A Primer on Quantum Computing and Algorithms

Geoffrey Bradway

15 min read · Jul 18, 2016

Intro

The best way your average computer scientist could describe a quantum algorithm is “a riddle, wrapped in a mystery, inside an enigma”. In this article I hope to answer the fundamental question every computer scientist asks themselves from time to time “wtf is going on?”, but on the more limited scope of quantum algorithms.

Useful resources I have found along the way are

- An awesome introductory blog series on quantum computing
- Quantum Algorithms via Linear Algebra [A ~200 page book which builds the theory up from scratch. Overall I thought it was pretty good, and readable, and most of this article is drawn from that book.]
- Quantum Algorithms: an overview
- Quantum Computation and Quantum Information [Described as the bible for quantum computing, but much like the real bible, it is fairly long and so I haven’t read all of it yet.]

Background

What should you know before reading this?

It turns out that Quantum Computing looks a lot like machine learning (read: linear algebra). That being said, to get the most out of this this post it would be nice if you knew some linear algebra— but I want to make this accessible, so I will try to limit the technical background needed and explain everything as I go.

What level of abstraction are we talking about?

Typically, when someone says algorithm, most people think of something like

```
def cool_sort(my_list):

if is_sorted(my_list): return my_list

return cool_sort(random_shuffle(my_list))
```

And this would do something like

```
cool_sort([3,2,1]) = [1, 2, 3]
```

But for this post we are considering algorithms on a much lower level — the circuitry itself. The pieces at our disposal are more analogous to adders, ALUs, etc. For example, a half adder is a boolean circuit that just adds two bits together. The function looks something like

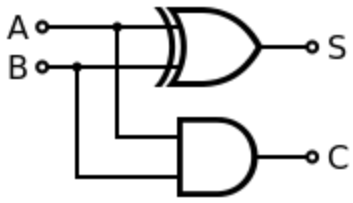
```
Half_Add(0, 0) = 00

Half_Add(0, 1) = 01

Half_Add(1, 0) = 01

Half_Add(1, 1) = 10
```

And this is done via circuits, and looks something like



Never mind exactly what the circuit physical does, that part is abstracted away from us, just keep in mind that we are doing the equivalent of a very low level algorithm, not the high level sorting we were doing before.

Later on when we talk about these U matrices, just keep in mind these are physical circuits operating on physical qubits. These are algorithms too, and I will be using "quantum algorithms" and "quantum circuits" interchangeably.

The question of designing a good quantum chip, programming language, etc are still very open. If you are curious about designing a full stack quantum computer, definitely check out the blog series I linked to above.

Creating a Quantum Circuit

Mechanics

In general, a function takes in some sort of input, does some useful calculations, and spits out an output. Quantum algorithms are no exception to this. In the quantum case — the input is some quantum bits, our program is some matrices U1, U2, U3,..., and the output is the same quantum bits we had before, but hopefully we have done some useful work, and the new state of those quantum bits is something meaningful. *

The basic mechanics of quantum circuit are actually quite simple — it is just linear algebra and a tiny bit of probability.

A basic circuit follows a simple outline, but it's okay to not understand exactly what this is doing right now. In fact I am trying to divorce it from what is going on physically to show that the math is easy. Once you are convinced it is easy, we will build up some intuition for what it is doing.

The outline of a quantum circuit looks like this

- Start out with a basic input vector, let's call it X.
- Multiply it by a bunch of square matrices U1, U2, U3, ..., Un. So we get something that looks like
  Un Un-1...U3 U2 U1 X = Y
- Sampling an output proportionally to |Y|**2, where **2 is the element wise square. So if Y^T was [1/2, 1/2, 1/2, 1/2], then the element-wise square is [1/4, 1/4, 1/4, 1/4], and you have an equal probability of each outcome. This is also true if Y was [1/2, 1/2, -1/2, -1/2] or even [1/2, 1/2, 1/2 i, 1/2 i]. And this sampled outcome of Y is the result of your algorithm!
- Voila! Pat yourself on the back because you just performed a quantum algorithm!

