponding to those vertices in the adjacency matrix of the DAG are all-0s: thus these vertices must be associated with solvable linear systems.

- Perform Gaussian elimination to find the set L of vertices whose associated system of linear equations is solvable. (If none exist, there can be no flow.) These are the vertices that are not followed by any not-yet-solved vertices in the partial order represented by the DAG $NC = N_{\mathcal{B}} C^{\mathcal{B}}$.
- For each vertex $v \in L$, remove the linear system associated with v from future consideration, i.e. ignore the column labelled v in N_L and in P. Recall that the rows of N_L and N_R are also labelled by non-output vertices. Knowing that v is maximal in the partial order among the remaining vertices, we can now remove the row v from consideration in each linear system, eliminating one equation given by N_R coefficients. The removal of the equation means that it is now possible to get a different number than initially required for the equation corresponding to the removed vertex. In other words, it is now possible to add to the partial order a constraint expressing that a vertex solved later precedes some vertex that was solved in earlier steps.

  If there remain systems of linear equations that have not been solved yet, go back to the previous step to look for a new set of vertices whose associated linear systems have become solvable.

This approach is somewhat similar to the ones in [126, 15, 153] (except that, there, larger matrices must be considered in each step). Since we only have to Gaussian eliminate with respect to the columns of N_R (of which there are $n_O - n_I$), we get an upper bound for the naïve approach of $O(n^3(n_O - n_I))$. However, we can do better by noticing that the matrices constructed for subsequent layers are very similar, and thus we can 'reuse' previous Gaussian elimination steps.

We thus continue with the more involved version of the layer-by-layer algorithm.

9. Construct two independent[a] copies K_{LS}, K_{ILS} of the following matrix.

$$\begin{array}{ccc}
\overbrace{\qquad\qquad}^{n_O - n_I} & \overbrace{\qquad\qquad\qquad}^{\bar{O}} & \overbrace{\qquad\qquad\qquad}^{\bar{O}} \\
\end{array}$$
$$\left.\begin{array}{|c|c|c|}
\hline
N_R & N_L & Id_{\bar{O}} \\
\hline
\end{array}\right\}\bar{O}$$

Explanation of the step: The first block of the system forms the coefficients of the system of linear equations. The second block consists of the attached vectors, i.e. the desired values for which we want to solve the system. Finally, the third block starts as the identity and will be used to keep track of Gaussian elimination steps throughout the process.

K_{LS} stands for 'linear system' and K_{ILS} for 'initial linear system'.

10. Perform Gaussian elimination on K_{LS} (with respect to the first $(n_O - n_I)$ columns).

Explanation of the step: The idea of the third block, i.e. initially the identity matrix, is to know which rows from the initial system K_{ILS} are used in each of the rows of K_{LS}, i.e. the performed row operations can be 'recovered' by reading the third block. The complexity of Gaussian elimination in this step is $O\left(n^2(n_O - n_I)\right)$, as we effectively have to perform Gaussian elimination of an $(n - n_O) \times (n_O - n_I)$ matrix (all rows, but only the first $n_O - n_I$ columns), yet the cost of each row operation is $(n_O - n_I) + (n - n_O) + (n - n_O) = 2n - n_O - n_I \in O(n)$, as this is the actual size of each row.

11. Define the (initially empty) set of already solved vertices $S = \emptyset$ and initialize the matrix of the solutions P as an (initially empty) $(n_O - n_I) \times (n - n_O)$ matrix with columns corresponding to non-outputs $\bar{O}$.

12. Repeat the following loop of substeps until $S = \bar{O}$, i.e. all vertices are solved. We will refer to this loop as the '`while` loop'.

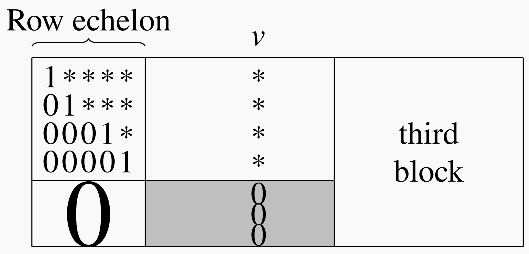
Explanation of the step: Each run of the `while` loop corresponds to solving a single layer of vertices. The loop invariant is that the first $(n_O - n_I)$ columns of K_{LS} are in row echelon form (but not necessarily row-reduced echelon form). The loop invariant holds at the beginning by Step 10.

(a) Identify a set L of vertices $v \in \bar{O} \setminus S$ whose associated system of linear equations can be solved at this step.

- If the set L turns out to be empty, return that there is no flow.

Explanation of the step: This can be done in $O(n^2)$, as K_{LS} is in row echelon form:

- The system of linear equations associated with a vertex v can be solved if it has not been solved yet and if its column in the second block (which determines the constants in each equation) has only zeros where it intersects rows for which all the coefficients in the first block are 0s:



Here, $*$ denotes matrix entries of arbitrary value.

- This means, the submatrix marked in grey must be checked for all-0 columns. Checking this submatrix takes $O(n^2)$ as its size is bounded above by $n \times n$.

If there remain vertices under consideration but we cannot solve any of the associated systems of linear equations, then Pauli flow cannot exist.

(b) Solve the linear systems associated with all vertices $v \in L$.

Explanation of the step: Since the matrix is in row echelon form, finding a solution for the linear system associated with v is fast and is done by backtracking over the coefficient matrix (i.e. the first block) and the column of the solved vertex.

(c) For each $v \in L$, place the solution to the system of linear equations associated with v in the v column of P.

Explanation of the step: Solving the system of linear equations associated with one vertex takes at most $O\left(n(n_O - n_I)\right)$ steps due to row echelon form, as there are only $n_O - n_I$ columns of the coefficient block. Hence the cost per layer (i.e. per round of the loop) of this step is

$O\left(ns(n_O - n_I)\right)$, where s is the number of vertices in the layer, i.e. $s = |L|$.

(d) Bring the linear system K_{LS} to the row echelon form that would be achieved by Gaussian elimination if the row and column vectors corresponding to vertices in L where not included in the starting matrix.

Explanation of the step: This is the step where we remove certain rows and columns from future consideration. In the case of columns, these are simply columns of constants: we ignore those systems of linear equations that have already been solved. In the case of rows, this involves setting an entire row of coefficients to 0 in a specific way. This step may therefore break row echelon form.

To remove the rows and columns, we iterate over all vertices in L; we will refer to this inner loop as the '`for` loop'. Each vertex appears in L during at most one round of the outer `while` loop. This means that we can spend $O(n^2)$ operations per vertex on modifying the set of systems of linear equations, while staying within the overall limit of $O(n^3)$.

If one layer (i.e. one round of the `while` loop) contains a substantial part of $\bar{O}$, then working out this layer in the `while` loop may take a number of operations which is not in $O(n^2)$. However, this does not increase the overall complexity to $O(n^4)$, as the cost for each vertex is still $O(n^2)$ and each vertex appears only once.

The process of modifying the systems of linear equations is performed as follows.

i. Let $v \in L$ be the vertex currently considered in the `for` loop. Add v to the set S.
   Explanation of the step: The addition to set S ensures that a vertex will not be solved for a second time. Next, we must remove the v row from all the systems of linear equations.

ii. Find all the rows of the linear system which contain a 1 in the v column of the third block, denote them by $r_1, r_2, \ldots, r_k$.
   Explanation of the step: These are the rows that depend on the original v-labelled row. They can be identified by iterating over the v column in the third block, which gives a complexity of $O(n)$.

iii. Add row r_k to rows $r_1, r_2, \ldots, r_{k-1}$.
   Explanation of the step: This removes the dependence on the orig-

inal v-labelled row from $r_1, r_2, \ldots, r_{k-1}$, meaning only r_k now depends on the original v row.

By the loop invariant, the system is in row echelon form initially. Adding a later row to earlier rows preserves row echelon form.

The additions take $O(nk)$ steps, as there are at most $O(n)$ columns in the system, and $k - 1 \in O(k)$ row operations are performed. Since k is bounded by the number of rows, we get $O(nk) \in O(n^2)$.

At the end, the matrix is in row echelon form, and only row r_k uses the original row of vertex v anywhere.

iv. Take the v-labelled row from K_{ILS}, the initial linear system, and add this row to r_k.

Explanation of the step: In this way, we remove the dependency of r_k on the initial row of v.

This operation can break row echelon form. Crucially, however, only r_k can break the row echelon form. We can correct this one row in the following step.

v. Add other rows of the current linear system K_{LS} to row r_k to simplify the latter as much as possible.

Explanation of the step: In particular, the row r is added to r_k when r_k has a 1 in the column corresponding to the leading 1 of row[b] r. These are the same operations that one would perform in Gaussian elimination, except that we do not use r_k itself to cancel 1s in other rows, we only use other rows to cancel 1s in r_k. This is because we only need the coefficient block to be in row echelon form (not row-reduced echelon form).

Hence, we need to perform at most $n_O - n_I$ row operations, as this is the maximum possible number of rows that would need to be added to row r_k. Hence, this step is bounded by $O\left(n(n_O - n_I)\right)$.

vi. Swap the rows as necessary to place what used to be the row r_k in the correct spot to get a row echelon form for the coefficient block.

Explanation of the step: There are at most $n_O - n_I$ non-zero rows in the coefficient block of the matrix. This is due to row-echelon form and the rank of the coefficient submatrix being bounded by its number of columns. Moreover, at most one swap will involve a row whose coefficient part is identically 0: if r_k is not identially 0 but it is initially in the block of all-0 rows, then a single swap suffices to place it adjacent to the block of non-trivial rows. Whether r_k is identically 0 or not, all subsequent swaps are done only between non-trivial rows and row r_k. Hence, at most $O\left(n(n_O - n_I)\right)$ operations are required. After that, the K_{LS} is in the row echelon form that would be achieved if there was no v row in the initial system (or, more precisely, if the v row of the initial system was identically 0).

Combining the steps above and assuming a layer of size s, the total run time of the process for modifying the system of linear equations is $O(n^2 s)$, as desired.

Since each vertex appears in at most one layer (i.e. one iteration of the `while` loop), and each substep has a complexity at most $O(n^2 s)$, the entire `while` loop runs in total time $O(n^3)$.

13. Construct $C^{\mathcal{B}}$ by stacking $Id_{\bar{O}}$ over P.

    Explanation of the step: Once the `while` loop has been completed, we know that all vertices have been solved. Thus, a matrix P such that $N_L + N_R P$ is a DAG has been found, and we get $C^{\mathcal{B}} = \left[\frac{Id_{\bar{O}}}{P}\right]$, cf. the illustration in (6.1).

14. Return the tuple $C'C^{\mathcal{B}}, NC'C^{\mathcal{B}}$ as the Pauli flow.

    Explanation of the step: We return the correction function c in matrix form and the relation $\triangleleft_c$, also in matrix form, cf. Step 7 for why the matrices above are the correct ones. Computing and outputting the matrix products takes at most $O(n^3)$ operations, assuming the standard matrix multiplication algorithm is used.

As previously, if one requires the partial order $\prec_c$, then the transitive closure of $\triangleleft_c$ must be computed, which can be done in $O(n^2)$ steps. All steps together give the desired bound of $O(n^3)$. □

[a]By independent, we mean that changes to one do not affect the other.

[b]It suffices to only reduce r_k until a 1 that cannot be eliminated appears in r_k, or until r_k becomes identically 0, whichever is first.

We provide pseudocode for this algorithm in Subsection 6.3.2. A worked example of running the algorithm on the labelled open graph from Example 5.1.2 is given in Subsection 6.3.3.

With the above procedure, we proved that Pauli flow on a labelled open graph can be found in $O(n^3)$ time, where n is the number of vertices in the underlying graph. However, extra care in bounding the complexity of individual steps could result in a slightly better overall bound: The only situation in which a round of the inner `for` loop (cf. Step 12d) may actually require $O(n^2)$ operations, rather than $O(n(n_O - n_I))$ operations, occurs when there are many rows which depend on a vertex that has just been solved, i.e. when the value k in Step 12(d)ii is large. Therefore, it is possible that with extra care one could show only $O\left(n^2(n_O - n_I)\right)$ operations are needed for the entire `while` loop of Step 12. Then, if there are methods faster than $O(n^3)$ for finding the right inverse, kernel, and product of $O(n) \times O(n)$ matrices, a better bound than $O(n^3)$ could be obtained for the entire algorithm even in the case of $n_I < n_O$.

Another possible optimisation is to notice that the order-demand matrix N has many rows which are identically 0. After the basis change to $N_{\mathcal{B}}$, these rows are still identically 0. Therefore, removing them from the matrix at the start is beneficial: then, the algorithm has to consider a matrix with fewer rows. The rows that are always identically 0 in the order-demand matrix are those of planar-measured vertices and XY-measured inputs. Hence, the number of rows of N can be reduced from $n - n_O$ to $\left|\Lambda_{planar} \setminus I\right|$.

Finally, we point out that finding the solutions for the individual systems of linear equations associated with the vertices in L and the removal of the rows corresponding to these vertices does not have to be split into two separate loops. However, without breaking these steps into two loops, an insufficiently careful method of finding the solution could impose an order relation $v \prec u$ on two vertices from the same layer[5].

6.3.2 Pseudocode for case of different numbers of inputs and outputs

This algorithm relates to Theorem 6.3.1. We list the corresponding steps from Theorem 6.3.1 in the comments. *GE* indicates that the step can be performed using Gaussian elimination.

