

Theory of Computation

Lectures delivered by Michael Sipser
Notes by Holden Lee

Fall 2012, MIT

Last updated Tue. 12/11/2012

Contents

Lecture 1 Thu. 9/6/12

§1 Overview 5 §2 Finite Automata 6 §3 Formalization 7

Lecture 2 Tue. 9/11/12

§1 Regular expressions 12 §2 Nondeterminism 13 §3 Using nondeterminism to show closure 17 §4 Converting a finite automaton into a regular expression 20

Lecture 3 Thu. 9/13/12

§1 Converting a DFA to a regular expression 22 §2 Non-regular languages 25

Lecture 4 Tue. 9/18/12

§1 Context-Free Grammars 30 §2 Pushdown automata 33 §3 Comparing pushdown and finite automata 33

Lecture 5 Thu. 9/20/12

§1 CFG's and PDA's recognize the same language 37 §2 Converting CFG→PDA 38
§3 Non-CFLs 39 §4 Turing machines 42

Lecture 6 Tue. 9/25/12

§1 Turing machines 44 §2 Philosophy: Church-Turing Thesis 50

Lecture 7 Thu. 9/27/12

§1 Examples of decidable problems: problems on FA's 52 §2 Problems on grammars 56

Lecture 8 Tue. 10/2/12

§1 Languages 58 §2 Diagonalization 59 §3 A_{TM} : Turing-recognizable but not decidable 63 §4 Showing a specific language is not recognizable 65

Lecture 9 Thu. 10/4/12

§1 Reducibility 67 §2 Mapping reducibility 69

Lecture 10 Thu. 10/11/12

§1 Post Correspondence Problem 72 §2 Computation Histories and Linearly Bounded Automata 73 §3 Proof of undecidability of PCP 76

Lecture 11 Tue. 10/16/12

§1 Computation history method 77 §2 Recursion theorem 79 §3 Logic 83

Lecture 12 Thu. 10/18/12

§1 Introduction to complexity theory 84 §2 Time Complexity: formal definition 87

Lecture 13 Tue. 10/23/12

Lecture 14 Tue. 10/30/12

§1 P vs. NP 91 §2 Polynomial reducibility 92 §3 NP completeness 95

Lecture 15 Thu. 11/1/12

§1 Cook-Levin Theorem 101 §2 Subset sum problem 105

Lecture 16 Tue. 11/6/12

§1 Space complexity 108 §2 Savitch's Theorem 112

Lecture 17 Thu. 11/8/12

§1 Savitch's Theorem 114 §2 PSPACE-completeness 116

Lecture 18 Thu. 10/11/12

§1 Games: Generalized Geography 120 §2 Log space 124 §3 $L, NL \subseteq P$ 126

Lecture 19 Thu. 11/15/12

§1 L vs. NL 129 §2 NL-completeness 130 §3 $NL=coNL$ 132

Lecture 20 Tue. 11/20/12

§1 Space hierarchy 134 §2 Time Hierarchy Theorem 137

Lecture 21 Tue. 11/27/12

§1 Intractable problems 138 §2 Oracles 142

Lecture 22 Thu. 11/29/12

§1 Primality testing 144 §2 Probabilistic Turing Machines 145 §3 Branching programs 147

Lecture 23 Thu. 10/11/12

§1 EQ_{ROBP} 150

Lecture 24 Thu. 12/6/12

§1 Interactive proofs 155 §2 $\text{IP}=\text{PSPACE}$ 158

Lecture 25 Tue. 12/11/2012

§1 $\text{coNP} \subseteq \text{IP}$ 161 §2 A summary of complexity classes 164

Introduction

Michael Sipser taught a course (18.404J) on Theory of Computation at MIT in Fall 2012. These are my “live- \TeX ed” notes from the course. The template is borrowed from Akhil Mathew.

Please email corrections to `holden1@mit.edu`.

Lecture 1

Thu. 9/6/12

Course information: Michael Sipser teaches the course. Alex Arkhipov and Zack Rumschirm teach the recitations. The website is <http://math.mit.edu/~sipser/18404>.

The 3rd edition of the textbook has an extra lesson on parsing and deterministic free languages (which we will not cover), and some additional problems. The 2nd edition is okay.

§1 Overview

1.1 Computability theory

In the first part of the course we will cover **computability theory**: what kinds of things can you solve with a computer and what kinds of things can you not solve with a computer? Computers are so powerful that you may think they can do anything. That's false. For example, consider the question

Does a given program meet a given specification?

Is it possible to build a computer that will answer this question, when you feed it any program and any specification? No; this problem is *uncomputable*, impossible to solve with a computer.

Is it possible to build a computer so that when I feed it math statements (for instance, Fermat's last theorem or the twin primes conjecture), it will output true or false? Again, no. No algorithm can tell whether a math statement is true or false, not even in principle (given sufficiently long time and large space).

We'll have to introduce a formal model of computation—what do we mean by a computer?—to talk about the subject in a mathematical way. There are several different models for computation. We'll talk about the simplest of these—finite automata—today.

Computability theory had its heyday 1930 to 1950's; it is pretty much finished as a field of research.

1.2 Complexity theory

By contrast, the second half of the course focuses on **complexity theory**. This picks up where computability left off, from 1960's to the present. It is still a major area of research, and focuses not on whether we *can* solve a problem using a computer, but how *hard* is it to solve? The classical example is factoring. You can easily multiply 2 large numbers (ex. 500-digit numbers) quickly on a laptop. No one knows how or if you can do the opposite—factor a large number (ex. 1000-digit number)—easily. The state of the art is 200 digits right now. 250 digits and up is way beyond what we can do in general.

We'll define different ways of measuring hardness: time and space (memory). We'll look at models that are specific to complexity theory, such as probabilistic models and interactive

proof systems. A famous unsolved problem is the P vs. NP problem, which will be a theme throughout our lessons on complexity. In these problems, some kind of “searching” is inevitable.

1.3 Why theory?

What is value of studying computer science theory? People question why a computer science department should invest in theoretical computer science. This used to be a big issue.

Firstly, theory has proved its value to other parts of computer science. Many technologists and companies grew up out of the work of theorists, for example, RSA cryptography. Akamai came out of looking at distributed systems from a theoretical point of view. Many key personnel in Google were theorists, because search has a theoretical side to it. Theorists have played a role in building the computer science industry.

Second, theory as a part of science. It’s not just driven by its applications, but by *curiosity*. For example, “how hard is factoring?” is a natural question that it is intrinsically worthwhile to answer. Our curiosity is makes us human.

Computer science theory may also help us understand the brain in the future. We understand heart and most of our other organs pretty well, but we have only the faintest idea how the brain works. Because the brain has a computation aspect to it, it’s entirely possible that some theory of computation will help solve this problem.

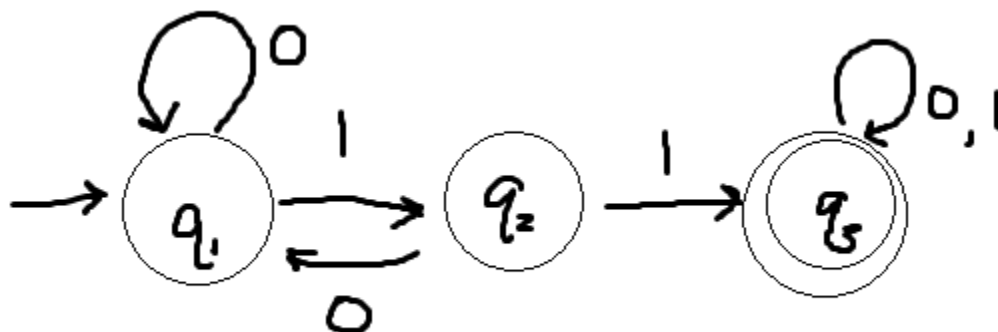
Is there more stuff to do? Certainly. We’re still at the very beginnings of computer science and theory. Whole realms out there are worthy of exploration.

That’s what and why we do this stuff.

§2 Finite Automata

2.1 An example

We need to set up our models for computability theory. The first one will be finite automata. An example of a finite automata is given by the following picture.



The **states** are $\{q_1, q_2, q_3\}$. The **transitions** are arrows with 0 or 1, such as $\xrightarrow{0}$. The **start** state is q_1 (it has a regular arrow leading to it) and the accept states is $\{q_3\}$ (it has a double circle). Note each state has 2 arrows exiting it, 0 and 1.

How does this automaton work when we feed it a string such as 010110? We start at the start state q_1 . Read in the input symbols one at a time, and follow the transition arrow given by the next bit.

- 0: take the arrow from q_1 back to q_1 .
- 1: take the arrow from q_1 to q_2 .
- 0: take the arrow back to q_1 .
- 1: get to q_2
- 1: get to q_3
- 0: stay at q_3 .

Since q_3 is an accept state, the output is “accept.” By contrast, the input state 101 ends at q_2 ; the machine does not accept, i.e. it rejects the input.

Problem 1.1: What strings does the machine accept?

The machine accepts exactly the strings with two consecutive 1's. The language of A , denoted $L(A)$, is the set of accepted strings, i.e. the language that the machine recognizes. (This term comes from linguistics.)

We say that the language of A is

$$L(A) = \{w : w \text{ has substring } 11\}.$$

§3 Formalization

We now give a formal definition of a finite automaton.

Definition 1.1: A **finite automaton** is a tuple $M = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states,
- Σ is a finite alphabet (collection of symbols, for instance $\{0, 1\}$),
- δ is the transition function that takes a state and input symbol and gives another state

$$\begin{aligned} \delta : Q \times \Sigma &\rightarrow Q \\ (q, a) &\mapsto r. \end{aligned}$$

We denote this with a circle q and an arrow \xrightarrow{a} leading to a circle r .

- $q_0 \in Q$ is a start state.
- $F \subseteq Q$ is a set of accept states.

To take this further, we're going to define the language of an automaton. (We did this informally by following our finger on a path. We're just doing this formally now.)

Definition 1.2: Say M **accepts input string** $W = W_1 \cdots W_n$ where each $W_i \in \Sigma$, if r_0, \dots, r_n is a sequence from Q (of states gone through) where

- $r_0 = q_0$ (start at start state),
- $r_n \in F$ (ends at an accept state),
- and for each $i > 0$ and each $i > 0$, $r_i = \delta(r_{i-1}, w_i)$ (each next state is obtained the previous state by reading the next symbol and using the transition function).

The **language** of M is

$$L(M) = \{w : M \text{ accepts } w\}.$$

Note M accepts certain strings and rejects certain strings, but M recognizes just 1 language, the *collection* of all recognized strings.¹

Note there is a special string, the empty string of length 0, denote ε . By contrast, the empty *language* is denoted by ϕ .

Definition 1.3: A language is **regular** if some finite automaton recognizes it.

For instance $\{w : w \text{ has substring } 11\}$ is a regular language because we exhibited a automaton that recognizes it.

3.1 Building automata

Problem 1.2: Build an automaton to recognize...

- The set of strings with an even number of 1's.
- The set of strings that start and end with the same symbol.

When we have a finite automaton, and we want to design an automaton for a certain task, think as follows:

¹If L' is a subset of L and M recognizes L , we *don't* say M recognizes L' .



The states of the automaton represent its memory. Use different states for different possibilities.

For example,

1. an automaton that accepts iff the string has an even number of 1's will have to count number of 1's mod 2. You want to have one state for each possibility.
2. an automaton that accepts iff the first equals the last symbol will have to keep track of what the first symbol is. It should have different states for different possibilities of the first symbol.

In the next lecture and a half we'll seek to understand the regular languages. There are simple languages that are not regular, for example, the language that has an equal number of 0's and 1's is not regular.

Proof sketch. Such an automaton would have to keep track of the difference between number of 0's and 1's so far, and there are an infinite number of possibilities to track; a finite automaton has only finitely many states and can keep track of finitely many possibilities. \square

3.2 Closure properties of languages

Definition 1.4: We call the following 3 operations on languages **regular operations**.

- \cup union: $A \cup B = \{w : w \in A \text{ or } w \in B\}$
- \circ concatenation:

$$A \circ B = AB = \{w : w = xy, x \in A, y \in B\}.$$

- $*$ Kleene star (unary operation)

$$A^* = \{w : w = X_1 X_2 \cdots X_k, k \geq 0, x_i \in A\}.$$

These are traditionally called the regular operations: they are in a sense minimal, because starting from a simple set of regular languages and applying these three operations we can get to all regular languages.

Example 1.5: If $A = \{\text{good}, \text{bad}\}$ and $B = \{\text{boy}, \text{girl}\}$ we get

$$A \circ B = \{\text{good boy}, \text{good girl}, \text{bad boy}, \text{bad girl}\}.$$

Note for $*$, we stick together words in any way we want to get longer string. We get an infinite language unless $A \subseteq \{\varepsilon\}$. Note $\varepsilon \in A^*$; in particular, $\phi^* = \{\varepsilon\}$.

Theorem 1.6: The collection of regular languages is closed under regular operations. In other words, if we take 2 regular languages (or 1 regular language, for $*$) and apply a regular operation, we get another regular language.

We say the integers are “closed” under multiplication and addition, but not “closed” under division, because if you divide one by another, you might not get an integer. Closed means “you can’t get out” by using the operation.

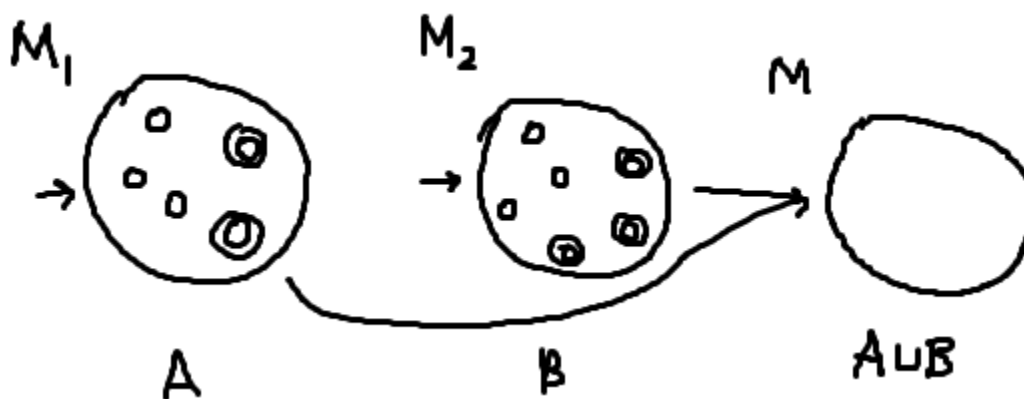
Proof of closure under \cup . We show that if A and B are regular, then so is $A \cup B$.

We have to show how to construct the automaton for the union language given the automata that recognize A and B , i.e. given


$$M_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1\} \text{ recognizing } A$$

$$M_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2\} \text{ recognizing } B$$

construct $M = (Q, \Sigma, \delta, q_0, F)$ recognizing $A \cup B$. (For simplicity, let $\Sigma_1 = \Sigma_2 = \Sigma$.)



You might think: run the string through M_1 , see whether M_1 accepts it, then run the string through M_2 and see whether M_2 accepts it. But you can’t try something on the whole input string, and try another thing on the whole input string! You get only 1 pass.

 Imagine yourself in the role of M .

The solution is to run both M_1 and M_2 at the same time. Imagine putting two fingers on the diagrams of the automata for M_1 and M_2 , and moving them around. At the end, if either finger is on an accept state, then we accept. This strategy we can implement in M . We now formalize this idea.

We should keep track of a state in M_1 and a state in M_2 as a single state in M . So each state in M corresponds to a pair of states, one in M_1 and M_2 ; let

$$Q = Q_1 \times Q_2 = \{(q, r) : q \in Q_1, r \in Q_2\}.$$

How to define δ ? When we get a new symbol coming in; we go to wherever q goes and wherever r goes, individually.

$$\delta((q, r), a) = (\delta_1(q, a), \delta_2(r, a)).$$

The start state is $q_0 = (q_1, q_2)$. The accept set is

$$F = (F_1 \times Q_2) \cup (Q_1 \times F_2).$$

(Note $F_1 \times F_2$ gives intersection.)

It is clear by induction that the k th state of M is just the k th state of M_1 and k th state of M_2 . \square

Problem 1.3: Prove that the collection of regular languages is closed under concatenation and Kleene star.

Note: The following is my solution. See the next lecture for an easier way to phrase it.

Proof of closure under \circ . To know whether a string w is in $A \circ B$, we think as follows: Suppose reading from the beginning of w we see a string in A , say $x_1 \cdots x_{a_1}$. In other words, we get to an accept state in Q_1 . Then maybe we have

$$\underbrace{x_1 \cdots x_{a_1}}_{\in A} \underbrace{x_{a_1+1} \cdots x_n}_{\in B}.$$

But maybe we should keep reading until next time we get to an accept state in Q_1 , say step a_2 , and

$$\underbrace{x_1 \cdots x_{a_2}}_{\in A} \underbrace{x_{a_2+1} \cdots x_n}_{\in B}.$$

But maybe we have

$$\underbrace{x_1 \cdots x_{a_3}}_{\in A} \underbrace{x_{a_3+1} \cdots x_n}_{\in B}!$$

So the possibilities “branch”—imagine putting one more finger on the diagram each time we get to an accept state; one finger then goes to Q_2 and the other stays at Q_1 . Our fingers will occupy a subset of the union $A \cup B$, so let

$$Q = 2^{Q_1 \cup Q_2}, \text{ the set of subsets of } Q_1 \cup Q_2.$$

Now define

$$\delta(S, a) = \begin{cases} \{\delta(s, a) : s \in S\}, & F_1 \cap S = \phi \\ \{\delta(s, a) : s \in S\} \cup \{\delta(q_2, a)\}, & F_1 \cap S \neq \phi. \end{cases}$$

The start state is $\{q_1\}$ and the accept set is

$$F = \{S \subseteq Q : F_2 \cap S \neq \phi\},$$

i.e. the set of subsets that contain at least one element of F_2 . Details of checking this works left to you! \square

Note this solution involves “keeping track of multiple possibilities.” We’ll need to do this often, so we’ll develop some machinery—namely, a type of finite automaton that can keep track of multiple possibilities—that simplifies the writing of these proofs.

Lecture 2

Tue. 9/11/12

The first problem set is out. Turn in the homework in 2-285. About homeworks: The optional problems are only for A+’s; we count how many optional problems you solved correctly.

Look at the homework before the day before it’s due! The problems aren’t tedious lemmas that Sipser doesn’t want to do in lectures. He chose them for creativity, the “aha” moment. They encourage you to play with examples, and don’t have overly long writeups. Write each problem on a separate sheet, and turn them in in separate boxes in 2-285.

Last time we talked about

- finite automata
- regular languages
- regular operations, and
- closure under \cup .

Today we’ll talk about

- regular expressions,
- nondeterminism,
- closure under \circ and $*$, and
- $FA \rightarrow$ regular expressions.

§1 Regular expressions

Recall that the regular operations are \cup , \circ , and $*$.

Definition 2.1: A **regular expression** is an expression built up from members of Σ (the alphabet) and ϕ , ε using \cup , \circ , and $*$.

For example, if $\Sigma = \{a, b\}$, we can build up regular expressions such as

$$(a^* \cup ab) = (a^* \cup a \circ b).$$

Here we consider a as a single string of length 1, so a is shorthand for $\{a\}$. ε might also appear, so we might have something like $a^* \cup ab \cup \varepsilon$ (which is the same since $\varepsilon \in a^*$; the language that the expression describes is the same). We also write $L(a^* \cup ab \cup \varepsilon)$ to emphasize that the regular expression describes a *language*.

Regular expressions are often used in text editors in string matching.

Our goal for the next $1\frac{1}{2}$ lectures is to prove the following.

Theorem 2.2: thm:regex-FA Regular expressions and finite automata describe the same class of languages. In other words,

1. Every finite automaton can be converted to a regular expression which generates the same language and
2. every regular expression can be converted to finite automaton that recognizes the same language.

Even though these 2 methods of computation (regular expressions and finite automata) seem very different, they capture the same language! To prove this, we'll first have to develop some technology.

§2 Nondeterminism

First, let's think about how to prove the closure properties from last time. We showed that if A_1 and A_2 are regular, so is $A_1 \cup A_2$. To do this, given a machine M_1 recognizing A_1 and a machine M_2 recognizing A_2 , we built a machine M that recognizes $A_1 \cup A_2$ by simulating A_1 and A_2 in parallel.

Now let's prove closure under concatenation: If A_1 and A_2 are regular, then so is $A_1 A_2$.

We start off the same way. Suppose M_1 recognizes A_1 and M_2 recognizes A_2 ; we want to construct M recognizing $A_1 A_2$.

What does M need to do? Imagine a string w going into M ... Pretend like you are M ; you have to answer if w is in the concatenation $A_1 A_2$ or not, i.e. you have to determine if it is possible to cut w into 2 pieces, the first of which is in A_1 and the second of which is in A_2 .

$$W \begin{array}{c} \hline \in A_1 \quad | \quad \in A_2 \end{array}$$

Why don't we feed W into M_1 until we get to an accept state, and then transition control to M_2 by going to the start state of M_2 ?

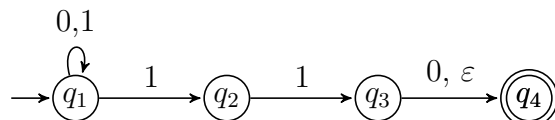
The problem with this approach is that just because you found an initial piece of W in A_1 does not necessarily mean you found the right place to cut W ! It's possible that the remainder is not in A_2 , and you wrongly reject the string. Maybe you should wait until later time to switch to A_2 . There are many possible ways of cutting.

$$W \xrightarrow{\quad \in A_1 \quad} \mid \xrightarrow{\quad \in A_2 \quad}$$

We introduce the idea of nondeterminism to give an elegant solution to this problem.

2.1 Nondeterministic Finite Automata

Consider, for example, the following automaton, which we'll call B .



How is this different from a finite automaton? Note that there are two “1” arrows from q_1 . In a **nondeterministic finite automaton** there may be several ways to proceed. The present state does NOT determine the next state; there are several possible futures. We also permit ε to be a label, as matter of convenience.

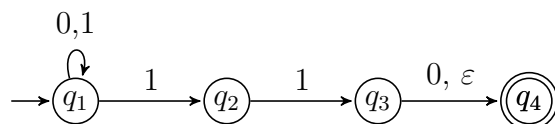
How does this automaton work?

We have multiple alternative computations on the input. When there is more than 1 possible way to proceed, we take all of them. Imagine a parallel computer following each of the paths independently. When the machine comes to point of nondeterminism, imagine it forking into multiple copies of itself, each going like a separate thread in a computer program.

An ε label means that you can take the transition for free. The other transitions also allow reading with 1 input symbol. (In some cases there is no arrow to follow. In those cases the thread just dies off.)

What do we do when parallel branches differ in their output? One choice might end up at q_4 , and another may end up not at q_4 . Only *one* path needs to lead to an accept state, for the entire machine to accept. If any computational branch leads to an accepting state, we say the machine accepts the input. Acceptance overrules rejection. We reject only if every possible way to proceed leads to rejection.

Although this seems more complicated than the finite automata we’ve studied, we’ll prove that it doesn’t give anything new. We’ll show that anything you can do with nondeterministic finite automata, you can also do with (deterministic) finite automata.



Let’s look at a specific example. Take 01011 as the input. Point your finger at the start state q_1 .

- Read 0. We follow the loop back to q_1 .

- Read 1. There are 2 arrows with “1” starting at q_1 , so split your finger into 2 fingers, to represent the 2 different places machine could be: q_1 and q_2 .
- 0. Now each finger proceeds independently, because they represent different threads of computation. The finger at q_1 goes back to q_1 . There is no place for the finger at q_2 to go (because there is no arrow with 0 from q_2), so remove that finger. We just have $\{q_1\}$ left.
- 1. We branch into q_1, q_2 .
- 1. Following “1” arrows from q_1 and q_2 , we can get to q_1, q_2, q_3 . But note there is an ε transition from q_3 to q_4 . This means we can take that transition for free. From a finger being on q_3 , a new thread gets opened on to q_4 . We end up with all states q_1, q_2, q_3 , and q_4 .

Each finger represents a different thread of the computation. Overall the machine accepts because at least 1 finger (thread of computation) ended up at an accepting state, q_4 . The NFA accepts this string, i.e. $01011 \in L(B)$. By contrast $0101 \notin L(B)$, because at this point we only have fingers on q_1, q_2 ; all possibilities are reject states.

We now make a formal definition.

Definition 2.3: Define a **nondeterministic finite automaton (NFA)** $M = (Q, \Sigma, \delta, q_0, F)$ as follows. Q , Σ , q_0 , and F are the same as in a finite automaton. Here

$$\delta : Q \times \Sigma_\varepsilon \rightarrow \mathcal{P}(Q),$$

where $\mathcal{P}(Q) = \{R : R \subseteq Q\}$ is the power set of Q , the collection of subsets of Q (all the different states you can get to from the input symbol.) and $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.

In our example, $\delta(q_1, 1) = \{q_1, q_2\}$ and $\delta(q_3, \varepsilon) = \{q_4\}$. Note δ may give you back the empty set, $\delta(q_2, 0) = \phi$.

The only thing that has a different form from a finite automaton is the transition function δ . δ might give you back several states, i.e. whole *set* of states.

2.2 Comparing NFA's with DFA's

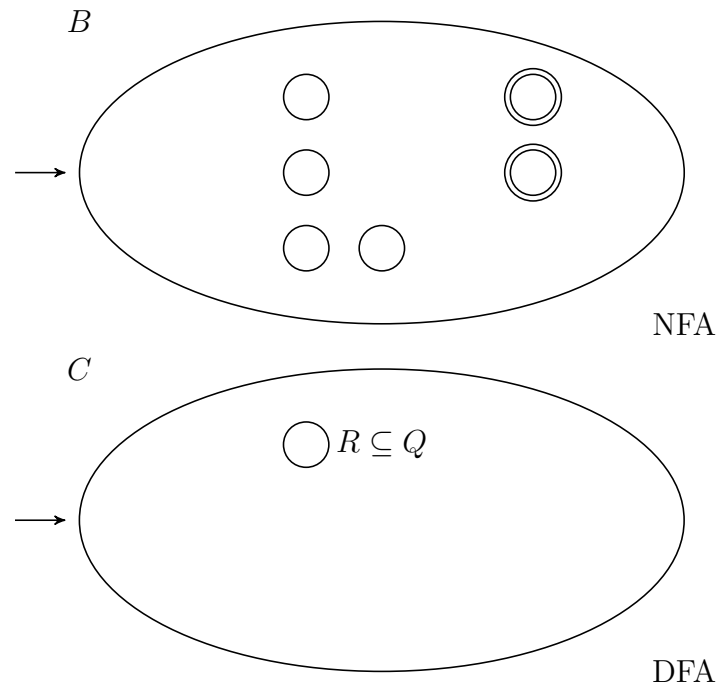
We now show that any language recognized by a NFA is also recognized by a DFA (deterministic finite automaton), i.e. is regular. This means they recognize the same class of languages.

Theorem 2.4 (NFA's and DFA's recognize the same languages): If $A = L(B)$ for a NFA B , then A is regular.

Proof. The idea is to convert a NFA B to DFA C .

