Complementary Lectures: Universality

Javier Orduz

July 21, 2021

Contents

Part I

Lectures about: Quantum Computing

Chapter 1

Models, Turing machine and universality

Next pages contain information about models, Turing machine and universality [1].

1 INTRODUCTION AND BACKGROUND

1.1 Overview

A computer is a physical device that helps us process information by executing algorithms. An algorithm is a well-defined procedure, with finite description, for realizing an information-processing task. An information-processing task can always be translated into a physical task.

When designing complex algorithms and protocols for various information-processing tasks, it is very helpful, perhaps essential, to work with some idealized computing model. However, when studying the true limitations of a computing device, especially for some practical reason, it is important not to forget the relationship between computing and physics. Real computing devices are embodied in a larger and often richer physical reality than is represented by the idealized computing model.

Quantum information processing is the result of using the physical reality that quantum theory tells us about for the purposes of performing tasks that were previously thought impossible or infeasible. Devices that perform quantum information processing are known as *quantum computers*. In this book we examine how quantum computers can be used to solve certain problems more efficiently than can be done with classical computers, and also how this can be done reliably even when there is a possibility for errors to occur.

In this first chapter we present some fundamental notions of computation theory and quantum physics that will form the basis for much of what follows. After this brief introduction, we will review the necessary tools from linear algebra in Chapter 2, and detail the framework of quantum mechanics, as relevant to our model of quantum computation, in Chapter 3. In the remainder of the book we examine quantum teleportation, quantum algorithms and quantum error correction in detail.

1.2 Computers and the Strong Church–Turing Thesis

We are often interested in the amount of *resources* used by a computer to solve a problem, and we refer to this as the *complexity* of the computation. An important resource for a computer is *time*. Another resource is *space*, which refers to the amount of memory used by the computer in performing the computation. We measure the amount of a resource used in a computation for solving a given problem as a function of the length of the input of an instance of that problem. For example, if the problem is to multiply two n bit numbers, a computer might solve this problem using up to $2n^2+3$ units of time (where the unit of time may be seconds, or the length of time required for the computer to perform a basic step).

Of course, the exact amount of resources used by a computer executing an algorithm depends on the physical architecture of the computer. A different computer multiplying the same numbers mentioned above might use up to time $4n^3+n+5$ to execute the same basic algorithm. This fact seems to present a problem if we are interested in studying the complexity of algorithms themselves, abstracted from the details of the machines that might be used to execute them. To avoid this problem we use a more coarse measure of complexity. One coarser measure is to consider only the highest-order terms in the expressions quantifying resource requirements, and to ignore constant multiplicative factors. For example, consider the two computers mentioned above that run a searching algorithm in times $2n^2+3$ and $4n^3+n+7$, respectively. The highest-order terms are n^2 and n^3, respectively (suppressing the constant multiplicative factors 2 and 4, respectively). We say that the running time of that algorithm for those computers is in $O(n^2)$ and $O(n^3)$, respectively.

We should note that $O\left(f(n)\right)$ denotes an *upper* bound on the running time of the algorithm. For example, if a running time complexity is in $O(n^2)$ or in $O(\log n)$, then it is also in $O(n^3)$. In this way, expressing the resource requirements using the O notation gives a hierarchy of complexities. If we wish to describe *lower* bounds, then we use the Ω notation.

It often is very convenient to go a step further and use an even more coarse description of resources used. As we describe in Section 9.1, in theoretical computer science, an algorithm is considered to be *efficient* with respect to some resource if the amount of that resource used in the algorithm is in $O(n^k)$ for some k. In this case we say that the algorithm is *polynomial* with respect to the resource. If an algorithm's running time is in $O(n)$, we say that it is *linear*, and if the running time is in $O(\log n)$ we say that it is *logarithmic*. Since linear and logarithmic functions do not grow faster than polynomial functions, these algorithms are also efficient. Algorithms that use $\Omega(c^n)$ resources, for some constant c, are said to be *exponential*, and are considered not to be efficient. If the running time of an algorithm cannot be bounded above by any polynomial, we say its running time is *superpolynomial*. The term 'exponential' is often used loosely to mean superpolynomial.

One advantage of this coarse measure of complexity, which we will elaborate on, is that it appears to be robust against reasonable changes to the computing model and how resources are counted. For example, one cost that is often ignored when measuring the complexity of a computing model is the time it takes to move information around. For example, if the physical bits are arranged along a line, then to bring together two bits that are n-units apart will take time proportional to n (due to special relativity, if nothing else). Ignoring this cost is in general justifiable, since in modern computers, for an n of practical size, this transportation time is negligible. Furthermore, properly accounting for this time only changes the complexity by a linear factor (and thus does not affect the polynomial versus superpolynomial dichotomy).