Algorithm 3 Finding flow in the general case

```
    Checks if a labelled open graph $\Gamma = (G, I, O, \lambda)$ has Pauli flow and, if it exists,
    returns the focused extensive correction function $c$ (in the form of a matrix)
    and the corresponding $\prec_c$ (also in the form of a matrix)
1:  procedure FindFlowGeneral($G, I, O, \lambda$)
2:      $M$ = Flow-DemandMatrix($G, I, O, \lambda$)                ▷ Step 1
3:      $N$ = Order-DemandMatrix($G, I, O, \lambda$)               ▷ Step 2
```

[5]In other words: while the flow returned by our algorithm is guaranteed to have the useful property of being 'maximally delayed' [126, 15, 153], such a change could break that guarantee.

```
 4: if rank $M \neq n - n_O$                                        ▷ GE; Step 3
 5:     return 'NO FLOW EXISTS'
 6: $C_0$ = any right inverse of $M$                      ▷ GE and backtracking; Step 4
 7: $F$ = matrix with columns forming basis of ker $M$               ▷ GE; Step 5
 8: $C' = [C_0 \mid F]$                                                   ▷ Step 6
 9: $N_{\mathcal{B}} = NC'$                                               ▷ Step 7
10: $N_L$ = first $n - n_O$ columns of $N_{\mathcal{B}}$                 ▷ Step 8
11: $N_R$ = last $n_O - n_I$ columns of $N_{\mathcal{B}}$
12: $K_{ILS}, K_{LS} = \left[N_R \mid N_L \mid Id_{\bar{O}}\right]$ ▷ Two independent copies of the same linear
    system; Step 9
    ▷ From now, we refer to the three parts of $K_{ILS}$ and $K_{LS}$ as the first block,
    the second block, and the third block
13: bring first block of $K_{LS}$ into row echelon form              ▷ GE; Step 10
14: $S = \emptyset$                                                       ▷ Step 11
15: initialize $(n_O - n_I) \times (n - n_O)$ matrix $P$ ▷ Columns of P correspond to non-
    outputs $\bar{O}$
16: while $S \neq \bar{O}$ :                                               ▷ Step 12
17:     find the first row $r_z$ in $K_{LS}$ whose first $n_O - n_I$ entries equal 0
                                                                         ▷ Step 12a
18:     $L$ = set of those $v \in \bar{O} \setminus S$ that column $v$ in the first $K_{LS}$ block has all
        entries from row $r_z$ on equal to 0
19:     if $L = \emptyset$
20:         return 'NO FLOW EXISTS'
21:     for $v \in L$ :                                           ▷ Steps 12b and 12c
22:         $P_{*,v}$ = SolveLinearSystem(first block of $K_{LS}$, $v$ column in second
            block of $K_{LS}$)
23:     for $v \in L$ : ▷ Step 12d; we need two loops, see text after Theorem 6.3.1
24:         $S = S \cup \{v\}$                                             ▷ Step 12(d)i
25:         $R = [r \mid r$ is a row whose intersection with the $v$ column in the third
            block of $K_{LS}$ is 1] ▷ R is an ordered list that corresponds to
            $r_1, r_2, \ldots, r_k$ in Step 12(d)ii
26:         $r_{last}$ = last element of $R$                       ▷ corresponds to $r_k$
27:         for $r \in R[: -1]$      ▷ Iterate over all but last element; Step 12(d)iii
28:             add $r_{last}$ of $K_{LS}$ to row $r$ in $K_{LS}$
29:         add row $v$ of $K_{ILS}$ to row $r_{last}$ in $K_{LS}$          ▷ Step 12(d)iv
30:         for $r \in$ rows of $K_{LS}$ except $r_{last}$: ▷ covers Step 12(d)v; go from top
```

```
31:                 if row $r$ of $K_{LS}$ has first $n_O - n_I$ entries 0
32:                     Break
33:                 $y$ = column of the leading 1 in row $r$ of $K_{LS}$
34:                 if intersection of $r_{last}$ and column $y$ in $K_{LS}$ contains 1
35:                     add row $r$ to $r_{last}$ in $K_{LS}$
36:             swap $r_{last}$ with other rows to bring first block of $K_{LS}$ back into row
            echelon form                                                    ▷ Step 12(d)vi
37:     $C^{\mathcal{B}} = \begin{bmatrix} Id_{\bar{O}} \\ P \end{bmatrix}$                 ▷ Step 13
38:     return $(C'C^{\mathcal{B}}, NC'C^{\mathcal{B}})$                            ▷ Step 14
```

6.3.3 Worked example of the Pauli-flow finding algorithm

This subsection contains a worked example of the complicated parts of the algorithm in Theorem 6.3.1, using the running example from the main body of the paper[6].

Consider the labelled open graph Γ from Example 5.1.2. The flow-demand matrix M and order-demand matrix N are presented in Examples 5.2.8 and 5.2.10. Suppose we do not yet know the correction function of Example 5.1.2 or the matrix C from Example 5.2.6. Instead, suppose we have found some right inverse C_0 of M as well as a matrix F whose columns form a basis of $\ker M$. At Step 6 of the algorithm, these are composed into the matrix $C' = [C_O \mid F]$ shown below:

$$C' = [C_0 \mid F] = \begin{array}{c|ccccc||c|} & i & a & b & e & d & F_1 \\ \hline a & 0 & 1 & 0 & 0 & 0 & 0 \\ b & 1 & 0 & 0 & 0 & 0 & 0 \\ e & 1 & 0 & 1 & 0 & 1 & 1 \\ d & 0 & 0 & 0 & 0 & 1 & 0 \\ o_1 & 1 & 0 & 0 & 1 & 1 & 1 \\ o_2 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

In general, F will have $n_O - n_I$ columns. Here, this number is one. In the following, we use F_1 as the label for the single column of the matrix F.

[6]Due to its length, I have opted not to enclose this example in an environment.

Then, for Step 7, the matrices $M_{\mathcal{B}} = MC'$ and $N_{\mathcal{B}} = NC'$ are as follows[7]:

$M_{\mathcal{B}}$	i	a	b	e	d	F_1
i	1	0	0	0	0	0
a	0	1	0	0	0	0
b	0	0	1	0	0	0
e	0	0	0	1	0	0
d	0	0	0	0	1	0

$N_{\mathcal{B}}$	i	a	b	e	d	F_1
i	0	0	0	0	0	0
a	0	1	0	0	1	1
b	0	0	0	0	0	0
e	1	0	1	0	1	1
d	0	0	0	0	0	0

In Steps 8 and 9, we break $N_{\mathcal{B}}$ into N_L and N_R, and obtain the following linear systems K_{LS}, K_{ILS}. We give columns from the third block primed labels, to distinguish them from the columns in the second block:

$K_{ILS} = K_{LS} =$

	F_1	i	a	b	e	d	i'	a'	b'	e'	d'
i	0	0	0	0	0	0	1	0	0	0	0
a	1	0	1	0	0	1	0	1	0	0	0
b	0	0	0	0	0	0	0	0	1	0	0
e	1	1	0	1	0	1	0	0	0	1	0
d	0	0	0	0	0	0	0	0	0	0	1

In Step 10, we apply Gaussian elimination to the first $n_O - n_I$ columns, i.e. in this case to the first column only. This means adding the second row to the fourth and then swapping the first two rows. We drop previous row labels, as they no longer make sense after performing row operations. (Nevertheless, the relationship to the original vertex-labelled rows continues to be encoded in the third block.) We obtain the following updated K_{LS} linear system:

	F_1	i	a	b	e	d	i'	a'	b'	e'	d'
1	1	0	1	0	0	1	0	1	0	0	0
2	0	0	0	0	0	0	1	0	0	0	0
3	0	0	0	0	0	0	0	0	1	0	0
4	0	1	1	1	0	0	0	1	0	1	0
5	0	0	0	0	0	0	0	0	0	0	1

(6.2)

Importantly, K_{ILS} remains unchanged. Next, in Step 11, we initialize the matrix of

[7]The matrix $M_{\mathcal{B}}$ is not required for the algorithm, we provide it here for completeness.

solutions P and the set of solved vertices S:

$P =$

	i	a	b	e	d
F_1	$*$	$*$	$*$	$*$	$*$

$S = \emptyset$

Recall the coefficient block of the system (i.e. just the first column) is in row echelon form as a result of Step 10. We now go into the `while` loop of Step 12.

In the first loop run, we can identify e and d as the vertices whose corresponding linear systems can be solved (Step 12a). This is because the coefficient block is 0 from the second row onwards and e, d are the only vertices whose columns are also 0 from the second row onwards. It means that $L = \{e, d\}$. We find the solutions of the associated linear systems (Step 12b): for e we must set the variable corresponding to the F_1 column to 0 and for d it must necessarily be 1. Thus the updated matrix P looks as follows (Step 12c):

	i	a	b	e	d
F_1	$*$	$*$	$*$	0	1

Now, we must transform the system of linear equations to the form it would have if e and d had never been included (Step 12d).

Consider e first. In Step 12(d)ii, we find which rows of the linear system in (6.2) use the original row e from the e' column – in this case, it is only the fourth row. Thus the fourth row is also the last row using the original e row. (If other rows were using e, we would add the fourth row to all other rows using e to remove their dependency on the original e row in Step 12(d)iii.) Next, in Step 12(d)iv, we add the e row of K_{ILS} to the fourth row of (6.2), obtaining the following system (importantly, this addition breaks the row echelon form in the coefficient block):

	F_1	i	a	b	e	d	i'	a'	b'	e'	d'
1	1	0	1	0	0	1	0	1	0	0	0
2	0	0	0	0	0	0	1	0	0	0	0
3	0	0	0	0	0	0	0	0	1	0	0
4	1	0	1	0	0	1	0	1	0	0	0
5	0	0	0	0	0	0	0	0	0	0	1

(6.3)

After that, in Step 12(d)v, we bring the coefficient block back into row echelon form: we cancel as much in the coefficient block of the fourth row as possible: here, we add the first row of (6.3) to the fourth row, obtaining the following linear system:

	F_1	i	a	b	e	d	i'	a'	b'	e'	d'
1	1	0	1	0	0	1	0	1	0	0	0
2	0	0	0	0	0	0	1	0	0	0	0
3	0	0	0	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	1

(6.4)

The coefficient block of the matrix is in row echelon form, so we do not have to perform any other operations. In particular, we do not need to swap any rows in Step 12(d)vi, though it might be necessary in general.

Next, we move to vertex d. Here the situation is even simpler. From column d' of (6.4), we read that only the fifth row uses the original d row. After adding the d row from K_{ILS} to the fifth row (Step 12(d)iv), we get the following system, where the coefficient block immediately is in row echelon form:

	F_1	i	a	b	e	d	i'	a'	b'	e'	d'
1	1	0	1	0	0	1	0	1	0	0	0
2	0	0	0	0	0	0	1	0	0	0	0
3	0	0	0	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0

(6.5)

While dealing with e and d, in Step 12(d)i, we also add them to the set of solved vertices S, i.e. $S = \{e, d\}$. This ends the first loop run.

In Step 12a of the second loop run, we observe that i, a, and b can all be solved. For i and b the value corresponding to variable F_1 must be 0, and for a it must necessarily be 1 (Step 12b). It leads to the following matrix P (Step 12c):

	i	a	b	e	d
F_1	0	1	0	0	1

While this is no longer necessary to find a solution, we nevertheless also show how the linear systems associated with these vertices would be removed from consideration here as this is what the algorithm will do. First, we ‘remove’ the row of i in Steps 12(d)ii–12(d)vi. From the i' column of (6.5), we know that only the second row uses the i row from K_{ILS}.

Adding the i row from K_{ILS} to the second row of the linear system in (6.5), we obtain:

	F_1	i	a	b	e	d	i'	a'	b'	e'	d'
1	1	0	1	0	0	1	0	1	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0

(6.6)

The coefficient block is in row echelon form. We move to vertex a. From the a' column, we read that only the first row uses the a row from K_{ILS}. We thus add the a row from K_{ILS} to the first row in (6.6), obtaining:

	F_1	i	a	b	e	d	i'	a'	b'	e'	d'
1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0

(6.7)

The coefficient block is in row echelon form, so we move to b. Again, we find that only one row of (6.7) – the third row – uses the original b row. After adding the b row from K_{ILS} to the third row of the linear system in (6.7), we get:

	F_1	i	a	b	e	d	i'	a'	b'	e'	d'
1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0

In the process, at Step 12(d)i, vertices i, a, b are added to set S. Hence, $S = \{i, a, b, e, d\}$ and the loop ends here.

Finally, in Step 13, we construct the matrix $C^{\mathcal{B}} = \left[\frac{Id_{\bar{O}}}{P}\right]$, i.e.:

$$C^{\mathcal{B}} = \begin{array}{|c|ccccc|} \hline & i & a & b & e & d \\ \hline i & 1 & 0 & 0 & 0 & 0 \\ a & 0 & 1 & 0 & 0 & 0 \\ b & 0 & 0 & 1 & 0 & 0 \\ e & 0 & 0 & 0 & 1 & 0 \\ d & 0 & 0 & 0 & 0 & 1 \\ \hline F_1 & 0 & 1 & 0 & 0 & 1 \\ \hline \end{array}$$

We finish the computation with Step 14 by computing the correction matrix $C = C'C^{\mathcal{B}}$ encoding the correction function, and the product $NC = NC'C^{\mathcal{B}}$ encoding the induced relation. The two matrices are the same as those shown in Examples 5.2.6 and 5.2.10.

6.4 Flow and unlabelled open graphs

This section provides one additional algorithm arising from the algebraic interpretation. We show that given an open graph, it is possible to decide (in polynomial time) the existence of measurement labelling, resulting in a labelled open graph with Pauli flow. Furthermore, we show that one can efficiently find such measurement labelling when it exists. These results contribute to the understanding of the necessary properties the graph must satisfy to contain flow. Such properties have also been considered, not from an algorithmic perspective, for instance in [125, 120, 153]. The presented algorithm also complements the Pauli flow finding algorithm by considering the flow on an input without explicit measurement labelling.

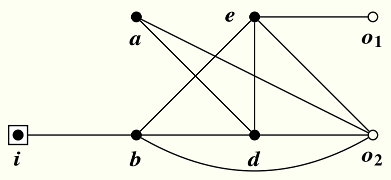
6.4.1 Problem statement

We consider the following problem:

MeasurementSearch
Input: An unlabelled open graph (G, I, O).
Output: *True* if there exists a measurement labelling λ such that the labelled open graph (G, I, O, λ) has Pauli flow, and *False* otherwise.

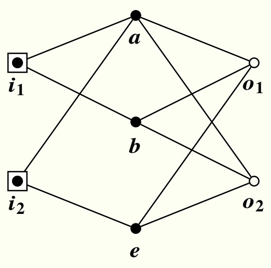
By unlabelled open graph, we mean an open graph. The word unlabelled is added to avoid confusion about the input structure. The flows are defined over labelled open graph.

Example 6.4.1. Consider the following open graph:



The answer to **MeasurementSearch** on this open graph is *True*, as the measurement labelling from Example 5.1.2 results in a labelled open graph with Pauli flow.

