QPC Practical: MBQC in Quantomatic

Aleks Kissinger and John van de Wetering*

June 11, 2018

1 Introduction

There are various models of quantum computation. In the lectures, we mostly discussed the *circuit model*, where quantum computation is performed in four steps:

1. Prepare some collection of qubits in an initial state.
2. Perform unitary *quantum gates* on these qubits.
3. Measure the result.
4. (Possibly) perform some classical post-processing.

In this model, steps 1 and 3 are basically fixed, whereas step 2 is where the interesting, quantum stuff happens. Hence, in this picture, the 'quantum program' is completely represented by a circuit diagram. However this is not the only paradigm for quantum computing. Another choice is *measurement based quantum computing* (MBQC). In this model, computation consists of three steps:

1. Prepare some qubits in a fixed *resource state*, which is typically highly-entangled.
2. Perform a series of single-qubit ONB measurements, *where measurements are chosen based on prior measurement outcomes.*
3. (Possibly) perform some classical post-processing.

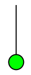
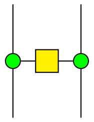
Notably, step 1 is fixed, so all of the interesting, quantum stuff happens in step 2. Hence, a 'quantum program' in MBQC consists of a list of single-qubit measurement bases, where each new basis choice is allowed to depend on choices that came before. Perhaps surprisingly, there exist MBQC schemes which are just as powerful as the circuit model. That is, they are *universal for quantum computing.*

In this practical, we will study one such model, called the *one way model of quantum computing* (1WQC). The resource states used in the one-way model are called *graph states*, as presented in Section 9.4.5 of *Picture Quantum Processes*. We can conveniently describe the preparation of a graph state as a composition of the following two operations:

1. N_i prepare a new qubit in the $|+\rangle$ state at position i.
2. $E_{i,j}$ entangle the i-th qubit to the j-th qubit using a controlled-Z gate.

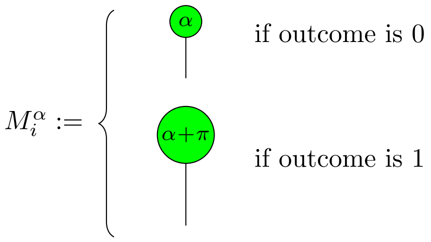
These can be depicted as ZX-diagrams as:

[*] ...with special thanks to Alex Merry for helping prepare a previous version of this document

$N_i :=$  $E_{i,j} :=$ 

Note we will adopt Quantomatic's convention of writing green dots as Z-spiders and red dots as X-spiders.

We then consider single-qubit measurements on the equator of the Block sphere, i.e. ONB measurements of this form:



1WQC programs can be written as composition of these operations. To mirror the order in which the associated linear/quantum maps compose, they are conventionally read from right-to-left. For example:

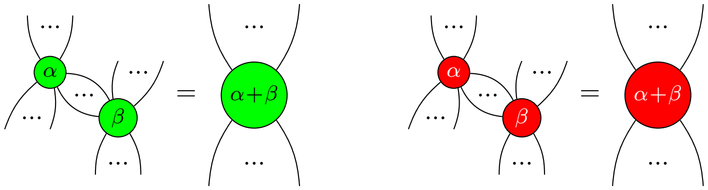
$$M_2^0 M_3^0 E_{1,3} E_{2,3} E_{3,4} N_3 N_4$$

This program prepares ancillae at 3 and 4, entangles 3 to 4, 2 to 3, and 1 to 3, then measures qubits 2 and 3 at angles 0 and 0.

Note that measurements are non-deterministic, so either of the two outcomes shown may happen when we measure a qubit. The main point of MBQC is we can recover *deterministic* computation by first performing measurements, then performing *corrections* later by applying unitaries or adjusting the angles at which we perform later measurements. We'll see more about corrections later.