Computers are used so extensively to solve such a wide variety of problems, that questions of their power and efficiency are of enormous practical importance, aside from being of theoretical interest. At first glance, the goal of characterizing the problems that can be solved on a computer, and to quantify the efficiency with which problems can be solved, seems a daunting one. The range of sizes and architectures of modern computers encompasses devices as simple as a single programmable logic chip in a household appliance, and as complex as the enormously powerful supercomputers used by NASA. So it appears that we would be faced with addressing the questions of computability and efficiency for computers in each of a vast number of categories.

The development of the mathematical theories of computability and computational complexity theory has shown us, however, that the situation is much better. The *Church–Turing Thesis* says that a computing problem can be solved on *any* computer that we could hope to build, if and only if it can be solved on a very simple 'machine', named a *Turing machine* (after the mathematician Alan Turing who conceived it). It should be emphasized that the Turing 'machine' is a mathematical abstraction (and not a physical device). A Turing machine is a computing model consisting of a finite set of states, an infinite 'tape' which symbols from a finite alphabet can be written to and read from using a moving head, and a transition function that specifies the next state in terms of the current state and symbol currently pointed to by the head.

If we believe the Church–Turing Thesis, then a function is computable by a Turing machine if and only if it is computable by some realistic computing device. In fact, the technical term *computable* corresponds to what can be computed by a Turing machine.

To understand the intuition behind the Church–Turing Thesis, consider some other computing device, A, which has some finite description, accepts input strings x, and has access to an arbitrary amount of workspace. We can write a computer program for our universal Turing machine that will *simulate* the evolution of A on input x. One could either simulate the logical evolution of A (much like one computer operating system can simulate another), or even more naively, given the complete physical description of the finite system A, and the laws of physics governing it, our universal Turing machine could alternatively simulate it at a physical level.

The original Church–Turing Thesis says nothing about the efficiency of computation. When one computer simulates another, there is usually some sort of 'overhead' cost associated with the simulation. For example, consider two types of computer, A and B. Suppose we want to write a program for A so that it simulates the behaviour of B. Suppose that in order to simulate a single step of the evolution of B, computer A requires 5 steps. Then a problem that is solved by B in time $O(n^3)$ is solved by A in time in $5 \cdot O(n^3) = O(n^3)$. This simulation is efficient. Simulations of one computer by another can also involve a trade-off between resources of different kinds, such as time and space. As an example, consider computer A simulating another computer C. Suppose that when computer C uses S units of space and T units of space, the simulation requires that A use up to $O(ST2^S)$ units of time. If C can solve a problem in time $O(n^2)$ using $O(n)$ space, then A uses up to $O(n^3 2^n)$ time to simulate C.

We say that a simulation of one computer by another is *efficient* if the 'overhead' in resources used by the simulation is *polynomial* (i.e. simulating an $O(f(n))$ algorithm uses $O(f(n)^k)$ resources for some fixed integer k). So in our above example, A can simulate B efficiently but not necessarily C (the running times listed are only upper bounds, so we do not know for sure if the exponential overhead is necessary).

One alternative computing model that is more closely related to how one typically describes algorithms and writes computer programs is the random access machine (RAM) model. A RAM machine can perform elementary computational operations including writing inputs into its memory (whose units are assumed to store integers), elementary arithmetic operations on values stored in its memory, and an operation conditioned on some value in memory. The classical algorithms we describe and analyse in this textbook implicitly are described in log-RAM model, where operations involving n-bit numbers take time n.

In order to extend the Church–Turing Thesis to say something useful about the efficiency of computation, it is useful to generalize the definition of a Turing machine slightly. A *probabilistic Turing machine* is one capable of making a random binary choice at each step, where the state transition rules are expanded to account for these random bits. We can say that a probabilistic Turing machine is a Turing machine with a built-in 'coin-flipper'. There are some important problems that we know how to solve efficiently using a probabilistic Turing machine, but do not know how to solve efficiently using a conventional Turing machine (without a coin-flipper). An example of such a problem is that of finding square roots modulo a prime.

It may seem strange that the addition of a source of randomness (the coin-flipper) could add power to a Turing machine. In fact, some results in computational complexity theory give reason to suspect that every problem (including the "square root modulo a prime" problem above) for which probabilistic Turing machine can efficiently guess the correct answer with high probability, can be solved efficiently by a deterministic Turing machine. However, since we do not have proof of this equivalence between Turing machines and probabilistic Turing machines, and problems such as the square root modulo a prime problem above are evidence that a coin-flipper may offer additional power, we will state the following thesis in terms of probabilistic Turing machines. This thesis will be very important in motivating the importance of quantum computing.