Example 6.4.2. Consider the following open graph:



It cannot have flow under any measurement labelling. By Theorem 5.6.2, it is sufficient to consider Pauli measurements. Informally, observe that a, b, and e form a complete bipartite graph with outputs. Thus, it would be impossible to choose suitable corrections for those vertices unless at least two of them are measured in Z basis, in which case the inputs i_1 and i_2 cannot be corrected. We skip the details.

6.4.2 Variable flow-demand matrix

While we skipped proof details in Example 6.4.2, the essential part was the use of Theorem 5.6.2: if there was some measurement labelling resulting in Pauli flow, then there would be one with just Pauli measurements. Hence, we can restrict the search to Pauli-only labels instead of looking for any measurement label.

By Observation 5.6.3, the order-demand matrix is identically zero when all measurements are Pauli. Combining this fact with Theorem 5.2.1, we find that Pauli-only measurement labelling results in Pauli flow if and only if the corresponding flow-demand matrix is right invertible.

As it turns out, it is possible to construct a matrix with variables that, after different valuations, corresponds to all possible flow-demand matrices:

Definition 6.4.3. Let (G, I, O) be an open graph. We define *variable flow-demand matrix* $W_{G,I,O}$ as follows:

- Start with the reduced adjacency matrix of (G, I, O), i.e. set $W_{G,I,O} := Adj_G |_{\bar{O}}^{\bar{I}}$.
- For each $v \in \bar{O}$, multiply v row of $W_{G,I,O}$ by x_v.
- For each $v \in \bar{O}$, set the intersection of v row and v column in $W_{G,I,O}$ to z_v.

Consider the following example:

Example 6.4.4. Let (G, I, O) be the open graph from Example 6.4.2. The corresponding reduced adjacency matrix is:

$Adj_G \|_{\bar{O}}^{\bar{I}}$	a	b	e	o_1	o_2
i_1	1	1	0	0	0
i_2	1	0	1	0	0
a	0	0	0	1	1
b	0	0	0	1	1
e	0	0	0	1	1

and the corresponding variable flow-demand matrix is:

W	a	b	e	o_1	o_2
i_1	x_{i_1}	x_{i_1}	0	0	0
i_2	x_{i_2}	0	x_{i_2}	0	0
a	z_a	0	0	x_a	x_a
b	0	z_b	0	x_b	x_b
e	0	0	z_e	x_e	x_e

The importance of the variable flow-demand matrix is captured by the following theorem:

Theorem 6.4.5. Let (G, I, O) be an open graph. Then there exists measurement labelling λ such that (G, I, O, λ) has Pauli flow if and only the variable flow-demand matrix W of (G, I, O) is right-invertible under some $\mathbb{F}_2$ valuation of its variables.

Proof. (⇒): let λ be such that (G, I, O, λ) has Pauli flow. By the explanation above, we may assume that λ assigns Pauli measurements only and the flow-demand matrix M of (G, I, O, λ) is right-invertible. Now, consider the following valuation σ of the variables in W, for each $v \in \bar{O}$:

- if $\lambda(v) = X$, set $\sigma(x_v) = 1$ and $\sigma(z_v) = 0$,
- if $\lambda(v) = Y$, set $\sigma(x_v) = 1$ and $\sigma(z_v) = 1$,
- if $\lambda(v) = Z$, set $\sigma(x_v) = 0$ and $\sigma(z_v) = 1$.

Let $\sigma(W)$ be W under valuation $\sqsubseteq$. By the constructions of order-demand matrix (cf. Definition 6.4.3) and flow-demand matrix (cf. Definition 5.2.7), $M = \sigma(W)$. As explained above, M is right-invertible and thus W is right-invertible under valuation σ.

(⇐): Let σ be such that $\sigma(W)$ is right-invertible. Suppose that $\sigma(x_v) = \sigma(z_v) = 0$ for some $v \in \bar{O}$. Then v row of $\sigma(W)$ equals identically zero, contradicting right invertability of $\sigma(W)$. Hence, $(\sigma(x_v), \sigma(z_v)) \in \{(1, 0), (0, 1), (1, 1)\}$ for all $v \in \bar{O}$. Now, let λ be the measurement labelling, reversing the approach from the proof in the other direction:

- if $(\sigma(x_v), \sigma(z_v)) = (1, 0)$, set $\lambda(v) = X$,
- if $(\sigma(x_v), \sigma(z_v)) = (1, 1)$, set $\lambda(v) = Y$,
- if $(\sigma(x_v), \sigma(z_v)) = (0, 1)$, set $\lambda(v) = Z$.

Again by construction, the flow-demand matrix M of (G, I, O, λ) equals $\sigma(W)$ and hence M is right-invertible. By the explanation above, this means that (G, I, O, λ) has Pauli flow. □

Therefore, we have reduced the problem **MeasurementSearch**, to deciding whether variable flow-demand matrix W of the input open graph is right-invertible under some valuation:

Corollary 6.4.6. The answer to **MeasurementSearch** instance (G, I, O) is $True$ if and only if the variable flow-demand matrix W of (G, I, O) is right-invertible under some valuation.

Equipped with the above corollary, we may return to the previous example:

Example 6.4.7. Consider the open graph (G, I, O) from Example 6.4.2 and its variable flow-demand matrix W from Example 6.4.4. The matrix W is square and hence it is *right-invertible* under some valuation if and only if it is *invertible* under some valuation. However, o_1 and o_2 columns of W are identical under all valuations. Hence, W is never invertible and by Corollary 6.4.6, the answer to **MeasurementSearch** instance (G, I, O) is *False*.

6.4.3 Known problems from linear algebra

The problem described in Corollary 6.4.6 is in fact an instance of a well-known problem: **MaxRank**. The following definition is based on [30] (in contrast, we do not require the input matrix to be square).

MaxRank
Fixed: A commutative ring R, and subsets $E, S \subseteq R$ of entries and solutions.
Input: Natural numbers m, n, t, r and $m \times n$ matrix M with entries from $E \cup \{x_1, \ldots, x_t\}$.
Output: *True* if $\operatorname{rank} M(a_1, \ldots, a_t) \geq r$ for some $a_1 \ldots a_t \in S^t$, and *False* otherwise.

$M(a_1, \ldots, a_t)$ stands for the matrix with substituted variables $x_1 \mapsto a_1, \ldots, x_t \mapsto a_t$. We skip the specification of m, n, t, as these numbers are explicit from the input matrix. Thus, we will write instances of **MaxRank** as pairs (M, r) of the matrix and the desired minimal rank under some valuation.

Example 6.4.8. Consider the following matrix over $\mathbb{F}_2$:

$$M = \begin{pmatrix} x_1 & 0 & 1 & 0 \\ 0 & x_2x_3 & 0 & 0 \\ 1 & 0 & x_1 & 0 \\ x_1 & x_2 & x_3 & 0 \end{pmatrix}$$

Setting $x_1 = 0$ and $x_2, x_3 = 1$ results in M having rank 3. However, there is no valuation resulting in the matrix having rank 4, as the fourth column contains 0s only. Hence, the answer to instances $(M, 1), (M, 2), (M, 3)$ of **MaxRank** is *True*, but the answer to $(M, 4)$ is *False*.

In [30], it is shown that when R is a finite field and each variable occurs at most once, the problem is in RP. We extend their approach to show that the problem is in RP also when each variable appears in at most one row or one column. For that, we need the notion of multi-affine polynomials (adapted from [30]). In general, the **MaxRank** problem with R being a finite field is NP-complete [30].

Definition 6.4.9. A multivariable polynomial is *multi-affine* when each variable has a degree at most one.

By extending the proof [30, Theorem 28], we get the following version. Here, the entries of the matrix are allowed to be given by multi-affine expressions over $E \cup \{x_1, \ldots, x_t\}$, not just elements of the set.

Theorem 6.4.10. The following version of **MaxRank** is in RP:

- $R = E = S = \mathbb{F}_s$ is a finite field,
- each variable appears in at most one row or at most one column,
- the entries are given by (polynomially long) multi-affine expressions over $E \cup \{x_1, \ldots, x_t\}$.

By saying that a variable appears in at most one row or at most one column we mean that if a variable appears in entries at positions $(a_1, b_1), (a_2, b_2), (a_3, b_3), \ldots$, then either $a_1 = a_2 = a_3 = \ldots$ or $b_1 = b_2 = b_3 = \ldots$. Some of the variables may appear only in one row, and some may appear in only one column.

In order to prove theorem 6.4.10, we need the following lemma.

Lemma 6.4.11. A multi-affine polynomial is 0 over a finite field $\mathbb{F}_s$ if and only if it is 0 over $\mathbb{F}_{s^k}$ for any $k \in \mathbb{Z}_+$.

Proof. This follows from [30, Lemma 25 and Corollary 26]. □

Because of the above, instead of testing whether a multi-affine polynomial is 0 over $\mathbb{F}_s$, we can test whether it is 0 over some large field extension. In particular, the extension can be taken sufficiently large to ensure that the Schwartz-Zippel lemma [139, 150, 63, 183] applies (adapted to $\mathbb{F}_{s^k}$ only):

Theorem 6.4.12. Let $P \in \mathbb{F}_{s^k}[x_1, \ldots, x_n]$ be a non-zero polynomial of total degree d over $\mathbb{F}_{s^k}$. Let $a_1, \ldots, a_n \in \mathbb{F}_{s^k}$ be chosen at random uniformly. Then:

$$\Pr[P(a_1, \ldots, a_n) = 0] \leq \frac{d}{s^k}.$$

We can now prove Theorem 6.4.10.

Proof of Theorem 6.4.10. Let M, r be a matrix and an integer forming an input to **MaxRank** satisfying the conditions from the theorem statement. If $r > \min(m, n)$ or $r < 0$, the answer is *False* and can be returned immediately, so assume $0 \leq r \leq \min(m, n)$. Consider any $r \times r$ minor M' of M. We show $\det M'$ is multi-affine: consider any variable x in M'. Then, x appears in at most one row or one column of M'. By performing Laplace expansion on M' in such row (column), we find that $\det M'$ is a sum of determinants of $(r-1) \times (r-1)$ minors of M' that do not contain x multiplied by elements of the row (column) used for the expansion that itself may contain x only in degree 1 due to multi-affinity assumption about matrix entries. Therefore, x appears in degree at most 1 in $\det M'$ and the same for all other variables of $\det M'$, so the determinant is multi-affine. Therefore, the method from [30, Theorem 28] applies. We present a modified version for clarity.
Consider the following procedure. Let $p \leq \frac{1}{2}$ be the desired error probability. It is sufficient to consider $p = \frac{1}{2}$, but we present also how to achieve arbitrarily small error probability. Given M, r, p, let k be such that $\mathbb{F}_{s^k}$ has at least $\frac{t}{p}$ elements, i.e. $k = \left\lceil \log_s \frac{t}{p} \right\rceil$. Let $a_1, \ldots, a_t$ be a randomly chosen valuation of $x_1, \ldots, x_t$ from $\mathbb{F}_{s^k}$. Let $r_a = \operatorname{rank} M(a_1, \ldots, a_t)$, which can be found by Gaussian elimination i.e. in polynomial time. The computations over finite field are possible in time polynomial in $\log_2 s$ and k [78]. Return $True$ if and only if $r_a \geq r$. We show, that the following procedure shows RP containment.
Suppose, that the actual answer to the instance is *True*. Then, under some valuation, M has rank at least r. Under such valuation, M must have $r \times r$ reversible minor. Let M' be such minor. By the previous part, $\det M'$ is multi-affine. Let d be the total degree of $\det M'$. Then, $d \leq t$ again by multi-affinity. By lemma 6.4.11, M' is 0 over $\mathbb{F}_s$ if and only if it is 0 over $\mathbb{F}_{s^k}$. Combining everything with the Schwartz-Zippel theorem 6.4.12, we get that:

$$\Pr[\det M'(a_1, \ldots, a_t) =_{\mathbb{F}_{s^k}} 0] \leq \frac{d}{s^k} \leq \frac{t}{s^k} \leq \frac{t}{t/p} = p$$

The same holds for all $r \times r$ minors of M that are invertible under some valuation. Hence, with error probability at most p, a random valuation from $\mathbb{F}_{s^k}$ results in a non-root of some $r \times r$ minor's determinant of M in which case rank of the minor under such valuation is r and so rank M is at least r, i.e. the procedure above would find $r_a \geq r$ and return *True*. Hence, the procedure described above returns the correct answer with probability at least $1 - p \geq \frac{1}{2}$.

Now suppose, that the answer to the instance is *False*. Then, all $r \times r$ minors of M must have determinants equal 0 over $\mathbb{F}_s$. By multi-affinity, they are also equal 0 over $\mathbb{F}_{s^k}$. Hence, a random valuation $a_1, \dots, a_t$ always results in rank $M(a_1, \dots, a_t) \leq k$. Hence, the procedure described above returns *False* with probability 1, ending the proof of containment in RP. □

The requirement that each variable appears in at most one row or column is essential: When each variable can appear at most twice in the matrix, but not necessarily in one row or one column, the problem already becomes NP-complete [85].

Some closely related problems also defined in [30] include **MinRank**, **Sing**, and **NonSing**. **MinRank** takes the same inputs as **MaxRank** and asks whether a rank $\leq r$ can be achieved. **Sing** and **NonSing** take a square matrix and ask whether the matrix can be made singular and non-singular respectively.

In general, these problems are hard or sometimes unsolvable. For instance, **MinRank** is undecidable when $R = \mathbb{Z}, E = S = \{0, 1\}$ [30]. The problems are very natural and often appear when working on any linear algebra problems. **Sing** is useful in cryptography due to its hardness (for example, see [17]). It is also interesting from a complexity perspective (for example, see [118]).

In our case, the instances of **MaxRank** are sufficiently simple to obtain the following result:

Theorem 6.4.13. MeasurementSearch $\in$ ***RP***.

Proof. By Corollary 6.4.6, **MeasurementSearch** reduces to instances of **MaxRank** over $\mathbb{F}_2$ with each variable appearing in at most one row. By Theorem 6.4.10, such instances are solvable by a random polynomial-time algorithm, ending the proof. □

The algorithm implied by the above theorem follows in the next subsection.

An analogous procedure can also determine whether a partial labelling $\lambda\colon \bar{O} \hookrightarrow \{X, Y, Z\}$ can be extended to a full labelling with flow. To do so, rather than using **MaxRank** on $W_{G,I,O}$, we can consider it on $W_{G,I,O}$ with some variables evaluated to 1 or 0 depending on λ. It means that the problem of finding λ such that (G, I, O, λ) has Pauli flow is also solvable in random polynomial time, as captured by the following corollary with initial $\lambda = \emptyset$.

