University of Amsterdam

MSc Mathematics

Master Thesis

Extraction of ZX-diagrams without gflow

Author:
Max Erné

Supervisor:
dr. J. Zuiddam

Examination date:
September 27th, 2024

Daily supervisor:
dr. J. van de Wetering (CWI)

Korteweg-de Vries Institute for
Mathematics

Abstract

The ZX-calculus is a convenient representation of quantum circuits that can be used for optimization. In this calculus, circuits are represented as graphs, called *ZX-diagrams*. An example of an optimization target that the ZX-calculus can be used for is reducing the number of T-gates, which may be expensive when error-corrected. However, it is possible for optimizations to break a criterion called *gflow*. This criterion is needed to efficiently extract a circuit from a ZX-diagram again, without ancillary qubits.

In this thesis, a weakening of gflow called *glack* is introduced, as a measure of how far a graph is removed from having gflow. Combining this notion with a straightforward extension of circuit extraction to isometries instead of unitaries, we will be able to extract circuits without gflow at the cost of ancillary qubits. Our extraction method uses a number of extra post-selected ancillae equal to the glack value. We will also analyze certain graph manipulations, which will result in a configurable trade-off between the T-count and post-selected ancillae.

Title: Extraction of ZX-diagrams without gflow
Author: Max Erné, max.erne@student.uva.nl, 11881666
Supervisor: dr. J. Zuiddam
Daily supervisor: dr. J. van de Wetering (CWI)
Second Examiner: dr. M. Ozols
Examination date: September 27th, 2024

Korteweg-de Vries Institute for Mathematics
University of Amsterdam
Science Park 105-107, 1098 XG Amsterdam
`http://kdvi.uva.nl`

Contents

1. Introduction

1.1. Quantum computation and the NISQ-era

Quantum computing is a relatively recent field, and promises significant improvements over classical computing in some scenarios. The most well-known example of this is *Shor's factoring algorithm* [28]. This is an algorithm that can factor the product of two primes exponentially faster than its classical counterpart. Another well-known example is *Grover's search algorithm* [18]. This search algorithm promises a quadratic speed-up over its classical counterpart. Quantum computers can also help simulate quantum behavior as encountered in, for example, quantum chemistry [29], without having to simulate the quantum part with a classical algorithm.

However, actually running these quantum algorithms has been a major hurdle for decades. Nowadays, quantum computers are a reality, but they only have access to a very limited amount of qubits. These computers are representative of the *noisy intermediate-scale quantum era*, or the NISQ-era [27]. As the name suggests, these quantum computers have a significant amount of noise, and do not have access to large amounts of qubits. The above-mentioned Shor's algorithm and Grover's search algorithm are sensitive to this noise and a poor fit for NISQ-era quantum computers [5]. However, other algorithms, such as *variational quantum eigensolver* algorithms, take into account this noise and the smaller scale. These algorithms may then be used on these NISQ-era machines to help with quantum chemistry or condensed matter physics [29].

NISQ-era quantum computers have some peculiarities. Due to the noise, computations should not take too long, and have limited *depth*. Otherwise, the qubits may decohere, and your computation becomes useless. Two-qubit gates are also bad for maintaining coherence. However, the limited depth makes *correcting* this noise infeasible. Instead, there are error *mitigation* techniques, such as *zero noise extrapolation* [16]. This technique considers the output of an algorithm at various noise levels, and uses this data to extrapolate back to what would have happened if there was no noise at all. This surprisingly effective technique is just one of the many tools introduced by the field of *quantum error mitigation* [6].

On the other hand, when we restrict ourselves to NISQ, the limitations open up new opportunities. Because NISQ-era quantum computers do not do error correction, gates that typically become expensive when error-corrected, such as the T-gate, can be used more freely. Similarly, because of the limited depth, introducing *post-selections* into algorithms does not pose too much overhead. Post-selections assert a specific measurement outcome, and require re-running the algorithm if the assertion failed and the wrong outcome was measured. When depth is limited, re-running the algorithm is not very expensive. For example, if an algorithm only succeeds with a probability of 1/1024, but can run in under a microsecond, we may not care about the large amount of "wasted" computation due to wrong outcomes.

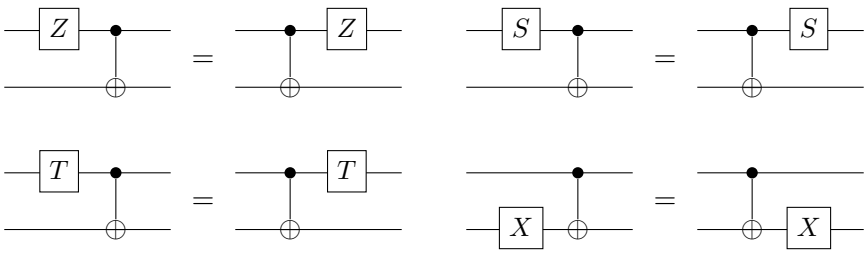
1.2. Optimization

Naturally, we want our quantum algorithms to be optimized in various metrics. However, what metrics we target depends on the quantum computer in question. As mentioned above, NISQ-era quantum computers prefer few two-qubit gates and low computation depth. On the other hand, the amount of T-gates does not matter very much, and having a larger amount of post-selections is also not too prohibitive. Quantum computers beyond the NISQ-era that have access to many qubits and do error correction to prevent the effects of noise are the opposite. These computers may not care as much about the two-qubit gate-count or computation depth, but the error-corrected T-gate may be expensive, and having to re-do a long computation due to a wrong post-selection may be prohibitively expensive.

In general, optimization parameters of interest are the number of qubits, the number of post-selections, the depth of computation, the number of two-qubit gates, and the T-count. Often, when optimizing for one of these, the performance of the others worsen. There are many results that can be summarized as "in order to improve this one parameter, this other parameter worsens" [1, 24]. This may not be a bad thing, if the "other parameter" is one our quantum computer inherently deals with easily. Having a whole catalog of such optimization techniques may also help migrate algorithms from targeting NISQ-era quantum computers to error-corrected computation in the future. In this thesis, we will be primarily interested in trade-offs involving the number of post-selections.

1.3. The role of the ZX-calculus

The most common model for representing quantum computation, *the circuit model*, is a poor fit for certain kinds of optimization. For instance, there are many similar rules for optimizing circuits that must be written separately in a rewriting engine, even if they are a part of the same phenomenon. Consider the following four equations:



In the model of the *ZX-calculus*, these four equalities (and more) are all the consequence of a single rule. This ZX-calculus is an even more recent field of study than quantum computation and made its debut in 2007 [9]. It is very useful for optimizing quantum circuits in a more convenient representation. Apart from improving the simplicity of rewrites, the ZX-calculus also offers a different perspective, and gives rise to some rewriting operations that do not have an intuitive counterpart in the circuit model, but are still useful.

The ZX-calculus also has other advantages. Proving correctness in the circuit model for quantum circuits frequently comes down to an opaque block of calculations, where intermediate results are computed after every single gate application. The rules of the

ZX-calculus allow you to simplify these calculations immensely.

The ZX-calculus has one major flaw, however. The calculus does not operate on circuits, but on *ZX-diagrams*. While conversion from circuit to diagram is easy, the converse is, in general, difficult. These ZX-diagrams can represent any linear map, which is much broader than the circuit model. As such, even deciding whether a diagram can be "extracted" as a unitary circuit is non-obvious. To do this extraction, a diagrammatic property called *gflow* is needed [4].

1.4. Contributions and organization of this thesis

In this thesis, we will relax this notion of gflow to *glack* extend the extraction procedure to arbitrary diagrams. Without gflow, this extraction comes at the cost of post-selected ancillae. When allowing for glack, more rewrite operations can become viable, giving a trade-off between the optimizations those new rewrites give, and the post-selected ancilla count.

This thesis is organized as follows. We first introduce the ZX-calculus as a replacement for the circuit model in Chapter 2. We will build it from the ground up, and include a few specific features that will interest us in later chapters. These include *phase gadgets* and *spider nests*.

It is from this basis that we will introduce the central concept of gflow in Chapter 3. First, we will introduce the motivating model of *measurement-based quantum computing*, and slowly build up all the requirements the gflow property encapsulates. Once we have done that, we will discuss its basic properties. Then, we will discuss the common extraction algorithm used for unitary graphs with gflow.

Finally, Chapter 4 will contain our new contributions. In this chapter, we will generalize the concept of gflow to glack in order to extract graphs the previous chapter could not handle. This generalization requires us to, in essence, re-do everything discussed in Chapter 3 in this broader context. We end the chapter with an example of how this analysis unlocks more graphical rewrites that may reduce the T-count.

We assume the reader is familiar with the basics of quantum computing already. For the unfamiliar reader, the first two chapters of [12] and some knowledge of the Bloch sphere [17] will suffice. Alternatively, chapter 4 of [25] covers more than enough. As we will be introducing entirely different models of computation, not much prerequisite knowledge is necessary. It is, however, helpful for the motivation. We do not require any prerequisite knowledge on the ZX-calculus.

1.5. Conventions used

Finally, we will give a small list of miscellaneous definitions and conventions used throughout this thesis.

- We denote the symmetric difference between two sets A and B as $A \,\Delta\, B := (A \cup B) \setminus (A \cap B)$.
- Let G be a graph. We write $V(G)$ for its vertices, and $E(G)$ for its edges. We will oftentimes abbreviate edges $\{u, v\}$ as simply uv. We will only be working with undirected graphs in this thesis.

- Let G be a graph and $u \in V(G)$ a vertex. We define the *neighborhood of* u as $N(u) := \{v \in V(G) \mid uv \in E\}$. (In particular, u itself is not included.) The neighborhood of a set of vertices U is $N(U) = \cup_{u \in U} N(u)$.
- We write $[k]$ for the range $\{1, \ldots, k\}$.
- The complement of a set A is denoted by $\overline{A}$.
- In the ZX-calculus, we will write "=" even if the equality only holds up to a non-zero scalar.
- When working with partial orderings $\prec$ on a set S, we will not just write $a \prec b$ for $a, b \in S$, but also use a shorthand when working with sets: with "$a \prec B$" where $B \subseteq S$, we mean "$a \prec b$ for all $b \in B$". The shorthands $A \prec b$ and $A \prec B$ are defined similarly.

2. The ZX-calculus

While the circuit model has its advantages, recently another model has gained popularity when it comes to optimizing quantum algorithms, namely the ZX-calculus. This calculus brings with it a new set of rewriting rules. In this chapter we will give a quick overview.

In Section 2.1, we will introduce the "syntax" of the calculus. We will describe *spiders*, and what the connections between them mean. We will also look at what common circuit model constructions look like in the ZX-calculus.

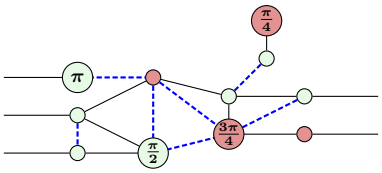
Next, in Section 2.2, we will look at what we can do with these diagrams, without changing the semantic meaning; the linear map represented is maintained. With the rules we introduce, we will see that these diagrams can be greatly simplified, up to the point we can consider these diagrams as simple graphs. After this, we will also introduce some rules that can remove specific spiders from these graphs.

In Section 2.3, we will describe a structure that is common in ZX-diagrams, namely the *phase gadget*. These have some additional rewrite rules, and also enable convenient formulation of the *spider nest identities*.

This is only a surface-level introduction. For a more in-depth treatment, see [30]. This is also the main reference for this chapter.

2.1. A tensor network

The ZX-calculus works with *ZX-diagrams*, which are a form of *tensor networks*: a graph with extra structure added on top. A ZX-diagram might look like this:



(2.1)

This diagram looks quite different from the circuit model we are used to:

- Instead of gates, we have vertices with one of two colors. These vertices may also have a phase.
- Instead of the ordered lines in the circuit model, these vertices are connected haphazardly. It is not clear what happens to each qubit wire anymore.
- Some of the edges are regular lines, but there are also dotted blue lines.
- Just like in the circuit model, we can consider fragments with a number of inputs on the left and a number of outputs on the right. Unlike the circuit model, it is not clear what was measured to reduce the number of qubits in this example.

This is quite a lot. We will slowly go through these points, starting with the vertices of this graph.

Definition 2.1.1 (Spider). A *Z-spider* with phase α is a linear map $\mathbb{C}^n \to \mathbb{C}^m$ of the form

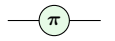
$$|0\cdots0\rangle\langle0\cdots0| + e^{i\alpha}|1\cdots1\rangle\langle1\cdots1|.$$

Similarly, an *X-spider* is a linear map

$$|+\cdots+\rangle\langle+\cdots+| + e^{i\alpha}|-\cdots-\rangle\langle-\cdots-|.$$

Diagrammatically, we write the Z-spiders and X-spiders respectively as follows:

Here, the n left wires correspond to the inputs of the spider, and the m right wires correspond to the outputs. If the phase α is zero, we do not write a number in the spider. If the phase is of the form $k\frac{\pi}{2}$ (for some integer k), we call the phase a *Clifford phase*, and the spider a *Clifford spider*.

A *ZX-diagram* is a collection of spiders connected to each other drawn in the plane. These spiders may also be connected to inputs or outputs on the left and right side of the diagram respectively, as in the example diagram (2.1) above. The names make sense: the spider  is simply the Pauli Z-gate, and is the Pauli X-gate. For larger spiders the names still make sense: the Z-spider is built from the eigenvectors $\{|0\rangle, |1\rangle\}$ of the Z-gate, and the X-spider is built from the eigenvectors $\{|+\rangle, |-\rangle\}$ of the X-gate.

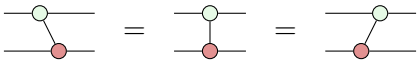
It is instructive to note that we can represent common basis states using spiders, up to a scalar. For instance, consider the spider  . By definition, this represents $|+\rangle - |-\rangle = \sqrt{2}|1\rangle$. The other base states are similar, giving the following:

$$|0\rangle = \frac{1}{\sqrt{2}}\ \text{} \qquad |+\rangle = \frac{1}{\sqrt{2}}\ \text{}$$
$$|1\rangle = \frac{1}{\sqrt{2}}\ \text{  } \qquad |-\rangle = \frac{1}{\sqrt{2}}\ \text{  } \tag{2.2}$$

In general, we will ignore non-zero scalars in the ZX-calculus and write "=", even if we are off by some non-zero factor. This is justified as we will always be working with unitaries or isometries; we can simply rescale the diagram once we're done. In particular, any component of the diagram not connected to the diagram's inputs and outputs represents a scalar factor, so we will ignore such components entirely if they are non-zero. (Zero factors are rare, and most commonly encountered as standalone  or spiders. We will not encounter any zero factor throughout this thesis.)

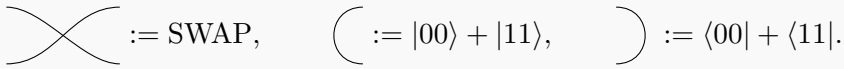
However, these spiders don't quite look like our example diagram (2.1): what do vertical connections mean? They're neither an input nor an output of a spider, so they seem ill-defined. This is less of a problem than it would seem, and it is actually just convenient notation. Consider for instance the following two ZX-diagrams:

One can verify that both of these actually represent the same map (a CNOT gate). Therefore, as "abuse of notation" we can also write a vertical wire, as it doesn't matter whether the Z-spider is to the left or the right of the X-spider:

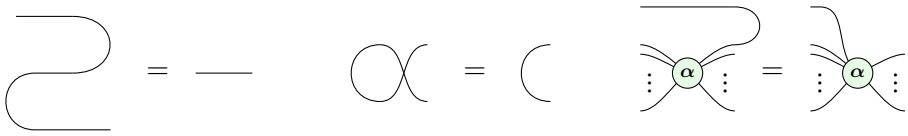


This is actually a general phenomenon: it does not in fact matter on which side edges connect to a spider, only that they are connected at all. Edges can also be bent in any way, or cross over. This does require us to assign consistent meaning to those structures, however.

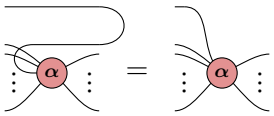
Definition 2.1.2. We define the *swap*, *cup*, and *cap* diagrams respectively as follows:



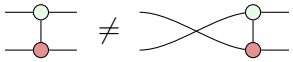
Using these definitions, it can be shown that it truly does not matter how you bend the wires between spiders, or whether you consider a wire an input or a bent output. In particular, we have the following:



Additionally, the definition of spiders is symmetric. This means we can conclude similar statements to the above hold for the other input and output wires of the spider as well. These statements are not limited to Z-spiders as in the image above, but also for X-spiders. For instance,the following equality holds:

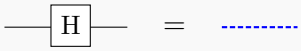


Our only use of these three formal definitions for these bends is to show the above symmetries. When considering a diagram, we will not actually use them explicitly. They only motivate that we can freely bend wires at will, and that only connections between spiders, inputs, and outputs matter. Note that the order of the inputs and outputs of the full diagram may not change. For instance, the following two diagrams represent different maps:

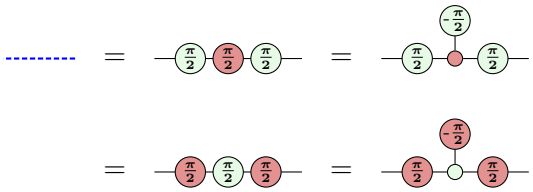


With these definitions, we can already represent any linear map as a ZX-diagram, up to a non-zero scalar. However, we will introduce one more definition, mainly for ease of notation.

Definition 2.1.3. We write the Hadamard gate diagrammatically as a *Hadamard-edge* as follows:



Because Hadamard gates are self-inverse, an edge can effectively have either zero or one Hadamard gate. As such, "regular edges" and "Hadamard-edges" cover all cases. As noted above, defining Hadamards is not truly necessary, as they can be written in a variety of ways using just Clifford spiders:



(2.3)

These expansions also hold with all signs flipped. They are unwieldy for something as fundamental as the Hadamard gate, so we will only be using these forms if we have to. In general, one can turn an arbitrary single-qubit unitary into Z-spiders and X-spiders like the Hadamard example above by using Euler angles [8]. With such a decomposition, one can see that Hadamard-edges can also be moved and bent around the diagram however we want, just like regular edges.

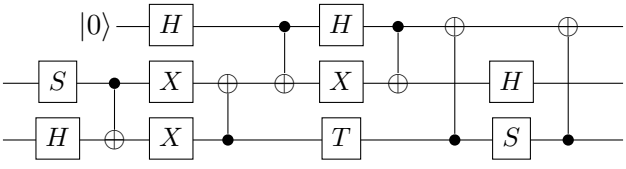
2.1.1. Translating circuits to ZX-diagrams

We have seen some translations between the circuit model and the ZX-calculus already. For instance, we know that Z gates are represented by a Z-spider  . We have also seen the SWAP gate and the CNOT gate implemented in ZX. We can in fact quite easily turn *any* circuit into a ZX-diagram.

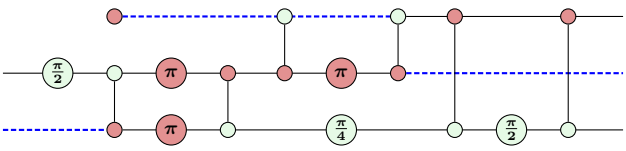
Using the T-gate (recall that $T^4 = Z$), the set $\{\text{CNOT}, H, T\}$ forms a *universal gate set* [12]: any unitary can be approximated arbitrarily well by a well-crafted circuit containing just those three types of gates. The set $\{\text{CNOT}, Z(\alpha), X(\beta) \mid \alpha, \beta \in [0, 2\pi)\}$ is also universal. These gates all have an easy representation in the ZX-calculus:

- Hadamards have a direct equivalent in the form of Hadamard-edges.
- Bloch sphere rotations about the Z- and X-axes have a direct equivalent. In particular, the Z, X, S, and T-gates can be directly converted from the circuit model to the ZX-calculus. As mentioned above, any single-qubit unitary can also be written as a composition of three rotations using Euler angles.
- Finally, the CNOT gate can be implemented by "replacing" the control with a Z-spider and the target with an X-spider.

Furthermore, prepared states are handled by Equation (2.2). With this, most of the circuit model can be translated already. Consider for instance the following example circuit:



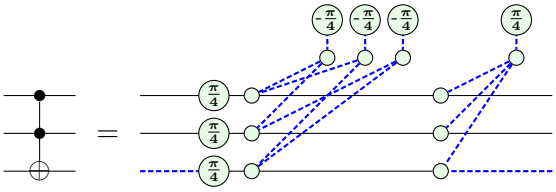
By doing the above replacements in-place, we get the following ZX version:



Measurements and measurement-dependent gates can also be translated into the ZX-calculus if variable-phase spiders are allowed. This is a bit more involved, however, and

will be discussed in Section 3.1.

Other gates, such as the CZ-gate or the Toffoli gate, are compositions of these gates, or can be approximated with these gates. While the CZ-gate is cheap (using only two Hadamards and one CNOT), the Toffoli gate is expensive. The following ZX-representation for the Toffoli gate is known:



Without ancillary qubits, this representation is also "the best" known representation in terms of *T-phases* [2]. These T-phases are all phases of the form $\frac{\pi}{4} + k\frac{\pi}{2}$, i.e. they differ from Clifford phases by $\pi/4$. There is no diagram known with fewer T-phases without ancillae.

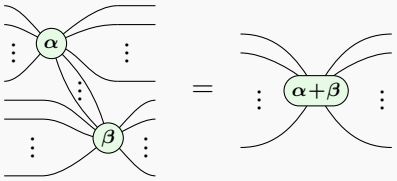
2.2. Operations in the calculus

Now that we have defined our playing field, it is time to look at what we can do with it, while not changing the semantic meaning of the diagrams. We want to optimize a ZX-diagram towards some goal, while ensuring it represents the same circuit we started with. In this section, we will start with the basic operations you will see described in any introduction to the ZX-calculus. We will immediately follow this up with the *local complementation* and *pivot* operations, as those are also essential for the later chapters.

In order to keep the statements concise, we will only list the Z-spider variant of the following statements. The X-spider variant is also true for all of these. We will include most proofs in this section so that a reader new to the ZX-calculus can familiarize themselves with the basic definition of spiders, and how rewrite rules are applied.

The most fundamental rule is that of *spider fusion*, which immediately gives us many opportunities to simplify our diagrams, and already makes use of the fact that these spiders are colored.

Proposition 2.2.1 (Spider fusion). You can fuse together two connected spiders of the same color, combining their inputs, outputs, and phases:



Proof. In order to get a non-zero result, all input wires of the α phase spider must be the same, either $|0\rangle$ or $|1\rangle$. The output wires then also share this property, except with an added phase of $e^{i\alpha}$ in the $|1\rangle$-case. (Note that this is true even if there are no inputs.) A similar statement holds for the β phase spider. Crucially, as the two spiders are connected, some outputs of the α spider must be the same as the inputs of the β spider in order to get a non-zero result.

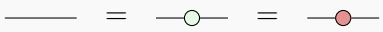
As such, the full map on the left-hand side must have *all* input wires set to either $|0\rangle$ or $|1\rangle$ to get a non-zero result, and if they are all $|1\rangle$, a phase of $e^{i\alpha}e^{i\beta}$ gets added. This is the same as the map on the right-hand side.

While it will frequently be useful to use this rule from the left-hand side to right-hand side to simplify our diagrams by fusing, the power of going from right to left to *unfuse* is not to be understated. There are many ways to write the phase of the spider on the right as $\alpha + \beta$, giving a lot of freedom for the resulting phases on the left-hand side. We can further freely choose what wires end up at one spider, and what wires end up at another.

By looking at this rule from both directions, we get a common proof technique in the ZX-calculus: first consider the statement where all phases are 0, and then consider what happens with arbitrary phases. We do this by unfusing the phases cleverly, applying the original statement, and then fusing the phases back. We will see the first instance of this technique in the proof of Proposition 2.2.3 already.

However, first we will need another convenient rule, the rule of "identity removal".

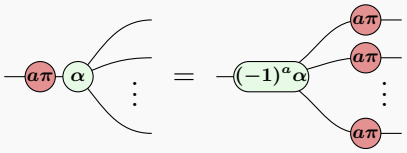
Proposition 2.2.2 (Identity removal). Phaseless spiders with exactly two neighbors can be dissolved:



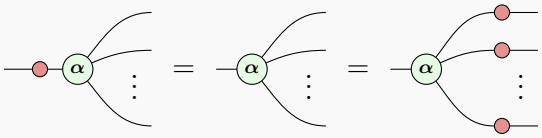
Proof. Showing that phaseless spiders are the identity quickly follows from the definition, as $|0\rangle\langle 0| + |1\rangle\langle 1| = |+\rangle\langle +| + |-\rangle\langle -| = \mathbb{I}$ (up to non-zero scalar).

As can be seen in their definition, spiders "copy" their input state to their outputs. It is then no surprise that this behavior can be used to our advantage in some scenarios.

Proposition 2.2.3 (π-copy). For $a \in \{0, 1\}$, we can propagate phase-$a\pi$ spiders through the opposite color as follows:



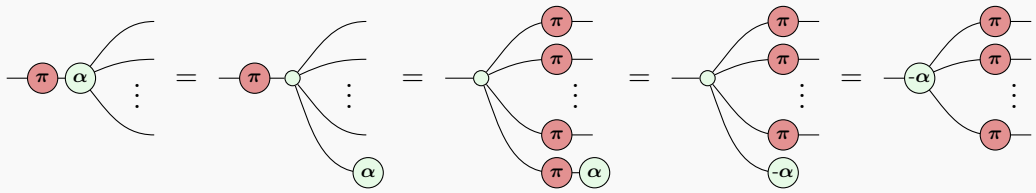
Proof. When $a = 0$, the proof consists of nothing more than using identity removal (both forwards and in reverse) to remove the X-spider on the left and add the X-spiders on the right:

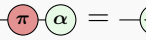


So instead consider the case where $a = 1$. Furthermore, consider $\alpha = 0$ so that we do not need to concern ourselves with the phase for the time being. In this case, equality is clear from the definition: the  spider swaps the $|0\rangle$ and $|1\rangle$ states, and for a map $|0\cdots 0\rangle\langle 0| + |1\cdots 1\rangle\langle 1|$ it does not matter whether it is applied before, or afterwards.

For arbitrary α, we unfuse the phase with spider fusion in reverse, and then use the

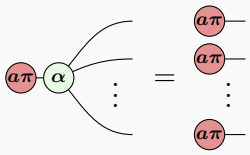
above phaseless case and re-fuse:



This relies on  which comes down to a simple computation using the definitions.

This π-copy rule also has a formulation for when the incoming spider is connected to nothing else.

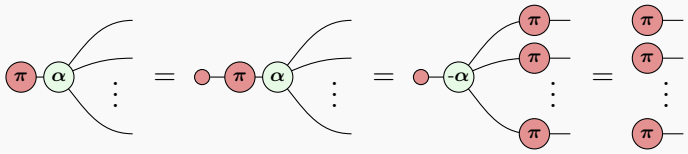
Proposition 2.2.4 (State copy). For $a \in \{0, 1\}$, we can propagate 1-legged phase-$a\pi$ spiders through the opposite color as follows:



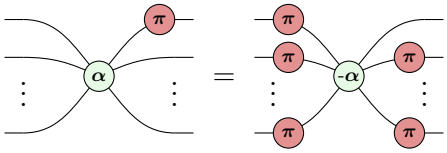
Proof. The $a = 0$ case is a computation with the spider definition and Equation (2.2):

$$\left(|0\cdots0\rangle\langle 0| + e^{i\alpha}|1\cdots1\rangle\langle 1|\right)|0\rangle = |0\cdots0\rangle.$$

When $a = 1$, we use spider fusion, the π-copy rule, and the $a = 0$ case:

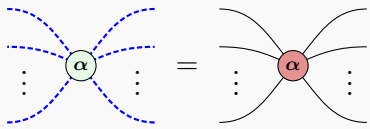


As the calculus does not care about whether something is an input or a bent output of a spider, it is worth emphasizing that these identities also hold when there are multiple inputs, or none at all:

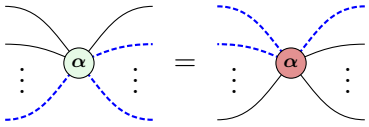


Z-spiders are based on the $|0\rangle$ and $|1\rangle$ states, and the X-spiders on the $|+\rangle$ and $|-\rangle$ states. It then is no surprise that Hadamard gates that swap between these two bases can interact meaningfully with the two kinds of spiders. In particular, the following proposition follows directly from the definitions.

Proposition 2.2.5 (Color change). We can surround a spider without self-loops by Hadamards in order to flip its color:



Note that if some edges are Hadamard-edges already, the two Hadamard gates cancel out and revert to regular edges:



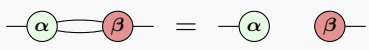
Color changing gives us access to a powerful "meta-rule".

Corollary 2.2.6. Any ZX-diagram identity holds with all spider colors flipped.

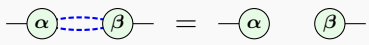
Proof. First, apply the color change rule to *all* spiders on both sides of the equation. On edges between two spiders, these introduced Hadamards will cancel out, leaving only an extra Hadamard on input and output wires.

Now consider the subdiagram without these extra input and output Hadamards on both sides. This is justified, as it is a change of basis. Doing this results in the same equation as before, but with all spider colors flipped.

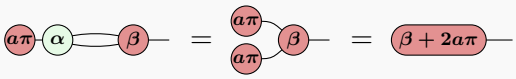
Proposition 2.2.7 (Hopf rule). When two spiders of opposite color are connected with two wires, those wires may be removed:



Equivalently, when two spiders of the same color are connected with two Hadamard wires, those wires may be removed:



Proof. It suffices to show that the two basis states $|a\rangle$ (with $a \in \{0, 1\}$) are mapped to the same thing by both sides of the equation (up to a non-zero scalar). We do this by putting an $a\pi$ spider before the input wire. We start with the left-hand side, where we will state copy our input through the Z-spider and fuse into the X-spider:



As we are working with phases, this $2a\pi$ factor contributes nothing, and we end up with β— . The right-hand side requires even less ZX-manipulations. Prepending the basis states gives us $a\pi$–α β—, which equals β— up to scalar value $a\pi$–α (as that component is not connected to any inputs or outputs). From the definition, $a\pi$–α $= \sqrt{2}e^{i\alpha a}\langle a|a\rangle \neq 0$, so we are done.

