Chapter 3

The Model

In this chapter we will describe the model of a quantum computer, what a quantum algorithm is, and what makes such an algorithm efficient. We will first give a short explanation of how the model is based on quantum mechanics, followed by a more detailed explanation of the model itself.

The model of a quantum computer is based on the (five) postulates of quantum mechanics, which are in turn based on the mathematics of Hilbert spaces. The first postulate says that the state of a system is described by a unit length vector in a Hilbert space. The second says that the evolution of the state is unitary. The third postulate states that the process of measuring a physical quantity, such as energy, position, momentum or angular momentum, has an associated linear Hermitian operator and that the results of the measurement are its eigenvalues. For simplicity we will only use one kind of measurement, measurements in the computational basis. These elements of the model are described in detail in Section 3.1. The model for the quantum computer then adds the assumption that

we can always start in a fixed state, the all zero state, and restricts the unitary operators to be "local". The latter condition is based on what is thought to be efficiently implementable physically. These restrictions are described in the next section.

3.1 Quantum Algorithms

A quantum algorithm starts with n qubits initially set to zero, evolves them using a unitary transformation, and then a measurement of the bits is performed. The efficiency of the algorithm is measured in terms of the complexity of the unitary transformation performed. We will now describe the setup in detail.

A quantum state is held in n qubits. A qubit is a unit vector in the Hilbert space $\mathbb{C}^2$ (with the inner product as in Chapter 2). The space has the standard basis $\{|0\rangle, |1\rangle\}$ (written in Dirac's ket notation), so an arbitrary state is written as $\alpha|0\rangle + \beta|1\rangle$, where $\alpha, \beta \in \mathbb{C}$. This can be thought of as a superposition of the two classical states. A state with n qubits is a norm one vector in the Hilbert space $\mathbb{C}^{2^n} = \mathbb{C}^2 \bigotimes \cdots \bigotimes \mathbb{C}^2$, where the tensor product has n terms. The inner product in this case is defined by $\langle a \otimes c, b \otimes d\rangle = \langle a, b\rangle \cdot \langle c, d\rangle$. This space has the standard basis of a tensor product of spaces, which is the set $\{|b_1\rangle \otimes \cdots \otimes |b_n\rangle \ | b_i \in \{0, 1\}\}$. We will write the tensor products of two states $|v\rangle$ and $|w\rangle$ as $|v\rangle \otimes |w\rangle$, $|v\rangle|w\rangle$, or $|v, w\rangle$. The set of binary strings of length n is then written as $\{|b_1 \cdots b_n\rangle | b_i \in \{0, 1\}\}$. This is referred to as the computational basis. An arbitrary state of this space has the form $\sum_{i \in \{0,1\}^n} v_i |i\rangle$, where $v_i \in \mathbb{C}$ and $\sum_i |v_i|^2 = 1$. This can be thought of as a superposition of all possible n bit classical states.

The second postulate says that evolution is unitary. The notion of complexity

arises from what kind of unitary map is used. A unitary transformation is simple if it operates on two qubits, that is, if it can be written as $U_{jk} \otimes I$, where U_{jk} is an arbitrary unitary operation on bits j and k, and the identity is performed on the rest of the bits. The complexity of an operation $U = U_1 \cdots U_k$, where each U_i is simple, is k quantum steps.

A measurement of the qubits of a state $\sum_{x \in \{0,1\}^n} v_x |x\rangle$ results in x with probability $|\alpha_x|^2$, and if x is measured, collapses to the state $|x\rangle$. Partial measurements of, say, one qubit, are also possible. In this case the state collapses to the state that is consistent with the measured bit, as shown in the examples below.

Before moving on to the algorithmic part, applying unitary operators, we will first give some examples of states and measurements.

Example 3.1 (States and Measurements)

- $|0\rangle$ We assume the starting state has all bits zero and that we have as many bits as we want.
- $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ The Bell state. Suppose we measure the first bit. Then we see a zero with probability 1/2 and if we do see zero, the state obtained after measurement is $|00\rangle$. Similarly, we see a one with probability 1/2 and if we see a one then the state obtained after measurement is $|11\rangle$.
- $\frac{1}{\sqrt{3}}(|000\rangle + |001\rangle + |010\rangle)$ If we measure the first bit, we will see a zero with probability 1, and the resulting state is the same. Measuring the second bit results in a zero with probability 2/3, resulting in the state $\frac{1}{\sqrt{2}}(|000\rangle + |001\rangle)$ if a zero is seen. A one is measured with probability 1/3, resulting in the state $|010\rangle$ if a one is seen.