Corollary 6.4.14. Given an open graph (G, I, O) and a partial measurement labelling λ with codomain $\{X, Y, Z\}$, it is in RP to check if λ can be extended to a full labelling λ' such that (G, I, O, λ') has Pauli flow.

6.4.4 Pseudocode

This algorithm relates to the construction of a variable flow-demand matrix from Definition 6.4.3, reduction from Corollary 6.4.6, and ***RP*** algorithm implied by Theorem 6.4.10.

Algorithm 4 Algorithms for unlabelled open graphs

```
Checks if a partial $X, Y, Z$ labelling can be extended so that $(G, I, O, \lambda)$ has Pauli flow. The error probability must be bounded above by $p$.
 1: procedure MeasurementLabellingAux($G, I, O, \lambda, p$)
 2:     $W = A_G |_{\bar{I}}^{\bar{O}}$                ▷ Construction of reduced adjacency matrix
 3:     $B = \bar{O} \cap \bar{I}$                     ▷ Name for set of measured non-inputs
 4:     $Vars = \emptyset$                             ▷ Initialise set of variables
 5:     for $v \in B$                                  ▷ Detecting unlabelled vertices
 6:         if $\lambda(v)$ is defined
 7:             if $\lambda(v) == Z$                   ▷ Updating the row of Z labelled vertex
 8:                 $W_{v,*}$ *= 0
 9:             if $\lambda(v) \in \{Y, Z\}$           ▷ Updating the intersection of row and column of Z or Y labelled vertex
10:                 $W_{v,v}$ to 1
11:         else
12:             $Vars \cup \{x_v, z_v\}$
13:     $k = \left\lceil \log_2 \frac{|Vars|}{p} \right\rceil$   ▷ Minimal k such that $\mathbb{F}_{2^k}$ has at least $\frac{|Vars|}{p}$ elements and the error probability is below p.
14:     Randomly sample $\sigma : Vars \longrightarrow \mathbb{F}_{2^k}$
15:     for $v \in B$                                  ▷ Construction of $W_{G,I,O}$ under valuation $\sigma$, computations are done in $\mathbb{F}_{2^k}$
16:         $W_{v,*}$ *= $\sigma(x_v)$
17:         $W_{v,v} = \sigma(z_v)$
18:     Gaussian eliminate $W$ return (rowrank $W == |\bar{O}|$)

Main algorithm, returns True if $(G, I, O, \lambda)$ has Pauli flow for some $\lambda$. The error probability must be bounded above by $p$.
19: procedure MeasurementLabelling($G, I, O, p$)
20:     return MeasurementLabellingAux($G, I, O, \emptyset, p$)
```

```
Finds $\lambda$ such that $(G, I, O, \lambda)$ has Pauli flow. Initially, checks whether any such
$\lambda$ exists, up to error probability $p$.
21: procedure FindLabelling($G, I, O, p$)
22:     if not MeasurementLabelling($G, I, O, p$)
23:         return "NO $\lambda$ EXISTS"
24:     $\lambda = \emptyset$                                   ▷ Initialization of $\lambda$
25:     for $v \in I$
26:         $\lambda(v) = X$                                    ▷ Inputs must be $X$ labelled
27:     for $v \in B$
28:         $confirm_v = False$
29:         $current_v = X$                                     ▷ Initially attempt $X$ label
30:         while not $confirm_v$          ▷ Alternate $X$, $Z$, and $Y$ labels until one works
31:             $\lambda(v) = current_v$
32:             if FlowSearchAux($G, I, O, \lambda, p$)  ▷ The error probability could be
                                                increased at the cost of possi-
                                                bly more tries being required.
33:                 $confirm_v = True$
34:             else
35:                 if $current_v == X$
36:                     $current_v = Z$
37:                 else if $current_v == Z$
38:                     $current_v = Y$
39:                 else
40:                     $current_v = X$
41:     return $\lambda$
```

In [132], I proved that we can work in an even greater restriction of only X and Z measurements, i.e. without Y measurements. However, I decided to omit it here, as the point remains the same: The algorithm still only looks for Pauli measurements.

6.4.5 Complexity of the algorithms on unlabelled open graphs

The most memory-expensive part of the algorithms is the creation of $W_{G,I,O}$ under some valuation from $\mathbb{F}_{2^k}$ where k depends on the desired error probability p and the size of the input graph. Since $k = \left\lceil \log_2 \frac{|Vars|}{p} \right\rceil$ and $|Vars| \leq 2 \cdot |B|$, we get that $k \in O\left(\log_2 \frac{|B|}{p}\right)$. The elements of such fields can be represented using $O(k)$ long vectors over $\mathbb{F}_2$ with the time complexity of basic arithmetic operations on such a field bounded above by $O(k^2)$ [78]. Therefore, the memory requirement can be bounded above by $O\left(|\bar{O}| \times |\bar{I}| \times \log_2 \frac{|B|}{p}\right) \in$

$O\left(n^2 \log_2 \frac{n}{p}\right)$ where $n = |V|$. The most time-consuming part of the algorithm is Gaussian elimination. Each Gaussian elimination requires $O\left(n^3\right)$ basic operations in $\mathbb{F}_{2^k}$. Thus, a (not very efficient) upper bound for the time complexity is $O\left(n^3 \log_2^2 \frac{n}{p}\right)$ for the decision variant and expected $O\left(n^4 \log_2^2 \frac{n}{p}\right)$ for the actual finding of the labelling resulting in a Pauli flow.

As a final word on the problem of finding measurement labelling that results in Pauli flow, we mention that it is possible to obtain a deterministic polynomial-time algorithm rather than a random polynomial-time algorithm. In [95], authors showed that when a variable can appear in at most one row or one column but an unlimited number of times than corresponding variant of **MaxRank** problem is in P. This result is not stronger than Theorem 6.4.10, as the entries of the matrix there can only include expressions constructed using addition and subtraction, but not products of variables. On the other hand, this result is in fact sufficient to argue that **MeasurementSearch** $\in \boldsymbol{P}$: Using this result, we could obtain a deterministic polynomial-time algorithm. However, such an algorithm would be less desirable for practical approaches, as it would require multiple Gaussian elimination steps to construct bases of various subspaces, see [95] for details.

6.5 Algorithms efficiency

While we provided bounds for the complexity of our algorithms, these are often worst-case bounds that do not necessarily apply to practical situations. Furthermore, the algorithms work in the general case for their respective problems, i.e. we did not assume anything about the structure of the underlying graphs on which the algorithms could be run. Here, we provide an extended commentary on the efficiency of the presented algorithms, taking into account current state-of-the-art algorithms for underlying problems, the parametrisation of the underlying graphs, and existing implementations of flow-finding procedures.

6.5.1 Subroutines for linear algebra problems

The flow-finding algorithm, in the case of an equal number of inputs and outputs, as described in Section 6.2, relies on matrix multiplication and matrix inversion. The naïve subroutines suffice for an overall $O\left(n^3\right)$ bound, yet faster methods for matrix inversion and multiplication exist, for instance, by the utilisation of Strassen's algorithm [159]. Using such methods reduces the complexity to $O\left(n^{\log_2 7}\right) \approx O\left(n^{2.807}\right)$. Recent breakthroughs suggest further improvements by decomposing matrices into different-sized block matrices, see for example [68]. Even faster algorithms for matrix multiplication and inversion exist [179], but they are *galactic*, meaning they are impractical due to their requirements on the size of n.

In Section 6.3, we also used the matrix multiplication subroutine in the general case algorithm for flow-finding. However, in that case, the matrices are no longer square. The above theoretical results still apply, provided that the input matrices are first padded to minimal-size square matrices. Solutions tailored specifically to non-square matrices are also known, for example [75], however, such approaches are both galactic and only provide benefit over padding to square matrices when matrix dimensions differ significantly.

Even if the theoretically best algorithms are impractical, the Strassen algorithm can still offer speed-ups for sufficiently big matrices, i.e. n must be on the order of hundreds or thousands to notice any difference [93]. Currently, algorithms based on even more advanced matrix decompositions are unlikely to be beneficial for flow-finding due to the constant factors involved. Still, recent breakthroughs in the area of fast matrix multiplication have been obtained and are already being utilised in the training of Google's large language model [138], and can likely be applied to even smaller instances in the future.

Similar to the reliance on matrix multiplication and inversion, algorithms from Sections 6.3 and 6.4 employ Gaussian elimination as a crucial step in solving the underlying linear systems. While the worst case of Gaussian elimination for $O(n) \times O(n)$ matrix is $O(n^3)$, there exists an asymptotically faster method that applies to our scenario of finite field matrices [8]. In more practical terms, for the case of $\mathbb{F}_2$ matrices relevant to general flow-finding, efficient implementations exist [7], achieving similar asymptotic results to those from [8].

Lastly, the algorithm in Section 6.4 requires efficient computation in finite fields. Many programming languages offer relevant packages for this exact purpose. For instance, in Python, one can use the Galois package [90] which works very well for $\mathbb{F}_{2^k}$ with k such that the precomputed Conway polynomial [117] is known, for example, all $1 \leq k \leq 91$. Such values of k are sufficient for all reasonable computations, as the error can be dropped below $\frac{1}{2^{50}}$. At that point, it is more likely for a random cosmic beam to corrupt the computation than for an error to occur due to the probabilistic nature of the algorithms.

6.5.2 Graph parametrisation

In the presented algorithms, we did not distinguish between expected complexities depending on the underlying graph structure, except for splitting the flow-finding into two cases based on whether the number of inputs equals the number of outputs. While in the general case, we established a lower bound (Thereom 6.1.2) for flow finding by connecting it to the problem of matrix inversion and multiplication, specific parametrisations of the graph can effectuate faster solutions than general methods for these linear algebra questions, or skip the cases we used to obtain the lower bound completely. Suppose that $\Gamma = (G, I, O, \lambda)$ is a labelled open graph, $n = |V|$, $m = |E|$, $n_I = |I|$, and $n_O = |O|$ are the number of vertices, edges, inputs, and outputs respectively; and d is the degree of the graph (the maximum degree of any vertex in the graph).

When G is sparse (for example, with $m \in O(n)$), then so is its adjacency matrix and thus also the flow-demand and order-demand matrices of Γ. Finding the inverse of a sparse matrix can be solved even faster than matrix multiplication [25, 36]. Similar approaches apply to cases of bounded degree d, where complexity reduces from $O(n^3)$ for standard approaches to $O(n^2 d)$ instead. Sparsity might seem very restrictive and thus uncommon practically, yet it is actually expected for most practical applications, at least in part of the labelled open graph. The reason behind this fact is that labelled open graphs are typically not random. Instead, they arise via flow-preserving transformations [121, 122, 16] of labelled open graphs corresponding to circuits [15], which have a bounded degree; for Clifford+T circuits, this bound is three. Another class of practical instances are labelled open graphs implementable on universal resource states, such as cluster states [26, 145]. These have a maximal degree of four or six, depending on the dimension of the used cluster state. When input Γ for flow-finding arises from such cases up to some transformations, it is reasonable to expect at least part of the vertices in the graph to inherit bounded degree.

When the 'width' of the graph (ratio $\frac{n}{n_I}$) is small, we can expect a smaller depth of the computation, which in the flow-based regime translates to a smaller number of layers in the layer-by-layer approach used in flow-finding procedures. In extreme cases, the number of layers can be equal to just one: that is, all measured vertices can be measured at the same time. This includes the class of labelled open graphs we used to find a lower bound in Theorem 6.1.2. Such cases are less practical, as they would correspond to circuits of lower depth [91], which, while interesting, are less likely to capture the full potential of quantum computing. Thus, practical cases are more likely to have high width and therefore not contain instances used to obtain the lower bound.

In the case of causal flow, there exists an explicit bound on the number of edges linking n and n_O, which is $m \leq (n-1)n_O - \binom{n_O}{2} \in O(nn_O)$ [126]. When n_O is constant, this puts us in the sparse graph territory. Similar bounds are not currently known for gflow, and they do not apply to Pauli flow: In particular, for $n_O = 0$ all Z-measured cliques and all X-measured even-sized cliques have Pauli flow. A very high number of edges indicates that the graph is highly entangled. Generally, highly entangled resource states are not universal for MBQC [82]; a similar result applies to some classes of graph states [84], suggesting highly entangled graph states are also less applicable as instances for running flow-finding. If the $O(nn_O)$ bound or similar could be applied to gflow, or adapted to Pauli flow (for instance, by treating Pauli measured vertices as outputs), then one could expect faster flow-finding for practical cases, as in those with large width: Large width means that n_I is small, and many practical cases have $n_I = n_O$, as explained in Section 5.6, and thus small n_O.

We consider one more class of 'practical' resource states on which Pauli flow can be used to reason about determinism, which arises in fusion-based quantum computing (FBQC) [18]. FBQC is one of the most promising models for photonic quantum computers. In FBQC, resource states are created by supplying small resource states and then combining them by application of fusions – multi-qubit entangling measurements. Generally, the flow regime does not apply to such settings. However, in [69], it was shown that for specific instances of fusions, namely X and Y fusions, it is possible to derive so-called XY flow for fusion networks. Contrary to what the name suggests, this is not an XY-only gflow, but rather a Pauli flow with certain restrictions on how X- and Y-measured vertices can be used in the correction mechanism [69, Definition 4.13]. Although not explicitly stated, these restrictions result in a rediscovery of focused Pauli flow, which can be identified through our flow-finding method. This means that our flow-finding procedure also applies to FBQC, provided fusions are restricted to X and Y. Such practical implementations of MBQC via FBQC again fall into cases of bounded degree: small resource states created before fusions have constant size and therefore bounded degree, and at most one fusion appears at each vertex. This means that, once again, a faster method for flow-finding could be derived by utilising linear algebra methods for sparse matrices.

6.5.3 Existing implementations of flow algorithms

Flow-finding algorithms are included in PyZX [108], t $|\text{ket}\rangle$ [155], and Graphix [160]. Very recently, my flow-finding algorithms were implemented in Graphix [169]. Based on my discussion with the Graphix team, the observed average runtime for test instances is close to $O(n^{2.6})$ for flow-finding, which also matches the observed average-case performance of flow-based circuit extraction in PyZX. These implementations provide empirical evidence that the theoretical worst-case bound of $O(n^3)$ is not a reasonable estimate of average-case performance. Implementations taking into account topics from this commentary, such as faster methods for matrix inversion or considerations of graph sparsity, do not exist as of now and are a natural next step to take in search of the optimal flow-finding method for practical purposes.

