A QUICK INTRODUCTION TO QUANTUM PROGRAMMING

Jules Jacobs

September 25, 2021

This note is a quick introduction to quantum programming in the circuit model. A quantum computer on k bits gets as input a *quantum circuit description*, and produces as output a random string of k bits according to a probability distribution determined by the quantum circuit. A quantum programming language in this model is a language for creating such quantum circuits.

After reading this note, you will be able to write a computer program that simulates such a quantum computer (albeit exponentially more slowly than an actual quantum computer would execute a circuit, which is the point!).

1 QUANTUM STATES

Imagine that we have a box with some physical system inside of it, with a finite set S of possible states. A probability distribution over S is a vector $\vec{p}$ of probabilities, one probability $p_x \in [0, 1]$ for each state $x \in S$, such that $\sum_x p_x = 1$.

A *quantum state* over S, on the other hand, is a vector $\vec{\phi}$ of *probability amplitudes*, one complex number $\phi_x \in \mathbb{C}$ for each state $x \in S$. If we *measure* such a quantum state, we obtain outcome $x \in S$ with probability $p_x = |\phi_x|^2$. Thus, in order for ϕ to be a proper quantum state, we must have $\sum_x |\phi_x|^2 = 1$.

2 TIME EVOLUTION IN QUANTUM MECHANICS

Imagine that the system in the box evolves in time according to some laws of physics. In quantum mechanics, the state evolution is given by a matrix U that multiplies the state every time step. If the state is currently ϕ, then at the next time step the state is $U\phi$. If there are $n = |S|$ possible states, then U is an $n \times n$ matrix. Only matrices that preserve the condition that the probabilities sum to 1 are allowed: if $\sum_x |\phi_x|^2 = 1$ we must have $\sum_x |(U\phi)_x|^2 = 1$. Such matrices are called *unitary*.

It might be helpful to compare with probabilistic evolution of the state as in a Markov chain. In that case we model the state with a probability vector $\vec{p}$ and we multiply this vector with a matrix M at each time step. If the state is currently p, then at the next time step the state is Mp. Matrices that preserve the condition that all probabilities are non-negative and that their sum remains 1 are called *stochastic matrices*. The entry $M_{x,y}$ of the matrix is the probability that the system will step to state y, if the state is currently x. In the quantum case, the entry $U_{x,y}$ of the unitary matrix, is the *probability amplitude* of next state being y, if the state is currently x.

3 WHAT A QUANTUM COMPUTER IS

A quantum computer with state set S is a device where we can *input* such a matrix U and an initial state $x \in S$. It will then do one step of time evolution using U, and it will *measure* the new state and tell us which outcome $y \in S$ it got. Thus, a quantum computer is a kind of universal quantum mechanics simulator:

1. We *input* the initial state $x \in S$ and matrix U
2. The quantum computer *outputs* answer $y \in S$ with probability $|U_{x,y}|^2$

One should probably not think of quantum computers as a replacement for classical computers, but rather as coprocessors (like GPUs) that allow you to perform *some* subroutines asymptotically faster. For instance, Shor's algorithm for prime factorization is a classical algorithm that makes use of a *period finding subroutine* that is run on a quantum computer.

We will now look at how the matrix U is represented as a quantum circuit.

4 QUANTUM CIRCUITS

In physics, the state set S is often infinite, and sometimes even uncountably infinite (e.g. the position of a particle), but in quantum programming the set $S = \{0,1\}^k$ is taken to be strings of k bits, so that $|S| = 2^k$. Still, this means that U is a 2^k-by-2^k matrix. One might wonder how we even input the U to the quantum computer, if it contains an exponential amount of data.

The answer is that we can't quite input *any* matrix U; it must be encoded as a *quantum circuit*. A quantum circuit is a sequence of unitary operations we do on the state of n bits, where each operation operates on some small subset of the bits and leaves the rest of the bits alone.

Often, a small set of primitive operations is used, such as the *Hadamard gate* and the *CCNOT gate*. The Hadamard gate operates on one bit, and the CCNOT gate operates on three bits.