There are a few rules we need to stick by to make this a valid algorithm

- ||X||^2 = 1
- U is unitary [meaning (U^T)U = I or U*U = I if U has complex entries].

See? Super easy. Now we will build up some math intuition, and then I will tell you how this relates to everything, and then we will retouch on the outline with our brand new awesome understanding.

So let's first do a little review of some linear algebra, and we will see how that ties into these unitary matrices.

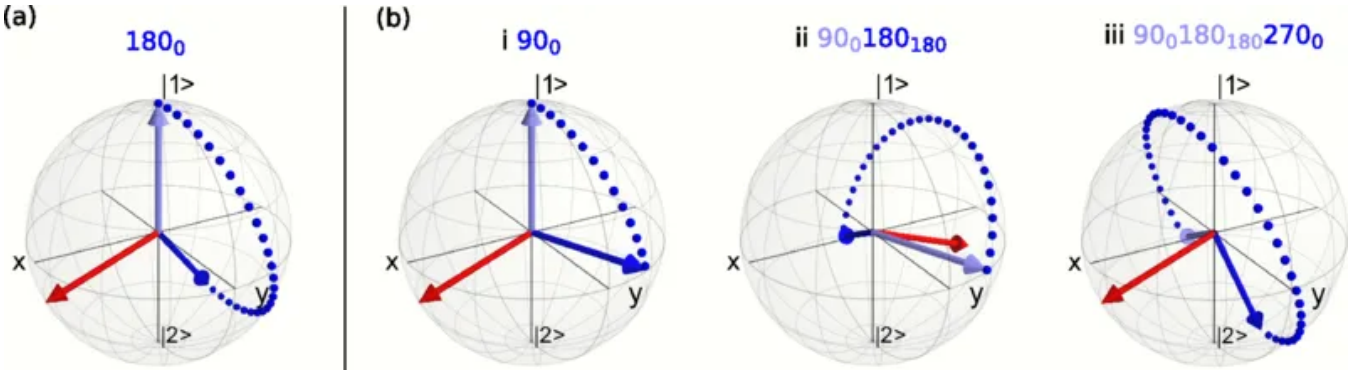
If we remember back, the determinant of a matrix measures how a matrix changes the space it operates on. So if we have a matrix A with det(A) = 2, then one unit of space before will double in size once you apply A. Or if det(A) = 0, then one unit of space will get completely destroyed when you apply A, and that is why A isn't invertible iff det(A) = 0, because once the space gets destroyed and you can't un-destroy it. This effect on the change of the size of the space is why the determinant

makes an appearance in the multivariable change of variables equation, because you need to re-scale based on how the system of equations resizes the space.

Well, one property that unitary matrices have is that |det(U)| = 1. This means that our unitary matrix will preserve things nicely, and it won't stretch or shrink anything — it will just move stuff around. In fact, if we are talking about R^2, then the only possible unitary matrices we have are the identity, rotations about the origin, and twists about other vectors, or some combination of rotations and twists. But while the vector points somewhere else now, it is still the same length.

So geometrically, we can interpret our algorithm as this — we are starting with a vector who lies on a unit circle, and our transformation has to keep it on the unit circle. So by the end of applying all these matrices we have ||Y||^2 = 1, and when we sample via the square-normed elements, we are sampling from a legitimate probability distribution. Hooray!

Visually, this looks something like the blue arrows in this picture, the red arrows aren't important for what we need to visualize.



If you want to sound *very very fancy* this circle is called the Bloch Sphere. And if you study really really hard, maybe one day you can have a trivial mathematical object named after yourself!

As we can see, we are applying various Us to the blue vector, but it is staying on the circle, it is just pointing in different directions!

Now we are ready to start talking about what is actually going on. These vectors are basically just fancy bookkeeping for the physical state the system could be in.