Chapter 7

Summary and Conclusions

In the closing chapter of my thesis, I provide a final summary of the presented work and directions for potential further work.

7.1 Summary

In this thesis, I have worked on connections between graphical calculi for quantum computation and computational complexity. My results provide both graphical calculi limitations and expand on methods for overcoming these limitations. On one side, I proved that specific problems about string diagrams can be too complex to solve efficiently. On the other hand, I have looked at the structures that imply efficient algorithms for circuit extraction, the most important of the problems mentioned. Since my work sits at the intersection of different subfields of computer science, two chapters have been fully dedicated to explaining the background material and the works of others.

7.1.1 Limitations of diagrammatic reasoning

In Chapter 4, I have shown that problems in graphical calculi can be too challenging to expect efficient solutions, even under generous assumptions about open problems in computational complexity theory. In particular, even if $\boldsymbol{P} = \boldsymbol{NP}$, the translation between different presentations of quantum computation may still be beyond our reach: most likely we will never be able to transform a diagram to a circuit, i.e. a valid input for quantum computers. This way, my work provides additional motivation to find partial solutions to such problems. My results in the case of circuit extraction suggest that methods conditional on the properties of the diagrams are the best we can hope for.

Formally, I extended the previous $\#\boldsymbol{P}$-hardness result [62] for **CircuitExtraction** to phase-free ZH diagrams. Furthermore, I showed that two problems arising in graphical calculi can be $\boldsymbol{NP^{\#P}}$-complete. In the process, I proved various properties of this obscure complexity class. Most importantly, I found a complete problem $\exists$**Compare#SAT** that could catalyse further complexity results about graphical calculi for quantum computation. My results extend to any graphical calculi for quantum computation with efficient constructions of **#SAT** instances and a few other tensors.

7.1.2 Algebraic interpretation of flow structure

In Chapter 5, I have looked at the structures that imply efficient algorithms for the circuit extraction. Flow properties are a glue between the Measurement-based Quantum Computation, circuits, and string diagrams. Originally arising as a guarantee of robust determinism in the Measurement-based Quantum Computation model, flows are now the critical foundation for the broader application of graphical calculi. Yet, the previously known presentation of flow conditions was very complicated, which hindered working with flow.

I simplified complicated definitions of flow structures to basic matrix operations in linear algebra. Given a labelled open graph Γ, I introduced the notions of a 'flow-demand matrix' M and an 'order-demand matrix' N. I proved that the Pauli flow on Γ corresponds to the existence of a right inverse C of M, such that NC forms the adjacency matrix of a directed acyclic graph. This provides a new algebraic interpretation of Pauli flow, extending previously known algebraic interpretations [125]. Furthermore, I used this algebraic presentation to carry out multiple proofs. I showed that the flow can be reversed if the number of inputs and outputs matches. When there are more outputs than inputs, I showed that the kernel of M characterises focused sets, in a way alternative to that from [153]. Finally, I proposed a mechanism to equalise the number of inputs and outputs, though it is restricted to the possibility of having to relabel some qubits. The new algorithms, discussed in the next subsection, are by far the most important immediate application of the new algebraic interpretation. The algebraic interpretation contributes to further theoretical understanding of Pauli flow.

7.1.3 Flows algorithmically

In Chapter 6, I designed a new Pauli flow algorithm, building on the algebraic interpretation from Chapter 5. Finding flow is of great interest due to its correspondence to robustly deterministic computation in MBQC and as a necessity for efficient circuit extraction from measurement patterns. By extension, flows are also essential for circuit extraction from ZX diagrams. Previous solutions to the problem of finding flow were unsatisfying because they required multiple Gaussian eliminations, which made them slow. The widely applied gflow-based circuit extraction runs in $O(n^3)$. It is faster than the best $O(n^4)$ method for finding gflow, the existence of which was necessary for circuit extraction to terminate. The high time complexity of $O(n^4)$ prevented Pauli flow-based circuit extraction from being applied as much as a gflow-based method.

Using our new algebraic interpretation, I developed $O(n^3)$ algorithms for finding flow. The algorithm for the general case works for any number of inputs and outputs and is an improvement of the layer-by-layer approach used in [126, 15, 153]. In the special case of equal numbers of inputs and outputs, the algebraic interpretation gives a new and simpler proof that a focused Pauli flow (if it exists) must be unique. As a consequence, there is a simpler flow-finding algorithm, which differs from other existing flow-finding algorithms. Further, I reduced the problem of finding flow on labelled open graphs with equal numbers of inputs and outputs to, and from, the problem of finding the inverse of a matrix and performing matrix multiplication. Thus, I argue that further improvements to flow-finding algorithms must necessarily lead to or come from new methods for these standard linear algebra problems.

7.2 Further work

There are many interesting possibilities for further work.

7.2.1 Other notions of MBQC

Pauli flow corresponds to robust determinism, i.e. strong uniform stepwise determinism, the standard notion of determinism for the one-way model. However, other relevant notions of determinism exist [125]. It may be interesting to check if algebraic interpretations can be established for those notions of determinism as well.

Similarly, other types of flow are defined for different quantum computation fragments, which I briefly discussed in Subsection 2.6.4. As of now, neither of Zd flow, hypergraph gflow, and shadow Pauli flow has a clean algebraic presentation. The algorithms for Zd flow and shadow Pauli flow exist, but run in $O(n^4)$ and possibly could be improved like I did for Pauli flow. In the case of hypergraph gflow, there is no polynomial-time algorithm. I plan to find a suitable algorithm and extend the notion to hypergraph Pauli flow.

Currently, almost all flow notions are restricted to instances where the underlying resource state does not change during the course of the computation. Here, the exception is the hypergraph flow, where multi-qubit controlled gates may appear in the correction mechanism. This is a change in the philosophy of deterministic MBQC with multi-qubit byproducts, where previous approaches overcame multi-qubit byproducts by switching measurement bases of future measurements, rather than explicitly altering the underlying graph structure [105, 163]. It is interesting to research if allowing more complex corrections, such as multi-qubit corrections, as opposed to only X and Z corrections, can lead to flow for more complex settings like measurements beyond standard planes of the Bloch sphere.

Finally, flow could be adapted to other settings than graph states. Sometimes computation in MBQC can be suboptimal when presented as measurements of graph states. Therefore, flow could be developed when the resource state is presented differently. For instance, in [87], a toric code is used to specify the resource state. One could adapt flow mechanisms to dynamically generated resource states. One possibility is to extend flow results for fusion-based quantum computation, building on the initial flow results presented in [69].

It is important to point that while our algorithms are restricted to robust determinism via the notion of Pauli flow, sometimes MBQC reaches outside of robustly deterministic measurement patterns. For example, robustly deterministic computation can be extended by further components without flow, or in the process of circuit optimisation, flow could be lost. Then, our algorithms do not apply; however, this does not mean that flow structure is useless in such cases. Since flow is the best tool we have for circuit extraction, it is interesting to consider how far the computation is from having flow. In [67], such a distance from gflow was considered by looking for a set of vertices that, when turned into outputs, would result in gflow. This approach could be extended to Pauli flow and other metrics, like the number of edges that would need to be altered to reach flow.

7.2.2 Circuit optimisation and extraction

In Subsection 3.3.2, I discussed circuit optimisation via ZX Calculus where flow plays a critical part. During the optimisation, the flow must be preserved. Proving that a rewrite rule preserves flow used to be very challenging, as all flow conditions had to be verified one by one [121, 122]. The new algebraic interpretation simplifies verification, for instance, see [16]. The new algebraic interpretation is also ‘computer-friendly’, which opens the possibility of searching for new flow-preserving rules with a computer. I believe the new algebraic interpretation may lead to the discovery of new flow-preserving rewrite rules.

My algorithm speeds up the procedure from [153]: A better Pauli flow-finding algorithm implies a faster Pauli-flow-based circuit extraction from ZX diagrams and measurement patterns. The existing method is split into two parts: first, a flow is found, and then so-called extraction strings are defined based on which one produces an equivalent circuit. The current method may introduce 2-qubit gates in the final step, which goes against some optimisation goals [157]. It could be interesting to research whether the second part can be incorporated into flow finding. While that would not lead to a faster circuit extraction overall, it could give better control over the produced circuit.

7.2.3 Counting classes and graphical calculi

Many of the problems encountered in graphical calculi remain mysterious. As mentioned in Subsection 4.7, the approach of reduction via Turing machines has its limits when trying to work out $\boldsymbol{coNP^{\#P}}$-hardness. However, there is a possibility of achieving the desired hardness results of problems encountered in graphical calculi via a different path. For instance, instead of considering $\boldsymbol{C_=P}$ oracle, one can switch to $\boldsymbol{coC_=P}$, which equals $\boldsymbol{NQP}$ [70], i.e. the class that has an approximate quantum circuit comparison as a complete problem. Similarly, the $\#\boldsymbol{P}$ oracle could be switched to $\boldsymbol{PP}$ and thus to $\boldsymbol{PostBQP}$. This way, one can study relations of phase-free ZH diagrams to Toffoli+H circuits with post-selection.

Besides theoretical results about the hardness of problems in graphical calculi, there is also a possibility of extending the graphical **#SAT** algorithm from [114] to achieve algorithms for other related classes, such as $\boldsymbol{NP^{\#P}}$.

7.2.4 Unlabelled open graphs

My method for finding measurement labelling resulting in Pauli flow only considers Pauli measurements. This way, a final labelled open graph corresponds to some Clifford computation. Thus, the inclusion of planar measurements is the natural next step. The problem of finding measurement labellings resulting in Pauli flow can also be adapted to settings involving other types of flow or other notions of determinism. It can also be generalised to the question of how to utilise open graphs for quantum computation.

Bibliography

[1] Scott Aaronson (2005): *Quantum Computing, Postselection, and Probabilistic Polynomial-Time*. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 461(2063), pp. 3473–3482, doi:10.1098/rspa.2005.1546.

[2] Scott Aaronson, Greg Kuperberg & Oliver Habryka (2002): *Complexity Zoo*. `https://complexityzoo.net/Complexity_Zoo`. Accessed: 4th May 2025.

[3] Antonio Acín, Dagmar Bruß, Maciej Lewenstein & Anna Sanpera (2001): *Classification of Mixed Three-Qubit States*. *Physical Review Letters* 87(4), p. 040401, doi:10.1103/PhysRevLett.87.040401.

[4] Leonard M. Adleman, Jonathan DeMarrais & Ming-Deh A. Huang (1997): *Quantum Computability*. *SIAM Journal on Computing* 26(5), pp. 1524–1540, doi:10.1137/S0097539795293639.

[5] Manindra Agrawal, Neeraj Kayal & Nitin Saxena (2004): *Primes Is in P*. *Annals of Mathematics* 160, pp. 781–793, doi:10.4007/annals.2004.160.781.

[6] Dorit Aharonov (2003): *A Simple Proof That Toffoli and Hadamard Are Quantum Universal*. *quant-ph/*0301040. arXiv:quant-ph/0301040.

[7] Martin R. Albrecht, Gregory V. Bard & Clément Pernet (2011): *Efficient Dense Gaussian Elimination over the Finite Field with Two Elements*. arXiv:1111.6549.

[8] Daniel Andrén, Lars Hellström & Klas Markström (2007): *On the Complexity of Matrix Reduction over Finite Fields*. *Advances in Applied Mathematics* 39(4), pp. 428–452, doi:10.1016/j.aam.2006.08.008.

[9] Sanjeev Arora & Boaz Barak (2009): *Computational Complexity: A Modern Approach*, 1st edition edition. Cambridge University Press, Cambridge ; New York.

[10] Miriam Backens (2014): *The ZX-calculus Is Complete for the Single-Qubit Clifford+T Group*. *Electronic Proceedings in Theoretical Computer Science* 172, pp. 293–303, doi:10.4204/EPTCS.172.21.

[11] Miriam Backens (2016): *Completeness and the ZX-calculus*. Ph.D. thesis, University of Oxford. arXiv:1602.08954.

[12] Miriam Backens (2021): *A full dichotomy for Holant^c, inspired by quantum computation*. *SIAM Journal on Computing* 50(6), pp. 1739–1799, doi:10.1137/20M1311557. arXiv:2201.03375.

[13] Miriam Backens & Aleks Kissinger (2019): *ZH: A Complete Graphical Calculus for Quantum Computations Involving Classical Non-linearity*. *Electronic Proceedings in Theoretical Computer Science* 287, pp. 23–42, doi:10.4204/EPTCS.287.2. arXiv:1805.02175.

[14] Miriam Backens, Aleks Kissinger, Hector Miller-Bakewell, John van de Wetering & Sal Wolffs (2023): *Completeness of the ZH-calculus*. *Compositionality* 5, p. 5, doi:10.32408/compositionality-5-5.

[15] Miriam Backens, Hector Miller-Bakewell, Giovanni de Felice, Leo Lobski & John van de Wetering (2021): *There and Back Again: A Circuit Extraction Tale*. *Quantum* 5, p. 421, doi:10.22331/q-2021-03-25-421. arXiv:2003.01664.

[16] Miriam Backens & Thomas Perez (2025): *Inserting Planar-Measured Qubits into MBQC Patterns while Preserving Flow*. *Electronic Proceedings in Theoretical Computer Science* 426, p. 100–126, doi:10.4204/eptcs.426.4. Available at `http://dx.doi.org/10.4204/EPTCS.426.4`.

[17] Magali Bardet, Maxime Bros, Daniel Cabarcas, Philippe Gaborit, Ray Perlner, Daniel Smith-Tone, Jean-Pierre Tillich & Javier Verbel (2020): *Improvements of Algebraic Attacks for Solving the Rank Decoding and MinRank Problems*. In Shiho Moriai & Huaxiong Wang, editors: *Advances in Cryptology – ASIACRYPT* 2020, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 507–536, doi:10.1007/978-3-030-64837-4_17.

