

An Introduction to Quantum Computing and its Applications

Andrew Rippy

Department of Physics, Wabash College, Crawfordsville, IN 47933

(Dated: June 24, 2024)

Abstract

Quantum Computing is emerging as a powerful tool to solve certain problems. This paper will endeavor to give an introduction to what exactly quantum computing is, with both the high level ideas compared to what we currently use for computing, and the Math and Computer Science needed to understand it. It will also discuss several applications of Quantum computing.

I. BACKGROUND OF COMPUTING: A BASELINE

First we will begin with a introductory baseline into computing, introducing several important concepts that are critical to understanding physical computing, as well as the theory behind it.

Our current model of physical computing

Our current model of physical computing can be boiled down to that of uniform families of reversible circuits. What do I mean by this? Circuits are networks composed of connections that carry bit values (0 for off, 1 for on) to various gates that perform the various base operations for computing. You may have heard of the elementary gates; AND, OR, XOR, NOT, NAND, NOR, and XNOR. These gates essentially boil down to a series of truth tables (1 for True, 0 for False), and form the elementary basis for all possible computing operations. They take one or two inputs, then based on the gate, perform a logical operation, then have one output. An example of a truth table for the AND gate is shown below:

Bit 1	Bit 2	AND
F (0)	F (0)	F (0)
F (0)	T (1)	F (0)
T (1)	F (0)	F (0)
T (1)	T (1)	T (1)

Something to note however, is that we do not actually need all of these gates to perform all operations, we actually only need one, NAND or NOR. The proof of this can be generated with a series of circuits and truth tables. Though these are the only gates we really need, the creation of the other gates allow for far more efficient calculations, as well omitting the mess of NAND or NOR combinations. But this still is an important concept, as in the event we cannot create specific gate, we can use the combination of a base gate to create an equivalent. (Consider limitations in the creation of quantum gates)

Algorithms and Complexity Theory

Now that we have a basis of how computing is achieved, we introduce an algorithm. An algorithm is defined as a procedure for realizing an information-processing task. In simpler terms, it is a series of instructions to run from start to finish with a well-defined input, and a well-defined output. With the notion of an algorithm, we introduce the concept of complexity. We can think of complexity as the amount of resources used by a computer to solve the algorithmic computation. Complexity is comprised of two main parts: *time complexity* and *space complexity*. Time complexity considers the amount of time taken by an algorithm to run, as a function of the length of the input. Space complexity considers the amount of space required by an algorithm to run, as a function of the length of the input. Consider two inputs of length n bits, then a computation of say, multiplication of two numbers. This computation could take on a time complexity of order n^2 , and the time required to compute the output would grow quadratically as the input increases. It is important to note however that the complexity of a computation can vary greatly based on the physical architecture of the computer. Perhaps on a architecture not optimized for multiplication, it could take on a complexity of order n^3 , resulting in significantly different time outcomes as the input increases. In the explicit calculations of time complexity via complexity analysis, the true run time may contain constants in front of the n -terms, in addition to lower order terms. The asymptotic analysis is governed by the highest order term, and we use the big \mathcal{O} bound to delineate the upper bound of how the computation runs as $n \rightarrow \infty$. For example, an algorithm with a run time of $3n^3 + 2n^2 + n + 3$ would be denoted as $\mathcal{O}(n^3)$. For expressing the resource requirements in terms the lower bound, we use Ω instead of \mathcal{O} .

Algorithmic Efficiency

With the notion of complexity defined, we introduce the definition of efficiency. An algorithm is said to be efficient if its complexity can be described in terms of a polynomial, $\mathcal{O}(n^k)$, where k is a constant and $k \in \mathbb{R}$, linear $\mathcal{O}(n)$, or logarithmic $\mathcal{O}(\log(n))$, or any combination of the three. If an algorithm has $\Omega(k^n)$ where k is a constant and $k \in \mathbb{R}$, then is considered to be exponential, and not efficient. If the \mathcal{O} bound of an algorithm cannot be

expressed by any of the three types of efficient bounds, then the algorithm's complexity is defined as superpolynomial. We will return to this algorithmic limitation later.

