

12

Quantum Computation

In the Name of the Pasta, and of the Sauce, and of the Holy Meatballs ...
– *Bobby Henderson*, *The Gospel of the Flying Spaghetti Monster*, 2006

After the conceptual comes the practical. While the study of quantum foundations is as old as quantum theory itself, the field of quantum computing is relatively new. So new in fact that large-scale, practical quantum computing is still not a reality. A typical ‘quantum computer’ takes many months to set up before performing such astounding tasks as factoring 6 into 3×2 . Nonetheless, if those machines would exist, we know that we would gain amazing speed-ups in solving some hard (classical) computational problems, such as those involved in breaking a huge portion of cryptographic systems in use today.

Before we get into ‘quantum computing’, we should say a couple of things about ‘computing’. What is computing? Our answer by now probably won’t come as such a shock: it’s a process theory! Computation is indeed all about wiring the inputs and outputs of small processes together to make bigger processes. More specifically, a *computation* consists of a (finite) set of basic processes, which are wired together according to some (also finite) instructions, which we refer to as an *algorithm* or simply a *program*.

The only essential difference between classical and quantum computation is the contents of the basic processes. For classical computation, these operations consist of things like logical operations (e.g. XOR) or reading/writing locations in memory. For quantum computation, we can extend this with quantum processes and classical-quantum interactions such as measurements. So quantum computing is all about figuring out how to write new kinds of programs that exploit these new building blocks to build faster algorithms or accomplish new kinds of tasks that aren’t possible classically.

The first quantum algorithms were ‘proofs of concept’, in the sense that they solved some problem much faster than a classical computer, but the kinds of problems they solved were not particularly interesting in their own right. However, this changed drastically with the advent of Grover’s *quantum search* and Shor’s *factoring* algorithms. The latter, which demonstrated the application of quantum computers to efficiently factor large numbers, is enormously interesting for one big reason: a huge percentage of the current cryptography

in use today relies on a cryptographic system called RSA, which in turn depends on the fact that it is computationally infeasible to factor large numbers. What's more, pretty much every alternative to RSA (except for a handful of so-called post-quantum systems) relies on the closely related problem of computing 'discrete logarithms'. But both factoring and this problem can be solved efficiently as special cases of the *hidden subgroup problem*, which we lay out in Section 12.2.4. So, once there are quantum computers around, the padlock on your web browser meaning your bank details are being encrypted becomes effectively meaningless (though a quantum hacker is probably more interested in other things besides the size of your overdraft).

In this chapter, we'll combine the classical-quantum building blocks we have already seen into computations. There is in fact not just one way to do this, but many possible *models of quantum computation*. We will focus on two such models, the first being the *quantum circuit model*, which extends the classical concept of computing with circuits to quantum processes in the more or less obvious way. Rather than starting with a bunch of bits and performing classical logic gates, we start with a bunch of qubits, perform a bunch of quantum gates, then measure what comes out.

The second and markedly funkier model is called *measurement-based quantum computation* (MBQC). Rather than relying on anything like logic gates, in MBQC, measurements do all of the work. MBQC relies on the uniquely quantum feature that measurements enable the kind of drastic changes to the state required to perform any computation, and hence this model is nothing like anything we've seen in classical computation.

Both of these models can be expressed using the ZX-calculus, which enables us to reason and prove things about them diagrammatically, as well as translate computations from one model to the other. In fact, this is a two-way street. Two of the most important theorems from Chapter 9 (universality and completeness of the ZX-calculus) come from the encoding of ZX-diagrams into the circuit model and MBQC, respectively.

12.1 The Circuit Model

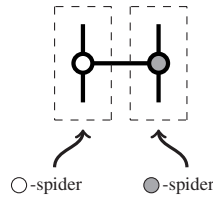
The *quantum circuit model* is a straightforward extension of computing with circuits of classical logic gates (e.g. AND, OR, NOT) to quantum processes. All computations in the circuit model are performed in three steps:

1. Prepare some qubits in a certain fixed state.
2. Perform a circuit consisting of basic quantum gates.
3. Measure (some of the) resulting qubits.

As in the case of classical circuits, we assume that the set of basic quantum gates is fixed in advance. Typical examples are phase gates:



and the CNOT gate:

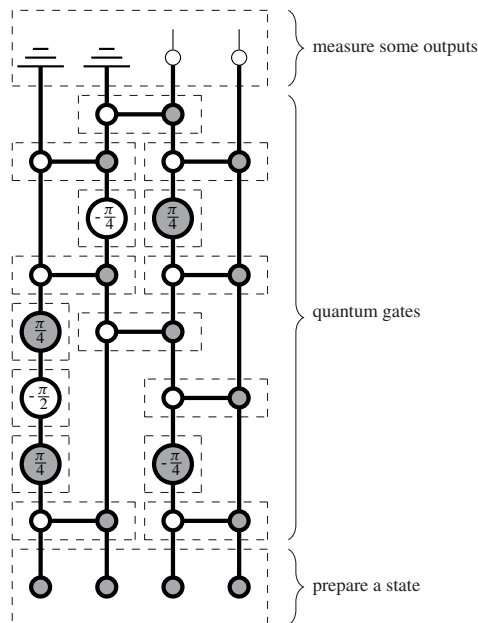


So, in general, spiders should be thought of not as gates, but rather as ‘gate pieces’. For instance, neither of the quantum spiders in the CNOT gate is a unitary on its own, but together they make up a unitary quantum gate. The ZX-calculus rules governing these spiders then yield equations between the resulting gates. In fact, one can even do the following (although we don’t really recommend it).

Exercise* 12.1 Give an equivalent presentation of the ZX-calculus using only equations between phase gates and the CNOT-gate.

12.1.1 Quantum Computing as ZX-Diagrams

The three steps making up a quantum computation in the circuit model together yield a ZX-diagram like this one:

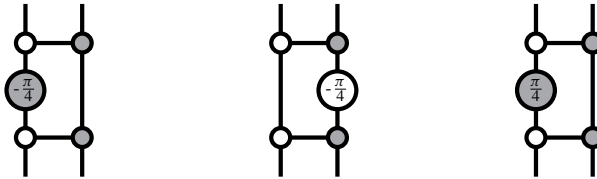


We can now use ZX-calculus for establishing equations between computations. If such a diagram only involves phases that are multiples of $\frac{\pi}{2}$, or only involves single qubit gates

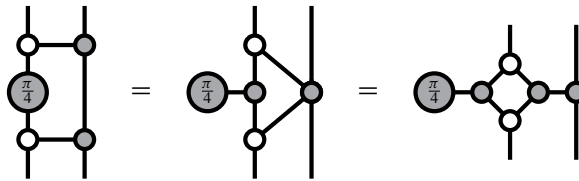
with phases that are multiples of $\frac{\pi}{4}$, then we know, by Theorem 9.129 and Theorem 9.132, respectively, that any equation that holds for circuits consisting of those gates can be derived in ZX-calculus. However, the ZX-calculus is still very useful even for more general circuits.

For example, let's see what the ZX-calculus tells us about the fairly complicated circuit depicted above. If we try to keep the quantum gates intact, we're pretty much stuck, but what if we temporarily forget that the chunk in the middle is made up of quantum gates, and just treat it as any ZX-diagram? That's when some ZX-magic happens!

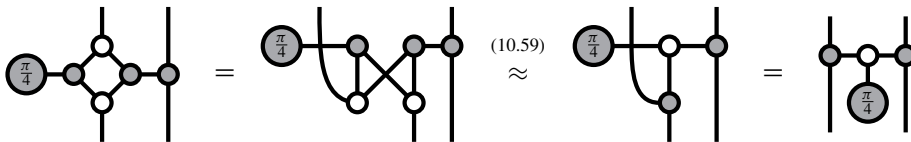
First, consider the following chunks of the diagram:



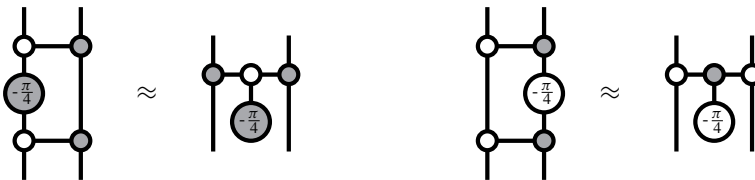
Forgetting that we are dealing with gates, we can identify a 4-cycle:



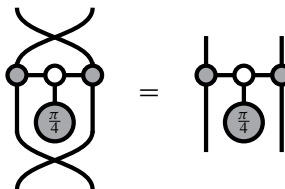
and make use of the strong complementarity rules:



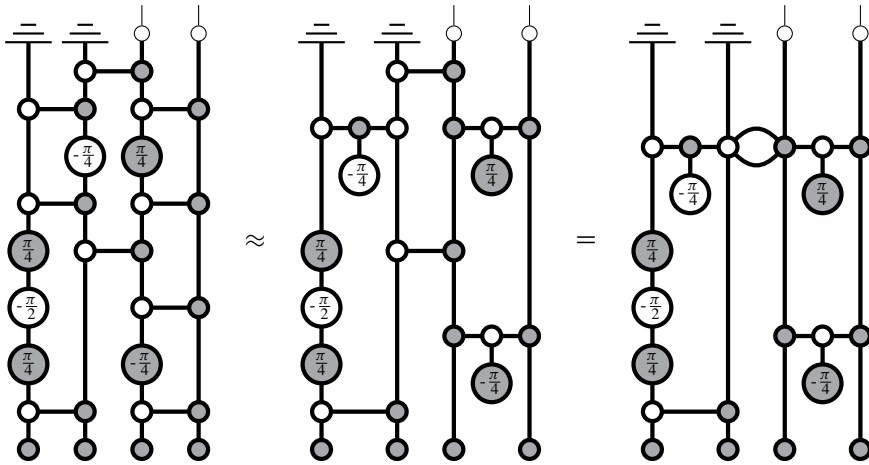
Similarly we have:



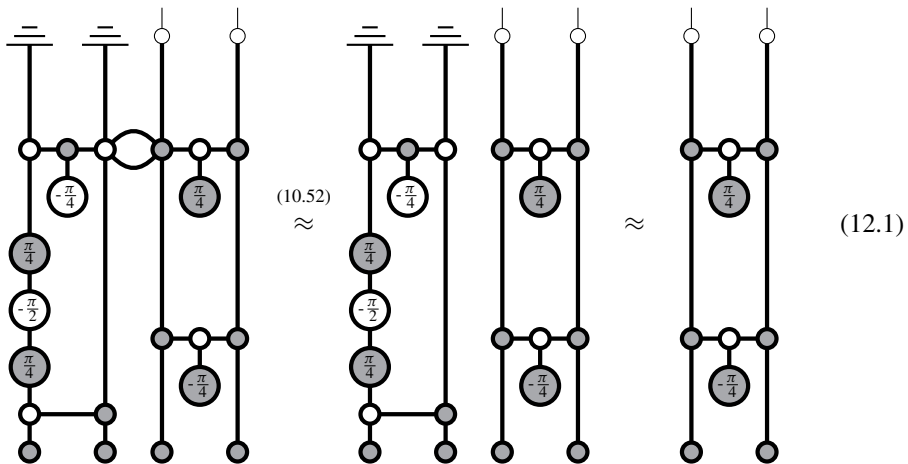
One immediate consequence, which wasn't visible before, is that these chunks are symmetric in their two inputs:



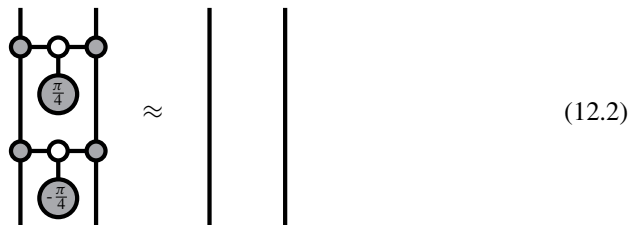
More strikingly, after substituting in the big circuit we obtain:



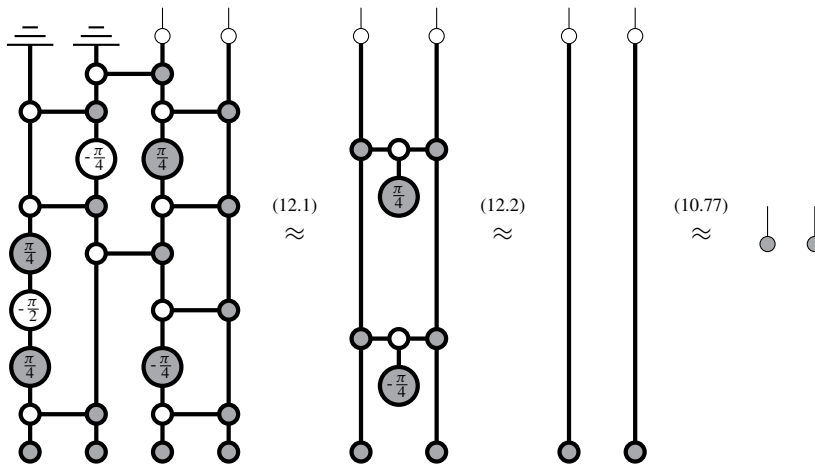
By complementarity, the left half of the circuit separates from the right half, so the (discarded) left half has no effect:



Since we moreover have:



(which is easy to show if we put these back in gate form), then our complicated circuit wasn't so complicated after all:



Exercise 12.2 Rather than reverting back to the gate form, prove equation (12.2) directly using strong complementarity.

12.1.2 Building Quantum Gates as ZX-Diagrams

Quantum gates are simple unitary quantum processes that can be used to construct more complicated ones. We have already seen some good candidates for quantum gates, such as phase gates and CNOT. We will now construct some more sophisticated ones using phase spiders as ‘gate pieces’.

Recall from Example 9.80 that we can use:

$$\begin{array}{c} \downarrow \\ \nabla \\ 0 \end{array} \stackrel{(10.65)}{\approx} \begin{array}{c} \bullet \\ 0 \end{array} \qquad \begin{array}{c} \downarrow \\ \nabla \\ 1 \end{array} \stackrel{(10.65)}{\approx} \begin{array}{c} \bullet \\ \pi \end{array}$$

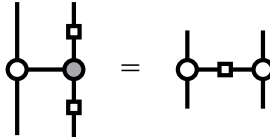
to derive the fact that the CNOT gate uses the left qubit, called the *control qubit*, to decide whether to do nothing to the right qubit or to apply a \bullet -phase of π :

$$\begin{array}{c} \begin{array}{c} \downarrow \\ \nabla \\ 0 \end{array} \begin{array}{c} \downarrow \\ \nabla \\ 1 \end{array} \end{array} \approx \begin{array}{c} \bullet \\ 0 \end{array} \quad \begin{array}{c} \downarrow \\ \nabla \\ 1 \end{array} \approx \begin{array}{c} \bullet \\ \pi \end{array} \begin{array}{c} \bullet \\ \pi \end{array}$$

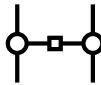
Naturally, we can ask if it is possible to construct controlled versions of other stuff such as, for example, a controlled \circ -phase of π . The equations above suggest an easy way to do this, using the colour-change rule (10.81):



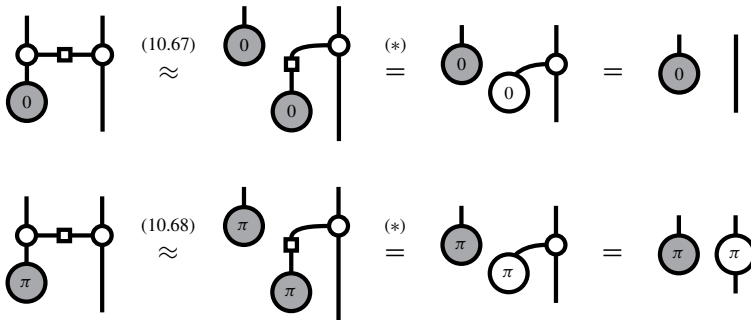
Simplifying this a bit gives:



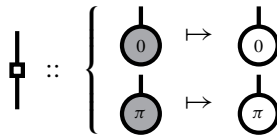
Definition 12.3 The *CZ-gate* is:



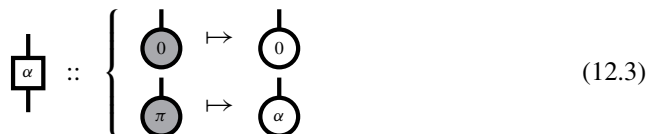
What if, instead of selectively applying a \odot -phase of π , we want to selectively apply any phase α ? How could we build such a thing? Looking more closely at how the CZ-gate works gives us a clue:



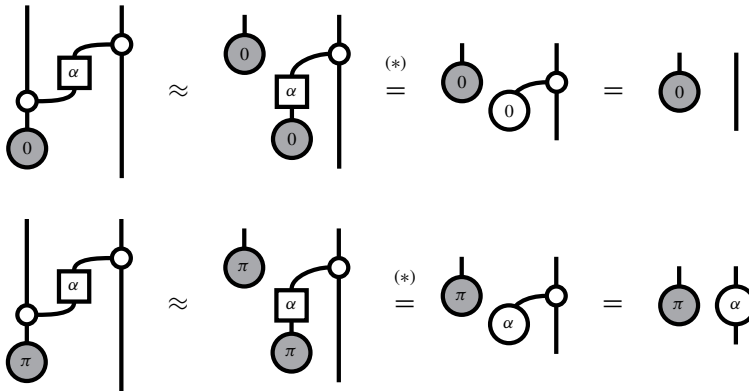
The crucial step in both derivations, marked (*), is where the *H-gate* turns a \bullet -phase state into a \odot -phase state:



If we could generalise this by replacing π with any phase α :



then we would be there:



With a bit of ZX-trickery, building (12.3) isn't too hard.