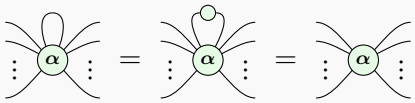
The Hadamard-edge variant is proven by applying a color change to the red spider on

both sides of the equation, and considering the resulting extra Hadamard gate on the output a basis change we can disregard.

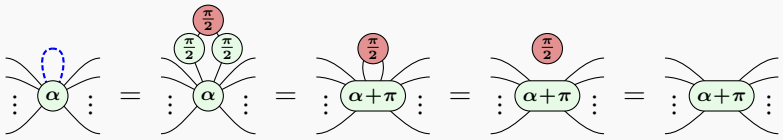
By applying some clever spider fusions, we see we can also apply the Hopf rule if the spiders have a number of outgoing edges other than 1.

Proposition 2.2.8 (Self-loop removal). Self loops with both regular and Hadamard-edges can be removed:

Proof. The left equation is proven by adding a spider with identity removal (in the opposite direction) and then applying spider fusion:



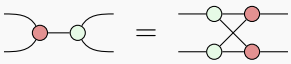
The right equation is proven by using an appropriate Euler-decomposition (Equation (2.3)) of the Hadamard gate, fusing two $\pi/2$ phases into the α phase, and finally disconnecting the resulting scalar factor with the Hopf rule:



Note that the $\frac{\pi}{2}$ spider contributes a scalar factor of $1 + i \neq 0$, that we ignore.

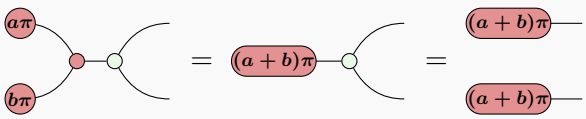
Usually, Z-spiders act as a barrier to X-spiders and vice versa, and they cannot be moved past one another. However, in specific cases, the spiders do "commute" in some sense.

Proposition 2.2.9 (The bialgebra rule). The following identity holds:

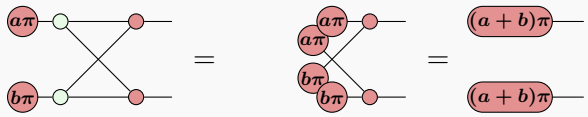


A similar statement holds for an arbitrary amount of inputs and outputs (which may be different). In this case, the right-hand side becomes a complete bipartite graph on an appropriate number of spiders.

Proof. It suffices to show that the four basis states $|ab\rangle$ (with $a, b \in \{0, 1\}$) are mapped to the same thing by both sides of the equation. This corresponds to putting $a\pi$– and $b\pi$– spiders on the two input wires. We start with the left-hand side, where we apply spider fusion and a state copy:



The right-hand side is very similar:



We conclude the two maps are the same.

This proof generalizes to larger numbers of inputs and outputs.

It is known that these rules already form a *complete* calculus when all phases are Clifford phases [30]: if two diagrams represent the same linear map (up to a non-zero scalar), there is some application of the above rules that transforms the one into the other.

2.2.1. Constructing graph-like diagrams

With these rules, we can already do quite some simplifications. These simplifications are a major motivation for the ZX-calculus when compared to the circuit model. If we simply convert the gates of a circuit into spiders as discussed in Section 2.1.1, the resulting ZX-diagram will still look like a circuit. There is a more specific form we want to mold this circuit into to simplify matters.

Definition 2.2.10 (Graph-like diagram). A *graph-like diagram* is a ZX-diagram without any X-spiders, duplicate edges, or self-loops, and where every edge between spiders is a Hadamard-edge.

The spiders may be connected to inputs or outputs with both regular edges or Hadamard-edges, but each input or output can be connected to only one spider.

As the name suggests, graph-like diagrams mostly have the structure of a simple graph. The vertices of these graphs are the Z-spiders, and the edges are the Hadamard-edges between these spiders. The only "extra" information is that some vertices are connected to some input or output by either a regular edge or a Hadamard-edge. Compare this to "normal" ZX-diagrams as discussed so far, in which all spiders have a color, and all edges have a type, which is a lot more data to keep track of.

Turning arbitrary ZX-diagrams into graph-like diagrams is not difficult.

Proposition 2.2.11 (Creating graph-like diagrams). Consider the following algorithm (where within every step, order does not matter):

(1) Convert all X-spiders into Z-spiders with the color change rule.

(2) Apply spider fusion until there is nothing left to fuse.

(3) Remove self-loops and Hadamard-self-loops with Proposition 2.2.8.

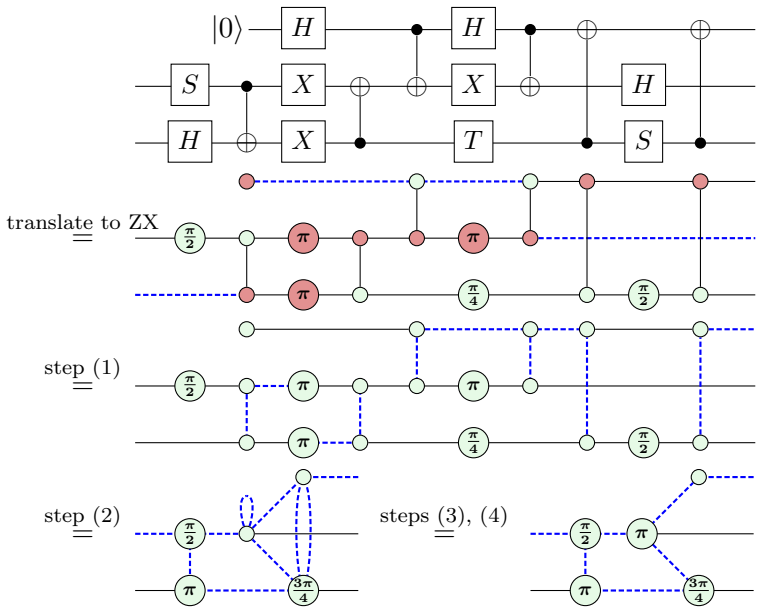
(4) Remove all pairs of Hadamard-edges between spiders with the Hopf rule.

(5) If a spider is connected to multiple inputs or outputs, unfuse this spider into multiple to ensure each spider is connected to only one. Each time we unfuse, replace the regular wire between the spiders in question with ----○---- .

This is a polynomial-time algorithm in the number of spiders that converts an arbitrary ZX-diagram into a graph-like diagram.

Proof. Each step maintains every modification done before: after (1), nothing introduces X-spiders anymore; after (2), nothing introduces spiders that can be fused anymore, etc. In particular, after (2), we only have Z-spiders that cannot be fused due to Hadamard-edges between them. Every step is also efficient.

An example of the full conversion process from circuit to graph-like diagram can be seen in the following (very artificial) example.



(We actually deviated from the algorithm a bit in step (2) to introduce a Hadamard self-edge by doing an *additional* optimization to showcase step (3).) This algorithm returns a fairly compact graph for us to do more interesting things with.

2.2.2. The local complementation and the pivot

Apart from the rules we discussed so far, there are still two more derivative operations we are interested in, as it will turn out that these will maintain the *gflow* condition that will be introduced in Chapter 3. These operations are the *local complementation* and the *pivot*. We will first introduce these statements purely graphically, and then in the ZX-calculus.

Throughout this section, diagrams are assumed to be in graph-like form.

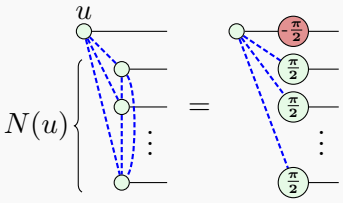
Definition 2.2.12 (Local complementation). Let G be a graph, and $u \in V(G)$ a vertex. Then the *local complementation about* u, denoted $G * u$, is the same graph as G, except that edges in $N(u)$ get *complemented*: connected vertices get disconnected, and disconnected vertices get connected. More explicitly,

$$\begin{aligned} V(G * u) &= V(G), \text{ and} \\ E(G * u) &= E(G) \,\Delta\, \{(v, w) \mid v, w \in N(u) \text{ with } v \neq w\}. \end{aligned}$$

(Recall that we do not have $u \in N(u)$.)

In the ZX-calculus, not just the edges get updated, but the phases change as well.

Proposition 2.2.13 (Local complementation in ZX, regular variant). The following equation holds:

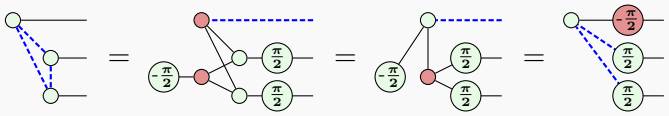


While the above figure only draws the case that $N(u)$ is a complete subgraph, all edges in $N(u)$ get complemented.

Of course, u's neighborhood may also contain non-zero phases (α_i) on the left-hand side, which then get transformed into ($\alpha_i + \pi/2$) on the right-hand side. The spiders in the neighborhood may also be connected with more than one spider outside the picture, or none at all. We omit these details from the statement as they come down to unfusing spiders, applying the statement, and refusing the unfused spiders with the result.

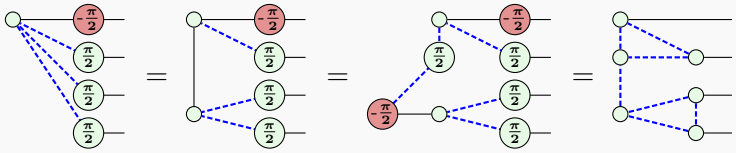
Proof. This is proven in [14], but it is a somewhat involved induction proof. For this reason, we will be limiting ourselves to the base case of two neighbors and the induction step into three neighbors, which will show all important steps. (The special cases with one or zero neighbors are trivial.)

The two-neighbor case goes as follows:



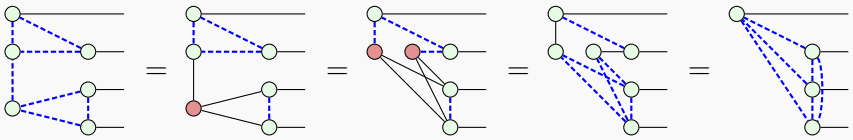
In the first step, we changed u's color, and wrote the Hadamard-edge between u's two neighbors as in the top right of Equation (2.3), in preparation for using the bialgebra rule. After applying said rule, we simplify the diagram back to its desired form with some more color changes and fusions.

Now consider the case with three neighbors. This time, we start with the statement's right-hand side. We first introduce a few dummy spiders to create the statement's right-hand side with lower spider count twice: once with two neighbors, and once with one neighbor less than we started with. This looks like the following:



In the first equality, u was unfused into two spiders. One of these spiders connects to the $-\frac{\pi}{2}$ spider and a single $\frac{\pi}{2}$ spider. The other spider connects with the remaining neighbors. Between these two spiders, we added a fancy identity: a Hadamard gate, a $\frac{\pi}{2}$ spider, another Hadamard gate, and a $-\frac{\pi}{2}$ spider. This allows us to apply the two-neighbor case twice in the second equality.

What remains is putting this graph into the form of the statement's left-hand side:



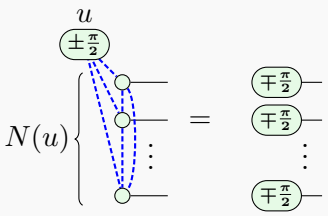
First, we changed a spider color in order to be able to apply the bialgebra rule again. After this, we changed the colors back and fused spiders to get the desired result.

Finally, if there already exist Hadamard-edges between spiders in $N(u)$, we will get double edges. These cancel with the Hopf rule.

We can also choose a different decomposition of the Hadamard gate in Equation (2.3). This gets you the same statement but with the signs of the $\frac{\pi}{2}$ and $-\frac{\pi}{2}$ spiders flipped.

The above statement suggestively calls it a "regular variant". We will also discuss a second variant. This is a variant that we can use to delete certain spiders.

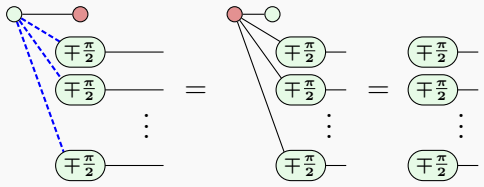
Proposition 2.2.14 (Local complementation in ZX, deleting variant).



The notes about Proposition 2.2.13 apply here as well.

Proof. The big trick is that we can use the identity  (which follows from the definition) to cancel out the X-spider introduced by Proposition 2.2.13:

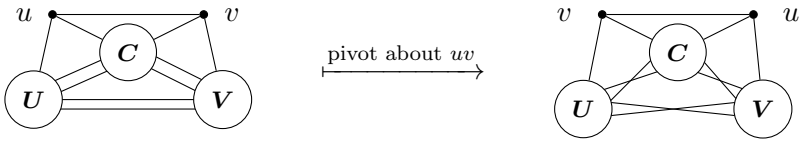
After the cancellation, we post-process it with two color changes and a state copy to obtain the right-hand side:



Just like the local complementation, the *pivot* we will now define will also have a graphical variant, a regular ZX-variant, and a deleting variant.

Definition 2.2.15 (Pivot). Let G be a graph, and $uv \in E(G)$ an edge. Then the *pivot about* uv is defined as $G \wedge uv := G * u * v * u$.

While local complementations can be understood by sketching a single example in a notebook, it is not immediately clear how pivoting influences the graph at all. To help with that, consider the following diagram.



(2.4)

If we denote the common neighbors of u and v as C, the remaining neighbors of u as U, and the remaining neighbors of v as V, a pivot does the following:

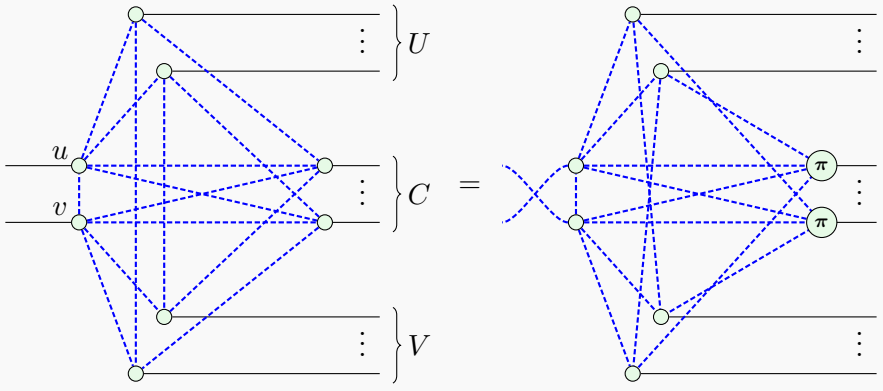
- Vertices u and v swap places;
- The edges between U and C get complemented. This complement is represented in the above diagram by changing parallel edges to crossing edges;
- The same happens also for edges between U and V, and V and C;
- The rest of the graph stays the same.

You can show this by manually applying the three local complementations. This operation is symmetric in u and v. That means that pivoting about uv is well-defined, as $G \wedge uv = G \wedge vu$.

Proposition 2.2.16 (Pivot in ZX, regular variant). Let uv be a Hadamard-edge a ZX-diagram. Denote the common neighbors of u and v as C, the remaining neighbors of u as U, and the remaining neighbors of v as V. Then the pivot about uv does the following:

- Vertices u and v's outgoing edges swap roles and become Hadamard-edges;
- The edges between U and C, U and V, and C and V get complemented;
- Spiders in C get an added π phase.

Diagrammatically:



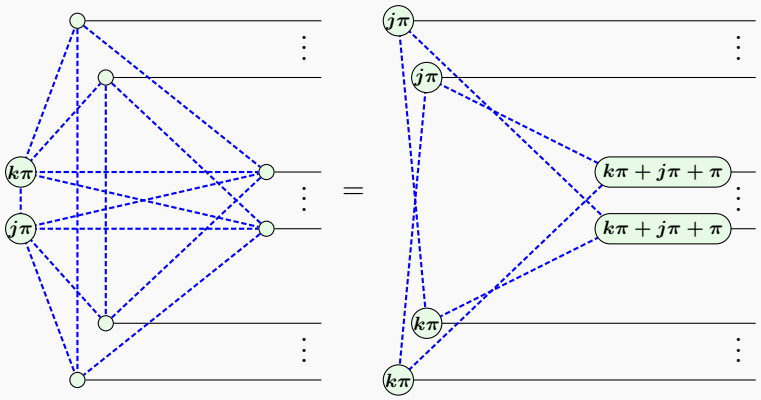
Proof. As a pivot consists of three local complementations, the proof consists of some tedious bookkeeping that we will not include here. We will only list the recipe.

Starting with the right-hand side, turn u's outgoing Hadamard into the Euler decomposition (Equation (2.3)) with two $\frac{\pi}{2}$ spiders, and v's outgoing Hadamard into the one with one $\frac{\pi}{2}$ spider. This allows us to apply three local complementations as in

Proposition 2.2.13, one at u, then one at v, and then one at u. Carefully doing this will then give the left-hand side.

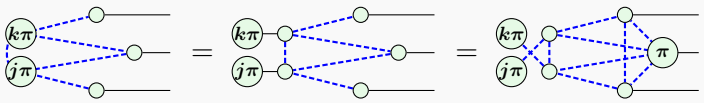
Similar remarks to the remarks made earlier for the local complementation apply here. The diagram is drawn with parallel and crossing lines as in (2.4), but any set of edges is allowed and will get complemented. The spiders, other than u and v, may also all have non-zero phases or have more than just one outgoing connection, and the right-hand side then updates accordingly. We will now discuss the deleting variant.

Proposition 2.2.17 (Pivot in ZX, deleting variant).

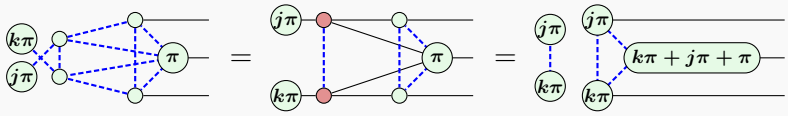


Proof. To reduce clutter, we will only be drawing one spider in each type of neighborhood. One can verify that this is representative of the general case.

The first thing we do is unfuse the phases and apply the regular pivot variant above:



After this, we clean up the graph by moving some spiders around, and doing two color changes. This allows us to apply the state copy rule twice, and we get the following:



For the remaining scalar component we have

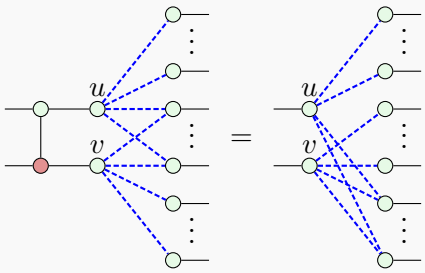
$$k\pi \text{ --- } j\pi = (\langle 0| \pm \langle 1|)\ (|+\rangle \pm |-\rangle) \neq 0,$$

with the signs depending on the values of k and j. In any case, we can disregard it.

By redoing the proof of this last variant, we can also get similar statements for what happens when one or both of the deleted spiders are connected to an input or output, or have a non-π-multiple phase. We will not be needing these variants.

Finally, there is one more operation that interests us.

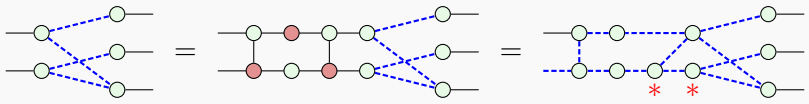
Proposition 2.2.18 (Adding CNOTs).



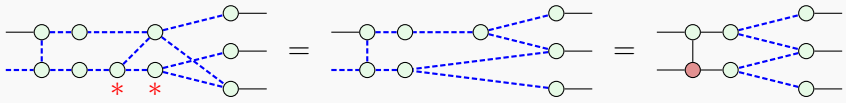
Applying a CNOT before spiders u (controlled) and v (targeted) has the same effect as updating $N(u)$ to $N(u) \,\Delta\, N(v)$, (not counting outgoing wires in $N(u)$ and $N(v)$).

Proof. The statement can be easily proven with the bialgebra rule. However, we will follow [13] instead, as this will give us a construction that we will use in Chapters 3 and 4.

We will work backwards from the right-hand side. Again, to reduce clutter, we will only be drawing one spider in each type of neighborhood. We first add two CNOTs and some identities, and do some color changes and a fusion.

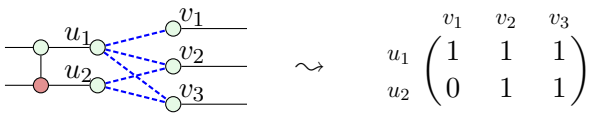


The next step is applying the deleting pivot, Proposition 2.2.17, in order to remove the $*$-marked spiders. Note that these do not have common neighbors, so that we do not introduce π phases anywhere. After this pivot, we simplify back.

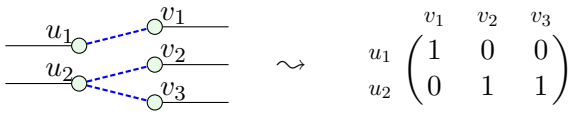


This results in the left-hand side we wanted.

This operation is especially interesting because we can link it to linear algebra. Consider the *biadjacency matrix* from the two spiders we put the CNOT before, to their neighbors, as in the following example.



In the biadjacency matrix, the entry at row u_i and column v_j is 1 if and only if u_i and v_j are connected by Hadamard-edge. When we apply the above proposition, the changes are clear in the biadjacency matrix.

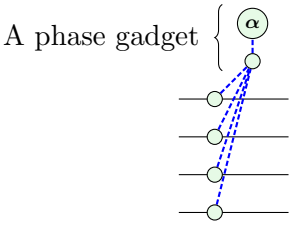


We simply added the row corresponding to u_2 (the CNOT's target) to u_1 (the CNOT's control) in $\mathbb{F}_2$. In this specific example, we end up with a slightly simpler graph.

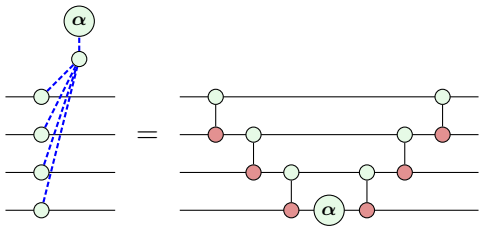
This can be extended to bigger biadjacency matrices, which gives us a link between Gaussian elimination in $\mathbb{F}_2$ and certain applications of CNOT gates. In the best case, matrices can go from being very dense to being reduced to the identity, which is a considerable reduction in edge count. This will become useful once we get to circuit extraction in Section 3.6.

2.3. Phase gadgets and phase polynomials

One structure in ZX-diagrams is common in particular: the *phase gadget*.



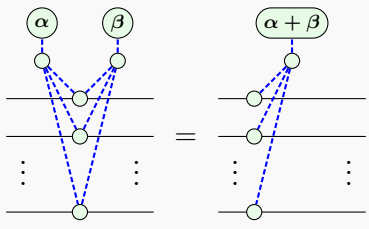
The term "phase gadget" refers to the top two spiders of this diagram: a one-legged spider with a phase, connected to a spider that is connected to one or multiple other spiders. Such phase gadgets add a phase of α only if an odd number of wires get an input $|1\rangle$. They are represented by diagonal matrices, and therefore commute with all other diagonal operators. In particular, phase gadgets commute with other phase gadgets. You can also see this property in their ZX-representation by applying spider fusion. In circuit form, they can be represented by a ladder of CNOT gates and a Z-rotation.



Note that a phase gadget connected to just one wire does the same as a regular Z-spider.

As they add an α phase when an odd number of wires are $|1\rangle$, multiple phase gadgets that are connected to the same spiders can be combined. This can also be shown in the ZX-calculus.

Proposition 2.3.1 (Gadget fusion). Phase gadgets connected to the same spiders can be fused, adding up their phases:



Proof. An application of the bialgebra rule after changing the colors of the neighbors (α)-- and (β)--.

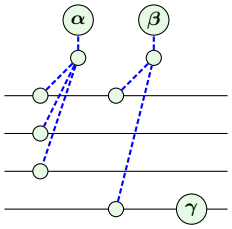
Phase gadgets are a special example of a slightly more general framework of *phase polynomials*. Suppose you apply a phase gadget of phase α on qubits $x_1, x_2, \dots x_n$. This represents a map

$$|x\rangle \mapsto e^{i\alpha(x_1 \oplus \cdots \oplus x_n)}|x\rangle.$$

Here, $\oplus$ is addition in $\mathbb{F}_2$. Instead of just the parity of x, $(x_1 \oplus \cdots \oplus x_n)$, this exponent leaves room for playing around with more complex expressions involving x.

Definition 2.3.2 (Phase polynomial). Let $\varphi : \mathbb{F}_2^n \to \mathbb{R}$. Then φ is called a *semi-boolean function*. If it appears in a unitary of the form $|x\rangle \mapsto e^{i\varphi(x)}|x\rangle$, we call it a *phase polynomial*.

For example, $\varphi(x) = \alpha(x_1 \oplus x_2 \oplus x_3) + \beta(x_1 \oplus x_4) + \gamma\, x_4$ is a phase polynomial (where $\alpha, \beta, \gamma \in \mathbb{R}$ represent arbitrary phases). In general, phase polynomials all look like this: a sum in which each term is some real number, multiplied by the parity of some collection of $\{x_i\}$. If you put each term in its own exp(), you can see that phase polynomials are nothing more than the composition of multiple phase gadgets. For instance, the example φ from above looks like this:



2.3.1. Spider nest identities

Apart from the representation of phase polynomials using the parity operator $\oplus$ discussed above, there is another: the AND operator $\wedge$.

Proposition 2.3.3 (Semi-boolean Fourier transform). Let $f : \mathbb{F}_2^n \to \mathbb{R}$ be such that

$$f(x) = \sum_{y \in \mathbb{F}_2^n} \lambda_y (x_1 y_1 \oplus \cdots \oplus x_n y_n) \tag{2.5}$$

with $\lambda_y \in \mathbb{R}$ for all y. Then there exist $\widehat{\lambda}_y \in \mathbb{R}$ such that

$$f(x) = \sum_{y \in \mathbb{F}_2^n} \widehat{\lambda}_y (x_1^{y_1} \wedge \cdots \wedge x_n^{y_n}). \tag{2.6}$$

(Here, $x_i^{y_i}$ equals 1 if y_i is 0, and x_i if y_i is 1.)

Conversely, given $\widehat{\lambda}_y$ such that (2.6) holds, there exist $\lambda_y \in \mathbb{R}$ such that (2.5) holds.

Proof. Both of $\{x \mapsto x_1 y_1 \oplus \cdots \oplus x_n y_n \mid y \in \mathbb{F}_2^n\}$ and $\{x \mapsto x_1^{y_1} \wedge \cdots \wedge x_n^{y_n} \mid y \in \mathbb{F}_2^n\}$ are a basis for the vector space of functions $\mathbb{F}_2^n \to \mathbb{R}$.

The knack of course lies in determining what this mapping from λ to $\widehat{\lambda}$ in this proposition is exactly. Consider the simple two-qubit case, which only has one interesting entry in the sum: $x_1 \oplus x_2$. One can verify that

$$x_1 \oplus x_2 = x_1 + x_2 - 2(x_1 \wedge x_2). \tag{2.7}$$

Then $\lambda_{(1,1)}$ contributes $+\lambda_{(1,1)}$ to both $\widehat{\lambda}_{(1,0)}$ and $\widehat{\lambda}_{(0,1)}$, and $-2\lambda_{(1,1)}$ to $\widehat{\lambda}_{(1,1)}$. Larger expressions such as $x_1 \oplus x_2 \oplus x_3$ can be handled by repeatedly applying Equation (2.7) and simplifying the result. A similar approach with

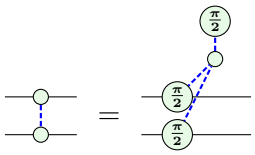
$$x_1 \wedge x_2 = \frac{1}{2}(x_1 + x_2 - x_1 \oplus x_2) \tag{2.8}$$

gives the inverse direction.

We can use this two-qubit case to show a neat identity. Applying a CZ-gate to $|x_1 x_2\rangle$ gives a result of $(-1)^{x_1 \wedge x_2}|x_1 x_2\rangle$. This can be rewritten as follows:

$$(-1)^{x_1 \wedge x_2}|x_1 x_2\rangle = e^{i\pi(x_1 \wedge x_2)}|x_1 x_2\rangle = e^{i\frac{\pi}{2}(x_1 + x_2 - x_1 \oplus x_2)}|x_1 x_2\rangle. \tag{2.9}$$

Here, we simply used Equation (2.8). We started with a CZ-gate applied to two qubits (which has a ZX-representation), and we end up with a phase polynomial expression (which also has a ZX-representation). Thus we can translate Equation (2.9) into the ZX-calculus:



This is actually just one of the Hadamard decompositions of Equation (2.3) that we have seen already. Unfortunately, this analysis will not generalize to higher-order AND expressions. In fact, while the parity formulation in Equation (2.5) has a clear ZX-representation, it is not at all clear how to represent the AND formulation in Equation (2.6)[1]. To work with the AND of multiple variables, we start with the following lemma.

Lemma 2.3.4. Let $x \in \mathbb{F}_2^n$. Then

$$x_1 \wedge \cdots \wedge x_n = -\frac{1}{2^{n-1}} \sum_{y \in \mathbb{F}_2^n \setminus \{0\}} (-1)^{|y|}(x_1 y_1 \oplus \cdots \oplus x_n y_n).$$

Here, $|y| = |\{i \in [n] \mid y_i \neq 0\}|$ is the Hamming weight of y.

Proof. This is a straightforward induction proof, with the base case already handled by Equation (2.8).

With this lemma, we can represent the map

$$|x\rangle \mapsto e^{i\alpha(x_1 \wedge \cdots \wedge x_n)}|x\rangle$$

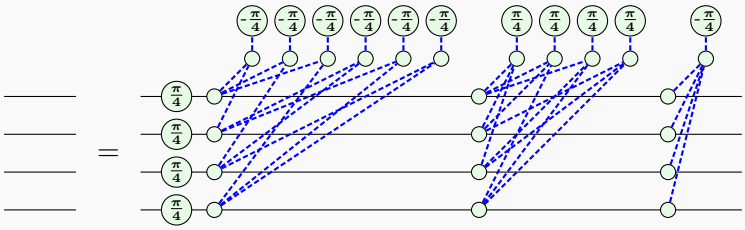
[1] The usual approach is to extend the ZX-calculus to the *ZH-calculus*. This calculus adds a generalization of Hadamard gates, which allows these AND semantics to be easily represented. However, this is not important to our story. For more information, see [3].

as a ZX-diagram for general α by using a lot of phase gadgets. But now consider the case where $\alpha = 2\pi$. This gives us the identity map. But when applying the above lemma, the result does not look like the identity at all:

$$e^{2\pi i(x_1 \wedge x_2 \wedge x_3 \wedge x_4)} = e^{-i\frac{\pi}{4}\sum_{y \in \mathbb{F}_2^4 \setminus \{0\}} (-1)^{|y|}(x_1 y_1 \oplus \cdots \oplus x_4 y_4)}. \tag{2.10}$$

Translating this phase polynomial into the ZX-calculus, we have proven the following.