■

We will now give some examples of unitary operators applied to various states. The main construction used below is called a controlled-U. The idea is that if we start with a unitary transformation U that operates on n bits, and we have an extra bit (or set of bits), we can apply U conditioned on the other bit. More precisely, if $|v\rangle$ is a state on n bits, and U operates on n bits, then the following map U' is a unitary operator and extends linearly:

$$U' : \quad |0\rangle|v\rangle \mapsto |0\rangle|v\rangle$$

$$|1\rangle|v\rangle \mapsto |1\rangle(U|v\rangle).$$

This map is used in the examples below. It may be helpful to notice why each example is reversible.

Example 3.2 (Unitary Operators on States)

- The controlled-not. Exclusive-ors (addition mod 2) one bit into the next: $|a, b\rangle \rightarrow |a, a \oplus b\rangle$. This operation replaces writing a bit to memory in the classical case by keeping it reversible. It is called a controlled-not because it flips the second bit iff the first bit (the control bit) is one.

- Evaluating an oracle. Using the controlled-not we can define an oracle evaluation U_f by $|x, y\rangle \xrightarrow{U_f} |x, y \oplus f(x)\rangle$. Notice that repeating this returns to the original state. We will usually assume $y = 0$, and will write $|x\rangle \longrightarrow |x, f(x)\rangle$

- Controlled-rotation. Change the phase of a basis vector $|p, i\rangle \longrightarrow \omega_p^i |p, i\rangle$, where i and p are positive integers. Here the state $|p, i\rangle$ is the control that specifies what phase

to use. Applied to a superposition we get $\sum_{i=0}^{p-1} v_i|i\rangle \longrightarrow \sum_{i=0}^{p-1} v_i\omega_p^i|i\rangle$ (when p is a constant throughout the algorithm, we will not write it down for clarity).

- The Fourier transform F_p, where p is a positive integer. Given a basis state $|c\rangle$, $F_p|c\rangle = \frac{1}{\sqrt{p}} \sum_{i=0}^{p-1} \omega_p^{ic}|i\rangle$. Measuring all the bits results in a uniformly distributed random number in $\{0, \ldots p-1\}$.

- The Fourier transform over any finite group is a unitary operator.

■

We will now describe efficiently implementable unitary transformations. A nice way to describe the allowed operations is to use quantum circuits. Recall that for concreteness, classical algorithms can be described by circuits, using gates like AND, OR, and NOT. Each gate has some wires as inputs and a wire as an output. For example, AND could take two bits b_1 and b_2 and would output $b_1 \wedge b_2$. Many gates are connected together so that they compute the output of the algorithm. The input to the algorithm is placed on the inputs to the circuit, the rules of the gates are applied, and the output bits contain the answer. The quantum case is similar, however the gates must be unitary. The condition of unitarity on the quantum gate forces it to have the same number of output wires as input wires. So quantum gates will have some number of inputs and outputs, which will be qubits. The action of the gates can be described by unitary matrices indicating what happens on the input bits. Here are a couple of simple examples.

Example 3.3 1. NOT. The NOT gate has one qubit as input and output, and flips its

value. It is described by the 2×2 matrix $N = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. On the basis states, this performs the maps $|0\rangle \xrightarrow{N} |1\rangle$ and $|1\rangle \xrightarrow{N} |0\rangle$. As is always the case, the operation on arbitrary superpositions can be computed using the linearity of the operation.

2. Controlled-not. A controlled-not has two qubits as inputs and outputs, and performs a NOT on the second bit iff the first bit (the control bit) is one. It is described by the 4×4 matrix

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

On the basis states this performs the maps $|0b\rangle \xrightarrow{C_N} |0b\rangle$ and $|1b\rangle \xrightarrow{C_N} |1\overline{b}\rangle$, where $b \in \{0, 1\}$ and $\overline{b}$ is not-b.

3. Controlled-rotation. "Rotation" refers to changing the phase of a state, and operates on one bit. Its matrix, for some $t \in \mathbb{Q}$, is $C_{R_t} = \begin{pmatrix} 1 & 0 \\ 0 & \omega^t \end{pmatrix}$. Recall that $\omega^t = e^{2\pi i t}$. This performs the maps $|0\rangle \xrightarrow{C_{R_t}} |0\rangle$ and $|1\rangle \xrightarrow{C_{R_t}} \omega^t |1\rangle$.

4. Hadamard transform. The Hadamard transform is a gate with one qubit as input and output. Its matrix is $H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$. On the basis states, this performs the maps $|0\rangle \xrightarrow{H} \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle \xrightarrow{H} \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. This is also the Fourier transform F_2 over the group $\mathbb{Z}_2$.

