The ZX calculus and PyZX

Tomas Sousa
A81411

Marco Barbosa
A81428

February 2021

Chapter 1

Introduction

In theoretical computer science the use of diagrams for studying processes has been widely adopted. In recent years interest has grown in applying these same techniques to quantum processes, in this report we will introduce ZX-Calculus, a calculus for reasoning with such processes. We will then present PyZX, a tool for doing optimization and verification of quantum circuits. After this we demonstrate the results of applying this tool on some ”real world” quantum circuits.

Chapter 2

Behind the gates there are spiders

After working with quantum circuits for a while, we start to get an intuitive understanding of the relationship between the quantum gates and the mathematical formulas describing their behaviour. This hints at a deeper relationship between diagrams and formulas. It turns out that if the diagrams and its elements are assigned precise meanings we can derive rules that allow us to manipulate the visual representation of the circuit.

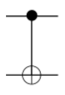
But before diving into that let's first make a quick recap:

Circuits and Gates

We usually work with the quantum circuit model of computation. In this model, computation is represented by gates, usually seen as unitary transformations acting on qubits.

In circuits, qubits are represented by wires. We can compose gates horizontally(on the same wire) or vertically(gates acting on different qubits). There are also gates that can take multiple qubits as input.

One very important result is the existence of sets of universal quantum gates. These are sets of gates, that can be used as building blocks for any other gate. One such set is the CNOT gate, the Hadamard gate and the Z_α [1]



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

Figure 2.1: CNOT

[1] Z_α is usually called R_ϕ, but we chose tu use the same notation as the PyZX creators

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

Figure 2.2: H

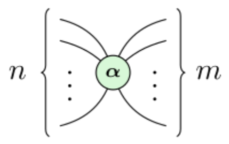
$$-\boxed{Z_\alpha}- = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & e^{i\alpha}\end{pmatrix} = |0\rangle\langle 0| + e^{i\alpha}|1\rangle\langle 1|$$

Figure 2.3: Z_α

What about the connection between the circuit and the mathematical structure modelling the physical behaviour of the qubits we talked about earlier? To achieve this we must have a formal definition of what the quantum circuits are, this is, we need a diagrammatic language that represents quantum circuits.

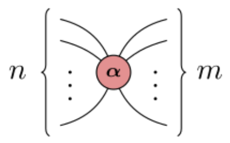
One such language is ZX-Calculus. ZX diagrams are comprised of wires and spiders. The diagrams are read from left to write, open wires on the left side represent *inputs* and those on the right side represent *outputs*. Spiders are linear operations with any number of input/output wires, if we want to be more formal we could say that spiders are processes. Connecting *output* wires of one diagram to the *input* wires of another represents composition, while stacking gives the tensor product.

Spiders can be either Z-spiders(green) or X-spiders(red).



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

(a) Z spider



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

(b) X spider

What we need at this point is to find a way to represent the 3 universal gates shown above in terms of spiders. If this is done we can turn any circuit into a ZX diagram by decomposing it's gates in terms of the universal ones, and then converting them to spiders.

The Z_α gate is trivial, it is simply a Z spider with one input and one output.

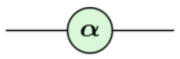


Figure 2.4: Representation of Z_α as diagram

We can represent the CNOT gate with the following diagram:

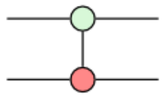


Figure 2.5: CNOT diagram

The Hadamard gate can be written in terms of rotations as:

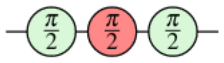


Figure 2.6: Diagram representing Hadamard gate in terms of spiders

Since, this representation can make diagrams harder to read the Hadamard gate is usually written as a yellow square:

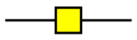


Figure 2.7: Simplified diagram of Hadamard gate

We can represent X_α as:

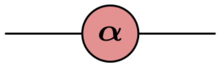


Figure 2.8: Representation of X_α as a diagram

Or using Z spiders and Hadamard as:

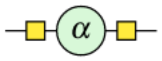


Figure 2.9: X_α in terms of Z spiders and Hadamard

We an also write a controlled Z gate as:

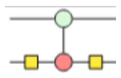


Figure 2.10: CZ represented as a diagram

We can build more complex diagrams, by composing these basic building blocks. Two diagrams are said to be equal if they can be deformed to the other. Besides this, there are also some *rewrite rules*, that establish equality between diagrams that would otherwise be considered different. The set of *rewrite rules* used by PyZX is:

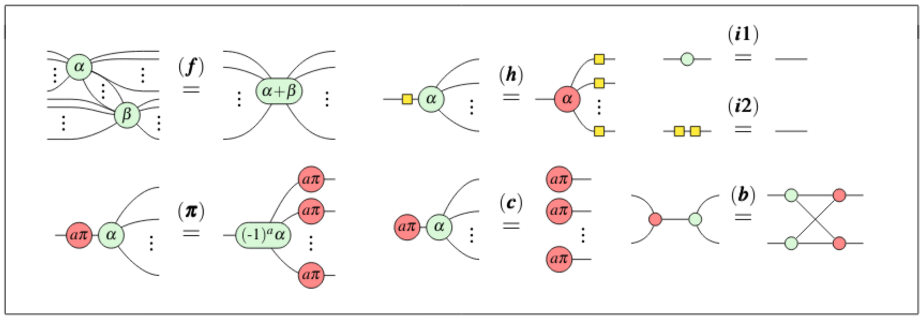


Figure 2.11: Rewrite rules of ZX Calculus

We could for example use these rules to manipulate the diagram representing the CZ gate. By applying ***(h)*** in reverse, obtaining the following diagram:

Looking at this diagram we might be left wondering how can what gates do we need to use to create a circuit that directly matches this diagram, in fact it has no match in terms of circuits. This illustrates a key point about ZX-diagrams: although every circuit has a diagram representation, not all diagrams represent circuits.

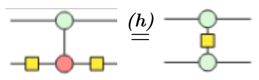


Figure 2.12: Application of rule ***(h)***
to CZ diagram

At this point we might be tempted to think this is all nonsense, after all we may be able to use these simplification rules and obtain a very nice and compact ZX-diagram, but what's the interest in that if we can't produce any circuit from the result? At the end of the day what we are interested about is running our algorithms on quantum computers, and those can only run quantum circuits. This means the simplification is only interesting if we are able to produce a simpler more compact circuit at the end.

This is were the ingenuity of the PyZX creators comes into play[2]. They realized that we can come up with a set of rewrite rules that are complete for Clifford gates[3].

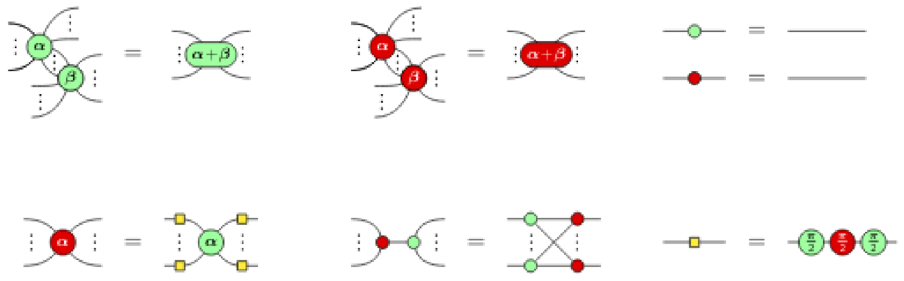


Figure 2.13: ZX Caculus rules that are complete for Clifford diagrams

By choosing to apply these in rules in a way that allows for the gate count to be minimized[2][5], they extract what is called a skeleton. This is a simplified ZX-diagram with Clifford gates reduced to the minimum. Coming from a circuit we know the ZX-diagram must represent a unitary linear operation. The problem now becomes that of extracting a circuit from this so called skeleton.

This problem is in principle hard to solve. Although if the diagram has *generalized flow(gFlow)* we can use a clever strategy to solve it in reasonable time. It turns out that circuits have *gFlow*, and the rewrite rules chosen preserve *gFlow*. This means we can always get a simplified circuit in reasonable time by applying this strategy to the obtained skeleton.

After all, this whole talk about ZX-Calculus wasn't in vain. We can at the end of the day, use it to obtain simpler circuits, which will be easier to run and less prone to errors.

