**CS 880: Quantum Information Processing** 9/2/2010

# Lecture 1: Introduction

Instructor: Dieter van Melkebeek Scribe: Dalibor Zelený

Welcome to CS 880, a course on quantum information processing. This area has seen a lot of development in the last twenty years, with interest increasing after Peter Shor found a polynomial-time quantum algorithm for factoring in 1994. In this course, we focus on computational aspects of quantum computing, quantum information, error-correction, and fault-tolerant computing. We do not focus on the issue of physical realizability of a quantum computer.

Today we begin with a brief account of the history of quantum computing that is relevant to our course, give an overview of topics covered in this course, and start developing a formal model of quantum computation.

## 1 Brief Historical Overview

### 1.1 Quantum Computing

In the 1980s, physicists Richard Feynman and Paul Benioff observed that simulating quantum systems on computers did not work well, and suggested that a different kind of a computer, based on quantum principles, could perform better. They also suggested that such a computer could speed up computation for other problems solved on classical computers.

David Deutsch, inspired by the Church-Turing thesis[1], gave the first formal model of a quantum computer in 1985. It is now known that the Church-Turing thesis holds for quantum computers; however, it is an open problem whether the strong version of the Church-Turing thesis holds for quantum computers. The consensus in the community is that the strong Church-Turing thesis does not hold for quantum computers.

A significant piece of evidence for the latter came in 1994 when Peter Shor gave polynomial-time quantum algorithms for factoring integers and for finding discrete logarithms. Today's best known classical algorithms for these problems still have exponential running times. Shor's algorithm motivated a vast amount of research in quantum algorithms. We briefly mention only a few as a full account would take too much time.

A key ingredient in Shor's algorithm, Fourier sampling, can be generalized to an efficient procedure for estimating (the phases of) eigenvalues of unitary operators. This phase/eigenvalue estimation estimation technique has found several applications, including a polylogarithmic algorithm for solving well-conditioned systems of linear equations. There are also some negative results in this line of research: it has been shown that we cannot solve graph isomorphism using Fourier sampling.

In 1995, Lov Grover showed how to search for an element in an unordered list of $n$ elements using $O(\sqrt{n})$ queries. In the classical and randomized settings, we require $\Theta(n)$ queries in the worst case. Applying Grover's algorithm to exhaustive search for a satisfying assignment to a Boolean

[1]The *Church-Turing thesis* states that Turing machines can decide the same set of languages as any other computational model such as lambda calculus. The *strong Church-Turing thesis*, in addition, says that we can simulate any other computational model on a Turing machine with only polynomial overhead.

formula yields a $O(2^{n/2})$ algorithm for satisfiability, while we don't know of any such algorithms in the classical setting. However, finding a subexponential algorithm for satisfiability is an open problem even on quantum computers. The consensus is that such algorithms do not exist.

Given that, a lot of the research in quantum computing focuses on NP-intermediate problems because some problems we suspect to be NP-intermediate do have efficient quantum algorithms. Examples of problems believed to be NP-intermediate are factoring, the discrete logarithm problem, the approximate shortest vector problem in lattices, and graph isomorphism. It may be the case, however, that all these problems have efficient quantum algorithms because they are actually in P.

### 1.2 Quantum Information

While the progress in realizing quantum computers has been slow, there has been more success in the area of quantum information. Shor's algorithm was a negative result for cryptography, as cryptosystems such as RSA rely on the hardness of factoring. On the other hand, quantum computation yields some more powerful cryptographic tools. For example, there is a key distribution protocol where an eavesdropper cannot observe a message without irreversibly altering it, thus making her presence known to the communicating parties.

Some research has focused on zero knowledge proofs[2] in the quantum setting. It turns out that the zero knowledge protocol for graph isomorphism, as well as several other zero knowledge protocols, remain zero knowledge even if we give the power of quantum computation to the verifier.

Other aspects studied in quantum information are teleportation, super-dense coding, non locality, and many more.