Proposition 2.3.5 (Four-qubit spider nest).



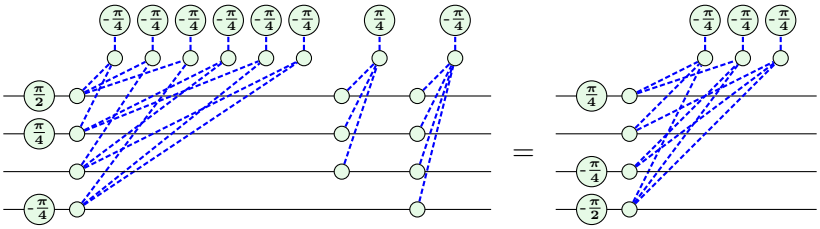
The same identity also holds with all phases' signs flipped. We grouped the phase gadgets, in order, as first all phase gadgets involving one spider, then all phase gadgets involving two spiders, then three, and then four.

While we have not discussed extraction of ZX-diagrams back to circuits yet, a reduction in these T-phases in diagrams often corresponds to a reduction in T-gates in the extracted circuits. The above proposition then gives a reduction of 15 T-gates! This situation is not very common, however. More reasonable is the following formulation that takes into account combining existing T-phases into $\pi/2$-multiples.

Corollary 2.3.6. Fix four spiders, and let $\{u_j\}$ be a set of phase gadgets only connected to those four spiders. Let ℓ be the number of phase gadgets u_j with a T-phase.

Then $\{u_j\}$ can be replaced with an alternate set of phase gadgets $\{\widehat{u}_j\}$, still only connected to those four spiders, of which $15 - \ell$ have a T-phase.

You can think of this as "toggling" which of the fifteen phase gadgets have a T-phase. For instance, we can use this to do the following rewrite to go from 10 T-phases to 5:

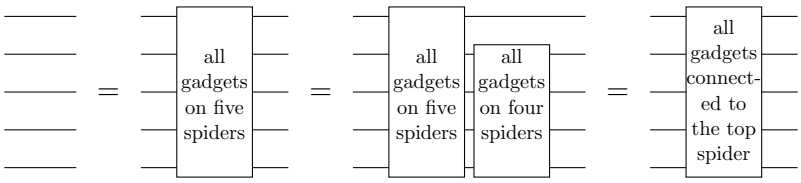


In this example, we only gave the single-qubit phase gadgets a phase different from Proposition 2.3.5 to illustrate how other phases work. The example's right-hand side is the result of "subtracting" the right-hand side of Proposition 2.3.5 from the example's left-hand side. This subtraction consists of applying gadget fusion to every gadget.

If you work with five qubits instead of four, you can do a similar trick as we did when proving Corollary 2.3.6 by looking at $4\pi(x_1 \wedge \cdots \wedge x_5)$ instead of $2\pi(x_1 \wedge \cdots \wedge x_4)$ in Equation (2.10). This gives another "spider nest identity", on 31 phase gadgets. However, this is unlikely to be worth the effort: you need to encounter at least sixteen

T-phase gadgets connected to the same five spiders in order to improve the T-count. This seems very unlikely in practice.

Spider nest identities such as these can also be "combined" to obtain new identities. For instance, consider the spider nest identities on five qubits, and the spider nest on four qubits with flipped phases (which are both equal to the identity). Then a lot of phase gadgets are canceled out by gadget fusion, resulting in a new identity.



While the five-qubit spider nest is unlikely to be useful by itself (a 31 phase gadget context is just too unlikely to happen), this derived spider nest is much more reasonable, toggling only $31 - 15 = 16$ phase gadgets. We will see the four-qubit spider nest that toggles 15 phase gadgets have use, so a spider nest with 16 phase gadgets may also have use still.

3. Gflow

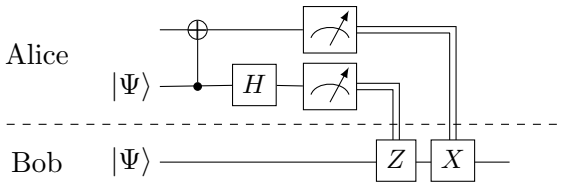
Gflow is a flow-like graph property that encapsulates "being extractable" for ZX-diagrams. It is also a highly unintuitive property. For this reason, we will spend a significant portion of this chapter building up the necessary intuition. After this intuition is built up, we will look at when gflow is maintained. Finally, we will discuss extraction.

In Section 3.1, we will discuss the motivation for gflow by means of a different model of quantum computation: *measurement-based quantum computation*. In this model, the gflow conditions can be summarized as "you can continue computation even when measurements give an unexpected outcome". This intuition translates somewhat directly to the ZX-calculus, but it still takes some time to properly build up. As such, in Section 3.2 we will go in steps, and discuss two easier forms of flow before we get to *triplanar* gflow. Both these sections will follow [4] fairly directly.

Once we have defined gflow, we want to show that the graphs we will be working with have this property. This is what we will prove in Section 3.3: any circuit translated into a ZX-diagram will have gflow. We also want to know when gflow is maintained, or not. This will be discussed in Sections 3.4 and 3.5. While the latter section consists mostly of examples, these examples will be important motivation once we consider *glack* in Chapter 4. Finally, with gflow fully introduced, Section 3.6 will discuss extracting unitary circuits with gflow.

3.1. Measurement-based quantum computation

The circuit model does computation by initializing a circuit (for example, to $|0^n\rangle$), applying gates, and doing measurements. Sometimes, these measurements affect what gates you apply in the future, as in the well-known *teleportation* protocol:



(3.1)

This protocol has Alice and Bob share a $|00\rangle + |11\rangle$ state to allow transmission of a single qubit from Alice to Bob. In this diagram, you can see how the measurement outcomes of Alice define whether Bob applies Z or X gates. This example consists mostly of corrections based on measurements. However, most algorithms in the circuit model do most of the computation with gates.

With *measurement-based quantum computation*, or *MBQC* for short, we do not rely on gates as in the circuit model. As the name suggests, computation is done by measuring qubits. Such computation requires highly-entangled initialization. (If you measure a

qubit that is not entangled, it does not affect anything else, so you would hardly call it "computation"!) We will now discuss a specific model of MBQC, the *one-way model*, in detail. We need to discuss two things: the initialization, and the measurements.

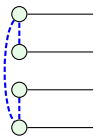
Definition 3.1.1 (Graph state). A *graph state* is a graph-like diagram (see Definition 2.2.10) with no inputs or phases, and where every spider is connected to one output of the diagram.

Said differently, if G is a graph, the corresponding graph state is given by

$$|G\rangle = \prod_{uv \in E} \mathrm{CZ}_{u,v} |+\rangle^{\otimes |V|}.$$

Here, $\mathrm{CZ}_{u,v}$ denotes the controlled-Z gate between qubit u and v.

A simple example graph with its corresponding graph state is the following:

$G =$  $\rightsquigarrow$ $|G\rangle =$ 

This gives us the initialization. We also need to discuss the measurements. First, let's define the measurements we will be using: we measure about arbitrary rotations on the Bloch sphere planes XY, XZ, and YZ. We write these measurements bases as follows:

$$\begin{aligned} |\pm_{\mathrm{XY},\alpha}\rangle &= |0\rangle \pm e^{i\alpha}|1\rangle; \\ |\pm_{\mathrm{XZ},\alpha}\rangle &= \cos(\tfrac{\alpha}{2})|0\rangle \pm \sin(\tfrac{\alpha}{2})|1\rangle; \\ |\pm_{\mathrm{YZ},\alpha}\rangle &= |+\rangle \pm e^{i\alpha}|-\rangle. \end{aligned} \tag{3.2}$$

For each of these, the "+" outcome represents the desired outcome "0", while the "−" outcome represents the outcome "1" that requires corrections later down the line.

These three measurement planes correspond to simple primitives in the ZX-calculus. Suppose we want to measure in the $|\pm_{\mathrm{YZ},0}\rangle$ basis, with outcome a. This corresponds to the operator $\langle +| + (-1)^a\langle -| = \sqrt{2}\langle a|$ on the measured qubit (and the identity everywhere else). But we know what simple states $|a\rangle$ look like in the ZX-calculus, so we can represent $\sqrt{2}\langle a|$ as . By using a variable, the ZX-diagram can represent both measurement outcomes at once. A similar derivation for the other angles and measurement planes gives the following ZX-formulation for all measurement bases we use:

Basis	ZX-diagram
$\|\pm_{\mathrm{XY},\alpha}\rangle$	
$\|\pm_{\mathrm{XZ},\alpha}\rangle$	
$\|\pm_{\mathrm{YZ},\alpha}\rangle$	

(3.3)

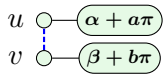
As in the simple example above, for each of these an outcome of $a = 0$ is the "desired" outcome that does not need any correction, and the $a = 1$ outcome introduces a π factor that needs to be handled later.

Definition 3.1.2 (The one-way model). A computation in the *one-way model* consists of:

- An initially specified graph state $|G\rangle$ on m qubits;
- A measurement order $M : [m] \to V(G)$, which is bijective;
- A measurement plane map $\lambda : V(G) \to \{\mathrm{XY}, \mathrm{XZ}, \mathrm{YZ}\}$, specifying the Bloch plane for each qubit;
- For each qubit $M(k)$, a measurement angle map $\alpha_k : \mathbb{F}_2^{k-1} \to \mathbb{R}$ that may depend on the outcome of the previous $k-1$ measurements;
- Some classical post-processing on the m measurement outcomes.

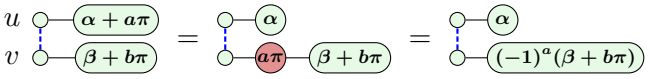
In step k of the computation, qubit $M(k)$ is measured in the basis $|\pm_{\lambda(M(k)),\alpha_k}\rangle$, and the outcome is stored for future $\alpha_{k'}$ to use. Note that while the angle may change during computation, the measurement plane is fixed.

Consider the following example diagram:



(3.4)

In this example, the initial graph state $|G\rangle$ consists of two $|+\rangle$ states with a CZ-gate in-between. As in the picture both u and v are measured in the XY-plane, $\lambda \equiv \mathrm{XY}$. We can then define a measurement order M by taking $M(1) = u$ and $M(2) = v$. The first qubit has no errors to correct, so it has a measurement angle of $\alpha_1 = \alpha$. However, the second qubit has to handle the case we accidentally measure the "wrong" outcome $a = 1$ for the first qubit, which corresponds to the "wrong" measurement plane $\pi + \alpha$. We can move this π factor through Equation (3.4) as follows:

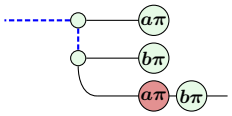


(3.5)

In the last step, we used the π-copy rule to absorb the $a\pi$ spider into the measured spider. This gives $\alpha_2 = (-1)^a(b\pi + \beta)$. After measuring this spider, the classical output consists of a and b, which can be used to post-process this computation.

The above definition and example of the one-way model represent a full computation. But just like in the circuit model, we also want to look at fragments of computation with open-ended inputs and outputs. A full MBQC computation completely uses up all of its qubits. If we want to do only a partial MBQC computation, some qubits will end up unmeasured. Because of the entangled initial state, the measured qubits affect the unmeasured qubits' outputs, so we need to consider what happens to these. We start with two examples to build the intuition of "what we want" and "what could happen".

The first example will be teleportation, as in Equation 3.1. We can translate the circuit to the following ZX-diagram[1]:

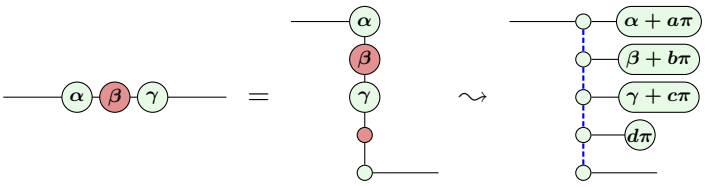


[1]The MBQC-formalism we will formulate does not allow Hadamard gates on the input or output wires, unlike this diagram.

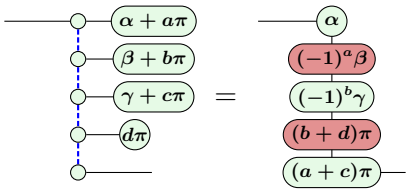
This looks exactly like a one-way model computation, but now we have inputs and outputs, and the output wire has some correcting spiders. Note that, if we apply the same measurement order as Equation 3.1, the $b\pi$ spider gets corrected to $(-1)^a b\pi$, which is the same phase as $b\pi$. Such "corrections" that do not actually do anything will become important in Section 3.2.

The errors on the output wires cannot be avoided, and the next diagram will need to deal with these incoming *feed-forward errors* $a\pi$ and $b\pi$.

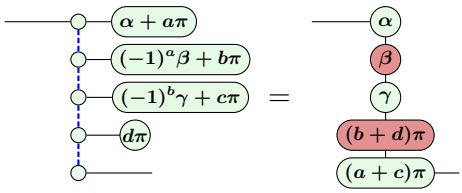
The second example shows what happens when we want to put a single-qubit unitary into MBQC-form. As mentioned earlier, we can decompose any single-qubit unitary as three spiders. Then, using Table (3.3) above, we convert each spider into a measurement. This process gives us the following:



However, we have not done anything with the measurement outcomes a through d yet. If you propagate these errors through the diagram, you will find that the above right-hand side does *not* implement the map on the left-hand side, but instead we have:

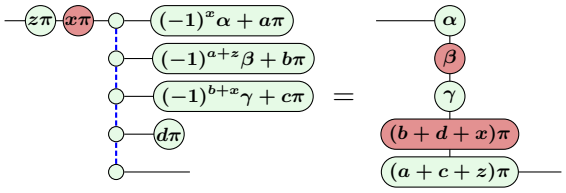


Just like the teleportation example, we get unavoidable feed-forward errors, $(b+d)\pi$ and $(a+c)\pi$ this time. But instead of —α—β—γ—, the remaining non-error part implements the map —α—$(-1)^a\beta$—$(-1)^b\gamma$—. These maps are not the same. Taking into account appropriate corrections for each measurement, we can achieve equality with the following corrections:



Up to the feed-forward error, this implements our original map —α—β—γ—. The error consists of Z and X gates, conditional on the measurement outcomes before. As all we are doing is adding π factors to the diagram, and moving around those factors only results in other π factors, these are the only type of error that will appear.

But if we can get erroneous Z and X gates as outputs, we can also get them as *inputs* from other fragments. These errors must also be taken into account. A complete picture with both fed-forward errors on the inputs and fully corrected measurements is the following:



Each measured spider now has taken into account two sources of error: the feed-forward errors on the inputs, and the fragment's own measurements. After this, feed-forward errors remain for another fragment after this one to handle.

We will formalize the above as a *measurement fragment*. For that, we first need to formalize the underlying structure also.

Definition 3.1.3 (Open graph). Let G be a graph. A *(labeled) open graph* is a tuple (G, I, O, λ), where $I, O, \subseteq V(G)$ are the *inputs* and *outputs* respectively, and $\lambda : \overline{O} \to \{\mathrm{XY}, \mathrm{XZ}, \mathrm{YZ}\}$ is a map assigning measurement planes to non-output vertices. Vertices in $I \cup O$ are called *boundary vertices*, while the others are *internal vertices*.

A note on terminology: throughout this thesis, we will only use "spider" to refer to the vertices in the underlying ZX-diagram. If we also want to include the interpretation of the open graph, we will call it a "vertex". For example, a single YZ-vertex consists of two spiders, as there is also the measurement effect from Table (3.3).

Definition 3.1.4 (Measurement fragment). A *measurement fragment* consists of:

- An open graph (G, I, O, λ), with $n_\mathrm{i} := |I|$, $n_\mathrm{o} := |O|$, and $m := \left|\overline{O}\right|$;
- A measurement order $M : [m] \to \overline{O}$, which is bijective;
- For each measured qubit $M(k) \in \overline{O}$, a measurement angle map

$$\alpha_k : \mathbb{F}_2^{k-1} \times \{\mathbb{I}, Z\}^{n_\mathrm{i}} \times \{\mathbb{I}, X\}^{n_\mathrm{i}} \to \mathbb{R}$$

  that may depend on the outcome of the previous $k-1$ measurement, and the feed-forward errors Z^z and X^x on the input qubits from a previous fragment;
- Maps

$$f_Z : \mathbb{F}_2^m \times \{\mathbb{I}, Z\}^{n_\mathrm{i}} \times \{\mathbb{I}, X\}^{n_\mathrm{i}} \to \{\mathbb{I}, Z\}^{n_\mathrm{o}}$$
$$f_X : \mathbb{F}_2^m \times \{\mathbb{I}, Z\}^{n_\mathrm{i}} \times \{\mathbb{I}, X\}^{n_\mathrm{i}} \to \{\mathbb{I}, X\}^{n_\mathrm{o}}$$

  giving the feed-forward errors Z^z and X^x on the output qubits of this fragment.

If you take a measurement fragment where $I = O = \varnothing$, the definition reverts to that of the one-way model (except for the post-processing). It is worth emphasizing that output vertices do not have a measurement plane or measurement angle.

With this framework, we want the fragment to have predictable behavior, no matter the measurement outcomes. Independent of the measurements, we want to implement the same linear map, where the only difference between measurement outcomes are the inevitable errors fed-forward to the next fragment. We want *determinism*.

Definition 3.1.5 (Determinism). A measurement fragment that, up to errors f_Z and f_X and a global phase, implements the same linear map no matter the measurement outcomes, is called *deterministic*.

We want this property, as deterministic measurement fragments can then be equated to some unitary, instead of also needing the measurement outcomes to interpret your map. An example of non-deterministic mechanisms are post-selections. When post-selecting, you either get the desired measurement outcome, or you discard the result of your algorithm entirely and try the whole computation again. These two behaviors are very different, and come down to chance! In the next section, it will be shown that the *gflow* condition implies determinism.

3.2. Flow

In this section, we will discuss converting a graph-like diagram to a measurement fragment. The first step is easy: to create the open graph, just consider every vertex an XY-plane vertex with its phase. (You could also consider a more sophisticated approach that recognizes the XZ-plane and YZ-plane measurement effects as in Table (3.3) in the diagram as well. This is unnecessary if your diagram comes from a circuit.) Inputs and outputs of the diagram also map naturally to the inputs and outputs of open graphs.

But once we have done this first step, defining a measurement order is a lot harder. This is where flow conditions come into play. In this section, we will slowly build up *flow conditions* until we get to *(triplanar) gflow*.

3.2.1. Causal flow

We will start with the simplest flow condition. It can only be applied to open graphs where all vertices are measured in the XY-plane, so it is somewhat limited, but it will prove to be a great stepping stone for us.

Definition 3.2.1 (Causal flow). Let (G, I, O, λ) be an open graph where $\lambda \equiv \mathrm{XY}$. Then G has *causal flow* if there exists a map $f : \overline{O} \to \overline{I}$ and a strict partial order $\prec$ on $V(G)$ such that the following flow conditions hold for all v in g's domain:

(f1) We have $v \prec f(v)$;

(f2) If $w \in N(\{f(v)\}) \setminus \{v\}$, then $v \prec w$;

(f3) Vertices v and $f(v)$ are neighbors.

If $u \prec v$, we say that u is in v's *past*, and v is in u's *future*.

The idea behind this definition is the following. If a vertex v has a measurement outcome of 1, we need to fix this undesired $+\pi$ phase, as we did in Example (3.4). We can fix this by changing the phase of $f(v)$ appropriately.

By changing the phase of $f(v)$, we will see that we add a phase of $+\pi$ to all its neighbors. This includes v itself, fixing its undesired $+\pi$ offset. We may only change the phase of unmeasured vertices, so both $f(v)$ and all of its neighbors must lie in v's future. The full argument is contained in the following lemma.

Lemma 3.2.2. Let (G, I, O, λ) be an open graph with causal flow. Then this graph can be efficiently transformed into a deterministic measurement fragment with the same underlying graph.

Proof. Let f and $\prec$ be the causal flow's map and partial order. To define a measurement fragment, we need to provide four things.

(i) We need an open graph.

This open graph can simply be chosen to be the open graph (G, I, O, λ) we started with.

(ii) We need a measurement order.

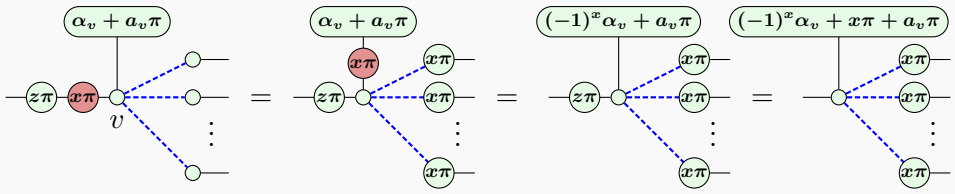
For this, we can take any order compatible with $\prec$. (As $\prec$ is a partial order, we can map vertices injectively into the integers where $\prec$ is maintained as $<$.)

(iii) We need a measurement angle map depending on incoming errors and prior measurements;

(iv) We need maps providing the feed-forward errors on the outputs.

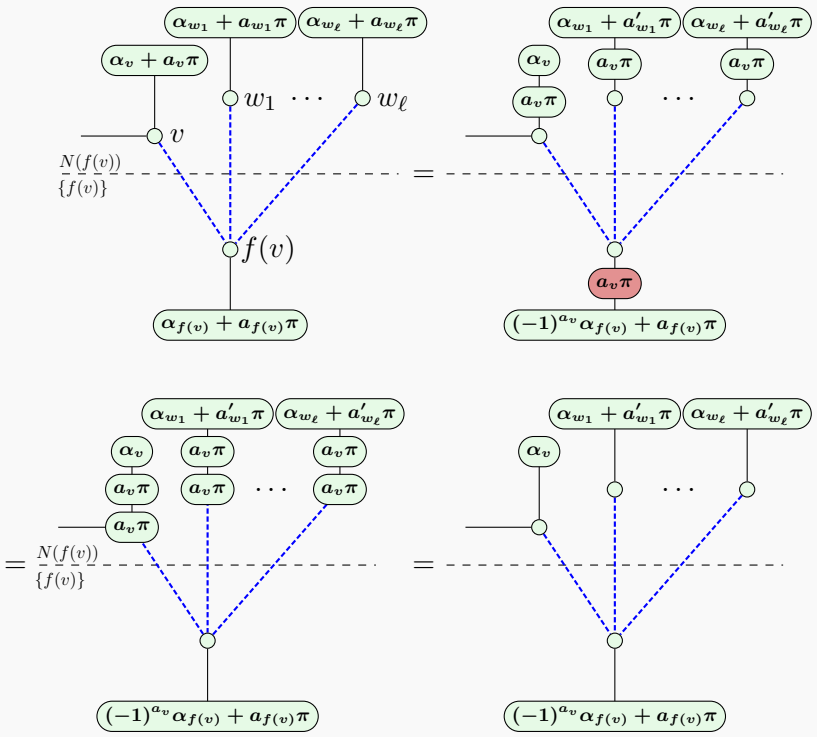
We will construct (iii) and (iv) from a different perspective: instead of looking at the resulting angle for each vertex, we look at how each input and vertex affects other vertices and outputs. This is equivalent. We want each vertex to only update future vertices.

We will start with input feed-forward errors. Suppose $v \in I$, which gets measured with phase α_v and has outcome a_v. This vertex may get some feed-forward error Z and X gates. We handle these as follows:



Here, we first use the π-copy rule to deal with the $x\pi$ spider. It gets absorbed into v's measurement effect by the π-copy rule again, and spreads to v's other neighbors with an additional color change. These neighbors absorb the $x\pi$ spider into their own measurement. After this, the $z\pi$ spider also gets absorbed into v's measurement effect. A similar computation holds if v is also an output, but in that case it forwards an error of $x\pi$.

We will now show what happens to an arbitrary vertex v. We can assume WLOG that v has no incoming errors, by absorbing these errors into its phase. Denote $f(v)$'s neighbor set to be $\{v, w_1, \ldots, w_\ell\}$. To correct an error a_v, we introduce a $a_v\,\pi$ spider at $f(v)$ and $a_v\,\pi$ spiders at all of $f(v)$'s neighbors. This notably removes the $a_v\pi$ phase from v's measurement. After this, we propagate the X-spider to $f(v)$'s neighbors as in the input case, with the π-copy rule. In the ZX-calculus, this looks as follows:



In the above equation, we wrote $f(v)$ and its neighbors in separate halves of the diagram. Each half introduces a different type of spider: the neighbor half introduces Z-spiders, while $f(v)$'s half introduces X-spiders. This will also be the case when considering the two other flows, later. We have also abbreviated $a'_{w_i} := a_{w_i} + a_v$.

On the right-hand side, we see that we have indeed removed any error at v by updating the measurement effect of some spiders. Furthermore, this update only happens to spiders in v's future by (f1) and (f2).

In the above derivation, we assume all vertices are internal vertices, but we can also introduce $\boxed{a_v\pi}$ spiders at outputs by introducing *two* (which maintains semantics). The second introduced spider then contributes to the output feed-forward error.

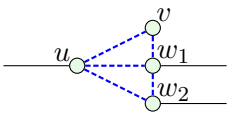
It is clear that the resulting map does not depend on the measurement outcomes in a non-scalar manner. While we did not discuss scalars in the ZX-calculus, it can indeed be shown that the measurement outcomes only influence the global phase, making it deterministic. Finally, all steps in this construction are only polynomial in the number of vertices, giving efficiency.

This "introduce spiders at $f(v)$ and its neighbors that cancel out"-trick fits into a larger framework of *stabilizers*. We will not be introducing this formalism, and simply redo this proof twice for the two more general flows.

While this is by far the most specialized and least-applicable flow, this also gives it more properties than the other flows. For instance, diagrams with causal flow can be extracted in a way where you know *a priori* on which qubit a vertex's resulting gates will lie [10]. We will not go into this, for we will be discussing a more general algorithm.

3.2.2. Uniplanar gflow

While causal flow is useful, it is applicable to only a small class of graphs. The first step to extending it, is allowing vertices v to not just have a single corrected vertex $f(v)$, but a whole set. This is needed in some cases. For example, consider the following diagram:



Here, suppose that we have already assigned $f(u) = v$, giving $u \prec v$ by (f1). By (f3), our only options to correct v are either $f(v) = u$ (giving a contradicting $v \prec u$ by (f1)), and $f(v) = w_1$ (giving a contradicting $v \prec u$ by (f2)). So, we're stuck.

However, the reason (f2) is formulated the way it is, is because each neighbor of $f(v)$ gains an additional $a_v\pi$ phase. If this happens once to u, we are in trouble, as that would change the past. But if this happens *twice* to u, the two $a_v\pi$ terms cancel out, leaving the past unchanged. This motivates taking $f(v)$ "=" $\{w_1, w_2\}$. In general, we will show below that we can relax "$f(v)$" to a set "$g(v)$" and still get a result like Lemma 3.2.2. We will still be limited to graphs that only have measurements in the XY-plane.

Motivated by this cancellation of $a_v\pi$ terms, we first start with the following definition.

Definition 3.2.3 (Odd neighborhood). Let G be a graph and $A \subseteq V(G)$ a set of vertices. Then the *odd neighborhood* of A is

$$\mathrm{Odd}(A) := \{v \in V(G) \mid v \in N(a) \text{ for an odd number of } a \in A\},$$

all vertices connected to an odd amount of vertices in A.

The odd neighborhood can be characterized as $\mathrm{Odd}(A) = \Delta_{a\in A} N(a)$, the symmetric difference of all individual neighborhoods of elements of A. We can now discuss the generalization of causal flow.

Definition 3.2.4 (Uniplanar gflow). Let (G, I, O, λ) be an open graph where $\lambda \equiv$ XY. Then G has *uniplanar gflow* if there exists a map $g : \overline{O} \to \mathcal{P}\left(\overline{I}\right)$ and a strict partial order $\prec$ on $V(G)$ such that the following flow conditions hold for all v in g's domain:

(g1) If $w \in g(v) \setminus \{v\}$, then $v \prec w$;

(g2) If $w \in \mathrm{Odd}(g(v)) \setminus \{v\}$, then $v \prec w$;

(g3) We have $v \notin g(v)$ and $v \in \mathrm{Odd}(g(v))$.

We call $g(v)$ the *correction set* of v.

These conditions (g1) through (g3) are generalizations of (f1) through (f3). We want the corrected vertices in v's future, which is the statement of both (f1) and (g1). We want every vertex that gets a $+\pi$ phase due to the corrections to also be in v's future, which is the statement of both (f2) and (g2). Finally, the odd neighborhood is adjacent to v itself, so (g3) generalizes (f3).

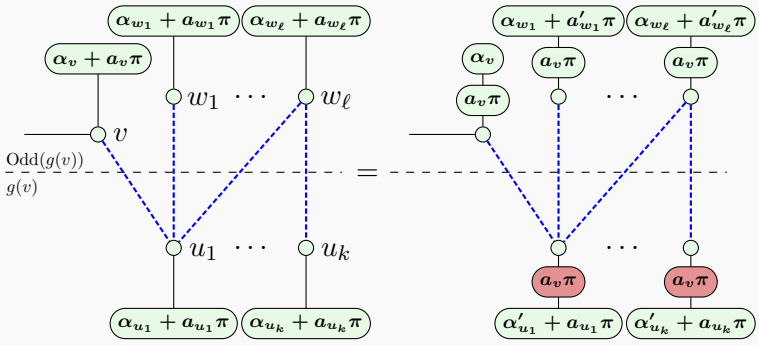
Note how specifically (g2) is a looser generalization of (f2) than one would expect at first glance: not *all* neighbors of $g(v)$ need to be in v's future. This is because of the cancellation of phases discussed above.

With this, we once again have the following result.

Lemma 3.2.5. Let (G, I, O, λ) be an open graph with uniplanar gflow. Then this graph can be efficiently transformed into a deterministic measurement fragment with the same underlying graph.

