6.080/6.089 GITCS Feb 12, 2008

Lecture 3

Lecturer: Scott Aaronson *Scribe: Adam Rogal*

1 Administrivia

1.1 Scribe notes

The purpose of scribe notes is to transcribe our lectures. Although I have formal notes of my own, these notes are intended to incorporate other information we may mention during class - a record for future reference.

1.2 Problem sets

A few comments on the problem sets. Firstly, you are welcome to collaborate, but please mark on your problem sets the names of whom you worked with. Our hope is to try all the problems. Some are harder than others; there are those marked challenge problems as well. If you can't solve the given problem, be sure to state what methods you tried and your process up to the point you could not continue. This is partial credit and much better than writing a complete, but incorrect solution. After all, according to Socrates *the key to knowledge is to know what you don't know.*

1.3 Office hours

We will have office hours once a week.

2 Recap

2.1 Computer science as a set of rules

We can view computer science as the study of simple set of rules and what you can and can't build with them. Maybe the first example of that could be considered Euclidian geometry. And the key to discovering what processes we can build is that these rules are well-defined.

2.2 Logic

The field of logic focuses on automating or systematizing not just any mechanical processes, but rational thought itself. If we could represent our thoughts by manipulations of sequences of symbols, then in principle we could program a computer to do our reasoning for us.

We talked the simplest logical systems which were the only ones for thousands of years. Syllogisms and propositional logic, the logic of Boolean variables that can be either true or false, and related to each other through operators like and, or, and not. We finally discussed first order logic.

2.2.1 First order logic

The system of first order logic is built up of sentences. Each of these sentences contain variables, such as x, y, and z. Furthermore we can define functions which take these variables as input.

For example: let's define a function $Prime(x)$. Given an integer, it will return true if the number is prime, false if it is composite. Just like functions in any programming languages, we can build functions out of other functions by calling them as subroutines. In fact, many programming languages themselves were modeled after first order logic.

Furthermore, as in propositional logic, symbols such as $\wedge$ (and), $\vee$ (or), $\neg$ (not), and $\rightarrow$ (implies) allow us to relate objects to each other.

Quantifiers are a crucial part of first order logic. Quantifiers allow us to state propositions such as "Every positive integer x is either prime or composite."

$$\forall x.Prime(x) \vee Composite(x)$$

There's a counterexample, of course, namely 1. We can also say: "There exists an x, such that something is true."

$$\exists x.Something(x)$$

When people talk about first-order logic, they also normally assume that the equals sign is available.

2.2.2 Inference rules

We want a set of rules that will allow us to form true statements from other true statements.
Propositional tautologies:

$$A \vee \neg A$$

Modus ponens:

$$A \wedge (A \rightarrow B) \rightarrow B$$

Equals:

$$Equals(X, X)$$

$$Equals(X, Y) \iff Equals(Y, X)$$

Transitivity property:

$$Equals(X, Y) \wedge Equals(Y, Z) \rightarrow Equals(X, Z)$$

Furthermore, we have the rule of change of variables. If you have a valid sentence, that sentence will remain valid if we change variables.

2.2.3 Quantifier rules

If $A(x)$ is a valid sentence for any choice of x, then for all x, $A(x)$ is a valid sentence. Conversely, if $A(x)$ is a valid sentence for all x, then any $A(x)$ for a fixed x is a valid sentence.

$$A(X) \iff \forall x.A(x)$$

We also have rules for dealing with quantifiers. For example, it is false, that for all x, $A(x)$ iff there exists an x, $\neg A(x)$.

$$\neg\forall x.A(x) \iff \exists\neg A(x)$$

2.2.4 Completeness theorem

Kurt Gödel proved that the rules thus stated were all the rules we need. He proved that if you could not derive a logical contradiction by using this set of rules, there must be a way of assigning variables, such that all the sentences are satisfied.

3 Circuits

Electrical engineers views circuits to be complete loops typically represented in figure 1. However, in computer science, circuits have no loops and are built with logic gates.

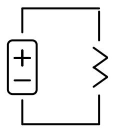

Figure 1: A simple EE circuit.

3.1 Logic gates

The three best-known logic gates are the *NOT*, *AND*, and *OR* gates shown in figure 2.

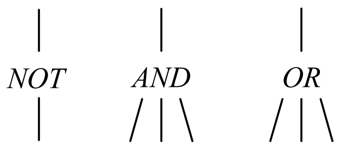

Figure 2: The logical gates *NOT*, *AND*, and *OR*.