### 1.3 Quantum Error Correcting and Fault Tolerant Computing

Quantum behavior of algorithms is negatively influenced by *decoherence*, which is a term for the interaction between the quantum system and the outside environment. We would like to design error-correcting algorithms that cope with it. Unfortunately, even the error-correcing algorithms themselves may suffer from decoherence. Fault-tolerant computing solves the latter problem, provided the amount of interaction between the error-correcting procedure and the environment stays below a certain threshold.

As we will see later, error-correction is harder in the quantum setting. In the classical setting, we only have two states, zero and one, whereas there are more states in the quantum setting.

[2]In a *zero knowledge proof*, the goal of the prover is to convince the verifier that a particular statement is true, without revealing any additional information. For example, an NP witness that two graphs are isomorphic reveals additional information—the isomorphism from the vertex set of one graph to the vertex set of the other graph—so it's not a zero-knowledge proof. We want the proof to be such that if the verifier can compute something efficiently after interacting with the prover, it could have computed it efficiently even before the interaction took place.

## 2 Topics Covered in This Course

This course covers topics listed below.

1. The model of a quantum computer
2. Paradigms for efficient quantum algorithms
   (a) Phase estimation
   (b) Hidden subgroup problem - In this problem we have a group action that, for each coset, behaves the same on all elements of that coset. We are interested in the subgroup used to define the cosets.
   (c) Quantum random walks
3. Quantum communications and other interactive processes
4. Time permitting: Quantum error correction and fault-tolerant computation

## 3 Modeling Computation

Today we describe classical computation in terminology that is more suitable for describing quantum computation. Next time, we will give formal descriptions of probabilistic computation and quantum computation.

Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ be a relation. In this course, we think of it as a relation between inputs and outputs for some problem. *Computation* is a process that evolves the state of a system from an initial state to a final state, while transforming the input into an output. If, for every input $x$, our computation produces an output $y$ such that $(x, y) \in R$, we say that it *realizes* $R$.

Note that a relation is more general than a function. A function is a relation such that at most one pair $(x, y)$ is in $R$ for each $x$. We consider relations because many problems allow multiple strings $y$ to be associated with a given string $x$. For example, in the shortest path problem, we can have multiple shortest paths $y$ in a single instance $x$.

We consider three stages of computation.

1. Initialization of the system with the input $x$.
2. Evolution of the system by performing a sequence of elementary steps according to some rules.
3. Observation of the final state of the system, from which we extract the output $y$.

We need not pay much attention to steps 1 and 3 above in classical computation. These steps require more consideration in the case of quantum computation, but we are not going to discuss them in this course. We focus on the second step, which should have the following properties.

1. It should be *physically realizable*. This means that elementary operations should act on a small part of the system, i.e., it should act on a small (constant) number of bits that are physically close to each other.

2. The sequence of elementary operations should follow "easily" from a short description. For example, we could describe it by a program of constant length.

We usually use Turing machines to describe classical computation. Unfortunately, quantum Turing machines are much more cumbersome to deal with. Thus, we present an alternative way of describing a computation. In particular, we use gates to describe the evolution of the state of a system from the initial configuration to the final configuration. We apply the gates in a sequence in order to carry out the evolution of state, thus producing a circuit. We then argue that this circuit is easily computable from a short description.

### 3.1 Circuits Describe Classical Computation

We first discuss how to use gates to describe computation, and then put them together to form a circuit. We also give a mathematical description of computation using linear transformations, which will be useful later.

We view the state of a classical computer as a concatenation of $m$ subsystems, each of which has two possible states, 0 and 1. This allows us to describe the state of the computer as a binary string $s$ of length $m$. We encode the input of length $n$ into the first $n$ parts of the system, and set all the other parts to 0. Once the computation terminates, we read the final state to extract the output. See Figure 1.