Pretend to be a DFA. How would we simulate a NFA? In the NFA B we put our fingers on some collection of states. Each possibility corresponds not to a single state, but to a *subset* of states of B .

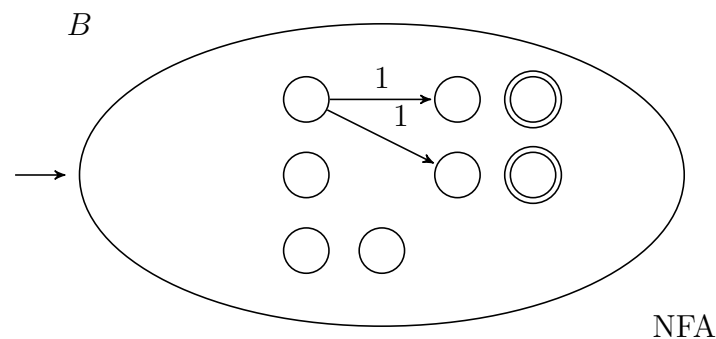
What should the states of C be? *The states of C should be the power set of B , i.e. the set of subsets of B .* In other words, each state of C corresponds to some $R \subseteq Q$.

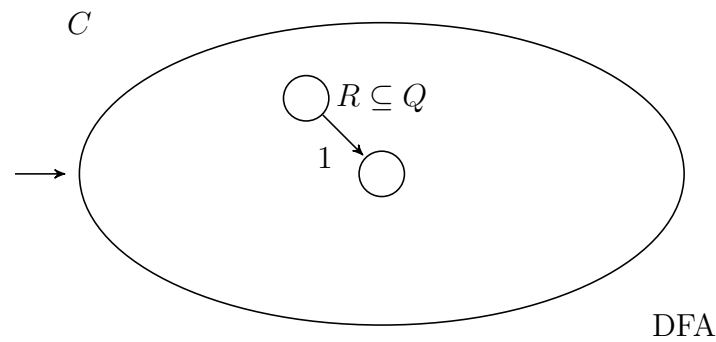


Let $B = (Q, \Sigma, \delta, q_0, F)$; we need to define $C = (Q', \Sigma, \delta', q'_0, F')$. Let $Q' = \mathcal{P}(Q)$ (the power set of Q), so that if B has n states, then C has 2^n states. For $R \subseteq Q$ (i.e. $R \in Q'$), define

$$\delta'(R, a) = \{q \in Q : q \in \delta(r, a), r \in R \text{ or following } \varepsilon\text{-arrows from } q \in \delta(r, a)\}.$$

(The textbook says it more precisely.)





The start state of C is a singleton set consisting of just the state and anything you can get to by ε -transitions. The accept states are the subsets containing at least one accept state in B . \square

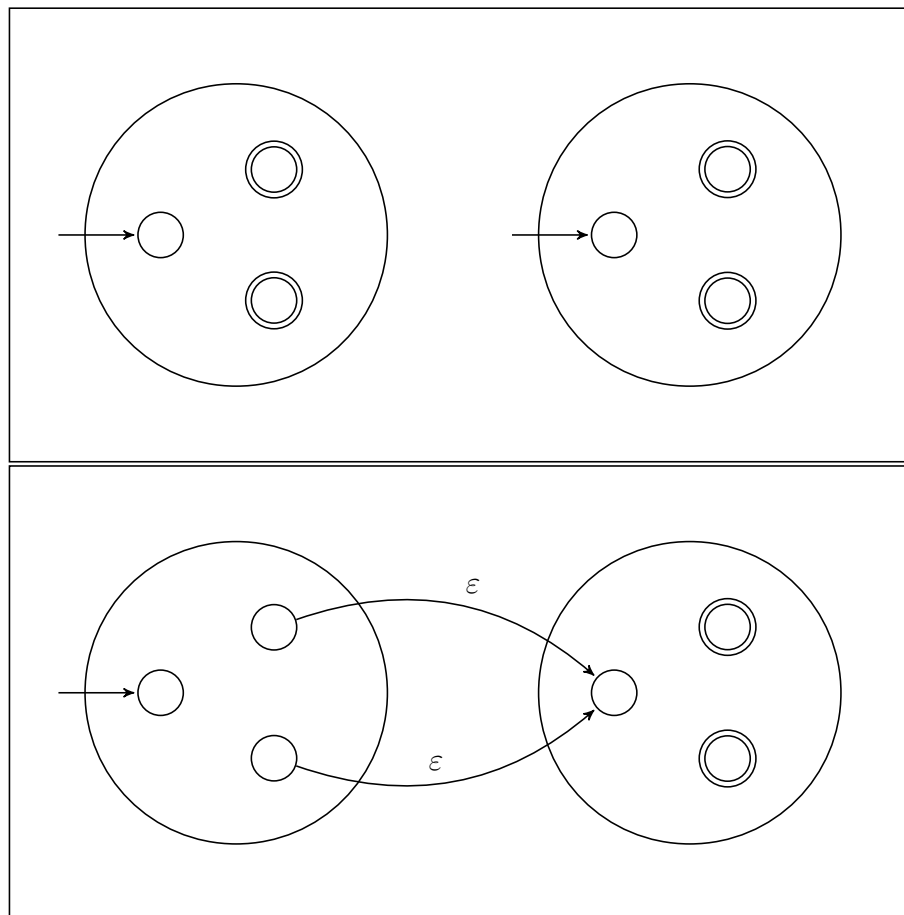
Key NFA's and DFA's describe same class of languages. Thus to show a language is a regular language, you can just build a NFA that recognizes it, rather than a DFA.

Many times it is more convenient to build a NFA rather than a DFA, especially if you want to keep track of multiple possibilities.

§3 Using nondeterminism to show closure

Nondeterminism is exactly what we need to show that the concatenation of two regular languages is regular. As we said, maybe we don't want to exit the first machine the first time we get to an accept state; maybe we want to stay in M_1 and jump later. We want *multiple possibilities*.

Proof of closure under \circ . Given M_1 recognizing A_1 and M_2 recognizing A_2 , define M as follows. Put the two machines M_1 and M_2 together. Every time you enter an accept state in M_1 , you are allowed to branch by an ε -transition to the start state of M_2 —this represents the fact that you can either start looking for a word in A_2 , or continue looking for a word in M_1 . Now eliminate the accepting states for M_2 . We're done!



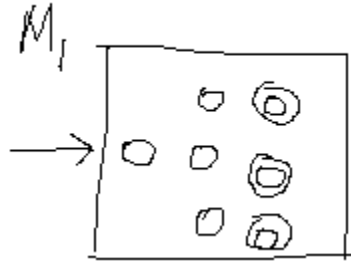
□

Nondeterminism keeps track of parallelism of possibilities. Maybe you got to an accepting state but you should have waited until a subsequent state. We have a thread for every possible place to transition from A_1 to A_2 ; we're basically trying all possible break points in parallel.

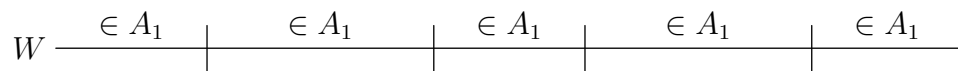
Another way to think of NFA's is that they enable "guessing." Our new machine M simulates M_1 until it guesses that it found the right transition point. We "guess" this is the right place to jump to M_2 . This is just another way of saying we make a different thread. We're not sure which is right thread, so we make a guess. We accept if there is at least one correct guess.

Next we show that if A_1 is regular, then so is A_1^* .

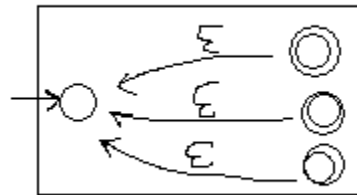
Proof of closure under $$.* Suppose M_1 recognizes A_1 . We construct M recognizing A_1^* . We will do a proof by picture.



What does it mean for a word W to be in A_1^* ? W is in A_1^* if we can break it up into pieces that are in the original language A_1 .



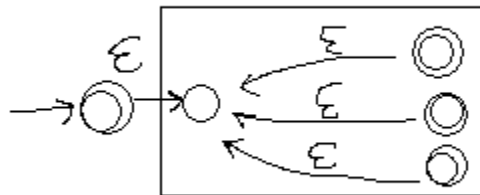
Every time we get to an accept state of M_1 , i.e. we've read a word in A_1 and we *might* want to start over. So we put ε -transition leading from the accept state to the start state.



As in the case with concatenation, we may not want to reset at the first cut point, because maybe there is no way to cut remaining piece into words in A_1 . So every time get to an accept, have the *choice* to restart—we split into 2 threads, one that looks to continue the current word, and one that restarts.

There is a slight problem: we need to accept the empty string as well.

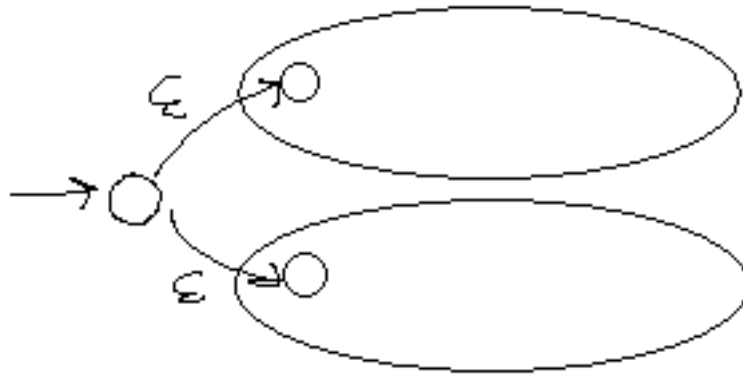
To do this we add a new start state, and add an ε -transition to the old start state. Then we're good.



□

NFA's also give us an easier way to prove closure under union.

Proof of closure under \cup . Suppose we're given M_1 recognizing A_1 and M_2 recognizing A_2 . To build M recognizing A_1 and A_2 , it needs to go through M_1 and M_2 in parallel. So we put the two machines together, add a new start state, and have it branch by ε -transitions to the start states *both* M_1 and M_2 . This way we'll have a finger in M_1 and a finger in M_2 at the same time.

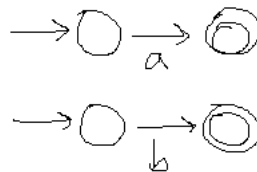


□

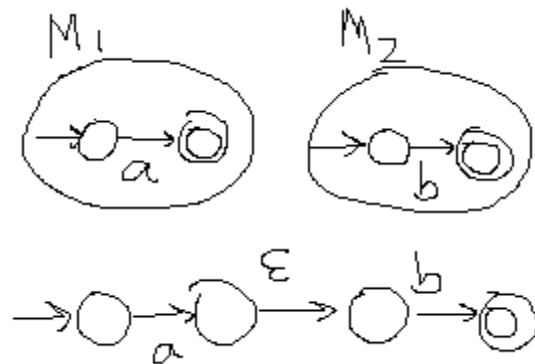
§4 Converting a finite automaton into a regular expression

The proof of the closure properties gives us a procedure for converting a regular expression into finite automaton. This procedure comes right out of the construction of machines for \cup , \circ , and $*$. This will prove part 2 of Theorem 2.2.

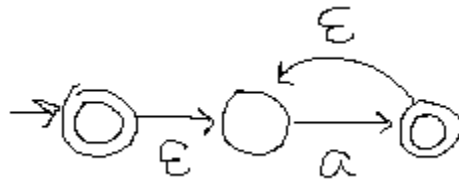
We do a proof by example: consider $(ab \cup a^*)$. We convert this to a finite automaton as follows. For a, b we make the following automata.



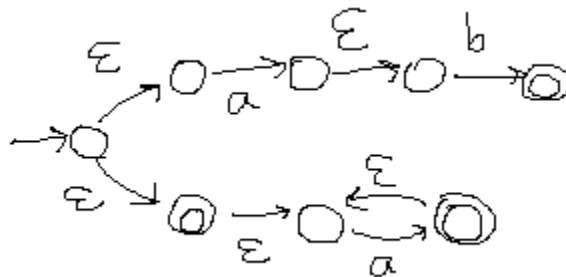
We build up our expression from small pieces and then combine. Let's make an automaton for ab . We use our construction for closure under concatenation.



This machine recognizes ab . Now we do a^* .



Finally we put the FA's for ab and a^* together, using the \cup construction, to get the FA recognizing $ab \cup a^*$.



Key The constructions for \cup , \circ , and $*$ give a way to construct a FA for any regular expression.

Lecture 3

Thu. 9/13/12

Last time we talked about

- nondeterminism and NFA's

- NFA \rightarrow DFA
- Regular expression \rightarrow NFA

Today we'll talk about

- DFA \rightarrow regular expression
- Non-regular languages

About the homework: By the end of today, you should have everything you need to solve all the homework problems except problem 6. Problem 3 (1.45) has a 1 line answer. As a hint, it's easier to show there *exists* a finite automaton; you don't have to give a procedure to construct it.

We will finish our discussion of finite automata today. We introduced deterministic and nondeterministic automata. Nondeterminism is a theme throughout the course, so get used to it.

We gave a procedure—the subset construction—to convert NFA to DFA. NFA helped achieve part of our goal to show regular expressions and NFAs recognize the same languages. We showed how to convert regular expressions to NFA, and NFA can be converted to DFA. To convert regular expressions, we used the constructions for closure under \cup , \circ , and $*$; we start with the atoms of the expression, and build up using more and more complex subexpressions, until we get the language recognized by the whole expression. This is a recursive construction, i.e. a proof by induction, a proof that calls on itself on smaller values.

Today we'll do the reverse, showing how to convert a DFA to a regular expressions, finishing our goal.

§1 Converting a DFA to a regular expression

Theorem 3.1 (Theorem 2.2, again): A is a regular language iff $A = L(r)$ for some regular expression r .

Proof. \Leftarrow : Show how to convert r to an equivalent NFA. We did this last time.

\Rightarrow : We need to convert a DFA to an equivalent r . This is harder, and will be the focus of this section. \square

We'll digress and introduce another model of an automaton, which is useful just for the purposes of this proof.

A **generalized nondeterministic finite automaton (GNFA)** has states, some accepting, one of which is starting. We have transitions as well. What's different is that we can write not just members of the alphabet and the empty string but any regular expression as a label for a transition. So for instance we could write ab .



Start at the start state. During a transition, the machine gets to read an entire chunk of the input in a single step, provided that the string is in the language described by the label on the associated transition.

There may be several ways to process the input string. The machine accepts if some possibility ends up at an accept state, i.e. there is some way to cut and read the input string. If all paths fail then the machine rejects the input.

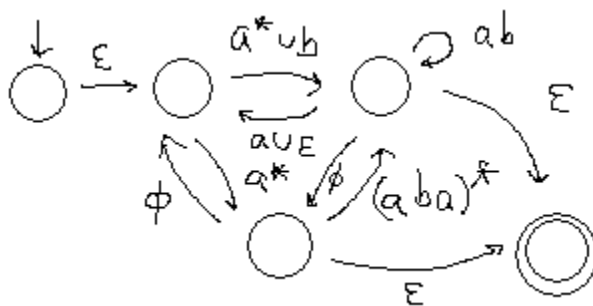
Although GNFA's look more complicated, they *still recognize the same languages as DFA's!*

It looks harder to convert a GNFA to a regular expression, $\text{GNFA} \rightarrow r$. However, for inductive proofs, it is often helpful to prove something stronger along the way, so we can carry through the statement. In other words, we *strengthen* the induction hypothesis.

To make life easier, we make a few assumptions about the GNFA.

- First, there is only 1 accept state. To achieve this, we can declassify accept states, and add empty transitions to new accept states.
- The accept state and start states are different (taken care of by 1st bullet).
- No incoming transitions come to the start state. To achieve this, make a new start state with an ε -transition going to the previous start state.
- There are only transitions to, not from, the accept state (taken care of by 1st bullet).
- Add all possible transitions between states except the start and end states. If we are lacking a transition, add ϕ transition. We can go along this transition by reading a language described by ϕ . This means we can never go along this transition, since ϕ describes no languages.

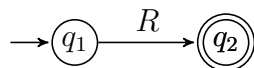
For instance, we can modify our example to satisfy these conditions as follows.



Lemma 3.2: For every $k \geq 2$, every GNFA with k states has an equivalent regular expression R .

Proof. We induct on k .

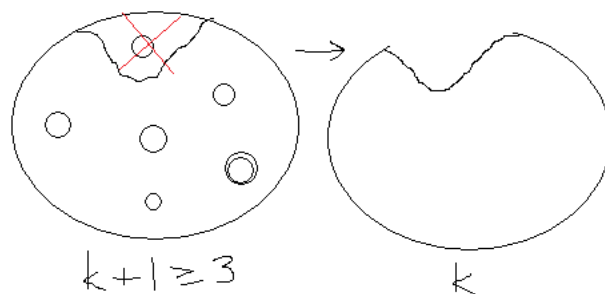
The base case is $k = 2$. We know what the states are: the machine has a start state (no incoming arrows) and an accept state. Assuming the conditions above, the only possible arrow is from the start to end, so the machine looks like the following. There are no return arrows or self-loops.



The only way to accept is to read a string in R ; the machine can only process input in its entirety with one bite, so the language is just the regular expression R . This is the easy part.

Now for the induction step. Assume the lemma true for k ; we prove it for $k + 1$. Suppose we're given a $(k + 1)$ -state GNFA. We need to show this has a corresponding regular expression. We know how to convert k -state GNFA to a regular expression. Thus, *if we can convert the $(k + 1)$ -state to a k -state GNFA, then we're done*. You can think of this as an iterative process: convert $(k + 1)$ to k to $k - 1$ states and so on, wiping out state after state, and keeping the language the same, until we get to just 2 states, where we can read off the regular expression from the single arrow.

We'll pick one state x (that is not the start or accept state) and remove it. Since $k + 1 \geq 3$, there is a state other than the start and accept state. But now the machine doesn't recognize the same language anymore. We broke the machine!

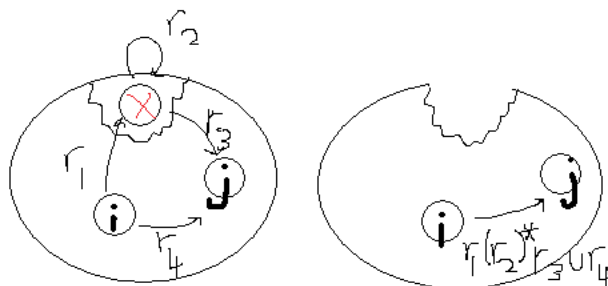


We have to repair the machine, by putting back the computation paths that got lost by removing the state.

This is where the magic of regular expressions come in.

Suppose we have arrows $i \rightarrow x \rightarrow j$. We can't follow this path because x is gone. In the arrow from i to j , we have to put back the strings that got lost. So if we have $i \xrightarrow{r_1} x \xrightarrow{r_3} j$, then we add in $r_1 r_3$ from i to j , so we can go directly from i to j via $r_1 r_3$. However, letting the self-loop at x be r_2 , we might go along r_1 , repeat r_2 for a while, and then go to r_3 ,

we so actually want $r_1(r_2^*)r_3$. Now take the union with the regular expression from i to j , $r_1(r_2^*)r_3 \cup r_4$.



So the construction is as follows. For each pair $i \xrightarrow{r_4} j$, replace r_4 with

$$r_1(r_2)^*r_3 \cup r_4$$

where r_1, r_2, r_3 are as above. All arrows adjusted in the same way. The computations that go from i to j via x in the old machine are still present in the new machine, and go directly from i to j .

Our modified machine is equivalent to the original machine. Taking any computation in first machine, there is a corresponding computation in second machine on the same input string, and vice versa. This finishes the proof. \square

Theorem 2.2 now follows, since a DFA is a GNFA.

§2 Non-regular languages

There are lots of languages that are not recognized by any finite automata. We see how to prove a specific language is non-regular.

Let

$$C = \{w : w \text{ has equal number of 0s and 1s}\}.$$

As we've said, it seems like C is not regular because it has to keep track of the difference between the number of 0s and 1s, and that would require infinitely many states. But be careful when you claim a machine can't do something—maybe the machine just can't do it the following the method you came up with!

! “I can't think of a way; if I try come up with one I fail” doesn't hold water as a proof!

As an example, consider

$$B = \{w : w \text{ has equal number of 01 and 10 substrings}\}.$$

For example $1010 \notin B$, but $101101 \in B$. This language may look nonregular because it looks like we have to count. But it is regular, because there is an alternative way to describe it that avoids counting.

Problem 3.1: Show that B is regular.

2.1 Pumping Lemma

We give a general method that works in large number of cases showing a language is not regular, called the Pumping Lemma. It is a formal method for proving nonregular not regular. Later on, we will see similar methods for proving that problems cannot be solved by other kinds of machines.

Lemma 3.3 (Pumping Lemma): **lem:pump** For any regular language A , there is a number p where if $s \in A$ and $|s| \geq p$ then $S = xyz$ where

1. $xy^iz \in A$ for any $i \geq 0$ (We can repeat the middle and stay in the language.)
2. $y \neq \varepsilon$ (Condition 1 is nontrivial.)
3. $|xy| \leq p$ (Useful for applications.)

What is this doing for us? The Pumping Lemma gives a property of regular languages. To show a language is not regular, we just need to show it doesn't have the property.

The property is that the language has a *pumping length*, or cutoff p . For any string s longer than the cutoff, we can repeat some middle piece (y^i) as much as we want and stay in the language. We call this *pumping up* s . Every long enough string in the regular language can be pumped up as much as we want and the string remains in the language.

Before we give a proof, let's see an example.

Example 3.4: Let

$$D = \{0^m 1^m : m \geq 0\}.$$

Show that D is not regular using the Pumping Lemma.



To show a language D is not regular, proceed by contradiction: If D is regular, then it must have the pumping property. Exhibit a string of D that cannot be pumped no matter how we cut it up. This shows D does not have the pumping property, so it can't be regular.

Assume D is regular. The pumping lemma gives a pumping length p . We find a string longer than p that can't be pumped: let $s = 0^p 1^p \in D$.

$$s = \frac{0 \cdots 0}{p} \mid \frac{1 \cdots 1}{p}$$

There must be some way to divide s into 3 pieces, so that if we repeat y we stay in the same language.

But we can't pump s no matter where y is. One of the following cases holds:

1. y is all 0's
2. y is all 1's
3. y has both 0's and 1's.

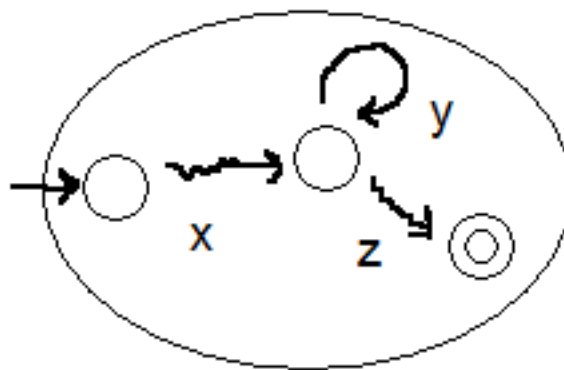
If y is all 0's, then repeating y gives too many 0's, and takes us out of the language. If y is all 1's, repeating gives too many 1's. If y has both 0's and 1's, they are out of order when we repeat. In each case, we are taken out of the language so pumping lemma fails, and D is not regular.

If we use condition 3 of the Pumping Lemma we get a simpler proof: xy is entirely in the first half of s , so y must be all 0's (case 1). Then xy^2z has excess 0's and so $xy^2z \notin D$.

Now we prove the Pumping Lemma.

Proof of Lemma 3.3. Let M be the DFA for A . Let p be the number of states of M . This will be our pumping length.

Suppose we have a string of length at least p . Something special has to happen when the machine reads the string: *We have to repeat a state!* We have to repeat a state within the first p steps (because after p steps we've made $p + 1$ visits to states, including the starting state). Consider the first repeated state, drawn in in the below diagram.



Divide the path into 3 parts: x , y , and z . Note we can choose y nonempty because we're saying the state is repeated. From this we see that we can repeat y as many times as we want. □

Example 3.5: Now we show

$$C = \{w : w \text{ has equal number of 0s and 1s}\}.$$

is not regular. There are two ways to proceed. One is to use the Pumping Lemma directly (this time we need to use condition 3) and the other way is to use the fact that we already know D is not regular.

What is wrong with the following proof? Because D is not regular and $D \subseteq C$, C is not regular.

! Regular languages can have nonregular languages as subsets, and vice versa. Subsets tell you nothing about regularity.

However, if we combine the fact that $D \subseteq C$ with some extra features of C , then we can come up with a proof. Note

$$D = C \cap 0^*1^*.$$

Note 0^*1^* is regular. If C were regular, then D would be regular, because the intersection of 2 regular languages is regular. Since D is not regular, neither is C .

key The Pumping Lemma is a powerful tool for showing languages are nonregular, especially when we combine it with the observation that regular languages are closed under regular operations.

Lecture 4

Tue. 9/18/12

Last time we talked about

- Regular expressions ← DFA
- Pumping lemma

Today we'll talk about CFG's, CFL's, and PDA's.

Homework 1 is due Thursday.

- Use separate sheets.
- No bibles, online solutions, etc.
- Office hours

- Michael Sipser: Monday 3-5
- Zack: Tuesday 4-6 32-6598
- Alex: Wednesday 2-4 32-6604

§0 Homework hints

Problem 2 (1.67, rotational closure):

If A is a language, $w = xy \in A$, then put $yx \in RC(A)$. Prove that if A is regular, then $RC(A)$ is also regular. If M is a finite automaton and $L(M) = A$, then you need to come up with a finite automaton that recognizes the rotational closure of A . The new automaton must be able to deal with inputs that look like yx .

Don't just try to twiddle M .

If you were pretending to be a finite automaton yourself, how you would go about determine if a string is in the rotational closure of the original language?

Recall, for yx to be in the rotational closure, the original automaton should accept xy . How would you run the original automaton to see whether the string is a rearranged input of something the original automaton would have accepted?

If only you could see x in advance, you would know what state you get to after running y ! Then you could start there, run y , then run x , and see if you get back where you started. But you have to pretend to be a finite automaton, so you can't see x first.

The magic of nondeterminism will be helpful here! You could guess all possible starting states, and see if any guess results in accept. "Guess and check" is a typical pattern in nondeterminism.

Problem 3 (1.45, A/B is regular, where A is regular and B is any): We get A/B as follows: start with A and remove all the endings that can be in B . In other words, A/B consists of all strings such that if you stick in some member of B , you get a member of A .

Note you don't necessarily have a finite automaton for B because B is not necessarily regular! This might be surprising. Think about how you would simulate a machine for A/B . If a string leads to one of the original accepting states, you might want accept it early. You don't want to see rest of string if the rest of the string is in B .

Looked at the right way, the solution is transparent and short.

Again, think of what you would do if you were given the input and wanted to test if it was in the language.

Problem 4 (1.46d): When you're using the pumping lemma, you have to be very careful. The language you're supposed to work with consists of strings wtw where $|w|, |t| \geq 1$. For example, 0001000 is in the language, because we can let

$$\underbrace{000}_w \underbrace{1}_t \underbrace{000}_w.$$

If we add another 0 to the front, it's tempting to say we're not out of the language. *But we're still in the language* because we can write

$$\underbrace{000}_w \underbrace{01}_t \underbrace{000}_w.$$

You don't get to say what w and t are. As long as there is some way of choosing w and t , it's in the language.