Computability and Efficiency

A keystone to understand what is and isn't computable depends on something called the Church-Turing Thesis. This states that a computing problem can be solved on any computer we build, if and only if it can be solved on a Turing machine. A Turing machine is a mathematical abstraction computational model consisting of a finite set of states, an infinite tape which contains symbols from a well-defined finite alphabet Σ that can be written to and read from the tape using a moving head and transition function that governs the movement of the head from state to state on the tape. The term computable actually corresponds to what can, and cannot be computed on a Turing machine. However, in order to extend this to computability thesis to computation efficiency, we introduce the notion of a probabilistic Turing machine. This functions identically to a standard Turing machine, except it has the ability to perform probabilistic functions, such as a "coinflip". Why would this be important? Some problems we only know how to solve efficiently using probabilistic operations. An example of such a problem is that of finding square roots modulo a prime, known as Cipolla's algorithm[1]. The notion of a probabilistic TM introduces the Strong Church-Turing Thesis, which states that a probabilistic Turing machine can *efficiently* simulate any realistic model of computation. To model this in a physical circuit, we construct a pseudo "coin flip" circuit gate, which can be implemented with the use of unreliable logic gates.[2] What this essentially means is that it is a logic gate who's output is not 100% certain, and can flip at any random point, similar to a coin. However, these are impossible to implement with any degree of consistency, and thus we find ourselves in need for a reliable, probabilistic alternative. I wonder what that could be?

Physical Limitations

Within the theory of computation, we can design complex algorithms for various tasks, basing the analysis on an idealized model of computing. But, as is the problem with an idealized theoretical model, we ignore the physical limitations of electronic circuits. Realistic

computing devices depend on the physics of electronics, and subsequently are bound by the limitations they incur. As a result, we introduce further additions to the complexity; the number of gates, depth, and width of the circuit. If we consider the application of a gate to be one 'depth-unit', the depth is the number of gate applications required to achieve the final output in the circuit. The depth can be cut by implementing gates in parallel. Finally, the width of a circuit is defined the total number of bits, or wire connections in the circuit.

II. QUANTUM COMPUTING

Now that we have a base understanding of computing, we can begin to delve into the quantum side of things. I briefly mentioned the idea of probabilistic operations earlier, and I will expand upon that further in this section.

Linear Algebra: Bits and Gates

To begin, let's consider a simple circuit of a singular wire and a gate. Remember the wire is equivalent to a bit, meaning it has 2 different states, on or off. We can express this wire's state in terms of a vector:

$$\begin{bmatrix} p_0 \\ p_1 \end{bmatrix} \quad (1)$$

Where p_0, p_1 is the probability for it to be in states 0 and 1 respectively. Now, let's consider the wire either 100% on or 100% off, thus the possible state vectors we have for this wire are state 0 with 100% probability:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad (2)$$

or state 1 with 100% probability:

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (3)$$

This matches our idea of an off-on circuit that we established earlier. Next, we introduce gates. If the wire's state is a vector, then the gate is a matrix operator on that state. To illustrate this point, let's consider the simple NOT gate. This gate negates the state, (0 goes

to 1, 1 goes to 0) thus we have:

$$\text{NOT} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad (4)$$

With a bit of linear algebra, we find the NOT operator is give by the matrix of the form:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (5)$$

Now that we've established a way to express a singular wire's state in terms of a vector, as well as gates as operators, what if we have more than one wire as an input? If you have worked with truth tables for different gates, you may see where this is going. With a two wire input, there are 4 possible states for the combination of wires. Let's consider the probabilities for bit 1 to be a_0, a_1 , and for bit 2, b_0, b_1 , where they represent the probability to be in states 0 or 1 respectively. Thus, the vector for their combined input is given by:

$$\begin{bmatrix} a_0 b_0 \\ a_0 b_1 \\ a_1 b_0 \\ a_1 b_1 \end{bmatrix} \quad (6)$$

This is equivalently the tensor product of the two 2-d vector states on their own.

$$\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \otimes \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \quad (7)$$