In order to describe what they do, we introduce a bit of notation for *basis states*. We use the notation $\phi = |01001\rangle$ for the basis state ϕ where $\phi_{01001} = 1$ and $\phi_x = 0$ otherwise, *i.e.*, the state that puts all probability amplitude on 01001. Because gates A are linear operators, it is sufficient to define their behavior on basis states $|x\rangle$ for $x \in \{0,1\}^k$, since for a superposition $\phi = \sum_x \phi_x |x\rangle$ we have $A\phi = \sum_x \phi_x A|x\rangle$.

4.1 THE HADAMARD GATE

The Hadamard gate H operates on one bit, and is defined as:

$$\mathrm{H}|0\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big)$$
$$\mathrm{H}|1\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle - |1\rangle\big)$$

Equivalently, we can define it using matrix notation, as

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

If we have n bits in the state, then we have Hadamard gates $\mathrm{H}_1, \mathrm{H}_2, \cdots, \mathrm{H}_n$, each operating on a different bit. This is what H_1 does:

$$\mathrm{H}_1|0b_1b_2\cdots b_n\rangle = \frac{1}{\sqrt{2}}\big(|0b_1b_2\cdots b_n\rangle + |1b_1b_2\cdots b_n\rangle\big) \tag{1}$$
$$\mathrm{H}_1|1b_1b_2\cdots b_n\rangle = \frac{1}{\sqrt{2}}\big(|0b_1b_2\cdots b_n\rangle - |1b_1b_2\cdots b_n\rangle\big) \tag{2}$$

Try writing down H_1 as a 2^n-by-2^n matrix, and you'll see why this notation is useful.

4.2 CLASSICAL GATES

Given a function $f : \{0,1\}^k \to \{0,1\}^k$ on bit strings of length k, we define the classical gate C_f:

$$C_f |x\rangle = |f(x)\rangle \tag{3}$$

That is, given a basis state $|x\rangle$, it outputs another basis state $|f(x)\rangle$. The function f must be bijective in order for this to be a unitary operator.

A common trick is to start with a boolean function $g : \{0,1\}^{k-1} \to \{0,1\}$ and define

$$f(b_1, \cdots, b_{k-1}, b_k) = (b_1, \cdots, b_{k-1}, b_k \oplus g(b_1, \cdots, b_{k-1})) \tag{4}$$

This function f leaves the first $k-1$ bits alone, and XORs the last bit with $g(b_1, \cdots, b_{k-1})$. You may verify that f is always bijective.

The CCNOT gate is an example of this: we take $g(b_1, b_2)$ to be the local AND of b_1 and b_2. This gives us a gate that operates on three bits, which leaves the first two bits alone and flips the third bit if the first two bits are both 1. The CCNOT gate is a universal classical gate: any bijection f can be built out of CCNOT gates (analogous to how any boolean function can be built out of NAND gates). As with Hadamard gates, we can apply a CCNOT gate to any three bits in the state, and thus we have CCNOT_{ijk} gates, which applies the gate to bits i, j, k. This is what CCNOT_{123} does:

$$\text{CCNOT}_{123} |b_1 b_2 b_3 b_4 \cdots b_k\rangle = |b_1 b_2 (b_3 \oplus (b_1 \wedge b_2)) b_4 \cdots b_k\rangle$$

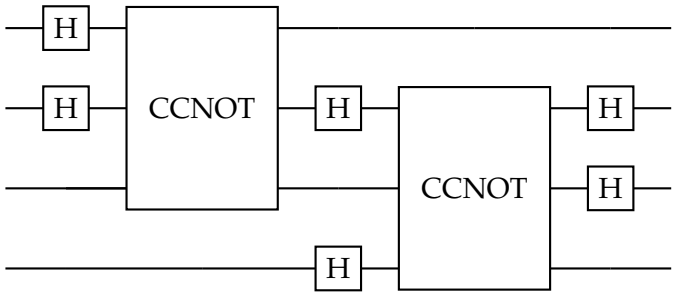
That is, it flips the third bit if the first two bits are 1, and leaves the other bits alone.