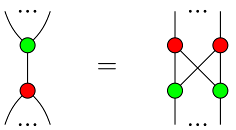
2 The ZX Calculus

We will analyse the one-way model using the ZX-calculus. As we saw in the book, there are several equivalent ways to present the ZX calculus. For us, it will be convenient to use a presentation using the spider laws:



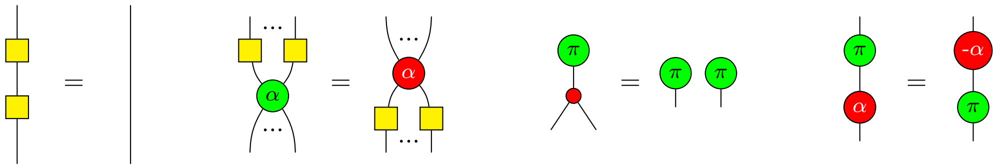
(1)

strong complementarity:



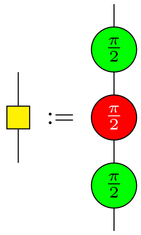
(2)

and four additional rules:



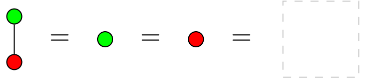
(3)

where the yellow boxes are Hadamard gates:



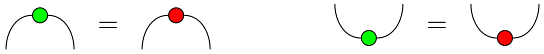
It was shown in the book that the last 2 rules in (3) actually follow from the rest (for α a multiple of $\frac{\pi}{2}$), but for simplicity we have decided to just take them as given.

We also assume a couple of "garbage collection" rules to eliminate non-zero scalars:



(4)

We will be working with undirected edges throughout, which makes implicit use of two more equations, namely that the caps and cups of the two colours coincide:



3 Quantomatic Basics

We will implement a fragment of the Z/X calculus in Quantomatic as a *graph rewrite system*. The difference in a rewrite system and a set of graphical axioms is that equations $G = H$ are replaced with *reduction rules* $G \mathrel{-\!\!\rhd} H$. First, we'll see how to input graphs and rules into Quantomatic, and how to express collections of edges with ellipses using graph patterns.

Disclaimer: Quantomatic is under constant development, so while we try to keep things as stable as possible, you may encounter some weird behaviour. Save often and let us know if you encounter a problem so we can file a bug report.

Get Quantomatic and the QPC project by going to this link:

https://github.com/Quantomatic/sample-projects/releases/tag/qpc2018

You need to download `Quantomatic.jar` and `qpc.zip`. Unzip `qpc.zip` on your computer and launch Quantomatic. If a recent version (≥ 8) of Java is installed properly, you should be able to run Quantomatic by double-clicking the JAR file or running the following command:

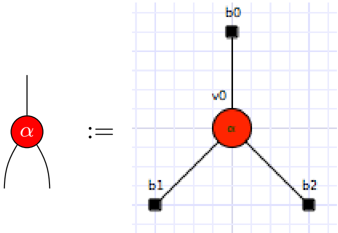
```
$ java -jar Quantomatic.jar
```

Open the QPC project by clicking "File > Open Project...", navigating to the project folder and opening `qpc.qproject`. Click on "File > New Graph" to open the graph editor.

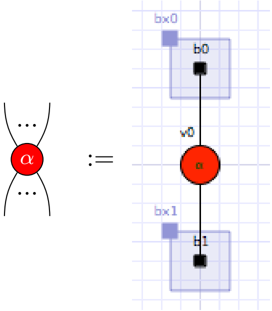
You will now be looking at a blank graph. Add some vertices by selecting the Add Vertex tool from the toolbar. Select the type of vertex from the dropbox on the bottom and click where you'd like to place them. If the Select tool is highlighted, you can double-click vertices to set phases. Note that it will recognise LaTeX symbols for Greek letters (`\alpha`, `\beta`, etc.).

Add edges by selecting the Add Edge tool from the toolbar. Click-and-drag from source to target. Quantomatic supports both directed and undirected edges, which can be toggled via the "directed" checkbox on the bottom. **Note that we will be using undirected edges throughout this practical.**

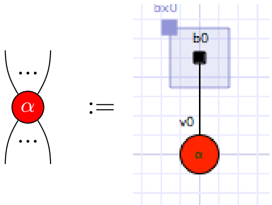
Boundary vertices provide named inputs and outputs for a graph, corresponding to dangling wires in the graphical calculus. Add them using the Add Boundary tool.