In the next section, we will discuss PyZX in more detail, focusing on implementation and some features.

Chapter 3

PyZX

PyZX[1][4], pronounced as ”Pysics” is a Python-based library designed to handle large quantum circuits using diagrammatic calculus as previously explained in this report. This Open Source Software aims to reduce the complexity and improve reasoning with ZX-diagrams. The project is hosted on GitHub and available at Quantomatic Github repository, and can be used in a Python document or can be called as a command-line tool for circuit-to-circuit optimization, for more detail see full Documentation.

Throughout this section, we will give a brief overview of the features of PyZX as well as some possible scenarios where this tool shows great results.

3.1 What can PyZx do?

This powerful tool offers several built-in features that allow users to manipulate quantum circuits. PyZx allows for circuit, graph, ZX-diagram visualization, and manipulation in QASM, QC, and Quipper formats, allowing even for real-time Z-diagram visualization. With PyZX it's possible to convert these ZX-diagrams into Numpy tensors. Besides all these manipulation features it also gives some unique rewrite and simplification strategies as well as some optimization functions.

Circuits

A Circuit consists of a list of Gates, which in turn are small classes containing some information about the gate and how to convert it into various representations, such as ZX-diagrams or the QASM format.

This class allows for importing and exporting circuits to and from PyZX. Providing methods to do gate-level operations, such as converting gates or taking their adjoint.

ZX-diagrams

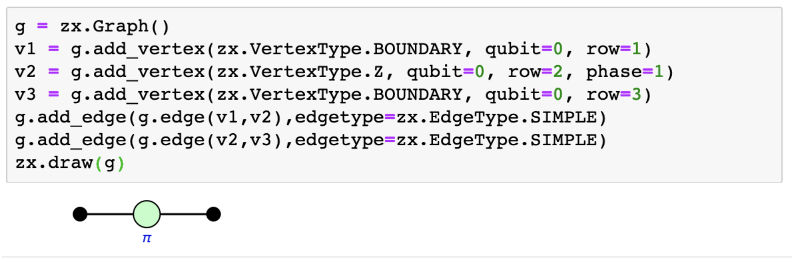
ZX-diagrams are represented in PyZX by instances of the Base Graph class.

The graphs in PyZX are undirected graphs with typed vertices and edges. There are 4 types of vertices:

- Boundaries;
- Z-spiders;
- X-spiders; H-boxes;

Boundary vertices represent an input or an output to the circuit and carry no further information. Z- and X-spider is the usual bread and butter of ZX-diagrams. H-boxes are used in ZH-diagrams as a generalization of the Hadamard gate.

A simple example could be:



```
g = zx.Graph()
v1 = g.add_vertex(zx.VertexType.BOUNDARY, qubit=0, row=1)
v2 = g.add_vertex(zx.VertexType.Z, qubit=0, row=2, phase=1)
v3 = g.add_vertex(zx.VertexType.BOUNDARY, qubit=0, row=3)
g.add_edge(g.edge(v1,v2),edgetype=zx.EdgeType.SIMPLE)
g.add_edge(g.edge(v2,v3),edgetype=zx.EdgeType.SIMPLE)
zx.draw(g)
```

Figure 3.1: Simple Graph

Edges in a PyZX graph can be of 2 kinds:

- Default edge (representing a regular connection);
- Hadamard-edge (connection between vertices with a Hadamard gate applied between them);

As ZX-diagrams allow parallel edges between spiders and self-loops. In PyZx is possible to deal with this by adding an edge where there is already one resent.

Often, it is more convenient to use the rules of the ZX-calculus to resolve parallel edges and self-loops whenever a new edge is added.

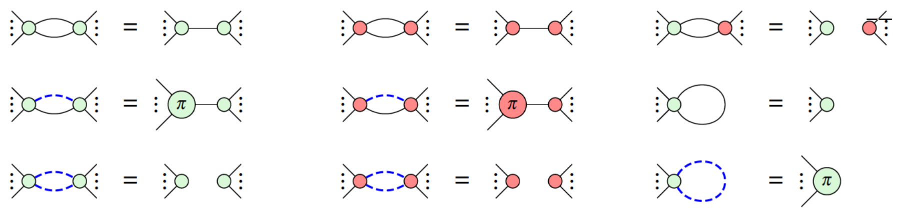


Figure 3.2: Caption

In order to use this built-in functionality to resolve the edges, one can use the Graph method add edge table, which takes in a list of edges and edge-types, and resolves what the ZX-diagram would look like when all the double edges and self-loops have been resolved.