■

Example 3.4 Classical computation. Performing a classical computation corresponds to a unitary transformation that is a permutation of the basis states. As the basis states must be permuted, the classical computation must be reversible. All classical computations can be simulated reversibly as long as the input is kept on the tape. Suppose some function f can be computed efficiently. Then it is possible to compute $|x\rangle \longrightarrow |x, f(x)\rangle$. If f^{-1} can be computed, then the computation $|x, f(x)\rangle \longrightarrow |0, f(x)\rangle$ is also possible. As an example it may be helpful to think of the function multiply: $|p, q\rangle \longrightarrow |p, q, pq\rangle$. To erase p and q it must be possible to efficiently compute them from pq. It is only known how to do this using the quantum factoring algorithm. ■

A polynomial number of gates are put together in a sequence to compute the output of the algorithm. The set of basic gates that we will allow are all one bit gates (i.e., all 2×2 unitary matrices), and the controlled-not. This set allows any polynomial time quantum computation to be described by a polynomial size circuit.

We will not describe operations by matrices, since they can get very large, and which bits they operate on must be specified in some way. Instead we will continue to use the other notation $|v\rangle \xrightarrow{U} U|v\rangle$. We will give some more examples of efficient operations, and any sequence of at most a polynomial in them will also be efficient.

Notice that because the gates can act on at most two bits at a time, computation is local in some sense. One can imagine being at one of the computational basis states $|i\rangle$ and being allowed to perform an operation of the type "if the first bit is one, then do..., otherwise do nothing." We will now give some more examples of operations that can be efficiently implemented.

Example 3.5 Controlled-rotation. Here we have a more complicated version than before. It is possible to implement the map $|x\rangle \longrightarrow \omega^{x/p}|x\rangle$, where x and p are positive integers. Note that this would require a matrix much larger than the previous example, because it conditions on all n bits instead of just two. ■

Example 3.6 Controlled-U. Fix some value y, and define the map $|y\rangle \longrightarrow U|y\rangle$, and $|x\rangle \longrightarrow |x\rangle$ if $x \neq y$. ■

Example 3.7 The Fourier transform over the group $\mathbb{Z}_2^n$. As mentioned before, the Fourier transform over any group is a unitary operator. We want to describe it in terms of a small circuit. Since $F_{\mathbb{Z}_2^n} = \bigotimes_{i=0}^{n-1} F_2$, we can refer to the previous example, and just apply the Hadamard transform to each bit. Given a basis state $|i\rangle = |i_1 \cdots i_n\rangle$, where the i_j are the bits of i, we have

$$|i\rangle = \bigotimes_{j=0}^{n-1} |i_j\rangle \stackrel{\otimes F_2}{\longrightarrow} \bigotimes_{j=0}^{n-1} (|0\rangle + (-1)^{i_j}|i_j\rangle) = \sum_{c=0}^{2^n-1} (-1)^{i \cdot c}|c\rangle,$$

where $i \cdot c$ is the inner product mod 2 of the bits of i and c. ■

A more complicated, but still relatively simple example, is the Fourier transform over $\mathbb{Z}_{2^n}$. The algorithm is as easy to understand as it is in the classical case [17]. The most confusing thing is to index everything properly, also as it is in the classical case. The quantum algorithm only takes about $(\log n)^2$ steps, unlike the classical FFT, which takes $O(n \log n)$. The advantage a quantum computer has is that in the recursion step, it can simultaneously compute all the recursive calls at once.

Example 3.8 The Fourier transform F_{2^n}. The ability to compute this kind of function is exactly why quantum computation gets a speedup over classical computation. Instead

of listing the algorithm, we will try to illustrate why the algorithm is so much faster by showing how it differs from the classical algorithm.

We will first describe the classical algorithm, taken from [17]. We will use ket notation for the vectors for ease of translation. Assume the algorithm is called on $|\alpha\rangle = \sum_{i=0}^{p-1} \alpha_i |i\rangle$, where p is a power of 2. Similarly, $|\beta\rangle$ will be the vector returned. Before listing the code we first illustrate how the recursion step works in both cases.

First rewrite the vector as follows, separating the even and odd coefficients, and then recursively compute the Fourier transform:

$$\begin{pmatrix} \alpha_0 \\ \alpha_2 \\ \alpha_4 \\ \vdots \\ \alpha_{p-2} \end{pmatrix} \xrightarrow{F_{p/2}} \begin{pmatrix} \widehat{\alpha}_0 \\ \widehat{\alpha}_2 \\ \widehat{\alpha}_4 \\ \vdots \\ \widehat{\alpha}_{p-2} \end{pmatrix}$$

$$\begin{pmatrix} \alpha_1 \\ \alpha_3 \\ \alpha_5 \\ \vdots \\ \alpha_{p-1} \end{pmatrix} \xrightarrow{F_{p/2}} \begin{pmatrix} \widetilde{\alpha}_1 \\ \widetilde{\alpha}_3 \\ \widetilde{\alpha}_5 \\ \vdots \\ \widetilde{\alpha}_{p-1} \end{pmatrix}$$