4.3 COMPOSITION OF GATES

We input the matrix U into the quantum computer as a sequence of operations, e.g.:

$$U = H_2 \cdot H_3 \cdot \text{CCNOT}_{234} \cdot H_2 \cdot H_4 \cdot \text{CCNOT}_{123} \cdot H_1 \cdot H_2$$

We can graphically represent this circuit as follows:



The Hadamard gate and CCNOT gate together are universal, in the sense that any quantum computation can be done using only these two gates.[1]

1 This more or less means that any unitary matrix can be approximated as product of Hadamard gates and CCNOT gates. Since these gates all have real valued matrices, this is not exactly true, see [Aha03] for the technical sense in which these two gates are universal.

5 THE DEUTSCH–JOZSA ALGORITHM

We are given a boolean function $g : \{0,1\}^n \to \{0,1\}$ with n inputs and 1 output. We are promised that g is either *constant* or *balanced* (meaning that it is 0 on half of its inputs and 1 on the other half). Our task is to determine whether it is constant or balanced. The function g is assumed to be efficiently implementable using logic gates.

Classically, it seems that we cannot do better than testing g on $2^n/2 + 1$ inputs in the worst case: if they are all the same, then g is constant, and if they are not all the same, then by assumption g must be balanced.

The Deutsch–Jozsa algorithm [Deu21] is a quantum algorithm for this problem that operates on $n + 1$ bits and is given by:

$$U = \mathrm{H}^{\otimes n+1} \cdot \mathrm{C}_f \cdot \mathrm{H}^{\otimes n+1}$$

where

- $\mathrm{H}^{\otimes n+1} = \mathrm{H}_1 \cdot \mathrm{H}_2 \cdots \mathrm{H}_n \cdot \mathrm{H}_{n+1}$ applies a Hadamard gate to every bit.
- C_f is the classical gate defined in (3) and (4), that XORs the $(n+1)$-th bit with $g(b_1, \cdots, b_n)$.

Running this algorithm on input $00\cdots01$ always produces the output $00\cdots01$ if g is constant, and is guaranteed to give a different output if g is balanced (see below). Thus, if we can encode g efficiently using gates, then a quantum computer can efficiently determine whether it is constant or balanced.

5.1 CORRECTNESS OF THE ALGORITHM

Let us first see what the operator $\mathrm{H}^{\otimes n}$ does on a general basis state $|x\rangle$, where x is a bit string of length n. Since H creates a superposition of the two bit positions for each bit, we will get a sum over all possible bit strings,

$$\mathrm{H}^{\otimes n} |x\rangle = \frac{1}{\sqrt{2}^n} \sum_{y \in \{0,1\}^n} (-1)^{\sigma_{x,y}} |y\rangle$$

where $(-1)^{\sigma_{x,y}}$ is the sign of $|y\rangle$ in the sum. What is this sign? By the definition of the Hadamard gate (1), we get a minus sign each each time the bits in *both* x *and* y are 1. Thus, $\sigma_{x,y}$ = the number of 1 bits in the bitwise AND $x\&y$. For example, if $x = (00\cdots0)$ then $\sigma_{x,y} = 0$ and therefore $\mathrm{H}^{\otimes n} |00\cdots0\rangle = \frac{1}{\sqrt{2}^n} \sum_y |y\rangle$.

We calculate what our U does on $|00\cdots01\rangle$:

$$\begin{aligned}
U|00\cdots01\rangle &= H^{\otimes n+1}\cdot C_f\cdot H^{\otimes n+1}|00\cdots01\rangle && \text{(perform H on the first } n \text{ bits } \mapsto)\\
&= H^{\otimes n+1}\cdot C_f\cdot H_{n+1}\frac{1}{\sqrt{2}^n}\sum_x|x1\rangle && \text{(perform H on the last bit } \mapsto)\\
&= H^{\otimes n+1}\cdot C_f\frac{1}{\sqrt{2}^{n+1}}\sum_x\big(|x0\rangle-|x1\rangle\big) && \text{(perform } C_f \mapsto)\\
&= H^{\otimes n+1}\frac{1}{\sqrt{2}^{n+1}}\sum_x\big(|x(0\oplus g(x))\rangle-|x(1\oplus g(x))\rangle\big) && \text{(rewrite } \mapsto)\\
&= H^{\otimes n+1}\frac{1}{\sqrt{2}^{n+1}}\sum_x(-1)^{g(x)}\big(|x0\rangle-|x1\rangle\big) && \text{(perform H on the last bit } \mapsto)\\
&= H^{\otimes n}\frac{1}{\sqrt{2}^{n}}\sum_x(-1)^{g(x)}|x1\rangle && \text{(perform H on the first } n \text{ bits } \mapsto)\\
&= \frac{1}{2^n}\sum_x\sum_y(-1)^{g(x)+\sigma_{x,y}}|y1\rangle && \text{(rewrite } \mapsto)\\
&= \sum_y\left(\sum_x\frac{(-1)^{g(x)+\sigma_{x,y}}}{2^n}\right)|y1\rangle
\end{aligned}$$