[18] Sara Bartolucci, Patrick Birchall, Hector Bombín, Hugo Cable, Chris Dawson, Mercedes Gimeno-Segovia, Eric Johnston, Konrad Kieling, Naomi Nickerson, Mihir Pant, Fernando Pastawski, Terry Rudolph & Chris Sparrow (2023): *Fusion-based quantum computation*. *Nature Communications* 14(1), p. 912, doi:10.1038/s41467-023-36493-1. Available at `https://doi.org/10.1038/s41467-023-36493-1`.

[19] Paul Benioff (1980): *The Computer as a Physical System: A Microscopic Quantum Mechanical Hamiltonian Model of Computers as Represented by Turing Machines*. *Journal of Statistical Physics* 22(5), pp. 563–591, doi:10.1007/BF01011339.

[20] Paul Benioff (1982): *Quantum Mechanical Models of Turing Machines That Dissipate No Energy*. *Physical Review Letters* 48(23), pp. 1581–1585, doi:10.1103/PhysRevLett.48.1581.

[21] Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres & William K. Wootters (1993): *Teleporting an Unknown Quantum State via Dual Classical and Einstein-Podolsky-Rosen Channels*. *Physical Review Letters* 70(13), pp. 1895–1899, doi:10.1103/PhysRevLett.70.1895.

[22] Ethan Bernstein & Umesh Vazirani (2006): *Quantum Complexity Theory*. *SIAM Journal on Computing*, doi:10.1137/S0097539796300921.

[23] Ronald V. Book, Timothy J. Long & Alan L. Selman (1984): *Quantitative Relativizations of Complexity Classes*. *SIAM Journal on Computing* 13(3), pp. 461–487, doi:10.1137/0213030.

[24] Robert I. Booth, Aleks Kissinger, Damian Markham, Clément Meignant & Simon Perdrix (2023): *Outcome Determinism in Measurement-Based Quantum Computation with Qudits*. *Journal of Physics A: Mathematical and Theoretical* 56(11), p. 115303, doi:10.1088/1751-8121/acbace. arXiv:2109.13810.

[25] Charles Bouillaguet & Claire Delaplace (2016): *Sparse Gaussian Elimination Modulo p: An Update*. In Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler & Evgenii V. Vorozhtsov, editors: *Computer Algebra in Scientific Computing*, Springer International Publishing, Cham, pp. 101–116, doi:10.1007/978-3-319-45641-6_8. Available at `https://hal.science/hal-01333670`.

[26] Hans J. Briegel & Robert Raussendorf (2001): *Persistent Entanglement in Arrays of Interacting Particles*. *Phys. Rev. Lett.* 86, pp. 910–913, doi:10.1103/PhysRevLett.86.910. Available at `https://link.aps.org/doi/10.1103/PhysRevLett.86.910`.

[27] Anne Broadbent, Joseph Fitzsimons & Elham Kashefi (2009): *Universal Blind Quantum Computation*. In: 2009 50*th Annual IEEE Symposium on Foundations of Computer Science*, pp. 517–526, doi:10.1109/FOCS.2009.36.

[28] Anne Broadbent & Elham Kashefi (2009): *Parallelizing Quantum Circuits*. *Theoretical Computer Science* 410(26), pp. 2489–2510, doi:10.1016/j.tcs.2008.12.046.

[29] Daniel E. Browne, Elham Kashefi, Mehdi Mhalla & Simon Perdrix (2007): *Generalized Flow and Determinism in Measurement-Based Quantum Computation*. *New Journal of Physics* 9(8), p. 250, doi:10.1088/1367-2630/9/8/250.

[30] Jonathan F. Buss, Gudmund S. Frandsen & Jeffrey O. Shallit (1999): *The Computational Complexity of Some Problems of Linear Algebra*. *Journal of Computer and System Sciences* 58(3), pp. 572–596, doi:10.1006/jcss.1998.1608.

[31] Jin-Yi Cai & Xi Chen (2017): *Complexity Dichotomies for Counting Problems: Volume 1: Boolean Domain*. 1, Cambridge University Press, Cambridge, doi:10.1017/9781107477063.

[32] Titouan Carette (2021): *Wielding the ZX-calculus, Flexsymmetry, Mixed States, and Scalable Notations*. Theses, Université de Lorraine. Available at `https://hal.science/tel-03468027`.

[33] Titouan Carette, Dominic Horsman & Simon Perdrix (2019): *SZX-Calculus: Scalable Graphical Quantum Reasoning*. In Peter Rossmanith, Pinar Heggernes & Joost-Pieter Katoen, editors: 44*th International Symposium on Mathematical Foundations of Computer Science* (*MFCS* 2019), *Leibniz International Proceedings in Informatics* (*LIPIcs*) 138, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 55:1–55:15, doi:10.4230/LIPIcs.MFCS.2019.55.

[34] Titouan Carette & Emmanuel Jeandel (2020): *On a Recipe for Quantum Graphical Languages*. *LIPIcs, Volume* 168, *ICALP* 2020 168, pp. 118:1–118:17, doi:10.4230/LIPIcs.ICALP.2020.118. arXiv:2008.04193.

[35] Titouan Carette, Etienne Moutot, Thomas Perez & Renaud Vilmart (2023): *Compositionality of Planar Perfect Matchings*, doi:10.48550/arXiv.2302.08767. arXiv:2302.08767.

[36] Sílvia Casacuberta & Rasmus Kyng (2022): *Faster Sparse Matrix Inversion and Rank Computation in Finite Fields*. *LIPIcs, Volume* 215, *ITCS* 2022 215, pp. 33:1–33:24, doi:10.4230/LIPICS.ITCS.2022.33.

[37] Nicholas Chancellor, Aleks Kissinger, Joschka Roffe, Stefan Zohren & Dominic Horsman (2023): *Graphical Structures for Design and Verification of Quantum Error Correction*. *Quantum Science and Technology* 8(4), p. 045028, doi:10.1088/2058-9565/acf157. arXiv:1611.08012.

[38] Yanbin Chen (2021): *Hypergraph MBQC in the ZH-calculus*. Master's thesis, University of Oxford. Available at https://vdwetering.name/pdfs/thesis-Chen.pdf.

[39] Andrew Chi-Chih Yao (1993): *Quantum Circuit Complexity*. In: *Proceedings of* 1993 *IEEE* 34*th Annual Foundations of Computer Science*, pp. 352–361, doi:10.1109/SFCS.1993.366852.

[40] Nai-Hui Chia, Chi-Ning Chou, Jiayu Zhang & Ruizhe Zhang (2021): *Quantum Meets the Minimum Circuit Size Problem*. arXiv:2108.03171.

[41] Nai-Hui Chia, Chi-Ning Chou, Jiayu Zhang & Ruizhe Zhang (2022): *Quantum Meets the Minimum Circuit Size Problem*. In Mark Braverman, editor: 13*th Innovations in Theoretical Computer Science Conference* (*ITCS* 2022), *Leibniz International Proceedings in Informatics* (*LIPIcs*) 215, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 47:1–47:16, doi:10.4230/LIPIcs.ITCS.2022.47. Available at https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITCS.2022.47.

[42] Alonzo Church (1936): *An Unsolvable Problem of Elementary Number Theory*. *American Journal of Mathematics* 58(2), pp. 345–363. Available at http://www.jstor.org/stable/2371045.

[43] Alexandre Clément, Nicolas Heurtel, Shane Mansfield, Simon Perdrix & Benoît Valiron (2023): *A Complete Equational Theory for Quantum Circuits*. In: 2023 38*th Annual ACM/IEEE Symposium on Logic in Computer Science* (*LICS*), pp. 1–13, doi:10.1109/LICS56636.2023.10175801.

[44] Bob Coecke & Ross Duncan (2007): *A Graphical Calculus for Quantum Observables*. https://www.cs.ox.ac.uk/people/bob.coecke/GreenRed.pdf.

[45] Bob Coecke & Ross Duncan (2011): *Interacting Quantum Observables: Categorical Algebra and Diagrammatics*. *New Journal of Physics* 13(4), p. 043016, doi:10.1088/1367-2630/13/4/043016. arXiv:0906.4725.

[46] Bob Coecke & Aleks Kissinger (2017): *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, Cambridge, doi:10.1017/9781316219317.

[47] Bob Coecke & Quanlong Wang (2018): *ZX-Rules for 2-Qubit Clifford+T Quantum Circuits*. In Jarkko Kari & Irek Ulidowski, editors: *Reversible Computation*,

Springer International Publishing, Cham, pp. 144–161, doi:10.1007/978-3-319-99498-7_10.

[48] Cole Comfort (2023): *The Algebra for Stabilizer Codes*, doi:10.48550/arXiv.2304.10584. arXiv:2304.10584.

[49] Stephen A. Cook (1971): *The Complexity of Theorem-Proving Procedures*. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, Association for Computing Machinery, New York, NY, USA, pp. 151–158, doi:10.1145/800157.805047.

[50] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein (2022): *Introduction to Algorithms, Fourth Edition*. MIT Press.

[51] Alexander Cowtan & Simon Burton (2024): *CSS Code Surgery as a Universal Construction*. *Quantum* 8, p. 1344, doi:10.22331/q-2024-05-14-1344. arXiv:2301.13738.

[52] Vincent Danos & Elham Kashefi (2006): *Determinism in the One-Way Model*. *Physical Review A* 74(5), p. 052310, doi:10.1103/PhysRevA.74.052310.

[53] Vincent Danos, Elham Kashefi & Prakash Panangaden (2007): *The Measurement Calculus*. *J. ACM* 54(2), pp. 8–es, doi:10.1145/1219092.1219096.

[54] Vincent Danos, Elham Kashefi, Prakash Panangaden & Simon Perdrix (2009): *Extended Measurement Calculus*. In Ian Mackie & Simon Gay, editors: *Semantic Techniques in Quantum Computation*, Cambridge University Press, Cambridge, pp. 235–310, doi:10.1017/CBO9781139193313.008.

[55] Niel de Beaudrap (2007): *A Complete Algorithm to Find Flows in the One-Way Measurement Model*, doi:10.48550/arXiv.quant-ph/0603072. arXiv:quant-ph/0603072.

[56] Niel de Beaudrap (2008): *Finding Flows in the One-Way Measurement Model*. *Physical Review A* 77(2), p. 022328, doi:10.1103/PhysRevA.77.022328.

[57] Niel de Beaudrap, Xiaoning Bian & Quanlong Wang (2020): *Fast and Effective Techniques for T-Count Reduction via Spider Nest Identities*. In Steven T. Flammia, editor: 15*th Conference on the Theory of Quantum Computation, Communication and Cryptography* (*TQC* 2020), *Leibniz International Proceedings in Informatics* (*LIPIcs*) 158, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 11:1–11:23, doi:10.4230/LIPIcs.TQC.2020.11. Available at `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.TQC.2020.11`.

[58] Niel de Beaudrap, Xiaoning Bian & Quanlong Wang (2020): *Techniques to Reduce $\pi/4$-Parity-Phase Circuits, Motivated by the ZX Calculus*. *Electronic Proceedings in Theoretical Computer Science* 318, pp. 131–149, doi:10.4204/EPTCS.318.9. arXiv:1911.09039.

[59] Niel de Beaudrap, Ross Duncan, Dominic Horsman & Simon Perdrix (2020): *Pauli Fusion: A Computational Model to Realise Quantum Transformations from ZX Terms*. *Electronic Proceedings in Theoretical Computer Science* 318, pp. 85–105, doi:10.4204/EPTCS.318.6. arXiv:1904.12817.

[60] Niel de Beaudrap & Richard D. P. East (2024): *Simple Qudit ZX and ZH Calculi, via Integrals*. In Rastislav Královič & Antonín Kučera, editors: 49*th International Symposium on Mathematical Foundations of Computer Science* (*MFCS* 2024), *Leibniz International Proceedings in Informatics* (*LIPIcs*) 306, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 20:1–20:20, doi:10.4230/LIPIcs.MFCS.2024.20. Available at `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.MFCS.2024.20`.

[61] Niel de Beaudrap, Aleks Kissinger & Konstantinos Meichanetzidis (2021): *Tensor Network Rewriting Strategies for Satisfiability and Counting*. *Electronic Proceedings in Theoretical Computer Science* 340, pp. 46–59, doi:10.4204/EPTCS.340.3. arXiv:2004.06455.

[62] Niel de Beaudrap, Aleks Kissinger & John van de Wetering (2022): *Circuit Extraction for ZX-Diagrams Can Be* #P*-Hard*. In: 49*th International Colloquium on Automata, Languages, and Programming* (*ICALP* 2022), *LIPIcs* 229, pp. 119:1–119:19, doi:10.4230/LIPIcs.ICALP.2022.119.

[63] Richard A. Demillo & Richard J. Lipton (1978): *A Probabilistic Remark on Algebraic Program Testing*. *Information Processing Letters* 7(4), pp. 193–195, doi:10.1016/0020-0190(78)90067-4.

[64] Ross Duncan, Aleks Kissinger, Simon Perdrix & John van de Wetering (2020): *Graph-Theoretic Simplification of Quantum Circuits with the ZX-calculus*. *Quantum* 4, p. 279, doi:10.22331/q-2020-06-04-279. arXiv:1902.03178.

[65] Ross Duncan & Maxime Lucas (2014): *Verifying the Steane Code with Quantomatic*. *Electronic Proceedings in Theoretical Computer Science* 171, pp. 33–49, doi:10.4204/EPTCS.171.4. arXiv:1306.4532.

[66] Selma Dündar-Coecke, Lia Yeh, Caterina Puca, Sieglinde M.-L. Pfaendler, Muhammad Hamza Waseem, Thomas Cervoni, Aleks Kissinger, Stefano Gogioso & Bob

Coecke (2023): *Quantum Picturalism: Learning Quantum Theory in High School*. In: 2023 *IEEE International Conference on Quantum Computing and Engineering (QCE)*, pp. 21–32, doi:10.1109/QCE57702.2023.20321. arXiv:2312.03653.

[67] Max Erné (2024): *Extraction of ZX-diagrams without Gflow*. Master's thesis, University of Amsterdam. Available at `https://vdwetering.name/pdfs/thesis-Erne.pdf`.

[68] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis & Pushmeet Kohli (2022): *Discovering Faster Matrix Multiplication Algorithms with Reinforcement Learning*. *Nature* 610(7930), pp. 47–53, doi:10.1038/s41586-022-05172-4.