Proposition 12.4 For:

$$\boxed{\alpha} := \begin{array}{c} \frac{\alpha}{2} \\ \bullet \\ -\frac{\alpha}{2} \\ \bullet \\ \frac{\alpha}{2} \end{array} \quad (12.4)$$

we have:

$$\begin{array}{c} \boxed{\alpha} \\ \bullet \\ 0 \end{array} = \begin{array}{c} \bullet \\ 0 \end{array} \quad \begin{array}{c} \boxed{\alpha} \\ \bullet \\ \pi \end{array} = \begin{array}{c} \bullet \\ \alpha \end{array} \quad (12.5)$$

Proof On the \bullet -phase of 0 we have:

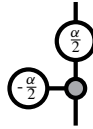
$$\begin{array}{c} \boxed{\alpha} \\ \bullet \\ 0 \end{array} = \begin{array}{c} \frac{\alpha}{2} \\ \bullet \\ -\frac{\alpha}{2} \\ \bullet \\ \frac{\alpha}{2} \end{array} \quad (10.84) = \begin{array}{c} \frac{\alpha}{2} \\ \bullet \\ -\frac{\alpha}{2} \end{array} = \begin{array}{c} \frac{\alpha}{2} \\ \bullet \\ \frac{\alpha}{2} \end{array} = \begin{array}{c} \bullet \\ 0 \end{array}$$

i.e. the phase $\frac{\alpha}{2}$ and $-\frac{\alpha}{2}$ cancel out, while on the \bullet -phase of π :

$$\begin{array}{c} \boxed{\alpha} \\ \bullet \\ \pi \end{array} = \begin{array}{c} \frac{\alpha}{2} \\ \bullet \\ -\frac{\alpha}{2} \\ \bullet \\ \frac{\alpha}{2} \\ \bullet \\ \pi \end{array} \quad (10.85) = \begin{array}{c} \frac{\alpha}{2} \\ \bullet \\ -\frac{\alpha}{2} \\ \bullet \\ \pi \end{array} = \begin{array}{c} \frac{\alpha}{2} \\ \bullet \\ \pi \\ \bullet \\ -\frac{\alpha}{2} \end{array} \quad (10.86) = \begin{array}{c} \frac{\alpha}{2} \\ \bullet \\ \frac{\alpha}{2} \end{array} = \begin{array}{c} \bullet \\ \alpha \end{array}$$

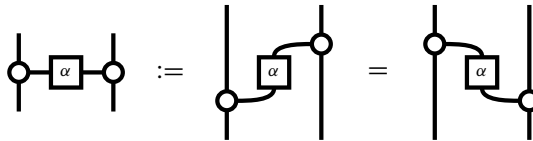
the phase $\frac{\alpha}{2}$ and $\frac{\alpha}{2}$ add up. □

Remark 12.5 When just looking at how the α -box acts on Z-basis states, the \bigcirc -phase at the bottom of the α -box doesn't play a role, i.e. this process:

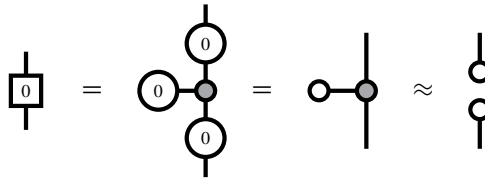


does the same job. The important feature of the α -box is that, depending on the input, the two phases will either add up or cancel out. However, including this extra phase is convenient because it makes the α -box self-transposed and makes several other things work out more nicely (cf. Proposition 12.7 below).

Like the H -gate, this α -box is self-transposed, so we can still ignore the direction of the wire without ambiguity:



However, unlike the H -gate, this α -box isn't unitary for all α . For example, it can even separate:

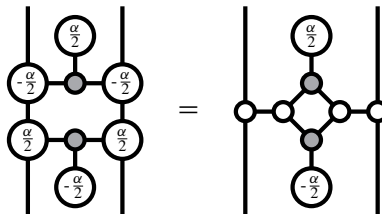


So, like the spiders in the CNOT gate, it should be treated as a 'gate piece' from which we can build interesting quantum gates, most notably the one we just saw.

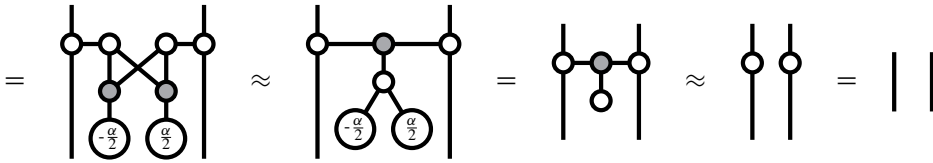
Proposition 12.6 The following $CZ(\alpha)$ -gate is unitary:

$$\begin{array}{c} \bigcirc \\ | \\ \square \alpha \\ | \\ \bullet \end{array} = \begin{array}{c} \bigcirc \frac{\alpha}{2} \\ | \\ \bullet \frac{\alpha}{2} \\ | \\ \bigcirc \frac{\alpha}{2} \end{array} \quad (12.6)$$

Proof We will use the same trick as we used in Section 12.1.1 in order to simplify the big diagram. Since we can identify a 4-cycle:



it is clear which rule we should use:



and similarly for the other composition. \square

One would expect to recover the CZ-gate in the case where $\alpha := \pi$. Indeed this does look promising, as specialising the equations from (12.5) yields:

$$\begin{array}{c} \square_{\pi} \\ \circlearrowleft 0 \end{array} = \circlearrowleft 0 \quad \begin{array}{c} \square_{\pi} \\ \circlearrowleft \pi \end{array} = \circlearrowleft \pi \quad (12.7)$$

So, this map sends states of the (doubled) Z-basis to states of the (doubled) X-basis. In other words, it seems to behave like a H -gate. However, we've known since Section 6.1.5 that a doubled ONB is not an ONB, so we still need to check that this actually is an H -gate.

Proposition 12.7 The π -box is equal to the H -gate:

$$\square_{\pi} = \square \quad (12.8)$$

Proof We have:

$$\square_{\pi} = \begin{array}{c} \circlearrowleft \frac{\pi}{2} \\ \circlearrowleft -\frac{\pi}{2} \end{array} \begin{array}{c} \square_{\pi} \\ \circlearrowleft \frac{\pi}{2} \end{array} \begin{array}{c} \circlearrowleft \frac{\pi}{2} \\ \circlearrowleft \frac{\pi}{2} \end{array} \stackrel{(10.87)}{=} \begin{array}{c} \circlearrowleft \frac{\pi}{2} \\ \circlearrowleft \frac{\pi}{2} \end{array} \begin{array}{c} \square_{\pi} \\ \circlearrowleft \frac{\pi}{2} \end{array} \begin{array}{c} \circlearrowleft \frac{\pi}{2} \\ \circlearrowleft \frac{\pi}{2} \end{array} = \begin{array}{c} \circlearrowleft \frac{\pi}{2} \\ \circlearrowleft \frac{\pi}{2} \\ \circlearrowleft \frac{\pi}{2} \end{array} = \square$$

\square

Proposition 12.6 yields a simple form for the $CZ(\alpha)$ -gate, which we built up as a ZX-diagram using 'gate pieces'. Now, is this really a new quantum gate, or can it be built from the basic gates that we already had around? Again, ZX-calculus provides the answer.

Exercise 12.8 Show using ZX-calculus that $CZ(\alpha)$ -gate can be built from CNOTs and \circlearrowleft phase gates as follows:

$$\text{Circuit with } \alpha \text{ gate} \approx \text{4-qubit circuit with } \alpha/2 \text{ and } -\alpha/2 \text{ gates} \quad (12.9)$$

(Hint: the RHS contains a 4-cycle.)

We can pass back to an X version of this gate, called the $CX(\alpha)$ -gate, by again pre- and post-composing H -gates on the second qubit:

$$\text{Circuit with } H \text{ gates} = \text{Circuit with } H \text{ gate before } \alpha = \text{Circuit with } H \text{ gate after } \alpha$$

still controlled by Z-basis states

where:

$$\boxed{\alpha} := \text{Circuit with } H \text{ gates before and after } \alpha$$

Now CNOT, (a.k.a. CX) arises as a special case where $\alpha = \pi$:

$$\text{Circuit with } H \text{ gates} = \text{Circuit with } H \text{ gate before } \pi = \text{Circuit with } H \text{ gate after } \pi = \text{Circuit with } H \text{ gate} \quad (12.8) \quad (10.82)$$

Since we can construct controlled phases of both colours, it is also possible to construct controlled unitaries, by exploiting the Euler angle decomposition from Proposition 9.100. If \hat{U} decomposes as:

$$\hat{U} = \text{Circuit with phase gates } \gamma, \beta, \alpha$$

then we can construct a controlled- \hat{U} gate as follows:

$$\text{Controlled-}\hat{U} := \text{Circuit with three qubits and unitaries } \gamma, \beta, \alpha \text{ (12.10)}$$

The unitary \hat{U} only ‘fires’ if the control qubit is ‘1’:

$$\text{Control qubit } 0 \rightarrow \text{No action} \quad \text{Control qubit } 1 \rightarrow \hat{U} \text{ (12.10)}$$

When pre- and post-composing the control qubit with NOTs, this is reversed:

$$\text{Control qubit } 0 \rightarrow \text{No action} \quad \text{Control qubit } 1 \rightarrow \text{No action (12.10)}$$

We can now combine these two to selectively perform \hat{U}_0 or \hat{U}_1 , depending on the value of the control qubit:

$$\text{Controlled-}\hat{U}_* := \text{Circuit with two qubits and unitaries } \hat{U}_1, \hat{U}_0 \text{ (12.11)}$$

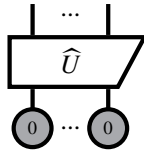
Exercise 12.9 Verify that the matrix of a *multiplexed unitary*, i.e. a gate of the form (12.11), is a block-diagonal matrix of the form:

$$\text{Controlled-}\hat{U}_* \leftrightarrow \begin{pmatrix} \mathbf{U}_0 & 0 \\ 0 & \mathbf{U}_1 \end{pmatrix} \quad (12.12)$$

where \mathbf{U}_0 and \mathbf{U}_1 are the matrices of U_0 and U_1 , respectively.

12.1.3 Circuit Universality

We now have enough tools to show that we can express any pure quantum map from qubits to qubits as a ZX-diagram. As we first noted in Section 9.4.1, it suffices to show that we can realise any unitary as a ZX-diagram. Once we have any unitary, we can obtain any state as:

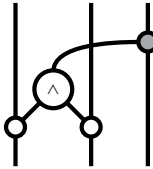


which can be transformed into an arbitrary map by process–state duality.

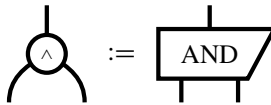
The reason for passing via unitaries is twofold. First, we will prove along the way that any unitary quantum map can be constructed using just a fixed set of unitary quantum gates, i.e. that we have a *universal set of gates* for quantum computation. Second, since interest in universal sets of gates pre-dates the birth of ZX-diagrams, most of the hard work has already been done by someone else.

We'll show that we can realise any unitary as a ZX-diagram in three steps:

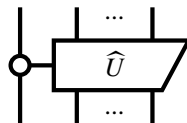
1. Construct the *Toffoli gate*:



using just the CNOT-gate and phase gates, where:



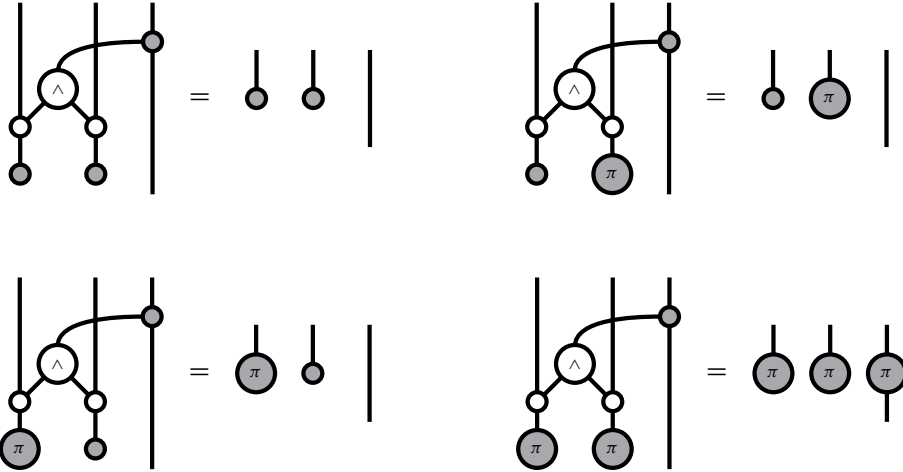
2. Use the Toffoli gate to show that, whenever we can construct an n -qubit gate \hat{U} using just the CNOT-gate and phase gates, we can also construct a controlled- \hat{U} gate:



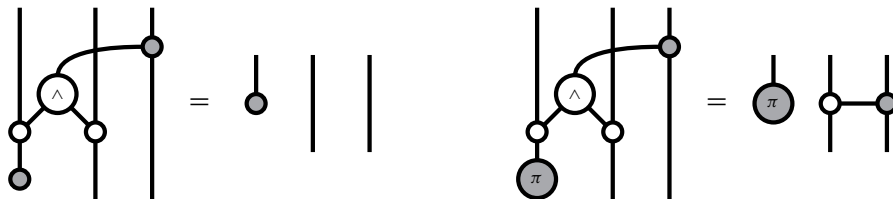
3. Use this fact to show that whenever we can build arbitrary $n - 1$ qubit unitaries, we can also build all n qubit unitaries.