§1 Context-Free Grammars

We now talk about more powerful ways of describing languages than finite automata: context-free grammars and pushdown automata. Context free grammars and pushdown automata have practical applications: we can use them to design controllers, and we can use them to describe languages, both natural languages and programming languages.

1.1 Example

We introduce context-free grammars with an example.

A context-free grammar has variables, terminals, and rules (or predictions).

$$S \rightarrow OS1$$

$$S \rightarrow R$$

$$R \rightarrow \varepsilon$$

In the above, the three statements above are **rules**, R is a variable, and the 1 at the end of $OS1$ is a terminal. The symbols on the left hand side are **variables**. The symbols that only appear on the right hand side are called **terminals**.

We use a grammar to generate a language as follows. Start out with the symbol on the LHS of the topmost rule, S here. The rules represent possibilities for substitution. Look for a variable in our current expression that appears on the LHS of a rule, substitute it with the RHS. For instance, in the following we replace each bold string by the string that is in blue in the next step.

$$\begin{aligned} & \mathbf{S} \\ & \mathbf{0S1} \\ & 00\mathbf{S11} \\ & 00\mathbf{R11} \\ & 00\mathbf{\varepsilon}11 \\ & 0011. \end{aligned}$$

When we have a string with only terminal symbols, we declare that string to be in the language of G . So here

$$0011 \in L(G).$$

Problem 4.1: What is the language of G ?

We can repeat the first rule until we get tired, and then terminate by the 2nd and 3rd rules. We find that

$$L(G) = \{0^k 1^k : k \geq 0\}.$$

The typical shorthand combines all rules that have the same left hand side into a single line, using the symbol $|$ to mean “or.” So we can rewrite our example as

$$\begin{aligned} S &\rightarrow OS1|R \\ R &\rightarrow \varepsilon. \end{aligned}$$

Example 4.1: Define G_2 to be the grammar

$$\begin{aligned} E &\rightarrow E + T | T \\ T &\rightarrow T \times F | F \\ F &\rightarrow (E) | a. \end{aligned}$$

The variables are

$$(V) = \{E, T, F\};$$

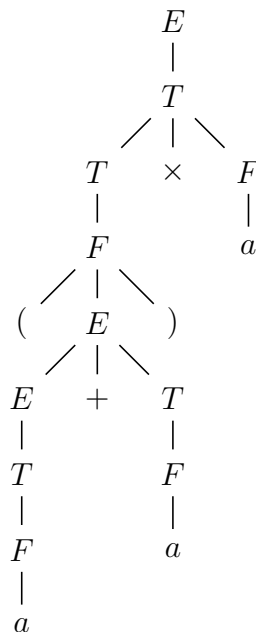
the terminals are

$$(\Sigma) = \{a, +, \times, (,)\}.$$

(We think of these as symbols.) This grammar represents arithmetical expressions in a using $+$, \times , and parentheses; for instance, $(a + a) \times a \in L(G)$.

This might appear as part of a larger grammar of a programming language.

Here is the parse tree for $(a + a) \times a$.



A derivation is a list of steps in linear form: When $U, V \in (V \cup \Sigma)^*$, we write $U \Rightarrow V$ if we get to v from u in one substitution. For instance we write $F \times F \Rightarrow (E) \times F$.

We write $u \xRightarrow{*} v$ if we can get from u to v in 0, 1, or more substitution steps.

1.2 Formal definition

We now give a formal definition, just like we did for a finite automaton.

Definition 4.2: A **context-free grammar (CFG)** is $G(V, \Sigma, S, R)$ where

- V is the set of **variables**,
- Σ is the set of **terminals**,
- $S \in V$ is the **start variable**, and
- R is the set of **rules**, in the form

variable \rightarrow string of variable and terminals.

We say S **derives** w if we can repeatedly make substitutions according to the rules to get from S to w . We write a derivation as

$$S \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_\ell \Rightarrow w, \quad S \xRightarrow{*} w.$$

(w only has terminals, but the other strings have variables too.) We say that G recognizes the language

$$L(G) = \{w \in \Sigma^* : S \xRightarrow{*} w\}.$$

There is a natural correspondence between a derivation and a parse tree. Parse tree may be more relevant to a particular applications.

Note $a + a \times a \in L(G_2)$. Take a look back at the parse tree for $a + a \times a$. Reading it from the bottom up, the parse tree first groups $a \times a$ into a subtree, and *then* puts in the \times . There is no way to put the $+$ first, unless we put in parentheses.

This is important in a programming language! Sometimes we can have multiple parse strings for the same string—an undesirable feature in general. That means we have two different interpretations for a particular string, that can give rise to two different semantic meanings. In a programming language, we do *not* want two different meanings for the same expression.

Definition 4.3: A string is derived **ambiguously** if it has two different parse trees. A grammar or language is **ambiguous** if some string can be derived ambiguously.

We won't discuss this further, but look at the section in the book for more.

1.3 Why care?

To describe a programming language in formal way, we can write it down in terms of a grammar. We can specify the whole syntax of the any programming language with context-free grammars. If we understand grammars well enough, we can generate a parser—the part of a compiler which will take the grammar representing the program, process a program, and group the pieces of code into recognizable expressions. The parser would then feed the expressions into another advice.

The key point is that we need to write down a grammar that represents the programming language.

Context-free grammars had their origin in the study of natural languages. For instance, S might represent sentence, and we may have rules such as

$$\begin{aligned} S &\rightarrow (\text{noun phrase}) (\text{verb phrase}) , \\ (\text{verb}) &\rightarrow (\text{adverb}) (\text{verb}) , \\ (\text{noun}) &\rightarrow (\text{adjective}) (\text{noun}) , \end{aligned}$$

and so forth. We can gain insight into the way a language works by specifying it this fashion.

This is a gross oversimplification, but both the study of programming and natural languages benefit from the study of grammars.

We're going to shift gears now, and then put everything together in the next lecture.

§2 Pushdown automata

Recall that we had 2 different ways for describing regular languages, using a

- *computational device*, a finite automaton, which recognize members of regular languages when it runs.
- *descriptive device*, a regular expression, which generates members of regular languages.

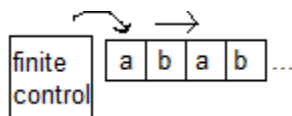
We found that finite automata and regular expressions recognize the same class of languages (Theorem 2.2). A CFG is a descriptive device, like a regular expression. We will find a computational device that recognizes the same languages as CFG's. First, a definition.

Definition 4.4: A **context-free language (CFL)** is one generated by a CFG.

We've already observed that there is a CFL that is not regular: we found a CFG generating the language $\{0^k 1^k\}$, which is not regular. We will show in fact that the CFL's include all regular languages. More on this later.

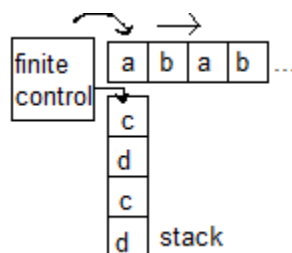
§3 Comparing pushdown and finite automata

We now introduce a computational device that recognizes exactly the context-free languages: a pushdown automaton (PDA). A pushdown automaton is like a finite automaton with a extra feature called a *stack*.



In a finite automaton, we have a finite control, i.e. different states with rules of how to transition between them. We draw a schematic version of a finite automaton, as above. A head starts at the beginning of the input string, and at each step, it moves to the next symbol to the right.

A pushdown automata has an extra feature. It is allowed to write symbols on the stack, not just read symbols.



However, there are some limitations. A pushdown automata can only look at the topmost symbol of a stack. When it writes a symbol to the stack, what's presently there gets pushed down, like a stack of plates in a cafeteria. When reading, the reverse happens. In one step the automata can only pop off the topmost symbol; then the remaining symbols all move back up. We use the following terminology:

- **push** means “add to stack,” and
- **pop** means “read and remove from stack.”

When we looked at FA's, we considered *deterministic* and *nondeterministic* variants. For PDA's, we'll only consider the nondeterministic variant. A deterministic version has been studied, but in the case of pushdown automata they are not equivalent. Some languages require nondeterministic PDA's.

Deterministic pushdown automata have practical applications to programming languages, because the process of testing whether a language is valid is especially efficient if the PDA is deterministic. This is covered in the 3rd edition of the textbook.

Let's give an example.

Example 4.5: ex:akbk We give a PDA for $A = \{0^k 1^k : k \geq 0\}$.

As we've said, a PDA is a device that looks like FA but also have stack can write on. Our PDA is supposed to test whether a string is in A .

If we used an ordinary FA, without a stack, then we're out of luck. Intuitively, a FA has finite memory, and we can't do this language with finite memory. The stack in a PDA, however, is just enough to allow us to “remember stuff.”

Problem 4.2: How would we design a PDA that recognizes A ? (How would you use the stack?)

We can use the stack to *record information*. The idea is that every time we read a 0, stick a 0 in; every time we read a 1, pop it out. If the stack becomes empty and has not become empty beforehand, then we accept. The 0's match off with 1's that come later.

We have to modify this idea a little bit, because *what if the 0's and 1's are out of order?* We don't want to accept strings where the 0's and 1's are out of order. If we insist that 0's come before 1's, we need a finite control mechanism.

We have a state for reading 0's and another state when reading 1's. In the "1" state the PDA no longer takes 0's and adds them to the stack. We see that a PDA combines the elements of FA with the power of a stack.

Now we ask: *how do we know when to transition from reading 0's to reading 1's?* We'd like to consider different possibilities for when to transition, i.e. let several parallel threads operate independently, and if any thread gets to an accept state, then have the machine accept the input. Hence we turn to nondeterminism: every time there's a choice, the machine splits into different machines which operate independently, each on its own stack.

At every step when the machine is reading 0's, we give it a nondeterministic choice: in the next step the machine can continue to push 0's on the stack, or transition into reading 1's and popping 0's off the stack.

3.1 Formal definition

Let's write down the formal definition.

Definition 4.6: A **pushdown automaton (PDA)** is a 6-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q are the states
- Σ is the input alphabet,
- Γ is the stack alphabet,
- δ is the transition function,
- q_0 is the start state,
- F is the accept states.

Here Q , Σ , q_0 , and F are as in a finite automata, but the transition function is a function $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$ (we explain this below).

On first thought, we may think to define the transition function as a function

$$\delta : Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma.$$

The function takes as input

- a state in Q —the current state of the machine,
- a symbol from Σ —the next symbol to read, and
- a symbol from Γ —the top-of-stack symbol.

It outputs a another state in Q to transition to, and a symbol from Γ —the next symbol to push on the stack.

However, we have to modify this: we want nondeterminism, so we allow the machine to transition to an entire *set* of possible next states and next symbols, and we represent this by having δ output a *subset*:

$$\delta : Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma).$$

We also allow δ to read an empty string, or read without popping a string on the stack, and proceed without writing a symbol, so we actually want

$$\delta : Q \times \underbrace{\Sigma_\varepsilon}_{\Sigma \cup \{\varepsilon\}} \times \underbrace{\Gamma_\varepsilon}_{\Gamma \cup \{\varepsilon\}} \rightarrow \mathcal{P}(Q \times \Gamma_\varepsilon).$$

We'll do one more example and save proofs to next time.

Example 4.7: Consider the language

$$\{ww^{\mathcal{R}} : w \in \{0, 1\}^*\},$$

where \mathcal{R} means “reverse the word.” This is the language of even-length palindromes such as 0110110110. A PDA recognizing this language uses nondeterminism in an essential way. We give a sketch of how to construct a PDA to recognize it. (See the book for details.)

The PDA has to answer: does the 2nd half of the word match the first half?

We should push the first half of the word on the stack. When we pop it off, the string comes out backwards, and we can match it with the second half of the word. This is exactly what we need.

But how do we know we're at the middle? When do you shift from pushing to popping and matching?

Can we find the length of the word? No. Instead, we guess nondeterministically at every point that we're at the middle! If the word is a palindrome, one of the threads will guess correctly the middle, and the machine will accept.

Lecture 5

Thu. 9/20/12

Problem set 2 is out.

Last time we talked about CFG's, CFL's, and PDA's. Today we will talk about

- CFG→PDA,
- non-CFL's
- Turing machines

Recall what nondeterminism means: every time there are multiple possibilities, the whole machine splits into independent parts. As long as one thread lands in an accept state, then we accept. Nondeterminism is a kind of guessing and checking that the guess is correct.

When we define the model for a PDA, the PDA can pop something from the stack. There is no hardware (built-in function) to test if the stack is empty, but we can use “software” (i.e. clever programming) to test if the stack is empty: to start off, write a \$, and when the machine sees \$, it knows that the stack is empty. Thus we can allow any PDA to test whether the stack is empty. We'll use this in many of our examples.

To jog your memory, a CFG is made up of a set of rules like the following:

$$\begin{aligned} E &\rightarrow E + T | T \\ T &\rightarrow T \times F | F \\ T &\rightarrow (E) | a. \end{aligned}$$

We saw that this CFG recognizes $a \times a + a$: we had a derivation of $a \times a \rightarrow a$ given by

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow T + F \Rightarrow \dots \Rightarrow a \times a + a.$$

§1 CFG's and PDA's recognize the same language

Our main theorem of today is the following.

Theorem 5.1: A is a CFL iff some PDA recognizes A .

In other words, CFG's and PDA's have exactly the same computing power; they generate the same class of languages. To prove this we'll simulate one kind of computation with another kind of computation.

Proof. We need to prove

1. CFG→PDA (we'll do this today)
2. PDA→CFG (skip, see the book. This is more technical.)

□

Corollary 5.2: 1. Every regular language is a CFL.

2. The intersection of a context-free language and a regular language is a context-free language.

$$\text{CFL} \cap \text{regular} = \text{CFL}.$$

Proof. 1. A finite automaton is a pushdown automaton that just doesn't use the stack.

2. Omitted. (See Exercise 2.18a in the book—basically just take the states to be the product set.)

□

Note 2 is weaker than the statement that the intersection of two context-free languages is a context-free language, which is not true.

Proposition 5.3: The intersection of two CFL's, or the complement of a CFL, is closed under $\cup, \circ, *$, but not under \cap or complementation.

Proof. Just give a construction using grammars or pushdown automata.

□

§2 Converting CFG \rightarrow PDA

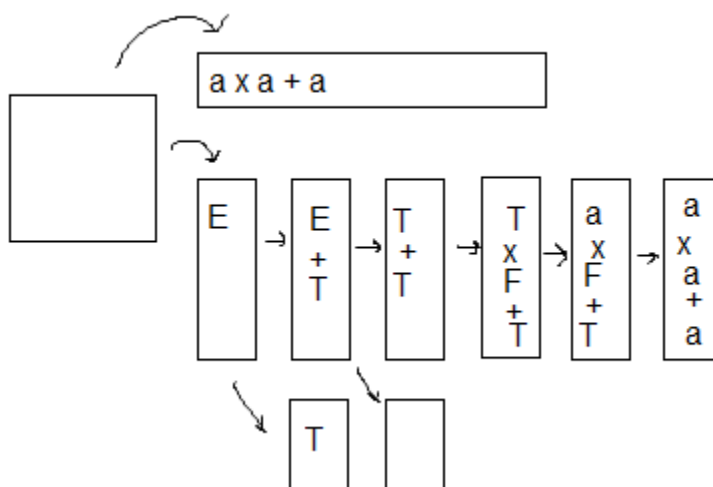
Proof sketch. We convert a CFG into a PDA.

The input is a string that may or may not be in the language; our PDA has to say whether it is.

Recall that the derivation is the sequence of strings we go through to get to a string in the language. *We use nondeterminism to guess the derivation.*

We first write down the start variable on the top of the stack. Take whatever string is written down on the stack to be the current working string. Take one of the variables on the stack and in the next step, replace it using one of the rules. There may be several possible steps; we consider all possibilities using nondeterminism.

For instance, we'd want the machine to operate as follows.



At the end we pop the symbols and compare with the input string, and then test the stack for emptiness at the end.

However, there's a problem: what if we want to replace some symbol *not* at the top? The idea is the following: if the top of the stack has a terminal symbol (which can't be replaced by anything), let's match it against the next symbol in the input word immediately. Whenever we have a terminal symbol at the top of the stack, we pop and compare until a variable (such as a F) is at the top. Sooner or later, we'll have a variable at the top, and then we can try one of the substitutions.

See the textbook for details. □

§3 Non-CFLs

CFG's are powerful but there are still many languages they can't recognize!

We will show the language

$$\{a^k b^k c^k : k \geq 0\}$$

is not a CFL. Note by contrast that $\{a^k b^k : k \geq 0\}$ is a CFL (Example 4.5).

An intuitive argument is the following: we can push the a 's, compare with the b 's by popping the a 's, but when we get to the c 's we're out of luck: the a 's were all popped off, and the system has not remembered any information about the a 's. However, as we've said, we have to be careful with any argument that says "I can't think of a way; thus it can't be done." How do you know the machine doesn't proceed in some other tricky way?

By contrast, if we look at the strings $a^k b^l c^m$ where either the number of a 's equal the number of b 's, *or* the number of a 's equal the number of c 's, this can be done with pushdown automaton. (Use nondeterminism, left as exercise.)

We'll give a technique to show that a language is not a CFL, a pumping lemma in the spirit of the pumping lemma for regular languages, changed to make it apply to CFL's.

Our notion of pumping is different. It is the same general notion: all long strings can be "pumped" up and stay in the language. However, we'll have to cut our string into 5 rather than 3 parts.

Lemma 5.4 (Pumping lemma for CFL's): **lem:pump-cfl** For a CFL A , there is a pumping length p where if $s \in A$ and $|s| \geq p$, then s can be broken up into $s = uvxyz$ such that

1. $uv^i xy^i z \in A$ for $i \geq 0$. (We have to pump v and y by the same amount.) The picture is as follows.

$$S = \overline{\quad} \begin{array}{c} | \\ u \end{array} \overline{\quad} \begin{array}{c} | \\ v \end{array} \overline{\quad} \begin{array}{c} | \\ x \end{array} \overline{\quad} \begin{array}{c} | \\ y \end{array} \overline{\quad} \begin{array}{c} | \\ z \end{array}$$

2. $vy \neq \varepsilon$. (We can't break it up so that the second and fourth string are empty, because in this case we won't be saying anything!)
3. $|vxy| \leq p$

Example 5.5: Let's show $\{a^k b^k : k \geq 0\}$ satisfies the pumping lemma. For instance, we can let

$$\underbrace{aaaa}_u \underbrace{a}_v \underbrace{ab}_x \underbrace{b}_y \underbrace{bbbb}_z.$$

Example 5.6: If $\{a^k b^k c^k : k \geq 0\}$ were a CFL, it would be satisfy the pumping lemma. We show this is not true, so it is not a CFL. Again, this is a proof by contradiction.

Suppose $\{a^k b^k c^k : k \geq 0\}$ satisfies the conclusion of the pumping lemma. Take the string

$$s = \underbrace{a \cdots a}_p \underbrace{b \cdots b}_p \underbrace{c \cdots c}_p = a^p b^p c^p$$

and let u, v, x, y, z satisfy the conclusions of the pumping lemma. First note that v can only have one kind of symbol, otherwise when we pump we would have letters out of order (instead of all a 's before b 's and all b 's before c 's), and the same is true of y . Thus when we pump up v and y , the count of at most 2 symbols will increase (rather than all 3 symbols), and we will not have an equal number of a 's, b 's, and c 's.

Thus $\{a^k b^k c^k : k \geq 0\}$ fails the pumping lemma, and hence is not context-free.

Proof of Pumping Lemma 5.4. We'll sketch the higher-level idea.

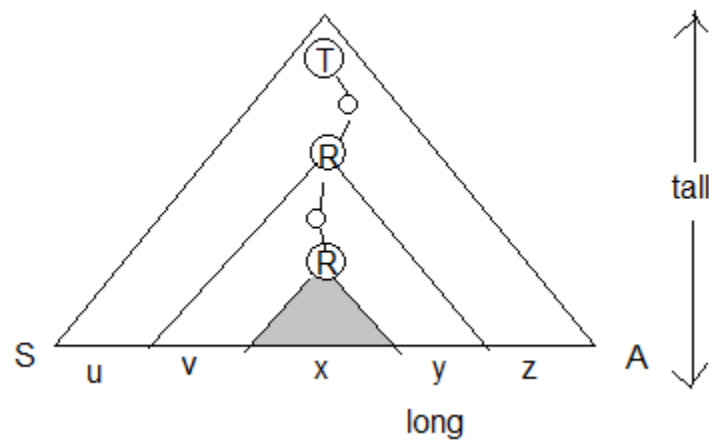
Qualitatively, the pumping lemma says that every long enough string can be pumped and stay in the language.

Let s be a *really* long string. We'll figure out what "really long" means later.

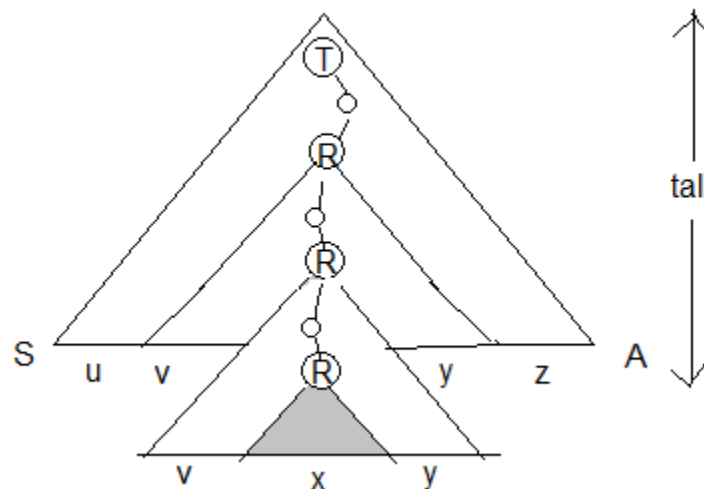
Let's look at the parse tree; suppose T is the start variable.

What do we know about the parse tree? It's really tall, because s is long, and a short parse tree generate can't generate a really wide tree (which corresponds to a long string). More precisely, the amount of "fan-out" is determined by the size of the longest right-hand string in the grammar. We determine what "long" and "tall" means after we look at the grammar.

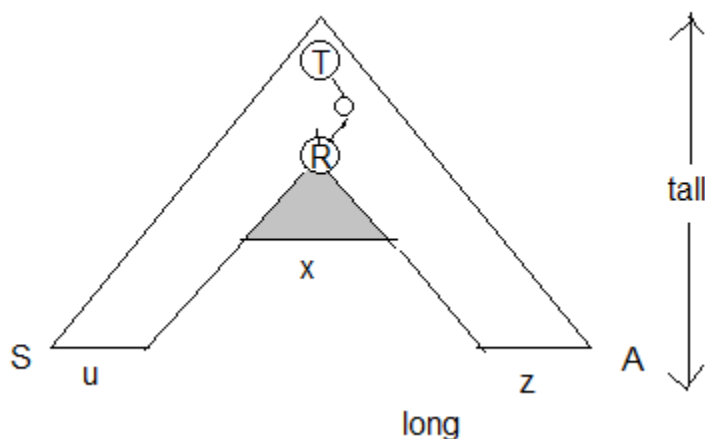
What does it mean when we have a really tall parse tree? If we take a path, then there has to be a long path, with lots of nodes—so many nodes that we have to repeat one of the variables, say R . Let u, v, x, y, z be as follows.



Look at the subtree that comes from the lower and upper instances of the repeated variable. Now let's make a "surgery": take the subtree under the higher R and stick it in place of the lower subtree. We now get another valid parse tree for $uvvxyz$. We can repeat this as many times as we'd like.



We get the $i = 0$ case by sticking the lower tree on the upper R .



There are some details to get the conditions to work.

- How do we know that v and y are not both empty? If they are, we've shown nothing. Let's start off with the parse tree with the *fewest* nodes. If v and y were both empty, then when we stick the lower R -subtree higher up as in the last picture above, we get fewer nodes, contradicting our minimality assumption. Hence v and y can't both be empty; this gives condition 2.
- Let's figure out p . Let b be the size of the largest right-hand side of a rule. We want the tallness to be at least $|V| + 1$ ($|V|$ is the number of variables.) At each level, the number of nodes multiplies by at most b . If we set $p = b^{|V|+1}$, then the tree would have at least $|V| + 1$ levels, so one of the symbols would repeat, as needed.
- To satisfy item 3 we take the lowest repetition of a symbol, so that there can be no repetitions below. This will give the bound $|vxy| \leq p$.

□

§4 Turing machines

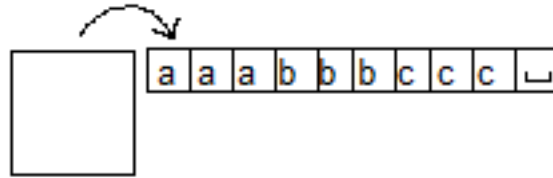
Everything we've done so far is a warm-up. We've given two models of computations that are deficient in a certain sense because they can't even do what we think computers can do, such as test whether a string is of the form $a^k b^k c^k$.

A Turing machine is vastly more powerful; it is a much closer model to what we think about when we think about a general-purpose computer.

The input tape of a Turing machine combines the features of both the input and stack. It is a place where we can both read and write.

- We can read and write on the tape.

This is the key difference. The following are other differences.

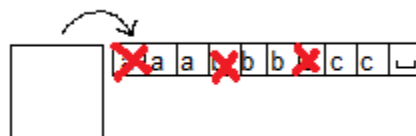


- We are able to both move the tape forward and back, so we can read what we wrote before. (It's a two way head.)
- The tape is infinite to the right. At the beginning, it is filled with a finite string, and the rest of the tape is filled with special symbols called blanks. The head starts on the leftmost tape cell.
- The machine accepts by entering an “accept” state anywhere. (It no longer makes sense to say the Turing machine accepts only at the end of the string—it might have erased or changed the last symbol!)
- There is a “reject” state; if the machine visits that state, stop and reject (reject by halting).
- A Turing machine can also reject by entering an infinite loop (“looping”).²

For the time being we'll just allow the deterministic variant.

Example 5.7: We outline how to build a Turing machine that recognizes $\{a^n b^n c^n\}$.

Let's assume we can test when we're at the beginning. We go through the string and cross out the first a , b , and c that appear.



If we find letters that are out of order, we reject. Otherwise we go back to the beginning and continue to cross off symbols a , b , and c one at a time. If we cross out the last a , b , and c on the same run, then accept.

When we cross a symbol off, write the symbol x to remember that we crossed out something there.

We'll write down the formal definition next time. Our transition function will depend on both the state and tape symbol.

²How does the Turing machine know it has entered an infinite loop? Mathematically being able to define when the machine rejects is different from what we can tell from the machine's operation. We'll talk more about this later.

Lecture 6

Tue. 9/25/12

Last time we talked about

- $\text{CFG} \leftrightarrow \text{PDA}$ (we only proved \rightarrow)
- Pumping lemma for CFL's
- Turing machines

Turing machines are an important model for us because they capture what we think of when we think of a general-purpose computer, without constraints like only having a stack memory or a finite memory. Instead, it has an unlimited memory.