Graphs with variable inputs and outputs (like spiders) can be represented using *bang-graphs*. These graphs indicate sub-graphs which can be repeated many times by using *bang-boxes* (by analogy to the “bang” operation !(−) from linear logic). To put vertices or boundaries in a bang-box, select the Add Bang Box tool, and click and drag a box around the vertices you wish to “bang”. We can represent a spider of any arity as follows:



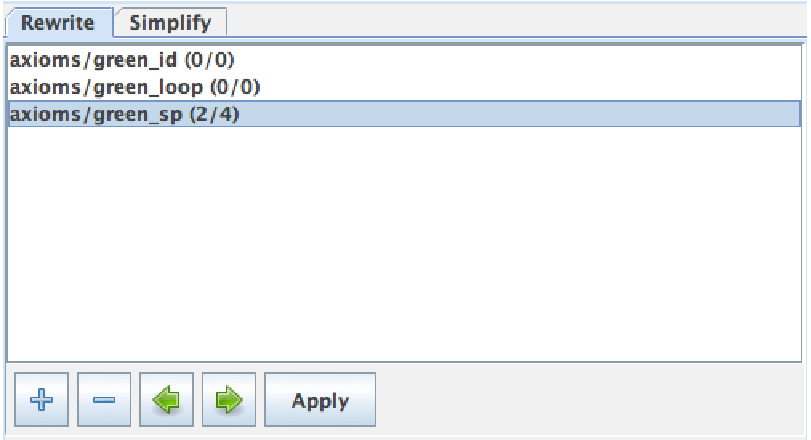
In fact, the since we will be working with undirected edges, the second !-box is redundant. We could just as well have written:



Note that boundaries and bang-boxes both have names, which are determined based on the order they are created. This becomes important when defining rules. To define a rule, select “File > New Axiom” from the menu. A valid rule must have:

1. the same number of boundary vertices and bang-boxes on both sides,
2. the same *names* for boundary vertices and bang-boxes on both sides, and
3. a *compatible boxing* of boundary vertices. That is, if a boundary vertex named “b3” is in the box named “bx1” on the LHS, it must also be in the box named “bx1” on the RHS and vice-versa.

Once you have created some rewrite rules, you can use them to build derivations. First, create a graph that will be the starting point for the derivation, then select “Derive > Start Derivation”. You will now see the Derivation Editor. Performing single rule applications is done via the Rewrite Panel:



axioms/green_id (0/0)
axioms/green_loop (0/0)
axioms/green_sp (2/4)

Add some rules by clicking the "+". By default, they will be applied left-to-right, but checking "inverse" in the Add Rule dialogue will add a version that goes from right-to-left. As soon as a rule is added, occurrences of the LHS will be searched for automatically. Use the arrow buttons to cycle through them, then click "Apply" to apply the desired rule. To restrict the search to a subgraph, select some vertices on the left.

Automatic (many-step) rewriting is done via simplification procedures, or *simprocs*. These are little pieces of Python code that define a rewrite strategy. To create a simproc, create a new Python script with "File > New Python Script" and save it in e.g. the `simprocs` folder. The simplest kind of simproc loads a ruleset, and applies rules repeatedly until nothing is applicable. For example, if we have rules in the `axioms` folder called `red-sp.qrule` and `green-sp.qrule` which fuse red and green spiders, respectively, this will apply those two rules as much as possible:

```
from quanto.util.Scripting import *

simps = load_rules(["axioms/red-fusion", "axioms/green-fusion"])
simproc = REDUCE(simps)
register_simproc("spider-fusion", simproc)
```

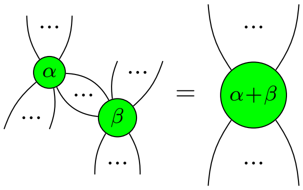
The first line loads some useful functions. The second loads a list of rules by giving their paths in the project (minus the ".qrule" extension). The third creates a new simproc, using the `REDUCE` strategy, and the fourth registers this simproc with the name "`spider-fusion`" so Quantomatic will be able to access it.