Proof. We will be mirroring the proof of Lemma 3.2.2. Parts (i) and (ii) are unchanged. For (iii) and (iv), the discussion about feed-forward errors also stays the same. We only need to update the computation of correcting a set $g(v)$ instead of only a single vertex $f(v)$ as in Lemma 3.2.2.

Just like before, suppose v has no incoming errors, and we get outcome a_v. Denote $g(v) = \{u_1, \ldots, u_k\}$ with $\mathrm{Odd}(g(v)) = \{v, w_1, \ldots, w_\ell\}$. To correct outcome a_v, we introduce $a_v\pi$ spiders at $g(v)$ and $a_v\pi$ spiders at $\mathrm{Odd}(g(v))$. This looks as follows:

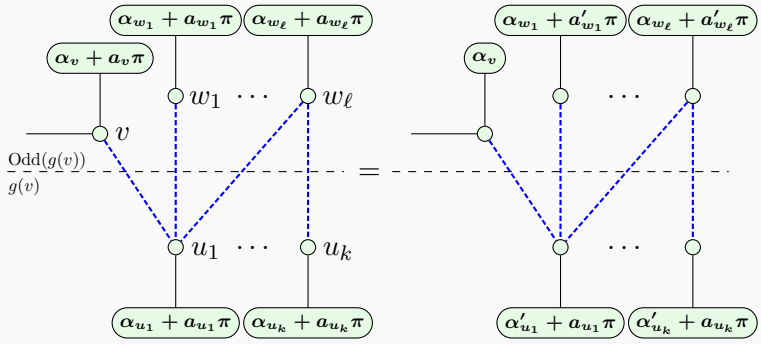


We have drawn this diagram with some edges in $g(v)$ connected to v and some not, and similarly from $g(v)$ to $\mathrm{Odd}(g(v))$. We have notably *excluded* from the diagram all neighbors of the u_i not in $\mathrm{Odd}(g(v))$. These may exist. We have also abbreviated $a'_{w_i} := a_{w_i} + a_v$ and $\alpha'_{u_i} := (-1)^{a_v}\alpha_{u_i}$. Finally, we have drawn $g(v)$ and $\mathrm{Odd}(g(v))$ disjoint, but this is not necessarily the case. One can check that introducing both spiders at vertices in $g(v) \cap \mathrm{Odd}(g(v))$ does not pose problems.

At this point, we do the same as in Lemma 3.2.2 and push the $a_v\pi$ spiders through the u_i. There are two cases to consider.

- Consider v, or any w_i. These lie in the odd neighborhood of $g(v)$, so they will all receive an odd number of $a_v\pi$ spiders. (Compare this with "1 spider" in Lemma 3.2.2.) Combined with their own $a_v\pi$ spider, all $a_v\pi$ spiders cancel out.
- Any other spider $\tilde{v}$ is necessarily connected to an even number of spiders in $g(v)$. This means that $\tilde{v}$ receives an even number of $a_v\pi$ spiders. As $\tilde{v}$ itself does not introduce an $a_v\pi$ spider, these $a_v\pi$ terms cancel out and nothing changes.

So just like before, we have now shown that:



By (g1) and (g2), we have only updated vertices in the future of v, so we are done. Again, this derivation assumes that all vertices are internal, but it can easily be extended to outputs as well.

3.2.3. Triplanar gflow

When going from causal flow to uniplanar flow, we went from single-vertex correction to entire correction sets. Both only apply to graphs with all measurements in the XY-plane. We will now finally use the other two Bloch planes as well.

Definition 3.2.6 (Triplanar gflow). Let (G, I, O, λ) be an open graph. Then G has *(triplanar) gflow* if there exists a map $g : \overline{O} \to \mathcal{P}\left(\overline{I}\right)$ and a strict partial order $\prec$ on $V(G)$ such that the *gflow conditions* hold for all v in g's domain:

(g0) If $w \in g(v)$, then $w \notin I$. *(This is already listed above, but worth emphasizing.)*

(g1) If $w \in g(v) \setminus \{v\}$, then $v \prec w$;

(g2) If $w \in \mathrm{Odd}(g(v)) \setminus \{v\}$, then $v \prec w$;

(g3) If $\lambda(v) = \mathrm{XY}$, then $v \notin g(v)$ and $v \in \mathrm{Odd}(g(v))$;

(g4) If $\lambda(v) = \mathrm{XZ}$, then $v \in g(v)$ and $v \in \mathrm{Odd}(g(v))$;

(g5) If $\lambda(v) = \mathrm{YZ}$, then $v \in g(v)$ and $v \notin \mathrm{Odd}(g(v))$.

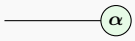
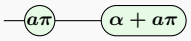
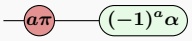
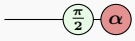
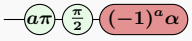
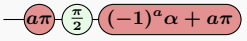
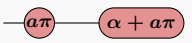
We call $g(v)$ the *correction set* of v.

In the sequel, whenever we refer to "gflow", we refer to this triplanar gflow. This is the same definition as Definition 3.2.4, but with two more planes and thus two more conditions added. Perhaps surprisingly, the two extra conditions (g4) and (g5) do not make our life much more difficult than what we have done so far. We only need two small lemmata to aid exposition.

Lemma 3.2.7. Let (G, I, O, λ) an open graph, and let $v \in I$ be such that $\lambda(v) \neq \mathrm{XY}$. Then G does not have gflow.

Proof. By (g0), v may not contain itself. However, if $\lambda(v) \neq \mathrm{XY}$, (g4) or (g5) forces this. As such, v cannot have a correction set.

Lemma 3.2.8. Diagrams in the same row of the following table are equal:

	Nothing added	$a\pi$ added	$a\pi$ added
Output wire			
XY-plane			
XZ-plane			
YZ-plane			

In other words, we may introduce $a\pi$ and $a\pi$ spiders at every measurement and output at will, at the cost of changing the measurement angle or adding a feed-forward error.

Proof. This is trivially true for the "output" row, and the "XY-plane" and "YZ-plane" rows are applications of spider fusion or the π-copy rule. The entries of the "XZ-plane" row require a bit more work.

We start with the case of adding a $a\pi$ spider. Fusing this spider with the $\frac{\pi}{2}$ spider may flip its phase. We can undo this flip by applying the π-copy rule on the leftmost spider, introducing *another* $a\pi$ spider to cancel this flip.

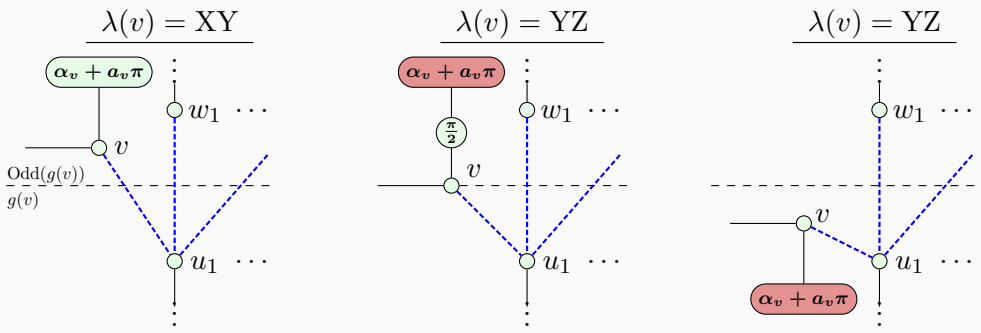
The $a\pi$ spider case is similar. When copying $\frac{\pi}{2}$ to $a\pi$'s other side, the $\frac{\pi}{2}$ spider may turn into a $-\frac{\pi}{2}$ spider. We can then fuse $a\pi$ into α, and conditionally flip the sign of the resulting $\alpha + a\pi$ spider as in the previous case to get a $\frac{\pi}{2}$ spider again.

Now we can prove that gflow gives us a deterministic measurement fragment, in full generality.

Proposition 3.2.9. Let (G, I, O, λ) be an open graph with gflow. Then this graph can be efficiently transformed into a deterministic measurement fragment with the same underlying graph.

Proof. We will once again redo the proof of Lemma 3.2.2, like we did in Lemma 3.2.5. Parts (i) and (ii) still remain unchanged, and we again need to look at (iii) and (iv). Note that despite allowing more than just the XY-plane, the discussion about feed-forward errors on inputs remains unchanged by Lemma 3.2.7. We only need to update the computation where we correct a set $g(v)$ for arbitrary planes.

Suppose vertex v has no incoming errors, and we measure a_v. We want to correct this like before. We have three cases to consider:



The three cases are not only different in what the measurement looks like, but also what sets v lies in. This is determined by (g3), (g4), and (g5). For instance, when $\lambda(v) = \text{YZ}$, v lies in both $g(v)$ and $\text{Odd}(g(v))$. In these diagrams, we drew spiders in $g(v)$ and $\text{Odd}(g(v))$ having an edge going into triple dots. This endpoint may either represent an output, or a measurement effect.

The approach of "extract an $a_v\pi$ for each spider in $g(v)$ and an $a_v\pi$ for each spider in $\text{Odd}(g(v))$" from Lemma 3.2.5 remains unchanged. We can do this for every spider in the diagram by Lemma 3.2.8. The consequences of this for every vertex other than v is also the same. What remains is to argue that each case gets rid of the $a_v\pi$ factor in v's measurement:

- When $\lambda(v) = \text{XY}$, v is only in $\text{Odd}(g(v))$ by (g3), and we extract an $a_v\pi$ spider. This leaves an α_v behind, like we wanted.
- When $\lambda(v) = \text{YZ}$, v is only in $g(v)$ by (g5), and we extract an $a_v\pi$ spider. This leaves an α_v behind, like we wanted.
- When $\lambda(v) = \text{XZ}$, v is part of both, by (g4), and we extract both an $a_v\pi$ spider and an $a_v\pi$ spider. Looking at Lemma 3.2.8, the $a_v\pi$ factor gets removed when extracting the X-spider, which also introduces a phase flip. Extracting the Z-spider also introduces a flip. After applying both of these, the end result removes the $a\pi$ error, leaving only an α_v phase behind, like we wanted.

It is known that it can be efficiently determined whether gflow exists, and if so, compute it.

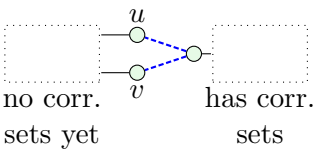
Proposition 3.2.10 (Theorem C.6 in [4]). There exists an algorithm that decides whether an open graph with n vertices has gflow that runs in $\mathcal{O}(n^4)$ time, and outputs it if it exists.

One large class of graphs in particular does not admit any gflow.

Proposition 3.2.11. No open graph (G, I, O, λ) with $|I| > |O|$ has gflow.

Proof. Proposition 7.3.4 in [11] states that deterministic measurement fragments represent unitary embeddings. An open graph with gflow (which is deterministic) with $|I| > |O|$ would contradict this.

This statement can also be proven directly from the definition of gflow instead of relying on determinism, by considering the following special case:



No matter what measurement effects we attach to u and v, this diagram does not have gflow. The idea for the general case is that any graph with $|I| > |O|$ encounters this subgraph at some point in the process of assigning correction sets to all vertices, but a formal proof of this relies on the algorithm for finding *maximally delayed flows* in [4].

One final thing to note is that gflow is still not the most elaborate form of flow. Gflow does not take into account measurement angles. For most angles, this is not relevant. However, the three Bloch sphere planes XY, XZ, and YZ intersect at $|0\rangle$, $|1\rangle$, $|+\rangle$, $|-\rangle$,

$|i\rangle$, and $|-i\rangle$. This creates a *choice* in what plane you are actually performing your measurement in. This choice is significant.

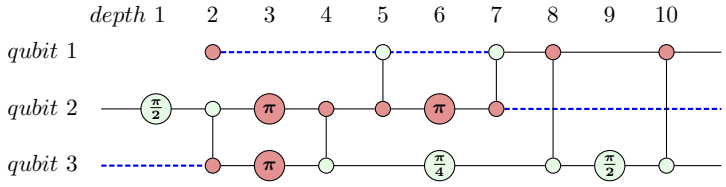
Consider for instance an input measured as $|\pm_{\mathrm{XY},0}\rangle$ (recall Equation (3.2)). This measures a vertex in the $|\pm\rangle$ basis. However, $|\pm_{\mathrm{XZ},\pi/2}\rangle$ *also* measures the $|\pm\rangle$ basis. By Lemma 3.2.7, this decision makes or breaks the existence of gflow.

This means that choosing the wrong representation for your diagram can make the difference between "having gflow" and "not having gflow". As we will see in Section 3.6, this can make the difference between "extractable" and "unextractable", which is quite a significant difference! This is a fundamental problem with gflow that can be solved by considering *Pauli flow* instead. This flow takes into account these special bases that can be seen as multiple planes at once. We will not go into this further, but more information about Pauli flow can be found in [23].

3.3. Graphs translated from circuits have gflow

Now that we have finally introduced gflow, we want graphs to have this property. Fortunately [13] showed that if you start out with a circuit, turn it into a ZX-diagram by directly translating the gates, and then turn it into a graph-like diagram as in Proposition 2.2.11, you end up with a graph with gflow. This is what we will discuss now.

First note that any causal flow as specified by f and $\prec$ can be converted to a gflow. We do this by setting $g(v) = \{f(v)\}$ for all v, and keeping $\prec$ as-is. As such, it suffices to show that a graph translated from a circuit has causal flow. We assume we start with a ZX-diagram that has every gate converted into spiders, but without any optimization applied. This gives every spider a well-defined *qubit* it lies on and *depth* in the circuit:



Definition 3.3.1 (Circuit-like diagram). A *circuit-like diagram* is a ZX-diagram in which every spider v has an assigned *qubit line* $q_v \in \mathbb{N}$ and *depth* $d_v \in \mathbb{N}$, such that:

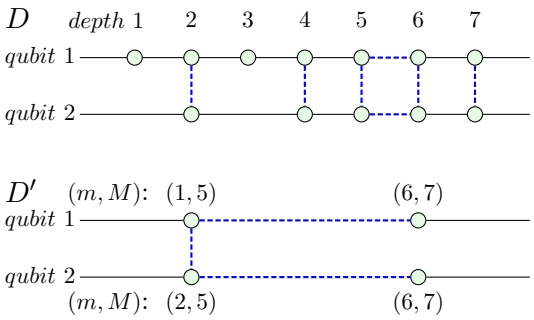
- No two spiders are at the same position: if $(q_v, d_v) = (q_w, d_w)$, we have $v = w$;
- Spiders lie on a grid: for edges vw, either $q_v = q_w$, or $d_v = d_w$; and no vertex has multiple neighbors of higher depth.

We will also assume that all X-spiders have been converted to Z-spiders already, and that if there are connected spiders $v \neq w$ with $d_v = d_w$, they are part of a CZ-gate. More complex multi-qubit gates can be decomposed so that the only multi-qubit gates are CZ. Denote this resulting diagram by D. This diagram corresponds to the diagram after step (1) of Proposition 2.2.11. Finally, denote the diagram after step (4) by D'. Going from step (4) to step (5) is a technicality that does not remove the existence of gflow, so we will ignore it.

Connected spiders on different qubit lines in D cannot be fused, as there is a Hadamard-edge between them. We only need to consider fusion on a single qubit line when looking at the transformation from D to D'. This motivates the following definition.

Definition 3.3.2. Let v be a spider in D'. Define q_v to be the qubit line value inherited from the spiders that fused into v from D, and define m_v and M_v as the minimum and maximum depth of these spiders, respectively.

To make this definition more clear, consider this simple diagram D that turns into D' as described above.



Take for instance the top left spider in D'. As there are no Hadamards between the spider on depth 1 and depth 5, these are all fused together. The Hadamard between depth 5 and depth 6 prevents fusion, and everything after can be fused again.

Due to the grid-like nature of D, we can relate the various m and M values in D' to one another.

Lemma 3.3.3. Let v be adjacent to w in D'.

- If $q_v = q_w$, then either $M_v < m_w$ or $M_w < m_v$;
- Otherwise, $m_v \leq M_w$ and $m_w \leq M_v$.

Proof. We begin with the first bullet point, where $q_v = q_w$. Suppose for a contradiction that $m_w \leq M_v$ and $m_v \leq M_w$. WLOG, let $m_v \leq m_w$. This gives us $m_w \leq m_v \leq M_w$, meaning that the range of spiders that merges into w intersects the range that merges into v. Then necessarily $v = w$, which contradicts the fact they are adjacent as D' has no self-loops.

In the second bullet point, v and w lie on qubit lines, meaning that some qubits that fused into v and some qubits that fused into w had to be connected by a Hadamard-edge. As Hadamard-edges are vertical in D, this gives us $m_v \leq d \leq M_v$ and $m_w \leq d \leq M_w$ for some depth d. This then gives $m_v \leq d \leq M_w$ and $m_w \leq d \leq M_v$.

Proposition 3.3.4. Let D' be the result of converting a circuit without measurements to a circuit-like diagram and then a graph-like diagram with Proposition 2.2.11. Let I be all spiders in D' connected to an input, O all spiders connected to an output, and let $\lambda \equiv \mathrm{XY}$. Then (G, I, O, λ) has a causal flow.

Proof. We first need to define an f and a partial order $\prec$. For $v \in \overline{O}$, let $f(v)$ be the unique spider connected to v after v on the same qubit line. This vertex exists, as we do not have measurements. This already takes care of (f3). Define $v \prec w$ when $M_v < M_w$, which gives a partial order. By the first part of the previous Lemma 3.3.3, $M_v < m_{f(v)} \leq M_{f(v)}$ giving (f1).

What remains is (f2). Let w be a non-v neighbor of $f(v)$. We want $v \prec w$. This spider either lives on the same qubit line as $f(v)$, or a different one.

First suppose $q_w = q_{f(v)}$. As $f(v)$ has at most two neighbors on the same line, and one of these is v, we must have $v \prec f(v) \prec f(f(v)) = w$, by Lemma 3.3.3's first part.

Finally, if $q_w \neq q_{f(v)}$, the same lemma's second part gives us $m_{f(v)} \leq M_w$. This can be combined with $M_v < m_{f(v)}$ into $M_v < M_w$, giving $v \prec w$, as we wanted.

Note that we do not assume the circuit fragment we start out with is unitary, and it may contain preparations. This means that we also have gflow when starting with isometries.

3.4. Operations that maintain gflow

Now that we know starting with circuits gives us gflow, we want to maintain it. In this section, we will give a comprehensive list of how various operations maintain the existence of gflow. While it is not the flow itself that we maintain, but merely the existence of *some* flow, we will abbreviate "maintain the existence of gflow" as "maintain gflow" from here on out.

While this section concerns itself with maintaining gflow, some operations that do not maintain gflow are discussed in the next section. The aim of these two sections is to be a quick reference of what is and is not allowed when rewriting the graphs.

Most of these statements come from [4]. The proofs of these lemmata do not contribute much to our story and have been moved to Appendix A. The first operation of note we will consider is the local complementation, as introduced in Definition 2.2.12.

Lemma 3.4.1 (Local complementations maintain gflow). Let (G, I, O, λ) be an open graph with gflow, and $u \in \overline{I}$ not be an input. Then $(G * u, I, O, \lambda')$ has gflow.

Here, λ' is the same as λ, except for two changes. For $v \in N(u) \cap \overline{O}$, the XZ- and YZ-planes swap, and if $u \in \overline{O}$, the XY- and XZ-planes swap.

This change in planes ultimately comes from the $\frac{\pi}{2}$ and $-\frac{\pi}{2}$ spiders Proposition 2.2.13 introduces. This also changes the measurement phases. Note that λ may not be defined for u or vertices in $N(u)$ if they are outputs. In these cases, the introduced spiders will be put after the outputs of the graph into the surrounding context.

We will not be applying local complementations to input spiders, as this would change their plane to the YZ-plane. The resulting graph cannot have gflow by Lemma 3.2.7.

As pivots are simply three local complementations, these also maintain gflow.

Lemma 3.4.2 (Pivoting maintains gflow). Let (G, I, O, λ) be an open graph with gflow, and $u, v \in \overline{I}$ be connected. Then $(G \wedge uv, I, O, \lambda')$ has gflow.

Here, λ' is the same as λ, except that for each of u and v, the XY- and YZ-planes swap if not in O.

Just like the local complementation, the pivot may also change phases in the graph, and spiders may also be introduced after outputs.

Having just these two operations is fairly powerful already, as we can restrict the graphs we consider even further: we can go from graph-like diagrams to diagrams in *phase gadget form*.

Definition 3.4.3 (Phase gadget form). An open graph (G, I, O, λ) is in *phase gadget form* if:

- There do not exist neighbors $v, w \in \overline{O}$ with $\lambda(v) = \lambda(w) = \text{YZ}$;
- There does not exist a $v \in \overline{O}$ with $\lambda(v) = \text{XZ}$.

Proposition 3.4.4. Let (G, I, O, λ) be an open graph with gflow. This graph can be put into phase gadget form efficiently, maintaining gflow.

Proof. Consider the algorithm that repeatedly tries to pivot all pairs of vertices violating the first bullet point of Definition 3.4.3, local complement all single vertices violating the second bullet point, and halts when neither can be found.

Each iteration that applies Lemma 3.4.2 turns two YZ-vertices (non-input by Lemma 3.2.7) into XY-vertices. All other measurement planes remain unaffected.

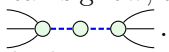
Each iteration that applies Lemma 3.4.1 turns an XZ-vertex into an XY-vertex, and maintains the measurement plane of XY-plane neighboring vertices.

Finding the vertices that these operations apply to and the operations themselves are polynomial-time in the number of vertices. Each step of the algorithm strictly reduces the amount of non-XY-vertices. As such, the entire process is of polynomial-time.

Apart from these two essential transformations, there are a lot more transformations interesting to us. The most interesting one states that we can delete many vertices in our graphs.

Lemma 3.4.5 (Deleting non-XY-vertices maintains gflow). Let (G, I, O, λ) be an open graph with gflow, and $u \in \overline{O}$ with $\lambda(u) \neq \text{XY}$. Then $(G \setminus \{u\}, I, O, \lambda)$ has gflow.

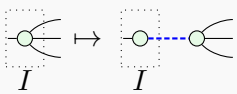
Lemma 3.4.6 (Deleting pivot maintains gflow). Let (G, I, O, λ) be an open graph. Let $u_1, v \in \overline{O}$ and $u_2 \in V$ such that $N_G(v) = \{u_1, u_2\}$ and $\lambda(u_1) \neq \text{YZ} \neq \lambda(v)$. Then $(G \wedge u_1 v \setminus \{u_1, v\}, I, O, \lambda)$ has gflow.

While this states that Proposition 2.2.17 maintains gflow, there is one particular common fragment this statement applies to, namely  . This fragment collapses into just a single vertex. In this case, it gets called *identity removal*, as it corresponds to the ZX-rewrites of Lemma 2.2.2.

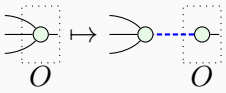
Note that the inverse of the above two statements do not hold; you cannot simply add vertices and expect to maintain gflow. We will discuss this some more in Section 3.5.

We can also consider some boundary-vertex-specific operations.

Lemma 3.4.7 (Extending inputs maintains gflow). Let (G, I, O, λ) be an open graph with gflow, and $u \in I$. Then the following rewrite that "prepends" a vertex to u maintains gflow:



Lemma 3.4.8 (Extending outputs maintains gflow). Let (G, I, O, λ) be an open graph with gflow, and $u \in O$. Then the following rewrite that "appends" a vertex to u maintains gflow:



(Note that after this rewrite, the former output is seen as an XY-vertex with measurement angle 0.)

The inverse operation also maintains gflow.

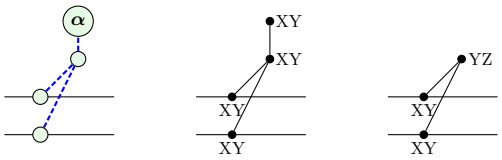
Note that many of the rewrites in this section maintain gflow, but not the semantic meaning of the graph. For instance, the last four lemmata change the meaning of the graph.

Lemma 3.4.9 (Changing edges between outputs maintains gflow). Let $(G = (V, E), I, O, \lambda)$ be an open graph with gflow, and let $u, v \in O$. Then $(G' = (V, E \,\Delta\, \{uv\}), I, O, \lambda)$ has gflow.

Lemma 3.4.10 (Adding CNOTs after outputs maintains gflow). Let (G, I, O, λ) be an open graph with gflow, and let $u, v \in O$. Then updating $N(u)$ to $N(u) \,\Delta\, N(v)$ and keeping the rest of the graph as-is maintains gflow.

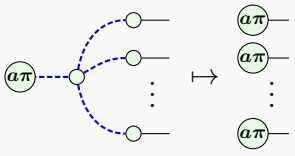
This change corresponds to adding a CNOT after u and v as in Proposition 2.2.18, where u corresponds to the control of the CNOT, while v corresponds to the target.

Phase gadgets also have some operations specific to them. We assume that the phase gadgets are implemented in the open graph as single YZ-vertices, and not as a pair of connected XY-vertices. More concretely, we assume that in the below figure, the ZX-diagram on the left does not represent the middle graph, but the right graph.



Lemma 3.4.11 (Gadget fusion maintains gflow). Let (G, I, O, λ) be an open graph with gflow. Then the rewrite of Proposition 2.3.1 maintains gflow.

Lemma 3.4.12 (Applying the state-copy rule maintains gflow). Let (G, I, O, λ) be an open graph with gflow. Then the following rewrite maintains gflow:



Finally, we can take the tensor product or compose graphs without losing gflow.

Lemma 3.4.13 (Combining graphs maintains gflow). Let $(G_1, I_1, O_1, \lambda_1)$ and $(G_2, I_2, O_2, \lambda_2)$ be open graphs with disjoint vertex sets, with gflow. Then $(G_1 \cup G_2, I_1 \cup I_2, O_1 \cup O_2, \lambda_1 \cup \lambda_2)$ also has gflow.

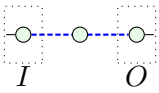
(Here, $G_1 \cup G_2$ means "union the vertex and edge sets", and $\lambda_1 \cup \lambda_2$ combines the two maps into one, which is well-defined due to their disjoint domain.)

Lemma 3.4.14 (Composing graphs maintains gflow). Let $(G_1, I_1, O_1, \lambda_1)$ and $(G_2, I_2, O_2, \lambda_2)$ be open graphs with disjoint vertex sets, with gflow. Let $|I_2| = |O_1|$. Then the concatenated graph that is the result of identifying the vertices of I_2 with O_1 (inheriting both graphs' measurements) results in a graph with gflow.

3.5. Operations that do not maintain gflow

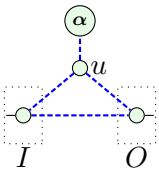
In Chapter 4, we will also be interested in operations that do not maintain gflow. As a bit of a preview, in this section we will discuss some of these.

The most obvious example is the exclusion of XY-vertices in Lemma 3.4.5. While in niche cases, removing XY-vertices may not remove gflow, in most cases it will. Consider the following example from [4].



Removing the middle vertex gives a part on the left with fewer outputs (zero) than inputs (one), which cannot have gflow by Proposition 3.2.11. In general, removal of an XY-plane vertex u is unsafe if $u \in g(v)$ for some v, which is *very* common. If your graph has many more outputs than inputs, the gflow may have an alternate path "around" u, but this is not guaranteed.

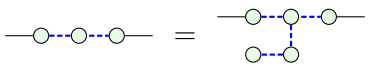
You also cannot do the opposite of Lemma 3.4.5 and add an arbitrary XZ- or YZ-vertex and expect to maintain gflow. Consider for instance the following fragment:



(3.6)

Here, u is a vertex measured in the YZ-plane with phase α. Before adding u (and its measurement effect --(α)), we have a simple line diagram with gflow. But with u present, there is no gflow. Showing this is easy: by (g5), u must be in $g(u)$, so it must lie in the past of both I and O. But by (g3), all possible correction sets for the input contain u either directly, or in the odd neighborhood, so u must lie in the future of I. These requirements are incompatible.

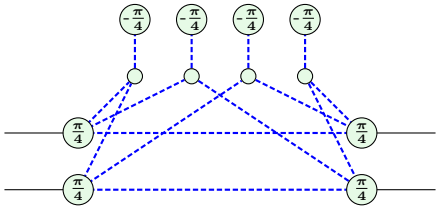
The next example is related to the above. While Lemma 3.4.6 maintains gflow, doing the opposite can remove gflow. A somewhat boring example is the following:



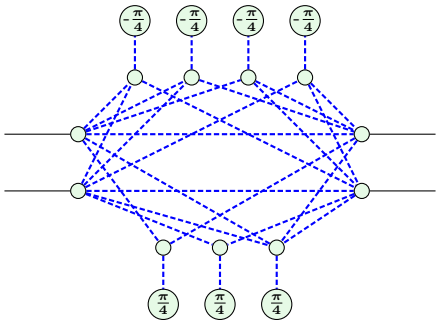
Here, every vertex is interpreted as an XY-vertex with zero phase. While the left-hand

side has gflow, the right-hand side does not. Note that [23] gives necessary and sufficient conditions for this to maintain gflow for diagrams implementing unitaries, and [7] shows that this example *does* maintain the more general *Pauli-flow*.

The elephants in the room are the spider nest identities. These identities are desirable, because they can reduce the T-count quite significantly. Yet their application may remove gflow. Consider for instance the following diagram.

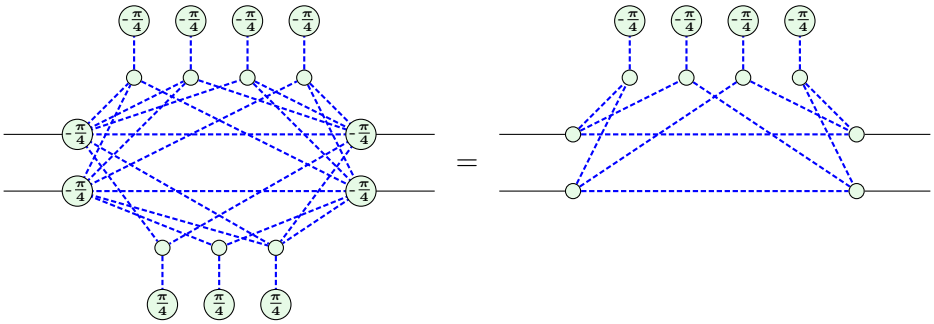


This diagram has gflow, which may be checked algorithmically. By applying the four-qubit spider nest identity, Proposition 2.3.5, we obtain the following diagram.