(Classical) Strong Church–Turing Thesis: *A probabilistic Turing machine can efficiently simulate any realistic model of computation.*

Accepting the Strong Church–Turing Thesis allows us to discuss the notion of the intrinsic complexity of a problem, independent of the details of the computing model.

The Strong Church–Turing Thesis has survived so many attempts to violate it that before the advent of quantum computing the thesis had come to be widely accepted. To understand its importance, consider again the problem of determining the computational resources required to solve computational problems. In light of the strong Church–Turing Thesis, the problem is vastly simplified. It will suffice to restrict our investigations to the capabilities of a probabilistic Turing machine (or any equivalent model of computation, such as a modern personal computer with access to an arbitrarily large amount of memory), since any realistic computing model will be roughly equivalent in power to it. You might wonder why the word 'realistic' appears in the statement of the strong Church–Turing Thesis. It is possible to describe special-purpose (classical) machines for solving certain problems in such a way that a probabilistic Turing machine simulation may require an exponential overhead in time or space. At first glance, such proposals seem to challenge the strong Church–Turing Thesis. However, these machines invariably 'cheat' by not accounting for all the resources they use. While it seems that the special-purpose machine uses exponentially less time and space than a probabilistic Turing machine solving the problem, the special-purpose machine needs to perform some physical task that implicitly requires superpolynomial resources. The term *realistic model of computation* in the statement of the strong Church–Turing Thesis refers to a model of computation which is consistent with the laws of physics and in which we explicitly account for *all* the physical resources used by that model.

It is important to note that in order to actually implement a Turing machine or something equivalent it, one must find a way to deal with realistic errors. Error-correcting codes were developed early in the history of computation in order to deal with the faults inherent with any practical implementation of a computer. However, the error-correcting procedures are also not perfect, and could introduce additional errors themselves. Thus, the error correction needs to be done in a *fault-tolerant* way. Fortunately for classical computation, efficient fault-tolerant error-correcting techniques have been found to deal with realistic error models.

The fundamental problem with the classical strong Church–Turing Thesis is that it appears that classical physics is not powerful enough to efficiently simulate quantum physics. The basic principle is still believed to be true; however, we need a computing model capable of simulating arbitrary 'realistic' physical devices, including quantum devices. The answer may be a quantum version of the strong Church–Turing Thesis, where we replace the probabilistic Turing machine with some reasonable type of *quantum* computing model. We describe a quantum model of computing in Chapter 4 that is equivalent in power to what is known as a quantum Turing machine.

Quantum Strong Church–Turing Thesis: *A quantum Turing machine can efficiently simulate any realistic model of computation.*

1.3 The Circuit Model of Computation

In Section 1.2, we discussed a prototypical computer (or *model of computation*) known as the probabilistic Turing machine. Another useful model of computation is that of a *uniform families of reversible circuits*. (We will see in Section 1.5 why we can restrict attention to reversible gates and circuits.) Circuits are networks composed of *wires* that carry bit values to *gates* that perform elementary operations on the bits. The circuits we consider will all be *acyclic*, meaning that the bits move through the circuit in a linear fashion, and the wires never feed back to a prior location in the circuit. A circuit C_n has n wires, and can be described by a circuit diagram similar to that shown in Figure 1.1 for $n = 4$. The input bits are written onto the wires entering the circuit from the left side of the diagram. At every time step t each wire can enter at most one gate G. The output bits are read-off the wires leaving the circuit at the right side of the diagram.

A circuit is an array or network of gates, which is the terminology often used in the quantum setting. The gates come from some finite family, and they take