Today we'll talk about

- Turing machine variants
 - Multitape
 - Nondeterministic
 - Enumerators
- Church-Turing thesis

In the first half of today's lecture, we'll develop some intuition for Turing machines. We will prove that several variations of Turing machines are actually all equivalent. As we'll see, this has philosophically important implications.

§1 Turing machines

We now give a formal definition of a Turing machine.

Definition 6.1: A **Turing machine (TM)** is a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$

where

- Q is the set of states,
- Σ is the input alphabet,
- Γ is the tape alphabet,
- δ is a function $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$. Here L or R denote the movement of the head.

- q_0 is the start state,
- q_a are accept states, and
- q_r are reject states.

If the machine tries to move off the left extreme of the tape, the machine instead just stay where it is.³

A Turing machine may halt (accept or reject) or loop (reject). If the machine loops we say the machine rejects, but think of it as rejecting after “infinite time”; we don’t know at any finite time that the machine has rejected.

Definition 6.2: Let

$$L(M) = \{w : M \text{ on input } w \text{ accepts}\}.$$

If $A = L(M)$ for some Turing Machine M , we say that A is **Turing-recognizable** (also called **recursively enumerable**).

An important subclass of Turing Machines are those that always halt.

Definition 6.3: A TM M is a **decider** if M halts on every input. If $A = L(M)$ for some decider, we say A is **decidable**.

Turing Machines which reject by halting are more desirable than those that reject by looping.

We just introduced 2 new classes of languages, Turing-recognizable languages and decidable languages. We have

$$\text{CFL's} \subset \text{decidable} \subset \text{T-recognizable}$$

where the inclusions are proper (we’ll show the right-hand inclusion is proper). We need to show containment in the left-hand side and nonequality in the RHS.

1.1 A brief history of Turing machines

Why are Turing machines so important, and why do we use them as a model for a general-purpose computer?

The concept of a Turing machines dates back to the 1930’s. It was one of a number of different models of computation that tried to capture *effective computability*, or *algorithm*, as we would now say. Other researchers came up with other models to capture computation; for example, Alonzo Church developed lambda calculus.

³Other treatments do different things. However, minor details don’t make any difference. We get the same computing power, and the same class of languages is recognized. The model of Turing machine is robust; it’s not sensitive to little details.

It wasn't obvious that these different models are equivalent, i.e., that they captured the same class of computations. However, they did.

Nowadays we have programming languages. Can today's more "advanced" programming languages (Python) do more than a FORTRAN program? It has a lot of new features compared to old boring "do" loops. It is conceivable that as we add more constructs to a programming language, it becomes more powerful, in the sense of computing functions and recognizing languages.

However, anything you can do with one language you can do with another. (It might simply be easier to program in one than another, or one might run faster.) How can we show we can do the same thing with Python as FORTRAN? We can convert python programs into FORTRAN, or convert FORTRAN programs to python. We can simulate one language with the other. This "proves" that they have the same computational power.

That's what the researchers of computation theory did. They gave ways to simulate Turing machines by lambda calculus, lambda calculus by Turing machines, as well as different variations of these models.

They found that all these models were doing the same thing!

We'll see this for ourselves, as we show that several variations of Turing machines all have the same computational power.

1.2 Multitape Turing machines

Definition 6.4: A **multitape Turing machine** is a Turing machine with multiple tapes. The input is written on the first tape.

The transition function can not look at all the symbols under each of the heads, and write and move on each tape.

We can define all this rigorously, if we wanted to.

Theorem 6.5: thm:multitape A is Turing-recognizable iff some multitape TM recognizes A .

In other words, Turing-recognizability with respect to one-tape Turing machines is the same as Turing-recognizability with respect to multi-tape Turing machines.

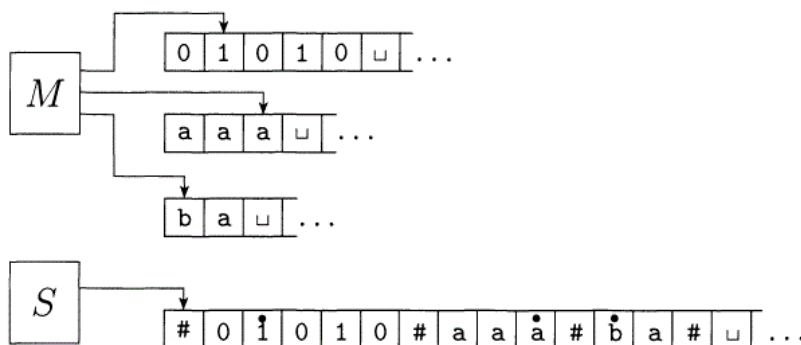
Proof. If A is Turing recognizable, then clearly a multitape TM recognizes A , because a single-tape TM is a multitape TM.

Suppose we have a language recognizable with a multitape TM. We need something like a compiler to convert a multitape TM to a one-tape TM, so that we can use a one-tape TM to simulate a multi-tape TM.

Let M be a multitape TM. M can do stuff with primitive operations that a single-tape TM S can't do. It can write to a 2nd tape, which S doesn't have! We need to use some data structure on S 's tape to represent what appears on the multitapes on M .

S initially formats its tape by writing separators $\#$ after the input string, one for each tape that M has. The string between two separators will represent the string on one of M 's tapes.

Next S moves into simulation phase. Every time M does one step, S simulates it with many steps. (This is just “programming,” in single-machine TM code.) S has to remember where the heads are on the multiple tapes of M . We enhance the alphabet on S to have symbols with dots on them \dot{a} to represent the positions of the heads. S update the locations of these markers to indicate the locations of the heads.



(Figure 3.14 from the textbook.)

There are details. For example, suppose M decides to move head to an initially blank part. S only has allocated finite memory to each tape! S has to go into an “interrupt” phase and move everything down one symbol, before carrying on. \square



A lot of models for computation turn out to be equivalent (especially variants of Turing machines). To show they are equivalent, give a way to simulate one model with the other.

The same proof carries through for deciders: A language is decidable by a multitape TM iff it is decidable by a single-tape TM.

Let's look at another example, similar but important for us down the road.

1.3 Nondeterministic TM

Definition 6.6: A **nondeterministic Turing machine** (NTM) is like a Turing machine except that the transition function now allows several possibilities at each step, i.e., it is a function

$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

If any thread of the computation accepts, then we say the Turing machine accepts. (Accepting overrules rejecting.)

We say that a nondeterministic TM is a decider if every branch halts on every input.

Theorem 6.7: thm:ntm A is Turing-recognizable iff some NTM recognizes A .

As we will see in the second half of the course, nondeterministic TM's are very important. For our purposes now, they have the same power as deterministic TM's, because they recognize the same class of languages.

Proof. Any deterministic Turing machine is a NTM, so this direction is obvious.

We want to convert a NTM N to a DTM M . M is supposed to accept exactly the same input N accepts, by simulating N . How does this work? This is trickier.

N may have made a nondeterministic move, resulting in 2 or more options. M doesn't know which to follow. If there are multiple ways to go, then take that piece of tape, make several copies of the tape separated by #, and carry on the simulation. This is just like the proof of Theorem 6.5, except that different segments of tape don't represent different tapes, they represent different threads.

We have to represent both the head and the state for each of the threads. The number of threads may grow in some unbounded way. M can't keep track of all the different states in finite memory, so we had better write them all down. To do this, allow a composite symbol

$\begin{smallmatrix} q \\ a \end{smallmatrix}$ to mean that the head is at a and the machine is in state q in that thread. M proceeds by taking a thread, seeing what N would do, and updating thread.

One of threads may again fork into multiple possibilities. In that case we have to open up room to write down copies of a thread, by moving stuff down.

M goes through each thread and repeats. The only thing we have to take note of is when should M end up accepting? If M enters an accept state on any thread, then M enters its accept state. If M notices some thread of N enters a reject state, then M collapse the thread down or marks it as rejecting, so it don't proceed with that further, and carries on with the other threads. \square

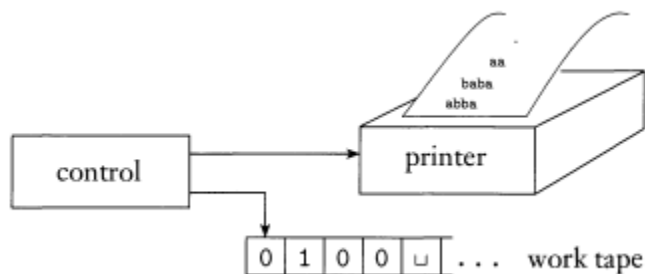
Question: When does nondeterminism help in a model of computation? In the second half of the course, when we care about how much *time* computation takes, the big question is whether NTM and TM are equivalent. It is not obvious when nondeterminism is equivalent to determinism. If we can answer this question for polynomial time TM's, then we've just solved a famous problem (P vs. NP).

Let's just do 1 more model, that has a different flavor than what we've done, and is slightly more interesting.

1.4 Turing enumerators

Instead of recognition, can you just list the members of a language?

Definition 6.8: A **Turing enumerator** is a Turing machine with a "printer" (output device). Start the Turing machine on an empty tape (all blanks). The Turing enumerator has a special feature that when it goes into "print" mode, it sends out a marked section of the tape to the printer to write out.



(Figure 3.20 in textbook)

The strings that are written out by an enumerator E are considered to be its language:

$$L(E) = \{w : E \text{ outputs } W \text{ at some point when started on blanks}\}.$$

If E halts, then the list is finite. It could also go on forever, so it can enumerate an infinite language.

Again, Turing enumerators capture the same class of languages.

Theorem 6.9: A is Turing-recognizable iff $A = L(E)$ for some enumerator E .

Proof. Here we need to prove both directions.

(\leftarrow) Convert E to an ordinary recognizer M . Given E , we construct a TM M with E built inside it.

Have M leave the input string w alone. M moves to the blank portion of the tape and runs E . When E decides to print something out, M takes a look to see if the string is w . If not, then M keeps simulating E . If the string is w , then M accepts.

Note that if M doesn't find a match, it may go on forever—this is okay, M can loop by rejecting. We have to take advantage of M being able to go on forever.

(\rightarrow) Convert M to enumerator E . The idea is to feed all possible strings to M in some reasonable order, for instance, lexicographic order $\varepsilon, 0, 1, 00, 01, 10, 11$.

However, we have to be careful. Suppose M is running on 101. If M accepts 101, then we print it out. If M halts and rejects 101, then E should move on to the next string. The only problem is when M runs forever. What is E supposed to do? E doesn't know M is going forever! We can't get hung up running M on 101. We need to check 110 too! The solution is to run M for a few steps on any given string, and if hasn't halted then move on, and come back to it later.

We share time among all strings where computation hasn't ended. Run more and more strings for longer and longer. More precisely, for $k = 1, 2, 3, \dots$, E runs M on the first k strings for k steps. If M ever accepts some string s , then print s . \square

§2 Philosophy: Church-Turing Thesis

The Church-Turing Thesis was important in the history of math. After proposing all these different models to capture what we can compute, people saw how they were all equivalent (in an non-obvious way).



Axiom 6.10 (Church-Turing Thesis): **church-turing** Our perception of what we can do with a computer (an algorithm, effective procedure) is exactly captured by Turing machine.

Our intuitive “Algorithm” is the precise notion of a “Turing machine.”

It might seem arbitrary for us to focus on Turing machines, when this is just one model of computation. But the Church-Turing Thesis tells us the models are all equivalent! The notion of algorithm is a natural, robust notion. This was a major step forward in our understanding of what computation is.

It’s almost saying something about the physical universe: there’s nothing we can build in the physical world that is more powerful than a Turing machine.

David Hilbert gave an address at the International Congress of Mathematicians in 1900. He was probably the last mathematician who knew what was going on in every field of mathematics at the same time. He knew the big questions in each of those fields. He made a list of 23 unsolved problems that he felt were a good challenge for the coming century; they are called the Hilbert problems.

Some of them are solved. Some of them are fuzzy, so it’s not clear whether they are solved. Some of them have multiple parts, just like homework.

One of the questions was about algorithms—Hilbert’s tenth problem, which I’ll describe.

Suppose we want to solve a polynomial equation $3x^2 + 17x - 22 = 0$. This is easily done. But suppose we don’t want to know if a polynomial equation has a root, but whether it have a root where variables are *integers*. Furthermore, we allow variables with several variables. This makes things a lot harder. For instance, we could have

$$17xy^2 + 2x - 21z^5 + xy + 1 = 0.$$

Is there an assignment of integers in x, y, z such that this equation is satisfied?

Hilbert asked: Is there a finite procedure which concludes after some finite number of steps, that tells us whether a given polynomial has an integer root?

We can put this in our modern framework. Hilbert didn’t know what a “procedure” was in a mathematical sense. In these days, this is how we would phrase this question.

Problem 6.1 (Hilbert’s Tenth Problem): Let

$$D = \{p : p \text{ is a multivariable polynomial that has a solution (root) in integers}\}.$$

Is D decidable?⁴

The answer is no, as Russian mathematician Matiasевич found when he was 20 years old.

Without a precise notion of procedure, there was no hope of answering the question. Hilbert originally said, give a finite procedure. There was no notion that there might not be a procedure! It took 35 years before the problem could be addressed because we needed a formal notion of procedure to prove there is none.

Here, the Church-Turing Thesis played a fundamental role.

Lecture 7

Thu. 9/27/12

Last time we talked about

- TM variants
- Church-Turing Thesis

Today we'll give examples of decidable problems about automata and grammars.

§0 Hints

Problem 1: Prove some language not context-free. Use the pumping lemma! The trick is to find the right string to use for pumping. Choose a string longer than the pumping length, no matter how you try to pump it up you get out of the language. The first string you think of pump may not work; probably the second one will work.

Problem 2: Show context-free. Give a grammar or a pushdown automata. At first glance it doesn't look like a context-free language. Look at the problem, and see this is a language written in terms of having to satisfy two conditions, each of which seems to use the condition. The problem seems to be if you use the stack for the first condition it's empty for the second condition. Instead of thinking of it as an AND of two conditions, think of it as an OR of several conditions.

Problem 3 and 4: easy. 4 about enumerator.

Problem 5: variant of a Turing machine. Practice with programming on automaton.

Problem 6: (4.17 in the 2nd edition and 4.18 in the 3rd edition) Let C be a language. Prove that C is Turing-recognizable iff a decidable language D exists such that

$$C = \{x : \text{for some } y, \langle x, y \rangle \in D\}.$$

We'll talk about this notation below.

⁴Note D is Turing recognizable. Just start plugging in all possible tuples of integers, in a systematic list that covers all tuples. If any one is a root, accept, otherwise carry on.

0.1 Encodings

We want to feed more complicated objects into Turing machines—but a Turing machine can only read strings.

If we want to feed a fancy object into a program we have to write it as a string. We need some way of encoding objects, and we'd like some notation for it.

For any formal finite object B , for instance, a polynomial, automaton, string, grammar, etc., we use $\langle B \rangle$ to denote a reasonable encoding of B into a binary string. $\langle B_1, \dots, B_k \rangle$ encodes several objects into a string.

For example, to encode an automaton, write out the list of states and list of transitions, and convert it into a long binary string, suitable for feeding in to a TM.

Problem 6 links recognizability and decidability in a nice way. You can think of it as saying: “The projection of a recognizable language is a decidable language.” Imagine we have a coordinate systems, x and y . Any point corresponds to some (x, y) .

Look at all x such that for some y , $\langle x, y \rangle$ is in D . So C consists of those points of x underneath some element of D . We're taking all the (x, y) pairs and remove the x . Shrinks 2-d shape into 1-d shadow; this is why we call it the projection. This will reappear later on the course when we talk about complexity theory!

You need to prove an “if and only if.” Reverse direction is easy. If D is decidable, and you can write C like this, we want C to be recognizable. We need to make a recognizer for C . It accepts for strings in the language but may go on forever for strings not in the language.

Accept if in C but don't know what y is. Well, let's not give it away more!

The other direction is harder. Given T-recognizable C , show that D is decidable, we don't even know what D is! We have to find an “easier” language, so y sort-of helps you determine whether $x \in C$. If C were decidable easy, just ignore y . Which y should you use? Make it a decidable test.

The y somehow proves that $x \in C$. For each $x \in C$ there has to be some y up there somewhere. What does y do? The nice thing about y in C , is that if the proof fails, the decider can see that the proof fails. (Whatever I mean by proof. Conceptually. Test for validity.) Go from recognizer to decider. Nice problem!

§1 Examples of decidable problems: problems on FA's

By the Church-Turing Thesis 6.10, algorithms are exactly captured by Turing machines. We'll talk about algorithms and Turing machines interchangeably (so we'll be a lot less formal about putting stuff in Turing machine language).

Theorem 7.1: thm:ADFA Let

$$A_{\text{DFA}} = \{ \langle B, w \rangle : B \text{ is a DFA and } B \text{ accepts } w \}.$$

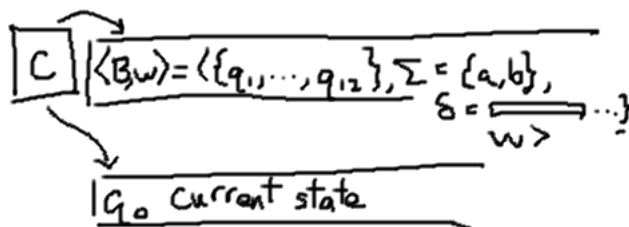
Then A_{DFA} is decidable.

The idea is to just run the DFA! We'll do some easy things to start.

Proof. We'll give the proof in high level descriptive language (like pseudocode), rather than explicitly draw out state diagrams. We'll write the proof in quotes to emphasize that our description is informal but there is a precise mathematical formulation we can make.

Let $C =$ "on input string x

1. Test if x legally encodes $\langle B, w \rangle$ for some DFA B and w . Does it actually encode a finite automata and string? If not, reject (it's a garbage string).
2. Now we know it's of the correct form. Run B on w . We'll give some details. We'll use a multi-tape Turing machine.



Find the start state, and write it on the working tape. Symbol by symbol, read w . At each step, see what the current state is, and transition to the next state based on the symbol read, until we get to end of w . Look up the state in B to see whether it is an accept state; if so accept, and otherwise reject.

3. Accept if B accepts. Reject if B rejects."

□

Under the high-level ideas, the details are there. From now on, we'll just give the high-level proof. This is the degree of formality that we'll provide and that you should provide in your proofs.

Brackets mean we agree on some encoding. We don't go through the gory details of spelling out exactly what it is; we just agree it's reasonable.

We go through some details here, so you can develop a feeling for what intuition can be made into simulations. Each stage should be obviously doable in finite time.

Turing machines are "powerful" enough: trust me or play around with them a bit to see they have the power any programming language has.

We'll do a bunch of examples, and then move into some harder ones. Let's do the same thing for NFA's.

Theorem 7.2: Let

$$A_{\text{NFA}} = \{ \langle B, w \rangle : B \text{ is a NFA and } B \text{ accepts } w \}.$$

Then A_{NFA} is decidable.

We can say exactly what we did before for NFA's instead of DFA's. However, we'll say it a slightly different way, to make a point.

Proof. We're going to use the fact that we already solved the problem for DFA's.

Turing machine $D =$ "on input $\langle B, q \rangle$, (By this we mean that we'll check at the beginning whether the input is of this form, and reject if not.)

1. Convert the NFA B to an equivalent DFA B' (using the subset construction).

All of those constructions can be implemented with Turing machines.

2. Run TM C (from the proof of Theorem 7.1) on input $\langle B', w \rangle$.

3. Accept if C accepts, reject if C rejects.

□

We see that in this type of problem, it doesn't matter whether we use NFA or DFA, or whether we use CFG or PDA, because each in the pair recognizes the same class of languages. In the future we won't spell out all all equivalent automata; we'll just choose one representative (DFA and CFG).

Let's do a slightly harder problem.

Theorem 7.3: EDFA Let

$$E_{\text{DFA}} = \{ \langle B \rangle : B \text{ is DFA and } L(B) = \phi \}.$$

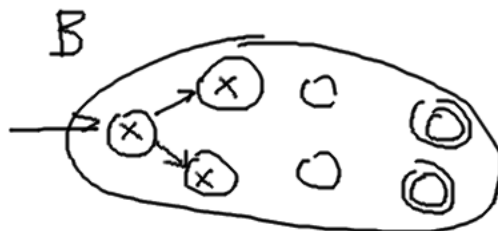
Then E_{DFA} is decidable.

This is the emptiness testing problem for DFA's: Is there one string out there that the DFA accepts?

Proof. How would you test if a DFA B has an empty language? Naively we could test all strings. That is not a good idea, because this is not something we can do in finite time.

Instead we test whether there is a path from the start state to any of the accept states: Mark the start state, mark any state emanating from a previously marked state, and so forth, until you can't mark anything new. We eventually get to all states that are reachable under some input.

If we've marked all reachable states, and haven't marked the accept state, then B has empty language.



With this idea, let's describe the Turing machine that decides E_{DFA} .

Let $S = \text{" on input } \langle B \rangle$.

1. Mark the start state.
2. Repeat until nothing new is marked: Mark all states emanating from previously marked states.
3. Accept if no accept state is marked. Reject otherwise.

□

This is detailed enough for us to build the Turing machine if we had the time, but high-level enough so that the focus is on big details and not on fussing with minor things. (This is how much detail I expect in your solutions.)

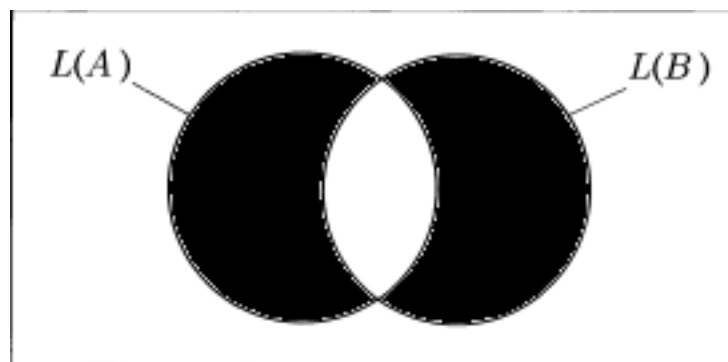
Note this applies to NFA's as well because we can convert NFA's to DFA's and carry out the algorithm we just described.

Theorem 7.4 (Equivalence problem for DFA's): **EQDFA**

$$EQ_{\text{DFA}} = \{\langle A, B \rangle : A, B \text{ DFA's and } L(A) = L(B)\}$$

Proof. Look at all the places where $L(A)$ and $L(B)$ are not the same. Another way to phrase the equivalence problem (is $L(A) = L(B)$) is as follows: Is the shaded area below, called the *symmetric difference*, empty?

$$A \triangle B = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$



Let $E = \text{" on input } \langle A, B \rangle$.

- Construct a DFA C which recognizes $A \triangle B$. Test if $L(C) = \phi$ using the TM S that tested for emptiness (Theorem 7.3).
- Accept if it is ϕ , reject if not.

□

§2 Problems on grammars

Let's shift gears and talk about grammars.

Theorem 7.5: ACFG Let

$$A_{\text{CFG}} = \{ \langle G, w \rangle : G \text{ is a CFG and } w \in L(G) \}.$$

Then A_{CFG} is decidable.

Proof. We want to know: does G generate w ?

We need an outside fact. We can try derivations coming from the start variable, and see if any of them lead to w . Unfortunately, without extra work, there are infinitely many things to test. For example, a word w may have infinitely many parse trees generating it, if we had a rule like $R \rightarrow \varepsilon | R$.

Definition 7.6: A CFG is in **Chomsky normal form** if all rules are of the form $S \rightarrow \varepsilon$, $A \rightarrow BC$, or $A \rightarrow a$, where S is the start variable, A, B, C are variables, B, C are not S , and a is a terminal.

The Chomsky normal form assures us that we don't have loops (like $R \rightarrow R$ would cause). A variable A can only be converted to something longer, so that the length can only increase.

We need two facts about Chomsky normal form.

Theorem 7.7: chomsky-nf

1. Any context-free language is generated by a CFG in Chomsky normal form.
2. For a grammar in Chomsky normal form, all derivations of a length n string have at most a certain number of steps, $2n - 1$.

Let $F = \text{"on } \langle G, w \rangle$.

1. Convert G to Chomsky normal form.
2. Try all derivations with $2n - 1$ steps where $n = |w|$.
3. Accept if any yield w and reject otherwise

□

Corollary 7.8: CFL-decidable Every CFL is decidable.

This is a different kind of theorem from what we've shown. We need to show *every* context-free language is decidable, and there are infinitely many CFL's.

Proof. Suppose A is a CFL generated by CFG G . We build a machine M_G (depending on the grammar G) deciding A : $M_G = \text{"on input } w$,

1. Run TM F deciding A_{CFG} (from Theorem 7.5) on $\langle G, w \rangle$. Accept if F does and reject if not.

□

Theorem 7.9 (Emptiness problem for CFG's): **EQCFG**

$$E_{\text{CFG}} = \{\langle A \rangle : L(A) = \phi\}$$

is decidable.

Proof. Define a Turing machine by the following. “On input $\langle G \rangle$,

1. First mark all terminal variables

$$\begin{aligned} S &\rightarrow \dot{a}S\dot{b}|T\dot{b} \\ T &\rightarrow \dot{a}|T\dot{a}. \end{aligned}$$

2. Now mark all variables that go to a marked variable:

$$\begin{aligned} S &\rightarrow \dot{a}S\dot{b}|\dot{T}\dot{b} \\ \dot{T} &\rightarrow \dot{a}|\dot{T}\dot{a}. \end{aligned}$$

and repeat until we can't mark any more

$$\begin{aligned} \dot{S} &\rightarrow \dot{a}\dot{S}\dot{b}|\dot{T}\dot{b} \\ \dot{T} &\rightarrow \dot{a}|\dot{T}\dot{a}. \end{aligned}$$

3. If S is marked at the end, accept, otherwise reject.

□



Turing machines can decide a lot of properties of DFA's and CFG's because DFA's and PDA's have finite memory. Thus we may say things like, “mark certain states/variables, then mark the states/variables connected to them, and so on, and accept if we eventually get to...”

In contrast to the previous theorems, however, we have the following.

Theorem 7.10 (Equivalence problem for CFG's): **EQCFG**

$$EQ_{\text{CFG}} = \{\langle A, B \rangle : A, B \text{ CFG's and } L(A) = L(B)\}$$

is *undecidable*.

(Note that the complement of EQ_{CFG} is recognizable. Decidability is closed under complement but not recognizability. In fact, the class of recognizable languages isn't closed under intersection or complement.)

We will prove this later. We also have the following theorem.

Theorem 7.11 (Acceptance problem for TM's):

$$A_{TM} = \{\langle M, w \rangle : \text{TM } M \text{ accepts } w\}$$

is *undecidable*. However it is T -recognizable.

To see that A_{TM} is T -recognizable, let U = “ on $\langle M, w \rangle$. Simulate T on w .” Note this may not stop.

This is a famous Turing machine. It is the “universal machine,” and the inspiration for von Neumann architecture. It is a machine that one can program, without having to rewire it each time, so it can do the work of any other machine.

Lecture 8

Tue. 10/2/12

Today Zack Remscrim is filling in for Michael Sipser.

We summarize the relationships between the three types of languages we've seen so far.

§1 Languages

Proposition 8.1: lang-subset Each of the following classes of language is a proper subset of the next.

1. Regular
2. CFL
3. Decidable
4. Turing-recognizable

Proof. We've already shown that the classes are subsets of each other.