Let's start simple— if we have a qubit, then we can measure it to be a 0 or a 1. But the

qubit doesn't have to be deterministically 0 or 1. We can have a qubit so that there is some chance it will be 0 or 1. Your first guess might be to represent these directly as probabilities, which would look like a vector [a, b], where a+b =1, and a,b>=0. But as you may have figured out from what I was talking about above, we can also represent it [x, y], where |x|^2+|y|^2 = 1, and our qubit is 0 with probability |x|^2 and 1 with probability |y|^2. And it actually turns out, this is better.

One thing to note is that for 1 bit we needed two state variables, one for 0 and one for 1, and in general for n bits, we need vectors of size 2^n, because we need one row for each possible state.

If you look at the 2 bit case, then we care about vectors that represent [00, 01, 10, 11] or in the 3 bit case [000, 001, 010, 011, 100, 101, 110, 111].

[Technical aside, feel free to ignore this part: if these were probabilities, we would need only the specify (2^n)-1 values. This is because the probabilities have to add up to one and so you can always figure out what the last one is if you have all the other values. But in general in the quantum case we use complex numbers, and so you are looking at |x1|^2+|x2|^2+.... = 1. This means we that while the magnitude of the last value can always be figured out, for the same reason we could always figure it out for the probabilities, the value still needs to be specified because it can be any vector with that magnitude].

So the basis of the vector space we are considering looks like this (in the 2 bit case), [E00, E01, E10, E11]. A vector of [1, 0, 0,0] means that we for-sure have a qubit of 00, and a vector of [1/2, 1/2, 1/2, 1/2] means that if we measured our qubit, then 25% of the time it would be 00, 01, 10, or 11.

So let's retouch on the outline of a quantum algorithm

- Start out with a basic input vector, let's call it X, where ||X||^2 = 1

X is the vector representation of the state of our input qubits, which is why ||X||^2 = 1 . In almost all of our quantum algorithms we want to start with qubits which we know the state. Since there is no uncertainty about what the state is the vector representation would be something like [1, 0, 0, 0] or [0, 1, 0, 0].

- Apply our unitary matrices U1, U2,... Un to get the result
  Un-1...U3 U2 U1 X = Y

Here we are running our "program". Each one of these matrices had to be unitary, so it changes the state of our qubits, but they are still valid qubits.

- Figure out the new state of our qubits by sampling an output proportionally to |Y|**2

Y is the new state representation of our qubits, and when we measure our qubits we are forcing them to take a specific value like 00 or 11. This is the same thing as picking one of our basis vectors E00, E01, etc. Then |Y|**2 just tells us how likely we are to see each possible configuration of qubits, and so this sampling procedure is equivalent to measuring the system.

Deutsch-Jozsa Algorithm

From wikipedia, the Deutsch-Jozsa Algorithm is the bee's knees because:

> *The Deutsch–Jozsa problem is specifically designed to be easy for a quantum algorithm and hard for any deterministic classical algorithm. The motivation is to show a black box problem that can be solved efficiently by a quantum computer with no error, whereas a deterministic classical computer would need exponentially many queries to the black box to solve the problem.*

The basic idea is that we want an algorithm that can figure out if some boolean function, F, is a constant function. This function F takes as input n-bits and returns as output a single bit or alternatively true or false. *

Mathematically, this looks like

F:{0,1}^n ->{0, 1}

So far so good. But because this is quantum computing, we can do something much fancier, and we can tell if F is constant only calling F once. Intuitively, this makes absolutely zero sense whatsoever , like seriously zero sense, because it seems like you would have to evaluate it on at least two inputs to at least tell if it wasn't *

constant. But the trick (which we will talk about more in the “the magic of quantum” section) is that we can do one function call but over a distribution of states.

For this blog post we are going to talk about the simple case, where F is a one-bit function, so F:{0,1} ->{0,1}.