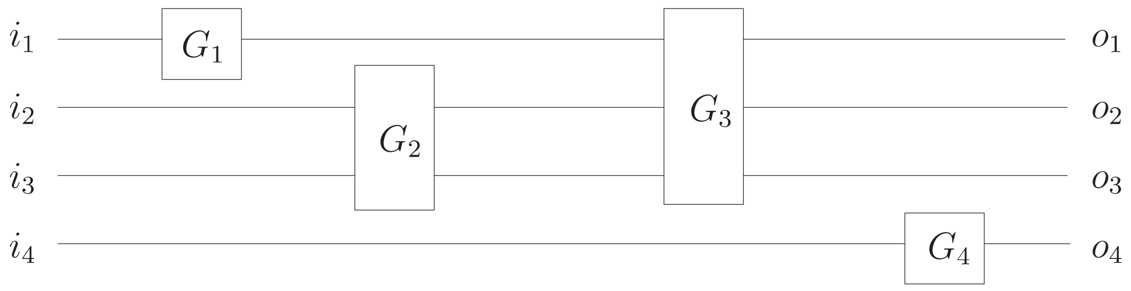


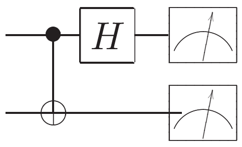
Fig. 1.1 A circuit diagram. The horizontal lines represent 'wires' carrying the bits, and the blocks represent gates. Bits propagate through the circuit from left to right. The input bits i_1, i_2, i_3, i_4 are written on the wires at the far left edge of the circuit, and the output bits o_1, o_2, o_3, o_4 are read-off the far right edge of the circuit.

Chapter 2

Universality

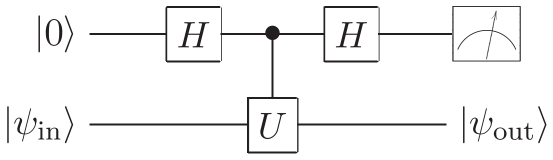
Next pages contain information about universality [2].

in the computational basis. However, often we want to perform a measurement in some other basis, defined by a complete set of orthonormal states. To perform this measurement, simply unitarily transform from the basis we wish to perform the measurement in to the computational basis, then measure. For example, show that the circuit

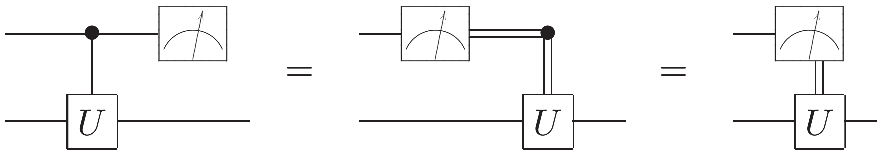


performs a measurement in the basis of the Bell states. More precisely, show that this circuit results in a measurement being performed with corresponding POVM elements the four projectors onto the Bell states. What are the corresponding measurement operators?

Exercise 4.34: (Measuring an operator) Suppose we have a single qubit operator U with eigenvalues ± 1, so that U is both Hermitian and unitary, so it can be regarded both as an observable and a quantum gate. Suppose we wish to measure the observable U. That is, we desire to obtain a measurement result indicating one of the two eigenvalues, and leaving a post-measurement state which is the corresponding eigenvector. How can this be implemented by a quantum circuit? Show that the following circuit implements a measurement of U:



Exercise 4.35: (Measurement commutes with controls) A consequence of the principle of deferred measurement is that measurements commute with quantum gates when the qubit being measured is a control qubit, that is:



(Recall that the double lines represent classical bits in this diagram.) Prove the first equality. The rightmost circuit is simply a convenient notation to depict the use of a measurement result to classically control a quantum gate.

4.5 Universal quantum gates

A small set of gates (e.g. AND, OR, NOT) can be used to compute an arbitrary classical function, as we saw in Section 3.1.2. We say that such a set of gates is *universal* for classical computation. In fact, since the Toffoli gate is universal for classical computation, quantum circuits subsume classical circuits. A similar universality result is true for quantum computation, where a set of gates is said to be *universal for quantum computation* if any unitary operation may be approximated to arbitrary accuracy by a quantum circuit involving only those gates. We now describe three universality constructions for quantum computation. These constructions build upon each other, and culminate in a proof that any unitary operation can be approximated to arbitrary accuracy using Hadamard, phase, CNOT, and $\pi/8$ gates. You may wonder why the phase gate appears in this list, since it can be constructed from two $\pi/8$ gates; it is included because of its natural role in the fault-tolerant constructions described in Chapter 10.

The first construction shows that an arbitrary unitary operator may be expressed *exactly* as a product of unitary operators that each acts non-trivially only on a subspace spanned by two computational basis states. The second construction combines the first construction with the results of the previous section to show that an arbitrary unitary operator may be expressed *exactly* using single qubit and CNOT gates. The third construction combines the second construction with a proof that single qubit operation may be approximated to arbitrary accuracy using the Hadamard, phase, and $\pi/8$ gates. This in turn implies that any unitary operation can be approximated to arbitrary accuracy using Hadamard, phase, CNOT, and $\pi/8$ gates.

Our constructions say little about efficiency – how many (polynomially or exponentially many) gates must be composed in order to create a given unitary transform. In Section 4.5.4 we show that there *exist* unitary transforms which require exponentially many gates to approximate. Of course, the goal of quantum computation is to find interesting families of unitary transformations that *can* be performed efficiently.