12.1.3.1 Constructing Toffoli

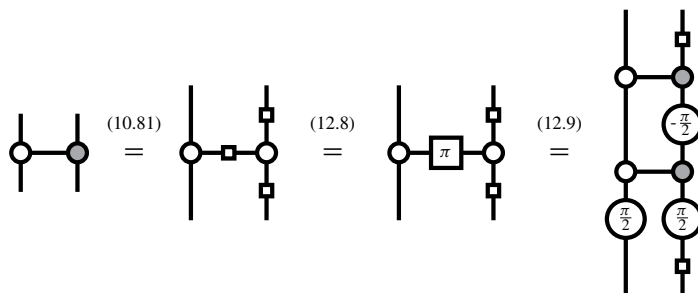
We depicted a Toffoli gate in terms of an AND gate above, to indicate that it applies a NOT gate to the third qubit precisely when both of the first two qubits are ‘1’ (i.e. both \bullet phases of π):



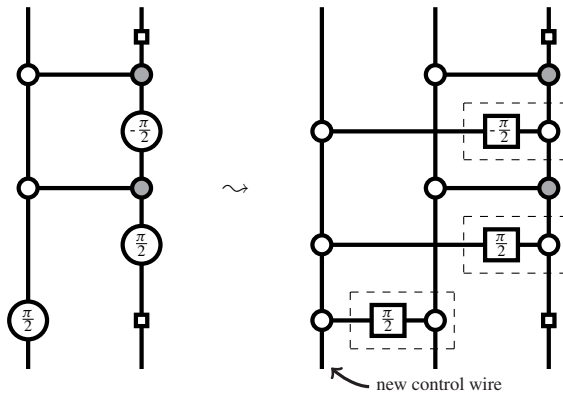
It turns out an AND gate is pretty tricky to build (cf. Exercise 12.10 below), so we’ll go straight for the Toffoli gate. First, realise that another way to think of Toffoli is as a ‘controlled-CNOT’:



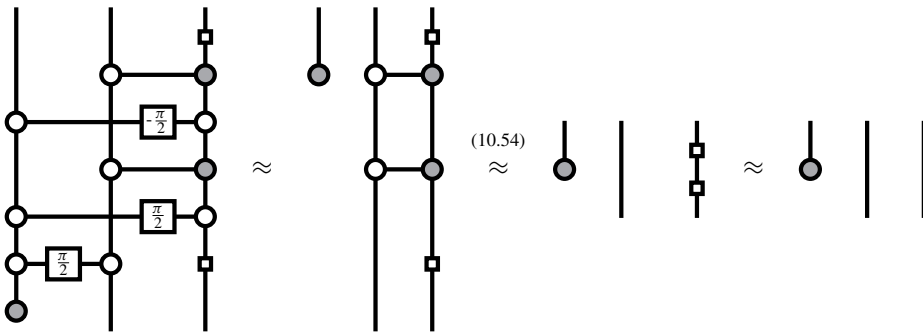
So, how can we ‘switch a CNOT off and on’? First, we’ll write CNOT in terms of the $CZ(\pi)$ -gate, which itself can be written in terms of CNOT gates and Z-phase gates:



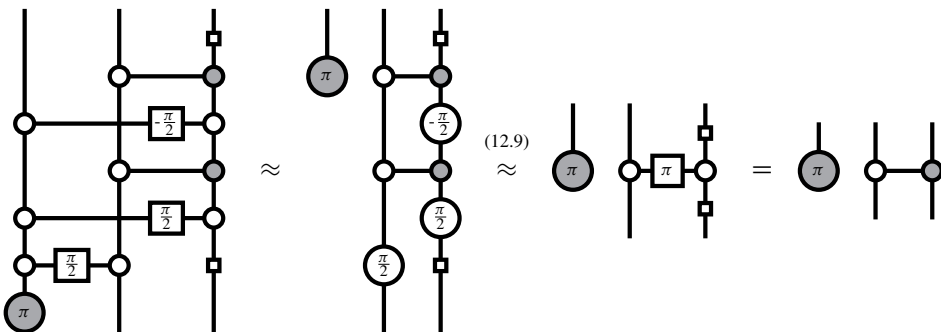
Then simply turn all of the phase gates into controlled-phase gates:



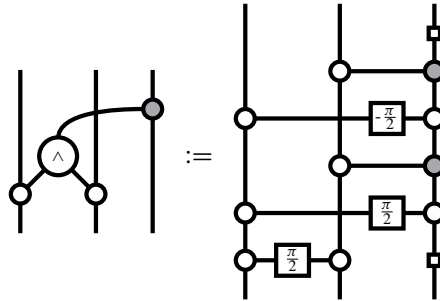
Now we have a brand new control wire. If we input '0' into the control qubit, nothing happens to the other two qubits:



whereas if we input '1', a CNOT-gate is applied:

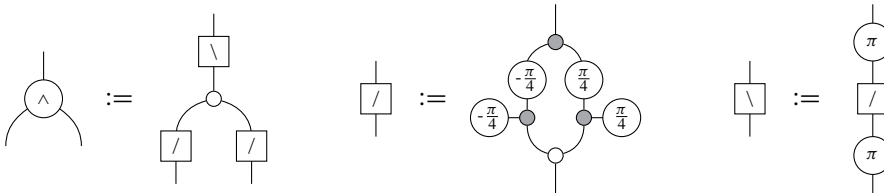


Nailed it! So let:



Since each of the $CZ(\alpha)$ -gates above can be written in terms of CNOT and phase gates, so too can the Toffoli gate.

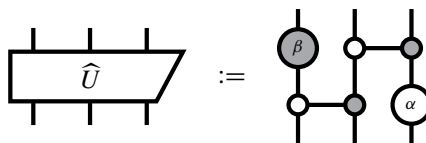
Exercise 12.10 Give an alternative construction of the Toffoli gate by first proving that the following linear map gives the AND gate:



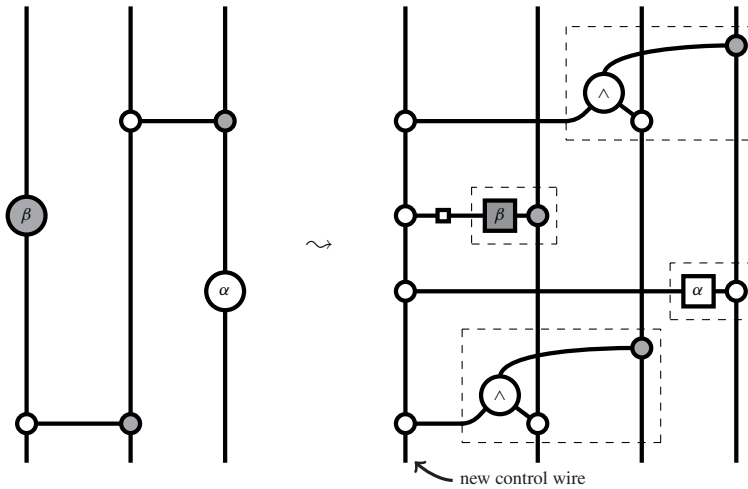
(Hint: first evaluate ‘/’ on Z-basis states, then show that ‘\’ is its inverse.)

12.1.3.2 Constructing Controlled Unitaries

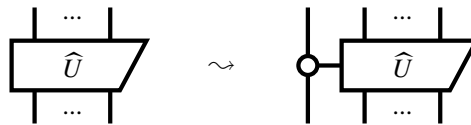
Now that step 1 is complete, we can accomplish step 2 using pretty much the same trick. Suppose \hat{U} is an n -qubit unitary built out of CNOTs and phase gates, for example:



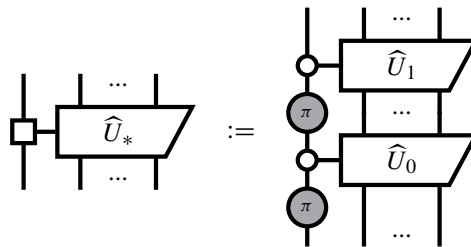
Then, we can construct controlled- \hat{U} by adding a new control line that ‘switches’ every gate in the circuit off or on. That is, every phase gate becomes a controlled-phase, and every CNOT becomes a ‘controlled-CNOT’, i.e. a Toffoli gate. Our example becomes:



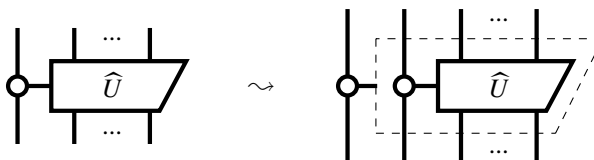
The resulting circuit will therefore, by construction, be a controlled- \hat{U} gate:



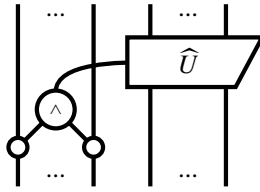
Once we have controlled operations, we can build a multiplexor just as we did in (12.11):



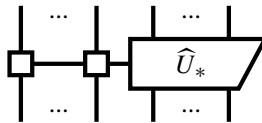
Furthermore, since the controlled- \hat{U} circuit is itself composed of CNOT and phase gates, we can repeat the whole process of adding a control line to obtain a ‘controlled-controlled- \hat{U} ’ gate:



We can also repeat this n times to obtain an n -controlled- \hat{U} gate, which we could write as:



Exercise 12.11 Following (12.11), use n -controlled unitaries to construct an n -qubit multiplexed unitary, that is, a gate:



such that for any bit string $\vec{i} := i_1 i_2 \dots i_n$, the gate applies a distinct unitary on the rightmost qubit:

$$(12.13)$$

12.1.3.3 Putting the Pieces Together

This section could also be titled ‘The Ugly Matrix Part’.

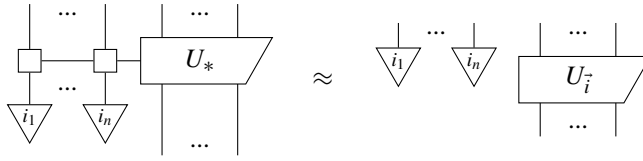
The main theorem depends on a generalisation of the Euler angle decomposition for larger unitary matrices, called the *cosine-sine decomposition*. As with the Euler decomposition, we’ll omit the proof, which is essentially just a lot of matrix manipulation.

Proposition 12.12 The matrix of any unitary can be decomposed as:

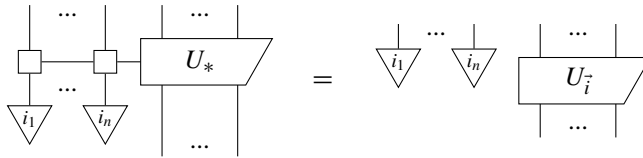
$$\begin{pmatrix} \boxed{\mathbf{U}_0} & 0 \\ 0 & \boxed{\mathbf{U}_1} \end{pmatrix} \begin{pmatrix} \boxed{\mathbf{C}} & \boxed{-\mathbf{S}} \\ \boxed{\mathbf{S}} & \boxed{\mathbf{C}} \end{pmatrix} \begin{pmatrix} \boxed{\mathbf{V}_0} & 0 \\ 0 & \boxed{\mathbf{V}_1} \end{pmatrix} \quad (12.14)$$

where \mathbf{U}_i and \mathbf{V}_i are matrices of unitary maps, and \mathbf{C} and \mathbf{S} are matrices whose (i, i) -th entries are $\cos \theta_i$ and $\sin \theta_i$, respectively, and all other entries are 0.

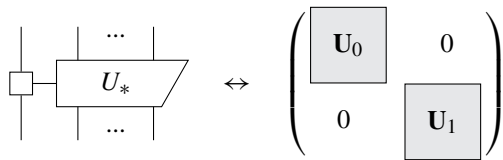
Now, we'll see how to construct these matrices from the gates we've built. Undoubtedly (12.13) gives:



Assuming the unitaries U_i are arbitrary, we can absorb the number (which is in fact just a global phase) into them, obtaining equality on the nose:



We already saw in Exercise 12.9 that the two block-diagonal matrices can be realised using multiplexors with single control wire:

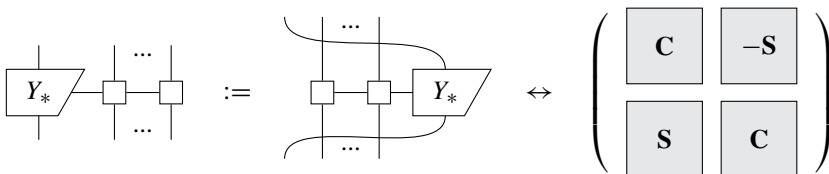


The middle matrix is a bit trickier.

Exercise* 12.13 Show that for the following unitary matrix:

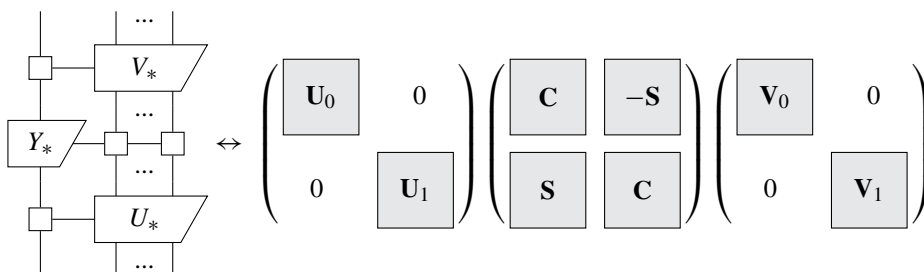
$$Y_i \leftrightarrow \begin{pmatrix} \cos \theta_i & -\sin \theta_i \\ \sin \theta_i & \cos \theta_i \end{pmatrix}$$

whose angles θ_i depend on a bit string \vec{i} , we have:



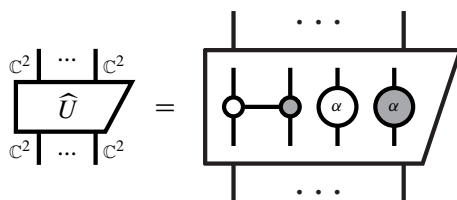
for an arbitrary matrix of sines and cosines as in Proposition 12.12.

We can therefore build arbitrary n -qubit unitaries inductively. We already know how to build arbitrary one-qubit unitaries. Assuming we can build arbitrary unitaries on $< n$ qubits, thanks to Proposition 12.12, we can construct n -qubit unitaries by means of:



This at last gives us the following theorem.

Theorem 12.14 Any n -qubit unitary can be constructed out of the CNOT-gate and phase gates:



12.2 Quantum Algorithms

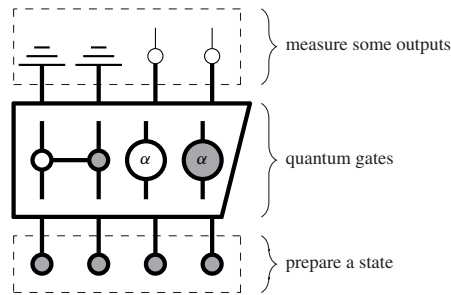
Now that we know how to build any unitary out of quantum gates, let's start to put those unitaries to work in some quantum algorithms.

Before we dig in, some disclaimers are in order. While quantum algorithms are really the 'killer app' that has generated the most excitement about new quantum features over the past two decades, all quantum algorithms that are currently known span only a fairly limited range of problems. So by no means is it the case that every problem can be solved more efficiently if we had quantum computers at hand.

Second, the analysis of quantum algorithms with diagrams is still pretty young. As you'll see, there has been some progress in this direction, but rather than a complete story, this section should be read more as a preview of (or better, an invitation to take part in!) things to come. In fact, the diagrammatic presentation of the hidden subgroup problem given in Section 12.2.4 (of which quantum factoring is an instance) and its connection to strong complementarity was only discovered in the last stages of writing this book.

12.2.1 A Quantum Oracle's (False?) Magic

We already encountered the general form that a quantum computation takes in the circuit model:

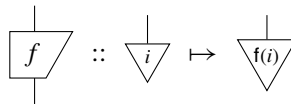


If we want to use quantum computation to beat a classical computer, we need some way or another to encode classical problems into quantum circuits. Clearly, the place to do so is the big unitary in the middle. Now, virtually any computational problem can be reduced to learning something about a function like this:

$$f : \{0, 1\}^n \rightarrow \{0, 1\}$$

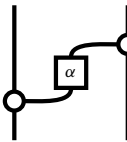
For instance, the *satisfiability* problem asks whether there exists some bit string i such that $f(i) = 1$.