We have that $\{a^n b^n : n \geq 0\}$ is a CFL but not a regular language, and $\{a^n b^n c^n : n \geq 0\}$ is decidable but not CFL.

Today we'll finish the proof by showing that decidable languages are a *proper* subset of T -recognizable languages, by showing that

$$A_{TM} = \{\langle M, w \rangle : M \text{ is a TM that accepts } w\}$$

is Turing-recognizable but not decidable. □

We'll also show there is a language that is not even Turing-recognizable.

Theorem 8.2: A_{TM} is Turing-recognizable.

Proof. Let U = “on input $\langle M, W \rangle$,

1. Run M on w .
2. If M accepts, then accept.
If M halts and rejects, then reject.”

M doesn't have to be a decider, it may reject by looping. Then U also rejects by looping. \square

We can't do something stronger, namely make a test for membership and be certain that it halts (i.e., make a decider).

§2 Diagonalization

Diagonalization is a technique originally introduced to compare the sizes of sets. We have a well-defined notion of size for finite sets. For infinite sets, it's not interesting just to call them all “infinite.” We'd also like to define the size of an infinite set, so that we can say one infinite set is larger or the same size as another.

Definition 8.3: Two sets A and B **have the same size** if there exists a one-to one (injective) and onto (surjective) function $f : A \rightarrow B$. Here,

- “one-to-one” means if $x \neq y$, then $f(x) \neq f(y)$.
- “onto” means for all $y \in B$ there exists $x \in A$ such that $f(x) = y$.

We also say that $f : A \rightarrow B$ is a 1-1 correspondence, or a **bijection**.

This agrees with our notion of size for finite sets: we can pair off elements in A and B (make a bijection) iff A and B have the same number of elements.

This might seem like an excessive definition but it's more interesting when applied to infinite sets.

Example 8.4: Let

$$\begin{aligned}\mathbb{N} &= \{1, 2, 3, 4, \dots, \} \\ \mathbb{E} &= \{2, 4, 6, 8, \dots, \}.\end{aligned}$$

Then \mathbb{N} and \mathbb{E} have the same size, because the function $f(n) = 2n$ gives a bijection $\mathbb{N} \rightarrow \mathbb{E}$.

n	$f(n)$
1	2
2	4
3	6
4	8
\vdots	\vdots

Note \mathbb{N} and \mathbb{E} have the same size even though \mathbb{E} is a proper subset of \mathbb{N} .

This will usefully separate different kinds of infinities. We're setting the definition to be useful for us. We want to distinguish sets that are much much bigger than \mathbb{N} , such as the real numbers.

Definition 8.5: A set is **countable** if it is finite or has the same size as \mathbb{N} .

Example 8.6: The set of positive rationals

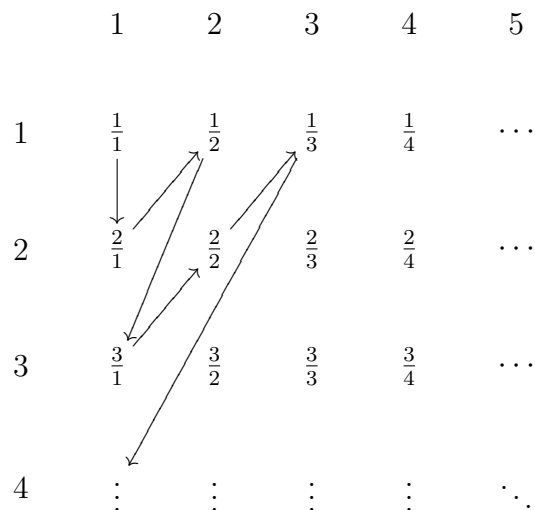
$$\mathbb{Q}^+ = \left\{ \frac{m}{n} : m, n \in \mathbb{N} \right\}$$

is countable.

To see this, we'll build up a grid of rational numbers in the following way.

	1	2	3	4	5
1	$\frac{1}{1}$	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	\dots
2	$\frac{2}{1}$	$\frac{2}{2}$	$\frac{2}{3}$	$\frac{2}{4}$	\dots
3	$\frac{3}{1}$	$\frac{3}{2}$	$\frac{3}{3}$	$\frac{3}{4}$	\dots
4	\vdots	\vdots	\vdots	\vdots	\ddots

Every rational number certainly appears in the table. We'll snake our way through the grid.



Now put the numbers in this order in a list next to $1, 2, 3, \dots$

n	$f(n)$
1	$\frac{1}{1}$
2	$\frac{2}{1}$
3	$\frac{1}{2}$
4	$\frac{2}{2}$
5	$\frac{1}{3}$
\vdots	\vdots

Note some rational numbers appear multiple times, for instance, 1 appears as $\frac{1}{1}, \frac{2}{2}, \dots$. In the correspondence we don't want to repeat these, we just go to the next value. This creates a bijection between \mathbb{N} and \mathbb{Q}^+ , showing \mathbb{Q}^+ is countable.

A lot of infinite sets seem to have the same size, so is this a completely useless definition? No, there are infinite sets bigger than others, so it is useful for us. Are the real numbers of the same size as rational numbers?

Theorem 8.7: thm:R-uncountable The set of real numbers \mathbb{R} is not countable.

Our proof uses the technique of *diagonalization*, which will also help us with the proof for A_{TM} .

Proof. Assume by contradiction that \mathbb{R} is countable; there exists a bijection $f : \mathbb{N} \rightarrow \mathbb{R}$. We're going to prove it's not a bijection, by showing that it misses some y .

Let's illustrate with a potential bijection f .

n	$f(n)$
1	1.4142
2	3.1415
3	2.7182
4	1.6108
\vdots	\vdots

We'll construct a number y that is missed by f in the following way: Let y differ from $f(i)$ at the i th place to the right of the decimal point.

n	$f(n)$
1	1. 4 142
2	3.1 4 15
3	2.71 8 2
4	1.610 8
\vdots	\vdots

For instance, let


$$y = 0.\mathbf{3725} \dots$$

We claim y can't show up in the image of f . Indeed, this is by construction: it differs from $f(i)$ in the i th place, so it can't be $f(i)$ for any i .

There's one little detail: 1 and .999... are equal even though their decimal representations are different. To remedy this, we'll just never use a 0 or 9 in y to the right of the decimal.

This is just to get around a little issue, though. The main idea is that given an alleged bijection, I can show it's not a bijection by constructing a value it misses.

We've shown that there can't be a bijection $\mathbb{N} \rightarrow \mathbb{R}$; therefore \mathbb{R} is uncountable. \square

 Use diagonalization when you want to construct an element that is different from every element on a given list. This is used in proofs by contradiction, for example, when you want to show a function can't hit every element of a set.

Theorem 8.8: Let

$$\mathcal{L} = \{L : L \text{ is a language}\}.$$

Then \mathcal{L} is uncountable.

The proof uses the same diagonalization idea.

Proof. It's enough to show just \mathcal{L} is uncountable when the alphabet is just 0, because every alphabet contains at least 1 symbol. The set of possible strings is

$$\{0\}^* = \{\varepsilon, 0, 00, 000, \dots\}.$$

For a language L , define the characteristic vector of χ_L by $\chi_L(v) = 0$ if $v \notin L$ and 1 if $v \in L$. χ_L simply records whether each word is in L or not.

There is a correspondence between each language and its characteristic vectors. All we have to show is the set of characteristic vectors is uncountable. The set of strings of countable length is uncountable. Assume by contradiction that $\{\chi_L : L \text{ is a language over } \{0\}\}$ is countable.

Suppose we have some bijection from \mathbb{N} to the set of characteristic vectors χ_L ,

n	$f(n)$
1	1011...
2	0000...
3	1111...
4	1010...
\vdots	\vdots

Again, there has to be some binary string that is missed by f . We choose y so it differs from $f(i)$ at the i th place.

n	$f(n)$
1	1 011
2	0 0 00
3	11 1 1
4	101 0
\vdots	\vdots
$y = \text{0101} \dots$	

This y can't ever appear in the table: Suppose $f(n) = y$. This is impossible since we said y differs from $f(n)$ in the n th place. This shows the set of languages really is uncountable. \square

Note that our proof works no matter what the alleged bijection f looks like. Whatever f does, it pairs up each n with one binary number. All I have to do is construct a y that differs from every single $f(n)$. It's constructed so it differs from every $f(i)$ somewhere.

This shows that no function f can work.

§3 A_{TM} : Turing-recognizable but not decidable

Consider

$$\mathcal{M} = \{M : M \text{ is a Turing machine}\}.$$

(We fix the tape alphabet.) This is countable because there is a way to encode a Turing machines using a finite alphabet, with a finite length word. Now some words represent valid Turing machines.

Now pair the first valid string representing a Turing machine with 1, the second valid string representing a Turing machine with 2, and so forth. This shows \mathcal{M} is countable.

The set of all languages is uncountable, but the set of Turing machines is countable. This implies the following fact.

Theorem 8.9: There exists a language L such that L is not Turing-recognizable.

Proof. If every language were Turing-recognizable, we can map every language to a Turing machine that recognizes it; this would give a correspondence between a uncountable and a countable set. \square

We're now ready to prove that A_{TM} is undecidable.

Theorem 8.10: ATM A_{TM} is undecidable.

Proof. We'll proceed by contradiction using diagonalization.

Assume for sake of contradiction that A_{TM} is decidable. Then there exists a decider H , such that

$$H(\langle M, w \rangle) = \begin{cases} \text{accept,} & \text{when } M \text{ accepts.} \\ \text{rejects,} & \text{when } M \text{ rejects.} \end{cases}$$

(Because H is a decider, it is guaranteed to halt.) Using this machine H we're going to make a machine D that does something utterly impossible. This will be our contradiction.

Let D = "On input $\langle M \rangle$,

1. Run H on $\langle M, \langle M \rangle \rangle$.⁵ H answers the A_{TM} problem, so it answers: does machine M accept its own description?⁶
2. If H accepts, reject.
If H rejects, accept.

Now for any Turing machine M , D accepts $\langle M \rangle$ iff M doesn't accept $\langle M \rangle$.

What happens if we feed $\langle D \rangle$ to D ? We get that D accepts $\langle D \rangle$ iff D doesn't accept $\langle D \rangle$. This is a contradiction!

Let's look at what we've done. Let's say A_{TM} were decidable. Let H decide the A_{TM} problem. We construct D that uses H as a subroutine, that does the opposite of what a machine M does when fed the description of M . Then when we feed $\langle D \rangle$ to D , D is now completely confused! We get a contradiction, hence A_{TM} can't be decidable.

(If you're completely confused, there's more explanation in the next lecture.) \square

This completes the picture in Proposition 8.1.

⁵Compare this to looking at the i th symbol of $f(i)$.

⁶This is a valid question, because we can encode the machine in a string, and the machine accepts strings. We can feed the code of a program to the program itself. For instance, we could have an optimizing compiler for C , written in C . Once we have the compiler, we might compile the compiler.

§4 Showing a specific language is not recognizable

So far we know that there are nonrecognizable languages, but we haven't given an explicit description of one. Now we'll show a specific language is not recognizable. For this the following lemma is useful.

Lemma 8.11: A is decidable iff A is T -recognizable and \overline{A} is T -recognizable (we say that A is **co-T-recognizable**).

This immediately implies that $\overline{A_{TM}}$ is not recognizable.

Proof. (\implies): Suppose A is decidable. Then A is T -recognizable. For the second part, if A is decidable, then \overline{A} is decidable (decidable languages are closed under complementation: just run the decider and do the opposite. You're allowed to do the opposite because the decider is guaranteed to halt). Hence \overline{A} is also T -recognizable.

(\impliedby): Suppose R recognizes A and S recognizes \overline{A} . We construct a decider T for A . If we can do this, we're done.

Construct T as follows. T = "on input w ,

1. Run R and S on w in parallel until one accepts. (We can't run R and see what it does, and then run S , because R and S may not be decidable— R might run forever, but T needs to be a decider.) This won't take forever: either R or S might run forever on a particular input, but at least one of them will accept eventually, because a string is either in A or \overline{A} .
2. If R accepts, then accept. If S accepts (i.e. $w \in \overline{A}$), then reject.

□

Lecture 9 Thu. 10/4/12

Last time we saw

- A_{TM} is undecidable.
- $\overline{A_{TM}}$ is T -unrecognizable.
- Diagonalization method

We showed how the diagonalization method proved the reals were uncountable, and also applied the same idea to decidability. We'll give a quick recap, and highlight why the idea behind the two diagonalization arguments are the same.

Theorem 9.1: \mathbb{R} is uncountable.

Proof. Assume for contradiction that \mathbb{R} is countable. Suppose we're given a bijection.

n	$f(n)$
1	2.71828...
2	3.14159...
3	0.11111...
4	\vdots
\vdots	\vdots

Take a number differing from $f(i)$ in i th place. For instance, take $x = 0.654\dots$ where $6 \neq 7$, $5 \neq 4$, and $4 \neq 1$.

Then x can't be on the list. For instance, it can't be the 17th number because it's different in 17th place. Thus f fails to be a bijection. This is Cantor's proof. \square

We applied diagonalization to decidability problems.

Theorem 9.2: A_{TM} is undecidable.

Proof. Assume A is decidable by a Turing machine H . Use H to get TM D , that does the following.

1. D on $\langle M \rangle$ rejects if M accepts $\langle M \rangle$ and accepts if M rejects (halt or loop) $\langle M \rangle$.

Then D accepts $\langle M \rangle$ iff M doesn't accept $\langle M \rangle$; hence D accepts $\langle D \rangle$ if D doesn't accept $\langle D \rangle$, contradiction.

This is the same idea as Cantor's diagonalization argument! To see this, let's make a table of how Turing machines respond to descriptions of Turing machines as inputs:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	\dots	$\langle D \rangle$
M_1	accept	reject	reject	\dots	
M_2	reject	reject	reject		
M_3	accept	accept	accept	\dots	
\vdots		\vdots		\ddots	
D	rejects	accept	reject		?

We programmed D so that it differed from what M_i decided on $\langle M_i \rangle$. However we get a contradiction because nothing can go in the box labeled "?", hence D can't be on the list of all Turing machines. \square

Today we'll show a lot of other problems are undecidable. There's now a shortcut: by proving that A_{TM} is undecidable, we will show a lot of other problems inherit A_{TM} 's undecidability. Then we won't need the diagonalization argument again.

Today we'll use

1. Reducibility to show undecidability
2. Mapping reducibility to show T-unrecognizability.

§1 Reducibility

Let

$$\text{HALT}_{TM} = \{ \langle M, w \rangle : \text{TM } M \text{ halts on input } w \}.$$

Theorem 9.3: HALT_{TM} is undecidable.

We can go back and use the diagonalization method. But we'll give a different technique.

Proof. Suppose we can decide the halting problem by some Turing machine. We're going to use that to decide A_{TM} , which we know is not decidable. Hence our assumption that HALT_{TM} is decidable must be false, i.e., the halting problem cannot be decided.

Assume for sake of contradiction that TM R decides HALT_{TM} . We will construct a TM S deciding A_{TM} .

Let S = "on input $\langle M, w \rangle$.


1. Use R to test if M halts on w . If not, reject. If yes, run M on w until it halts."

Why does this work? If M doesn't halt on w , then we know M doesn't accept, so reject.

Suppose R says M does halt. We don't know whether it accepts right off. Our algorithm says to run M on w . We don't have to worry about M going forever, because R has told us that M halts! We'll eventually come to the end, M will accept or reject, and we can give our answer about A_{TM} .

Thus we can use our HALT_{TM} machine to decide A_{TM} . □

This is called **reducing** A_{TM} to the HALT_{TM} problem.

 **Reducibility:** One way to show a problem is undecidable is by reducing it from a problem we already know is undecidable, such as A_{TM} .

Concretely, to show a problem P_1 is undecidable, suppose it had a decider. Use the decider for P_1 to decide an undecidable problem (e.g. A_{TM}). This gives a contradiction.

If some problem has already been solved, and we reduce a new problem to an old problem, then we've solved it too. For instance, consider the acceptance problem for DFA's. We showed that A_{DFA} is decidable (Theorem ??). Then it immediately follows that A_{NFA} is decidable, because we can reduce the A_{NFA} problem to a A_{DFA} problem (Theorem ??). We converted the new problem into the solved problem.

Definition 9.4: We say A is **reducible** to B if a solution to B gives a solution to A .

Here we used reducibility in a twisted way. If A is reducible to B , we know that if we can solve B then we can solve A . Hence if we can't solve A then we can't solve B .

We used a HALT_{TM} machine to decide A_{TM} , so we reduced A_{TM} to HALT_{TM} .

All “natural” problems which are undecidable can be shown to be undecidable by reducing A_{TM} to them or their complement.

! When trying to show problems are undecidable, reduce *from* A_{TM} (not to A_{TM}).^a

^aOn an undecidability problem on the exam, if you just write “reduction from A_{TM} ” you will get partial credit. If you write “reduction to A_{TM} ” you will get less credit.

Let

$$E_{TM} = \{\langle M \rangle : \text{TM } M \text{ and } L(M) = \emptyset\}.$$

Theorem 9.5: thm:etm E_{TM} is undecidable.

Proof. Use reduction *from* A_{TM} to E_{TM} .

Here’s the idea. Assume R decides E_{TM} . We construct S deciding A_{TM} . How do we do this? S wants to decide whether a certain string is accepted; R only tells whether the entire language is empty. We’re going to trick R into giving me the answer we’re looking for.

Instead of feeding the TM M into R , we’re going to modify M . In the modified version of M it’s going to have w built in: M_w . When start up M_w on any input it will ignore that input, and just run M on w . It doesn’t matter what I feed it; it will run as if the input were w . The first thing it does is erase the input and writes w . M_w will always to do the same thing: always accept or always reject, depending on what M does to w . (The language is everything or nothing.)

Now we feed M_w into R . The only way the language can be nonempty is if M accepts w . We’ve forced R to give us the answer we’re looking for, i.e., we’ve converted acceptance into emptiness problem. Now we’re ready to write out the proof.

$S =$ “On input $\langle M, w \rangle$,

1. Construct $M_w =$ “ignore input.
 - (a) Run M on w .
 - (b) Accept if M accepts.”
2. Run R on $\langle M_w \rangle$.
3. Give opposite answer. (R is a decider. If R accepts $\langle M_w \rangle$, then M_w ’s language is empty, so M did not accept w , so reject.)

This machine decides A_{TM} , which is a contradiction. Hence our assumption was incorrect; E_{TM} is undecidable. \square

§2 Mapping reducibility

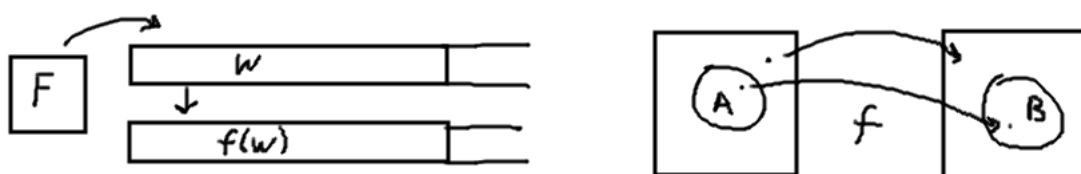
We gave a general notion of reducibility, but not a specific definition. In this section we introduce a specific method called mapping reducibility.

Definition 9.6: Let A and B be languages. We say that A is **mapping reducible** to B , and write⁷

$$A \leq_m B$$

if there is a **computable function** $f : \Sigma^* \rightarrow \Sigma^*$ and for all w , $w \in A$ iff $f(w) \in B$.

We say $f : \Sigma^* \rightarrow \Sigma^*$ is **computable** if some TM F halts with $f(w)$ on the tape when started on input w .



Why is mapping reducibility useful? Suppose we have a decider for B , and we have f , computed by a decider. We can use the two deciders together to decide whether a string is in A !

Proposition 9.7: pr:map-reduce If $A \leq_m B$ and B is decidable (recognizable) so is A .

Proof. Say R decides B . Let $S =$ “On w ,

1. Compute $f(w)$.
2. Accept if $f(w) \in B$. Reject otherwise.”

For B recognizable, just remove the last line “reject if $f(w) \notin B$.” (We don’t know that R halts.) □

Think of f as a “transformer”: it transforms a problem in A to a problem in B . If A is reducible to B , and A is not decidable, then neither is B .

This will also help us prove a problem is non-T-recognizable.

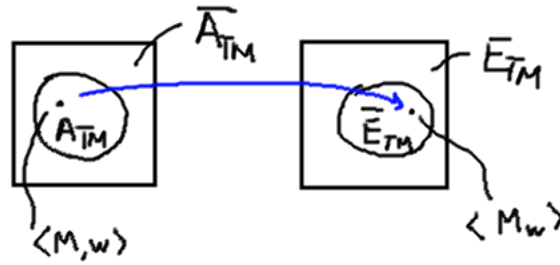
Let’s recast our previous results in the language of mapping reducibility.

In the proof of Theorem 9.5 we showed that

$$A_{TM} \leq_m \overline{E_{TM}}.$$

We converted a problem about A_{TM} to a problem about E_{TM} . Given $\langle M, w \rangle \in A_{TM}$, let f map it to $\langle M_w \rangle$. We have $\langle M, w \rangle \in A_{TM}$ iff $M_w \notin E_{TM}$.

⁷Think of the notation as saying A is “easier” than B



A useful fact is that

$$A \leq_m B \iff \overline{A} \leq_m \overline{B};$$

by using the same f .

We make one more observation, then prove another theorem.

We actually have the following strengthened version of Theorem 9.5.

Theorem 9.8: thm:etm2 E_{TM} is not recognizable.

Proof. We showed $A_{TM} \leq_m \overline{E_{TM}}$, so $\overline{A_{TM}} \leq_m E_{TM}$. Since $\overline{A_{TM}}$ is not recognizable, E_{TM} is not recognizable. \square

We'll now use mapping reducibility to give an example of a language such that neither it nor its complement is recognizable. We will prove this by reduction from A_{TM} .

Theorem 9.9: EQTM EQ_{TM} and $\overline{EQ_{TM}}$ are both T -unrecognizable.

Recall that the equality problem is that given 2 Turing machines, we want to know whether they recognize the same language.

Proof. We show that

1. $\overline{A_{TM}} \leq_m EQ_{TM}$, or equivalently,

$$A_{TM} \leq_m \overline{EQ_{TM}}.$$

We have to give a function

$$f : \langle M, w \rangle \mapsto \langle M_1, M_2 \rangle.$$

We let M_2 be the machine that always rejects. Let $M_1 = M_w$, the machine that simulates M on w . If M accepts/rejects w then the first will accept/reject everything and M_2 will reject everything, so $\langle M, w \rangle \in A_{TM}$ iff $\langle M_1, M_2 \rangle \in \overline{EQ_{TM}}$.

2. $\overline{A_{TM}} \leq_m \overline{EQ_{TM}}$, or equivalently,

$$A_{TM} \leq_m EQ_{TM}.$$

We have to give a function

$$f : \langle M, w \rangle \mapsto \langle M_1, M_2 \rangle.$$

We let M_2 be the machine that always accepts. Again let $M_1 = M_w$, the machine that simulates M on w .

□

In the remaining 2 minutes, we'll look at a cool fact that we'll continue next time.

Lots of undecidable problems appear throughout math have nothing to do with Turing machines.

We'll give the simplest example. Let's define dominoes as pairs of strings of a 's and b 's, such as

$$\left\{ \begin{bmatrix} aba \\ ab \end{bmatrix}, \begin{bmatrix} aa \\ ab \end{bmatrix}, \begin{bmatrix} \\ \end{bmatrix}, \dots \right\}$$

Given a set of dominoes, can we construct a **match**, which is an ordering of dominoes such that the string along the top is the same as the string along the bottom? One little point: each domino can be reused as many times as we want. This means we have a potentially unlimited set of dominoes.

Is it possible to construct a match?

This is an undecidable problem!

And it has nothing to do with automata. But next time we will show we can reduce A_{TM} to this problem; therefore it's undecidable.

Lecture 10

Thu. 10/11/12

Midterm Thu. 10/25 in walker (up through next week's material. Everything on computability theory but not complexity theory.)

Homework due next week.

Handout: sample problems for midterm

Last time we talked about

- reducibility
- mapping reducibility

Today we will talk about

- Post Correspondence Problem
- LBA's
- Computation history method

We have been proving undecidability. First we proved A_{TM} is undecidable by *diagonalization*. Next, by *reducing* ATM to another problem, we show that if the other problem were decidable so is A_{TM} ; hence the other problem must also be undecidable.

Today we'll look at a fancier undecidable problem. It is a prototype for undecidable problems that are not superficially related to computability theory. All proofs for these problems use the method we'll introduce today, the *computation history method*.

Ours is a toy problem, with no independent interest. But it is nice to illustrate the method, and it is relatively clean.

Even the solution to Hilbert's tenth problem uses the computation history method (though there are many complications involved).

§1 Post Correspondence Problem

Given a finite collection of dominoes

$$P = \left\{ \begin{pmatrix} u_1 \\ v_1 \end{pmatrix}, \begin{pmatrix} u_2 \\ v_2 \end{pmatrix}, \dots, \begin{pmatrix} u_k \\ v_k \end{pmatrix} \right\}$$

a **match** is a sequence of dominoes from P (repetitions allowed) where

$$u_{i_1} \cdots u_{i_\ell} = v_{i_1} \cdots v_{i_\ell}$$

The question is: Is there a match in P ?

For example, if our collection of dominoes is

$$P = \left\{ \begin{pmatrix} aa \\ aba \end{pmatrix}, \begin{pmatrix} ab \\ aba \end{pmatrix}, \begin{pmatrix} ba \\ aa \end{pmatrix}, \begin{pmatrix} abab \\ b \end{pmatrix} \right\}$$

then we do have a match because

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline a & b & a & a & b & a & a & a & a & b & a & b \\ \hline a & b & a & a & b & a & a & a & a & b & a & b. \\ \hline \end{array}$$

Formally, define

$$PCP = \{ \langle P \rangle : P \text{ has a match} \}.$$

We will show PCP is undecidable. (Note it is Turing recognizable because for a given arrangement it's easy to see if it's a match; just enumerate all possible arrangements. If we find a match, accept.)

This is an old problem, the first shown undecidable by Post in 1950's.⁸

Let's modify the PCP so that the match has to start with the starting domino (to see how to fix this, see the book).

Theorem 10.1: **PCP** PCP is undecidable.

⁸Don't confuse the Post Correspondence Problem with probabilistic checkable proofs, also abbreviated PCP.

The proof has two ideas. Each takes some time to introduce. Instead of doing them at once (they intertwine), we'll defer the proof and prove a different theorem that uses only one of the ideas. Then we prove this theorem and use both ideas.

To introduce the first idea we'll go back to a problem in computation.

§2 Computation Histories and Linearly Bounded Automata

Definition 10.2: A **linearly bounded automaton** (LBA) is a modified Turing machine, where the tape is only as long as the input string.⁹

The head doesn't move off the left or right hand sides. The machine limited in how much memory it has: if you have an amount of tape, the amount of memory is linearly bounded in terms of the size of the input (you might have a constant better memory because you're allowed a larger tape alphabet, but you can't for instance have n^2 memory). Now let the acceptance and empty problems be

$$\begin{aligned} A_{LBA} &= \{ \langle M, w \rangle : \text{LBA } M \text{ accepts } w \} \\ E_{LBA} &= \{ \langle M \rangle : \text{LBA } M \text{ and } L(M) = \phi \} \end{aligned}$$

Even though A_{TM} was undecidable, A_{LBA} is decidable.