To determine the probability of measuring output $|00\cdots01\rangle$, we square its coefficient in this sum:

$$p_{00\cdots01}=\left|\sum_x\frac{(-1)^{g(x)}}{2^n}\right|^2$$

(since $\sigma_{x,y}=0$, if $x=(00\cdots0)$)

Notice that the terms sum to 1 if g is constant. If g is balanced then precisely half the terms are $+1$ and half the terms are -1, so they sum to 0. Therefore, if g is constant, then we are certain to get the answer $00\cdots01$, and if g is balanced then we are certain *not* to get this answer.

6 A QUANTUM SIMULATOR

Here is a Python program that allows us to apply Hadamard gates and classical gates to a state vector:

```
from math import sqrt

n = 5 # number of bits in our state
      # we represent bit strings as integers

def print_state(s):
  print(" + ".join([f'{s[i]:.5f}|{bin(i)[2:].zfill(n)}>'
                for i in range(len(s)) if s[i] != 0]))

# gives the basis state |x>, where x is a string of 0's and 1's
def basis(x):
  s = [0]*2**n
  s[int(x,base=2)] = 1
  return s
```

```
# apply the classical gate C_f, where f is a bijective function on bit strings
def classical(s,f):
  s2 = [0]*2**n
  for x in range(2**n):
    s2[f(x)] = s[x]
  return s2

# apply the Hadamard gate H_k, where k is the bit to apply the gate to
def hadamard(s,k):
  bitk = 1 << k
  s2 = [0]*2**n
  for x in range(2**n):
    sign = (-1)**((x >> k) & 1)
    s2[x] = (s[x & ~bitk] + sign*s[x | bitk])/sqrt(2)
  return s2

# example
s = basis("10101")
print_state(s) # 1.00000|10101>
s = hadamard(s,1)
print_state(s) # 0.70711|10101> + 0.70711|10111>
s = classical(s, lambda x: ~x)
print_state(s) # 0.70711|01000> + 0.70711|01010>
s = hadamard(s,3)
print_state(s) # 0.50000|00000> + 0.50000|00010> + -0.50000|01000> + -0.50000|01010>
s = hadamard(s,1)
print_state(s) # 0.70711|00000> + -0.70711|01000>
```

Download here: https://julesjacobs.com/notes/quantum/qsim.py

7 Quantum programming languages

A quantum programming language is a high level language for assembling quantum circuits. These languages often include the ability to simulate a quantum circuit (which takes exponential time in the worst case), and some allow you to take classical computation written using ordinary code and turn it into a quantum circuit (e.g. using a large number of CCNOT gates). See https://en.wikipedia.org/wiki/Quantum_programming for a list of quantum programming languages and their distinguishing features.

Acknowledgements. I thank Arjen Rouvoet for his suggestions that improved the clarity and correctness of this note (all remaining errors were likely introduced afterwards), and Dong-Ho Lee for introducing me to quantum computation and answering my questions.

References

[Aha03] Dorit Aharonov. A Simple Proof that Toffoli and Hadamard are Quantum Universal. *arXiv:quant-ph/0301040*, January 2003. URL: http://arxiv.org/abs/quant-ph/0301040, arXiv:quant-ph/0301040.

[Deu21] Deutsch–Jozsa algorithm. *Wikipedia*, August 2021. URL: https://en.wikipedia.org/w/index.php?title=Deutsch%E2%80%93Jozsa_algorithm&oldid=1040726429.