So, how do we turn f into a unitary? Well, first we can encode it as a linear map, as we did for classical logic gates back in Section 5.3.4:

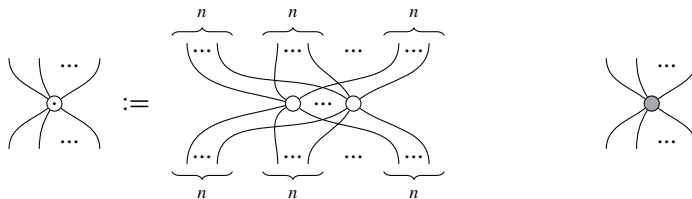


but this linear map (and hence the quantum map \hat{f}) will be unitary only if f is a bijection from its inputs to its outputs. That's no good! Especially when $N > 1$, that's just not going to happen.

But maybe not all is lost! Remember the trick we used to turn our non-unitary α -box into a two-qubit unitary? We did it like this:

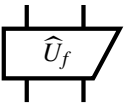


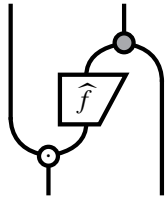
We can try a similar trick for \hat{f} . First, fix spiders for both the input and the output type of f :



Instead of choosing \bigcirc -spiders for the output system, we take a complementary spider. Why? Since this is vital for establishing unitarity.

Proposition 12.15 The quantum map:

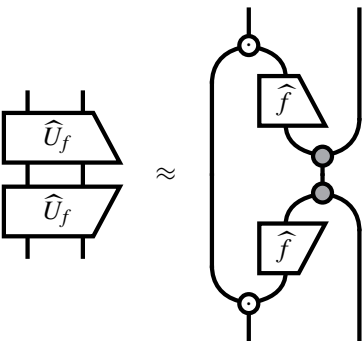
 $:=$

D 

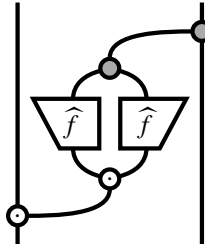
(12.15)

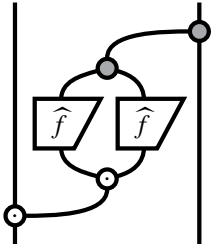
is unitary for any function f if and only if \circ and \bullet are complementary.

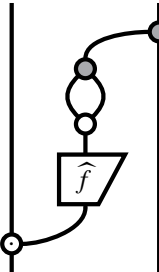
Proof First assume that \circ and \bullet are complementary. Then, also using the fact that f is a function map we have:

 \approx

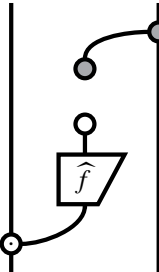
$=$

 $\stackrel{(10.40)}{=}$

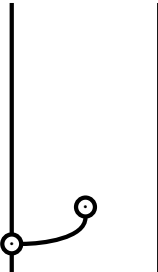


$\stackrel{(10.43)}{=}$ 


$\stackrel{(10.52)}{\approx}$

$\stackrel{(10.43)}{=}$ 

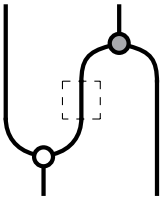
$=$




$=$



The proof for the other composition is similar. Conversely, the fact that unitarity of (12.15) implies complementarity comes from taking \hat{f} to be a plain wire. Then:

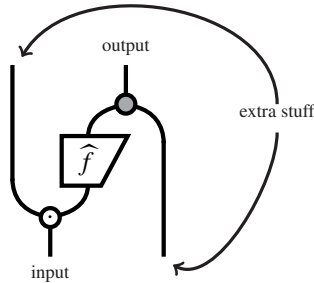
D 

 $=$

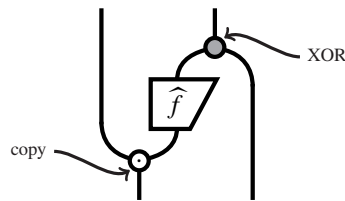
D 

is unitary, which by Proposition 9.50 implies complementarity of \circ and \bullet . □

So what does \widehat{U}_f do? It turns the quantum map \widehat{f} into a unitary by adjoining an extra input matching \widehat{f} 's output, and extra outputs matching \widehat{f} 's input:



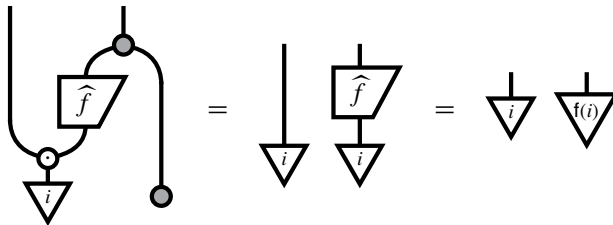
or more specifically:



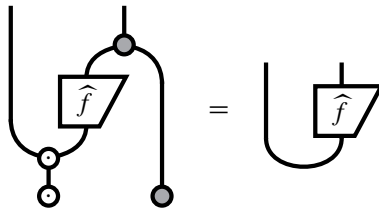
So a \bigcirc basis state at the left input of \widehat{U}_f gets copied. The first of these copies is fed to the first output of \widehat{U}_f , while the second one is fed into the input of \widehat{f} , and the corresponding output is then XOR'ed with the second input of \widehat{U}_f . If we take the second input to \widehat{U}_f to be:



then we can evaluate the function f for any input:



Pretty cool, right? Now, check out what happens when we put something that isn't a classical bit string into the first input:



Rather than getting as output the value of f at one particular input, we get the entire function f , encoded as a state. In other words, we have a superposition of the values of f at every possible input:

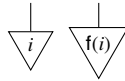
$$\text{U}_{\hat{f}} = \text{double} \left(\sum_{i \in \{0,1\}^n} \downarrow i \downarrow f(i) \right) = \text{double} \left(\sum_i \downarrow i \downarrow f(i) \right)$$

Whoa! Now you probably get what all the fuss about quantum computing is, namely:

A single quantum process can evaluate a function at all inputs simultaneously!

This single process \hat{U}_f with seemingly god-like knowledge of f is called a *quantum oracle*. But then, just how god-like is it?

Sure, we now have heaps of information about f , but how are we going to get to it? It's encoded in a quantum state, so the only thing we can do is measure it. If we're a bit thick and decide to measure the first system with respect to \odot , all that wonderful information becomes just one measurement outcome:



and we don't even get to choose i ! In other words, after spending 10 million dollars to build a fancy quantum computer, all it does is evaluate f once at some random i . What a shockingly monumental waste of money!

12.2.2 The Deutsch–Jozsa Algorithm

So is quantum computing just one big scam? Of course not! But one has to be really clever to circumvent all the harm caused by quantum measurements.

While we won't ever get access to all of those input-output pairs of f , there are lot of other, highly non-trivial things we can ask about f that still have single answers. With a bit of cleverness, we could hope to extract with a single measurement the answer to a question that classically requires knowing many (if not all) input-output pairs of f . This would genuinely allow us to drastically out-perform a classical computer for certain tasks.

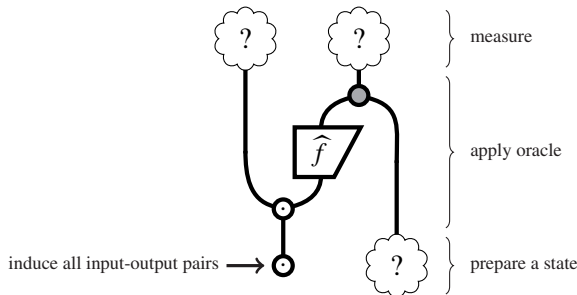
The new bad news (yes, there is more of it) is that the type of questions with single answers that can be obtained from a single measurement is very limited. For example, the 'satisfiability' problem we mentioned in the last section is provably not a question that can be answered by a single measurement.

Another question we could ask is whether a function f always returns the same answer for any input, i.e. whether a function is *constant*. Of course, if we knew the answer to this question, the 'satisfiability' problem becomes trivial (why?). Hence, we won't be able

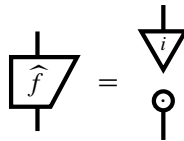
to find a solution to this problem either. However, maybe we can get some traction by assuming something about f .

The types of questions that lend themselves well to solutions by quantum algorithms are certain *promise problems*, i.e. problems where we know some piece of information about a function in advance (a ‘promise’), and we are trying to learn something additional. In the case of the Deutsch–Jozsa algorithm, the question is whether a function either is constant or satisfies some other property ‘X’. What makes it a promise problem is that ‘X’ is more specific than just ‘not constant’. To figure out what ‘X’ should be, let’s go ahead and start to build the algorithm and see what we need.

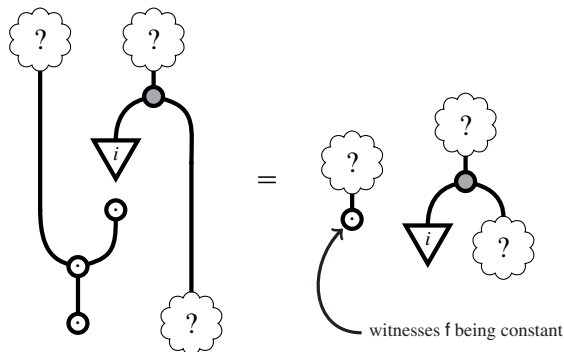
We’ll start very generically by assuming that we apply the quantum oracle for a given function f on a superposition of all inputs:



If f is constant, or equivalently, if \hat{f} is constant, then it has to be of the following form:



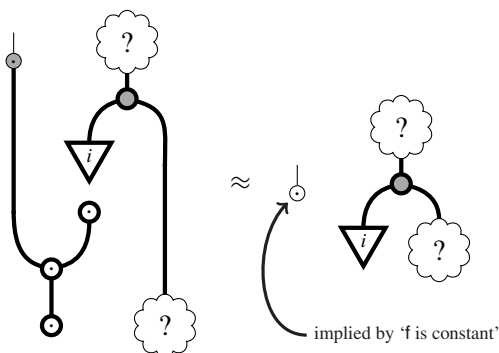
That is, it simply deletes its input and provides the constant value i as output. Substituting this in our algorithm (to be) gives:



So the diagram disconnects and the chunk on the left no longer has any dependency on the values of f . Instead, it contains a state that precisely witnesses the fact that f is constant. This state moreover happens to be an eigenstate of the \bullet -measurement:

$$\begin{array}{c} \bullet \\ | \\ \bigcirc \end{array} \stackrel{(10.77)}{\approx} \begin{array}{c} | \\ \bigcirc \end{array}$$

Hence:



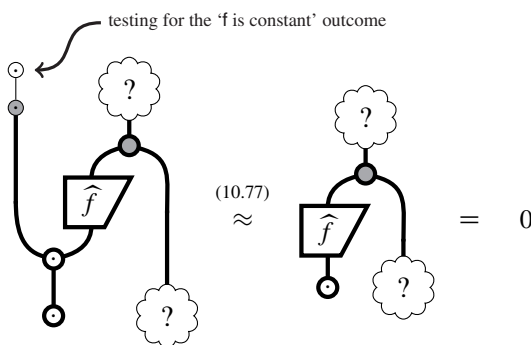
Great! So we now know that a constant f will guarantee that:

$$\begin{array}{c} | \\ \bigcirc \end{array} = \begin{array}{c} | \\ \nabla \\ 0 \end{array} \dots \begin{array}{c} | \\ \nabla \\ 0 \end{array} \quad (12.16)$$

will be the outcome for a \bullet -measurement at the oracle's first output.

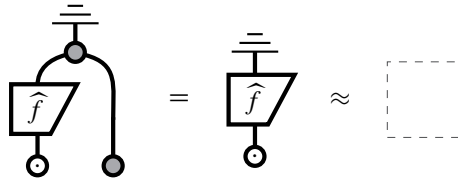
Remark 12.16 Note that here, we are not measuring the output of the function, but rather the 'extra stuff' that we added to maintain unitarity of the quantum oracle. So this 'extra stuff' has become very important in its own right!

So we now know that if f is constant, we always get that outcome. However, for a generic function f , we might also get that outcome if f is not constant. This is where our 'X' comes in. The property 'X' of a function should be chosen such that outcome (12.16) can never occur, which will guarantee that, whenever we do see this outcome, f must be constant. 'Never occurring' means probability zero, so f should satisfy 'X' if and only if:

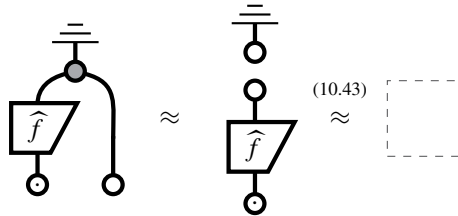


We still have some freedom in choosing ? appropriately. Thanks to causality, the cloud on top isn't of any use, so we can treat it as discarding. This just leaves the state that

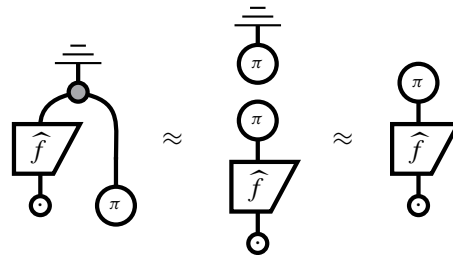
we feed into the second input of the oracle to play with. First, we try the \bullet -state we used in the previous section. But by causality we have:



So that doesn't give us anything. Next, let's try the other colour. Unfortunately, since \hat{f} comes from a function on the \circ -basis, this goes just as poorly:



We're running out of options here! We give it one more go:



Aha! This seems like it at least might go to zero for some functions f . So what does this:

$$\begin{array}{c} \pi \\ \circ \\ \hline f \\ \circ \end{array} = 0 \quad (12.17)$$

mean for the function f ? The \circ -effect with a π phase is almost like deleting, except we pick up a -1 phase when we 'delete' the second basis state:

$$\begin{array}{c} \pi \\ \circ \\ \hline 0 \end{array} = \text{dashed box} \quad \begin{array}{c} \pi \\ \circ \\ \hline 1 \end{array} = -1 \text{ dashed box}$$

So, if:

$$\begin{array}{c} \pi \\ \downarrow \\ \boxed{f} \\ \downarrow \\ \odot \end{array} = \begin{array}{c} \pi \\ \downarrow \\ \boxed{f} \\ \downarrow \\ \sum_i \downarrow i \end{array} = \sum_i \begin{array}{c} \pi \\ \downarrow \\ \nabla f(i) \end{array} = \sum_{f(i)=0} 1 + \sum_{f(i)=1} (-1) = 0$$

then the number of values i such that $f(i) = 0$ must be the same as the number where $f(i) = 1$. This means the function f is *balanced*.

So we found ourselves a genuine quantum algorithm! Namely, the following one.

Given. A function:

$$f : \{0, 1\}^n \rightarrow \{0, 1\}$$

with the promise that it is either *constant* :=

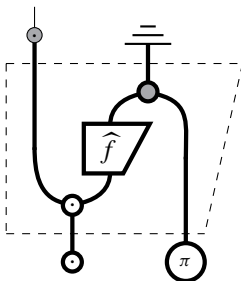
- f either always returns 0 or always returns 1,

or *balanced* :=

- f returns 0 for the same number of inputs as it returns 1.

Problem. Is f constant or balanced?

Quantum algorithm. Perform:



and if the outcome is:

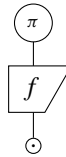


then f is constant. Otherwise it is balanced.

Classically, to be totally sure, we will need to examine at least half of the outputs, plus 1, to determine if f is constant or balanced. In other words, we will need to *query* f at least $2^{n-1} + 1$ times. Shockingly, the quantum version requires only one query!

Remark 12.17 Note that, for this to be efficient, we must also assume that the unitary oracle can be implemented efficiently, e.g. by using quantum gates. This of course depends on the function f . However, if there exists an efficient way to implement f on a classical computer, it can be efficiently implemented using quantum gates. We refer the interested reader to further reading in the historical notes and references at the end of this chapter.