Exercise 4.36: Construct a quantum circuit to add two two-bit numbers x and y modulo 4. That is, the circuit should perform the transformation $|x, y\rangle \to |x, x + y \bmod 4\rangle$.

4.5.1 Two-level unitary gates are universal

Consider a unitary matrix U which acts on a d-dimensional Hilbert space. In this section we explain how U may be decomposed into a product of *two-level unitary matrices*; that is, unitary matrices which act non-trivially only on two-or-fewer vector components. The essential idea behind this decomposition may be understood by considering the case when U is 3×3, so suppose that U has the form

$$U = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & j \end{bmatrix}. \tag{4.41}$$

We will find two-level unitary matrices $U_1, \ldots, U_3$ such that

$$U_3 U_2 U_1 U = I\,. \tag{4.42}$$

It follows that

$$U = U_1^\dagger U_2^\dagger U_3^\dagger\,. \tag{4.43}$$

U_1, U_2 and U_3 are all two-level unitary matrices, and it is easy to see that their inverses, $U_1^\dagger, U_2^\dagger$ and $U_3^\dagger$ are also two-level unitary matrices. Thus, if we can demonstrate (4.42), then we will have shown how to break U up into a product of two-level unitary matrices.

Use the following procedure to construct U_1: if $b = 0$ then set

$$U_1 \equiv \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{4.44}$$

If $b \neq 0$ then set

$$U_1 \equiv \begin{bmatrix} \frac{a^*}{\sqrt{|a|^2+|b|^2}} & \frac{b^*}{\sqrt{|a|^2+|b|^2}} & 0 \\ \frac{b}{\sqrt{|a|^2+|b|^2}} & \frac{-a}{\sqrt{|a|^2+|b|^2}} & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{4.45}$$

Note that in either case U_1 is a two-level unitary matrix, and when we multiply the matrices out we get

$$U_1 U = \begin{bmatrix} a' & d' & g' \\ 0 & e' & h' \\ c' & f' & j' \end{bmatrix}. \tag{4.46}$$

The key point to note is that the middle entry in the left hand column is zero. We denote the other entries in the matrix with a generic prime $'$; their actual values do not matter.

Now apply a similar procedure to find a two-level matrix U_2 such that $U_2 U_1 U$ has no entry in the *bottom left* corner. That is, if $c' = 0$ we set

$$U_2 \equiv \begin{bmatrix} a'^* & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{4.47}$$

while if $c' \neq 0$ then we set

$$U_2 \equiv \begin{bmatrix} \frac{a'^*}{\sqrt{|a'|^2+|c'|^2}} & 0 & \frac{c'^*}{\sqrt{|a'|^2+|c'|^2}} \\ 0 & 1 & 0 \\ \frac{c'}{\sqrt{|a'|^2+|c'|^2}} & 0 & \frac{-a'}{\sqrt{|a'|^2+|c'|^2}} \end{bmatrix}. \tag{4.48}$$

In either case, when we carry out the matrix multiplication we find that

$$U_2 U_1 U = \begin{bmatrix} 1 & d'' & g'' \\ 0 & e'' & h'' \\ 0 & f'' & j'' \end{bmatrix}. \tag{4.49}$$

Since U, U_1 and U_2 are unitary, it follows that $U_2 U_1 U$ is unitary, and thus $d'' = g'' = 0$, since the first row of $U_2 U_1 U$ must have norm 1. Finally, set

$$U_3 \equiv \begin{bmatrix} 1 & 0 & 0 \\ 0 & e''^* & f''^* \\ 0 & h''^* & j''^* \end{bmatrix}. \tag{4.50}$$

It is now easy to verify that $U_3 U_2 U_1 U = I$, and thus $U = U_1^\dagger U_2^\dagger U_3^\dagger$, which is a decomposition of U into two-level unitaries.

More generally, suppose U acts on a d-dimensional space. Then, in a similar fashion to the 3×3 case, we can find two-level unitary matrices $U_1, \ldots, U_{d-1}$ such that the matrix

$U_{d-1}U_{d-2}\dots U_1U$ has a one in the top left hand corner, and all zeroes elsewhere in the first row and column. We then repeat this procedure for the $d-1$ by $d-1$ unitary submatrix in the lower right hand corner of $U_{d-1}U_{d-2}\dots U_1U$, and so on, with the end result that an arbitrary $d\times d$ unitary matrix may be written

$$U = V_1 \dots V_k, \tag{4.51}$$

where the matrices V_i are two-level unitary matrices, and $k \le (d-1)+(d-2)+\dots+1 = d(d-1)/2$.