Theorem 10.3: **ALBA** A_{LBA} is decidable.

This is a dramatic change in what we can do computationally!

The key difference that makes A_{LBA} decidable is that linearly bounded automata have finitely many configurations (see below).

As a side remark, A_{LBA} is not decidable by LBA's. In the theorem, by decidable we mean it's decidable by ordinary Turing machines. It is decidable but only by using a lot of memory.

Proof. Define a configuration to be a total snapshot of a machine at a given time:

$$(q, t, p)$$

where q is the state, t is the tape contents, and p is the head position. For an input size, the number of configurations of a LBA is finite.

If we run the LBA for certain amount of time T , then it has to repeat a configuration. If it has halted by time T , then we know the answer. If the machine hasn't halted, it's in a loop and will run forever. The reject.

(We don't even have to remember configurations.)

For a string w of length n , the number of configurations of length n is

$$|Q| \cdot |\Gamma|^n \cdot n.$$

⁹The LBA doesn't have a limited tape. (Then it would be finite automaton.) The tape is allowed to be enough to grow just enough to fit the input, but it can't grow any further.

We now write down the TM that decides A_{LBA} .

“On input $\langle M, w \rangle$,

1. Compute $|Q| \cdot |\Gamma|^n \cdot n$.
2. Run M on w for that many steps.
3. Accept if accepted.
Reject if not yet accepted after that many steps.

□

Note that to write the number $|Q||\Gamma|^n n$ down requires on the order of $n \ln n$ length tape, so intuitively A_{LBA} is not decidable by LBA's. The same diagonalization method can prove that A_{LBA} is not decidable by LBA's. In general, we can't have a class of automata which decide whether automata of that class accept.

In contrast with A_{LBA} , E_{LBA} is still undecidable.

Theorem 10.4: ELBA E_{LBA} is undecidable.

We prove this in a totally different way. Before, to prove E_{TM} is undecidable (Theorem 9.5), we showed A_{TM} reduces to E_{TM} . We also have A_{LBA} reduces to E_{LBA} , but doesn't tell us anything because A_{LBA} is decidable!

Instead we reduce A_{TM} to E_{LBA} . This is not obvious! Assume R decides E_{LBA} . We'll construct S deciding A_{TM} . (This is our standard reduction framework.) This is hard because we can't feed Turing machines into E_{LBA} : it only accepts LBA's as input, not general Turing machines.

We use the idea of computation history.

Definition 10.5: Define an **accepting computation history** of a TM T on input w to be

$$C_1, C_2, \dots, C_{\text{accept}}$$

where C_1 is the start configuration, each C_i leads to C_{i+1} , and C_{accept} is in an accepting state.¹⁰

If a Turing machine does not accept its input, it does not have an accepting computation history. An accepting configuration history exists iff the machine accepts.

It's convenient to have a format for writing this down (this will come up later in complexity theory). Write down (q, t, p) as

$$t_1 q t_2.$$

¹⁰The computation history stores a record of all motions the machine goes through, just as a debugger stores a snapshot of what all the registers contain at each moment.

This means: split the tape into 2 parts, the part before the head is t_1 , the part after the head is t_2 and q points to the first symbol in t_2 . All I'm doing here is indicating the position of the head by inserting a symbol representing the state in between the two parts.

Write the computation history as a sequence of strings like this separated by pound signs.

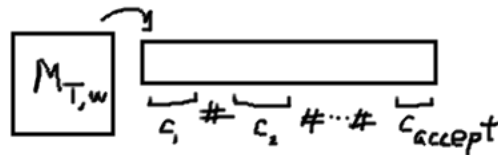
$$C_1 \# C_2 \# \cdots \# C_{\text{accept}}.$$

Here C_1 is represented by $q_0 w_1 \cdots w_n$.

Proof of Theorem 10.4. Let $S = \text{"on input } \langle T, w \rangle$,

1. Construct LBA $M_{T,w} = \text{"On input } z$,
 - (a) test if z is an accepting computation history for T on w .
 - (b) Accept if yes.
Reject if not.

Note $M_{T,w}$ does not simulate T . It simply checks if the input is a valid computation of T , in the form $C_1 \# C_2 \# \cdots \# C_{\text{accept}}$.



Why is this a LBA? It doesn't actually simulate T , it just checks the computation; this doesn't require running off the tape.

What does C_1 need to look like? It must look like $q_0 w_1 \cdots w_n$. How do we check C_2 ? This is a delicate point. See whether C_1 updated correctly to C_2 . Zigzag back and forth between C_1 and C_2 to check that everything follows legally. If anything is wrong, reject. It can put little markers on the input. It repeats all the way to C_{accept} , then check that C_{accept} is in an accept state.

Now the only string our LBA $M_{T,w}$ could possibly accept, by design, is an accepting computation history. The point is that *checking a computation is a lot easier than doing it yourself*; a LBA is enough to check a TM's computation.

Now we have 0 or 1 string is in $M_{T,w}$. If T does not accept, then $L(M_{T,w})$ is empty. If T accepts, there is exactly one accepting computation history, namely the correct $C_1 \# \cdots \# C_{\text{accept}}$. Each configuration forces the next all the way to the accepting configuration. (We've built the Turing machine based on the particular Turing machine T and string w .) Hence $M_{T,w}$ is empty if and only if T does not accept w .

We've reduced A_{TM} to E_{LBA} . This proves E_{LBA} is undecidable. \square

Note the computational history method is especially useful to show the undecidability of a problem that has to do with *weaker* models of computation.

§3 Proof of undecidability of PCP

Proof. (This is slightly rushed; see the book for a complete treatment.) The idea is to construct a collection of dominoes where a match has to be an accepting computation history. (For simplicity, we'll deal with the modified PCP problem where we designate a certain domino to be the starting domino.)

Given T , w , we construct a PCP problem $P_{T,w}$ where a match corresponds to an accepting computation history. We construct the dominoes in order to force any match to simulate a computation history.

- Let the start domino be

$$\begin{pmatrix} \# \\ \#q_0w_1 \dots w_n\# \end{pmatrix}.$$

- If in T we have $\delta(q, a) = (r, b, R)$, put the domino

$$\begin{pmatrix} qa \\ rb \end{pmatrix}$$

in.

Similarly we have a domino for left transitions (omitted).

- For all tape symbols $a \in \Gamma$, put in $\begin{pmatrix} a \\ a \end{pmatrix}$.

Consider a concrete example, $w = 011$, and $\delta(q_0, 0) = (q_5, 2, R)$. We have the domino $\begin{pmatrix} q_00 \\ 2q_5 \end{pmatrix}$. (The construction has one simple idea really: the transition dominoes, like $\begin{pmatrix} qa \\ rb \end{pmatrix}$, force each configuration to lead to the next configuration.) We start with

$$\left| \begin{array}{ccccccc} \# & q_0 & 0 & & & & \\ \# & q_0 & 0 & 1 & 1 & \# & 2 & q_5 \end{array} \right|$$

We've managed to push forward the match on one more domino. Now we have to copy everything. We use $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$, $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $\begin{pmatrix} 2 \\ 2 \end{pmatrix}$, $\begin{pmatrix} \# \\ \# \end{pmatrix}$:

$$\left| \begin{array}{ccccccccccc} \# & q_0 & 0 & 1 & 1 & \# & & & & & \\ \# & q_0 & 0 & 1 & 1 & \# & 2 & q_5 & 1 & 1 & \# \end{array} \right|$$

At end, computation history is done, but match isn't. We add dominoes $\begin{pmatrix} q_{\text{accept}}^C \\ q_{\text{accept}} \end{pmatrix}$, $\begin{pmatrix} Cq_{\text{accept}} \\ q_{\text{accept}} \end{pmatrix}$, and $\begin{pmatrix} q_{\text{accept}}\#\# \\ \# \end{pmatrix}$.

These dominoes "eat" the tape one symbol at time, around the accept state. Putting in one last domino finishes it off.

The start domino is a technicality.

We've reduced A_{TM} to PCP. Therefore, PCP is undecidable. \square



A Turing machine accepts an input if and only if it has an accepting computation history for that input. Thus the problem of whether T accepts w can be formulated as: does T have an accepting computation history for w ?

This formulation is more concrete, and it is much easier to check whether a computation history is correct, then ask whether T accepts w .



To show a problem that isn't related to computability theory is undecidable, find a way to simulate/encode an undecidable problem (such as A_{TM}) with that problem. It is useful to encode computation histories.

Lecture 11

Tue. 10/16/12

Last time we talked about

- the computation history method
- E_{LBA} , PCP undecidable.

Today we'll do the following.

- Review above, ALL_{PDA} undecidable
- Recursion theorem
- Introduction to logic

§0 Homework

Enumerate collection of deciders hit every decidable language. Impossible.

Argue reduction works as claimed.

4. computation history

5-6. today. 6. model for particular sentence. if haven't seen logic before, read section.

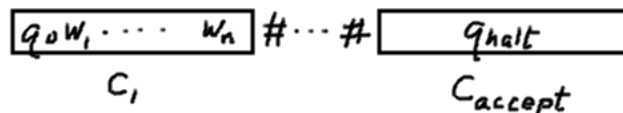
Hint built into problem if you read it carefully.

§1 Computation history method

The state doesn't give us all the information about a Turing machine. Recall that a **configuration** of a Turing machine M consists of the state, tape contents, and head position. We have a convenient representation, where q is written at the head position.



A **computation history** of M on w is $C_1, \dots, C_{\text{halt}}$ a sequence of configurations M enters. It's much easier to check these than to simulate a Turing machine outright. It is possible to check computation history with many kinds of automata or combinatorial objects.



We started with Turing machine M and w , and the problem of whether M accepts w . We found we can convert this into an instance of PCP where the only possible match corresponds to an accepting computation history. The undecidability of Hilbert's tenth problem is shown using the same idea. Here one has to construct polynomial in several variables (originally it was 13 variables). One variable plays the role of the input to the polynomial. The only way for the polynomial to have an integral solution is for the assignment to x to be an accepting computational history suitably encoded in an integer. The other variables are helpers, to make sure the polynomial evaluates to 0 exactly when x is an accepting computational history. Polynomials are a rather restricted comput model, so the polynomial is rather painful to present. (It would take an entire semester.)

Let

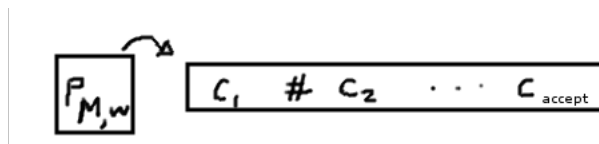
$$\text{ALL}_{PDA} = \{ \langle P \rangle : P \text{ a PDA and } L(P) = \Sigma^* \}.$$

It is the problem: does a pushdown automaton accept all strings? We use the computational history method to show the following.

Theorem 11.1: ALLPDA ALL_{PDA} is undecidable.

Proof. We reduce A_{TM} to ALL_{PDA} . We take $\langle M, w \rangle$ and convert it to a pushdown automaton $P_{M,w}$, such that if can tell whether $P_{M,w}$ accepts all inputs, we can tell whether M accept w .

We construct $P_{M,w}$ by having it operate on computation histories. However, instead of having $P_{M,w}$ accept an accepting computation history, we have it accept every string except for this string. It is the sanitation engineer that accepts all the bad strings, junk, garbage.



If M doesn't accept w , then there is no accepting history, so everything is junk, and $P_{M,w}$ accepts everything. If M accepts w , then $P_{M,w}$ accepts everything except one string. We feed $P_{M,w}$ into a machine for ALL_{PDA} to decide A_{TM} .

How can we make a PDA to accept all the junk? It will use nondeterminism. It checks the configuration history to see if it

- fails to start correctly,
- fails to end correctly,
- or fails to go from one step to the next correctly.

$P_{M,w}$ has the starting configuration built in, so it can check whether the history starts correctly. If not, accept. One branch looks at the last configuration; if that is not right, accept.

Now $P_{M,w}$ scans through the configuration history (nondeterministically). At a place where it guesses there may be an error, it pushes C_i onto the stack, then pop C_i off as it compares C_i to C_{i+1} , seeing if everything matches except that stuff near the head is updated correctly. However, C_i comes out in the reverse order that it was put in. The trick is to write every other configuration in reverse. C_2^R, C_4^R . If $P_{M,w}$ finds a bug, then it accepts. \square

Remark: Why don't we do the original thing, accept only the accepting computation history and nothing else? But that would only prove E_{PDA} is undecidable.

And in fact, we can't do that because E_{PDA} is *decidable*! We have to check each configuration legally follows the next. For instance, if we want to check C_3 legally yields C_4 , we have a problem because we've already read C_3 when comparing it to C_2 . We can't push C_3 and match it with C_4 . This is an unfixable problem.

§2 Recursion theorem

The recursion theorem is an amazing theorem that gives a fitting end to the computability part of the course.

It does some things that seem counter-intuitive.

Can we make a Turing machine (or any reasonable computation model, such as a computer program), such that when we turn it on, it prints out its own description? I.e., can we make a self-reproducing Turing machine? Can we write a piece of code which outputs an exact copy of itself?

We might argue as follows: We'd have to take a copy of the program and put it inside itself, and then another copy inside that copy, and so forth. We'd have to have an infinite program, with copies of itself down forever.

But in fact we *can* make such a program, and it is useful in many cases.

This theorem answers one paradox of life: Living things reproduce—make copies of themselves. Does that mean each living thing had its descendants inside, descendants of descendants inside those, down forever? No. Today, thoughts like that are so absurd they don't even bear consideration. We don't need to do that.

Let's make the paradox more concrete. Can we make a machine to build other machines? We can make a factory (suppose it's fully automated) that makes cars. The factory is more

complicated than the cars: It is at least as complicated because it has instructions for building the cars. It's more complicated because it has all the machinery (robots and so forth). What if we wanted to make a factory that builds factories, identical copies of itself? It has robots which assemble a factory; it seems the factory would have to be more complicated than itself!

But the Recursion Theorem says our intuition is false. We can make a factory-producing factory.

There are practical situations where a program would produce copies of itself. Generally these are malicious programs (depend on whose side you're on). This is one way to make a computer virus—the virus obtains an exact copy of itself, transmits it to the victim computer, installs virus, and continues spreading). One way to transmit virus is the Recursion Theorem. (The other way is to use the special nature of machine to find the address of own executable and get the code.)

Theorem 11.2 (Recursion Theorem): **recursion** We can make a Turing machine “SELF” where on blank input, SELF outputs $\langle \text{SELF} \rangle$.



We can do this in any programming language!

The proof relies on the following lemma.

Lemma 11.3: There is a computable function $q : \Sigma^* \rightarrow \Sigma^*$ such that for every x ,

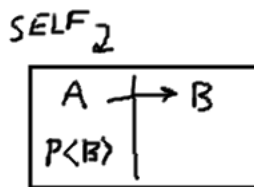
$$q(x) = \langle P_x \rangle$$

where P_x is a Turing machine that prints x (on any input). Moreover, we can determine that Turing machine from x in a computable way.

Proof. Let $P_x = \text{“print } x\text{.”}$ □

(Note the function is called q for quote, because in LISP, this function is represented by sending x to “ x .”)

Proof. The TM SELF will have 2 phases A and B . Control passes from A to B after A is done.



A is very simple: $A = P_{\langle B \rangle}$. Now we have to say what B is.

Why don't we do the same thing to get the A part? Try to set $B = P_{\langle A \rangle}$. This is not possible. A is much bigger than B . A is a B -factory. You can't take the description of A and stuff it into B ; the same circular reasoning got us into trouble in first place.

We don't print out A by having a copy of A inside B . So how does B find what A is? It computes q of the string on the tape. q of that string is A ! Let B = "compute $q(\text{tape})$ and prepend to tape." \square

We can do this in any programming language. Let's do it in English.

Print out this sentence.

If you execute this code, out comes a copy of itself. It tells you as executer to print out copy of itself. However, it cheats, because "this" is a pointer to self. In general, there is no pointer referring to the code. We show how to get the same effect, in software, without self-reference. It achieves the same goal without "this" referring to itself.

Here is the legit version.

Print out two copies of the following, the second one in quotes.

"Print out two copies of the following, the second one in quotes."

If you execute this command, you write the same thing. The A part is below, the B part is above. A is slightly bigger than B by virtue of quotes.

Why is the Recursion Theorem more than just a curiosity? Besides being philosophically interesting, it has applications in computability theory and logic.

The Recursion Theorem in full generality says that we can obtain a complete description and process that copy. Sometimes this is very helpful.

Theorem 11.4 (Full recursion theorem): For any Turing machine T , there is a TM R where $R(x)$ behaves the same as $T(\langle R, x \rangle)$.

Think of R as a compiler which computes its own description.

Proof. Figure 7.

R has 3 pieces now, A , B , and T , where $A = P_{\langle BT \rangle}$ and B is as before. \square

Moral of story:



We can use "get own description" in Turing machine code.

Why would we want to do that? We give a new proof that A_{TM} is undecidable.

Proof. Assume for sake of contradiction that H decides A_{TM} . Consider the following Turing machine: Construct TM R = "on input x ,

1. Get own description $\langle R \rangle$.
2. Run R on $\langle R, x \rangle$ to see if R accepts x .
3. Do the opposite of what R did on $\langle R, x \rangle$

This is a contradiction because R accepts iff R says R doesn't accept x . □

In a sense, the recursion method is standing in for the diagonalization argument here. Let's give another application to something we haven't proved yet. Let

$\text{MIN} = \{ \langle M \rangle : M \text{ is a TM with the shortest description among all equivalent TM's} \}$.

Theorem 11.5: MIN MIN is not Turing-recognizable.

Proof. Recognizable means enumerable.

Assume by way of contradiction that E enumerates MIN. Make R = "on x ,

1. Get $\langle R \rangle$.
2. Run E until some machine M appears where $\langle M \rangle$ is longer than R .
3. Simulate M on x .

Our R will simulate the smallest machine in MIN larger than R , which contradicts the definition of MIN. □

As a summary, here are a list of problems we've proven to be decidable, undecidable, and unrecognizable. (Keep in mind CFG=PDA for the purposes of computation.)

- **Decidable:** A_{DFA} (Theorem 7.1), E_{DFA} (Theorem 7.3), EQ_{DFA} (Theorem 7.4), A_{CFG} (Theorem 7.5), E_{PDA} (exercise), A_{LBA} (Theorem 10.3).
- **Undecidable:** A_{TM} (Theorem 8.10), HALT_{TM} (Theorem 9.3), ALL_{PDA} (Theorem 11.1), EQ_{CFG} (Theorem 7.10), E_{LBA} (Theorem 10.4), PCP (Theorem 10.1). (Note: I haven't checked whether these are recognizable.)
- **Unrecognizable:** $\overline{A_{TM}}$, E_{TM} (Theorem 9.8), EQ_{TM} , $\overline{EQ_{TM}}$ (Theorem 9.9), MIN (Theorem 11.5).

§3 Logic

We'll finish today with a quick tour of logic. This is something that takes weeks in logic course; we'll do it in 10 minutes.

Logic is the math of math itself. Logic formalizes what we mean by mathematical statements. For instance,

$$\phi : \quad \forall x \exists y [y < x].$$

We all believe we can formalize math and define what quantifiers mean. (This is several weeks in logic.) This statement has meaning. It depends on what universe the quantifiers quantifying over. For natural numbers with usual interpretation, this is false. If we instead interpret over \mathbb{R} or \mathbb{Z} , then it is true.

We have to give a universe for quantifiers to range over and define all relation symbols “<.” Ordinary boolean logic allows us to combine statements. We get a meaning for sentence, and it is either true or false in a given model.

Definition 11.6: A **model** is a universe with all relation symbols defined.

For instance, a model for a statement ϕ has ϕ true.

Let the universe be \mathbb{N} and the relations for $+$ and \times . Let

$$\text{Th}(\mathbb{N}, +, \times) = \{\text{all true sentences for this model}\}.$$

Skipping over details, you can imagine what we mean. Some sentences are true, others won't be true. Considering the sentences as strings, is this set decidable? Gödel and others showed it is not decidable. We can write down sentences to describe what Turing machines do; $+$ and \times are expressive enough to describe Turing machines.

There are two notions, truth and provability. What does it mean to give a proof of a true statement? We mean that from the axioms, and simple rules of implication, you can have a chain of reasoning that gets to that statement.

Consider the famous axioms called Peano axioms. Can you prove all true things from Peano axioms? You cannot! You can make a recognizer for all provable things: Search through all possible proofs, until find proof in question. If everything is either provable or its complement is provable, you can search for the proof of the statement or its negation, and this would give a decider for its provability. In truth, this doesn't exist.

Can we exhibit a statement which is unprovable? Try: “This statement is unprovable.” If the sentence were false it would be provable; so it must be true, hence unprovable. This statement is true and unprovable. However, we've actually cheated by using self-reference, however, but one can fix this using the recursion theorem.

Lecture 12

Thu. 10/18/12

Now that we've wrapped up the first half on computability theory, we have a midterm next Thursday, on the 3rd floor of Walker, at the usual time 11:00–12:30. It is open book (the book for this course only)/handouts/notes and covers everything through the last lecture. The test will contain about 4 problems.

Last time we talked about

- the recursion theorem, and
- an introduction to logic.

Today we'll talk about

- an introduction to complexity theory,
- TIME ($t(n)$), and
- P .

We're shifting gears to talk about complexity theory. Today we will introduce the subject and set up the basic model and definitions.

§1 Introduction to complexity theory

We have to go slow at the beginning to get the framework clear. We'll begin by a simple example. In computability theory a typical question is whether a problem is decidable or not. As a research area that was mostly finished in the 50's. Now we'll restrict our attention to decidable languages. The question now becomes

how many time or resources do we need to decide?

This has been an ongoing area of research since the 60's.

Let $A = \{0^k 1^k : k \geq 0\}$. This is a decidable language (in fact, context-free). We want to know how hard it is to see whether a string is in A . We could measure hardness in terms of number of steps, but the number of steps depend on the input. For longer strings it may take more time, and within strings of same length, some strings may take longer than others. For instance, if the string starts with 1, we can reject immediately. The picture is a bit messy, so to simplify, we'll only consider (other do diff things), we'll only consider how much time is necessary as a function of the length n of the input. Among all inputs of given length, we'll try to determine what the worst case is, that is, we consider the worst case complexity.

Summarizing, we consider how the number of Turing machine steps depends on the input length n , and look at the worst case.

Recall that no matter whether we considered single tape, multi-tape, or nondeterministic Turing machines, what is computable remains invariant. This is not true for complexity theory: The picture will change depending on what model you use.

We're not seeking to develop a theory of one-tape Turing machines. Turing machines are a stand-in for computation. We want to try to understand computation. What can do in principle, in a reasonable amount of time? We don't want to just focus on Turing machines.

The fact that complexity depends on the model is a problem. Which should we pick? But as we will see, although it depends on model, it doesn't depend *too much*, and we *can* recover useful theorems.

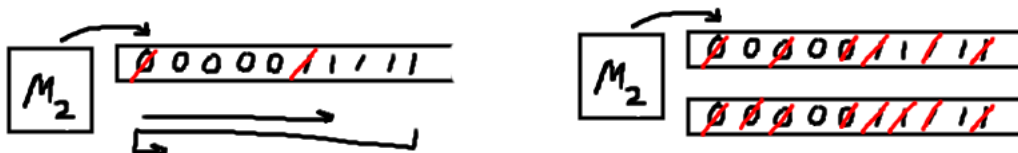
1.1 An example

Example 12.1: We analyze how much space it takes to decide $A = \{0^k 1^k : k \geq 0\}$.

Let $M_1 =$ “(1-tape Turing machine)

1. Scan the input to test whether $w \in 0^* 1^*$. We'll make a pass over the input just to see it's of the right form: no 1's before 0's.
2. Go back to the beginning. Mark off 0 (turn it into another symbol), mark off a 1, then go back to mark off the next 0, then the next 1, and so forth. Continue until we run out of 0's, 1's, or both. If we run out of 0's or 1's first, then reject. If they run out at the same time, accept.

(We needed to spell this out to know how much time the machine is using.)



In summary, repeat until all symbols are crossed off:

- (a) Pass over input, cross off 0's and 1's.
 - (b) If either finishes before other, reject.
3. If all symbols are crossed off, accept.

For this course, we won't care about the constant factors in the time used: $10n^2$ steps and $20n^2$ are equivalent for us. (We could have the machine do some work on the reverse, but we won't bother.)

How much time does M_1 take?

Theorem 12.2: A 1-tape Turing machine can decide A using cn^2 steps for all inputs of length n and some fixed constant c .

We specify number of steps up to constants, so it's convenient to have notation for this. We'll refer to cn^2 as $O(n^2)$. This means at most a constant times n^2 , where the constant is independent of n .

The definition is spelled out in the book; see the definition there.

Proof. Construct M_1 as above. How long does each step take?

1. Scan input to test $w \in 0^*1^*$. This takes $O(n)$ time: n steps to go forward and n steps to go back.
2. Repeat until all symbols are crossed off: We need at most $\frac{n}{2}$ steps.
 - (a) Pass over input, cross off 0's and 1's. This takes $O(n)$ time.
 - (b) If either finishes before other, reject.
3. If all crossed off, accept.

Thus M_1 takes time

$$O(n) + \frac{n}{2}O(n) = O(n^2).$$

(the nice thing about O notation is that we only have to look at the dominant term when adding. We can throw away smaller terms because we can absorb them into the constant in the dominant term.) \square

Is this possible, or can we do better? Let's still stick to one-tape Turing machine. This is not the best algorithm out there; we can find a better one. Here is a suggestion. Zigzagging over the input costs us a lot more time. What if we cross off more 0's and 1's on a pass? We can cross off two 0's and 1's, so this takes half as much time. But we ignore the constant factor, so for our purposes this isn't really an improvement.

We don't ignore these improvements not because they're unimportant. In the real world, it's good to save factor of 2, that's good. However, we choose to ignore these questions because we are looking at a different realm: questions that don't depend on constant factors, or even larger variations.

By ignoring some things, other things come out more visibly. For example, everything reduces to quarks, but it would not benefit biologists to study everything on the level of quarks.

1.2 An improvement

We *can* improve running time to $O(n \log n)$; this is significant from our standpoint. Instead of crossing out a fixed number of 0's and 1's, we'll cross off every other 0 and every other 1 (Fig. 2), remember the even/odd parity of the number of 0's and 1's, and makes sure

the parities agree at every pass. After every step we go to the beginning and repeat, but we ignore the crossed off symbols. We always check the parity at each step. If they ever disagree, which can only happen if the number of 0's and 1's are different, we reject. If the parity is the same at each step, then there must be same number of 0's as 1's. This is because the parities are giving the representation of number of symbols in binary (details omitted).

This is a more efficient algorithm but less obvious than the original algorithm. It looks like there is room for improvement, because we only need n steps to read the input. Could the algorithm do $O(n)$?

We can do $O(n)$ with 2 tapes as follows. Read across the 0's, and copy them onto the 2nd tape. Then read 1's on the 1st tape and match the 0's against the 1's on the second tape. We don't need to zigzag, and we can finish in $2n$ steps.