We would then write say, an AND operator as a 4x4 matrix to operate on this vector to output the same corresponding values as we'd expect. While we are working with strictly 0's and 1's for probabilities, the calculations are relatively straight-forward. However, we are setting up our notation with a probabilistic future in mind, where things aren't so cut and dry. An important thing to note: some operations are not reversible, ie, if you AND two inputs, a, b , and get an output of 0, given only that output, what was the input? (0,0)? (1,0)? (0,1)? Reversibility is very important to quantum computing, and every single quantum gate that forms the basis of the computations must be reversible.[3] In a classical computing sense, we can amend this by adding a few extra outputs to record the inputs so we have more information in the output, allowing for reversibility. The need for reversibility can then be kept in mind as we are constructing our quantum gate basis later on.

Relevant Quantum Mechanics

Now that we have an understanding of the notation for different operation on circuit states, we can begin to lay the groundwork for the quantum side of things. To set the stage, we must consider the concept of superposition of states. In quantum mechanics, we can measure an eigenvalue of a quantum system comprised of a superposition of states, and thus force the state into a corresponding eigenstate to that eigenvalue with a probability denoted by the original system. First, let's consider a simple example quantum system that is a superposition of two states, 0 and 1.

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle \quad (8)$$

We express the system in terms of ket notation denoting state $0 = |0\rangle$ and state $1 = |1\rangle$. For the simplicity of this example, we will define the basis we are expressing the system in such that the eigenstates correspond to the pure states of 0 and 1 respectively. With this in mind, we can also express $|\psi\rangle$ in terms of a vector:

$$|\psi\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix} \quad (9)$$

With some quick ket probability calculations, we find that the probabilities of finding the particle in state 0 or 1 when we measure the system are α_0^2 and α_1^2 respectively. This notion of superposition of states is where the probabilistic properties come in to play that we mentioned earlier.

The jump from classical to quantum: Qubits, what are they?

Just like a bit in classical computing, a qubit is the basic unit upon which quantum computing is built. A qubit, is a two-state quantum mechanical system, exactly like the one described above. In practice, states 0 and 1 correspond to a spin-1/2 system, where state 0 is represented by spin up, and state 1 is represented by spin down. Alternatively, states 0 and 1 could also be expressed as vertical and horizontal polarization respectively of a single photon. The important takeaway regardless of the implementation of the qubit is that it is a simple superposition of two states. Just like the classical bit, we can have more than one qubit-input. A set of 2 or more qubits is called a quantum register. The resulting tensor

product of n qubits is of dimension $2^n \times 1$. For example, consider a two qubit register. We can express it as follows:

$$|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad (10)$$

Where $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ represent the pure eigenstates of finding it in that configuration.

In practice, the resultant qubit register would be a mix of two ψ ket tensor products resulting in varying probabilities. Given two qubits, represented by $|\psi_1\rangle = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$ and $|\psi_2\rangle = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$

The new register would be represented as:

$$\begin{bmatrix} a_0b_0 \\ a_0b_1 \\ a_1b_0 \\ a_1b_1 \end{bmatrix} \quad (11)$$

with probabilities $(a_0b_0)^2, (a_0b_1)^2, (a_1b_0)^2, (a_1b_1)^2$ for states $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ respectively.

The Quantum Model of Computing

With a representation for both a single qubit and a qubit register in terms of vectors we can begin to build upon our basis with further. The next step is to take a look at quantum gates, and just like in the classical sense, we consider the gates to be operators on a vector that represents the input state. It is important to first note the properties of quantum gates. The gates are reversible and unitary. Reversible means that you can apply the inverse of the gate operator matrix to the output and retrieve the original input back. Unitary means that, given a gate operator, U , the transpose conjugate of U is also equal to U^{-1} . This is also referred to as the Hermitian adjoint of the matrix, denoted by a \dagger . Using this notation, we can also write $U^\dagger = U^{-1}$. The basis for quantum gates are given below:













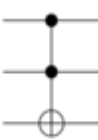
Operator	Gate(s)	Matrix
Pauli-X (X)	 	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)	 	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP	 	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

FIG. 1. A list of the basic quantum gates that form the basis of quantum computations[4]

III. APPLICATIONS

Finally, with the basic ideas and tools for quantum computing introduced, we can look at some of the specific applications of quantum computing, giving us reason for everything we have set up to this point.