Exercise 4.37: Provide a decomposition of the transform

$$\frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \tag{4.52}$$

into a product of two-level unitaries. This is a special case of the quantum Fourier transform, which we study in more detail in the next chapter.

A corollary of the above result is that an arbitrary unitary matrix on an n qubit system may be written as a product of at most $2^{n-1}(2^n-1)$ two-level unitary matrices. For specific unitary matrices, it may be possible to find much more efficient decompositions, but as you will now show there exist matrices which *cannot* be decomposed as a product of fewer than $d-1$ two-level unitary matrices!

Exercise 4.38: Prove that there exists a $d\times d$ unitary matrix U which cannot be decomposed as a product of fewer than $d-1$ two-level unitary matrices.

4.5.2 Single qubit and CNOT gates are universal

We have just shown that an arbitrary unitary matrix on a d-dimensional Hilbert space may be written as a product of two-level unitary matrices. Now we show that single qubit and CNOT gates together can be used to implement an arbitrary two-level unitary operation on the state space of n qubits. Combining these results we see that single qubit and CNOT gates can be used to implement an arbitrary unitary operation on n qubits, and therefore are universal for quantum computation.

Suppose U is a two-level unitary matrix on an n qubit quantum computer. Suppose in particular that U acts non-trivially on the space spanned by the computational basis states $|s\rangle$ and $|t\rangle$, where $s = s_1\dots s_n$ and $t = t_1\dots t_n$ are the binary expansions for s and t. Let $\tilde{U}$ be the non-trivial 2×2 unitary submatrix of U; $\tilde{U}$ can be thought of as a unitary operator on a single qubit.

Our immediate goal is to construct a circuit implementing U, built from single qubit and CNOT gates. To do this, we need to make use of *Gray codes*. Suppose we have distinct binary numbers, s and t. A *Gray code* connecting s and t is a sequence of binary numbers, starting with s and concluding with t, such that adjacent members of the list differ in exactly one bit. For instance, with $s = 101001$ and $t = 110011$ we have the Gray code

$$\begin{matrix} 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \end{matrix} \tag{4.53}$$

Let g_1 through g_m be the elements of a Gray code connecting s and t, with $g_1 = s$ and $g_m = t$. Note that we can always find a Gray code such that $m \leq n+1$ since s and t can differ in at most n locations.

The basic idea of the quantum circuit implementing U is to perform a sequence of gates effecting the state changes $|g_1\rangle \to |g_2\rangle \to \ldots \to |g_{m-1}\rangle$, then to perform a controlled-$\tilde{U}$ operation, with the target qubit located at the single bit where g_{m-1} and g_m differ, and then to undo the first stage, transforming $|g_{m-1}\rangle \to |g_{m-2}\rangle \to \ldots \to |g_1\rangle$. Each of these steps can be easily implemented using operations developed earlier in this chapter, and the final result is an implementation of U.

A more precise description of the implementation is as follows. The first step is to swap the states $|g_1\rangle$ and $|g_2\rangle$. Suppose g_1 and g_2 differ at the ith digit. Then we accomplish the swap by performing a controlled bit flip on the ith qubit, conditional on the values of the other qubits being identical to those in both g_1 and g_2. Next we use a controlled operation to swap $|g_2\rangle$ and $|g_3\rangle$. We continue in this fashion until we swap $|g_{m-2}\rangle$ with $|g_{m-1}\rangle$. The effect of this sequence of $m-2$ operations is to achieve the operation

$$|g_1\rangle \to |g_{m-1}\rangle \tag{4.54}$$
$$|g_2\rangle \to |g_1\rangle \tag{4.55}$$
$$|g_3\rangle \to |g_2\rangle \tag{4.56}$$
$$\cdots\cdots\cdots$$
$$|g_{m-1}\rangle \to |g_{m-2}\rangle. \tag{4.57}$$

All other computational basis states are left unchanged by this sequence of operations. Next, suppose g_{m-1} and g_m differ in the jth bit. We apply a controlled-$\tilde{U}$ operation with the jth qubit as target, conditional on the other qubits having the same values as appear in both g_m and g_{m-1}. Finally, we complete the U operation by undoing the swap operations: we swap $|g_{m-1}\rangle$ with $|g_{m-2}\rangle$, then $|g_{m-2}\rangle$ with $|g_{m-3}\rangle$ and so on, until we swap $|g_2\rangle$ with $|g_1\rangle$.

A simple example illuminates the procedure further. Suppose we wish to implement the two-level unitary transformation

$$U = \begin{bmatrix} a & 0 & 0 & 0 & 0 & 0 & 0 & c \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ b & 0 & 0 & 0 & 0 & 0 & 0 & d \end{bmatrix}. \tag{4.58}$$

Here, a, b, c and d are any complex numbers such that $\tilde{U} \equiv \begin{bmatrix} a & c \\ b & d \end{bmatrix}$ is a unitary matrix.