Generalising (12.17), one could say that the number:



can be used to gauge ‘how balanced’ f is. That is, it becomes positive if f produces more 0s than 1s, and negative if there are fewer 0s than 1, and the further the number is from 0, the greater the difference. We will make use of this fact in the next algorithm.

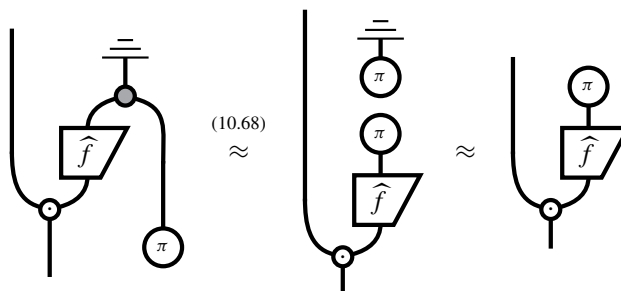
12.2.3 Quantum Search

We now know that quantum processes allow us to speed up at least one task. Unfortunately, there just aren’t that many times when knowing if a function is constant or balanced is a matter of life and death. But maybe we can learn from the tricks we used and exploit these to do something useful.

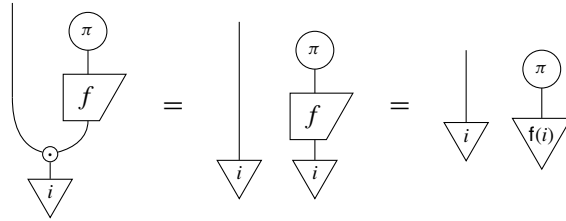
The crux of the Deutsch–Jozsa algorithm is the fact that a \odot -effect with a π phase deletes the result of f if it is 0, and introduces a -1 phase when it is 1:

$$\begin{array}{c} \pi \\ \circlearrowleft \\ \nabla \\ 0 \end{array} = \boxed{} \qquad \begin{array}{c} \pi \\ \circlearrowleft \\ \nabla \\ 1 \end{array} = -1 \boxed{} \qquad (12.18)$$

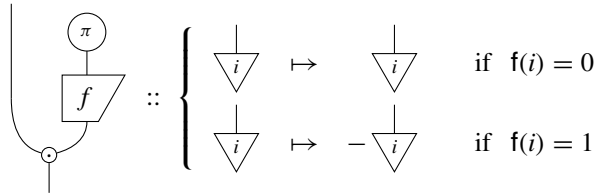
Plugging a \odot -state with a π into the second input of our oracle gives:



The (undoubled version of) this process acts on classical inputs as follows:



Hence by (12.18) we have:



So any i where $f(i) = 1$ gets ‘marked’ by flipping its sign. Hence, by applying this process to ‘all inputs together’ we obtain:

$$\begin{aligned} & \text{Circuit with } f \text{ box, } \pi \text{ circle, and } i \text{ triangle} = \text{Circuit with } f \text{ box, } \pi \text{ circle, and } \sum_i i \text{ triangle} \\ & = \sum_{f(i)=0} i \text{ triangle} - \sum_{f(i)=1} i \text{ triangle} \end{aligned} \quad (12.19)$$

In the Deutsch–Jozsa algorithm, we exploited this state to detect constant versus balanced by means of a single \bullet -measurement. But what if, on the other hand, we could devise a measurement that gives us just one of the ‘marked’ bit strings i as an outcome? Well, this solves quite a useful problem indeed: the *search problem*. Many difficult computational problems boil down to search.

Suppose we have a set of things (e.g. apples), some of which are good (e.g. fresh) and some are bad (e.g. rotten). We want a fresh apple. This is easy if there are many apples and only one is rotten, but if it’s the other way around, this is very hard, since we have to check the apples one by one. Putting this in terms of a function, suppose we have:

$$f : \{0, 1\}^n \rightarrow \{0, 1\}$$

where 0 stands for rotten and 1 stands for fresh. Can we find a fresh apple, i.e. some i such that $f(i) = 1$?

Simplifying the LHS of (12.19) gives:

$$\begin{aligned} & \text{Circuit with } f \text{ box, } \pi \text{ circle, and } i \text{ triangle} = \sum_{f(i)=0} i \text{ triangle} - \sum_{f(i)=1} i \text{ triangle} \end{aligned}$$

All of the good stuff now carries a minus sign. Comparing this with the following state:

$$\bigcirc = \sum_i \nabla_i \quad (12.20)$$

we see that subtracting these two states eliminates the rotten stuff:

$$\bigcirc - \begin{array}{c} \text{---} \\ \diagup \text{ } f \text{ } \diagdown \\ \text{---} \\ \bigcirc \pi \end{array} = 2 \sum_{f(i)=1} \nabla_i$$

Great! So we just measure the result and we'll get a fresh apple. So the search problem is now reduced to finding a unitary that does this:

$$\begin{array}{c} \text{---} \\ \diagup \text{ } f \text{ } \diagdown \\ \text{---} \\ \bigcirc \pi \end{array} \mapsto \bigcirc - \begin{array}{c} \text{---} \\ \diagup \text{ } f \text{ } \diagdown \\ \text{---} \\ \bigcirc \pi \end{array}$$

Clearly it should consist of two terms, one that produces (12.20) as a constant, erasing the input, and one that does nothing except introducing a -1 phase. So let:

$$\begin{array}{c} \text{---} \\ \diagup \text{ } d \text{ } \diagdown \\ \text{---} \end{array} := \lambda \begin{array}{c} \bigcirc \\ \bigcirc \end{array} - \begin{array}{c} \text{---} \\ \text{---} \end{array}$$

Then:

$$\begin{array}{c} \text{---} \\ \diagup \text{ } d \text{ } \diagdown \\ \text{---} \end{array} :: \begin{array}{c} \text{---} \\ \diagup \text{ } f \text{ } \diagdown \\ \text{---} \\ \bigcirc \pi \end{array} \mapsto \left(\lambda \begin{array}{c} \bigcirc \\ \diagup \text{ } f \text{ } \diagdown \\ \text{---} \\ \bigcirc \pi \end{array} \right) - \begin{array}{c} \text{---} \\ \diagup \text{ } f \text{ } \diagdown \\ \text{---} \\ \bigcirc \pi \end{array}$$

Now we can choose λ to cancel out the extra number:

$$\begin{array}{c} \bigcirc \\ \diagup \text{ } f \text{ } \diagdown \\ \text{---} \\ \bigcirc \pi \end{array}$$

in the first term. In the previous section, we saw that this extra number measures the 'balance' of f :

$$\begin{array}{c} \bigcirc \\ \diagup \text{ } f \text{ } \diagdown \\ \text{---} \\ \bigcirc \pi \end{array} = \begin{array}{c} \bigcirc \pi \\ \diagup \text{ } f \text{ } \diagdown \\ \text{---} \\ \bigcirc \end{array} = \sum_{f(i)=0} 1 + \sum_{f(i)=1} (-1) = N_0 - N_1$$

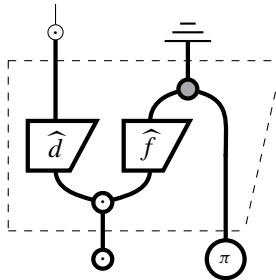
where N_0 is the number of i s where $f(i) = 0$, and N_1 is the number of i s where $f(i) = 1$. So if we take λ to be $\frac{1}{N_0 - N_1}$ then:

$$\lambda \begin{array}{c} \textcircled{\pi} \\ | \\ \text{---} \text{ / } f \text{ \textbackslash } \text{---} \\ | \\ \textcircled{} \end{array} = \text{---} \text{---} \text{---} \quad (12.21)$$

and hence:

$$\begin{array}{c} \text{---} \\ | \\ \text{---} \text{ / } d \text{ \textbackslash } \text{---} \\ | \\ \text{---} \text{ / } f \text{ \textbackslash } \text{---} \\ | \\ \textcircled{\pi} \end{array} = 2 \sum_{f(i)=1} \begin{array}{c} \text{---} \\ | \\ \text{---} \text{ / } i \text{ \textbackslash } \text{---} \\ | \\ \text{---} \end{array}$$

only contains the good stuff. So:



will always give us a fresh apple!

But, as you are probably getting used to by now, there's a catch.

Exercise 12.18 Show that, for d to be unitary, it must be the case that:

$$\lambda = \frac{2}{N}$$

where λ is assumed to be a real number and $N := N_0 + N_1 = 2^n$.

Putting the two equations for λ together, we have:

$$\frac{1}{N_0 - N_1} = \frac{2}{N_0 + N_1}$$

which, after a bit of simple algebra, gives:

$$\frac{N_1}{N_0 + N_1} = \frac{1}{4}$$

Hence exactly one in four apples must be fresh. So we (re-)discovered the following quantum search algorithm.

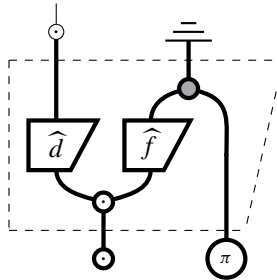
Given. A function:

$$f : \{0, 1\}^n \rightarrow \{0, 1\}$$

with the promise that exactly one out of four bit strings are mapped to 1.

Problem. Find an bit string that is mapped to 1.

Quantum algorithm. Perform:



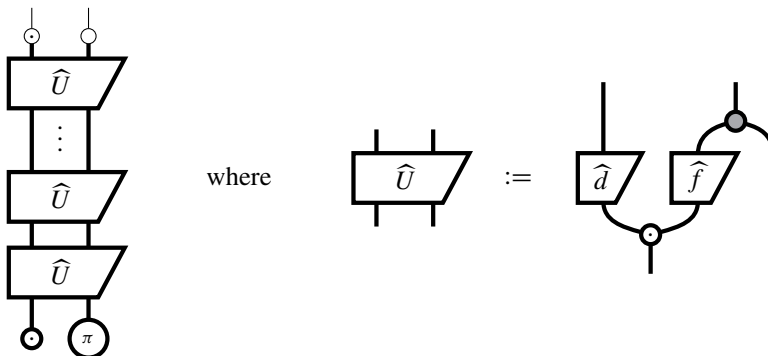
where:

$$d := \frac{2}{N} \left(\text{control point} - \text{target point} \right)$$

Then we always find such a bit string.

Classically, one has a probability $\frac{1}{4}$ to find a good bitstring each time one tries, and if one is really unlucky, one has to try $\frac{3N}{4} + 1$ times to find one. Again, in the quantum version it's bingo after the first try!

Of course, if fewer (or more!) than $D/4$ elements are marked, the probability of getting an unmarked element is non-zero. However, we can improve our chances if we simply iterate the unitary part of the protocol:



With each iteration, the probability of getting a marked element goes up, and one can show that for a single marked element, the probability of getting an unmarked outcome goes to zero after about \sqrt{D} iterations. This multistep version is called *Grover's algorithm*.

12.2.4 The Hidden Subgroup Problem

To cap off our discussion of quantum algorithms, we'll now look at the *hidden subgroup problem*, whose solution is probably the most important quantum algorithm to date. Why is that? Since it's not clear why one should care about 'subgroups' or how they can 'hidden', it is of course not clear why one should care about the hidden subgroup problem. Well, at least one really good reason is that if we can efficiently solve the hidden subgroup problem, we can break lots of cryptography! That is, the factoring algorithm and discrete logarithm algorithms, which as we mentioned in the introduction to this chapter can be used to break many cryptographic systems, occur as special cases of the solution to the hidden subgroup problem. Who knew obscure problems about groups had anything to do with cryptography? (Answer: any cryptographer.)

We'll focus on commutative groups and make use of the classification of strongly complementary spiders from Section 9.3.6. For any commutative group G , there exists a system (which we'll also call G) and a pair of strongly complementary spiders \bigcirc / \bullet that encode G . That is, for:

$$\left\{ \begin{array}{c} | \\ \bigcirc \\ \text{g} \end{array} \right\}_{g \in G} \cong \left\{ \begin{array}{c} | \\ \bullet \\ \kappa_g \end{array} \right\}_{g \in G}$$

we have:

$$\begin{array}{c} \begin{array}{c} | \\ \bigcirc \\ \kappa_g \end{array} \approx \begin{array}{c} | \\ \bullet \\ \kappa_g \end{array} \quad \begin{array}{c} | \\ \bullet \\ \kappa_{g'} \end{array} \end{array} \quad \begin{array}{c} | \\ \bullet \\ \kappa_g \end{array} \quad \begin{array}{c} | \\ \bullet \\ \kappa_{g'} \end{array} = \begin{array}{c} | \\ \bullet \\ \kappa_{g+g'} \end{array} \quad (12.22)$$

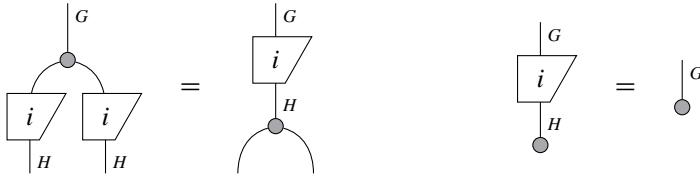
Note that we carefully label wires above. This is because we now use a different pair of strongly complementary spiders to encode a subgroup $H \subseteq G$:

$$\begin{array}{c} \begin{array}{c} | \\ \bigcirc \\ \kappa_h \end{array} \approx \begin{array}{c} | \\ \bullet \\ \kappa_h \end{array} \quad \begin{array}{c} | \\ \bullet \\ \kappa_{h'} \end{array} \end{array} \quad \begin{array}{c} | \\ \bullet \\ \kappa_h \end{array} \quad \begin{array}{c} | \\ \bullet \\ \kappa_{h'} \end{array} = \begin{array}{c} | \\ \bullet \\ \kappa_{h+h'} \end{array} \quad (12.23)$$

To witness the fact that this is a subgroup of G , we give an *inclusion map*, which embeds the group elements of H into G :

$$\begin{array}{c} | \\ \text{ } \\ \text{ } \\ | \\ H \end{array} \quad \begin{array}{c} | \\ H \\ \kappa_h \end{array} \mapsto \begin{array}{c} | \\ G \\ \kappa_h \end{array}$$

The unit and group-sum from H are preserved by inclusion, so i is a *group homomorphism*:



Note we again use wire labels to distinguish the spiders for G in (12.22) from the spiders for H in (12.23).

The way H is ‘hidden’ is via the *quotient group* G/H . This is a new group whose elements are sets of G elements, called *equivalence classes*. If we let:

$$[g] := \{ g' \in G \mid \exists h \in H . g' = g + h \}$$

then the set:

$$G/H := \{ [g] \mid g \in G \}$$

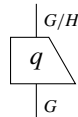
becomes a group, with unit $[0]$ and group-sum:

$$[g] + [g'] := [g + g']$$

As with the other two groups, we fix a strongly complementary pair of spiders to encode G/H :



This time we get a map coming out of G , called the *quotient map*:



which sends every $g \in G$ to $[g] \in G/H$. Just like with i , q is a group homomorphism:



Notably, if $h \in H$, the quotient map sends h to $[h] \in G/H$. But then, $h = 0 + h$, so $[h] = [0]$. Hence, every element in h gets sent to the unit. Diagrammatically, this means that if we compose i and q , this results in deleting h and sending out the unit:

$$\begin{array}{c} G/H \\ | \\ \text{q} \\ | \\ G \\ | \\ \text{i} \\ | \\ H \end{array} \approx \begin{array}{c} G/H \\ \bullet \\ | \\ \circ \\ | \\ H \end{array} \quad (12.25)$$

Okay, we are ready to ‘hide’ H . Suppose we are given a function:

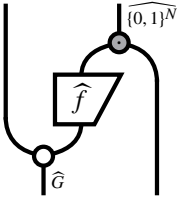
$$f : G \rightarrow \{0, 1\}^N$$

with the promise that it decomposes as follows:

$$\begin{array}{c} | \\ \text{f} \\ | \end{array} = \begin{array}{c} \{0, 1\}^N \\ | \\ \text{f}' \\ | \\ G/H \\ | \\ \text{q} \\ | \\ G \end{array} \quad \begin{array}{l} \leftarrow \text{injective function} \\ \leftarrow H \text{ 'hidden inside' } f \\ \leftarrow \text{quotient map} \end{array} \quad (12.26)$$

where ‘injective function’ just means f' is a function map and an isometry. Can we figure out what H is?