With this, we went from eight T-phases to seven. However, the diagram no longer has gflow! This is easiest seen by the multiple instances of Equation (3.6) in this result.

This is not all, however, as the converse also holds. By changing a few phases in the above diagrams we get another example:



In this example, not only did we go from eleven T-phases to just four T-phases, we also *gained* gflow. We will study this behavior more in-depth in Section 4.4 once we have defined *glack*.

3.6. Circuit extraction for unitary graphs with gflow

Going from a circuit to a graph is easy, as we saw in Section 2.1.1. On the other hand, the reverse direction is much more difficult, and requires gflow, which took quite some time to introduce. In this section, we will finally discuss the two extraction algorithms

in [4] to go from a ZX-diagram back to a circuit. This algorithm handles the special case of *unitary* graphs with gflow.

3.6.1. The basic algorithm

The general idea of the algorithm is that we start at the outputs, and use the existence of gflow to turn a vertex in the graph into regular circuit gates. We work our way back until we reach the inputs, and then we are as good as done. We need some tools before we can get to the algorithm itself.

Lemma 3.6.1. Let (G, I, O, λ) be an open graph in phase gadget form with gflow, and with $\overline{O} \neq \varnothing$. Then there exists a gflow $(g, \prec)$ with the following property: among vertices in $\overline{O}$, there is $\prec$-maximal vertex v connected to an output vertex. This vertex can be found efficiently.

Proof. This is Lemma 5.4 in [4].

We do not include this proof, because it relies on *maximally delayed gflows*, which we do not discuss in this thesis.

Lemma 3.6.2. Let (G, I, O, λ) be an open graph in phase gadget form with gflow, such that:

- Output vertices are not connected;
- There exists an XY-vertex v as in Lemma 3.6.1 connected to output w.

For each $w' \in g(v) \setminus \{w\}$, apply a CNOT with control w and target w' as in Lemma 2.2.18, where we consider the CNOT itself part of the extracted diagram. Then in the resulting graph, w is only connected to v. This operation maintains gflow.

Proof. Lemma 2.2.18 maintains gflow by Lemma 3.4.10, meaning that this operation maintains gflow. We now need to show that this results in a graph where w is only connected to v.

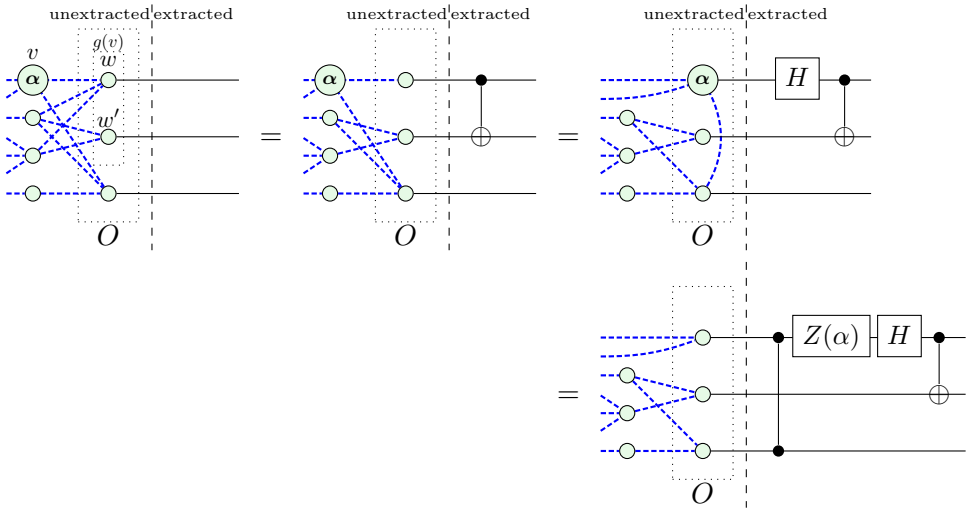
As v is $\prec$-maximal in $\overline{O}$ and $\lambda(v) = \mathrm{XY}$, $g(v)$ is contained in O. Now consider the biadjacency matrix M between $g(v) = \{w = w_1, \ldots, w_k\}$ and $N(g(v)) = \{v, u_1, \ldots, u_\ell\}$. (Note the assumption that the output vertices are not connected.)

Ensuring w is only connected to v in the graph corresponds to making row w of M zero at all u_i, and 1 at v in the biadjacency matrix. As noted in the discussion below Lemma 2.2.18, adding CNOTs corresponds to adding rows of the biadjacency matrix (in $\mathbb{F}_2$). This means that we want to show that adding all rows w_i for $i > 1$ to row w_1 results in what we described above.

But by (g3), we know $\mathrm{Odd}(g(v)) \cap N(g(v)) = \{v\}$. This means that an odd number of rows w_i contain a 1 at v, resulting in a sum of 1. Similarly, every u_j is not part of this odd neighborhood, meaning that an even number of rows w_i contain a 1 at u_j, resulting in a sum of 0. This means that adding all the rows of the biadjacency matrix together, and thus adding all CNOTs as in the statement, isolate w as desired.

While we care about gflow on a theoretical level because it guarantees the diagram is deterministic, this lemma is where we use the existence of gflow in practice. Without gflow, we would not be able to apply CNOT gates to isolate w, which in turn prevents extraction.

The algorithm makes use of a *frontier*, separating the unextracted and extracted parts of the diagram. This frontier is the current set of output vertices. The step-by-step process mentioned above moves these output vertices of the frontier into the extracted part, and then gives an internal vertex the role of being a new output in its stead. We will illustrate this idea with a small example, where we free frontier vertex w with Lemma 3.6.2 and then move v to be a new output vertex:



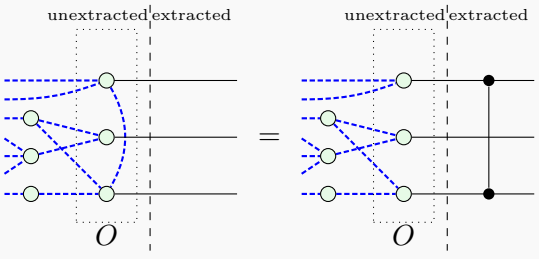
The first equality is Lemma 3.6.2. After this, we move the (α) spider into the frontier, which turns the Hadamard wire into a Hadamard gate in the extracted circuit. Finally, we clean up the frontier by extracting the α phase as a $Z(\alpha)$ gate, and the connection within the frontier as a CZ-gate. We do this clean-up because output vertices are not measured, meaning they may not have a phase, and Lemma 3.6.2 assumes there is no connection between output vertices.

However, there are some complications when trying to do this for every vertex. For instance, in the above example, there is no vertex as in Lemma 3.6.1 anymore in the diagram we end up with, so we are stuck. We will now turn to the full extraction algorithm to handle all complications that arise.

Theorem 3.6.3. Let (G, I, O, λ) be an open graph with gflow representing a unitary. Consider the following algorithm:

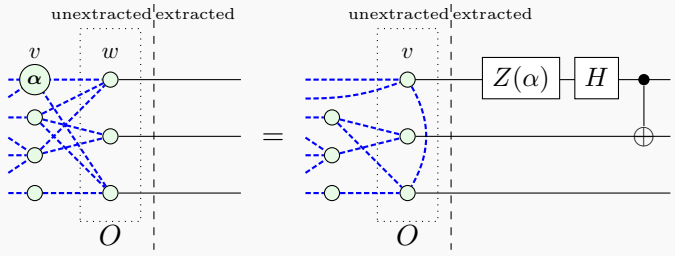
(Step 0) Convert the graph into phase gadget form with Proposition 3.4.4.

(Step 1) Unfuse connected output vertices as CZ-gates in the extracted circuit:

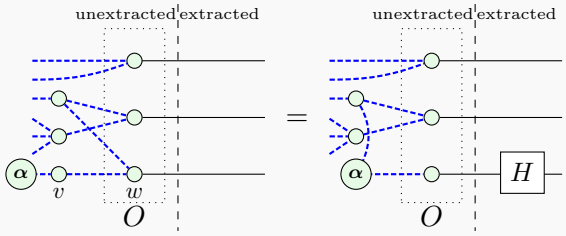


(Step 2) Pick a vertex $v \in \overline{I \cup O}$ as in Lemma 3.6.1, with an output neighbor as w. If $\lambda(v) = \text{XY}$, go to Step 3. If no such XY-vertex exists, $\lambda(v) = \text{YZ}$ and go to Step 4. If the only remaining vertices are inputs, go to Step 5 instead.

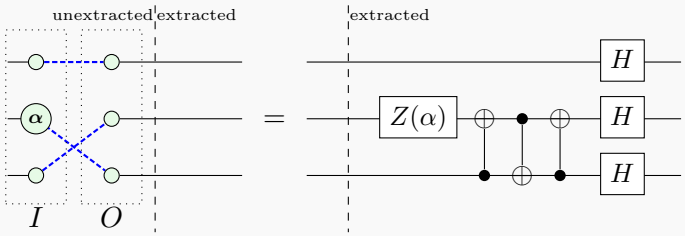
(Step 3) Apply CNOT gates as in Lemma 3.6.2. This allows us to extract w, a Hadamard, and v's phase (as in the example above). Denote (the now zero-phase) v as the new output in w's place. Go to Step 1.



(Step 4) Pivot about vw with Lemma 3.4.2. This converts w into an XY-vertex and introduces Hadamards on output wires. Go to Step 1.



(Step 5) Extract the permutation that remains as SWAP gates, and extract the Hadamard-edges as a row of Hadamard gates. Some of the XY-inputs may have a non-zero phase, which is also extracted:



This is a polynomial-time algorithm in $|V(G)|$ that turns the open graph into a circuit without introducing ancillae.

Proof. It is mostly clear that these rewrites do not change the meaning of the circuit, but there are a few assumptions in the algorithm that are worth elaborating on.

In Step 2, we claim that if there is no $\prec$-maximal-among-$\overline{O}$ XY-vertex v in $\overline{O}$ adjacent to an output, there must be one such YZ-vertex instead. We show this as follows. First, there are no XZ-vertices by Step 0. Second, note that any $\prec$-maximal vertex in $\overline{O}$ requires a correction set, which is only possible if it is adjacent to vertices in O. This means that we indeed have such a YZ-vertex if there is no such XY-vertex.

In Step 4, we claim the pivot introduces Hadamard gates at the outputs. This follows

from Proposition 2.2.16. We also implicitly claim we do *not* introduce Z-gates at the outputs. We get an additional π phase in the common neighborhood of v and w, but as Step 1 guaranteed w is not connected to other outputs, this phase is not added to any frontier vertex.

In Step 5, the result is guaranteed to be a permutation, as our map is a unitary, and Step 0 through Step 4 do not change this.

We will now check that we have gflow each time we reach Step 2, which is necessary if we want to use Lemma 3.6.1.

By Proposition 4.1.6, Step 0 maintains gflow. In Step 1, we change the connections between outputs, which maintains gflow by Lemma 3.4.9. We start Step 3 by applying CNOTs, which maintains gflow by Lemma 3.6.2. After this, we remove w as output and make v a new output instead, which maintains gflow by 3.4.8. In Step 4, we only do a pivot, which maintains gflow by 3.4.2. Finally, we cannot reach Step 2 from Step 5, so we do not need to consider it.

Finally, we consider efficiency. Every iteration of the form Step 1 $\rightarrow$ Step 2 $\rightarrow$ Step 3 $\rightarrow$ Step 1 removes one vertex from the diagram, without changing measurement planes. Each operation in this chain is also of polynomial-time. Every iteration of the form Step 1 $\rightarrow$ Step 2 $\rightarrow$ Step 4 $\rightarrow$ Step 1 turns one YZ-vertex into an XY-vertex. This does not change any other vertices, as output w does not have a measurement plane to modify. Each operation in this chain is also of polynomial-time. As such, if we have n vertices, we reach Step 5 within $2n$ of the above iterations, i.e. efficiently. Finally, Step 5 itself is also efficient. This means that as a whole, this algorithm is of polynomial-time.

Note that the theorem makes no claims about how many gates the result uses. In fact, when applying this to extraction algorithm to a diagram that was originally a circuit, the number of two-qubit gates can be *larger* than what we started with. This is especially the case because we have only discussed the most basic optimizations so far. (This increase is especially evident in Figure 6 of [19].) Putting diagrams into *reduced phase gadget form* [4], or doing *simulated annealing* [22] or even applying machine learning techniques [26], are examples of ways to reduce the number of two-qubit gates.

3.6.2. Optimized algorithm

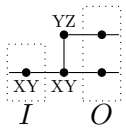
The basic algorithm uses gflow explicitly to justify the existence of a vertex v such that Step 3 executes correctly. This is because the odd neighborhood in (g3) gives us a lot of info. In Lemma 3.6.2, we add CNOTs based on this neighborhood. This corresponds to turning a row in the biadjacency matrix into a row with just a single 1. This frees up w and allows us to extract it, and put v in its place.

However, we do not need an explicit gflow to tell us how to make a row with a single 1. We can simply perform Gaussian elimination to get one, or even multiple such rows, and handle them at the same time. Not having to compute the gflow every iteration gives us much better performance.m To discuss this second algorithm in [4], we need one more lemma.

Lemma 3.6.4. Let (G, I, O, λ) be an open graph with gflow and $|I| = |O|$. Let $v \in O \setminus I$ be an output that is not an input. Finally, assume v has a unique neighbor $u \in \overline{O}$. Then $\lambda(u) = \mathrm{XY}$.

Proof. Suppose $\lambda(u) \neq \mathrm{XY}$. In this case, we can remove u from the graph by Lemma 3.4.5 without losing gflow. However, as u is the only connection to v, we now have a component with $|I|$ inputs and $|I| - 1$ outputs. This cannot have gflow by Proposition 3.2.11, which gives us a contradiction.

Particularly significant is the requirement that $|I| = |O|$. The optimization we will discuss does not work anymore when we consider isometries in Chapter 4, as this lemma does not hold for isometries with gflow. Consider for instance the following isometry with gflow, where the lemma fails for the YZ-vertex[2]:



With this lemma, we can now discuss the alternate, optimized algorithm.

Theorem 3.6.5. Consider the following modification of the algorithm in Theorem 3.6.3:

(Step 2') If the only remaining vertices are inputs, go to Step 5. Otherwise, perform a full Gaussian elimination on the biadjacency matrix between the frontier and the neighbors. If there is a row with just a single 1, go to Step 3'. Otherwise, go to Step 4'.

(Step 3') Implement the row operations the full Gaussian elimination did as CNOT gates, as discussed after Proposition 2.2.18. After this, every output vertex corresponding to a row with just a single 1 is connected to only one other non-output vertex. Extract all of these at the same time, just like we extracted a single vertex in Step 3.

(Step 4') We are in the situation of Step 4. Find a YZ-labeled vertex v connected to an output w and proceed as in Step 4.

Just like the unoptimized algorithm, this also is a polynomial-time algorithm that turns an open graph into a circuit without introducing ancillae.

In particular, if the open graph has q inputs and outputs, and n vertices, this algorithm has a runtime of $\mathcal{O}(q^2n^2 + q^3)$.

Proof. Suppose we reach Step 3'. All rows with a single 1 are XY-vertices by Lemma 3.6.4, and applying the CNOT gates maintains gflow.

If we instead reach Step 4', there is no possible XY-vertex to extract no matter what CNOTs we apply. As there are vertices connected to the outputs, one of these must be a YZ-vertex, and the proof of Step 4 holds here as well.

The most expensive operations in this algorithm are Gaussian elimination and pivots. Each Gaussian elimination is applied to a matrix with at most q rows and n columns,

[2]When the measurement angles do not matter, we will frequently write measurement fragments as a graph with the measurement planes attached, instead of a ZX-diagram, in order to reduce the clutter of the measurement effects.

> giving a runtime of $\mathcal{O}(q^2 n)$. For pivots, the neighborhoods we are toggling connectivity between are of size up to $\mathcal{O}(n)$, meaning up to $\mathcal{O}(n^2)$ edges change. Both of these can happen once per vertex, giving a total runtime of $\mathcal{O}(q^2 n^2 + n^3)$ for these steps.

This is indeed much faster than the basic algorithm. For each vertex, the basic algorithm has to find a gflow as in Proposition 3.2.10, which gives a runtime of at least $\mathcal{O}(n \cdot n^4)$ for just that part of the algorithm.

Note that the worst-case bound above is difficult to achieve. It needs both the Gaussian elimination and pivot operations to encounter their worst cases each step. This worst case only happens with many edges. However, pivots and CNOTs toggle the presence of edges, which makes it hard to have consistently "full" graphs.

4. Glack

This section contains our new contributions. While gflow is a binary "present/absent" property, we loosen it to *glack*: a measure of how far a graph is removed from having gflow. More specifically, glack counts the number of times we cannot give vertices a correction set. In Section 4.1 we will introduce this definition and discuss what happens to the operations we discussed in Section 3.4. In Section 4.2 we will show that under our scheme glack translates directly to the number of extra post-selected ancillae when extracting. This extraction procedure is a simple generalization of the unitary gflow case. It relies on *certificates*, which we will say a few words about in Section 4.3.

Finally, Section 4.4 will discuss an application of this measure in the form of the *spider nest identities*. In certain situations, we can choose whether we want a higher T-count, or a lower T-count but more post-selections. We conclude with an illustrative example of the controlled-Toffoli gate. While this is not the most meaningful trade-off (for NISQ machines, neither is much of a problem, while beyond the NISQ-era, both might be undesirable), it does showcase how glack allows for new analyses.

4.1. Definition and properties

The main restriction gflow has is that *every* non-output vertex needs a correction set. In some cases, this is not feasible. To that end, we can consider what happens when we simply allow vertices to remain uncorrected.

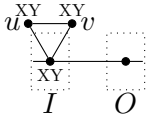
Definition 4.1.1 (Glack). Let (G, I, O, λ) be an open graph. Then G has a *glack* of k if:

- There exists a *lacking* set $L \subseteq \overline{O}$ of size k, a map $g : \overline{O} \setminus L \to \mathcal{P}\left(\overline{I}\right)$, and a strict partial order $\prec$ on $V(G)$, such that the gflow conditions (g0) through (g5) hold for all v in g's domain.
- There does not exist such $(L', g', \prec')$ with $|L'| < k$.

We will call such a tuple $(L, g, \prec)$ a *certificate* of the glack. Vertices in L are *lacking* vertices.

A vertex u is a *completely lacking* vertex if for every certificate $(L, g, \prec)$, we have that u is not $\prec$-minimal and $u \in L$. We will denote the set of completely lacking vertices on a graph by $\mathcal{L}$.

If the glack is k, there will be at least k vertices that cannot be corrected. For $k = 0$, this is simply the same as gflow. However, higher values of glack need analysis. An example of a lacking graph is the following:



Here, we do not have gflow. However, letting $L = \{u\}$, defining $g(I) = \{O\}$, $g(v) = \{u\}$, and taking $v \prec u, I, O$ and $I \prec O$, we get a valid certificate for a glack of 1. This graph is symmetric in u and v, so we can also swap their roles and obtain a certificate with $L = \{v\}$. This means that for this graph, $\mathcal{L}$ is empty. With $\lambda(u) = \lambda(v) \in \{\mathrm{XZ}, \mathrm{YZ}\}$, we get the same results, albeit with different correction sets.

Because we need to compute flows, lacking vertices are difficult to detect in arbitrary graphs. Completely lacking vertices even more so. This is despite the fact that removing lacking vertices can increase glack, which we will see in Lemmata 4.1.7 and 4.1.14. This is bad, as higher glack will result in more expensive circuits, which we will see when we start extracting circuits with Theorem 4.2.3. Lemma 4.1.14 will also explain the somewhat unmotivated definition of completely lacking vertices, as $\mathcal{L}$ will be the set of vertices that is "unsafe" to remove.

We will need one somewhat technical tool. It will help limit the above-mentioned increase in glack when removing vertices, but it will also be useful for other purposes.

Lemma 4.1.2. Let (G, I, O, λ) be an open graph with a glack certificate $(L, g, \prec)$, and let $u \in V(G)$. Then there exists a g' such that:

- The graph is also certified by $(L, g', \prec)$;
- There exists at most one vertex v_a with $u \in g'(v_a)$;
- Other than v_a, there exists at most one vertex v_b with $u \in \mathrm{Odd}(g'(v_b))$.

Proof. We will transform our certificate in steps to reduce these dependencies. We will first start reducing the number of vertices that contain u directly in their correction set. To that end, consider v_1 and v_2 with $u \in g(v_1)$ and $u \in g(v_2)$.

- If WLOG $v_1 \prec v_2$, then also $v_1 \prec g(v_2)$ and $v_1 \prec \mathrm{Odd}(g(v_2))$ by (g1) and (g2). In particular, $v_1 \notin g(v_2) \cup \mathrm{Odd}(g(v_2))$. This means we can safely update $g(v_1) \leftarrow g(v_1) \,\Delta\, g(v_2)$ without affecting (g3) through (g5). We also maintain (g1) and (g2), as $g(v_1)$ and $\mathrm{Odd}(g(v_1))$ only get changed by vertices in the future of v_1.
- Otherwise, v_1 and v_2 are $\prec$-unrelated. Now consider $w \in g(v_2) \cup \mathrm{Odd}(g(v_2))$. As $v_2 \prec w$, we do not have $w \prec v_1$, as otherwise v_1 and v_2 would be $\prec$-related. In particular, $w \neq v_1$. This means we may put all w into v_1's future and safely update $g(v_1) \leftarrow g(v_1) \,\Delta\, g(v_2)$ in this case as well.

We can repeat this until at most one vertex v_a contains u in its correction set.

We will now reduce the number of vertices v with $u \in \mathrm{Odd}(g(v))$. Among the vertices in $S = \{v \neq v_a \mid u \in \mathrm{Odd}(g(v))\}$, pick v_b to be a $\prec$-maximal. Note that $u \notin g(v_b)$. For each $v \in S \setminus \{v_b\}$, update $g(v) \leftarrow g(v) \,\Delta\, g(v_b)$. Similar to before, whether v_b and v are $\prec$-related or not, $g(v)$ is a valid correction set. Importantly, after this update, $u \notin \mathrm{Odd}(g(v))$, and we still have $u \notin g(v)$. As such, after doing this for all vertices in $S \setminus \{v_b\}$, there still is at most one v_a with u in its correction set, and the only vertex in S with u in its odd neighborhood is v_b.

If we now call g' our updated g, we get a certificate as in the statement, as we did not change lacking sets, and the partial order is compatible with our changes.

4.1.1. Glack and operations that do not remove vertices

Just like we discussed what operations maintain (or do not maintain) gflow, we will spend some time looking at how various operations interact with glack. Many of the operations we care about behave well even without gflow. However, some crucial operations are much less well-behaved.

This section and the next consist of a long list of statements about how glack and certificates change when doing certain graph operations. These statements are written to be as specific as possible, and often go into much more detail than just "the glack value changes by this much". This is because the changes to the certificate are also relevant. The following table is a summary of the statements.

Proposition 4.1.3 (Common graph operations). Let (G, I, O, λ) be an open graph with a glack certificate $(L, g, \prec)$. We have the following behavior for the respective operations:

Operation	Glack	Maintained	Statement
Local complementation	The same	L, $\prec$	Lem. 4.1.4
Pivot	The same	L, $\prec$	Lem. 4.1.5
Turn into phase gadget form	The same	L, $\prec$	Prop. 4.1.6
Prepend input (to input)	The same	L, g, $\prec$	Lem. 4.1.8
Append output (to output)	The same	L, g, $\prec$	Lem. 4.1.9
Append output (to lacking)	The same or better	L, g, $\prec$	Lem. 4.1.9
Add post-selected ancilla	The same	L, g, $\prec$	Lem. 4.1.10
Connect outputs/lacking	The same	L, g, $\prec$	Lem. 4.1.11
Add graphs	Summed	See statement	Lem. 4.1.12
Concatenate graphs	Summed or better	See statement	Lem. 4.1.13
Removal (any)	At most 1 worse		Lem. 4.1.7
Removal (under conditions)	The same or better	L, $\prec$	Lem. 4.1.14
Append CNOT gates	The same	L, $\prec$	Lem. 4.1.15
Gadget merge (XY-neighbors)	The same	L, $\prec$	Lem. 4.1.16
Gadget merge (any neighbors)	The same or better	L, $\prec$	Lem. 4.1.16
State copy (to one XY)	The same	L, $\prec$	Lem. 4.1.17
State copy (to one any)	The same or better	L, $\prec$	Lem. 4.1.17

If the glack stays the same, the "Maintained" column describes what part of existing certificates is maintained for already existing vertices (although certificates' flow and partial order may be extended). If the glack changes, this column is inapplicable, as there is an entirely new set of certificates for the better glack.

If an operation maintains glack exactly, the same holds for the inverse operation.

The goal of this list is two-fold. First, we need some of these statements to justify that extraction works with glack instead of just gflow. But we also want to have as many of the statements that hold for gflow, to also hold for glack. This way, optimization schemes for gflow translate into optimization schemes for glack.

We can consider lacking vertices and output vertices the same in many proofs, as they share two crucial properties. Lacking vertices and output vertices both do not have a correction set. Lacking vertices and output vertices can both be taken to be $\prec$-maximal, as there is no $g(u)$ or $\mathrm{Odd}(g(u))$ to put in the future for any $u \in L$. Because of this, some proofs about gflow remain the same for glack. We will include one proof as an

example of this; the other proofs from this section can be found in Appendix B.

Lemma 4.1.4 (Glack ver. of Lem. 3.4.1). Let (G, I, O, λ) be an open graph with a glack certificate $(L, g, \prec)$, and let $u \in \overline{I}$ not be an input. Then there is some g' such that $(L, g', \prec)$ certifies $(G * u, I, O, \lambda')$.

(Here, λ' is as in Lemma 3.4.1.)

Proof. If $u \notin O \cup L$, define updated flow g' as follows:

$$g'(u) = \begin{cases} g(u) \,\Delta\, \{u\} & \text{if } \lambda(u) \in \{\mathrm{XY}, \mathrm{XZ}\} \\ g(u) & \text{otherwise,} \end{cases}$$

and for $v \in \overline{O \cup L} \setminus \{u\}$:

$$g'(v) = \begin{cases} g(v) \,\Delta\, g'(u) \,\Delta\, \{u\} & \text{if } u \in \mathrm{Odd}_G(g(v)) \\ g(v) & \text{otherwise.} \end{cases}$$

Otherwise, if $u \in O \cup L$, define updated flow g' for $v \in \overline{O \cup L}$ by

$$g'(v) = \begin{cases} g(v) \,\Delta\, \{u\} & \text{if } u \in \mathrm{Odd}_G(g(v)) \\ g(v) & \text{otherwise,} \end{cases}$$

In both cases, the proof of Lemma 3.4.1 proceeds exactly the same for all non-lacking vertices. This gives a candidate certificate $(L, g', \prec)$ for $(G*u, I, O, \lambda')$, where we keep all lacking vertices of G lacking in $G*u$. This means that glack will not increase under local complementations.

But glack cannot decrease under local complementations either. Suppose G has a glack of k and $G * u$ has a glack of $\ell < k$. As graphically $(G * u) * u = G$, this result must also have a glack of at most ℓ, contradicting the minimality of k. This means that our candidate certificates certify the actual glack, and we are done.

The important part of this lemma's statement is that glack is maintained under local complementations, and we can trace exactly what happens to lacking vertices. The same then also holds for pivoting, of course.

Lemma 4.1.5 (Glack ver. of Lem. 3.4.2). Let (G, I, O, λ) be an open graph with a glack certificate $(L, g, \prec)$, and let $u, v \in \overline{I}$ be connected. Then there exists some g' such that $(L, g', \prec)$ certifies $(G \wedge uv, I, O, \lambda')$.

(Here, λ' is as in Lemma 3.4.2.)

Just as in the gflow case, this is enough to eliminate XZ-vertices and turn the graph into phase gadget form.

Proposition 4.1.6 (Glack ver. of Prop. 3.4.4). Let (G, I, O, λ) be an open graph with a glack certificate $(L, g, \prec)$. This graph can be put into phase gadget form efficiently, and there exists a g' such that $(L, g', \prec)$ certifies the result.

The next operation we considered after local complementations and pivoting in Section 3.4 is deletion of certain vertices. Unlike the gflow case, this is much more tricky if we do not want to increase glack. As such, we will postpone it and all statements depending

on it until the next section. However, when allowing for increased glack, the following bound is easily shown with Lemma 4.1.2.

Lemma 4.1.7. Let (G, I, O, λ) be an open graph with a glack of k, and let u be *any* vertex. Then $(G \setminus \{u\}, I \setminus \{u\}, O \setminus \{u\}, \lambda)$ has a glack of at most $k+1$.

We now turn to the generalizations of what happens when appending vertices to inputs and outputs. In light of Proposition 3.2.11, lacking vertices play a special role here. If you have a unitary and add an input, we must end up with a non-zero glack. Conversely, adding a well-connected output to a lacking graph may reduce glack.

Lemma 4.1.8 (Glack ver. of Lem. 3.4.7). Let (G, I, O, λ) be an open graph with a glack certificate $(L, g, \prec)$. Let G' be the result of connecting any $i \in I$ to a new input vertex measured in the XY-plane in its stead. Then there are extensions g' and $\prec'$ of g and $\prec$ respectively such that $(L, g', \prec')$ certifies G'.

Lemma 4.1.9 (Glack ver. of Lem. 3.4.8). Let (G, I, O, λ) be an open graph with a glack of k with certificate $(L, g, \prec)$.

- Let G' be the result of appending to any $o \in O$ a vertex measured in the XY-plane and connecting it to a new output in its stead. Then there are extensions g' and $\prec'$ of g and $\prec$ respectively such that $(L, g', \prec')$ certifies G'.
- Let G'' be the result of appending to any $\ell \in L$ a new output vertex. If $\lambda(\ell) \neq$ XY, G'' has a glack of either $k-1$ or k. Otherwise, it is exactly $k-1$. If the resulting glack is exactly k, there are extensions g'' and $\prec''$ of g and $\prec$ respectively such that $(L, g'', \prec'')$ certifies G''.