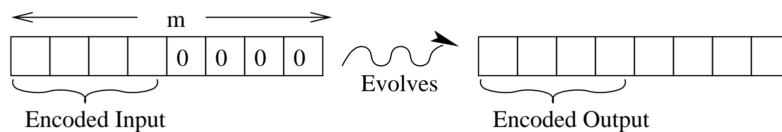


Figure 1: Computation of a classical computer

We often use the "ket" notation $|s\rangle$ to denote a state $s$ of length $m$, and we think of $|s\rangle$ as a column vector of length $2^m$ with zeros everywhere but the position indexed by the number represented by $s$ in binary (so $|s\rangle$ is the characteristic vector of the state $s$, with positions in the vector corresponding to states in lexicographical ordering). We then think of the action of an operator on a state $s$ as a linear transformation acting on the vector $|s\rangle$. We can represent this linear transformation by a $2^m \times 2^m$ matrix.

Recall that we want operators to be physically realizable. We can represent them as gates, for example NAND gates. In Figure 2, we show a NAND gate with two inputs and two outputs, where the first output is the NAND of the inputs, and the second output is the same as the second input.

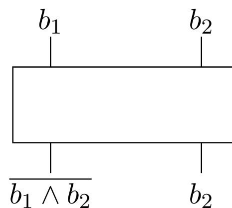


Figure 2: A NAND gate

*Example:* The NAND gate from Figure 2 operates on a state $s$ of length $m = 2$. We represent the states using the "ket" notation as follows:

$$|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad \text{and } |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

The NAND gate causes the following changes to the current state:

$$\begin{aligned} |00\rangle &\rightarrow |10\rangle, \\ |01\rangle &\rightarrow |11\rangle, \\ |10\rangle &\rightarrow |10\rangle, \\ |11\rangle &\rightarrow |01\rangle. \end{aligned}$$

The matrix corresponding to this transformation is

$$T = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

What the matrix $T$ really does to $|s\rangle$ is that it moves the position of the only one in $|s\rangle$ to the position at index whose binary representation corresponds to the new state. If $|s\rangle$ has its only one in position $j$, and $s$ gets mapped to $s'$, the $j$-th column of the matrix $T$ is going to have its only one in position $i$ where $i$ is the only index where $|s'\rangle$ has a one.

In our case, we can see that $T$ moves the one from position $0 = (00)_2$ to position $2 = (10)_2$ (so it adds 2 to 0). Similarly, it moves the one from position 1 to 3 (so it adds 2 again), does nothing when the one is in position 2, and moves the one two positions back when it's in position 3.

Our goal now is to describe the matrix for the transformation of states of length $m$ when two consecutive states are connected by a NAND gate. The matrix changes depending on the bits the NAND gate acts on. Consider the case where the states are represented by $m$ bits, and where the NAND gate acts on two consecutive bits at positions $m - k - 2$ and $m - k - 1$ (where the last bit has index $m - 1$). We use $T_{m,k}$ to denote the matrix. Figure 3 shows the case $k = 0$. We start by describing $T_{m,0}$ and then use the intuition behind its description to describe $T_{m,k}$.

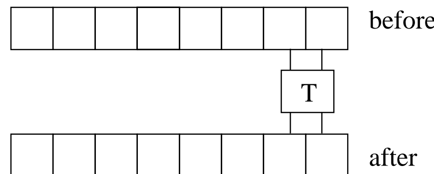


Figure 3: Two consecutive states with the last two bits connected by a gate

The output of the transformation represented by $T_{m,0}$ depends only on the last two bits of the state, modifies only those last two bits, and keeps the first $m - 2$ bits fixed. We discussed how the only one in $|s\rangle$ is moved by $T$ for each of the combinations of the last two bits in $|s\rangle$. The combinations of the last two bits repeat "periodically", which immediately gives us the matrix