Now we use the results from the recursion step to form the answer:

$$\begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{p/2} \\ \beta_{p/2+1} \\ \vdots \\ \beta_{p-1} \end{pmatrix} = \begin{pmatrix} \widehat{\alpha}_0 \\ \widehat{\alpha}_2 \\ \vdots \\ \widehat{\alpha}_{p-2} \end{pmatrix} + \begin{pmatrix} \omega_p^0 \widetilde{\alpha}_1 \\ \omega_p^1 \widetilde{\alpha}_3 \\ \vdots \\ \omega_p^{p/2-1} \widetilde{\alpha}_{p-1} \end{pmatrix} \begin{pmatrix} \widehat{\alpha}_0 \\ \widehat{\alpha}_2 \\ \vdots \\ \widehat{\alpha}_{p-2} \end{pmatrix} - \begin{pmatrix} \omega_p^0 \widetilde{\alpha}_1 \\ \omega_p^1 \widetilde{\alpha}_3 \\ \vdots \\ \omega_p^{p/2-1} \widetilde{\alpha}_{p-1} \end{pmatrix}$$

Algorithm 3.1 (Recursive-FFT($|\alpha\rangle$,p) (assume p is the length of $|\alpha\rangle$))

1. *if* $p = 1$, *return* $|\alpha\rangle$

2. $t := 0$

3. $\sum_{i=0}^{p/2-1} \widehat{\alpha}_{2i}|2i\rangle :=$ *Recursive-FFT*$(\sum_{i=0}^{p/2-1} \alpha_{2i}|i\rangle)$; *Recurse on even coefficients*

4. $\sum_{i=0}^{p/2-1} \widetilde{\alpha}_{2i+1}|2i+1\rangle :=$ *Recursive-FFT*$(\sum_{i=0}^{p/2-1} \alpha_{2i+1}|i\rangle)$; *Recurse on odd coefficients*

5. *for* $k = 0$ *to* $p/2 - 1$

   (a) $\beta_k := \widehat{\alpha}_{2k} + \omega_p^t \widetilde{\alpha}_{2k+1}$

   (b) $\beta_{k+p/2} := \widehat{\alpha}_{2k} - \omega_p^t \widetilde{\alpha}_{2k+1}$

   (c) *t:=t+1*

6. *return* $|\beta\rangle$

The recursion tree has depth $\log p$ and width p. This algorithm should be compared with the following, which is the recursive step in the quantum algorithm.

$$\sum_{i=0}^{p-1} \alpha_i |i\rangle = \sum_{i=0}^{p/2-1} \alpha_{2i}|i\rangle|0\rangle + \sum_{i=0}^{p/2-1} \alpha_{2i+1}|i\rangle|1\rangle \tag{3.1}$$

$$\xrightarrow{F_{p/2}} \sum_{i=0}^{p/2-1} \widehat{\alpha}_{2i}|i\rangle|0\rangle + \sum_{i=0}^{p/2-1} \widetilde{\alpha}_{2i+1}|i\rangle|1\rangle \tag{3.2}$$

$$\xrightarrow{C-R(\omega_p^t)} \sum_{i=0}^{p/2-1} \widehat{\alpha}_{2i}|i\rangle|0\rangle + \sum_{i=0}^{p/2-1} \omega_p^t \widetilde{\alpha}_{2i+1}|i\rangle|1\rangle \tag{3.3}$$

$$\xrightarrow{F_2} \sum_{i=0}^{p/2-1} (\widehat{\alpha}_{2i} + \omega_p^t \widetilde{\alpha}_{2i+1})|i\rangle|0\rangle + \sum_{i=0}^{p/2-1} (\widehat{\alpha}_{2i} - \omega_p^t \widetilde{\alpha}_{2i+1})|i\rangle|1\rangle \tag{3.4}$$

In (3.1) the equation is rewritten to emphasize that the vector lies in two different subspaces. In (3.2) the Fourier transform is recursively computed on the the first $n-1$ bits. Notice that this simultaneously computes the Fourier transform on both subspaces at once! In (3.3) the phase is added, and in (3.4) the results from the two subspaces are combined. One more minor step must be taken that we did not write, which is to reorder the bits by making the least significant bit the most significant bit. The recursion tree has depth $\log p$, but only has width one.

■

Example 3.9 F_p, where p is a prime. It was first shown how to ϵ-approximate this in [36]. In Chapter 5 we will give a faster and simpler algorithm. ■

The Fourier transform over an arbitrary abelian group can be computed efficiently since it is a tensor product of the Fourier transform over the cyclic factors.