[69] Giovanni de Felice, Boldizsár Poór, Lia Yeh & William Cashman (2024): *Fusion and Flow: Formal Protocols to Reliably Build Photonic Graph States*, doi:10.48550/arXiv.2409.13541. arXiv:2409.13541.

[70] Stephen Fenner, Frederic Green, Steven Homer & Randall Pruim (1999): *Determining Acceptance Possibility for a Quantum Computation Is Hard for the Polynomial Hierarchy*. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 455(1991), pp. 3953–3966, doi:10.1098/rspa.1999.0485.

[71] Richard P. Feynman (1986): *Quantum Mechanical Computers*. *Foundations of Physics* 16(6), pp. 507–531, doi:10.1007/BF01886518.

[72] Tobias Fischbach, Pierre Talbot & Pascal Bouvry (2025): *A Review on Quantum Circuit Optimization using ZX-Calculus*. arXiv:2509.20663.

[73] Joseph F. Fitzsimons & Elham Kashefi (2017): *Unconditionally Verifiable Blind Quantum Computation*. *Physical Review A* 96(1), p. 012303, doi:10.1103/PhysRevA.96.012303.

[74] Julio C. Magdalena de la Fuente, Josias Old, Alex Townsend-Teague, Manuel Rispler, Jens Eisert & Markus Müller (2025): *XYZ Ruby Code: Making a Case for a Three-Colored Graphical Calculus for Quantum Error Correction in Spacetime*. *PRX Quantum* 6(1), p. 010360, doi:10.1103/PRXQuantum.6.010360. arXiv:2407.08566.

[75] Francois Le Gall & Florent Urrutia (2018): *Improved Rectangular Matrix Multiplication using Powers of the Coppersmith-Winograd Tensor*, p. 1029–1046. Society for Industrial and Applied Mathematics, doi:10.1137/1.9781611975031.67. Available at `http://dx.doi.org/10.1137/1.9781611975031.67`.

[76] Michael R. Garey, David S. Johnson & Larry Stockmeyer (1976): *Some Simplified* NP*-Complete Graph Problems*. *Theoretical Computer Science* 1(3), pp. 237–267, doi:10.1016/0304-3975(76)90059-1.

[77] Liam Garvie & Ross Duncan (2018): *Verifying the Smallest Interesting Colour Code with Quantomatic*. *Electronic Proceedings in Theoretical Computer Science* 266, pp. 147–163, doi:10.4204/EPTCS.266.10. arXiv:1706.02717.

[78] Sergey B. Gashkov & Igor S. Sergeev (2013): *Complexity of Computation in Finite Fields*. *Journal of Mathematical Sciences* 191(5), pp. 661–685, doi:10.1007/s10958-013-1350-5.

[79] Alexandru Gheorghiu, Theodoros Kapourniotis & Elham Kashefi (2019): *Verification of Quantum Computation: An Overview of Existing Approaches*. *Theory of Computing Systems* 63(4), pp. 715–808, doi:10.1007/s00224-018-9872-3.

[80] John Gill (1977): *Computational Complexity of Probabilistic Turing Machines*. *SIAM Journal on Computing* 6(4), pp. 675–695, doi:10.1137/0206049.

[81] Daniel M. Greenberger, Michael A. Horne & Anton Zeilinger (1989): *Going Beyond Bell's Theorem*. In Menas Kafatos, editor: *Bell's Theorem, Quantum Theory and Conceptions of the Universe*, Springer Netherlands, Dordrecht, pp. 69–72, doi:10.1007/978-94-017-0849-4_10.

[82] D. Gross, S. T. Flammia & J. Eisert (2009): *Most Quantum States Are Too Entangled To Be Useful As Computational Resources*. *Phys. Rev. Lett.* 102, p. 190501, doi:10.1103/PhysRevLett.102.190501. Available at `https://link.aps.org/doi/10.1103/PhysRevLett.102.190501`.

[83] Lov K. Grover (1996): *A Fast Quantum Mechanical Algorithm for Database Search*. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, Association for Computing Machinery, New York, NY, USA, pp. 212–219, doi:10.1145/237814.237866.

[84] Brent Harrison, Vishnu Iyer, Ojas Parekh, Kevin Thompson & Andrew Zhao (2025): *Fermionic Insights into Measurement-Based Quantum Computation: Circle Graph States Are Not Universal Resources*. arXiv:2510.05557.

[85] Nicholas J. A. Harvey, David R. Karger & Sergey Yekhanin (2006): *The Complexity of Matrix Completion*. In: *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm - SODA* '06, ACM Press, Miami, Florida, pp. 1103–1111, doi:10.1145/1109557.1109679.

[86] M. Hein, Jens Eisert & Hans J. Briegel (2004): *Multi-Party Entanglement in Graph States*. *Physical Review A* 69(6), p. 062311, doi:10.1103/PhysRevA.69.062311. arXiv:quant-ph/0307130.

[87] Paul Herringer, Vir B. Bulchandani, Younes Javanmard, David T. Stephen & Robert Raussendorf (2024): *Duality between String and Computational Order in Symmetry-Enriched Topological Phases*, doi:10.48550/arXiv.2410.02716. arXiv:2410.02716.

[88] Chris Heunen, Jamie Vicary, Chris Heunen & Jamie Vicary (2019): *Categories for Quantum Theory: An Introduction*. Oxford Graduate Texts in Mathematics, Oxford University Press, Oxford, New York.

[89] Calum Holker (2024): *Causal Flow Preserving Optimisation of Quantum Circuits in the ZX-calculus*, doi:10.48550/arXiv.2312.02793. arXiv:2312.02793.

[90] Matt Hostetter (2020): *Galois: A performant NumPy extension for Galois fields*. `https://github.com/mhostetter/galois`. Accessed: 6th May 2025.

[91] Peter Hoyer & Robert Spalek (2005): *Quantum Fan-out is Powerful*. *Theory of Computing* 1(1), p. 81–103, doi:10.4086/toc.2005.v001a005. Available at `http://dx.doi.org/10.4086/toc.2005.v001a005`.

[92] Hsin-Yuan Huang, Michael Broughton, Jordan Cotler, Sitan Chen, Jerry Li, Masoud Mohseni, Hartmut Neven, Ryan Babbush, Richard Kueng, John Preskill & Jarrod R. McClean (2022): *Quantum Advantage in Learning from Experiments*. *Science* 376(6598), pp. 1182–1186, doi:10.1126/science.abn7293.

[93] Jianyu Huang, Tyler M. Smith, Greg M. Henry & Robert A. Van De Geijn (2016): *Strassen's Algorithm Reloaded*. In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 690–701, doi:10.1109/SC.2016.58.

[94] Jiaxin Huang, Sarah Meng Li, Lia Yeh, Aleks Kissinger, Michele Mosca & Michael Vasmer (2023): *Graphical CSS Code Transformation Using ZX Calculus*. *Electronic Proceedings in Theoretical Computer Science* 384, pp. 1–19, doi:10.4204/EPTCS.384.1. arXiv:2307.02437.

[95] Gábor Ivanyos, Marek Karpinski & Nitin Saxena (2010): *Deterministic Polynomial Time Algorithms for Matrix Completion Problems*. *SIAM Journal on Computing*, doi:10.1137/090781231.

[96] Dominik Janzing, Pawel Wocjan & Thomas Beth (2005): *"non-Identity-Check" Is Qma-Complete*. *International Journal of Quantum Information* 03(03), pp. 463–473, doi:10.1142/S0219749905001067.

[97] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson & Jay M. Gambetta (2024): *Quantum Computing with Qiskit*, doi:10.48550/arXiv.2405.08810. arXiv:2405.08810.

[98] Emmanuel Jeandel, Simon Perdrix & Renaud Vilmart (2018): *A Complete Axiomatisation of the ZX-Calculus for Clifford+T Quantum Mechanics*. In: *Proceedings of the* 33*rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, Association for Computing Machinery, New York, NY, USA, pp. 559–568, doi:10.1145/3209108.3209131.

[99] Emmanuel Jeandel, Simon Perdrix & Renaud Vilmart (2018): *Diagrammatic Reasoning beyond Clifford+T Quantum Mechanics*. In: *Proceedings of the* 33*rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, Association for Computing Machinery, New York, NY, USA, pp. 569–578, doi:10.1145/3209108.3209139.

[100] Theodoros Kapourniotis, Elham Kashefi, Dominik Leichtle, Luka Music & Harold Ollivier (2024): *Unifying Quantum Verification and Error-Detection: Theory and Tools for Optimisations*. *Quantum Science and Technology* 9(3), p. 035036, doi:10.1088/2058-9565/ad466d.

[101] Elham Kashefi, Damian Markham, Mehdi Mhalla & Simon Perdrix (2009): *Information Flow in Secret Sharing Protocols*. *Electronic Proceedings in Theoretical Computer Science* 9, pp. 87–97, doi:10.4204/EPTCS.9.10. arXiv:0909.4479.

[102] Elham Kashefi & Anna Pappa (2017): *Multiparty Delegated Quantum Computing*. *Cryptography* 1(2), p. 12, doi:10.3390/cryptography1020012.

[103] Andrey Boris Khesin & Alexander Li (2024): *Equivalence Classes of Quantum Error-Correcting Codes*, doi:10.48550/arXiv.2406.12083. arXiv:2406.12083.

[104] Aleks Kissinger (2022): *Phase-Free ZX Diagrams Are CSS Codes (...or How to Graphically Grok the Surface Code)*, doi:10.48550/arXiv.2204.14038. arXiv:2204.14038.

[105] Aleks Kissinger & John van de Wetering (2019): *Universal MBQC with Generalised Parity-Phase Interactions and Pauli Measurements*. *Quantum* 3, p. 134, doi:10.22331/q-2019-04-26-134. arXiv:1704.06504.

[106] Aleks Kissinger & John van de Wetering (2020): *Reducing T-count with the ZX-calculus*. *Physical Review A* 102(2), p. 022406, doi:10.1103/PhysRevA.102.022406. arXiv:1903.10477.

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

[108] Aleks Kissinger & John van de Wetering (2020): *PyZX: Large Scale Automated Diagrammatic Reasoning*. *Electronic Proceedings in Theoretical Computer Science* 318, pp. 229–241, doi:10.4204/EPTCS.318.14. arXiv:1904.04735.

[109] Aleks Kissinger & Vladimir Zamdzhiev (2015): *Quantomatic: A Proof Assistant for Diagrammatic Reasoning*. In Amy P. Felty & Aart Middeldorp, editors: *Automated Deduction - CADE*-25, Springer International Publishing, Cham, pp. 326–336, doi:10.1007/978-3-319-21401-6_22.

[110] Mark Koch (2022): *Quantum Machine Learning Using the ZXW-Calculus*, doi:10.48550/arXiv.2210.11523. arXiv:2210.11523.

[111] Mark W. Krentel (1988): *The Complexity of Optimization Problems*. *Journal of Computer and System Sciences* 36(3), pp. 490–509, doi:10.1016/0022-0000(88)90039-6.

[112] Stach Kuijpers, John van de Wetering & Aleks Kissinger (2019): *Graphical Fourier Theory and the Cost of Quantum Addition*. *arXiv*:1904.07551 [*quant-ph*]. arXiv:1904.07551.

[113] Mateusz Kupper (2020): *Analysis of Quantum Hypergraph States in the ZH-calculus*. Master's thesis, University of Edinburgh.

[114] Tuomas Laakkonen, Konstantinos Meichanetzidis & John van de Wetering (2022): *A Graphical #SAT Algorithm for Formulae with Small Clause Density*, doi:10.48550/arXiv.2212.08048. arXiv:2212.08048.

[115] Tuomas Laakkonen, Konstantinos Meichanetzidis & John van de Wetering (2023): *Picturing Counting Reductions with the ZH-Calculus*. *Electronic Proceedings in Theoretical Computer Science* 384, pp. 89–113, doi:10.4204/EPTCS.384.6. arXiv:2304.02524.

[116] Clemens Lautemann (1983): *BPP and the Polynomial Hierarchy*. *Information Processing Letters* 17(4), pp. 215–217, doi:10.1016/0020-0190(83)90044-3.

[117] Frank Luebeck: *Conway Polynomials for Finite Fields*. `https://www.math.rwth-aachen.de/~Frank.Luebeck/data/ConwayPol/index.html`. Accessed: 12th February 2024.

[118] Meena Mahajan & Jayalal M. N. Sarma (2010): *On the Complexity of Matrix Rank and Rigidity*. *Theory of Computing Systems* 46(1), pp. 9–26, doi:10.1007/s00224-008-9136-8.

[119] Atul Mantri, Tommaso F. Demarie & Joseph F. Fitzsimons (2017): *Universality of Quantum Computation with Cluster States and (X, Y)-Plane Measurements*. *Scientific Reports* 7(1), p. 42861, doi:10.1038/srep42861.

[120] Damian Markham & Elham Kashefi (2014): *Entanglement, Flow and Classical Simulatability in Measurement Based Quantum Computation*. In Franck van Breugel, Elham Kashefi, Catuscia Palamidessi & Jan Rutten, editors: *Horizons of the Mind. A Tribute to Prakash Panangaden: Essays Dedicated to Prakash Panangaden on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 427–453, doi:10.1007/978-3-319-06880-0_22.

[121] Tommy McElvanney & Miriam Backens (2023): *Complete Flow-Preserving Rewrite Rules for MBQC Patterns with Pauli Measurements*. *Electronic Proceedings in Theoretical Computer Science* 394, pp. 66–82, doi:10.4204/EPTCS.394.5.

[122] Tommy McElvanney & Miriam Backens (2023): *Flow-Preserving ZX-calculus Rewrite Rules for Optimisation and Obfuscation*. *Electronic Proceedings in Theoretical Computer Science* 384, pp. 203–219, doi:10.4204/EPTCS.384.12. arXiv:2304.08166.

[123] Konstantinos Meichanetzidis, Stefano Gogioso, Giovanni de Felice, Nicolò Chiappori, Alexis Toumi & Bob Coecke (2021): *Quantum Natural Language Processing on Near-Term Quantum Computers*. *Electronic Proceedings in Theoretical Computer Science* 340, pp. 213–229, doi:10.4204/EPTCS.340.11. arXiv:2005.04147.

[124] A. R. Meyer & L. J. Stockmeyer (1972): *The equivalence problem for regular expressions with squaring requires exponential space*. In: *13th Annual Symposium on Switching and Automata Theory (swat* 1972), pp. 125–129, doi:10.1109/SWAT.1972.29.