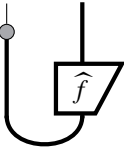
As usual, we use spiders to build the oracle for f :



where \odot/\ominus is any complementary pair of spiders for the bit-string system (last one, we promise!). Now, the first thing we thought was really cool about quantum oracles was that they let us prepare states like this:

$$\begin{array}{c} | \\ \text{f-hat} \\ | \end{array} \odot \begin{array}{c} \{0, 1\}^N \\ | \\ \text{f-hat} \\ | \\ G \end{array} = \begin{array}{c} | \\ \text{f-hat} \\ | \end{array} \quad (12.27)$$

Let’s see what happens when we measure the left system using \bullet :



Individual measurement outcomes correspond to \bullet -ONB effects, which are the same as group elements, encoded as \circ -phases:

$$\begin{array}{c} \text{g} \\ \uparrow \\ \bullet \\ \uparrow \\ \widehat{f} \end{array} \approx \begin{array}{c} \text{kg} \\ \circ \\ \uparrow \\ \bullet \\ \uparrow \\ \widehat{f} \end{array} \stackrel{(10.56)}{\approx} \begin{array}{c} \text{kg} \\ \circ \\ \uparrow \\ \bullet \\ \uparrow \\ \widehat{f} \end{array} \quad (12.28)$$

We can figure out which group elements we obtain this way with the help of a lemma about the (adjoint of the) quotient map.

Lemma 12.19 For i the subgroup map and q the quotient map we have:

$$\begin{array}{c} G \\ | \\ \bullet \\ / \quad \backslash \\ \boxed{i} \quad \boxed{q} \\ | \quad | \\ H \quad G/H \end{array} \approx \begin{array}{c} G \\ | \\ \circ \\ | \\ H \end{array} \quad \begin{array}{c} G \\ | \\ \boxed{q} \\ | \\ G/H \end{array} \quad (12.29)$$

Proof We have:

The diagram illustrates the derivation of the Frobenius property for the comultiplication q . It consists of several steps:

- Top Row:** A sequence of four diagrams connected by equals signs. The first diagram shows a multiplication i and a comultiplication q on a single wire. The second diagram shows a wire with a dot (multiplication) and a comultiplication q . The third diagram shows a wire with a comultiplication q and a dot (multiplication). The fourth diagram shows a wire with a comultiplication q and a dot (multiplication). The label (10.40) is placed between the third and fourth diagrams.
- Middle Row:** A sequence of three diagrams connected by equals signs. The first diagram shows a wire with a comultiplication q and a dot (multiplication). The second diagram shows a wire with a comultiplication q and a dot (multiplication). The third diagram shows a wire with a comultiplication q and a dot (multiplication). The label (12.24) is placed between the first and second diagrams, and (12.25) is placed between the second and third diagrams.
- Bottom Row:** A sequence of three diagrams connected by equals signs. The first diagram shows a wire with a comultiplication q and a dot (multiplication). The second diagram shows a wire with a comultiplication q and a dot (multiplication). The third diagram shows a wire with a comultiplication q and a dot (multiplication). The label (10.40) is placed between the first and second diagrams.

Looking at our promise for \mathbf{f} in the context of (12.27), we have:

$$\begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ f \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \stackrel{(12.26)}{=} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ f' \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} = \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ q \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ f' \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array}$$

If we \otimes -compose with a \circ -effect on the left, we obtain:

$$\begin{array}{c} \circ \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ q \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ f' \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \stackrel{(12.29)}{=} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \bullet \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ i \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ q \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ f' \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array}$$

Now, plugging in a \circ -phase of κ_g gives:

$$\begin{array}{c} \kappa_g \\ \circ \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ q \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ f' \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \approx \begin{array}{c} \kappa_g \\ \circ \\ | \\ \text{---} \end{array} \begin{array}{c} \bullet \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ i \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ q \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ f' \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \approx \begin{array}{c} \kappa_g \\ \circ \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ i \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \kappa_g \\ \circ \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ q \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ f' \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \quad (12.30)$$

So, it is either the case that:

$$\begin{array}{c} \kappa_g \\ \circ \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ q \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ f' \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} = 0$$

in which case the probability of getting the outcome corresponding to κ_g is:

$$\begin{array}{c} \kappa_g \\ \circ \\ | \\ \text{---} \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ \hat{f} \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} = \begin{array}{c} \kappa_g \\ \circ \\ | \\ \text{---} \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ \hat{q} \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ \hat{f}' \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} = 0$$

or we can cancel out this state from both sides of (12.30), giving:

$$\begin{array}{c} \kappa_g \\ \circ \\ | \\ \text{---} \end{array} \begin{array}{c} G \\ | \\ \text{---} \end{array} \begin{array}{c} \diagup \\ i \\ \diagdown \end{array} \begin{array}{c} \text{---} \\ | \\ \text{---} \end{array} \begin{array}{c} H \\ | \\ \text{---} \end{array} \approx \begin{array}{c} \circ \\ | \\ \text{---} \end{array} \begin{array}{c} H \\ | \\ \text{---} \end{array}$$

In fact, it is not hard from here to show that the \approx -equation above actually holds on the nose. So, we can conclude that, by measuring the left system of the oracle as in (12.28), we will always obtain outcomes corresponding to group elements $g \in G$ where:

$$\begin{array}{c} \textcircled{\kappa_g} \\ | \\ \text{---} G \text{---} \\ \text{---} i \text{---} \\ | \\ \text{---} H \text{---} \end{array} = \begin{array}{c} \textcircled{} \\ | \\ \text{---} H \text{---} \end{array} \quad (12.31)$$

This is all we need to solve the hidden subgroup problem. But to see this, we need to have a closer look at what the equation above means. Even though κ_g is not deleting:

$$\begin{array}{c} \textcircled{\kappa_g} \\ | \\ \text{---} \end{array} \neq \begin{array}{c} \textcircled{} \\ | \\ \text{---} \end{array}$$

because of (12.31), it acts just like deleting when restricted to the subgroup H ; that is, if $h \in H$ then:

$$\begin{array}{c} \textcircled{\kappa_g} \\ | \\ \frac{1}{\sqrt{D}} \textcircled{\kappa_h} \end{array} = \begin{array}{c} \text{---} \end{array}$$

In group theory, the set of phases that ‘locally’ delete a subgroup H is called the *annihilator* of H :

$$H' := \left\{ g \in G \mid \forall h \in H : \begin{array}{c} \textcircled{\kappa_g} \\ | \\ \frac{1}{\sqrt{D}} \textcircled{\kappa_h} \end{array} = \begin{array}{c} \text{---} \end{array} \right\}$$

Now, if we have a subgroup, there exists an efficient classical algorithm for computing its annihilator, which basically amounts to solving some system of equations. But what if we only have the annihilator of a subgroup? In that case, we can exploit the following fact.

Exercise* 12.20 Assuming we have labelled classical phases such that:

$$\begin{array}{c} \textcircled{\kappa_h} \\ | \\ \textcircled{\kappa_g} \end{array} = \begin{array}{c} \textcircled{\kappa_g} \\ | \\ \textcircled{\kappa_h} \end{array}$$

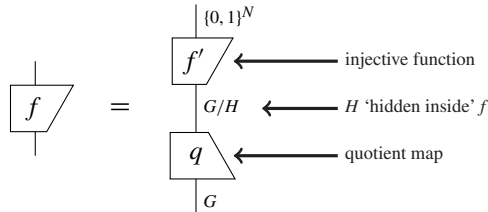
show that $(H')' = H$. (Hint: prove that $H \subseteq (H')'$ and that H and $(H')'$ are the same size. For the latter, first show that H' is the same size as G/H .)

Voila! After not many uses of the oracle, our quantum measurement gives us the *generators* of H' , i.e. enough elements to obtain any element in H' via group-sums. Then, we can use some classical post-processing to compute generators for $(H')'$, which by Exercise* 12.20, is H . In summary:

Given. A commutative group G and a function:

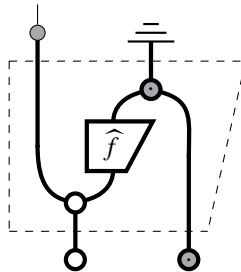
$$f : G \rightarrow \{0, 1\}^N$$

with a promise that there exists a subgroup $H \subseteq G$ such that:



Problem. Find H .

Quantum algorithm. Perform:



and obtain outcome:



for $g \in H'$. Repeat until we obtain a generating set for H' . Use this set to (classically) compute $(H')' = H$.

So how do we go from here to factoring? Suppose we start with a function:

$$f : \mathbb{Z} \rightarrow \{0, 1\}^N$$

and we run the hidden subgroup algorithm to discover the subgroup H is:

$$\{kr \bmod D\}_k \subseteq \mathbb{Z}$$

that is, the group consisting of all the multiples of r , modulo D . Then, we have discovered that f has *period* r ; i.e. for all x :

$$f(x) = f(x + r)$$

But why is that interesting? Well, if we can find the period of:

$$f(x) := a^x \bmod D$$

for randomly chosen values of a , then we can efficiently factor D ! To see this, suppose the function above has period r :

$$a^x = a^{x+r} \pmod{D}$$

Then after a bit of algebra we have:

$$a^r - 1 = 0 \pmod{D}$$

Now, if we're only a little bit lucky, r is an even number, so letting $b := a^{r/2}$, we have:

$$b^2 - 1 = 0 \pmod{D}$$

Factoring the LHS gives:

$$(b + 1)(b - 1) = 0 \pmod{D}$$

So D divides the product of $b + 1$ and $b - 1$. This means one of two things is true: either D divides $b + 1$ or $b - 1$ or these both contain non-trivial factors of D . If the latter is true (which is at least as likely as the former), we can recover a factor efficiently as the greatest common divisor of $b + 1$ and D .

Remark 12.21 The careful reader will note that there was one small problem with this derivation: the commutative group \mathbb{Z} is not finite! However, if we choose some q very large, then we do this algorithm with a function $f : \mathbb{Z}_q \rightarrow \{0, 1\}^N$ from the cyclic group \mathbb{Z}_q and get the same result with high probability.

Exercise 12.22 Show that the one-bit Deutsch–Jozsa problem for a function:

$$f : \{0, 1\} \rightarrow \{0, 1\}$$

is an instance of the hidden subgroup problem for f . Show that for $N > 1$ and:

$$f : \{0, 1\}^N \rightarrow \{0, 1\}$$

there is no group-sum one can fix for the set $\{0, 1\}^N$ that makes the Deutsch–Jozsa problem into an instance of the hidden subgroup problem.

12.3 Measurement-Based Quantum Computation

Measurement-based quantum computing (MBQC) is an alternative way to provide universal quantum computation. Rather than stuffing all the computational structure in unitaries, in MBQC all quantum processes are measurements. We first encountered something like this in Section 7.2.2 in the form of gate teleportation. Here we present an MBQC model based on *graph states* (cf. Section 9.4.5) and single-qubit measurements, also known as

the *one-way model* of quantum computation. In this model, a computation consists of three steps:

1. Prepare a graph state.
2. Perform single-qubit measurements, where later measurements can be controlled by previous measurement outcomes, using *feed-forward*.
3. Possibly do some classical post-processing on measurement outcomes.

The key is to exploit the backaction (cf. Section 7.2.1) of single-qubit measurements to (non-deterministically) realise arbitrary quantum effects, which, when applied a graph state, are enough to realise any quantum computation. For example, if we choose measurements that give us the effects:



then we can turn this piece of a graph state:



into any single-qubit unitary:

$$(12.32)$$

Pretty cool, right? In this section, we'll see not only how to recover arbitrary single- and multiqubit gates using measurements, but also how to do it deterministically, using a technique similar to the one we used for quantum gate teleportation.

The advantage of the MBQC paradigm over the circuit model is that, once we have a graph state, all subsequent computation is done using only single-qubit processes. With

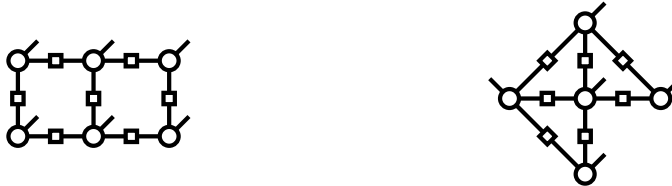
current technology, multiqubit operations like CNOT can be very tricky, and often introduce too much decoherence to be practical for quantum computing. This, combined with the fact that certain graph states are relatively straightforward to prepare in the lab has made MBQC a promising choice for actually implementing quantum computations.

12.3.1 Graph States and Cluster States

We introduced graph states:

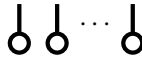


in Section 9.4.5 to prove completeness of the ZX-calculus. MBQC is based on *quantum graph states*, obtained by doubling:

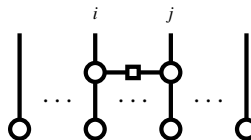


These many-qubit states can be realised using a simple circuit as follows:

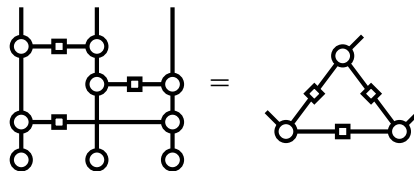
1. Prepare n qubits in the \bigcirc -state:



2. To introduce an edge between the i -th and j -th qubit, apply a CZ-gate:

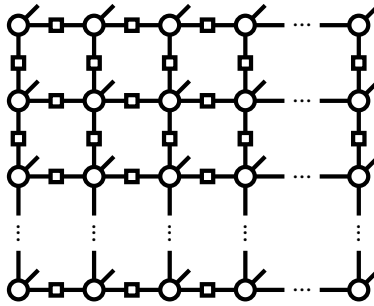


Since the spiders all fuse, it doesn't matter which order we apply CZ-gates. We will always get a graph state:



The most commonly studied type of graph state is a *cluster state*.

Definition 12.23 A two-dimensional cluster state is a graph state whose nodes form an $m \times n$ grid:

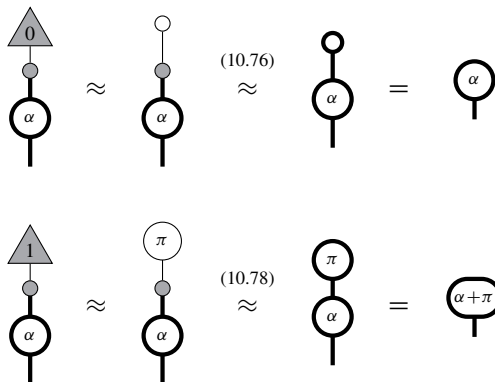


12.3.2 Measuring Graph States

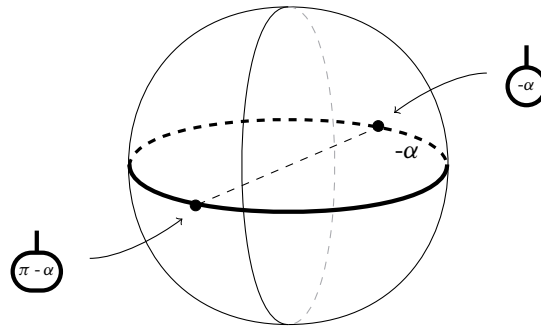
In MBQC, all the magic comes from single-qubit measurements. In fact, it will suffice to consider just two kinds of measurements:



Since an X_α -measurement consists of a \circ -phase of α followed by an X -measurement, its associated quantum effects are:



Hence, an X_α -measurement amounts to a measurement for some ONB on the equator of the Bloch sphere:



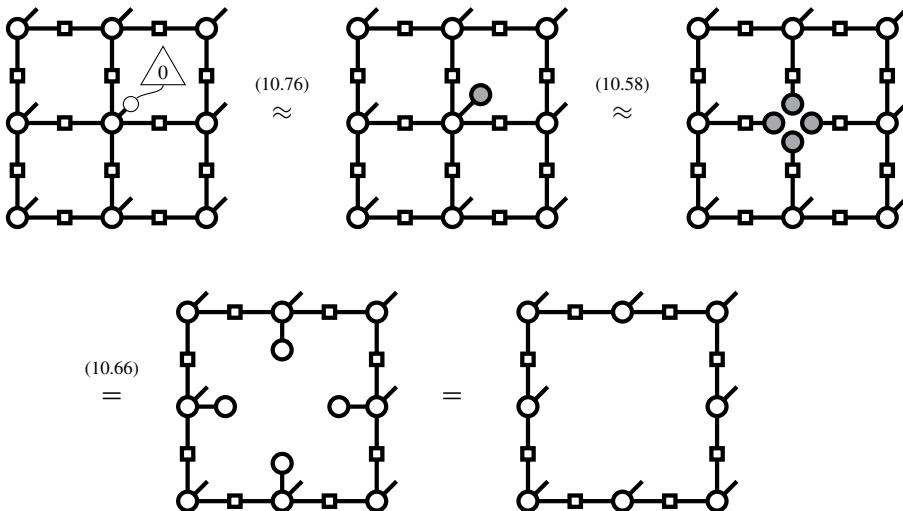
As a special case, X_0 is of course just a normal X -measurement.