Note that these bounds are tight, and necessary. Consider the following two graphs:

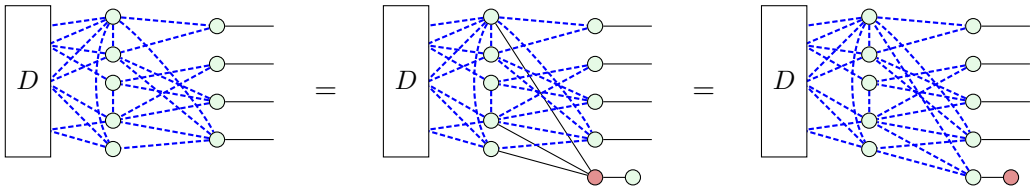
Both graphs have a glack of 1, but behave differently when connecting a YZ-vertex to a new output. The new output allows the left graph to get gflow, but the right graph remains at a glack of 1.

Lemma 4.1.10. Let (G, I, O, λ) be an open graph with a glack of k with certificate $(L, g, \prec)$, and let $A \subseteq \overline{O}$. Define G' to be the same as G except:

- We add an additional output o to the graph. Beyond the graph, we post-select this new output as $\langle 0|$.
- We add edges (o, a) for each $a \in A$.

Then (G, I, O, λ) and $(G', I, O \cup \{o\}, \lambda)$ with the post-selection represent the same map, and G' has a glack of either $k-1$ or k. If the resulting glack is exactly k, $(L, g, \prec)$ certifies it.

What we are doing can be derived in ZX as nothing but the π-copy rule and a color change to introduce this post-selected output:



In this example, A consists of three arbitrary vertices in the middle column, and D is an arbitrary diagram. The post-selection introduced by this lemma always has some chance to succeed, as the rule does not reduce the diagram to the scalar 0. Note that applying this lemma to a set $A = \{\ell\}$ consisting of a single lacking vertex reduces to the second bullet of Lemma 4.1.9.

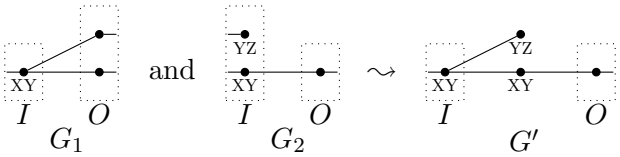
Lemma 4.1.11 (Glack ver. of Lem. 3.4.9). Let (G, I, O, λ) be an open graph with a glack certificate $(L, g, \prec)$. Then adding or removing edges between vertices in $O \cup L$ maintains this certificate.

Lemma 4.1.12 (Glack ver. of Lem. 3.4.13). Let $(G_1, I_1, O_1, \lambda_1)$ and $(G_2, I_2, O_2, \lambda_2)$ be open graphs certified by $(L_1, g_1, \prec_1)$ and $(L_2, g_2, \prec_2)$ respectively, with $V(G_1)$ and $V(G_2)$ disjoint. Then $(L_1 \cup L_2, g_1 \cup g_2, \prec_1 \cup \prec_2)$ certifies the combined graph.

Lemma 4.1.13 (Glack ver. of Lem. 3.4.14). Let $(G_1, I_1, O_1, \lambda_1)$ and $(G_2, I_2, O_2, \lambda_2)$ be open graphs certified by $(L_1, g_1, \prec_1)$ and $(L_2, g_2, \prec_2)$ respectively, with $V(G_1)$ and $V(G_2)$ disjoint, and $|O_1| = |I_2|$. Then concatenating the two graphs by identifying the vertices of O_1 with I_2 results in a graph with a glack of at most $k_1 + k_2$.

If the resulting glack is exactly $k_1 + k_2$, then $(L_1 \cup L_2, g_1 \cup g_2, \prec_1 \cup \prec_2)$ is a certificate of the concatenated graph, taking into account the identification.

This lemma can result in a lower glack than the two separate graphs together. Consider the following two graphs G_1, G_2, and their composition G':



Here, G_1 has gflow, and G_2 has a glack of 1 (as the YZ-vertex has no correction set). Composing them yields G', which has gflow again. This means that the "at most" in the above statement is necessary. This also means that as the inverse operation, "splitting" a graph into two pieces may increase glack.

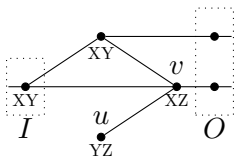
4.1.2. Glack and operations that remove vertices

As stated in the previous section, operations that have to do with vertex deletions are more complicated to analyze if we do not want to increase glack. Once again, the proofs of these statements can be found in Appendix B.

Lemma 4.1.14 (Glack ver. of Lem. 3.4.5). Let (G, I, O, λ) be an open graph with a glack of k with certificate $(L, g, \prec)$, and let $u \in \overline{O \cup \mathcal{L}}$ with $\lambda(u) \neq \mathrm{XY}$. Consider $(G \setminus \{u\}, I \setminus \{u\}, O, \lambda)$.

- If we can choose the certificate so that $u \in L$ is $\prec$-minimal, the glack is at most $k-1$. If the resulting glack is exactly $k-1$, there exists some g' such that $(L \setminus \{u\}, g', \prec)$ certifies it.
- Otherwise, the glack is at most k. If the resulting glack is exactly k, there exists some g' such that $(L, g', \prec)$ certifies it.
- The above bounds are tight unless for some certificate $(L, g, \prec)$, there exists a vertex $v \in L$ with $u \prec v$ such that the following holds: there exists some set $g(v) \in \overline{I}$ that satisfies the gflow conditions (g0) through (g5), *except* that we ignore (g2) for $u \in \mathrm{Odd}_G(g(v))$.

Another way to phrase this extra condition is the following: we have a valid correction set for v as long as we ignore the dependency $u \prec v$. The only reason it is not a valid correction set, is because we get a cyclic dependency $u \prec v$ and $v \prec u$. Unfortunately, this extra condition for tightness is necessary. Consider the following graph:



It can be shown that this graph has a glack of 1, but removing u results in a graph with gflow. However, there is no certificate in which u is both lacking and minimal, so we are in the case of the second bullet point, and the reduction in glack is because the third bullet point holds. The vertices u and v fail this: consider a certificate in which v is lacking, and $g(u) = \{u\}$, so that $u \prec v$. If we ignore this relation, $g(v) = \{u, v\}$ with $v \prec u$ would be a perfectly valid correction set, but unfortunately, we already have $u \prec v$.

Also note that removing non-XY input vertices is covered by the first bullet point. In light of Proposition 3.2.11, it makes sense that removing extra inputs allows gflow to exist more easily. (A more precise version of Proposition 3.2.11 is Lemma 4.2.7 that we will discuss later, which tells us removing any input reduces glack by at least 1, which agrees with this.)

If we wish to remove a vertex not covered by this statement, we have to fall back to Lemma 4.1.7. Unfortunately, while XY-plane vertices are easy to detect, completely lacking vertices are nearly undetectable, making it dangerous to remove a vertex where you do not have a certificate in which it is not lacking.

We can only track what happens to L if the tightness condition holds. Otherwise, our certificate no longer certifies the current glack, making it a lot less useful. To still keep track of the lacking vertices if the tightness condition fails, you need to explicitly find the certificate that causes the condition to fail. This is expensive, and it is not even clear *when* this condition fails. This is troublesome if you want to keep a certificate on hand every step of the way.

The inferior certificate is not *completely* useless, however. You can keep it on hand as a proof of a glack of *at most* k. When glack increases again, perhaps beyond k, it may become a proper certificate again.

It is perhaps surprising to see here, but the proof showing appending a CNOT maintains semantics involves a deletion, so we can only discuss it now.

Lemma 4.1.15 (Glack ver. of Lem. 3.4.10). Let (G, I, O, λ) be an open graph with a glack certificate $(L, g, \prec)$. Then there exists a flow g' such that the changes as described in Lemma 3.4.10 are certified by $(L, g', \prec)$.

Finally, we move on to two operations involving phase gadgets.

Lemma 4.1.16 (Glack ver. of Lem. 3.4.11). Let (G, I, O, λ) be an open graph with a glack of k certified by $(L, g, \prec)$, and let $u \neq v$ be interior vertices measured in the YZ-plane with $N(u) = N(v)$. Then $G \setminus \{u\}$ has a glack of at most k.

If the resulting glack is exactly k, there exists a g' such that $(L, g', \prec)$ certifies it.

Lemma 4.1.17. Let (G, I, O, λ) be an open graph with a glack of k. Suppose we apply the state copy rule, Lemma 3.4.12, to a YZ-vertex $u \in \overline{O}$ with one neighbor v. Then $G \setminus \{u\}$ has a glack of at most k. If $\lambda(v) = \mathrm{XY}$, it will be exactly k.

If the resulting glack is exactly k, there exists a g' such that $(L, g', \prec)$ certifies it.

This lemma in particular is much more useful in phase gadget form, as all neighbors of YZ-vertices will be XY-vertices by default.

4.2. Circuit extraction for arbitrary graphs

Theorem 3.6.3 and its optimized version can only extract ZX-diagrams that represent unitary maps, and have gflow. Having gflow is necessary if you do not want to introduce ancillae and have an efficient extraction procedure [4]. However, failing this, we can still extract by adding more qubits. In this section, we will show how the basic extraction procedure we discussed in Section 3.6 can be extended to handle arbitrary graphs.

We will start with a short informal summary of the algorithm in Theorem 3.6.3:

(Step 0) Convert the graph into phase gadget form.

(Step 1) Unfuse connected output vertices.

(Step 2) Find either a "nice" non-input XY-vertex near the frontier, or get a non-input YZ-vertex.

(Step 3) If Step 2 found an XY-vertex, use it to extract, and go back to Step 1.

(Step 4) Otherwise, pivot to introduce a "nice" XY-vertex, and go back to Step 1.

(Step 5) Once we only have inputs and outputs, untangle the remaining permutation.

When allowing lacking vertices and non-unitary maps, two steps need significant updates. Step 2 can run into the situation where neither "nice" XY- nor YZ-vertices exist, due to lacking vertices. Step 5 can encounter an arbitrary bipartite graph instead of just a permutation.

We will handle lacking vertices in Step 2 by applying Lemma 4.1.10 to turn a lacking vertex into a "nice" vertex that can be extracted, at the cost of an extra ancilla.

For Step 5, we need a more in-depth discussion. If we want to extract the bipartite graph to a circuit, the easiest way to do that is reducing it back into something that

resembles a permutation. We do this in two steps: first, we remove all cycles, and after, we shrink all paths down to one edge.

Lemma 4.2.1. Let (G, I, O, λ) be a bipartite open graph with parts I and O, and let $C = (v_1, \dots, v_k, v_1)$ be a cycle in G where $v_1 \in I$. Then we can break up C by turning v_1 into a post-selected ancilla and applying CZ-gates to certain outputs, maintaining the bipartite structure.

Proof. Breaking up the cycle consists of the following steps:

(i) First, consider v_1 as part of O as well, with this new output post-selected as $\langle +|$.

(ii) Next, as G was bipartite, each neighbor w of v_1 is in O. This justifies extracting the edges $\{v_1, w\}$ as CZ-gates. In particular, edges $\{v_1, v_2\}$ and $\{v_1, v_k\}$ get removed from G, breaking up the cycle.

(iii) Finally, we want to return to a bipartite structure. As $v_1 \in I$, we can prepend two Hadamard gates. One of the Hadamard gates is a gate in the extracted circuit, while the other is used to insert a new vertex in I in place of v_1. This new edge does not introduce a cycle, and I and O have become disjointed again.

We now have an additional frontier: instead of extracting only after the outputs, now we also put gates before the inputs.

Lemma 4.2.2. Let (G, I, O, λ) be a bipartite open graph with parts I and O with no cycles. Then by applying certain CNOT gates on either side, G can be transformed into an open graph such that no vertex has more than one neighbor.

Proof. For $\ell > 2$, let $P = (v_1, v_2, \dots, v_\ell)$ be a maximally extended path in G. We will assume $v_1 \in O$ and extract CNOT gates beyond the outputs, but the approach works if $v_1 \in I$, prepending CNOT gates before the inputs instead.

As our graph is bipartite, the vertices of the path alternate between I and O. In particular $v_3 \in O$. This means that we can apply a CNOT gate with the control qubit at v_3's and the target qubit at v_1's. By Proposition 2.2.18, we update $N(v_3)$ to $N(v_3) \,\Delta\, N(v_1)$. By our path being maximally extended, $N(v_1) = \{v_2\}$, so the update comes down to removing v_2 from $N(v_3)$, and keeping the rest of the graph unchanged. This means that our path split into (v_1, v_2), and if $\ell > 3$, $(v_3, \dots, v_\ell)$.

By repeatedly applying the above, we slowly whittle away longer stretches of edges, until there no longer is such a path $(v_1, v_2, \dots, v_\ell)$ for $\ell > 2$. In other words, we end up with a graph where no vertex has more than one neighbor.

Note that the above process can introduce vertices connected to nothing at all. This is then either a post-selected output before we even start applying our circuit, or a prepared qubit.

With these extra ingredients, we can now turn to extracting arbitrary graphs. At this point, it will be difficult to bound the number of introduced ancillae: graphs with a glack of k give our scheme a completely useless bound of "*at least* k ancillae". However, after this we will be able to use certificates to improve it to "*exactly* k ancillae".

For clarity, we refer to the steps of this algorithm as "(i)" or "Step (i)" with parentheses, while we refer to the steps of the algorithm in Theorem 3.6.3 as "Step i", without parentheses.

Theorem 4.2.3 (General extraction). Let (G, I, O, λ) be a graph with a glack of k and no components not connected to I or O. Consider the following algorithm.

(0) Apply Step 0 of the original algorithm: Convert the graph into phase gadget form. Additionally, ensure all inputs are XY-vertices by applying Lemma 4.1.8 to non-XY inputs, and extracting a Hadamard before those inputs.

(1) Exactly Step 1 of the original algorithm: Unfuse connected output vertices as CZ-gates in the extracted circuit.

(2) Pick a certificate $(L, g, \prec)$ and vertex $v \in \overline{O}$ (if they exist) such that the following holds: among vertices in $\overline{O}$, there is a $\prec$-maximal vertex v connected to an output vertex w.

  (i) If the only remaining vertices are inputs, go to (5).

  (ii) If $\lambda(v) = \mathrm{XY}$, go to (3).

  (iii) If no such XY-vertex exists, but $\lambda(v) = \mathrm{YZ}$, go to (4).

  (iv) If no such vertex and certificate exists at all, go to $(*)$.

(3) Apply CNOT gates to extract w as in Step 3 of the original algorithm. If, afterwards, v is not connected to any internal vertex, extract it as a $|+\rangle$ preparation. Go back to (1).

(4) Exactly Step 4 of the original algorithm: Pivot about vw to create a maximal XY-vertex, and add the resulting gates to the extracted circuit. Go back to (1).

$(*)$ Choose a frontier vertex $f \in O$. Apply Lemma 4.1.10 to introduce an ancilla that has the same neighborhood as f, except for one vertex. This allows us to extract f by taking v to be the different vertex, $w = f$, after which we go to (3).

(5) Apply Lemmata 4.2.1 and then 4.2.2 to ensure our graph is a permutation up to some vertices not connected to anything. Extract the vertices without neighbors as either a post-selection or preparation, and handle the rest as in Step 5 of the original algorithm.

This is a polynomial-time algorithm that turns the open graph into a circuit on $\max(|I|, |O|)$ qubits with at least k additional post-selections. This circuit represents the open graph up to a non-zero scalar.

Proof. Step (0) does not change glack by Proposition 4.1.6 and Lemma 4.1.8. Semantics are also maintained when applying the latter, as we also extract an extra Hadamard, canceling out to the identity. Step (1) also does not change glack by Lemma 4.1.11.

In Step (2), we either have gflow when restricting our view to $O \cup N(O)$, or one of the output vertices is adjacent to a lacking vertex. This gives the extra possibility of going to $(*)$ as compared to Step 2 of the original algorithm. In particular, determining which of these four cases applies can be done efficiently, by running any polynomial-time algorithm for finding gflow or reporting none exist (such as Algorithm 1 in [4]) on $O \cup N(O)$.

In the original algorithm, Step 3 consisted of applying CNOTs. This means that Step (3) maintains glack by Lemma 4.1.15. Removing an output v not connected to anything also does not affect glack.

Step (4) mirrors Step 4 of the original algorithm. This step maintains glack by Lemma

4.1.5.

In $(*)$, we could in particular not apply Step (3). This means that the biadjacency matrix between O and $N(O)$ does not have the property that there exists some sum of rows that add up to a row consisting of just a single 1 (as we did in Lemma 3.6.2). In this case, doing as $(*)$ prescribes fixes this issue; the additional vertex corresponds to adding a row in the biadjacency matrix that differs from another row in just one index. This allows us to extract a vertex as in Step (3). Note that this reduces the glack by *at most* one by Lemma 4.1.10.

One can check the operations in Lemmata 4.2.1 and 4.2.2 can be implemented efficiently, resulting in Step (5) being efficient as well. By the assumption that no components don't intersect $I \cup O$, after Step (5), no spiders remain.

The correctness and efficiency analysis of Theorem 3.6.3 is mostly applicable here as well. Note that with a glack of k, if we visit Step $(*)$ at least $k - \ell$ times for some ℓ, the extracted output has at least $k - \ell$ ancillae, and the graph going into Step (5) has a glack of at least ℓ. At Step (5), the only structure that can introduce glack are cycles. (As the inputs are XY-vertices, we cannot get glack from (g0)-violations.) These at least ℓ cycles then get removed by at least ℓ applications of Lemma 4.2.1, each application introducing an ancilla. As such, the result has at least k ancillae, as stated.

We now need to show that introducing Step $(*)$ does not allow for an infinite loop, as it *adds* an XY-vertex to the graph, and the original algorithm relied on every step reducing the lexicographic order (#unextracted vertices, #unextracted YZ-vertices). However, this algorithm reduces the lexicographic order (glack in unextracted parts, #unextracted vertices, #unextracted YZ-vertices) each time we visit Step (2). This means that we do not enter an infinite loop, and instead run efficiently.

Note that when the graph we are extracting is not a unitary, the scalar suppressed in all ZX-rewrites actually becomes relevant, and should not be ignored, as we did so far. Most scalar components also contribute at least 1 glack even if they are irrelevant to extraction. However, as we will concern ourselves with diagrams originating from Proposition 3.3.4, we will be working with isometries, and not encounter these problems. We will now move on to extracting these isometries.

Corollary 4.2.4 (Gflow isometry extraction). Let (G, I, O, λ) represent an isometry with gflow. Then applying the algorithm described in Theorem 4.2.3 does not introduce any extra ancillae, and as a result, gives a polynomial-time algorithm that turns the open graph into a circuit on $|O|$ qubits.

Proof. Extra ancillae are only introduced in contexts with lacking vertices in the proof of Theorem 4.2.3. As we have gflow, these do not exist. In particular, we never visit Step $(*)$, and in Step (5), we apply Lemma 4.2.1 zero times.

Occasionally we saw output vertices and lacking vertices as the same thing. Now we will do this explicitly to improve the "at least k" bound to "exactly k".

Definition 4.2.5 (Output form). Let (G, I, O, λ) be an open graph in phase gadget form with a glack of k and certificate $(L, g, \prec)$. We create the *output form*, $(G, I, O \cup L, \lambda)$ by turning lacking vertices $\ell \in L$ into outputs. To maintain semantics, these outputs are post-selected as determined by the measurement plane and angles:

- If $\lambda(\ell) = \mathrm{XY}$, post-select ℓ as $|\pm_{\mathrm{XY},\ell}\rangle$ with outcome 0;
- If $\lambda(\ell) = \mathrm{YZ}$, post-select ℓ as $|\pm_{\mathrm{YZ},\ell}\rangle$ with outcome 0.

Lemma 4.2.6. Let (G, I, O, λ) be an open graph with a glack of k. Then its output form has gflow.

Proof. All vertices in $\overline{O \cup L}$ in the original graph have a correction set, which is enough to give the output form gflow.

Lemma 4.2.7. Let (G, I, O, λ) be an open graph with $|I| \geq |O|$. Then G has a glack of at least $|I| - |O|$.

Proof. Suppose G has a glack of k. The output form of G then has $|I|$ inputs, $|O| + k$ outputs, and by the previous lemma, gflow. By Proposition 3.2.11, the output form can only have gflow if $|I| \leq |O| + k$. As such, $k \geq |I| - |O|$.

Corollary 4.2.8. The output form of any open graph represents an isometry.

Theorem 4.2.9 (Certificate extraction). Let (G, I, O, λ) be an open graph with a glack of k, certified by $(L, g, \prec)$. Consider the following algorithm.

(1) Convert the graph into phase gadget form with Proposition 4.1.6.

(2) Convert the graph into output form.

(3) Apply Corollary 4.2.4 to the above graph.

This is a polynomial-time algorithm that converts a graph into a circuit on $|O| + k$ qubits, of which k are post-selected.

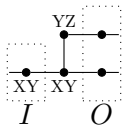
Proof. By Lemma 4.2.8, the output form is an isometry, which Corollary 4.2.4 can then extract. All three steps are efficient.

The downside of this approach is that the post-selections cannot be corrected for in the circuit with just measurement-output-conditional X and Z gates, as otherwise these vertices would have had gflow. In some cases, there may be more complex corrections possible. For instance, in the *ZH-calculus* [3], an $a\pi$ spider can commute with a certain structure of this ZH-calculus into a conditional CZ-gate, which is genuinely different from the behavior we saw in the proof of Proposition 3.2.9, where we only commuted into other $a\pi$ spiders. These more complex corrections will not be considered in this thesis.

Another issue is that we need a certificate ready. For arbitrary graphs, this is difficult, which we will touch upon in Section 4.3. However, when starting with a circuit and applying rewrites as in Section 4.1, you will usually have a good idea where the lacking vertices are.

Note that the approach of "rewrite into an isometry and extract that" also works for the optimized algorithm of Theorem 3.6.5, mostly. There is only one issue: the optimized

algorithm depends on Lemma 3.6.4. This lemma states that for unitaries, if an output is connected to only a single neighbor, that neighbor is an XY-vertex. Unfortunately, this does not hold for arbitrary isometries, even with gflow.



Despite having gflow, the above figure has an output that is only connected to a YZ-vertex. The workaround for this is simple. Instead of Step 3' extracting *every* vertex, we limit ourselves to XY-vertices (which are easily detected). In the case that Step 3' leaves us with *only* YZ-vertices, we instead move on to Step 4'.

This gives an optimized algorithm for extracting isometries with gflow. Just like for the unoptimized algorithm, we can turn lacking graphs into output form so that there is gflow, and extract the graph with it. This gives the same k ancillae as the unoptimized algorithm.

However, just like the unoptimized algorithm, we need a certificate to be able to convert the graph into output form, which is still a problem.

4.2.1. Optimizing graphs with glack

The motivation for the pipeline "circuit $\mapsto$ ZX-diagram $\mapsto$ circuit" is optimizing quantum circuits. We have not talked much about this yet, with only brief mentions so far. When optimizing unitary diagrams with gflow, two main optimization targets are the two-qubit gate count, and the T-count.

As mentioned before, there exist various techniques to reduce the two-qubit gate count, such as putting diagrams into *reduced phase gadget form*, or doing *simulated annealing* to improve that result further. The simulated annealing-approaches generally only apply local complementations, which maintain glack by Lemma 4.1.4. This means that applying those methods to graphs with glack poses no problem.

However, the more fundamental reduced phase gadget form is more difficult to implement. Applying this optimization not only requires local complementations and pivots, but also removes vertices and applies gadget optimizations. While merging gadgets or removing one-legged YZ-vertices does not increase glack by Lemmata 4.1.16 and 4.1.17, we cannot always safely remove non-XY vertices. After all, Lemma 4.1.14 additionally requires the deleted vertex to not be in $\mathcal{L}$, which is difficult to check for. If the vertex *is* in $\mathcal{L}$, we fall back to Lemma 4.1.7, and we will increase glack.

Putting graphs into reduced phase gadget form is also how [21] does *phase teleportation*. Phase teleportation is a technique that determines how phases can be moved across vertices, non-locally. This can reduce the T-count without changing the structure of the graph. In this case, accidentally deleting a vertex in $\mathcal{L}$ and increasing glack is not a problem, because we discard most of the resulting graph, and only use the phase data.

One solution to Lemma 4.1.7 is to put the graph in output form. This is a graph with gflow, so all of the existing optimization literature can be applied. However, while this ensures we do not increase glack, we also cannot *decrease* glack, as our lacking vertices have become outputs, and outputs are not affected by any optimization. In particular, the glack-decreasing mechanisms of Lemma 4.1.14 will never apply.

4.3. Finding certificates

We have two main extraction algorithms for a graph with a glack of k. The algorithm that introduces *at least* k extra post-selections does not require a certificate, but the algorithm that introduces *exactly* k does. We need some way to find this certificate.

Proposition 4.3.1. Let (G, I, O, λ) be an open graph on n vertices with a glack of k. Then there exists an algorithm with a complexity of $\mathcal{O}(n^{4+k})$ that finds a certificate of this glack. In addition, if this algorithm is applied to a graph with larger glack, it will output that there exists no certificate.

Proof. Consider Theorem C.6 in [4], which provides an algorithm with a runtime of $\mathcal{O}(n^4)$ that decides whether a graph has gflow, and if so, outputs it.

We will choose k arbitrary vertices $L := \{v_1, \ldots, v_k\}$ in this graph as the potential lacking vertices, and consider $(G, I, O \cup L, \lambda)$. We require all other vertices to have correction sets, so run the above cited algorithm to see if this graph has gflow. If it finds a gflow $(g, \prec)$, we have found a certificate $(L, g, \prec)$.

Repeat this for all possible choices for sets L until we find a choice that gives us a certificate. If our graph has a glack of k, we will find it. If our graph does not have glack, we will know because none of the possibilities give a certificate. There are $\binom{n}{k} = \mathcal{O}(n^k)$ sets to choose from, giving a total runtime of $\mathcal{O}(n^{k+4})$.

This gives us a simple (albeit terrible) scheme for finding the glack of arbitrary graphs.

Corollary 4.3.2. Let (G, I, O, λ) be an open graph on n vertices. Then there exists an algorithm with a complexity of $\mathcal{O}(2^n)$ that determines the glack and finds a certificate.

Proof. Iteratively run Proposition 4.3.1 from $k = 0$ up to n until you find a certificate. Adding up all iterations gives a runtime of $\mathcal{O}(n^n)$. But the performance guarantee in Proposition 4.3.1 only had a term n^k because of the $\binom{n}{k}$ term. We can get a marginally better performance guarantee when we realize we test all $L \in \overline{O}$, giving a performance of $\mathcal{O}(2^n\, n^4)$ instead.

Finding the glack is of course more difficult than finding gflow. As such, we do not expect very efficient bounds on complexity. However, this bound is very crude and pessimistic. While it is indeed possible for the glack k to be of order $\mathcal{O}(n)$, this is unlikely in practice. This is because we always start with a glack of 0 after converting a circuit, as per Section 3.3. Most operations we are interested in maintain glack or decrease glack.

Graphs with a large glack may not even be worth considering: if we extract a graph with Theorem 4.2.9, we get k post-selections. The chance we discard the result of a run of this extracted circuit as "invalid" generally increases exponentially with the number of post-selections. As such, we may as well assume (or try to ensure) that k is of the order $\log n$. This nullifies the exponential nature of the discard chance somewhat, and speeds up the algorithm's runtime similarly.

Finally, while the proof of Proposition 4.3.1 grabs arbitrary sets of vertices, we can make informed choices about what vertices to include or exclude. Most statements in Sections 4.1.1 and 4.1.2 allow us to trace the lacking set throughout our manipulations. Additionally, on a case-by-case basis, we can reduce the exponent in the complexity depending on the structure of the graph, as in the following three propositions.

Proposition 4.3.3. Let (G, I, O, λ) be an open graph, and let $v \in I$ with $\lambda(v) \neq \mathrm{XY}$. Then v is lacking in every certificate.

Proof. To assign a correction set, by either (g4) or (g5), we must have $v \in g(v)$. This contradicts (g0).

As such, when choosing a set of k vertices, we must always include such a v, which removes quite some possibilities. In addition, checking for vertices whether this condition holds is insignificant compared to the algorithm's runtime.

Proposition 4.3.4. Let (G, I, O, λ) be an open graph, $(L, g, \prec)$ a glack certificate, u be a vertex of degree 1, and v its neighbor. If $\lambda(u) = \mathrm{YZ}$ or $\lambda(v) = \mathrm{XY}$, then $u \notin L$ or $v \notin L$.

Proof. Both cases have the same proof. Suppose we have a certificate in which both u and v are lacking. Then setting the correction set of u (in the first case) or v (in the second case) to $\{u\}$ gives a strictly better certificate, which is a contradiction.

Proposition 4.3.5. Let (G, I, O, λ) be an open graph, $(L, g, \prec)$ a glack certificate, and let $u, v \in V(G)$.

- If $\lambda(u) \neq \mathrm{YZ} \neq \lambda(v)$ and $N(u) = N(v)$, then $u \in L$ or $v \in L$.
- If $\lambda(u) \neq \mathrm{XZ} \neq \lambda(v)$, $u \in N(v)$, and $N(u) \setminus \{v\} = N(v) \setminus \{u\}$, then $u \in L$ or $v \in L$.

Proof. The first bullet point is the same as the proof of Lemma 4.19 in [4], except we use both (g3) and (g4) instead of just (g3), as we also allow for XZ vertices. The proof argues that as $N(u) = N(v)$, $u \in \mathrm{Odd}(S) \Leftrightarrow v \in \mathrm{Odd}(S)$ for any set S. If neither are lacking, this includes in particular $g(u)$ and $g(v)$, giving both $u \prec v$ and $v \prec u$, which is a contradiction.

For the second bullet point, if these two vertices have no other neighbors, one of the two must clearly be lacking. As such, assume they share a common neighbor w. In $G * w$, u and v are disconnected, and still have the same neighbor set. The planes of u and v are also updated from non-XZ to non-YZ. So by the first part, there is no certificate with neither lacking. By Lemma 4.1.4, local complementations maintain lacking sets, so the same holds for G.

These statements reduce the choice in your potential lacking set somewhat, and these conditions can be efficiently detected when compared to the algorithm's runtime.

4.4. Analyzing spider nest identities

In Section 2.3.1, we introduced spider nest identities as means to reduce T-gate count. These have been known to not maintain gflow, but with glack introduced, we can now give definite bounds on how "bad" a specific spider nest is for a given graph. Conversely, this also tells us how "good" a specific spider nest can be for reducing glack.