First things first, there is a little bit of busy work to get out of the way before we talk about the algorithm, and it will be nice to have it out of the way so we don’t have to do it in the middle, so let’s do that. Now there are four possible one-bit F’s, F(x) = x, F(x) = not x, F(x) = 0, and F(x) =1. But we care about a different object related to F. For this section, xy means x concat y, and they are both bits. The function we care about is G where

G_F(xy) = x concat (F(x) XOR y)

So if F is the identity,
G_I(00) = 0 concat (F(0) XOR 0) = 0 concat (0 XOR o) = 00

Let’s make some G truth table matrices for each possible F. Remember, our basis [E00, E01, E10, E11]^T. So if we look at G*[1, 0, 0, 0]^T, that is the same as G(00), and G*[0, 0, 1, 0]^T will tell us what happened if did G(10). Since one specific input to G will have one specific output, we expect that each row will just have a single 1, indicating what the output bits will be.

Let’s work through a tangible example. Let us consider the case where F(x) = not x

Then G looks like

G(00) = 0 concat (F(0) XOR o) = 0 concat (1 XOR 0) = 01

G(01) = 0 concat (F(0) XOR 1) = 0 concat (1 XOR 1) = 00

G(10) = 1 concat (F(1) XOR o) = 1 concat (0 XOR 0) = 10

G(11) = 1 concat (F(1) XOR 1) = 1 concat (0 XOR 1) = 11

So the G_not matrix looks like

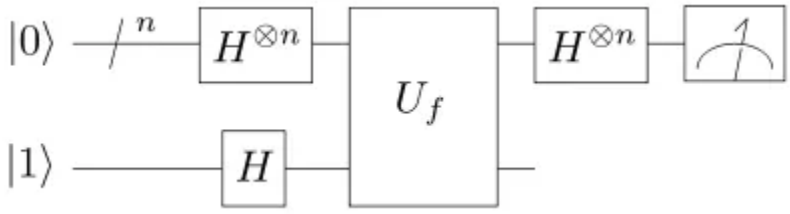
$$G_{not} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And similarly, the other values of G look like (G_T is G true, G_F is G false, and G_I is G identity)

$$G_T = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, G_F = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, G_I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Now, constructing G is where it seems like we are cheating. I said we could do this algorithm with one function call, but clearly we had to evaluate all possible function values to make G, and we are going to use G in our algorithm — so did I do more than one call in the algorithm?

The answer is no. G is only for bookkeeping. If we look at the circuit for the Deutsch-Jozsa Algorithm itself it looks like



Ignore the Hs and weird looking boxes

Where the part that they call Uf we are calling G, but you can see that it's just circuit * that operates on the two bit inputs. We needed to fill in all of the values for our analysis, just like you would have had to do with a normal truth table, but when the circuit runs, the two qubits go through that part only once.

With all the funness out of the way, we have that out of the way, we can present our algorithm

1. Take X = [0, 1, 0, 0]^T (So we are starting with qubits 01)
2. Apply H4
3. Apply G
4. Apply R
5. Measure the output and look at the first bit

Where H4 and R are [Note: I forgot a normalization constant on H4 to make it unitary, but that just means that the norm of all these vectors will be c instead of 1, and we were really missing a 1/c]

$$H_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}, R = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

Let's do a step by step walkthrough of this

If we look at H4X we get the following results

$$H_4X = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

The resulting vector of [1, -1, 1, -1]^T means that if we were to measure now, we

have a 25% chance of seeing a qubit in any state — 00, 01, 10, 11. So we have taken our starting vector an we have almost “uniformly” distributed it across all possible states —but even though we would measure uniformly, there is more structure in the space since some entries have different signs!

Now if we look at GH4X for all possible G’s we get

$$G_I H_4 X = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}, G_{not} H_4 X = \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix},$$

$$G_T H_4 X = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix}, G_F H_4 X = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix},$$

Which looks like a jumbled mess — but it will make sense in after we apply the next matrix.