Load a simproc into Quantomatic by opening the Python script where it is defined and clicking the play button in the toolbar. This will now be available under the "Simplify" tab in the derivation editor, though you may need to click refresh.

Once a derivation is complete, it can be exported as a theorem by clicking "Export Theorem" in the toolbar of the derivation editor. Note that the derivation should be saved first.

Exercise 1: Implementing the Spider Laws

The spider theorem for green dots looks like this:



Rather than implement the spider theorem as a single reduction rule, we can implement it as:

1. A rule that merges dots of the same colour **connected by a single edge**.
2. A rule that removes self-loops.
3. A rule that turns spiders with 1 input and 1 output (and 0 phase) into identities.

Note the the first and second rule need to handle any number of inputs and outputs on the spiders. Bang-boxes can be used for this.

Implement the spider laws for red and green dots as rules (6 in total) in Quantomatic. Give them the following names:

- `axioms/green-fusion.qrule`
- `axioms/green-loop.qrule`
- `axioms/green-id.qrule`
- `axioms/red-fusion.qrule`
- `axioms/red-loop.qrule`
- `axioms/red-id.qrule`

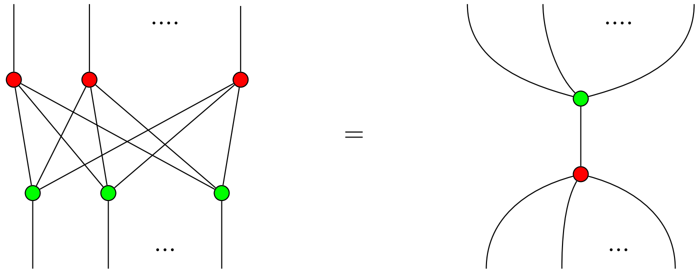
Use these rules to simplify a graph consisting of red and green spiders, as well as some phases to a normal form. You can do this manually, or by adding your new rules to the simproc defined in `mbqc.py`.

To get you started on Quantomatic, here are a few tips:

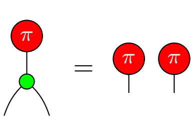
- *A spider's type can be changed from Z to X or vice-versa by CTRL+double-clicking it.*
- *Rules are sensitive to the names of boundaries and bang-boxes. It is often easiesr to build the LHS, copy and paste to the RHS, and only modify the parts that should change.*
- *Nodes can be added/removed from bang-boxes. With the bang-box tool selected, click the upper-left corner of a bang-box and drag to the node you wish to add or remove.*

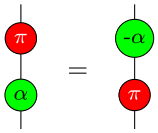
Exercise 2: Proving some theorems using the ZX-calculus

After that warmup, it's time to prove some theorems. Note that the rest of the ZX rules are already implemented in the `axioms` folder of the `qpc` project. Probably the most interesting rule is `strong-comp`, which states that



Using the color change rules, prove the following two "colour-reversed" versions of the ZX rules:





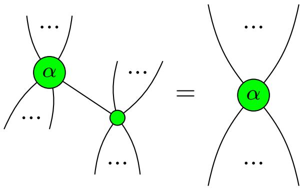
Save the derivations as:

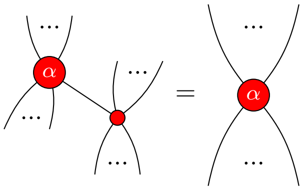
- `derivations/red-pi-copy.qderive`
- `derivations/red-pi-commute.qderive`

Export these as the following theorems:

- `theorems/red-pi-copy.qrule`
- `theorems/red-pi-commute.qrule`

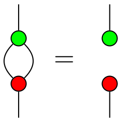
Next, we can prove a couple of well-known consequences of the ZX-calculus. Along the way, you'll need to use the inverse version of some rules. For the spider laws, these less-general versions are easier to work with in reverse:

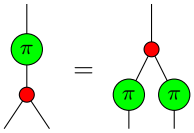




(5)

Prove the simplified spider laws and export them as theorems. Then, prove the Hopf law and this generalised π-copy law:



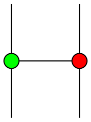


Use the following names for the associated `.qderive` and `.qrule` files:

- `green-fusion-simple`
- `red-fusion-simple`
- `hopf`
- `pi-map-copy`

Exercise 3: The CNOT Gate Using MBQC

It is a well-known result that single-qubit unitaries and the CNOT gate are universal for quantum computing. In the Z/X calculus, we can write the CNOT gate as:



(6)

Recall the MBQC program from section 1:

$$M_2^0 M_3^0 E_{1,3} E_{2,3} E_{3,4} N_3 N_4$$

Use the translation from Section 1 to write this program as a graph in Quantomatic, assuming measurement outcomes $\{M_2^0 \to 0, M_3^0 \to 0\}$. Rewrite the measurement pattern into (6). Save the associated derivation as:

- `derivations/cnot-00.qderive`

Even in the cases where we don't get outcome 00, we can still produce a CNOT gate by performing single-qubit corrections.

For each of the other three outcomes, find red and green phases to apply to qubits 1 and 4 to produce a CNOT gate. Show that all four cases reduce to (6) in Quantomatic. Save the associated derivations as:

- `derivations/cnot-01.qderive`
- `derivations/cnot-10.qderive`
- `derivations/cnot-11.qderive`

Let $M_{i,A}^{\alpha}$ be a measurement labelled A of the i-th qubit at angle α. Let Z_j^A be the identity for the outcome 0 of the measurement labelled A and a Z_π gate for outcome 1. Define X_j^A similarly. These are called *conditional corrections*.

Using conditional corrections, write (on paper) an MBQC program that produces the CNOT gate, regardless of the measurement outcomes.

Exercise 4: Implementing Single Qubit Unitaries

Any single qubit unitary can be written using three phase gates, $U = Z_\alpha X_\beta Z_\gamma$.

Write an MBQC program that implements an arbitrary single qubit unitary when the measurement outcomes are all 0. Save this derivation as:

- `derivations/single-qubit-0000.qderive`

Bonus: Completing the Universality Proof

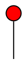
BONUS: Add conditional corrections (and/or conditionally adjust measurement angles) such that for all measurement outcomes, one obtains a single-qubit unitary. Show that this works for a few example cases, named e.g.

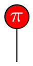
- `derivations/single-qubit-0010.qderive`
- `derivations/single-qubit-1010.qderive`
- ...

This idea that measurement angles can be chosen based on previous measurement outcomes is known as *feed-forward* in MBQC. The existence of these two programs shows that the one-way model can implement CNOT gates and any single qubit gate. It is therefore as powerful as the circuit model, i.e. universal for quantum computing.

The M_i^α operations are measurements in the XY-plane. Another operation that is sometimes considered in the measurement calulus is:

$M_i^Z :=$

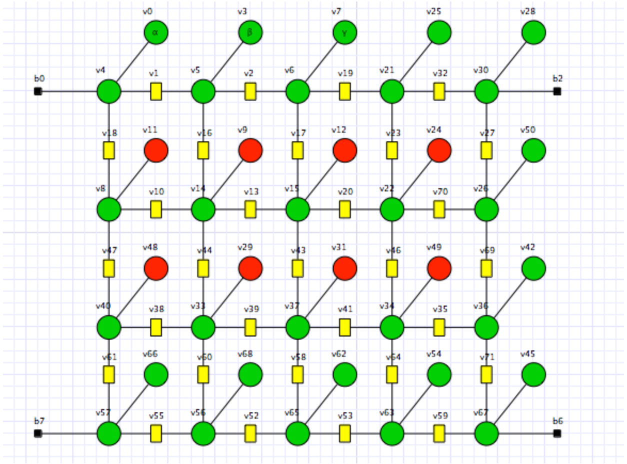
 if outcome is 0



if outcome is 1

These Z-measurements allow one to “cut holes” in a graph state, which can be used to trasform some generic shape (e.g. a grid) into one with the desired topology.

BONUS: Show that the following sequence of measurments reduces to a quantum circuit on two qubits:



What circuit is it?