Notice that U acts non-trivially only on the states $|000\rangle$ and $|111\rangle$. We write a Gray code connecting 000 and 111:

$$\begin{array}{ccc} A & B & C \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{array} . \tag{4.59}$$

From this we read off the required circuit, shown in Figure 4.16. The first two gates shuffle the states so that $|000\rangle$ gets swapped with $|011\rangle$. Next, the operation $\tilde{U}$ is applied to the first qubit of the states $|011\rangle$ and $|111\rangle$, conditional on the second and third qubits being in the state $|11\rangle$. Finally, we unshuffle the states, ensuring that $|011\rangle$ gets swapped back with the state $|000\rangle$.

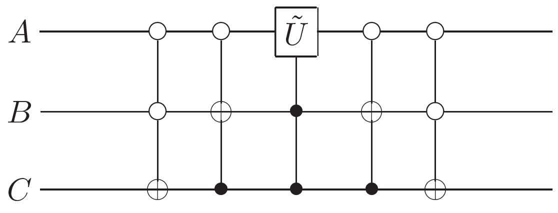


Figure 4.16. Circuit implementing the two-level unitary operation defined by (4.58).

Returning to the general case, we see that implementing the two-level unitary operation U requires at most $2(n-1)$ controlled operations to swap $|g_1\rangle$ with $|g_{m-1}\rangle$ and then back again. Each of these controlled operations can be realized using $O(n)$ single qubit and CNOT gates; the controlled-$\tilde{U}$ operation also requires $O(n)$ gates. Thus, implementing U requires $O(n^2)$ single qubit and CNOT gates. We saw in the previous section that an arbitrary unitary matrix on the 2^n-dimensional state space of n qubits may be written as a product of $O(2^{2n}) = O(4^n)$ two-level unitary operations. Combining these results, we see that an arbitrary unitary operation on n qubits can be implemented using a circuit containing $O(n^2 4^n)$ single qubit and CNOT gates. Obviously, this construction does not provide terribly efficient quantum circuits! However, we show in Section 4.5.4 that the construction is close to optimal in the sense that there are unitary operations that require an exponential number of gates to implement. Thus, to find fast quantum algorithms we will clearly need a different approach than is taken in the universality construction.

Exercise 4.39: Find a quantum circuit using single qubit operations and CNOTs to implement the transformation

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a & 0 & 0 & 0 & 0 & c \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & b & 0 & 0 & 0 & 0 & d \end{bmatrix} , \tag{4.60}$$

where $\tilde{U} = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$ is an arbitrary 2×2 unitary matrix.

4.5.3 A discrete set of universal operations

In the previous section we proved that the CNOT and single qubit unitaries together form a universal set for quantum computation. Unfortunately, no straightforward method is known to implement all these gates in a fashion which is resistant to errors. Fortunately, in this section we'll find a discrete set of gates which can be used to perform universal quantum computation, and in Chaper 10 we'll show how to perform these gates in an error-resistant fashion, using quantum error-correcting codes.

Approximating unitary operators

Obviously, a discrete set of gates can't be used to implement an arbitrary unitary operation *exactly*, since the set of unitary operations is continuous. Rather, it turns out that a discrete set can be used to *approximate* any unitary operation. To understand how this works, we first need to study what it means to approximate a unitary operation. Suppose U and V are two unitary operators on the same state space. U is the target unitary operator that we wish to implement, and V is the unitary operator that is actually implemented in practice. We define the *error* when V is implemented instead of U by

$$E(U, V) \equiv \max_{|\psi\rangle} \|(U - V)|\psi\rangle\|, \tag{4.61}$$

where the maximum is over all normalized quantum states $|\psi\rangle$ in the state space. In Box 4.1 on page 195 we show that this measure of error has the interpretation that if $E(U, V)$ is small, then any measurement performed on the state $V|\psi\rangle$ will give approximately the same measurement statistics as a measurement of $U|\psi\rangle$, for any initial state $|\psi\rangle$. More precisely, we show that if M is a POVM element in an arbitrary POVM, and P_U (or P_V) is the probability of obtaining this outcome if U (or V) were performed with a starting state $|\psi\rangle$, then

$$|P_U - P_V| \le 2E(U, V)\,. \tag{4.62}$$

Thus, if $E(U, V)$ is small, then measurement outcomes occur with similar probabilities, regardless of whether U or V were performed. Also shown in Box 4.1 is that if we perform a sequence of gates $V_1, \ldots, V_m$ intended to approximate some other sequence of gates $U_1, \ldots, U_m$, then the errors add at most linearly,

$$E(U_m U_{m-1} \ldots U_1, V_m V_{m-1} \ldots V_1) \le \sum_{j=1}^{m} E(U_j, V_j)\,. \tag{4.63}$$