Though primitive on their own, these logic gates can be strung together to form complex logical operations. For example, we can design a circuit, shown in figure 3, that takes the majority of 3 variables: x, y, and z. We can also use De Morgan's law to form a *AND* gate from an *OR* gate and vice versa as shown figure 4.

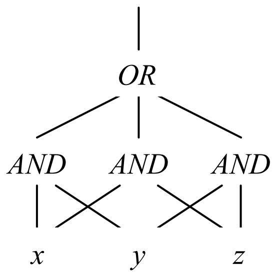

Figure 3: The majority circuit.

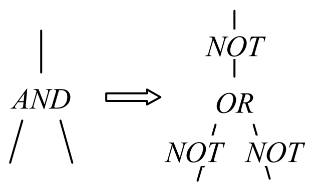

Figure 4: An *AND* gate can be constructed from an *OR* and three *NOT* gates by using De Morgan's law.

These logic gates can also be combined to form other gates such as the *XOR* and *NAND* gates shown in figure 5. Conversely, by starting with the *NAND* gate, we can build any other gate we want.

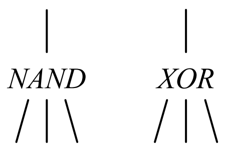

Figure 5: *NAND* and *XOR* gates.

On the other hand, no matter how we construct a circuit with *AND* and *OR* gates, if the input is all 1's we can never get an output of 0. We call a Boolean function that can be built solely out of *AND* and *OR* gates a *monotone* Boolean function.

Are there any other interesting sets of gates that *don't* let us express all Boolean functions? Yes: the *XOR* and *NOT* gates. Because of their linearity, no matter how we compose these gates we can never get functions like *AND* and *OR*.

4 Puzzle

Here's an amusing puzzle: can you compute the NOT's of 3 input variables, using as many AND/OR gates as you like but only 2 NOT gates?

4.0.1 Limitations

Although we have discovered that circuits can be a powerful tool, as a model of computation they have some clear limitations. Firstly, circuits offer no form of storage or memory. They also have no feedback; the output of a gate never gets fed as the input. But from a modern standpoint, the *biggest* limitation of circuits is that (much like computers from the 1930s) they can only be designed for a fixed-size task. For instance, one might design a circuit to sort 100 numbers. But to sort 1000 numbers, one would need to design a completely new circuit. There's no *general-purpose* circuit for the sorting task, one able to process inputs of arbitrary sizes.

5 Finite automata

We'll now consider a model of computation that *can* handle inputs of arbitrary length, unlike circuits – though as we'll see, this model has complementary limitations of its own.

5.1 Description

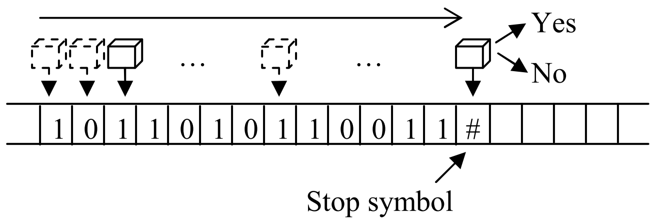

Figure 6: At any given time, the machine ahs some unique state. The machine reads the tape in one motion (in this case left to right) and the state changes depending on the value of the current square. When the reaches the stop state (signaled by the # sign, the machine returns a yes or no answer - an accept or reject state respectively.)

The simple way of thinking of a finite automaton is that it's a crippled computer that can only move along memory in one direction. As shown in figure 6, a computer with some information written along a tape, in some sort of encoding, will scan this tape one square at a time, until it reaches the stop symbol. The output of this machine will be a yes or no - accept or reject. This will be the machine's answer to some question that it was posed about the input.

5.2 State and internal configuration

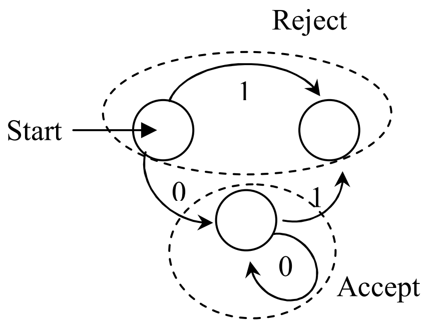

Figure 7: This simple machine has 3 states. Given an input of 0 or 1, the state will transition to a new state. The final state will determine its output - accept or reject.