In fact there is a theorem that we cannot decide A in $O(n)$ steps with a 1-tape Turing machine. If we can do a problem with $O(n)$ steps on a 1-tape Turing machine, then it is a regular language! (This is not obvious.) If the machine can only use order n time, then the only thing it can do is regular languages: the ability to write doesn't help.

In fact, anything that takes time $o(n \log n)$ must be a regular language as well.

§2 Time Complexity: formal definition

Which model should we pick to see how much time it takes? In computability theory we had *model independence*, the Church-Turing Thesis. Any model that we pick captures the same class of languages. Unfortunately, in complexity theory we have *model dependence*. Fortunately, for reasonable models, the dependence is not very big. Some interesting questions don't depend (much) on the choice of "reasonable" models.

In the meantime, we'll fix a particular model, set things up using that model, and show things don't change too much if we choose a different model. For convenience we'll choose the same model we had all along, a 1-tape Turing machine, then show that it doesn't change too much for other models.

Definition 12.3: For $t : \mathbb{N} \rightarrow \mathbb{N}$, we say a Turing machine M runs in time $t(n)$ if for all inputs w of length n , M on w halts in at most $t(n)$ steps.

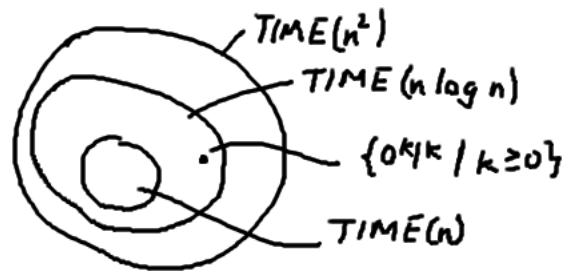
For instance, we say M runs in n^2 time if M always halts in n^2 steps when we give it an input of length n .

We now define the class of languages we can do in a certain number of steps.

Definition 12.4: Define

$$\text{TIME}(t(n)) := \{A : \text{some TM decides } A \text{ and runs in } O(t(n)) \text{ times}\}.$$

This is called a **time complexity class**.



We defined the time complexity using 1-tape turing machines. For a 2-tape TM, what is in the classes could change. Once we draw the picture, we can ask: is there a language we can do in n^2 time but can't do in $n \log n$ time? We'll look at questions like this later on, and be able to answer of them.

2.1 Polynomial equivalence

Even though the theory depends on model, it doesn't depend too much. This comes from the following statement.

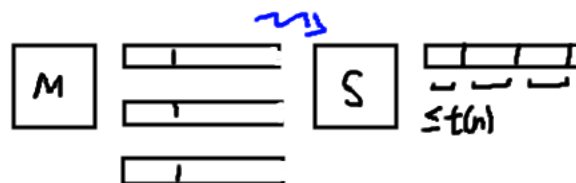
Theorem 12.5: Let $t(n) \geq n$. Then every multi-tape TM M that runs in $t(n)$ time has an equivalent 1-tape Turing machine S that runs in order $O(t^2(n))$ time.

In other words, converting a multi-tape TM to a single tape TM can only blow up the amount of time by squaring; the single tape TM can polynomially simulate the multi-tape TM.

You might think this is bad, but for a computer, this is not too bad. It could be worse (exponential).

Proof. We analyze the standard simulation (from the proof of Theorem 6.7).

The conversion only ends up squaring the amount of time used. Indeed, it took the tapes and wrote them down next to each other on the tape. Every time the multitape machine M did one step, the single-tape machine S had to do a lot of steps, and then do an update. One step of M might have S pass over entire portion of tape. *Each tape can be at most $t(n)$ symbols long, because there are only $t(n)$ steps where it can write symbols.* There are a constant number of tapes. *Thus one pass at most $O(t(n))$ steps.* *The machine has make at most $t(n)$ passes. Thus the order is $O(t(n)^2)$.*



□

Here is an informal definition.

Definition 12.6: Two computational models are **polynomially equivalent** if each can simulate the other with at most polynomial increase ($t(n)$ can go to $O(t(n)^k)$ for some k).

All reasonable deterministic models of computation turn out to be polynomially equivalent. This is the complexity analogue of the Church-Turing Thesis.

Axiom 12.7 (Church-Turing Thesis): **church-turing-complexity** All reasonable deterministic models are polynomially equivalent.

This includes one-tape TM's, multi-tape TM's, 2-dimensional TM's, and random access machines (which are closer to a real computer) which can write an index and grab the memory cell at that location (the address).

A real computer is a messy thing to discuss mathematically. It doesn't have infinite amount of memory. From some points of view, it is like a finite automaton. The most useful way to abstractify it is as a random access machine (RAM) or a parallel RAM (PRAM). If the machine only has polynomial parallelism, then it is also polynomially equivalent.

The analogous question with nondeterministic TM's is hard. No one knows a polynomial simulation. It is a famous open problem whether we convert a nondeterministic TM to a deterministic TM with a polynomial increase in time.

2.2 P

The complexity version of the Church-Turing Thesis 12.7 tells us the following.



All reasonable deterministic models are polynomially equivalent. Thus, if we ignore polynomial differences, we can recover a complexity class independent of the model.

Definition 12.8: Let

$$P = \bigcup_k \text{TIME}(n^k) = \text{TIME}(\text{poly}(n)).$$

In other words, P consists of all languages solvable in $O(n^k)$ time for some k . Why is P important?

1. The class P is invariant under choice of reasonable deterministic model. Time classes change when we go from 1-tape to multi-tape TM's. But by using polynomial equivalence—taking the union over all $O(n^k)$ —the class P is not going to change from model to model. We get the same class P .

Mathematically speaking, this invariance is natural. P not a class to do with Turing machines. It's to do with the *nature of computation*.

2. Polynomial time computability roughly corresponds to *practical computability*. It is a good litmus test: a good way of capturing what it means for a problem to be solvable practically.

Of course, practicality depends on context. There is a continuum between practical and impractical algorithms, but polynomial computability is a good dividing line.

One feature of P makes it mathematically nice, and one feature tell you something practical to real world. A math notion with both these aspects is very good.

This is why P is such an influential notion in complexity theory and throughout math.

2.3 Examples

Let's look at something we can solve in polynomial time. Let

$$\text{PATH} = \{ \langle G, s, t \rangle : G \text{ is a directed graph with a path from } s \text{ to } t \}.$$

Theorem 12.9: $\text{PATH} \in P$.

The way to prove something like this is to give an algorithm that runs in polynomial time.

Proof. “One input $\langle G, s, t \rangle$,

1. Mark a node s .
Repeat until nothing new is marked:
 - Mark any node pointed to by previously a marked node.
2. Accept if t is marked and reject if not.

□



We start at s , mark everything we can get to in 1 step by marking nodes adjacent to s ; then we mark nodes adjacent to *those*... This is a simple breadth-first search, not the best, but it runs in polynomial time.

We will often omit time analyses unless it is not obvious. If each step runs in polynomial time, and all repetitions involve a polynomial number of repeats, then the problem is solvable in P .

If we look at a similar problem, however, everything changes.

Definition 12.10: A **Hamiltonian path** goes through every node exactly once.

Is $\text{HAMPATH} \in P$? The algorithm above doesn't answer this question. It's a decidable problem because we can try every possible path, but there can be an exponential number of paths (in terms of the size of the graph).

The answer is not known! This is a very famous unsolved problem.

Lecture 13

Tue. 10/23/12

Absent because of sickness.

Lecture 14

Tue. 10/30/12

Last time we talked about

- $\text{NTIME}(t(n))$
- NP

Today we'll talk about NP-completeness.

§1 P vs. NP

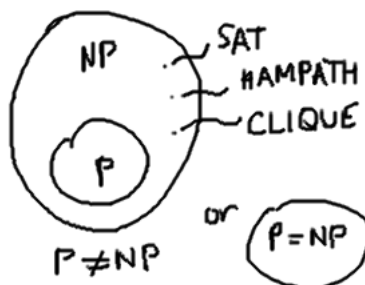
Recall that P is the class of problems (languages) where we can *test* membership quickly (in polynomial time in the size of the input). NP is the class of problems where we can *verify* membership quickly. We verify via a “short certificate” or a “short proof.” The verifier would be convinced that the string is in the language. Hamiltonian path is a good example: membership is easily verified by giving the path. Nonmembership is trickier: No one knows whether there is a way to exhibit short proof for non-existence of a Hamiltonian path. The complement of HAMPATH is not known to be in NP.

We can always flip the answer in P. However, we can't do so easily in NP: the acceptance structure can't be complemented easily in nondeterministic Turing machine.

! The complement of a language in P is in P ($\text{coP}=\text{P}$). However, the complement of a language in NP may not be in NP, because a NTM *can't easily do the opposite* of what another NTM does.

The big problem in theoretical computer science is **P versus NP**. Most people believe $\text{P} \neq \text{NP}$: there is a larger class of languages that can be verified in polynomial time than can

be solved in polynomial time. The other alternative is that $P = NP$. We've seen that SAT, HAMPATH, CLIQUE, etc. are in NP.



This problem was posed in the early 1970's, though it had precursors in the literature 10–15 years prior. There is an amazing letter Kurt Gödel sent to John von Neumann in 1955–1956 about the problem, using different language: Do we have to look for proofs by brute force or is there some quicker way? The problem has spread outside the computer science community to the math community. P vs. NP is one of Millenium problems, put together by a committee in 2000 as the analogue to Hilbert's problems in 1900. Langton Clay put in prize money for a solution: one million dollars.

§2 Polynomial reducibility

Early progress on the P vs. NP problem gave the amazing theorem.

Theorem 14.1: thm:sat-np $SAT \in P$ iff $P = NP$.

This would be important in a proof of the $P \stackrel{?}{=} NP$ problem. It might seem that you have to find an algorithm for all NP problems. If you believe $P = NP$, all you have to do is find an algorithm for SAT. On the flip side, to show $P \neq NP$, all you have to do is pick one problem and show it's in NP but not in P. But you might pick the wrong problem, for instance compositeness (primality testing), which is actually in P. This theorem tells you you can just focus on SAT.

This is an application of the theorem to understanding the P vs. NP problem. If you think of problems in P as being easy, and problems outside being hard, and if you assume that $P \neq NP$, then this theorem tells you that SAT is not easy. This gives evidence that SAT does not have a polynomial time algorithm.

Enough philosophical musings; let's do math. We'll work our way towards the proof of Theorem 14.1 today and finish next time.

We use a notion that we've seen before—reducibility.

Definition 14.2: A is **polynomial time mapping reducible** to B ($A \leq_P B$) if $A \leq_m B$ (A is mapping reducible to B) and the reduction is computable in polynomial time.

In other words, the thing that does the mapping can be done quickly. Not only can you translate A -questions to B -questions, you can do so by a polynomial time algorithm.

Just as we proved Proposition 9.7, we can show the following.

Theorem 14.3: If $A \leq_P B$ and $B \in P$, then $A \in P$.

Let's do an example.

2.1 An example: 3SAT reduces to CLIQUE

Example 14.4: $3SAT \leq_P CLIQUE$.

Recall that

$$SAT = \{ \langle \phi \rangle : \phi \text{ is a satisfiable Boolean formula} \}.$$

In other words, it is the set of statements ϕ that is true, under some truth assignment to its variables.

It's convenient to consider Boolean formulas in a special form, 3CNF (conjunctive normal form). This means the formula looks something like

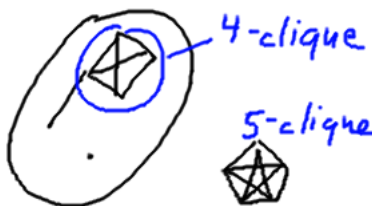
$$(x \vee y \vee \bar{z}) \wedge (\bar{x} \vee w \vee \bar{y}) \wedge \cdots \wedge (\bar{u} \vee \bar{w} \vee \bar{x}).$$

It is written as a bunch of clauses and'd together, and each clause is an "or" of 3 literals (variables or negated variables). That's all we're allowed to do. The "3" means that we have 3 variables in each clause. Thus we see this is a special case of the SAT problem, which we call 3SAT.

$$3SAT = \{ \langle \phi \rangle : \phi \text{ is a satisfiable 3CNF formula} \}.$$

We'll focus on the 3SAT problem and the CLIQUE problem.

The CLIQUE problem is very different. Given an undirected graph with nodes and edges, a k -clique is k vertices all connected to one another.



Define

$$CLIQUE = \{ \langle G, k \rangle : G \text{ contains a } k\text{-clique} \}.$$

I'm going to give a way to convert problem about whether or not a formula is in the 3SAT language to whether a graph contains a k -clique. This is surprising! We'll see that such conversions (reductions) are not just an interesting curiosity, but very important.

We'll do a proof by example. Suppose

$$\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4) \wedge \cdots \wedge (\cdots).$$

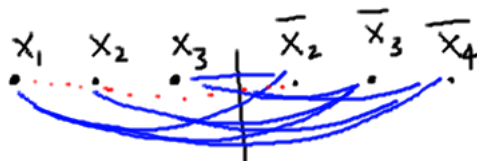
A satisfying assignment is an assignment that makes the whole thing true. Because ϕ is made up of clauses and'd together, each clause has to be true. What does it mean for each clause to be true? We have to make at least one of the literals true.

We have to pick out one literal and make it true. Thinking of the problem this way will be helpful to understanding the reduction to the CLIQUE problem.

We will now convert ϕ to $\langle G, k \rangle$. We will have one node for each literal variable. It's helpful to think of each node as being labeled by the associated literal. Now we put in the edges. We put in all possible edges with two exceptions.

1. Don't put edges inside a clause (internal to one of the triples associated to a clause). Thus edges can only go from one clause to another clause.
2. Never join two nodes that are associated to contradictory labels.

All other edges will be there.



As long as two literals are not contradictory in different clauses, they are connected by an edge.

Let k be the number of clauses.

We just have to show that this is actually a reduction. That this can be done in polynomial time is clear: by looking at the formula, we can easily write down the graph.

We have to show two directions. Now is where the interesting stuff happens; we'll understand what's going on; why did we draw this strange graph?

1. $\phi \in 3\text{SAT} \implies \langle G, k \rangle \in \text{CLIQUE}$.

Suppose ϕ is 3-satisfiable; we have to exhibit a k -clique. Each clause has at least one true literal. Pick out a true literal in each clause. Maybe the assignment makes x_2 true. Obviously it cannot make x_2 true; maybe it makes \bar{x}_3 true. Now pick out the associated nodes.

I claim those nodes form a clique. I have to show that every pair of nodes I've picked are connected by an edge. We put in all possible edge with 2 exceptions. We have to show we don't run into any of the exceptions.

1. We only pick 1 node from each clause.

2. We never pick two nodes with contradictory labels. We can't pick two nodes with contradictory labels because they can't be both true; we could not have picked both of them as the *true* literal in the clauses. One will be true and the other false in any assignment.

We started with the certificate from 3SAT and produced a certificate for CLIQUE.

2. $\phi \in 3SAT \Leftrightarrow \langle G, k \rangle \in CLIQUE$.

Now we start with a k -clique. We reverse the argument. Look at the nodes we picked out as being in the same clique. Every node has to be from a different clause, because nodes in the same clause are not connected (1). Since there are k clauses, we took one node from each clause.

Take the nodes in the clique and let the corresponding literal be true. For instance, if x_2 and \bar{x}_3 are in the clique, make x_2 true and \bar{x}_3 true, i.e., x_3 false. If a variable is unassigned, assign any which way. How do we know we didn't run into trouble? We won't assign a variable true and its complement true, because contradictory nodes can't be in the same clique (2).

This gives at least one 1 node in each clause.

We're done but we had to show both directions.

This means that if we find a polynomial time algorithm for CLIQUE, then we can solve 3SAT quickly. We can convert 3SAT into a special CLIQUE problem. If you can solve general CLIQUE problems, then you can solve these special CLIQUE problems too, using our magical polynomial time algorithm to CLIQUE.

Let's lay out our game plan. We'll show next lecture that every NP problem can be reduced to SAT. We'll show

$$SAT \leq_P 3SAT \leq_P CLIQUE, HAMPATH, \dots$$

(we just did $3SAT \leq_P CLIQUE$). What we did for 1 problem we'll have to do for infinitely many problems. We'll use the Boolean logic of SAT to simulate a Turing machine. This is similar to the proof of undecidability of PCP: we use combinatorial structure to simulate a Turing machine.

Note that polynomial time reducibility is preserved by composition (exercise).

§3 NP completeness

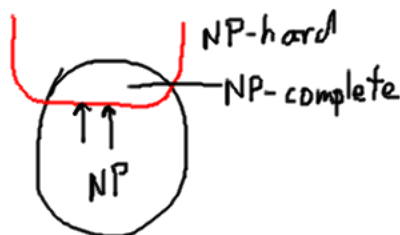
We have a special name for problems that *every* NP problem can reduce to.

Definition 14.5: A language B is **NP-complete** if

1. $B \in NP$.
2. For every $A \in NP$, $A \leq_P B$ (A is reducible to B in polynomial time).

If we can reduce everything else in NP to B , then B is a NP-complete problem. Condition 2 by itself is called **NP-hard**. Rephrasing, B is NP-complete if $B \in \text{NP}$ and is NP-hard. (A problem that is just NP-hard may be worse than NP.)

The picture is that NP-complete problems are at the “top” of the NP problems:



Proving the non-existence of reductions within NP is tricky business. A common question is to give an example of NP problem which is not NP-complete. But if $P = \text{NP}$, then all problems in NP are reducible to each other, essentially. If you can prove some NP problem is not reducible to another NP problem, then you have a good result—you’ve just shown $P \neq \text{NP}$. We’re not going to show that in class. Otherwise, I’d be off celebrating somewhere in the Caribbean.

There is a special analogy between P and decidability and NP and recognizability. One key element is not in place, though. We don’t know whether the classes are different. Still, there are a lot of similarities.

As we will show, everything is reducible to SAT, so SAT is NP-problem (Cook-Levin Theorem).

Theorem 14.6 (Cook-Levin): **thm:cook-levin** SAT is NP-complete.

(This is equivalent to Theorem 14.1.)

By composition of reductions, if SAT reduces to some other problem, that problem is also a NP-problem. This will show that 3SAT, CLIQUE, HAMPATH, etc. are also NP-complete, provided that we have the reductions.

Key Because 3SAT is NP-complete, to show another problem is NP-complete, you just have to do things:

- Show it is in NP.
- Give a polynomial-time reduction from 3SAT to NP: $3\text{SAT} \leq_P \text{NP}$.

When we’re doing reductions, we’re trying to find a way to simulate Boolean variables with structures in the target problems.



To reduce from one 3SAT to another language, design features or structures that have the same kind of feature as a variable or clause in 3SAT. (Think of this as “learning to program” using CLIQUE, HAMPATH, etc. languages/) These features are called *gadgets*, substructures in the target language which operate in the same way a variable or clause do.

The best way to understand this is through example.

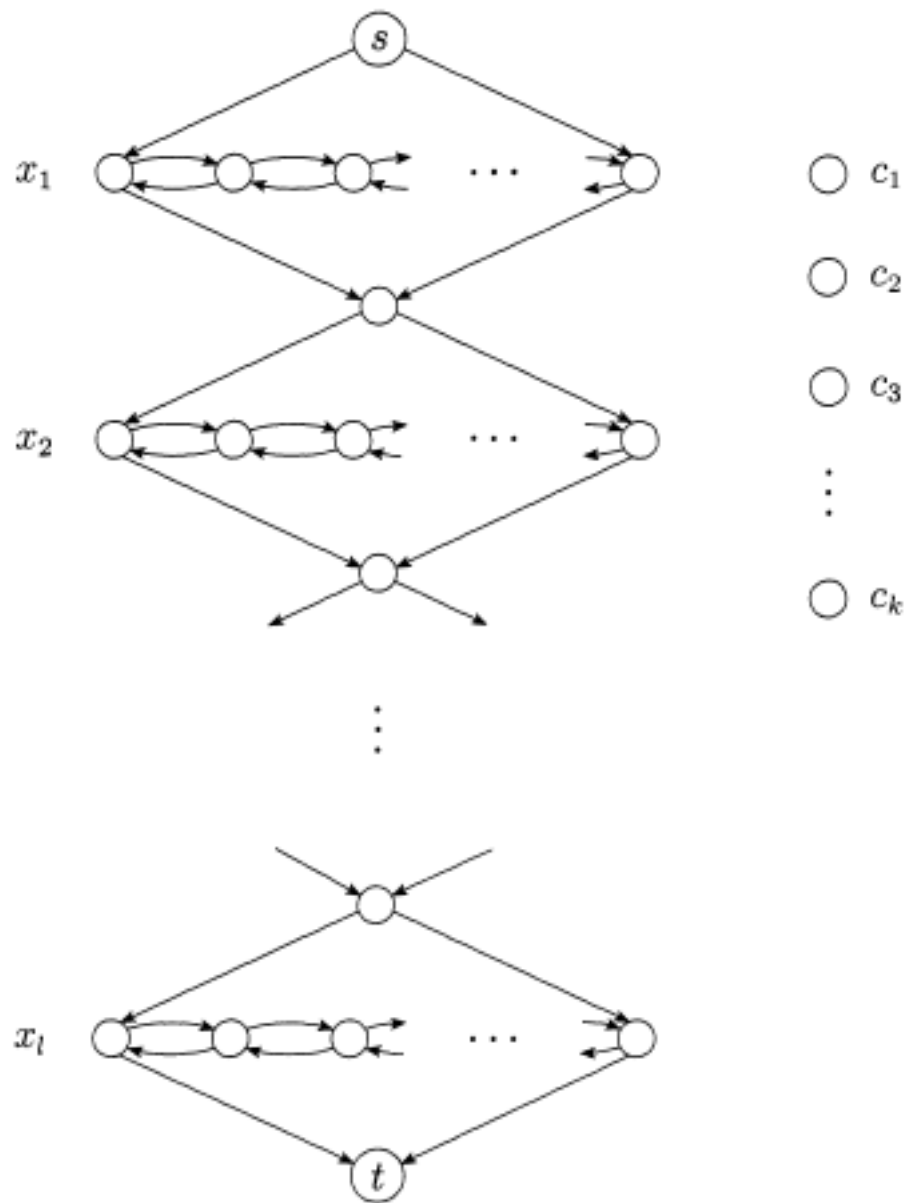
Theorem 14.7: $3\text{SAT} \leq_P \text{HAMPATH}$.

Proof. Start with a 3CNF, say $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4) \cdots$. We construct $\langle G, s, t \rangle$. We build a graph that has a Hamiltonian path in it exactly when ϕ is satisfiable. (fig 6).

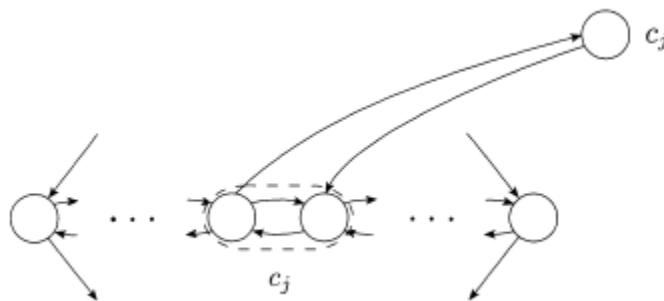
We put in a bunch of nodes; all edges are directed downwards or horizontally. The diamond structures will be associated to the variables, there will be one structure corresponding to each variable (a bit different from last time, where we had one structure for each appearance of a literal). The bottom node of a diamond is the same as the top node of the next. For each diamond we have horizontal connections.

We have a hamiltonian path right now. For each diamond we could zig-zag or zag-zig independently through each of the variable gadgets; we pick up all the nodes, and there’s nothing else we could do. Zig-zag is going to correspond to “true” and zag-zig is going to correspond to “false.” The Hamiltonian path is going to correspond to the truth assignment.

An important feature we haven’t done yet is the clauses. We have to have an assignment which makes one literal in each clause true. We let each clause gadget be a node. A Hamiltonian path has to go through each. If $x_1 \in C_1$ (clause 1), then we put in arrows like in the diagram, allow a detour to visit x_1 if we’re zig-zagging (going from left to right in a diamond) ,but not if we’re zag-zigging (going from right to left in a diamond): (figure from textbook)



This corresponds for x_1 being a positive literal. How do we implement the fact that $\bar{x}_3 \in C_1$? We allow the detour only to go in the right-to-left direction.



We leave a space before putting the next node, to give an opportunity to make several detours.

Suppose an assignment has 2 true literals in some clause C_1 . But that gives 2 detours to C_1 . We can only visit C_1 once. Is that a problem? No. A detour is an option—it's not a broken-road detour, it's a rest-stop type detour, if you don't have to go, don't.

We have to prove that if we have a satisfying assignment, then we have a Hamiltonian path. We zig-zag or zag-zig according to assignment, visit all detours.

For the converse, if the path is nice (consisting of zig-zag and zag-zigs), then we get a satisfying assignment, and we're done. If the path is not nice, i.e., it goes to a different diamond from one it came from at some stage, then the path cannot be Hamiltonian because of the spacer nodes. \square

Lecture 15

Thu. 11/1/12

Last time we talked about

- NP-completeness
- $3SAT \leq_P CLIQUE$
- $3SAT \leq_P HAMPATH$

Today we'll prove the Cook-Levin Theorem: SAT is NP-complete.

We have

(Every NP problem) $\leq_P SAT \leq_P 3SAT \leq_P CLIQUE, HAMPATH$, many others

We'll show the first inequality today and the second inequality in recitation. We know every problem on the right is NP-complete. (We don't necessarily have to start with SAT or 3SAT. Sometimes it's easier to study another NP-complete problem. For instance, to show UHAMPATH, the undirected version of Hamiltonian path, is NP-complete, we can just reduce the directed to the undirected version, $HAMPATH \leq_P UHAMPATH$.)

If we assume $P \neq NP$, and if we show a problem is NP-complete, then it cannot be solved in polynomial time. Thus being NP-complete is very strong evidence for intractability: the problem is too hard to solve in practice. What is remarkable (and not well understood) is that typical problems in NP, with few exceptions, turn out to be in P or NP-complete. This is mysterious and currently has no theoretical basis.

Thus, given a problem, researchers often spend part of the time showing it's solvable in polynomial time and part of the time showing it's NP-complete. This works well most of the time.

There are some problems, though, that seem to be outside of P, but we don't know how prove they are NP-complete. For instance, the problem of testing if 2 graphs are isomorphic—whether they are the same graph but labeled differently—is NP: the short proof is the mapping of the vertices. No one knows whether the graph isomorphism problem is solvable in polynomial time, nor has anyone shown it is NP-complete. It's one of few problems that seem to be hovering in between. Another example is factoring integers.

Define

$$\text{CoNP} = \{\bar{A} : A \in \text{NP}\}.$$

We have $P \subseteq \text{NP} \cap \text{CoNP}$. (P is closed under complement so $P = \text{coP}$. It's generally believed that NP-complete problems cannot be in coNP because otherwise $\text{NP} = \text{coNP}$.)