Throughout this section, we will write $\{v_i\}$ for the set of XY-plane vertices that the spider nest identities apply to, and we will write $\{u_j\}$ for the set of YZ-plane phase gadgets that exist before the spider nest identity. For convenience, we will index the

gadgets with a multi-index j for which vertices v_i it is connected to. For instance, phase gadget u_{124} is connected to v_1, v_2, and v_4.

We will freely switch between the interpretation that u_i is a separate gadget connected to only v_i, and that we unify $u_i = v_i$. This is justified by Lemma 4.1.17, which maintains glack, as we only consider gadgets connected to XY-vertices.

Additionally, we will not be concerning ourselves with specific phases of the gadgets. We will only consider whether a phase gadget is of Clifford phase, or of T phase. We can do this, because applying Lemma 4.1.16 in reverse states that splitting these gadgets maintains glack. For instance, if we have a phase gadget with a phase of $\frac{5}{4}\pi$, it can be split into a gadget with a phase of $\frac{\pi}{4}$ and gadget with a phase of π.

Finally, throughout this analysis, we will only be changing the u_j. This will usually maintain all gflow conditions (g0) through (g5) for other vertices w. Conditions (g0) and (g3) through (g5) will never change. For the other two conditions, if some u_j is in $g(w)$ or $\mathrm{Odd}(g(w))$, we get $w \prec u_j$ by (g1) or (g2). Most of the time, our changes will not invalidate these relations. In particular, defining $g(u_j) = \{u_j\}$ does not put these relations in jeopardy. For that reason, we will not mention these w in the proofs unless something more interesting happens.

We will now start our analysis with two special cases: applying spider nest identities to only the input vertices, or only the output vertices.

Proposition 4.4.1. Let (G, I, O, λ) be an open graph. Let $\{v_i\} \subseteq O$ be output vertices. Then adding or removing any u_j maintains glack.

Proof. First consider adding a phase gadget u_j to G. Let $(L, g, \prec)$ certify the glack before this addition. We want to define $g'(u_j) = \{u_j\}$ to satisfy (g5), but we need to ensure this does not introduce cyclic dependencies. As vertices in O do not have correction sets, u_j's neighbors can be taken to be $\prec$-maximal. Then $g'(u_j)$ does not introduce cycles in $\prec$. Note that this addition cannot decrease glack.

This immediately implies that removing a gadget u_j from G also maintains glack, as the reverse operation of adding it back maintains it.

Proposition 4.4.2. Let (G, I, O, λ) be an open graph. Let $\{v_i\} \subseteq I$ be input XY-vertices. Then adding or removing any u_j maintains glack.

Proof. First consider the graph G_0 with no gadgets $\{u_j\}$ present, which has some certificate $(L_0, g_0, \prec_0)$. Define $L_{\{v_i\}} = L_0 \cap \{v_i\}$.

Now consider the graph G_1 with all possible gadgets $\{u_j\}$ present. We can modify G_0's certificate into a candidate certificate of G_1 by setting $g_0(u_j) = \{u_j\}$, and updating $\prec_0$ appropriately. This is because the v_i are not part of any correction set themselves by (g0). We will show by induction on $|\{u_j\}|$ (downwards from "all gadgets" to "no gadgets") that for the graph with a certain set of gadgets $\{u_j\}$ added, there exists a candidate certificate $(L, g, \prec)$ such that:

i. We have $L \cap \{v_i\} = L_{\{v_i\}}$;

ii. None of the u_j are lacking;

iii. There are no indices i, j such that $v_i \prec u_j$.

Additionally, we will be using only operations that maintain glack. This means that all of our candidate certificates are actual certificates, as we eventually reach G_0. They

are also all the same size, so the glack of all of these graphs is the same. No matter what G is, it will be visited during induction at some point, and so will G with any gadget removed/added. As all of these share the same glack, the statement follows.

Note that for both G_0 and G_1, the above certificates satisfy (i) through (iii).

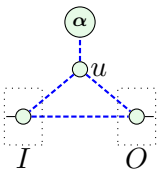
Let G' now be a graph with a non-full constellation $\{u_j\} \not\ni u_{j_0}$. By the induction hypothesis, adding u_{j_0} back gives us a graph G'' with a certificate $(L'', g'', \prec'')$ satisfying (i) through (iii). If we remove u_{j_0} with Lemma 4.1.14, we maintain lacking sets (and thus (i), (ii)) and order (and thus (iii)) if:

1. There is no certificate $(L, g, \prec)$ of G'' where u_{j_0} is lacking and minimal;
2. There is no certificate $(L, g, \prec)$ of G'' where $w \in L$, $u_{j_0} \in \mathrm{Odd}_{G''}(g(w))$, and certain other properties hold.

As the only $\prec$-minimal lacking YZ-vertices are input vertices and $u_{j_0} \notin I$, (1) holds. Because all neighbors of u_{j_0} are inputs, (2) is also true by (g0). As such, we are done.

Corollary 4.4.3. Let (G, I, O, λ) be an open graph. Let either $\{v_i\} \subseteq I$ be input XY-vertices, or $\{v_i\} \subseteq O$ be output vertices. Then applying *any* spider nest identity to $\{v_i\}$ maintains glack.

It is very important to note that the above statement is *not* "Let $\{v_i\} \subseteq I \cup O$ be input XY-vertices or output vertices". In this case, glack may actually change, because the vertices can depend on one another. For example, removing u in the following graph reduces the glack from 1 to 0.



(4.1)

There are in fact two different mechanisms at play that can change the glack when applying spider nest identities. The first can be seen in the above figure: if different vertices in $\{v_i\}$ are connected, there may be some temporal dependence. In other words, we might have $v_{i_1} \in g(v_{i_2})$ or $v_{i_1} \in \mathrm{Odd}(g(v_{i_2}))$ forced into our certificates. In this case, the simple $g(u) = \{u\}$ we used in the above proofs may cause a cyclic dependency, if u is in the odd neighborhood of $g(v_{i_1})$ or $g(v_{i_2})$. This problem reaches even further than directly connected vertices, as $\prec$ is transitive. Ideally, we would like all vertices $\{v_i\}$ to exist "at the same time", but is this untrue in general. We will dive deeper into this in Section 4.4.1.

The second mechanism has to do with specifically removing a vertex u_j. If u_j appears in some $g(w)$ and we remove it, we need a substitute for it in correction sets. For instance, if we apply an identity that removes u_{12}, but adds u_{34} and u_{1234}, we can replace the former with the latter two in any correction set without changing odd neighborhoods. If this is not possible, we may create lack.

In both the input and output cases, we did not have these issues by the special properties "being an input" and "being an output" give us. In the general case however, we do not have the ability to bound the introduced lack very well.

Proposition 4.4.4. Let (G, I, O, λ) be an open graph. Then applying *any* spider nest identity to $\{v_i\}$ increases the glack by at most $|\{v_i\}|$.

Proof. This is simply the difference between the "best before" (with no lack) and the "worst after". This "worst after" consists of designating every v_i as lacking, and setting $g(u_j) = \{u_j\}$ for all u_j in the resulting graph.

Note that removing any u_j from any $g(w)$ does not introduce any problems, as we can substitute u_j for $\{u_i \mid i \in j\}$. If these u_i do not exist separately from the v_i yet, we can freely introduce them with Lemma 4.1.17 applied in reverse.

In theory, this is not even terrible: we can remove an exponential number of T gadgets at only a cost of linear lack. However, exponential amounts of T gadgets are seldom encountered in practice.

We can achieve better bounds by looking more into the "at the same time" mentioned above.

4.4.1. Simultaneity

We start with the eponymous definition.

Definition 4.4.5 (Simultaneity). A set of XY-vertices $\{v_i\}$ is *simultaneous under* a certificate $(L, g, \prec)$ if there are no i, j with $v_i \prec v_j$. If such a certificate exists, we call $\{v_i\}$ *simultaneous*.

While this definition is that of the *anti-chain*, we want to emphasize the temporal interpretation of $\prec$.

This is the first definition that does not (just) care about what vertices in the certificate are lacking, but also the ordering of the vertices. For this reason, all statements on graph manipulations in Section 4.1 include the effect on the *entire* certificate, and not just the lacking vertices. This gives us a different subset of safe operators to use when we care specifically about simultaneity. Fortunately for us, all operations in Proposition 4.1.3 maintain L and $\prec$ for all existing vertices if the glack stays the same.

Simultaneity will turn out to solve the first of the two mechanisms introducing glack. We can work around the second mechanism that appears when removing vertices, by not removing vertices. Consider any spider nest identity. Usually, when gadgets of phase 0 or phase π remain, the graph gets simplified and those gadgets get removed. Call spider nest identities that instead keep these spiders in the graph *non-removing spider nest identities*.

Proposition 4.4.6. Let (G, I, O, λ) be an open graph, and let $\{v_i\}$ be simultaneous. Applying *any* non-removing spider nest identity to $\{v_i\}$ does not increase glack.

Proof. Let $\{v_i\}$ be simultaneous under $(L, g, \prec)$. From a graphical perspective, the only thing non-removing spider nest identities do, is add vertices to the graph. To that end, suppose we add gadget u_j to the graph, for some j.

We want to define $g(u_j) = \{u_j\}$. As this would give $u_j \prec v_i$ for all $i \in j$, we may not have $v_i \prec u_j$. As u_j does not appear in any correction set, the only way for dependencies like this to appear is via odd neighborhoods. In particular, we require

vertices v_{i_1} with $i_1 \in j$, and w in v_{i_1}'s future such that u_j lies in $\mathrm{Odd}(g(w))$. This, in turn, requires the existence of some $v_{i_2} \in \{v_{i_1}\}$ with $v_{i_2} \in g(w)$. But this would give $v_{i_1} \prec w \prec v_{i_2}$, which this cannot happen, as it would contradict simultaneity.

Corollary 4.4.7. Let (G, I, O, λ) be an open graph, and let $\{v_i\}$ be arbitrary. Consider

$$D := \{v_i \mid \exists i_0 \text{ such that } v_{i_0} \prec v_i\} \subset \{v_i\}\,.$$

Applying *any* non-removing spider nest identity to $\{v_i\}$ increases glack by at most $|D|$.

Proof. If we mark all of D as lacking, and thus $\prec$-maximal, and thus simultaneous, we can apply Proposition 4.4.6. There may exist a better certificate than the one marking all of D as lacking, however.

Note that "not removing gadgets" is partially a hack to achieve more consistent bounds. In particular, compare how Corollary 4.4.7's worst case adds $|D| \leq |\{v_i\}| - 1$ glack, instead of Proposition 4.4.4's $|\{v_i\}|$. But this does not tell the whole story. Removing gadgets may not only increase the glack, but it may *decrease* the glack by the same amount as well. This is because spider nests are their own inverse. By keeping phase-0 and phase-π gadgets in the graph, we might be passing up opportunities to reduce glack without knowing, as in Figure (4.1) earlier. Nevertheless, with simultaneity, we can achieve even better bounds.

Lemma 4.4.8. If $M \in \mathbb{F}_2^{r\times c}$ has r distinct non-zero rows, then $\mathrm{rank}(M) > \lfloor \log_2 r \rfloor$.

Proof. Suppose that instead $\mathrm{rank}(M) \leq \lfloor \log_2 r \rfloor$. Then the rows of M span a total of $2^{\mathrm{rank}(M)} \leq 2^{\lfloor \log_2 r \rfloor} \leq r$ vectors in $\mathbb{F}_2^c$. This span contains 0, so the rows of M span at most $r - 1$ non-zero vectors. This contradicts M having r distinct non-zero rows.

Proposition 4.4.9. Let (G, I, O, λ) be an open graph with a glack of k, and let $\{v_i\}$ be simultaneous with $|\{v_i\}| = 4$. Suppose we apply the spider nest identity on four vertices to this set, which transforms t_1 gadgets into t_2 gadgets.

- If $t_1 < 2^{4-1} \leq t_2$, the resulting glack is at most k.
- If $t_1 \geq 2^{4-1} > t_2$, the resulting glack is at most $k + 4 - \lceil \log_2(t_2 + 1) \rceil$.

Less compactly written, if we end up with 7, 6, 5, or 4 gadgets, the glack can increase by at most one. If we end up with 3 or 2 gadgets, it can increase by at most 2. If we end up with just one gadget, it can increase by at most three. Finally, if nothing remains, we get our original bound of Proposition 4.4.4.

Proof. Let $\{v_i\}$ be simultaneous under $(L, g, \prec)$. In Proposition 4.4.6, we already showed that adding phase gadgets does not increase glack. We now need to consider what happens when removing a gadget u_j.

When considering (g2), removing vertices can only remove dependencies that appear because of odd neighborhoods, as there are simply fewer neighbors. Thus, we do not need to care about w with $u_j \in \mathrm{Odd}(g(w))$. However, removing vertices can influence (g1) by changing the odd neighborhoods, so we want a replacement for u_j.

Consider the biadjacency matrix M where the rows correspond to the phase gadgets

after deletion of u_j, and the columns correspond to the $\{v_i\}$. If we want a replacement for u_j in the odd neighborhood, we need some $u_{j_0}, \ldots, u_{j_\ell}$ with the same odd neighborhood as u_j. In the biadjacency matrix, this is equivalent to finding rows indexed by $u_{j_0}, \ldots, u_{j_\ell}$ to the biadjacency vector between u_j and $\{v_i\}$.

If M is of full rank, we can guarantee replacements for u_j. Otherwise, we introduce rows to make M full rank. This goes as follows: whenever two columns at index v_{i_1} and v_{i_2} in M are the same, add a row $e_{v_{i_1}}$ to M. In the graph, this corresponds to creating a phase gadget u_{i_1}, which we can do by (the inverse of) Lemma 4.1.17, which maintains both lacking sets and $\prec$.

However, while odd neighborhoods are maintained by this substitution, correction sets themselves are not, and this process may create cyclic dependencies in $\prec$. When substituting in u_{i_1} creates such a dependency, it can be solved by marking v_{i_1} as lacking.

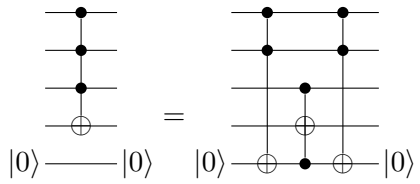
Now suppose we go from 8 to 7 gadgets. In the worst case, this leaves us with just seven rows $\{u_j\}$ in M, which may have rank 3 instead of 4. This necessitates the introduction of a vertex that is possibly lacking. This exact same argument holds with the same results when going from 9 to 6, 10 to 5, and 11 to 4 gadgets. Then, when going from 12 to 3 gadgets, the rank of M is at most 2. In general, this argument for arbitrary values of t_2 gives the values as in the statement.

Proposition 4.4.9 is suggestively written with the number 4 everywhere. Similar statements hold for other spider nest identities as well. For instance, the five-qubit spider nest will replace all "4"'s in the statement with "5". Then the conclusion becomes "ending up with 15 through 8 gadgets increases glack by at most 1, ending up with 7 through 4 gadgets by at most 2, etc.".

This bound is still somewhat unsatisfying. Vertices u_j we remove do not necessarily lie in correction sets. In addition, the argument is based on matrix rank to guarantee the existence of replacements, no matter what we remove. But even if the rank is too low, replacements may still exist for specific instances.

4.4.2. The Controlled-Toffoli example

The controlled-Toffoli gate, or CCCNOT for short, applies an X gate to a qubit if three control qubits are all set. We can implement this gate with only Toffoli gates as follows:

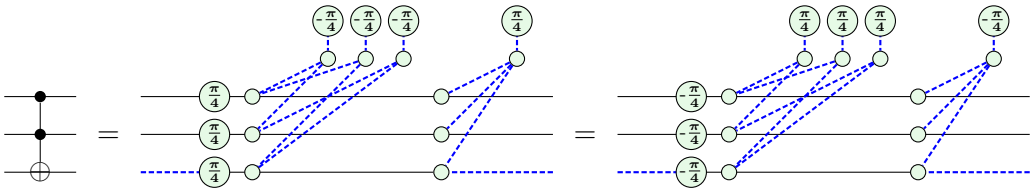


(4.2)

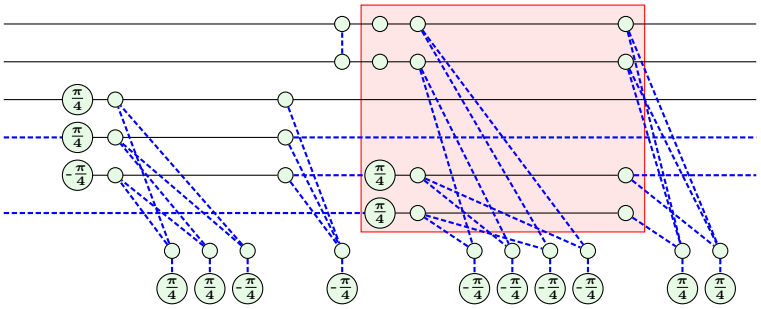
The first three qubit wires are the inputs, the fourth wire is the output, and the last qubit is an auxiliary qubit that must start in the $|0\rangle$ state and outputs that state as well.

We will use this gate as an example of how we can use the spider nest identities to trade between the T-count, and the number of post-selections. For that, we first need to convert the CCCNOT gate into a ZX-diagram.

Converting this circuit to a ZX-diagram requires an implementation of the Toffoli gate. It is known that the best you can do, without ancillae, is 7 T-gates [2]:

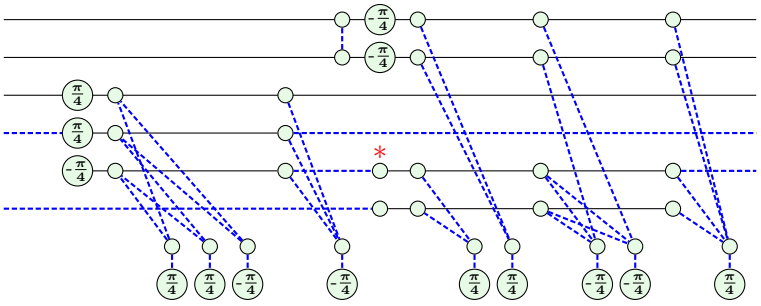


Applying this to Equation (4.2), we get 21 T-phases. However, with some simplifications, some gadgets can be canceled out. More specifically, all phase gadgets involving only the first two qubit wires in Equation (4.2) cancel out. This gives a total of 15 T-phases.[1] After some further changes to make the diagram more presentable, we get the following. (We do not write the $|0\rangle$ assumption on the last input and output wire.)



Fusing the spiders on the same wires in the red box, we can see that we have four XY-vertices, with a total of 8 T-phase gadgets attached. This means we can use the four-qubit spider nest identity on those vertices. This will turn the constellation of 8 T-phases into a constellation of 7 T-phases.

When considering the open graph the above fragment represents, it has gflow. The top two wires can be simultaneous in that flow. However, the bottom two wires will always have a dependency, no matter what flow you choose. This is the only forced dependency. By Corollary 4.4.7, this means that the glack can increase by at most 1. Unfortunately for us, this bound is reached, and applying the spider nest identities, we go from a diagram with gflow to a diagram without; the following diagram has a glack of 1:



This means that extracting the above diagram with Theorem 4.2.9 will result in a circuit with 5 inputs, one extra prepared qubit, and 6 outputs, where the additional output is post-selected. In fact, after converting to an open graph, the $*$-marked vertex will be the one lacking. In output form, this vertex will be post-selected as $\langle 0|$. This extracted diagram has a T-count of 14, compared to the 15 we had before applying the spider nest identity.

[1] We used the PyZX software package [20] to automate the cancellation of this diagram.

We must note that if one allows for post-selections, as we do, this scheme is not optimal. By using a completely different analysis of *H-boxes* in the *ZH-calculus*, the T-count can be further reduced to only 11 if you allow for one post-selection, and to 8 if you allow for two post-selections. There is also a C^nNOT-specific analysis that implements the CCCNOT gate with only 6 T-gates and one corrected ancilla [15].

5. Conclusion

We have introduced the definition of *glack* to generalize the concept of *gflow* from a boolean to a numeric metric. Next, we looked at how operations that are known to preserve gflow behave under this more general glack. Using this, we could then argue what optimizations are still applicable, without increasing glack. Quite some operations behave well under glack, but crucially, deletion of non-XY vertices has the additional requirement of vertices not being *completely lacking*, as it will otherwise add 1 glack. Because of this, converting a diagram into *reduced phase gadget form*, or applying other optimizations that depend on this form, may unfortunately increase glack.

Lacking graphs can be extracted, but unlike graphs with gflow, gain a number of extra post-selected ancillae equal to the glack. For this extraction scheme we need *certificates*. For graphs with n vertices and a glack of k, this requirement is relatively harmless if k is of the order $\mathcal{O}(\log n)$, as these certificates can be found somewhat efficiently in this case. If we do not provide a certificate, we can still extract efficiently, but we instead get a circuit that uses *at least* k post-selections, which can be suboptimal.

We also gave an example of how glack allows us to formalize new trade-offs by example of spider nest identities. This trade-off allows us to prefer either a higher T-count with fewer ancillae, or a lower T-count with more ancillae.

5.1. Further directions

As we have only defined the basic properties of glack, there is still a lot to explore.

The most significant property that I believe to hold true, but could not prove, is that for any open graph, $\mathcal{L} = \varnothing$. While part of it is motivated by "I have not been able to find an example", a part of it is also motivated by Lemma 4.1.2, which limits how often vertices have to be referenced; for any vertex u, there is some certificate in which there is at most one v_a with $u \in g(v_a)$, and other than v_a there is at most one v_b with $u \in \mathrm{Odd}(g(v_b))$.

Suppose that in particular u is lacking. If neither v_a nor v_b exist, u is $\prec$-minimal and not in $\mathcal{L}$. If only one of the two exists, we can make that vertex lacking, and assign u any correction set, and $u \notin \mathcal{L}$. Only if for *every* certificate a v_a and v_b exist, we're stuck. Any examples that I can come up with that fall into this "bad case" still *do* have a certificate $(L, g, \prec)$ in which $u \notin L$, however.

The most obvious application of "$\mathcal{L} = \varnothing$" is that Lemma 4.1.14, deletion of vertices, becomes just as powerful as in the gflow case. As a result, we can then simplify the graph in all the same ways as in the gflow case without increasing glack – in essence, we would be able to include *all* statements that Section 3.4 contained but Section 4.1 omitted. This would mean that all regular gflow optimization schemes can be applied to lacking graphs, without worsening their glack. Beyond this, being able to assume non-minimal lacking vertices are non-lacking in some other certificate is a somewhat powerful tool that allows for tighter bounds when proving statements that describe a glack reduction.

For instance, Lemma 4.1.16 can then be proven to maintain glack *exactly*, instead of being an upper bound. Proposition 4.4.9 would also simply state "spider nest identities on simultaneous vertices do not increase glack", instead of the current bound.

There are other points of interest as well. While we have shown that glack allows for new trade-offs, we have not looked too in-depth into optimization. For instance, there may be optimization schemes that introduce glack, do some rewrites, and then remove that glack again, predictably.

We also assume the circuits we start with are isometries. If we have with circuits with measurements, the resulting diagrams may start out with glack. We have not looked into this.

Much of what is described in this thesis is also considerably "worst-case". There are likely improvements to be made to our schemes for finding certificates, and for extracting circuits. It may also be possible to give Theorem 4.2.3 a (probabilistic) upper bound for the number of extra ancillae. This may make the requirement of having a certificate on hand less stringent.

While we have discussed a generalization of gflow, a similar generalization of Pauli flow may also be worth looking into. As mentioned before, Pauli flow is an even more general flow than gflow, and solves the ambiguity when measuring for $\{|0\rangle, |1\rangle\}$, $\{|+\rangle, |-\rangle\}$, or $\{|i\rangle, |-i\rangle\}$. With gflow, choosing different planes results in graphs with genuinely different behavior, which culminates in this choice being the difference between being extractable, or not, in some cases. Pauli flow does not have this problem, which may allow for a better behaved "Pauli lack".

Finally, research into the correct pronunciation of *glack* is still ongoing. While /ˈd͡ʒiːlæk/ is consistent with the pronunciation of *gflow*, the inherent comedic value of /glæk/ ought to be considered as well.

Bibliography

[1] Matthew Amy, Dmitri Maslov, and Michele Mosca. Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1476–1489, 2014. doi:10.1109/TCAD.2014.2341953.

[2] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):818–830, 2013. doi:10.1109/TCAD.2013.2244643.

[3] Miriam Backens and Aleks Kissinger. ZH: A Complete Graphical Calculus for Quantum Computations Involving Classical Non-linearity. In Peter Selinger and Giulio Chiribella, editors, *Proceedings of the 15th International Conference on Quantum Physics and Logic, Halifax, Canada, 3-7th June 2018*, volume 287 of *Electronic Proceedings in Theoretical Computer Science*, pages 23–42. Open Publishing Association, 2019. doi:10.4204/EPTCS.287.2.

[4] Miriam Backens, Hector Miller-Bakewell, Giovanni de Felice, Leo Lobski, and John van de Wetering. There and back again: A circuit extraction tale. *Quantum*, 5:421, March 2021. doi:10.22331/q-2021-03-25-421.

[5] Jin-Yi Cai. Shor's algorithm does not factor large integers in the presence of noise. *Science China Information Sciences*, 67(7):173501, Jun 2024. doi:10.1007/s11432-023-3961-3.

[6] Zhenyu Cai, Ryan Babbush, Simon C. Benjamin, Suguru Endo, William J. Huggins, Ying Li, Jarrod R. McClean, and Thomas E. O'Brien. Quantum error mitigation. *Rev. Mod. Phys.*, 95:045005, Dec 2023. doi:10.1103/RevModPhys.95.045005.

[7] Shuxiang Cao. Multi-agent blind quantum computation without universal cluster state. *New Journal of Physics*, 25(103028), 10 2023. doi:10.1088/1367-2630/acfab6.

[8] K. Ch. Chatzisavvas, G. Chadzitaskos, C. Daskaloyannis, and S. G. Schirmer. Improving quantum gate fidelities using optimized Euler angles. *Phys. Rev. A*, 80:052329, Nov 2009. doi:10.1103/PhysRevA.80.052329.

[9] Bob Coecke and Ross Duncan. A graphical calculus for quantum observables. *Preprint*, 2007.

[10] Vincent Danos and Elham Kashefi. Determinism in the one-way model. *Phys. Rev. A*, 74:052310, Nov 2006. doi:10.1103/PhysRevA.74.052310.

[11] Vincent Danos, Elham Kashefi, Prakash Panangaden, and Simon Perdrix. *Extended Measurement Calculus*, page 235–310. Cambridge University Press, 2009.

[12] Ronald de Wolf. Quantum Computing: Lecture Notes, 2023. arXiv:1907.09415.

[13] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John van de Wetering. Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus. *Quantum*, 4:279, June 2020. doi:10.22331/q-2020-06-04-279.

[14] Ross Duncan and Simon Perdrix. Graph States and the Necessity of Euler Decomposition. In Klaus Ambos-Spies, Benedikt Löwe, and Wolfgang Merkle, editors, *Mathematical Theory and Computational Practice*, pages 167–177, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[15] Craig Gidney and N. Cody Jones. A CCCZ gate performed with 6 T gates. *arXiv e-prints*, page arXiv:2106.11513, June 2021. doi:10.48550/arXiv.2106.11513.

[16] T. Giurgica-Tiron, Y. Hindy, R. LaRose, A. Mari, and W. J. Zeng. Digital zero noise extrapolation for quantum error mitigation. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 306–316, Los Alamitos, CA, USA, oct 2020. IEEE Computer Society. doi:10.1109/QCE49297.2020.00045.

[17] Ian Glendinning. The bloch sphere. In *QIA Meeting*, pages 3–18, 2005.

[18] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/237814.237866.

[19] Calum Holker. Causal flow preserving optimisation of quantum circuits in the ZX-calculus. *arXiv e-prints*, 2024. arXiv:2312.02793.

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

[21] Aleks Kissinger and John van de Wetering. Reducing the number of non-Clifford gates in quantum circuits. *Phys. Rev. A*, 102:022406, Aug 2020. doi:10.1103/PhysRevA.102.022406.

[22] Ryan Krueger. Vanishing 2-Qubit Gates with Non-Simplification ZX-Rules. Master's thesis, University of Oxford, 2021.

[23] Tommy McElvanney and Miriam Backens. Flow-preserving ZX-calculus Rewrite Rules for Optimisation and Obfuscation. In Shane Mansfield, Benoit Valîron, and Vladimir Zamdzhiev, editors, *Proceedings of the Twentieth International Conference on Quantum Physics and Logic, Paris, France, 17-21st July 2023*, volume 384 of *Electronic Proceedings in Theoretical Computer Science*, pages 203–219. Open Publishing Association, 2023. doi:10.4204/EPTCS.384.12.

[24] Jisho Miyazaki, Michal Hajdušek, and Mio Murao. Analysis of the trade-off between spatial and temporal resources for measurement-based quantum computation. *Phys. Rev. A*, 91:052302, May 2015. doi:10.1103/PhysRevA.91.052302.

[25] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[26] Maximilian Nägele and Florian Marquardt. Optimizing ZX-Diagrams with Deep Reinforcement Learning. *arXiv e-prints*, 2024. arXiv:2311.18588.

[27] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018. doi:10.22331/q-2018-08-06-79.

[28] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi:10.1109/SFCS.1994.365700.

[29] Jules Tilly, Hongxiang Chen, Shuxiang Cao, Dario Picozzi, Kanav Setia, Ying Li, Edward Grant, Leonard Wossnig, Ivan Rungger, George H. Booth, and Jonathan Tennyson. The Variational Quantum Eigensolver: A review of methods and best practices. *Physics Reports*, 986:1–128, 2022. The Variational Quantum Eigensolver: a review of methods and best practices. doi:10.1016/j.physrep.2022.08.003.

[30] John van de Wetering. ZX-calculus for the working quantum computer scientist, 2020. arXiv:2012.13966.

Popular summary

When a programmer writes an algorithm, a *compiler* converts this code into a format a computer can execute. However, converting code is not the compiler's only job. The arguably more important task of a compiler is to optimize the programmer's code. This optimization can be both general, and specific to the target hardware.

With quantum computers, this is no different. Quantum algorithms are frequently specified as a *quantum circuit*. While computers work with *bits*, quantum computers work on *qubits*. These quantum circuits use a variety of *gates*, tiny programs that do very specific operations on at most a few qubits at once.

However, as of 2024, quantum computers are still very primitive. Not a lot of different kinds of gates can be used (and which gates can be used depends on the machine), and only few qubits can be used at once. This makes it all the more important to have access to a compiler that can optimize circuits well.