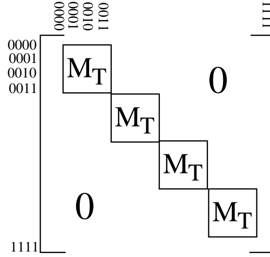


Figure 4: The matrix $T_{4,0} = I_4 \otimes T$

$T_{m,0} = I_{2^{m-2}} \otimes T$. Figure 4 shows $T_{m,0}$ for $m = 4$. We get $T_{m,k} = I_{2^{m-2-k}} \otimes T \otimes I_{2^k}$ by similar reasoning. ⊠

If we consider putting together multiple copies of Figure 3 in a series, with the "after" part of one application of a gate being the "before" part of the next application of a gate (perhaps at different positions of the state), we obtain a circuit that describes computation. We give more details after a brief discussion of Turing machines.

### 3.2 Turing Machines

In the previous section, we described computation using a circuit. However, it is possible that the circuit is different for each input length, and maybe even for two inputs of the same length. Therefore, the next question we must answer is how to compute the description of any of these circuits from one short description.

The Turing machine is the standard formal model of computation. Here we show that we can represent the computation of a Turing machine by a circuit, and that these two models of computation are equivalent, which implies that we can use either one to represent computation. In fact, we need a little more. Recall that we want our basic operations (e.g., gates in Section 3.1) to be computable from a short description. We show that the circuit described in Section 3.1 is computable from a short descirption (we call such circuits *uniform*, and we use the term *polynomial-time uniform* if we can compute them in time polynomial in the circuit size).

A Turing machine acts on a register of $m$ bits, and the actions are governed by some finite control. The finite control has a pointer $p$ to one bit in the register. In each step, the finite control reads the bit of the register pointed to by $p$, and looks at the current state. Afterwards, it writes a bit at the position pointed to by $p$, goes to another state, and moves $p$ to the left, to the right, or chooses not to move it. We can describe the action in each step by a transition function

$$\delta : Q \setminus \{q_{\text{halt}}\} \times \Gamma \to Q \times \Gamma \times \{L, P, R\},$$

where $Q$ is the finite set of states, $q_{\text{halt}}$ is the state in which computation terminates, $\Gamma$ is the finite alphabet of the register, and the set $\{L, P, R\}$ represents the three possibilities for the movement of the pointer $p$: move left, stay put, and move right.

Note that the actions of the transition function are local (they only affect the bit of the register pointed to by $p$, and move $p$ to an adjacent bit). This satisfies the physical realizability requirement mentioned earlier. Furthermore, we only need to describe the finite control in order to describe a Turing machine. Since the transition function has finite size, Turing machines satisfy the computability from a short description requirement as well. We can describe the transition function by

a circuit with a fixed number of gates, and we can think of that circuit as just one big gate. We can also encode where $p$ points as part of the state, and modify our gate to account for that. Now we use this gate to construct the circuit in Section 3.1. Since the gate is easily computable, so is the circuit, provided we can compute which bits of two consecutive states are connected by that gate for each pair of consecutive states. It turns out that this is possible and can be done in time that is polynomial in the Turing machine's running time. For another construction of a circuit that represents a Turing machine's computation, see Chapter 6 of [1].

It is easy to show that computation done on polynomial-time uniform circuits can be carried out on a Turing machine. A Turing machine can compute a polynomial-time uniform circuit in time polynomial in the circuit size, and then evaluate that circuit again in time polynomial in the circuit size. Therefore, we get the following theorem.

**Theorem 1.** *Turing machine computations and polynomial-time uniform circuits are equivalent in power up to polynomial factors in the running time.*

It turns out that Theorem 1 transfers to the quantum setting as well.

Note that we use Turing machines in two ways here. One use is as a model of computation that is equivalent to uniform circuits, and another use is to compute the description of a uniform circuit. This is not circular logic.

# 4 Next Time

Next time we will describe probabilistic and quantum computation using the formalism developed in this lecture.

# References

[1] Sanjeev Arora and Boaz Barak. Computational Complexity: A Modern Approach. *Cambridge*, 2009.