It is unnecessary to determine what the internal configuration of this machine is. We can abstract this notion into the statement that this machine will have some state and the ability to transition between states given a certain input. The machine will begin with a start state, before it has read any input. When the machine reads the stop symbol, the correct state will determine if the machine should output an accept or reject.

It is crucial that we define the machine as having a finite number of states. If the machine had an infinite number of states, then it could compute absolutely *anything*, but such an assumption is physically unrealistic.

5.3 Some examples

Let us design a machine that determines if any 1's exist in a stream given the alphabet of 0 or 1. We define two states of the machine - 0 and 1. The 0 represents the state that the machine has not seen a 1 yet. The 1 state represents the state that the machine has seen a 1. When the machine has transitioned to the 1 state, neither a 1 or 0 will ever change the state back to 0. That is, regardless of input or length of input, our question, "Are there any 1's in the stream?" has been answered. Therefore, the 1 state should produce an accept, while the 0 state should produce a reject when a stop symbol has been reached.

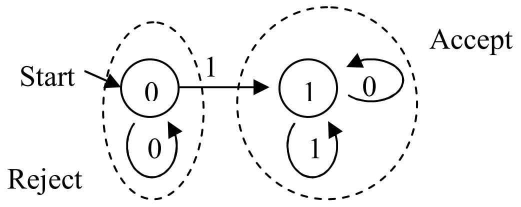

Figure 8: This FA determines if any 1's exist in our data stream.

Let us now design a machine that determines if the number of 1's is even or odd in the stream. We define two states again - 0 and 1. The 0 state represents a machine that has seen an even number of 1's and the 1 state describes a machine that has seen an odd number of 1's. An input of 0 will only transition the state to itself. That is, we are only concerned about the number of 1's in this stream. At each input of a 1, the machine will alternate state between 0 and 1. The final state will determine if the data stream has seen an even or odd number of 1's, with 1 being set as the acceptance state.

It should be noted that regardless of input size, this machine will determine the correct answer to the question we posed. Unlike with circuits, our machine size was not dictated by the size of the input.

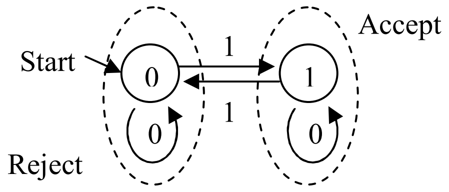

Figure 9: This FA determines if there are an even or odd number of 1's in our data stream.

5.4 Palindromes

Let us now explore if we could create a finite machine that can determine if an input string is a palindrome, a string that reads the same backwards and forwards. The input will be finite, and there will be a terminator at the end. We begin by defining the possible states of the machine. If we let our machine contain 2^N states, then as shown in figure 10, we could just label each final leaf as an accept or reject for every possible sequence of 1's and 0's.

The question still remains, can we create a machine with a finite number of states that can

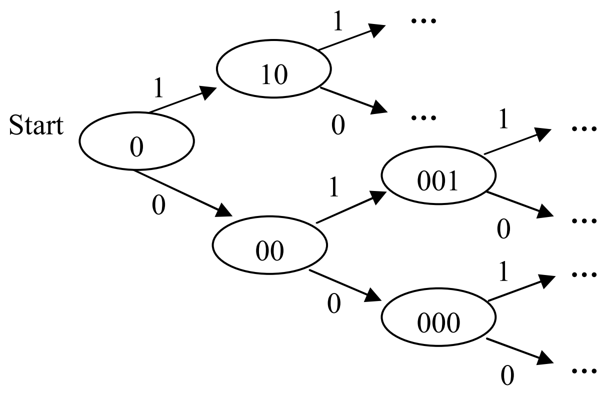

Figure 10: For a stream of N bits, a finite automaton, intended to determine if the stream is a palindrome, grows exponentially. For N bits, 2^N states are required.

act as a palindrome detector. The answer lies in using the Pigeonhole Principle to analyze the limitations of finite automata.

5.5 The Pigeonhole Principle

The Pigeonhole Principle states that if we have N pigeons and we want put them into $N - 1$ holes, at least one hole will have two or more pigeons. Although very simple, this principle allows us to prove that no finite automaton can act as a palindrome detector.