The *ZX-calculus* is a graphical language that represents quantum circuits as *ZX-diagrams*. These diagrams are more flexible to work with, and can be optimized in ways quantum circuits cannot. However, there is a problem. While turning quantum circuits into ZX-diagrams is easy, the opposite direction is not.

This *extraction* problem is known to be computationally difficult. However, if ZX-diagrams have a property called *gflow*, extraction becomes easy again. If we turn a quantum circuit into a ZX-diagram, that diagram has gflow. We do not want to lose this gflow when doing optimizations. Fortunately, gflow plays nice with most optimizations, and optimized diagrams can be extracted easily again.

However, some interesting optimizations do *not* maintain gflow. Even if we start out with a diagram that has gflow, it can be lost if we apply them. In this thesis, we introduce the notion of *glack*, that allows us to quantify this behavior more precisely. Instead of saying that a diagram has gflow, it "has a glack of 0". And if a diagram does not have gflow, it has a glack of 1, or 2, or 230, etc., where higher values are worse. Now, instead of saying that an optimization "maintains gflow", we can look at how the glack changes. Ideally, it does not increase, but if it does, it hopefully doesn't increase by a lot.

This notion of glack would not be very useful if we could not also extract diagrams with glack, so we show how we can recover a quantum circuit from such diagrams. Unfortunately, if a diagram has a glack of k, it results in k extra qubits being necessary. As modern quantum computers only have access to few qubits, we want to avoid high glack. What's worse, these extra qubits are *post-selections*, a special kind of "bad" qubit. This gives us the opportunity to make a trade-off. Is the optimization that increases glack really worth these extra qubits? A question like this does not have a single, well-defined answer, but instead depends on the specifics of the quantum computer you are targeting.

A. Gflow preservation proofs

This appendix contains the proofs Section 3.4 omitted. Just like the content of Section 3.4, most of these proofs come from [4]. We present the proofs for Lemmata 3.4.1 and 3.4.2 slightly differently. The proof of Lemma 3.4.10 comes from [13].

Before we can prove how gflow interacts with local complementations, we need to know how odd neighborhoods interact with local complementations. We will need the following definition to keep the exposition compact.

Definition A.1.1 (Augmented odd neighborhood). Suppose we are considering the local complementation about u. Then the *augmented odd neighborhood* of A is $\mathrm{AugOdd}(A) = \mathrm{Odd}_G(A) \,\Delta\, (N_G(u) \cap A)$.

This set has only one property that we will care about: $\mathrm{AugOdd}(g(v))$ lies in the future of v, as it only consists of vertices in $g(v)$ or $\mathrm{Odd}_G(g(v))$.

Lemma A.1.2. Let G be a graph, $A \in V(G)$ a set of vertices, and $u, v \in V(G)$ vertices. Then

$$N_{G*u}(v) = \begin{cases} N_G(v) \,\Delta\, N_G(u) \,\Delta\, \{v\} & \text{if } v \in N_G(u) \\ N_G(v) & \text{otherwise,} \end{cases}$$

and

$$\mathrm{Odd}_{G*u}(A) = \begin{cases} \mathrm{AugOdd}(A) \,\Delta\, N_G(u) & \text{if } u \in \mathrm{Odd}_G(A) \\ \mathrm{AugOdd}(A) & \text{otherwise.} \end{cases}$$

Proof. In $G * u$, all edges inside $N_G(u)$ get complemented. When $v \in N_G(u)$, this means we toggle connection with all non-v neighbors of u, $N_G(u) \,\Delta\, \{v\}$. This gets us the first part of the statement. Now consider the second part. First, we get

$$\mathrm{Odd}_{G*u}(A) = \bigtriangleup_{v \in A} N_{G*u}(v) = \bigtriangleup_{v \in A \cap N_G(u)} N_{G*u}(v) \,\Delta \bigtriangleup_{v \in A \setminus N_G(u)} N_{G*u}(v),$$

where we split A into two parts, $A \cap N_G(u)$ and $A \setminus N_G(u)$. By the above, the first part turns into

$$\bigtriangleup_{v \in A \cap N_G(u)} N_{G*u}(v) = \bigtriangleup_{v \in A \cap N_G(u)} N_G(v) \,\Delta \bigtriangleup_{v \in A \cap N_G(u)} N_G(u) \,\Delta \bigtriangleup_{v \in A \cap N_G(u)} \{v\},$$

and the second part turns into

$$\bigtriangleup_{v \in A \setminus N_G(u)} N_{G*u}(v) = \bigtriangleup_{v \in A \setminus N_G(u)} N_G(v).$$

Now looking at both parts, among other things we take the symmetric difference of $N_G(v)$ over all $v \in A$, which equals $\mathrm{Odd}_G(A)$. What remains is the symmetric

difference with

$$\bigtriangleup_{v\in A\cap N_G(u)} N_G(u) \,\Delta\, \bigtriangleup_{v\in A\cap N_G(u)} \{v\} = \bigtriangleup_{v\in A\cap N_G(u)} N_G(u)\,\Delta\,(A\cap N_G(u)).$$

This expression contains a total of $|A\cap N_G(u)|$ of $N_G(u)$ terms, most of which cancel out with each other. Combining everything, we get

$$\mathrm{Odd}_{G*u}(A) = \begin{cases} \mathrm{Odd}_G(A)\,\Delta\, N_G(u)\,\Delta\,(A\cap N_G(u)) & |A\cap N_G(u)| \text{ is odd} \\ \mathrm{Odd}_G(A)\,\Delta\,(A\cap N_G(u)) & \text{otherwise.} \end{cases}$$

This is the same as what we wanted to prove.

We also need to show that the changes in planes correspond to changes in the underlying ZX-diagram.

Lemma A.1.3. Let (G, I, O, λ) be an open graph, and $u \in \overline{I}$ not be an input. Then $G * u$ has planes λ' as described in Lemma 3.4.1: for $v \in N(u) \cap \overline{O}$, the XZ- and YZ-planes get swapped, and if $u \in \overline{O}$, its XY- and XZ-planes get swapped.

Proof. The local complementation introduces $\frac{\pi}{2}$ spiders at $N(u)$ and a $-\frac{\pi}{2}$ spider at u itself. These spiders are put before the measurement effects in Table 3.3, and we need to rewrite the result to be of that form again. For brevity, we will omit the $+a\pi$ term that is maintained throughout the computations.

We start with vertices $v \in N(u)$. If $\lambda(v) = \mathrm{XY}$, we get $-\frac{\pi}{2}-\alpha = -\alpha+\frac{\pi}{2}$. If $\lambda(v) = \mathrm{XZ}$, we get $-\frac{\pi}{2}-\frac{\pi}{2}-\alpha = -\pi-\alpha = --\alpha$ by Lemma 2.2.3. Finally, if $\lambda(v) = \mathrm{YZ}$, we get $-\frac{\pi}{2}-\alpha$. We can see that the XY-plane is maintained, but the XZ- and YZ-planes swap places.

Now consider u. This time, we will manipulate the diagrams such that we can use Equation 2.3 and a color change. If $\lambda(u) = \mathrm{XY}$, we get

$$-\tfrac{\pi}{2}-\alpha = -\tfrac{\pi}{2}-\tfrac{\pi}{2}-\tfrac{\pi}{2}-\tfrac{\pi}{2}-\alpha+\tfrac{\pi}{2} = -\tfrac{\pi}{2}\cdots\alpha+\tfrac{\pi}{2} = -\tfrac{\pi}{2}-\alpha+\tfrac{\pi}{2}.$$

If $\lambda(u) = \mathrm{XZ}$, we get

$$-\tfrac{\pi}{2}-\tfrac{\pi}{2}-\alpha = -\tfrac{\pi}{2}-\tfrac{\pi}{2}-\tfrac{\pi}{2}-\tfrac{\pi}{2}-\pi-\alpha = -\tfrac{\pi}{2}\cdots-\alpha = -\tfrac{\pi}{2}-\alpha.$$

Finally, if $\lambda(u) = \mathrm{YZ}$, we get $-\frac{\pi}{2}-\alpha = -\alpha-\frac{\pi}{2}$. We can see that the YZ-plane is maintained, but the XY- and XZ-planes swap places.

We can now proceed with proving that local complementations maintain gflow.

Proof of Lemma 3.4.1. First assume $u \notin O$. Define updated flow g' as follows:

$$g'(u) = \begin{cases} g(u)\,\Delta\,\{u\} & \text{if } \lambda(u) \in \{\mathrm{XY}, \mathrm{XZ}\} \\ g(u) & \text{otherwise,} \end{cases}$$

and for $v \in \overline{O} \setminus \{u\}$

$$g'(v) = \begin{cases} g(v) \,\Delta\, g'(u) \,\Delta\, \{u\} & \text{if } u \in \mathrm{Odd}_G(g(v)) \\ g(v) & \text{otherwise.} \end{cases}$$

The partial order $\prec$ does not change. By the above Lemma A.1.3, the measurement planes indeed change from λ to λ' as described. We need to show that this g' combined with λ' gives a valid gflow. We will do this in steps.

Properties (g0) through (g5) hold for $g'(u)$. Note that (g0) and (g1) hold as $g'(u)$ at most changes by u. By using the above Lemma A.1.2 and the above definition of $g'(u)$, one can show that no matter $\lambda(u)$, we have

$$\mathrm{Odd}_{G*u}(g'(u)) = \mathrm{AugOdd}(g(u)). \tag{A.1}$$

As this lies in the future of u, (g2) is maintained. Also note that the inclusion of u does not change between $\mathrm{Odd}_G(g(u))$ and $\mathrm{Odd}_{G*u}(g'(u))$, as $u \notin N_G(u)$. We now only need to consider how the inclusion of u in $g(u)$ and $g'(u)$ changes for each plane to show (g3) through (g5).

- If $\lambda(u) \in \{\mathrm{XY}, \mathrm{XZ}\}$, then $\lambda'(u)$ is the other one, and we need to show (g4) and (g3) respectively. These are true: as $u \notin g(u)$ if and only if $\lambda(u) = \mathrm{XY}$, by definition of g' we have $u \in g'(u)$ if and only if $\lambda(u) = \mathrm{XY}$.
- If $\lambda(u) = \mathrm{YZ}$, then $\lambda'(u) = \mathrm{YZ}$, and we need to show (g5). But this is true as well as the correction set does not update.

Properties (g0) through (g2) hold for $g'(v)$ for $v \neq u$. Note that (g0) holds trivially. If $u \in \mathrm{Odd}_G(g(v))$, we have $v \prec u \prec g'(u)$, ensuring $g'(v)$ lies in v's future, giving (g1). When $u \notin \mathrm{Odd}_G(g(v))$, the correction set is unchanged, so in this case we also get (g1). Now we need to prove (g2).

Similar to and by using Equation (A.1), one can show that

$$\mathrm{Odd}_{G*u}(g'(v)) = \begin{cases} \mathrm{AugOdd}(g(v)) \,\Delta\, \mathrm{AugOdd}(g(u)) & \text{if } u \in \mathrm{Odd}_G(g(v)) \\ \mathrm{AugOdd}(g(v)) & \text{otherwise.} \end{cases} \tag{A.2}$$

As mentioned before, $\mathrm{AugOdd}(g(v))$ lies in v's future. Additionally, if $u \in \mathrm{Odd}_G(g(v))$, we get $v \prec u \preceq \mathrm{AugOdd}(g(u))$. This means that in both cases, $\mathrm{Odd}_{G*u}(g'(v))$ lies in v's future and (g2) is satisfied.

Properties (g3) through (g5) hold for $g'(v)$ for $v \neq u$. First, note that v's inclusion in $g(v)$ and $g'(v)$ is the same, as we never have $v \in g'(u)$. However, $\mathrm{AugOdd}(g(v))$ flips the presence of v in $\mathrm{Odd}_{G*u}(g'(v))$ in Equation (A.2) if $v \in N_G(u) \cap g(v)$. This gives us yet another case distinction.

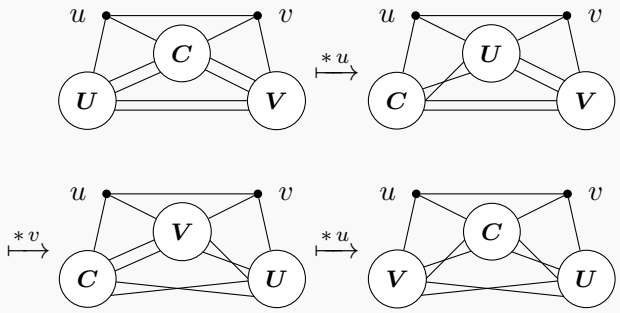
- If $v \notin g(v)$ or $v \notin N_G(u)$, then $\lambda'(v) = \lambda(v)$, and the inclusion of v in the odd neighborhood does not get flipped. This means (g3) through (g5) hold for g' because they hold for g.
- Otherwise, $v \in N_G(u)$, $\lambda(v) \in \{\mathrm{XZ}, \mathrm{YZ}\}$, $\lambda'(v)$ is the other one, and the inclusion of v in the odd neighborhood gets flipped. When $\lambda(v) = \mathrm{XZ}$, this flip removes v, giving the YZ-plane and (g5). When $\lambda(v) = \mathrm{YZ}$, this flip adds v, giving the XZ-plane and (g4).

Now, if $u \in O$, the proof actually becomes simpler. We do not need to define $g'(u)$ at all, and the following definition for $g'(v)$ for $v \neq u$ suffices:

$$g'(v) = \begin{cases} g(v) \,\Delta\, \{u\} & \text{if } u \in \mathrm{Odd}_G(g(v)) \\ g(v) & \text{otherwise.} \end{cases}$$

With this, Equation (A.2) simplifies to $\mathrm{Odd}_{G*u}(g'(v)) = \mathrm{AugOdd}(g(v))$, giving (g0) through (g2) as before. Consequently, the proof for properties (g3) through (g5) holds the same.

Proof of Lemma 3.4.2. As $G \wedge uv = G*u*v*u$, by applying 3.4.1 three times, gflow is maintained. We only need to show that the measurement planes change as described. We will copy the notation in Equation (2.4), and go through each individual step of the pivot. This means that we need to consider the following four diagrams:



We need to show what happens to the measurement planes to each vertex or neighborhood of interest here. As a shorthand, we will write (p_1, p_2, p_3) for "in G, XY-planes update to plane p_1, XZ-planes to p_2, and YZ-planes to p_3". This means that before the first local complementation, we start out with $(\mathrm{XY}, \mathrm{XZ}, \mathrm{YZ})$ for all vertices.

We will write all measurement plane changes for each of the four diagrams above in tabular form below, and highlight the updates of Lemma 3.4.1 in that step. Note that the neighborhoods U, C, and V move around with each local complementations (which can be seen in the figure above), which affects the next application of Lemma 3.4.1.

	G	$G*u$	$G*u*v$	$G*u*v*u$
u	(XY,XZ,YZ)	(XZ,XY,YZ)	(YZ,XY,XZ)	(YZ,XZ,XY)
v	(XY,XZ,YZ)	(XY,YZ,XZ)	(XZ,YZ,XY)	(YZ,XZ,XY)
U	(XY,XZ,YZ)	(XY,YZ,XZ)	(XY,XZ,YZ)	(XY,XZ,YZ)
C	(XY,XZ,YZ)	(XY,YZ,XZ)	(XY,YZ,XZ)	(XY,XZ,YZ)
V	(XY,XZ,YZ)	(XY,XZ,YZ)	(XY,YZ,XZ)	(XY,XZ,YZ)

Just as the statement claims, u and v have the XY- and YZ-plane swap places in $G*u*v*u$, while the rest of the graph remains the same.

Proof of Lemma 3.4.5. Define updated flow g' for $v \neq u$ as follows:

$$g'(v) = \begin{cases} g(v) \,\Delta\, g(u) & \text{if } u \in g(v) \\ g(v) & \text{otherwise.} \end{cases}$$

Note that by (g4) or (g5), $g'(v)$ is well-defined if $u \in g(v)$, as also $y \in g(u)$. For $g'(v)$ with $u \notin g(v)$, the correctness of (g0) through (g5) is inherited, so now consider the case where $u \in g(v)$.

In this case, we have $v \prec u \prec g(u)$ by (g1), and by (g2) we have $v \prec \mathrm{Odd}_G(g(v))$ and $u \prec \mathrm{Odd}_G(g(u))$. So even with $g'(v) = g(v) \,\Delta\, g(u)$ we still have (g1) and (g2). As inputs are unchanged, we also still have (g0).

For (g3) through (g5), note we do not have $v \in g(u)$ or $v \in \mathrm{Odd}_G(g(u))$ as that would give a contradicting $u \prec v$. This means that v's inclusion in $g'(v)$ or $\mathrm{Odd}_{G\setminus\{u\}}(v)$ does not change compared to $g(v)$ or $\mathrm{Odd}_G(v)$. Combined with the measurement planes not changing, this means (g3) through (g5) are also still satisfied.

Proof of Lemma 3.4.6. Applying a pivot to $u_1 v$ maintains gflow by Lemma 3.4.2, and turns the non-YZ vertices into non-XY vertices. These can then be deleted by Lemma 3.4.5 without losing gflow.

Proof of Lemma 3.4.7. Let g be the flow before adding the new input i' before the old input i in the original graph. If we update g into g' by adding $g'(i') = \{i\}$, (g1) and (g2) force $i' \prec i$ and $i' \prec N(i)$, which is compatible with what we already had.

Proof of Lemma 3.4.8. Let g be the flow before adding the new output o' after the old output o in the original graph. Then similarly to the above, we can update g into g' by adding $g'(o) = \{o'\}$ without introducing conflicts in the partial order.

For the inverse operation, we have an output o' with only neighbor o, and we want to remove o' from the graph. After the removal, o is an output, so it does not need to have a correction set anymore. As such, the only way the removal of o' can affect the graph, is by changing whether $o' \in g(v)$ and $o \in \mathrm{Odd}(g(v))$ for some unrelated vertex v. This can either require or remove the requirement of (g2). But (g2) is always true, as any output vertex can be taken to be $\prec$-maximal, so this is not a problem.

Proof of Lemma 3.4.9. Let $o_1, o_2 \in O$. As far as gflow is concerned, toggling the edge $o_1 o_2$ only updates odd neighborhoods, so we need to check (g2). (In particular, (g3) through (g5) do not apply as the o_i do not have correction sets.) But (g2) also poses no problem, as output vertices can always be taken $\prec$-maximal.

Proof of Lemma 3.4.10. Consider the proof of Proposition 2.2.18. Before the pivot, we only add outputs (as in Lemma 3.4.8) and add a CZ between outputs (as in Lemma 3.4.9). These operations both maintain gflow. Then, we pivot and remove the two resulting non-XY vertices (as in Lemma 3.4.6), maintaining gflow. Finally, we do the preparation in reverse to get rid of the CNOT itself as we do not care about it, which also maintains gflow. As such, the entire operation maintains gflow.

Proof of Lemma 3.4.11. While the phase updates are interesting, as far as the open graph is concerned, the only change is removal of a YZ-vertex. This maintains gflow by Lemma 3.4.5.

Proof of Lemma 3.4.12. Once again, as far as the open graph is concerned, this is simply an application of Lemma 3.4.5.

Proof of Lemma 3.4.13. The two graphs' gflows and partial orders are completely independent and can be combined without losing any of (g0) through (g5).

Proof of Lemma 3.4.14. Let $(g_1, \prec_1)$ denote the gflow of $(G_1, I_1, O_1, \lambda_1)$ and $(g_2, \prec_2)$ the gflow of $(G_2, I_2, O_2, \lambda_2)$. Assume that all vertices in O_1 are $\prec_1$-maximal. Taking into account the identification, on the concatenated graph $g := g_1 \cup g_2$ satisfies (g0) and (g3) through (g5). Note that g is well-defined as g_1 and g_2 have disjoint domains. We now need to consider the partial order.

On the concatenated graph, define the partial order $u \prec v$ by setting

(i) $u \prec v$ if $u \prec_1 v$ or $u \prec_2 v$;

(ii) $u \prec v$ if $u \in V(G_1)$, $v \in V(G_2)$.

This order is well-defined under the identification. We also claim that this order makes g satisfy (g1) and (g2).

As g_2 is a gflow, there is no vertex $v \in G_2$ whose correction set $g_2(v)$ intersects I_2. The same then holds for g, so by (i), for all vertices in G_2, we get (g1) and (g2). Similarly, for all vertices $v \in G_1$ whose correction set does not intersect O_1, we also have (g1) and (g2).

Now, if $o \in O_1$ intersects $g(v)$, by (ii) we get (g1). The odd neighborhood of $g(v)$ has a part that uses vertices of G_1 and a part that uses vertices of G_2. The part that uses vertices of G_1 satisfies (g2) by (i), and the part that uses vertices of G_2 satisfies (g2) by (ii).

B. Glack update proofs

This appendix contains the proofs Sections 4.1.1 and 4.1.2 omitted.

Proof of Lemma 4.1.5. As a pivot consists of three local complementations, each of which only update the flow part of the certificate by Lemma 4.1.4. The plane changes are exactly the same as in Lemma 3.4.2.

Proof of Proposition 4.1.6. Turning the graph into phase gadget form consists of only local complementations, and pivots. By Lemmata 4.1.4 and 4.1.5, only the flow of the certificate gets changed.

Proof of Lemma 4.1.7. With Lemma 4.1.2, we can assume that u is referenced by at most v_a and v_b, with $u \in g(v_a)$, $u \in \mathrm{Odd}(g(v_b))$, and possibly $u \in \mathrm{Odd}(g(v_a))$. We simply mark v_a as lacking, and remove u from the graph. Similar to the proof of Lemma 3.4.5, all other correction sets can stay unchanged. With this, $(L \cup \{v_a\} \setminus \{u\}, g, \prec)$ is a candidate certificate for $(G \setminus \{u\}, I \setminus \{u\}, O \setminus \{u\}, \lambda)$, but there may exist better certificates.

Proof of Lemma 4.1.8. We prepend XY-vertex i' to input vertex i as a new input. If vertex i is an XY-vertex, the proof is the same as in Lemma 3.4.7, giving us a candidate certificate. Otherwise, it must be lacking, as (g4) or (g5) requires $i \in g(i)$, but inputs may not be put into correction sets.

If i is lacking, we can set $g(i') = \{i\}$ and put i' before everything else in the partial order. Keeping lacking vertices the same, we get a candidate certificate for the same glack in this new graph. Note that i' does not appear in any odd neighborhoods, as i did not appear in any correction sets.

We now claim there is no better certificate. Suppose for a contradiction there is a better certificate, $(L', g', \prec')$. Here, i' may not appear in correction sets, but i can appear in correction sets as it's no longer an input. With Lemma 4.1.2, we can assume this is limited to at most one vertex v_a. If were is no such vertex, this would immediately result in a better certificate for our original graph, so v_a must exist.

If $\lambda(i) \neq \mathrm{XY}$, by (g4) or (g5), we get $v_a = i$. Then $i' \in L'$. As i does not appear in correction sets of other vertices, we can mark i as lacking and remove i' without changing other vertices' correction sets or the glack. This means that our original graph had a better certificate.

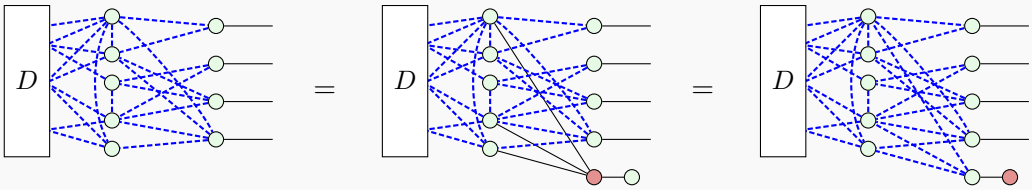
Now consider the case where $\lambda(i) = \mathrm{XY}$. We can assume WLOG that $v_a = i'$, as otherwise we can assign $g'(i') = \{i\}$ and mark v_a as lacking. Again, we can remove i' without changing anything, giving a better certificate and finishing the contradiction.

Proof of Lemma 4.1.9. When appending an output vertex o' to an output o, nothing changes as compared to Lemma 3.4.8, as being able to toggle inclusion of the $\prec$-maximal o in odd neighborhoods does not achieve anything.

If we append an output vertex o' to a lacking vertex ℓ, the old certificate is still a valid candidate certificate, but there may be a better certificate. In particular, when $\lambda(\ell) = \mathrm{XY}$, we can assign $g(\ell) = \{o'\}$, with o' a $\prec$-maximal vertex. This does not introduce cyclic dependencies, and turns one lacking vertex in a regular vertex, so this gives a candidate certificate for a glack of $k - 1$.

Note that undoing this operation by removing the vertex again increases the glack by at most 1 by Lemma 4.1.7. This means that the glack cannot be reduced below $k - 1$.

Proof of Lemma 4.1.10. The update is an application of the π-copy rule (with zero phase) and some color changes:



In the above example, A consists of three arbitrary vertices in the middle column. The semantic meaning is maintained.

We also need to show glack changes as in the statement. The gflow conditions do not care about the post-selection, and only see the new output vertex o that does not need a correction set. By doing nothing but setting o to be $\prec$-maximal, we obtain a candidate certificate for a glack of k for the resulting graph. However, there may be another, more optimal certificate. However, similarly to the previous proof, the glack cannot be reduced below $k - 1$, because undoing this addition can increase glack by at most 1.

Proof of Lemma 4.1.11. As lacking vertices do not have correction sets and can be taken to be $\prec$-maximal, the proof of Lemma 3.4.9 works exactly the same, and tells us that a certificate stays valid as a candidate certificate. Note that like local complementations, this operation is self-inverse. This means that glack does not change, with the same line of reasoning as in the proof of Lemma 4.1.4, and our candidate certificate is a proper certificate.

Proof of Lemma 4.1.12. The proof of Lemma 3.4.13 works exactly the same, and we obtain a candidate certificate for the result. As the two graphs are completely unrelated, if there was a better certificate than this candidate, it would mean one of the original certificates was not optimal, which is a contradiction.

Proof of Lemma 4.1.13. The proof of Lemma 3.4.14 works exactly the same, and gives us a candidate certificate. However, this certificate may not be optimal.

Proof of Lemma 4.1.14. Let $(L, g, \prec)$ be a certificate where $u \notin L$ or u is $\prec$-minimal. For $v \in \overline{O \cup L}$ we can copy the flow of the proof in Lemma 3.4.5 and define

$$g'(v) = \begin{cases} g(v) \,\Delta\, g(u) & \text{if } u \in g(v) \\ g(v) & \text{otherwise.} \end{cases}$$

If $u \notin L$, both $g(u)$ and $g(v)$ are well-defined. If otherwise u is $\prec$-minimal, there are no v for which $u \in g(v)$. In either case, $g'(v)$ is well-defined, and the proof that we obtain valid correction sets goes exactly the same as in Lemma 3.4.5. This gives a candidate certificate $(L, g', \prec)$ for $G \setminus \{u\}$.

Note that if u is lacking and $\prec'$-minimal in some certificate $(L', g'', \prec')$, it does not appear in any correction sets or odd neighborhoods, allowing us to instead take $(L' \setminus \{u\}, g'', \prec')$ as candidate certificate, for some flow g''.

We now want to show that we can only improve this certificate as stated in the lemma. To that end, consider $v \in L \setminus \{u\}$. In most cases, this vertex will stay lacking – the above does not influence (g0), (g3), (g4), or (g5). We also do not need to consider (g1): before we removed u, whether $u \in g(v)$ or not was a choice that *could* help make v non-lacking, but did not. Removing u only results in this choice being made for us.

However, (g2) does change significantly, as removing u changes the possibilities for odd neighborhoods. In particular, if we have $u \prec v$ and v was denied a correction set $g(v)$, only because $u \in \mathrm{Odd}(g(v))$ for some set $g(v)$ that satisfies all other requirements (g0) through , this set now has a correction set in this certificate. This is the only mechanism in which v can turn into a non-lacking vertex, and gives us the requirement in the lemma.

Proof of Lemma 4.1.15. In the proof of Lemma 3.4.10, all operations maintained gflow. In the glack case, we use Lemmata 4.1.9, 4.1.11, and 4.1.14. The first two only change the flow, but the third one is a bit more interesting.

The two vertices we delete are neither lacking nor $\prec$-minimal in any certificate, as we would otherwise have a better certificate for the original graph. These two vertices' future contains only have vertices not in the original graph. This means that applying Lemma 4.1.14 satisfies the tightness condition and our case is the second bullet point. This means that like the other two lemmata, the only thing that changes is the flow.

Proof of Lemma 4.1.16. In order to apply Lemma 4.1.14, we need to show that there is some certificate in which u (or by symmetry, v) is not lacking. Let $(L, g, \prec)$ be an arbitrary certificate. Unless both u and v are in L, we are done, so assume $u, v \in L$. We will use this certificate to define a new certificate $(L', g', \prec')$ with $|L'| < |L|$ to contradict L's minimality.

The approach is simple: let g' be the same as g, except that we additionally define $g'(u) = \{u, v\}$. Because u and v share their neighborhood, $\mathrm{Odd}(g'(v)) = \varnothing$. We need to show that this additional relation $u \prec' v$ is compatible with $\prec$. As $v \in L$, we can assume v is $\prec$-maximal, so this poses no problem. This means we can define $\prec' = (u, v) \cup \bigcup_{(w_1, w_2) \in \prec} (w_1, w_2)$, and we are done with showing there is a certificate in which u is not lacking.

This means we can apply Lemma 4.1.14 and get a graph with a glack of at most k.

Proof of Lemma 4.1.17. We want a certificate to show $u \notin \mathcal{L}$. By Proposition 4.1.2, we may take a certificate in which at most two vertices v_a and v_b exist with u in $g(v_a)$ or $\text{Odd}(g(v_b))$. If u is lacking and $v \prec v_a \prec u$ or $v \prec v_b \prec u$, we can make v lacking so that we do not have $v \prec u$ any longer. In either case, we can now assign $g'(u) = \{u\}$, which gives us a certificate in which u is not lacking.

Then by Lemma 4.1.14, we can remove this vertex and the glack and certificate update as stated. Now we will show that if the neighbors are all XY-vertices, the inverse operation cannot increase glack, giving a glack of exactly k.

In this case, the inverse operation is adding a YZ-vertex u to XY-vertex v. As $v \notin g(v)$, and the future of v does not refer to v either, we can safely assign $g'(u) = \{u\}$ without introducing cyclic dependencies in $\prec$.