The approximation results (4.62) and (4.63) are extremely useful. Suppose we wish to perform a quantum circuit containing m gates, U_1 through U_m. Unfortunately, we are only able to approximate the gate U_j by the gate V_j. In order that the probabilities of different measurement outcomes obtained from the approximate circuit be within a tolerance $\Delta > 0$ of the correct probabilities, it suffices that $E(U_j, V_j) \le \Delta/(2m)$, by the results (4.62) and (4.63).

Universality of Hadamard + phase + CNOT + $\pi/8$ gates

We're now in a good position to study the approximation of arbitrary unitary operations by discrete sets of gates. We're going to consider two different discrete sets of gates, both

Box 4.1: Approximating quantum circuits

Suppose a quantum system starts in the state $|\psi\rangle$, and we perform either the unitary operation U, or the unitary operation V. Following this, we perform a measurement. Let M be a POVM element associated with the measurement, and let P_U (or P_V) be the probability of obtaining the corresponding measurement outcome if the operation U (or V) was performed. Then

$$|P_U - P_V| = \left|\langle\psi|U^\dagger MU|\psi\rangle - \langle\psi|V^\dagger MV|\psi\rangle\right|. \tag{4.64}$$

Let $|\Delta\rangle \equiv (U - V)|\psi\rangle$. Simple algebra and the Cauchy–Schwarz inequality show that

$$\begin{aligned} |P_U - P_V| &= \left|\langle\psi|U^\dagger M|\Delta\rangle + \langle\Delta|MV|\psi\rangle\right|. & (4.65)\\ &\leq |\langle\psi|U^\dagger M|\Delta\rangle| + |\langle\Delta|MV|\psi\rangle| & (4.66)\\ &\leq \||\Delta\rangle\| + \||\Delta\rangle\| & (4.67)\\ &\leq 2E(U,V). & (4.68)\end{aligned}$$

The inequality $|P_U - P_V| \leq 2E(U,V)$ gives quantitative expression to the idea that when the error $E(U,V)$ is small, the difference in probabilities between measurement outcomes is also small.

Suppose we perform a sequence $V_1, V_2, \ldots, V_m$ of gates intended to approximate some other sequence of gates, $U_1, U_2, \ldots, U_m$. Then it turns out that the error caused by the entire sequence of imperfect gates is at most the sum of the errors in the individual gates,

$$E(U_m U_{m-1} \ldots U_1, V_m V_{m-1} \ldots V_1) \leq \sum_{j=1}^{m} E(U_j, V_j). \tag{4.69}$$

To prove this we start with the case $m = 2$. Note that for some state $|\psi\rangle$ we have

$$\begin{aligned} E(U_2U_1, V_2V_1) &= \|(U_2U_1 - V_2V_1)|\psi\rangle\| & (4.70)\\ &= \|(U_2U_1 - V_2U_1)|\psi\rangle + (V_2U_1 - V_2V_1)|\psi\rangle\|. & (4.71)\end{aligned}$$

Using the triangle inequality $\||a\rangle + |b\rangle\| \leq \||a\rangle\| + \||b\rangle\|$, we obtain

$$\begin{aligned} E(U_2U_1, V_2V_1) &\leq \|(U_2 - V_2)U_1|\psi\rangle\| + \|V_2(U_1 - V_1)|\psi\rangle\| & (4.72)\\ &\leq E(U_2, V_2) + E(U_1, V_1), & (4.73)\end{aligned}$$

which was the desired result. The result for general m follows by induction.

of which are universal. The first set, the *standard set* of universal gates, consists of the Hadamard, phase, controlled-NOT and $\pi/8$ gates. We provide fault-tolerant constructions for these gates in Chapter 10; they also provide an exceptionally simple universality construction. The second set of gates we consider consists of the Hadamard gate, phase gate, the controlled-NOT gate, and the Toffoli gate. These gates can also all be done fault-tolerantly; however, the universality proof and fault-tolerance construction for these gates is a little less appealing.

We begin the universality proof by showing that the Hadamard and $\pi/8$ gates can be

Bibliography

[1] Phillip Kaye, Raymond Laflamme, Michele Mosca, et al. *An introduction to quantum computing*. Oxford University Press on Demand, 2007.

[2] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.