Rewrite Rules

This module contains all the rewrite rules on ZX-diagrams in PyZX.

Each rewrite rule consists of two methods: a matcher and a rewriter.

The **matcher** finds as many non-overlapping places where the rewrite rule can be applied. The match functions take Graphs as inputs and an optional "filter" in order to specify the vertices or edges according to the filter.

The **rewriter** takes in a list of matches and performs the necessary changes on the graph to implement the rewrite.

Simplification

This module contains the ZX-diagram simplification strategies of PyZX, which are built-in hierarchically, each strategy is based on applying some combination of the match and rewrite rules.

There three main types of simplification strategies *identity removal*, *basic simplifiers*, *compound simplifiers*.

In the *identity removal*, the matcher searches for vertices that have phase zero and exactly two neighbors. When this happens the matcher makes a match that contains the neighbors and the type of edge that should be made. Then the rewriter takes the list of matches and builds a list of vertices to be removed, and edges to be added, and applies this all at once.

The *basic simplifiers* are built based only on one rule. These types of simplifiers apply the matcher and the rewriter iteratively until it stops finding matches. Nevertheless, this could lead the simplifier to enter an infinite loop in order to prevent this is important that the rule actually simplifies the diagram in some manner. Normally in PyZx the way to maintain this state is by evaluating if the number of vertices in the graph. If the number of vertices is reduced, every iteration of the loop, then the simplifier is running properly.

Finally, the more complex simplifier is the *compound simplifier*, these basically mix the other type of simplifiers in some particular order potentially looping over this combination until none of the simplifiers finds any more reductions.

In this module of PyZx the main procedures of interest are:

- clifford_simp (for simple reductions);
- full_reduce (for a full rewrite);
- teleport_reduce (for a full rewrite as the full reduce without changing the structure of the graph);

Optimization

This module implements several optimization methods on Circuits. Functions such as:

- basic_optimization (to run a set of back-and-forth gate commutation and cancellation routines);
- phase_block_optimize (does phase polynomial optimization using the TODD algorithm);
- full_optimize combines these two methods;

Verification

At the end of the day, what is really important is that the manipulations made to the ZX-diagram preserve its properties and is semantics.

PyZX offers a straightforward way to do this by allowing ZX-diagrams to be converted into their underlying linear maps using NumPy. To test the equality of diagrams we can then simply test whether all the elements in the two tensors are equal up to a global non-zero scalar. Nonetheless, this uses an exponential amount of memory in terms of qubits, and this could be a drawback for PyZx use.

But a clever way to verify is by using the engine itself to validate equality. Considering c1 to be the circuit and c2 to be the adjoint of c1, then if we combine the two in one circuit c and perform a full simplification we should end up with the identity circuit. The function $verifyequality$ implements this idea.

If this method fails to reduce the circuit to the identity, we say that the method is inconclusive and have to try other as is not possible to infer that the simplification is incorrect.

Bibliography

[1] URL: https://pyzx.readthedocs.io/.

[2] Ross Duncan et al. "Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus". In: *Quantum* 4 (2020), p. 279.

[3] Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. *A Complete Axiomatisation of the ZX-Calculus for Clifford+T Quantum Mechanics*. 2018. arXiv: 1705.11151 [quant-ph].

[4] Aleks Kissinger and John van de Wetering. "PyZX: Large Scale Automated Diagrammatic Reasoning". In: *Electronic Proceedings in Theoretical Computer Science* 318 (May 2020), pp. 229–241. ISSN: 2075-2180. DOI: 10.4204/eptcs.318.14. URL: http://dx.doi.org/10.4204/EPTCS.318.14.

[5] Aleks Kissinger and John van de Wetering. "Reducing T-count with the ZX-calculus". In: *arXiv preprint arXiv:1903.10477* (2019).