[125] Mehdi Mhalla, Mio Murao, Simon Perdrix, Masato Someya & Peter S. Turner (2014): *Which Graph States Are Useful for Quantum Information Processing?* In Dave Bacon, Miguel Martin-Delgado & Martin Roetteler, editors: *Theory of Quantum Computation, Communication, and Cryptography*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 174–187, doi:10.1007/978-3-642-54429-3_12.

[126] Mehdi Mhalla & Simon Perdrix (2008): *Finding Optimal Flows Efficiently*. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir & Igor Walukiewicz, editors: *Automata, Languages and Programming*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 857–868, doi:10.1007/978-3-540-70575-8_70.

[127] Mehdi Mhalla & Simon Perdrix (2012): *Graph States, Pivot Minor, and Universality of (X,Z)-Measurements*, doi:10.48550/arXiv.1202.6551. arXiv:1202.6551.

[128] Mehdi Mhalla, Simon Perdrix & Luc Sanselme (2025): *Shadow Pauli Flow: Characterising Determinism in MBQCs Involving Pauli Measurements*, doi:10.48550/arXiv.2207.09368. arXiv:2207.09368.

[129] Hector Miller-Bakewell (2020): *Entanglement and Quaternions: The Graphical Calculus ZQ*, doi:10.48550/arXiv.2003.09999. arXiv:2003.09999.

[130] Hector Miller-Bakewell & John van de Wetering: *The ZX-calculus*. `https://zxcalculus.com/`. Accessed: 28th February 2025.

[131] Piotr Mitosek (2024): *Constructing* $\mathrm{NP}^{\#\mathrm{P}}$*-Complete Problems and* #P*-Hardness of Circuit Extraction in Phase-Free ZH*. arXiv:2404.10913.

[132] Piotr Mitosek (2024): *Pauli Flow on Open Graphs with Unknown Measurement Labels*. *Electronic Proceedings in Theoretical Computer Science* 406, pp. 117–136, doi:10.4204/EPTCS.406.6. arXiv:2408.06059.

[133] Piotr Mitosek & Miriam Backens (2024): *An Algebraic Interpretation of Pauli Flow, Leading to Faster Flow-Finding Algorithms*, doi:10.48550/arXiv.2410.23439. arXiv:2410.23439.

[134] David Monniaux (2022): $\mathrm{NP}^{\#\mathrm{P}} = \exists\mathrm{PP}$ *and Other Remarks about Maximized Counting*, doi:10.48550/arXiv.2202.11955. arXiv:2202.11955.

[135] Tomoyuki Morimae, Yuki Takeuchi & Masahito Hayashi (2017): *Verified Measurement-Based Quantum Computing with Hypergraph States*. *Physical Review A* 96(6), p. 062321, doi:10.1103/PhysRevA.96.062321. arXiv:1701.05688.

[136] Kang Feng Ng & Quanlong Wang (2017): *A Universal Completion of the ZX-calculus*. *arXiv*:1706.09877 [*quant-ph*]. arXiv:1706.09877.

[137] Michael A. Nielsen & Isaac L. Chuang (2010): *Quantum Computation and Quantum Information*, 10th anniversary ed edition. Cambridge University Press, Cambridge ; New York.

[138] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli & Matej Balog (2025): *AlphaEvolve: A coding agent for scientific and algorithmic discovery*. arXiv:2506.13131.

[139] Øystein Ore (1921): *Über Höhere Kongruenzen*. Norsk Matematisk Forenings Skrifter, Grøndahl.

[140] Boldizsár Poór, Quanlong Wang, Razin A. Shaikh, Lia Yeh, Richie Yeung & Bob Coecke (2023): *Completeness for Arbitrary Finite Dimensions of ZXW-calculus, a Unifying Calculus*. In: 2023 38*th Annual ACM/IEEE Symposium on Logic in Computer Science* (*LICS*), pp. 1–14, doi:10.1109/LICS56636.2023.10175672.

[141] John Preskill (2012): *Quantum Computing and the Entanglement Frontier*, doi:10.48550/arXiv.1203.5813. arXiv:1203.5813.

[142] Michael O Rabin (1980): *Probabilistic Algorithm for Testing Primality*. *Journal of Number Theory* 12(1), pp. 128–138, doi:10.1016/0022-314X(80)90084-0.

[143] Robert Raussendorf & Hans J. Briegel (2001): *A One-Way Quantum Computer*. *Physical Review Letters* 86(22), pp. 5188–5191, doi:10.1103/PhysRevLett.86.5188.

[144] Robert Raussendorf, Daniel E. Browne & Hans J. Briegel (2002): *The One-Way Quantum Computer - a Non-Network Model of Quantum Computation*. *Journal of Modern Optics* 49(8), pp. 1299–1306, doi:10.1080/09500340110107487. arXiv:quant-ph/0108118.

[145] Robert Raussendorf, Daniel E. Browne & Hans J. Briegel (2003): *Measurement-Based Quantum Computation on Cluster States*. *Physical Review A* 68(2), p. 022312, doi:10.1103/PhysRevA.68.022312.

[146] Robert Raussendorf, Jim Harrington & Kovid Goyal (2006): *A Fault-Tolerant One-Way Quantum Computer*. *Annals of Physics* 321(9), pp. 2242–2270, doi:10.1016/j.aop.2006.01.012.

[147] Alexander A Razborov & Steven Rudich (1997): *Natural Proofs*. *Journal of Computer and System Sciences* 55(1), pp. 24–35, doi:https://doi.org/10.1006/jcss.1997.1494. Available at `https://www.sciencedirect.com/science/article/pii/S002200009791494X`.

[148] Ronald L. Rivest, Adi Shamir & Leonard Adleman (1978): *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. *Commun. ACM* 21(2), pp. 120–126, doi:10.1145/359340.359342.

[149] Matteo Rossi, Marcus Huber, Dagmar Bruß & Chiara Macchiavello (2013): *Quantum Hypergraph States*. *New Journal of Physics* 15(11), p. 113022, doi:10.1088/1367-2630/15/11/113022.

[150] Jacob T. Schwartz (1980): *Fast Probabilistic Algorithms for Verification of Polynomial Identities*. *Journal of the ACM* 27(4), pp. 701–717, doi:10.1145/322217.322225.

[151] Peter Selinger (2007): *Dagger Compact Closed Categories and Completely Positive Maps: (Extended Abstract)*. *Electronic Notes in Theoretical Computer Science* 170, pp. 139–163, doi:10.1016/j.entcs.2006.12.018.

[152] Peter W. Shor (1997): *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*. *SIAM Journal on Computing* 26(5), pp. 1484–1509, doi:10.1137/S0097539795293172. arXiv:quant-ph/9508027.

[153] Will Simmons (2021): *Relating Measurement Patterns to Circuits via Pauli Flow*. *Electronic Proceedings in Theoretical Computer Science* 343, pp. 50–101, doi:10.4204/EPTCS.343.4. arXiv:2109.05654.

[154] Will Simmons (2022): *[TKET-597] MBQC Primitives and Flow Analysis #218*. `https://github.com/CQCL/tket/pull/218#issue-1137544514`. Accessed: 7th January 2025.

[155] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington & Ross Duncan (2021): *$T|ket\rangle$: A Retargetable Compiler for NISQ Devices*. *Quantum Science and Technology* 6(1), p. 014003, doi:10.1088/2058-9565/ab8e92. arXiv:2003.10611.

[156] Isaac D. Smith, Hendrik Poulsen Nautrup & Hans J. Briegel (2024): *Parity Quantum Computing as YZ-Plane Measurement-Based Quantum Computing*. *Physical Review Letters* 132(22), p. 220602, doi:10.1103/PhysRevLett.132.220602. arXiv:2401.10079.

[157] Korbinian Staudacher, Tobias Guggemos, Sophia Grundner-Culemann & Wolfgang Gehrke (2023): *Reducing 2-QuBit Gate Count for ZX-Calculus Based Quantum Circuit Optimization*. *EPTCS* 394, pp. 29–45, doi:10.4204/EPTCS.394.3.

[158] Larry J. Stockmeyer (1976): *The Polynomial-Time Hierarchy*. *Theoretical Computer Science* 3(1), pp. 1–22, doi:10.1016/0304-3975(76)90061-X.

[159] Volker Strassen (1969): *Gaussian Elimination Is Not Optimal*. *Numerische Mathematik* 13(4), pp. 354–356, doi:10.1007/BF02165411.

[160] Shinichi Sunami & Masato Fukushima (2023): *Graphix*, doi:10.5281/zenodo.7861382.

[161] Matthew Sutcliffe & Aleks Kissinger (2024): *Procedurally Optimised ZX-Diagram Cutting for Efficient T-Decomposition in Classical Simulation*. *Electronic Proceedings in Theoretical Computer Science* 406, pp. 63–78, doi:10.4204/EPTCS.406.3. arXiv:2403.10964.

[162] S. Takeda & A. Furusawa (2019): *Toward Large-Scale Fault-Tolerant Universal Photonic Quantum Computing*. *APL Photonics* 4(6), p. 060902, doi:10.1063/1.5100160.

[163] Yuki Takeuchi, Tomoyuki Morimae & Masahito Hayashi (2019): *Quantum Computational Universality of Hypergraph States with Pauli-X and Z Basis Measurements*. *Scientific Reports* 9(1), p. 13585, doi:10.1038/s41598-019-49968-3.

[164] Yu Tanaka (2010): *EXACT NON-IDENTITY CHECK IS NQP-COMPLETE*. *International Journal of Quantum Information* 08(05), pp. 807–819, doi:10.1142/S0219749910006599.

[165] Seinosuke Toda (1991): *PP Is as Hard as the Polynomial-Time Hierarchy*. *SIAM Journal on Computing* 20(5), pp. 865–877, doi:10.1137/0220053.

[166] Jacobo Torán (1991): *Complexity Classes Defined by Counting Quantifiers*. *Journal of the ACM* 38(3), pp. 752–773, doi:10.1145/116825.116858.

[167] Alan M. Turing (1937): *On Computable Numbers, with an Application to the Entscheidungsproblem*. *Proceedings of the London Mathematical Society* s2-42(1), pp. 230–265, doi:10.1112/plms/s2-42.1.230.

[168] Christian Ufrecht, Maniraman Periyasamy, Sebastian Rietsch, Daniel D. Scherer, Axel Plinge & Christopher Mutschler (2023): *Cutting Multi-Control Quantum Gates with ZX Calculus*. *Quantum* 7, p. 1147, doi:10.22331/q-2023-10-23-1147.

[169] Mateo Uldemolins (2025): *$O(N^3)$ Pauli-flow finding algorithm (#337)*. `https://github.com/TeamGraphix/graphix/commit/c9e392645e4fc0ea7163b55735ce8a4270b10750`. Accessed: 14th October 2025.

[170] Leslie G. Valiant (1979): *The Complexity of Computing the Permanent*. *Theoretical Computer Science* 8(2), pp. 189–201, doi:10.1016/0304-3975(79)90044-6.

[171] John van de Wetering (2020): *ZX-calculus for the Working Quantum Computer Scientist*, doi:10.48550/arXiv.2012.13966. arXiv:2012.13966.

[172] John van de Wetering, Richie Yeung, Tuomas Laakkonen & Aleks Kissinger (2024): *Optimal Compilation of Parametrised Quantum Circuits*, doi:10.48550/arXiv.2401.12877. arXiv:2401.12877.

[173] Martin van IJcken (2024): *Generalized Flow for Hypergraph Measurement Patterns*. Master's thesis, University of Oxford. Available at `www.cs.ox.ac.uk/people/aleks.kissinger/theses/vanijcken-thesis.pdf`.

[174] Vivien Vandaele (2024): *Qubit-Count Optimization Using ZX-calculus*, doi:10.48550/arXiv.2407.10171. arXiv:2407.10171.

[175] Klaus W. Wagner (1987): *More Complicated Questions about Maxima and Minima, and Some Closures of NP*. *Theoretical Computer Science* 51(1), pp. 53–80, doi:10.1016/0304-3975(87)90049-1.

[176] Quanlong Wang (2018): *Qutrit ZX-calculus Is Complete for Stabilizer Quantum Mechanics*. *Electronic Proceedings in Theoretical Computer Science* 266, pp. 58–70, doi:10.4204/EPTCS.266.3. arXiv:1803.00696.

[177] Quanlong Wang, Boldizsár Poór & Razin A. Shaikh (2024): *Completeness of Qufinite ZXW Calculus, a Graphical Language for Finite-Dimensional Quantum Theory*, doi:10.48550/arXiv.2309.13014. arXiv:2309.13014.

[178] John Watrous (2000): *Succinct Quantum Proofs for Properties of Finite Groups*. In: *Proceedings* 41*st Annual Symposium on Foundations of Computer Science*, pp. 537–546, doi:10.1109/SFCS.2000.892141.

[179] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu & Renfei Zhou (2024): *New Bounds for Matrix Multiplication: From Alpha to Omega*. In: *Proceedings of the* 2024 *Annual ACM-SIAM Symposium on Discrete Algorithms* (*SODA*), SIAM, pp. 3792–3835, doi:10.1137/1.9781611977912.134.

[180] Celia Wrathall (1976): *Complete sets and the polynomial-time hierarchy*. *Theoretical Computer Science* 3(1), pp. 23–33, doi:https://doi.org/10.1016/0304-3975(76)90062-1. Available at `https://www.sciencedirect.com/science/article/pii/0304397576900621`.

[181] Hayata Yamasaki, Kosuke Fukui, Yuki Takeuchi, Seiichiro Tani & Masato Koashi (2020): *Polylog-Overhead Highly Fault-Tolerant Measurement-Based Quantum Computation: All-Gaussian Implementation with Gottesman-Kitaev-Preskill Code*, doi:10.48550/arXiv.2006.05416. arXiv:2006.05416.

[182] Felix Zilk, Korbinian Staudacher, Tobias Guggemos, Karl Fürlinger, Dieter Kranzlmüller & Philip Walther (2022): *A Compiler for Universal Photonic Quantum Computers*, doi:10.48550/arXiv.2210.09251. arXiv:2210.09251.

[183] Richard Zippel (1979): *Probabilistic Algorithms for Sparse Polynomials*. In Edward W. Ng, editor: *Symbolic and Algebraic Computation*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 216–226, doi:10.1007/3-540-09519-5_73.