5.5.1 A digression: proving the pigeonhole principle

Even though the pigeonhole principle is simple, it is non-trivial to prove in simple systems of logic. We can express the requirements that every pigeon goes into some hole, and that no two pigeons go into the same hole, using propositional logic. The challenge is then to prove that not all the statements can be true, using only mechanical logical manipulation of the statements (and not higher-order reasoning about what they ”mean”).

In other words, the pigeonhole principle seems obvious to us because we can stand back and see the larger picture. But a propositional proof system like the ones we saw in the last lecture can't do this; it can only reason locally. (“Let's see: if I put this pigeon here and that one there ... darn, *still* doesn't work!”) A famous theorem of Haken states that any proof of the Pigeonhole Principle based on ”resolution” of logical statements, requires a number of steps that increases exponentially with N (the number of pigeons). This is an example of something studied by a field called *proof complexity*, which deals with questions like, “does any proof have to have a size that is exponentially larger than the theorem we are trying to prove?”

5.6 Using the Pigeonhole Principle for palindromes

We use the Pigeonhole Principle to prove that no finite automaton that can be constructed such that we can detect if any string is a palindrome.

To begin this proof, let us split a palindrome down the middle. We will ignore everything about the finite automaton except its state at the middle point; any information that the automaton will carry over to the second half of the string, must be encoded in that state.

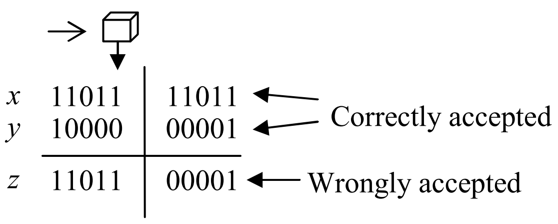

Figure 11: By using the Pigeonhole principle, we can show that we can split two strings at their reflection points such that a finite automaton will be in at the same state for both sub strings. We can then cross the two strings to form a new string that "tricks" the machine into thinking that it has correctly accepted a string as a palindrome.

A finite automaton must have a fixed number of states. On the other hand, there are infinitely many possibilities for the first half of the string. Certainly, you can't put infinitely many pigeons into a finite number of holes without having at least one hole with at least two pigeons. This means that there is at least one state that does "double duty," in that two different first halves of the string lead to the same state.

As shown in figure 11, we consider two palindromes x and y. If the machine works correctly, then it has to accept both of them. On the other hand, for some x, y pair, the machine will lie in the same state for both x and y when it's at the halfway point. Then by crossing the remaining halves of x and y, we can create a new string, z, which is accepted by the machine even though it's not a palindrome. This proves that no finite automaton exists that recognizes all and only the palindromes.

5.7 Regular expressions

Regular expressions allow us to search for a keyword in a large string. Yet, they are more powerful than simply searching for the keyword 110 in the string 001100. We can use regular expressions to locate patterns as well.

For example, we can create an expression like (0110)|(0001) which will either match the keyword 0110 or 0001. We can also create expressions that will find any 3 bit string with a 1 in the middle: (0|1)1(0|1).

We can also use more advanced characters such as the asterisk to represent repetition. (0|1)1(0|1)0* searches for any 3 bit string with a 1 in the middle followed by any number of 0's. We can also repeat larger patterns such as [(0|1)1(0|1)]*. This states that we would like to match any number of 3 bit strings with 1's in the middle. It should be noted that each time the pattern repeats, the 0 or 1's can be chosen differently.

We can now state (without proof) a very interesting theorem: any language is expressible by a regular expression, if and only if it's recognized by a finite automaton. Regular expressions and finite automaton are different ways of looking at the same thing.

To give an example: earlier we created a finite automaton that was able to recognize all strings with an even number of 1's. According to the theorem, there must be regular expression that generates this same set of strings. And indeed there is: 0*(0*10*1)*.

6 Nondeterministic finite automata

Nondeterministic finite automata represent machines that can not only transition between states, but between sets of states. As before, we have a machine that reads a tape from left to right with a finite number of states. When the machine reads an input, each state that the machine is now on, is allowed to transition to any other states emanating from the previous states based on the input. The machine is in acceptance if any final state is an accept state.

You might guess that NDFA's (nondeterministic finite automata) would be much more powerful than DFA's (deterministic finite automata). This is not the case, however: given an NDFA with N states, we can always simulate it by a DFA with 2^N states, by creating a single state in the DFA to represent each *set* of states in the NDFA.