The utility of an X_α -measurement is to introduce phases into a graph state. For example, if we measure a single qubit and get outcome 0, we will introduce a \circ -phase of α :

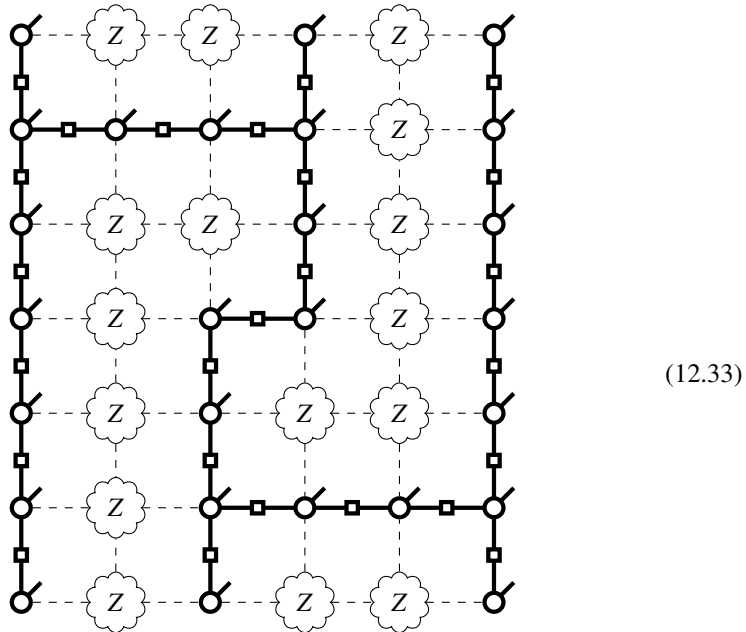
$$\begin{array}{c} \vdots \\ | \\ \text{---} \circ \text{---} \\ | \\ \vdots \end{array} \begin{array}{c} \circ \\ \alpha \end{array} = \begin{array}{c} \vdots \\ | \\ \text{---} \circ \text{---} \\ | \\ \vdots \end{array} \begin{array}{c} \alpha \end{array}$$

If we get outcome 1, this produces a phase of $\alpha + \pi$ instead of α , which we treat as an error that will need to be corrected later. We will see how to do this in the next section. By carefully selecting where we perform X_α -measurements, we can produce \bullet -phases as well as \circ -phases, as we did in (12.32) to realise an arbitrary single-qubit unitary.

Whereas X_α -measurements introduce phases, Z -measurements can be used to cut out unwanted qubits from a graph state. For example, if we perform a Z -measurement on a two-dimensional cluster state and get outcome 0, this leaves a hole:



Repeating this process many times, we can carve out arbitrary shapes:



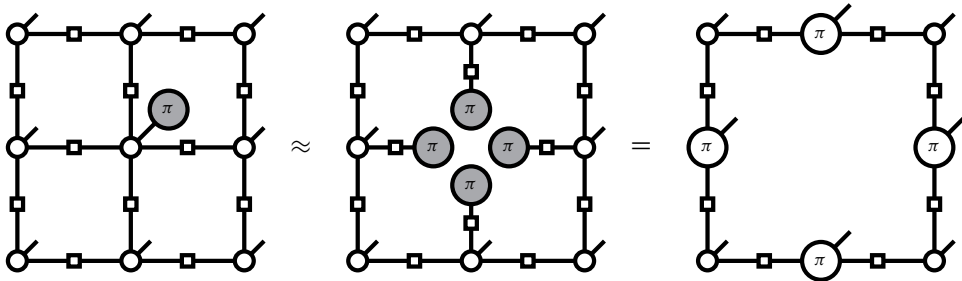
As in the case of X_α -measurements, we should treat outcome 1 as an error, which in this case will introduce extra π phases on all of the neighbours of the deleted qubit. We will now see how to correct both kinds of error using a technique called *feed-forward*.

12.3.3 Feed-Forward

In quantum teleportation (and quantum gate teleportation) we achieve an overall deterministic process by matching unitary corrections up with measurement outcomes. In some sense, MBQC is a vast, many-qubit generalisation of quantum gate teleportation. Thus, corrections are the key to eliminating errors and obtaining a deterministic computation. The ONB-measurements from the previous section will introduce the following effects:

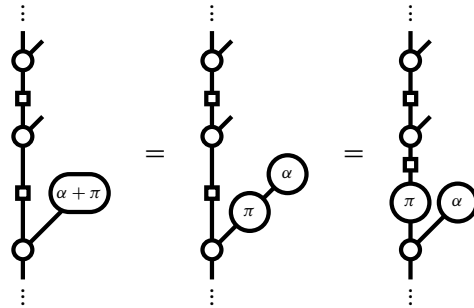
$$\text{Z-measurement: } \left\{ \begin{array}{c} \text{cloud} \\ \text{with } \kappa \end{array} \right\}_\kappa \quad X_\alpha\text{-measurement: } \left\{ \begin{array}{c} \text{cloud} \\ \text{with } \alpha + \kappa \end{array} \right\}_\kappa$$

where $\kappa \in \{0, \pi\}$. If $\kappa = \pi$, we treat this as an error, which we can correct by applying phase gates to qubits near the one we measured. For a Z-measurement, we obtain:

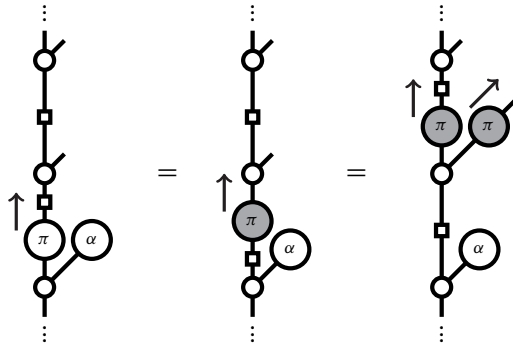


Hence we can correct the error by simply applying a \odot -phase of π to all the neighbours of the qubit we measured.

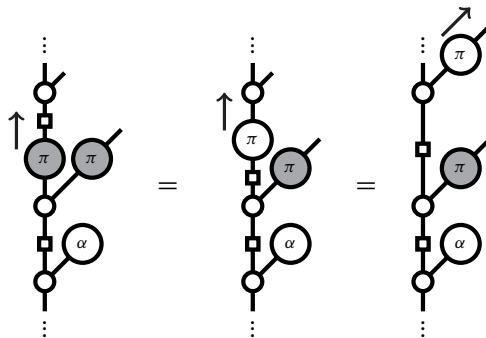
In the case of an X_α -measurement, we can make the correction by ‘pushing’ the error along a graph state until it only appears on output wires, where it can be corrected. Let’s see how this works for just a single measurement on a chain of three qubits. Measuring the first qubit produces an error of π . Using the spider-fusion rules, we can shift this error onto an edge of the graph state:



Now, using colour-change, π -copy, and spider rules, the π can be pushed upwards. After passing through the second qubit, an error still remains on the graph state:



but after passing through the third qubit, the error only occurs on output wires:



So, chaining everything together we obtain:

$$(12.34)$$

We can therefore apply corrections on the second and third qubits to correct the error from the first measurement:

$$(12.35)$$

which yields an α phase gate deterministically.

If we are also measuring the second and third qubits, we get these corrections for free, simply by adjusting our later measurement choices:

$$(12.34) = (10.86)$$

Of course, these measurements could also produce errors, which also need to be corrected.

Exercise 12.24 How would you correct an error for an X_α measurement in a cluster state? What's the general rule for any graph state?

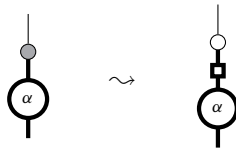
We can obviously adjust measurement choices only for measurements that haven't happened yet. So, the order in which we choose to perform measurements has an effect on when (and if) we make these corrections.

Exercise* 12.25 Can we measure the qubits in a cluster state in some order such that it is possible to feed-forward all corrections? How about for any graph state?

12.3.4 Feed-Forward with Classical Wires

Feed-forward seems to work pretty well, but somewhere along the way, we started working with effects rather than measurements, so we lost the classical wires and hence the explicit flow of classical data. Can we get them back? Of course! Here's one way to do this.

First, we make all classical data the same colour by making a minor modification to the X_α -measurement from before:

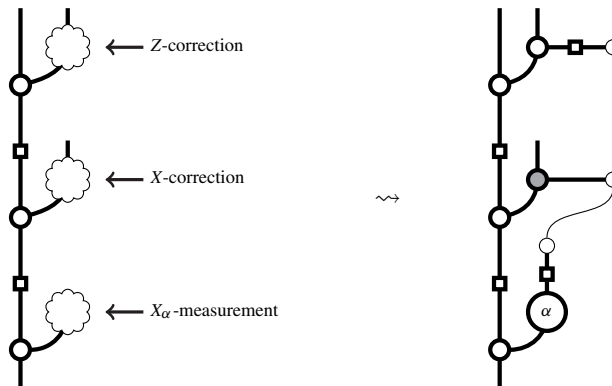


For corrections, we define cq-maps that, depending on a classical input bit, will either apply a π phase or do nothing. The two kinds of corrections we need are:

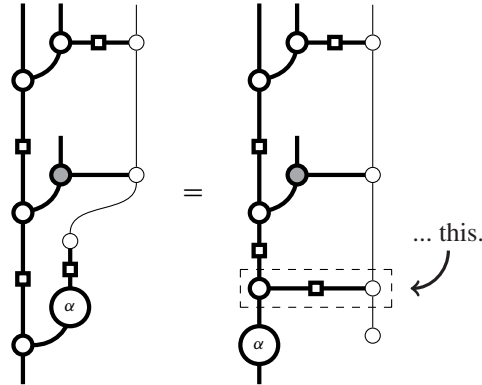


Note that these are nearly the same as the corrections (10.47), which were used for quantum teleportation before. The only differences are making the classical data the same colour to match the associated measurements and keeping a copy of the classical data, rather than deleting it. We'll see soon why we do this.

Now, the correction procedure given by (12.35) translates as follows:



The measurement result is being fed-forward along a classical wire, and used twice to make the two corrections. Now, we will see that we can still reason in the same way as before, even with all the extra wires around. The only difference is, rather than pushing the π through the diagram as we did in (12.35), the thing we should now try to push through the diagram is ...



As before, there are a few crucial moves we use to push the error out. The first kind of move commutes a correction past an H -gate, which changes its (quantum) colour:

$$\begin{array}{c} \text{Diagram 1} \end{array} \stackrel{(10.81)}{=} \begin{array}{c} \text{Diagram 2} \end{array} \quad \begin{array}{c} \text{Diagram 3} \end{array} \stackrel{(10.81)}{=} \begin{array}{c} \text{Diagram 4} \end{array} \quad (12.36)$$

The second kind slides a Z -correction through a \bigcirc -spider:

$$\begin{array}{c} \text{Diagram 1} \end{array} = \begin{array}{c} \text{Diagram 2} \end{array} \quad (12.37)$$

which is just quantum spider fusion. The third kind copies an X -correction through a \bullet -spider:

$$\begin{array}{c} \text{Diagram 1} \end{array} \approx \begin{array}{c} \text{Diagram 2} \end{array} \quad (12.38)$$

This rule follows from strong complementarity:

$$\begin{array}{c} \text{Diagram 1} \end{array} \stackrel{(10.59)}{\approx} \begin{array}{c} \text{Diagram 2} \end{array} = \begin{array}{c} \text{Diagram 3} \end{array} = \begin{array}{c} \text{Diagram 4} \end{array}$$

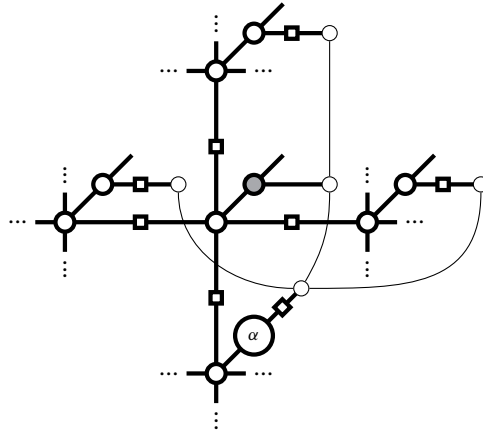
$$\begin{array}{c} \text{---} \bullet \text{---} \bullet \text{---} \\ \text{---} \bullet \text{---} \bullet \text{---} \end{array} \approx \begin{array}{c} | \\ | \end{array} \qquad \begin{array}{c} \text{---} \bullet \text{---} \square \text{---} \bullet \text{---} \\ \text{---} \bullet \text{---} \square \text{---} \bullet \text{---} \end{array} \approx \begin{array}{c} | \\ | \end{array} \qquad (12.39)$$

$$\begin{array}{c} \bullet \\ \bullet \end{array} \begin{array}{c} \circ \\ \circ \end{array} = \begin{array}{c} \bullet \\ \bullet \end{array} \begin{array}{c} \circ \\ \circ \end{array} \approx \begin{array}{c} | \\ | \end{array} \quad (10.53)$$

$$\begin{array}{c} \circ \\ \circ \end{array} \begin{array}{c} \bullet \\ \bullet \end{array} = \begin{array}{c} \bullet \\ \bullet \end{array} \begin{array}{c} \circ \\ \circ \end{array} \stackrel{(10.81)}{=} \begin{array}{c} \bullet \\ \bullet \end{array} \begin{array}{c} \circ \\ \circ \end{array} \stackrel{(10.53)}{\approx} \begin{array}{c} | \\ | \end{array} \stackrel{(10.82)}{=} \begin{array}{c} | \\ | \end{array}$$

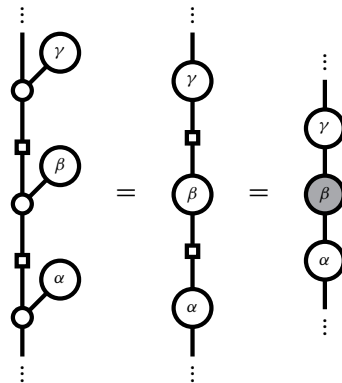
Downloaded from <https://www.cambridge.org/core>. Bodleian Libraries of the University of Oxford, on 18 May 2021 at 16:47:35, subject to the Cambridge Core terms of use, available at <https://www.cambridge.org/core/terms>. <https://doi.org/10.1017/9781316219317.013>

Exercise 12.26 Use the feed-forward rules to prove that the following diagram yields a phase of α on the bottom qubit, deterministically:

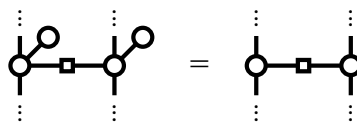


12.3.5 Universality

As we've seen, we can build arbitrary single-qubit unitaries like this:

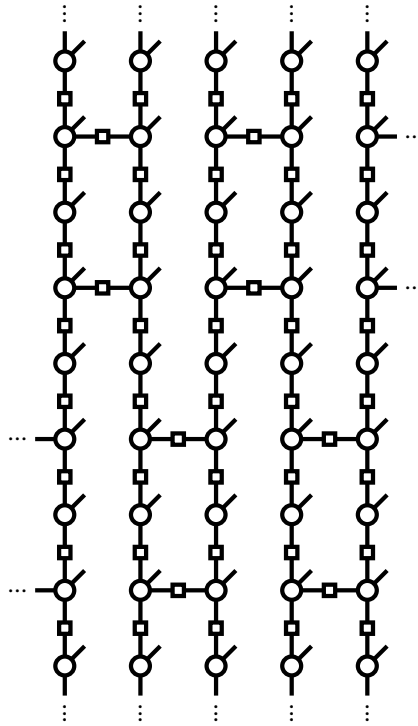


and it's even easier to produce CZ-gates:

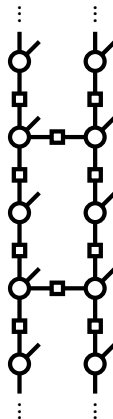


So we have, in principle, everything we need for universal quantum computation (cf. the construction throughout Section 12.1). But how do we put it all together into a circuit? One solution is to start with a big cluster state and 'carve out' the shape of the circuit we want by means of Z-measurements, as in (12.33). Then, we can use X_α -measurements to fill in all the phases. This will work, and in fact, it was how universality was originally shown.