Algorithms and advantages

Quantum algorithms are defined as a quantum circuit which acts on some input of qubits and ends with a measurement of the qubits. Some of the notable quantum algorithms include the Quantum Phase estimation, Quantum Fourier Transform, and Shor's Algorithm.

Quantum Phase Estimation

The Quantum Phase Estimation, also known as the quantum eigenvalue estimation algorithm, is used to estimate the phase or eigenvalue of a eigenvector of a Hermitian operator. What this essentially means is that given a wave ket, $|\psi\rangle$, and a Hermitian operator A such that $A|\psi\rangle = e^{i2\pi\theta}|\psi\rangle$, the algorithm estimates θ with high probability, with the drawback of only a small additive error ε . This algorithm is an essential building block for many other quantum algorithms, such as Shor's algorithm.[5]

Quantum Fourier Transform

The quantum Fourier transform, or QFT for short, is a linear transformation on qubits, and is the quantum equivalent of the inverse discrete Fourier transform. This algorithm is used as a building block in several other notable algorithms, like Shor's Algorithm. The QFT uses a simple decomposition of the input into Hermitian matrices, the discrete Fourier transform on 2^n amplitudes can be implemented as a quantum circuit consisting of $\mathcal{O}(n^2)$ Hadamard gates and controlled phase shift gates, where n is the number of qubits. Compared to the classical discrete Fourier transform, which takes $\mathcal{O}(n2^n)$ gates, exponentially more than the quantum equivalent. However, the quantum Fourier transform acts on a quantum state, whereas the classical Fourier transform acts on a vector, so not every instance that uses the classical Fourier transform can take use of this exponential speedup.[6]

Shor's Algorithm

A major advantage of Shor's Algorithm is that a previously superpolynomial complexity problem, integer factorization, can now be completed in polynomial time. The implications for finding algorithms for problems that were previously only superpolynomial given our

limited hardware are resounding. Part one of Shor's Algorithm turns the factoring problem into the problem of finding the period of a function and may be implemented classically. The quantum part comes is part two of the algorithm, where it finds the period using the quantum Fourier transform and this part is responsible for the quantum speedup.

The drawbacks of Shor's algorithm come from quantum modular exponentiation, which is significantly slower than the quantum Fourier transform and classical pre/post-processing. Currently, the best practical approach to deal with this limitation is to mimic conventional arithmetic circuits with reversible gates, starting with ripple-carry adders. These circuits however, use on the order of n^3 gates for n qubits. [7]

CONCLUSION

Quantum computing isn't a magical answer to all questions we face in computing. Problems which are undecidable using classical computers still remain undecidable using quantum computers. However, recall the issue we mentioned with unreliable logic gates. We cannot make a truly probabilistic classical gate. What makes quantum computing and quantum algorithms powerful is that they have the potential to solve problems faster than classical algorithms because they exploit the probability of quantum superposition and quantum entanglement, something that cannot be efficiently simulated on classical computers.

-
- [1] Leonard Eugene Dickson, *"History of the Theory of Numbers", Volume 1* p218
<https://archive.org/details/historyoftheoryo01dick/page/218/mode/2up?view=theater>
 - [2] Lakshmi N. B. Chakrapani, Krishna V. Palem, *"Probabilistic Boolean Logic"*, College of Computing Georgia Institute of Technology
 - [3] Prof. Dr. Alexis De Vos, *"Reversible Computing: Fundamentals, Quantum Computing, and Applications"*, Universiteit Gent
 - [4] *"Quantum Logic Gates"* https://en.wikipedia.org/wiki/Quantum_logic_gate
 - [5] Kitaev, A. Yu *"Quantum measurements and the Abelian Stabilizer Problem"*
 - [6] Coppersmith, D, *"An approximate Fourier transform useful in quantum factoring"*
 - [7] Shor, P.W, *"Algorithms for quantum computation: discrete logarithms and factoring"*