RGH4X then looks like the following

$$RG_I H_4 X = \begin{bmatrix} 0 \\ 0 \\ 2 \\ -2 \end{bmatrix}, RG_{not} H_4 X = \begin{bmatrix} 0 \\ 0 \\ -2 \\ 2 \end{bmatrix},$$

$$RG_T H_4 X = \begin{bmatrix} -2 \\ 2 \\ 0 \\ 0 \end{bmatrix}, RG_F H_4 X = \begin{bmatrix} 2 \\ -2 \\ 0 \\ 0 \end{bmatrix}$$

Now something really cool just happened — If you notice True and False, which are the only ways that F is constant, map only to E00 and E01, and Not and Identity map to E10 and E11. So if F was the True or False function, we don't know if the output to our program will be 00 or 01, since it will happen with a 50/50 chance, but we know that the first bit will always be 0. Similarly for I and Not, there is a 50/50 chance it will be 10 or 11, but again the first bit is always 1. This means that if we look at the left-most bit of the output, we can tell if F was constant or not!

The Magic of Quantum

Now that we have all of the setup out of the way there are two reasons that quantum algorithms are cool

1. The representation we picked (well mother nature picked for us), is a lot stronger than simple probability estimates, and can express more complicated things
2. Qubits store "distributions" and not instances

The reason that I did the algorithm before this section is because we can see both reasons in the algorithm itself.

First let's look at the representation but in the context of the Deutsch-Jozsa

Algorithm. If we were to measure after GHX, it wouldn't be very exciting. Each vector is different, but each component of each vector has the same magnitude. Any F function would make this act the same way at that point in the algorithm — an equal chance of measuring any state. So from a probabilistic perspective, this isn't very interesting, they are all just uniform distributions. So we couldn't do anything at this point, because each distribution is the same, and so there is nothing we could do to an observation to figure out which distribution it came from.

Here is a good example of the added expressiveness of this representation, because we aren't representing it directly as probabilities, instead complex numbers, there is a lot more stuff we can do with it. In this particular case, we were able to use the R matrix to actually distinguish which class of function it came from, because there was more structure.

And if you think about it, there is one way to represent a uniform distribution over N numbers, namely a probability vector [1/N, 1/N,..., 1/N], but the space of x where ||x||^2 = 1, there are infinitely many points that could represent that distribution all of the form [e^(ia)/sqrt(N), e^(ib)/sqrt(N), e^(ic)/sqrt(N), ...] and it even includes what we had before with the point [1/sqrt(N), 1/sqrt(N), ...]. So that is cool because you have distributions that from a probabilistic standpoint are the same, but mean different things, and interact differently.

The second really cool part is that qubits store "distributions" and not instances. To examine this, let us consider the step where we applied G to H4X

Let's remember what H4X looks like

$$H_4X = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

And we can see that part of our qubits are in each possible configuration. Then

when we multiply G by H4X, its like we get to do a G call on each possible state. **Four for the price of one!** (Or in the general case, 2^n for the price of 1!)

To better see this, let's look at what would be the classical analog. Remember before when I mentioned that if you have a deterministic boolean function, then using our truth-table matrix you can do one look up it at once — this is is the same as doing G * E, where E^T is one of the following [1, 0, 0, 0], [0, 1, 0, 0],[0, 0, 1, 0], or [0, 0, 0, 1]. So the equivalent classical operation is

$$\begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Decomposing H4X

$$G \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} = G \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} - G \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + G \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} - G \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Which is more inline with what we would have expected — us needing 4 evaluations of F to tell if all outputs are equal.

But, because qubits don't take on a value until we measure them, we can get away with some crazy stuff.

So in conclusion, by using a better representation and by being able to carry more state, you can get away with some pretty cool stuff!

Lastly, I want to thank my friends who I conscripted to give me feedback — Matt, Eric, Yunus, and Cathy.