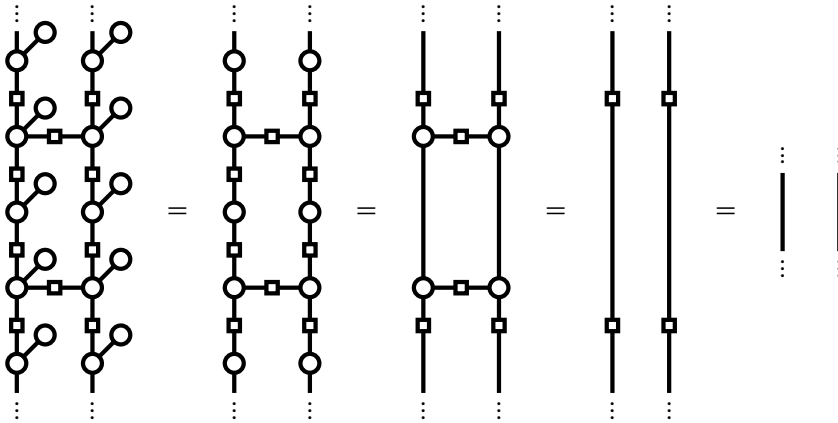
However, if we are more clever about what kind of graph state we start with, we might be able to find something that's already universal using only X_α measurements. In other words: no carving necessary. This has the appealing feature that the entire computation now consists only of a list of angles. Such a state does exist, called the *brickwork state*:



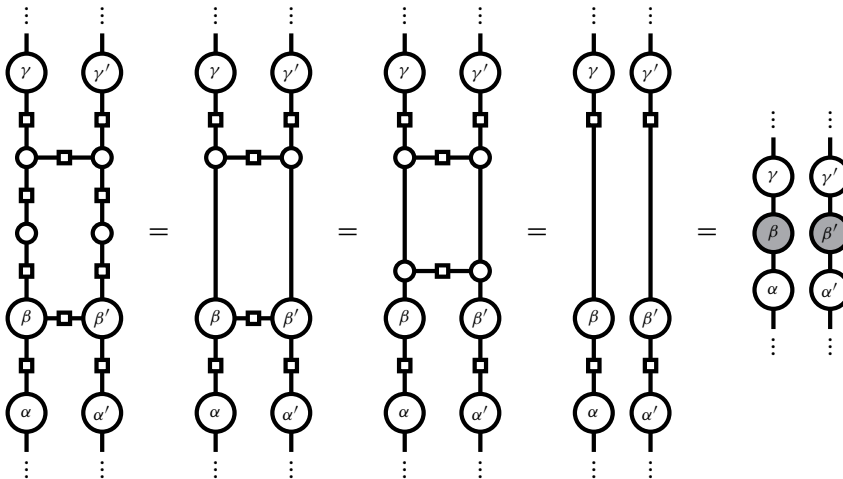
It consists of a repeating pattern of 'bricks':



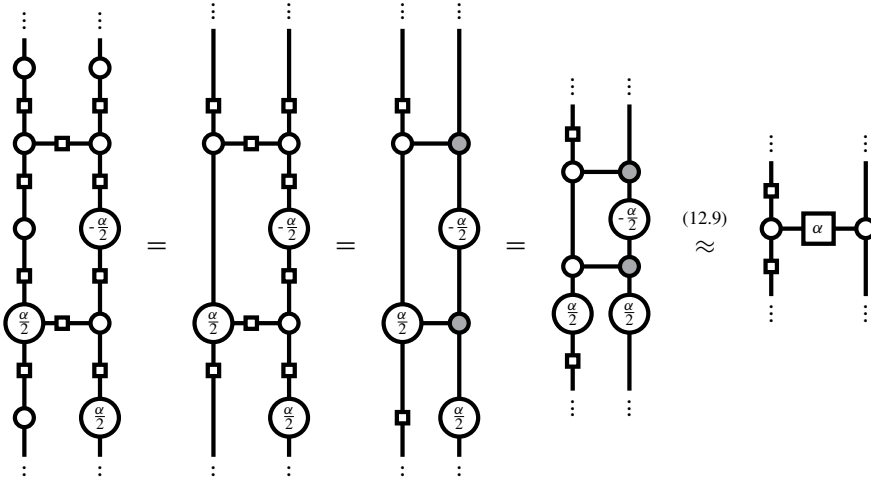
which can be made to do our bidding solely by choosing angles. If we choose 0 everywhere, a brick does nothing at all:



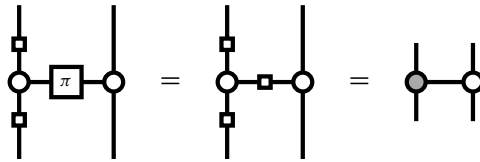
Whereas if we do this:



the result is two arbitrary single-qubit unitaries. And now to seal the deal. If we could turn a brick into a CZ-gate, we would of course be done already, since CZ-gates plus single-qubit gates are universal. But as we saw above, putting 0-phases everywhere will create two CZ-gates, which cancel out. However, if we do something a bit more clever:

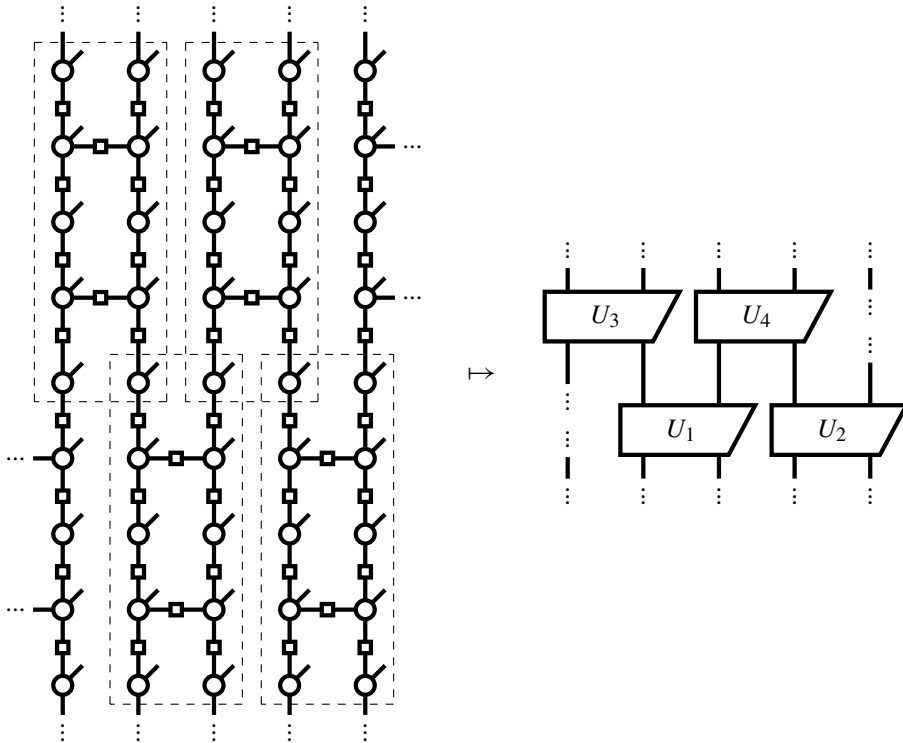


a controlled-phase gate appears. That should do the trick! Indeed, choosing $\alpha := \pi$, this particular brick gives us a CNOT:



Hence we have a universal set of quantum gates.

In order to assemble these gates into arbitrary circuits, we can choose measurement angles for each of the bricks (and summing the angles when the bricks overlap). Then, the whole brickwork state can be turned into a circuit of interleaved quantum gates:



which is indeed enough to produce any circuit.

Exercise 12.27 Show that it is possible to turn an interleaved circuit consisting of single qubit unitaries and CNOT into an arbitrary circuit.

The final piece of the puzzle is that we should not only be able to recover any circuit, we should be able to do it deterministically. For this to be the case, there should exist some order for measuring the qubits such that we can always feed errors forward onto qubits we haven't measured yet.

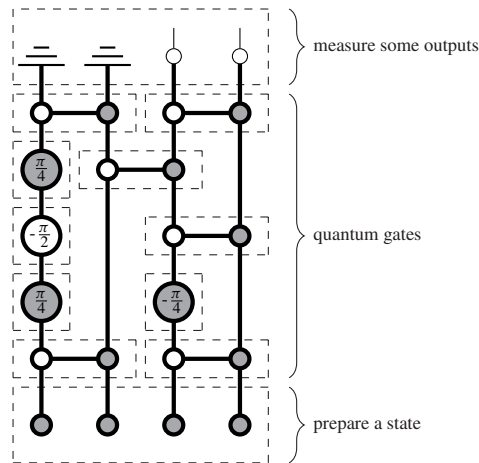
Exercise 12.28 Show that any error on row k can be fixed by applying corrections only on rows $k + 1$ and $k + 2$.

Hence, it is possible to obtain any circuit deterministically by measuring qubits row by row, and therefore we have the following.

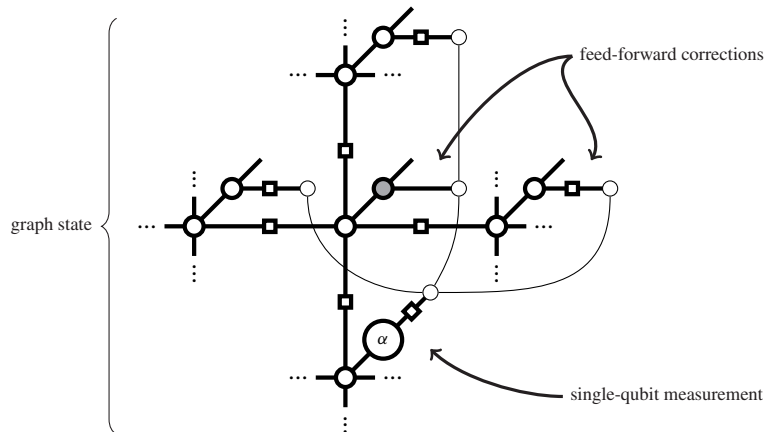
Theorem 12.29 MBQC with brickwork states and X_α -measurements is universal for quantum computation.

12.4 Summary: What to Remember

- Two important models of quantum computation are the *quantum circuit model*:

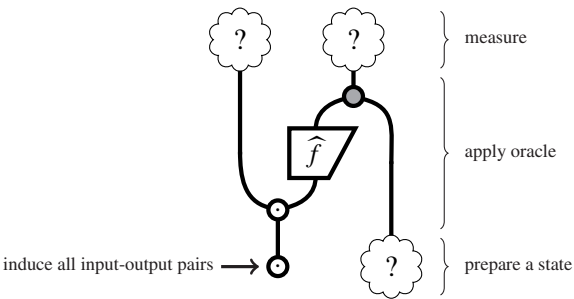


and *measurement-based quantum computation*:



Both are *universal*, in the sense that they can implement any unitary from qubits to qubits.

- Any function map can be turned into a unitary quantum map called a *quantum oracle*, using a complementary pair of spiders. These form the basis of most quantum algorithms:



3. Most quantum algorithms solve *promise problems*. We covered these ones:

Algorithm	Promise	Problem	Diagram
Deutsch–Jozsa	f is constant or balanced.	Which is it?	
Quantum search	f maps one in four bit strings to 1.	Find i such that $f(i) = 1$.	
Hidden subgroup	f factors through G/H for some subgroup $H \subseteq G$.	Find H .	

A special case of the hidden subgroup problem is *period finding*, which allows efficient factorisation of large numbers.

12.5 Historical Notes and References

The first hints towards quantum computing were given by Paul Benioff (1980), where quantum mechanical models of computers were described, and by renowned mathematician

Yuri Manin (1980), who proposed the idea of quantum computing. This was, however, still at the time of the Cold War, and it was written in Russian. Therefore, many attribute quantum computing to Richard Feynman due to a talk he gave at MIT in 1981. (Note that this is the second time in this book that we find Feynman in a scooping mode.) The first universal quantum computer was described by David Deutsch (1985), using the notion of a *quantum Turing machine*. This has since largely been superseded by the (simpler) circuit model, also due to Deutsch (1989). The first proof of universality of the circuit model was given in Barenco et al. (1995). The proof we give is based (partially) on a substantially simpler proof from Shende et al. (2006).

The Deutsch–Jozsa algorithm appeared in Deutsch and Jozsa (1992); Shor’s factoring algorithm appeared in Shor (1994, 1997); and the quantum search algorithm first appeared in Grover (1996). The hidden subgroup algorithm, along with its encoding of Shor’s factoring algorithm and Simon’s problem, was given by Jozsa (1997). A common misconception in the quantum computing community is that the Deutsch–Jozsa algorithm arises as a special case, but as highlighted in Exercise 12.22, this is only the case when $N = 1$. Recent surveys of quantum algorithms include Ambainis (2010) and Montanaro (2015).

Applications of quantum pictorialism to quantum circuits include Boixo and Heunen (2012) and Ranchin and Coecke (2014), which includes the solution to Exercise* 12.1. The diagrammatic treatment of quantum algorithms was introduced by Vicary (2013) and further developed in Zeng and Vicary (2014) and Zeng (2015). The diagrammatic derivation of the hidden subgroup algorithm is given by Gogioso and Kissinger (2016).

The one-way MBQC model was first proposed by Raussendorf and Briegel (2001) and further elaborated on in Raussendorf et al. (2003). Graph states were first proposed in Hein et al. (2004). The diagrammatic treatment was first proposed in Coecke and Duncan (2008, 2011) and elaborated in Duncan and Perdrix (2010). Existing techniques in MBQC were improved upon using the ZX-calculus in Duncan and Perdrix (2010); Horsman (2011) gives a ZX-calculus-based presentation of *topological MBQC* (Raussendorf et al., 2007). A survey of MBQC in the ZX-calculus is Duncan (2012).

While practical quantum computers do not exist yet, there are notable achievements in the lab. The first implementation of a quantum algorithm on a three-qubit nuclear magnetic resonance quantum computer was realised in Oxford by Jones et al. (1998). In 2000, a five-qubit one was realised in Munich, as well as a seven-qubit one at Los Alamos. The fidelity of quantum computations has improved as well. For example, single- and double-qubit gates have been implemented up to a 99.9% fidelity using trapped ions by Ballance et al. (2016). The first realisation of MBQC was done in Vienna by Walther et al. (2005).