There are problems in the intersection, for instance, factoring is a problem in $\text{NP} \cap \text{coNP}$. Naively it's a function, but we can turn it into a decision problem. Think of numbers as written in binary, and call it the bit factoring problem:

$$\text{BIT-Factoring} = \{\langle x, i \rangle : i\text{th bit of largest prime factor of } x \text{ is } 1\}.$$

BIT-Factoring is in NP because nondeterministically we can guess the prime factorization of x and check that the largest prime factor has a 1 in the i th place.

The complement is also a NP-problem: The i th bit is a 0. We can check that in exactly the same way.

If BIT-Factoring is in P, then we can factor numbers in polynomial time. We believe that factoring is not in P, so this problem seems to not be in P. This suggests the problem is not NP-complete.

§0 Homework

The first four questions are clear. For one of them keep in mind dynamic programming as a technique. (Context-free languages are testable in polynomial time. It is in a sense the most basic polynomial time algorithm.)

Problem 5 asks you to show that under the assumption $P=NP$, there exists an algorithm that operates in polynomial time which not only test whether a statement is satisfiable, but produce the satisfying assignment. A tempting algorithm is that there is a nondeterministic algorithm which finds the assignment, and because $P=NP$, there is a deterministic algorithm which finds the assignment. But it is conceivable that the polynomial time algorithm for satisfiability operates not by finding the assignment, but saying whether it is satisfiable.

You have to show that if the program operates by some other way, you can turn it into an algorithm to find the assignment.

In order to produce a satisfying assignment, you will end up testing whether multiple formulas are satisfiable. Out of the decisions from the tests, you can assemble the satisfying assignment to the original formula. How can you at least get a little *bit* of information about the satisfying assignment?

Problem 6 says that minimizing NFA's cannot be done unless $P=NP$. By contrast, it is known that the conversion for DFA can be done in polynomial time.

§1 Cook-Levin Theorem

Theorem 15.1 (Cook-Levin, Theorem 14.6 again): **thm:cook-levin2** SAT is NP-complete.

Proof.

1. $SAT \in NP$: This is easy: guess a satisfying assignment.
2. Let $A \in NP$. We have to show $A \leq_P SAT$. Assume we have a NTM M for A so that M runs in n^k time.

The idea is as follows. We have to give a polynomial time reduction $f : A \rightarrow SAT$. It will take a string w and convert it to some formula ϕ_w . The function f maps a membership question in A to a membership question in SAT; we will have $w \in A$ exactly when ϕ_w is satisfiable.

$$\begin{aligned} f : A &\rightarrow SAT \\ w &\mapsto \phi_w \\ w \in A &\text{ iff } \phi_w \text{ is satisfiable.} \end{aligned}$$

Think of ϕ_w as saying whether M accepts w .

The construction of ϕ_w as follows. It will be in 4 pieces and'd together:

$$\phi_w = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

We'll describe the computation of M on w in a certain way.

Define a **tableaux** for M on w to be a table where the rows are configurations of M on w . Write down the tape with the head symbol to the left of the symbol it's looking at (cf. the PCP proof 10.1). Each row is a configuration. The sequence of rows you

get is a computation history. Remember M is nondeterministic, so there may be multiple computation histories. If M accepts w , there is an accepting branch, and we can write down an accepting computation history with the starting configuration at the top and the accepting configuration at the bottom.

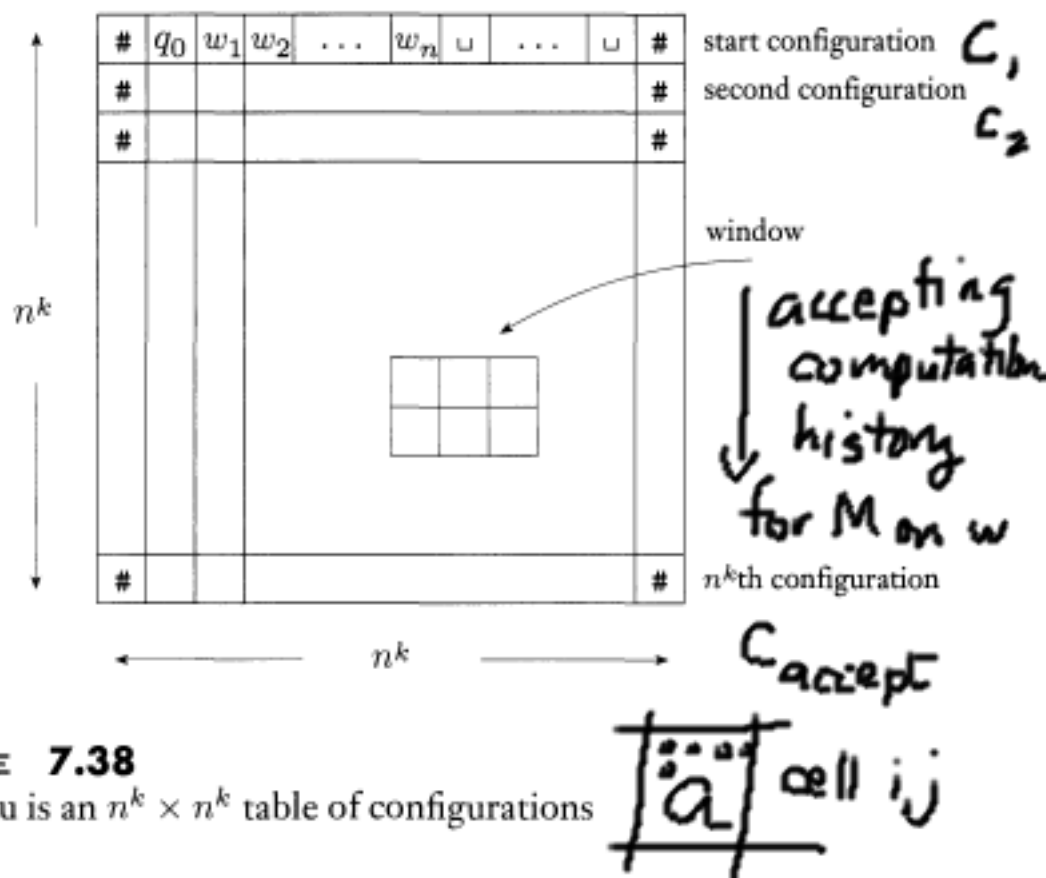


FIGURE 7.38

A tableau is an $n^k \times n^k$ table of configurations

Does there exist such a tableaux? If M does not accept w there is no accepting computation history so there is no tableaux. The question we're trying to answer is whether a tableaux exists.

We're trying to make a formula which says a tableaux exists. Is there some way of setting cells to symbols such that the whole thing is a legitimate tableaux? We make indicator variables for each cell: think of each cell as having a bunch of little lights; one light for each possible setting the cell could be: a , b , q_0 , etc. If the light for a is on, then the cell has an a in it.

The variables of ϕ_w are $x_{ij\sigma}$ where $1 \leq i, j \leq n^k$ (we're assuming the machine runs for n^k steps; the most number of cells it could use is n^k)¹¹ and $\sigma \in \Gamma \cup Q$ (σ is in the tape alphabet or σ is a state). There are $|\Gamma \cup Q|n^{2k}$ variables $x_{ij\sigma}$, which is polynomial in n .

¹¹Technically we may need cn^k just to cover $n = 1$ but this is a minor issue.

ϕ_{cell} : In order for variables to correspond to valid tableaux, exactly 1 cell per symbol has to get assigned. If we turn on several lights for some cell, this would correspond to multiple symbols, and we don't want that. We have to make sure we're turning on exactly one light; exactly one variable becomes true for each (i, j) . This is the first piece ϕ_{cell} .

ϕ_{cell} says that there is exactly one symbol per cell or equivalently, exactly one $x_{ij\sigma}$ is true for each i, j :

$$\phi_{\text{cell}} := \bigwedge_{1 \leq i, j \leq n^k} \left(\bigvee_{\sigma \in \Gamma \cup Q} x_{ij\sigma} \wedge \bigwedge_{\sigma \neq \tau} \overline{x_{ij\sigma}} \vee \overline{x_{ij\tau}} \right).$$

The first formula ensures that one of these lights is “on,” and the second ensures that at most one of the lights is on (for every pair of lights which are not the same, at least one of them is on). Together they say exactly 1 variable is true. The assignment has to correspond to one symbol in each cell of the tableaux.

ϕ_{start} : Now we want to say in the very first row, the variables are set to be the start configuration. ϕ_{start} says that the start configuration is

$$\underbrace{q_0 w_1 w_2 \cdots w_{n^k} \cdots}_{n^k}$$

Hence we let

$$\phi_{\text{start}} = x_{11q_0} \wedge x_{12w_1} \wedge x_{13w_2} \wedge \cdots \wedge x_{1,n+1,w_n} \wedge x_{1,n+2,\sqcup} \wedge \cdots x_{1,n^k,\sqcup}.$$

ϕ_{accept} : Now let's do ϕ_{accept} . The very last row is an accepting configuration; namely the machine is in the accept state. (What if the machine stops sometime earlier? We assume that the rules of the machine say it stays in the accepting state for the “pseudo-steps” afterward.) We let

$$\phi_{\text{accept}} = \bigwedge_{1 \leq j \leq n^k} x_{n^k j q_{\text{accept}}}.$$

ϕ_{move} : Finally, we need to say the machine moves completely. To do this out in full gory detail is a bit of a mess (like the PCP problem). I'll just convince you that you can do it.

We pick out a 2×3 neighborhood, or **window** from the tableaux, and specify what it means for it to be a legal neighborhood. **figure 3** For any given setting of symbols in the 2×3 neighborhood, we can ask whether it could possibly arise according to the rules of the machine. There are certain legal settings and certain illegal settings. For instance if when in state q_3 and the machine reads an a , writes c , moves to the right, and goes to state q_5 in a possible nondeterministic step, then

q_3	a	b
c	q_5	b

is legal, whereas

q_3	a	b
c	q_5	d

is illegal.

There are some subtleties, for instance,

a	b	c
d	b	c

may be a state with where the head changed a (the head being to the left of a), but something like

a	b	c
a	d	c

is never possible. By looking at the transition function of M , we can determine which of the 6-symbol settings are legal and which are not. We need to check whether every single window is legal. If every single window is legal then all moves are legal.

This depends critically on the window being 2×3 . If it were just a 2×2 window it wouldn't work. The tableaux can be globally wrong but locally right if we only look at 2×2 windows. If the machine is in state q_2 , and it can go to q_3 and go left, or q_5 and go right, then you have to make sure you exclude things like

a	q_2	a
q_3	a	q_5

A 2×3 window just big enough to catch this; this is the only thing that can go wrong.

Thus we let

$$\phi_{\text{move}} = \bigwedge_{1 \leq i, j \leq n^k} (i, j \text{ neighborhood is legal}),$$

i.e., more precisely,

$$\phi_{\text{move}} = \bigwedge_{1 < i < n^k, 1 \leq j < n^k} \bigvee_{\begin{array}{|c|c|c|} \hline a & b & c \\ \hline d & e & f \\ \hline \end{array} \text{ is legal}} x_{i-1,j,a} \wedge x_{i,j,b} \wedge x_{i+1,j,c} \wedge x_{i-1,j+1,d} \wedge x_{i,j+1,e} \wedge x_{i+1,j+1,f}.$$

We “or” over all possible ways to set cells to symbols to get a legal window. That can be a lot but it's a fixed number.

We have 2 things that remain: first, we need to show this is correct, i.e., w is in the language iff ϕ_w satisfied. Now w being in the language means there is some accepting computation history, i.e., some valid tableaux, i.e., some setting of variables that satisfies ϕ_w . This should be clear from the construction. The pieces of the formula are designed to force the variables to be set according to some valid accepting tableaux.

We also have to check the reduction can be done in polynomial time. This is easy to confirm. First, how large is ϕ_w ? Ignoring constant factors, the size is about as large as the number of cells in the tableaux, which is polynomial in n . Actually, writing down the formula can be done in about the same time as the size of the formula. The steps themselves are simple. It's just a lot of output, but still polynomial. The actual thinking to produce the output is simple. \square

§2 Subset sum problem

Let's look the subset sum problem:

$$\text{SubSum} = \{(a_1, \dots, a_k, t) : \text{some subset of } a_1, \dots, a_k \text{ sums to } t\}.$$

This is a NP-problem because you can just guess the subset that sums to t .

Theorem 15.2: The subset sum problem is NP-complete.

Proof. We show that 3SAT reduces to SubSum. Suppose we are given a 3-cnf $\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\dots) \dots (\dots)$. How do we make gadgets in SubSum but simulate the variables and clauses of the 3SAT problem?

In the choice of what the subset looks like, there are some binary choices: pick or not pick. We want to make them correspond to binary choices for the variables.

A binary choice is whether or not a_1 in the subset. We modify this a bit. x_1 set to true or false is somehow symmetrical. a_1 being in the subset or not is less symmetrical. We'll do something in the same spirit. Each variable represented is represented by 2 values. The target sum is designed in such a way so that exactly one value has to appear in the subset.

Here's the construction. We'll write the values in decimal. Having 1's in t forces exactly one of a_1, a_2 to appear, and similarly for each pair a_{2k-1}, a_{2k} . a_1, a_2 is the x_1 gadget, a_3, a_4 is the x_2 gadget, and so forth; a_1 corresponds to x_1 true and a_2 corresponds to x_1 false, and so forth. In the table below, we write a_{2k-1}, a_{2k} as y_k, z_k . We have columns corresponding to each clause, and put 1's in cells when the literal corresponding to the row is in the clause corresponding to the column.

	1	2	3	4	...	l	c_1	c_2	...	c_k
y_1	1	0	0	0	...	0	1	0	...	0
z_1	1	0	0	0	...	0	0	0	...	0
y_2		1	0	0	...	0	0	1	...	0
z_2		1	0	0	...	0	1	0	...	0
y_3			1	0	...	0	1	1	...	0
z_3			1	0	...	0	0	0	...	1
\vdots					\ddots	\vdots	\vdots		\vdots	\vdots
y_l						1	0	0	...	0
z_l						1	0	0	...	0
g_1							1	0	...	0
h_1							1	0	...	0
g_2								1	...	0
h_2								1	...	0
\vdots									\ddots	\vdots
g_k										1
h_k										1
t	1	1	1	1	...	1	3	3	...	3

Now we put 2 extra 1's in each column. If there are no 1's in the formula part, then we are not going to get 3. If we have at least 1 in the formula part, then we can add 1's to get 3, and we are done. \square

Lecture 16

Tue. 11/6/12

We're going to shift gears a little bit. Having finished our discussion of time complexity—the number of steps it needs to solve one problem—we're going to look at how much memory (space) is needed to solve various problems. We'll introduce complexity levels for space complexity analogous to time complexity, and complete problems for these classes.

Last time we proved the Cook-Levin Theorem: SAT is NP-complete.

Today we'll do

- space complexity

- $\text{SPACE}(s(n))$, $\text{NSPACE}(s(n))$
- PSPACE, NPSPACE
- Examples: TQBF, $\text{LADDER}_{\text{DFA}}$
- Savitch's Theorem.

§0 Homework

Problem 1:

On exponentiation modulo a number. We can do the test even though the numbers are very big, say all n -bit numbers. The naive algorithm—just multiplying over and over—takes exponential time, because the magnitude of the number is exponential in the size of the number.

If you want to raise a number to the 4th power, you can multiply it 3 times or square it twice. Using this squaring trick you can raise number to high powers, even if they are not powers of two.

There are real applications of raising numbers to powers in modular arithmetic, for instance, in cryptography.

Problem 2 (Unary subset sum problem):

A number in unary is much bigger to write down than, say, in binary. The straightforward algorithm—looking through all possible subsets—doesn't give a polynomial time algorithm because there are exponentially many subsets. Instead, use dynamic programming. The key observation is that you can ignore the target. Just calculate all possible values you can get by looking at the subsets. There are exponentially many subsets, but only polynomially many different values you can obtain for their sums. Think about how to organize your progress carefully. Dynamic programming gives you a way to organize your progress.

Problem 3:

This is an important problem. If $P=NP$, then everything in NP is NP complete. This is important for 2 reasons. This shows that proving a problem is not NP-complete is pretty hopeless. There can be no simple way of showing a problem not NP-complete, because then we get this amazing consequence $P \neq NP$.

The fact really comes from the fact that all problems in P are polynomial time reducible to one another. This is important to understand, because the issue comes up repeatedly in different guises. This is a nice exam-type question that can be asked in a variety of different ways.

This is similar to the fact that all decidable problems are mapping-reducible to one another. This is a basic concept to understand in putting together the theory the way we do it.

Problem 4:

The 3-coloring problem is NP-complete. The book gives gadgets you might use. The palette is a structure you might want to use in your reduction. If you imagine trying to color your graph in 3 colors, and you have this structure, the 3 colors must all appear in the palette. (The palette is like the set of colors the artist has to work with.) When you color the graph with 3 colors, we don't know what colors they are but we can arbitrarily assign them names, say True, False, and Red. Thinking of colors as truth values helps you understand the rest of the connection.

In the variable gadget, a node of the palette (the red node) happens to be connected to 2 other nodes connected to each other. If it is 3-colorable, then we know the 2 nodes are not red, so are either true-false or false-true. That binary choice mirrors the choice of truth assignment to some variable. That's why this structure is called a variable gadget. It has two possibilities.

But you have to make sure the coloring corresponds to satisfying assignment. That's what the other gadgets help you to do. Play with the or-gadget. Try assigning values at the bottom and see what values are forced elsewhere.

Problem 5:

If $P=NP$ then you can not only test formulas, you can find the assignment. Find the assignment a little bit at a time.

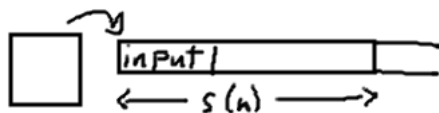
Problem 6 (Minimizing NFA's):

Find an equivalent automaton with the fewest number of states possible, equivalent to original one. For DFA's, there is a poly time algorithm. No such algorithm is known for NFA's. In fact, if you could do that then $P=NP$. Imagine what would happen if you could minimize the automaton you ended up constructing. That would turn out to be useful.

§1 Space complexity

1.1 Definitions

Definition 16.1: A Turing machine runs in **space** $s(n)$, where $s : \mathbb{N} \rightarrow \mathbb{N}$, if it halts using at most $s(n)$ tape cells on every input of length n , for every n .



The machine uses a tape cell if its head moves over that position on the tape at some time. We're only going to consider the case $s(n) \geq n$, so we at least read the entire input. The head has at least passed over the input; it might use additional space beyond the input. We assume the machine halts on input of every length.

This is entirely analogous to time complexity. There, instead of measuring space used, we measured time used.

We can define space use for deterministic and nondeterministic machines. For a nondeterministic machine to run in space $s(n)$, it has to use at most $s(n)$ in every branch. We treat each branch independently, seeing how many tape cells are used on that branch alone.

We now define space complexity classes.

Definition 16.2: Define

$$\begin{aligned}\text{SPACE}(s(n)) &= \{A : \text{some TM decides } A \text{ running in } O(s(n)) \text{ space}\} \\ \text{NSPACE}(s(n)) &= \{A : \text{some NTM decides } A \text{ running in } O(s(n)) \text{ space}\}.\end{aligned}$$

Think of these as the collection of languages some machine can do within $s(n)$ space.

1.2 Basic facts

Let's show some easy facts, some relationships between space and time complexity.

Proposition 16.3: For $s(n) \geq n$,

$$\text{TIME}(s(n)) \subseteq \text{SPACE}(s(n)).$$

This also works for NSPACE and NTIME.

Proof. Suppose we can do some problem with $s(n)$ time. Then there is a TM that can solve that problem with at most $s(n)$ steps on any problem of input length n . I claim that language is also solvable in space $s(n)$. If you can do something with $s(n)$ steps you can do it in $s(n)$ space, by using the same algorithm. The machine can only use at most $s(n)$ tape cells because in each additional step it uses at most 1 more tape cell. \square

Let's do containment in the other direction. Space seems to be more powerful than time: the amount of stuff doable in space n might take a lot more time.

Proposition 16.4: For $s(n) \geq n$,

$$\text{SPACE}(s(n)) \subseteq \text{TIME}(2^{O(s(n))}) = \bigcup_{c>0} \text{TIME}(c^{s(n)}).$$

This also works for NSPACE and NTIME.

Think of c as the size of the tape alphabet.

Proof. Consider a machine running in space $s(n)$.

It can't go on too long without repeating a configuration; if it halts it can't repeat a configuration. The number of configurations is at most exponential in $s(n)$, so the time is at most exponential in $s(n)$. \square

Definition 16.5: Define

$$\begin{aligned} \text{PSPACE} &= \bigcup_k \text{SPACE}(n^k) \\ \text{NPSPACE} &= \bigcup_k \text{NSPACE}(n^k) \end{aligned}$$

We define these because they're model independent like P and NP.

Corollary 16.6: $\text{P} \subseteq \text{PSPACE}$ and $\text{NP} \subseteq \text{NPSPACE}$.

Proof. This follows from $\text{TIME}(s(n)) \subseteq \text{SPACE}(s(n))$. □

The following starts to show you why space is more powerful than time.

Theorem 16.7: $\text{NP} \subseteq \text{PSPACE}$.

Now we have to do something nontrivial. All we know is that we have a nondeterministic polynomial time algorithm for the language. It's not going to tell you that you can decide the same language with a polynomial time algorithm on a deterministic machine.

Proof. 1. We first show $\text{SAT} \in \text{PSPACE}$: Use a brute force algorithm. You wouldn't want to write down the whole truth table. But you can cycle through all truth assignments one by one, reusing space to check whether they are satisfying assignments. If you go through all assignments and there are no satisfying assignments, then you can reject. The total space used is just enough to write down the current assignment. Thus $\text{SAT} \in \text{SPACE}(n)$.

2. If $A \in \text{NP}$ then $A \leq_P \text{SAT}$. The polynomial time reduction can be carried out in polynomial space. If you have an instance of a NP problem, then you can map it to SAT in polynomial time, and use the fact that the SAT problem can be done in polynomial space. □

This theorem illustrates the power of completeness. Note that we had to make sure the reduction is being capable of being computed by algorithms within the class (PSPACE). Then we showed a NP-problem is in PSPACE for a complete problem in that class (SAT), so we get that all problems reducible to it are also in that class. Thus the whole class (NP) becomes subset of class you're working with (PSPACE).

(You can also give a more direct proof.)

Theorem 16.8: $\text{CoNP} \subseteq \text{PSPACE}$.

Proof. When you have deterministic machines, and you want the complementary language, you can just flip the answer at the end. *Deterministic complexity classes are closed under complement.* Just solve the NP problem and take the complement. □

For instance, the unsatisfiability problem is in CoNP, hence is in PSPACE. We have $\overline{\text{HAMPATH}} \in \text{CoNP}$, hence is in PSPACE. In fact, UNSAT and $\overline{\text{HAMPATH}}$ are CoNP-complete.

1.3 Examples

Let's do a slightly less trivial example of a problem in PSPACE. Then we'll give an example of a problem in NPSPACE.

Example 16.9: Here is a Boolean formula:

$$(x \vee \bar{y}) \wedge (\bar{x} \vee y \vee z).$$

We put quantifiers in front. Quantifiers range over boolean values.

$$\forall x \exists y \forall z [(x \vee \bar{y}) \wedge (\bar{x} \vee y \vee z)].$$

This formula says: For every truth assignment to x there exists a truth assignment to y such that for every truth assignment to z the statement is true. This is a **quantified Boolean formula**. We assume every variable gets quantified.

We formulate the general computational problem as a language: **TQBF, true quantified Boolean formulas**.

$$\text{TQBF} = \{ \langle \phi \rangle : \phi \text{ is a true quantified Boolean formula} \}.$$

This problem is in a sense a generalization of satisfiabilities. The satisfiability problem is the special case where all quantifiers out front are \exists : is there a setting to all variables that makes the formula true.

TQBF seems to be harder. It is in polynomial space, but not known to be in NP. Why is it solvable in polynomial space?

It turns out TQBF is PSPACE-complete. We first have to show it's in PSPACE. This isn't too hard.

Theorem 16.10: thm:tqbf-pspace $\text{TQBF} \in \text{PSPACE}$.

Let's assume that you can plug in constant values (trues/falses) in certain locations.

Proof. Break into cases. On input $\langle \phi \rangle$,

1. If there are no quantifiers, then there are no variables, so evaluate ϕ and accept if true.
2. We give a recursion. If ϕ starts with $\exists x$, evaluate recursively for x true and false. Accept if either accepts.
3. If ϕ starts with $\forall x$, evaluate recursively for x true and false. Accept if both accept.

In this way the machine evaluates all possibilities while resusing the space!

This uses space $O(n)$. □

Now let's look at nondeterministic space complexity.

Here's a word puzzle: convert one word to another by changing one word at a time, and staying in the language. For instance, suppose we want to convert ROCK to ROLL. We can't convert ROCK to ROCL because ROCL is not an English word. It may be helpful to change the R to something else to enable us to change last letters: ROCK, SOCK, SULK, BULK, BULL, BOLL, ROLL. We'll consider a similar problem.

Define the language as the set of strings some finite automaton accepts.

Definition 16.11: df:ladder Define

$$\begin{aligned} \text{LADDER}_{\text{DFA}} \\ = \{ \langle B, s, t \rangle : \text{there is a sequence } s = s_0, s_1, \dots, s_k = t, s_i, s_{i+1} \text{ differ in one character, all } s_i \in L(B) \} \end{aligned}$$

What's the complexity of testing this? This problem is solvable in nondeterministic polynomial space.

Theorem 16.12: thm:ladder-npspace $\text{LADDER}_{\text{DFA}} \in \text{NPSpace}$.

Proof. (by example) Nondeterministically change one letter, check to see if the word is still in the language. We test at every stage if we ended up at ROLL. We have to be careful not end up in a loop. The machine cannot remember everything it's done. Instead, it counts how many words it has looked at so far. If the number is too high, it must have looped. □

1.4 PSPACE vs. NPSpace

There is a rather surprising general theorem that tells you PSPACE and NPSpace are the same. The analogue to P vs. NP for space complexity is solved.

$$\text{PSPACE} = \text{NPSpace}.$$

This is not obvious! If you try a backtracking algorithm in the obvious way, then it blows up the space to be exponential.

Is this is a NP problem? The certificate (ladder) could be exponentially long! The space is allowed to guess on the fly. The amount of steps is potentially exponential. It's not known to be in NP. (The input consists of the automaton, starting string, and ending string. The automaton is the not dominant piece; the starting and ending string are.)

§2 Savitch's Theorem

We have the following remarkable theorem.

Theorem 16.13 (Savitch): thm:savitch For $s(n) \geq n$,

$$\text{NSPACE}(s(n)) \subseteq \text{SPACE}(s(n)^2)$$

Corollary 16.14: $\text{PSPACE} = \text{NPSPACE}$.

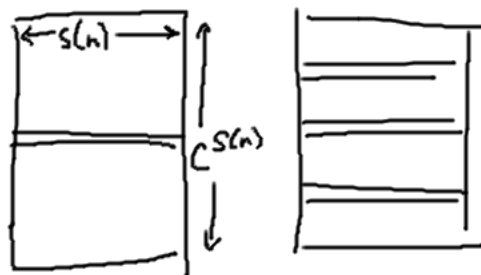
This is because we have just a squaring.

Proof. Given $S(n)$ -SPACE NTM N , we construct an equivalent TM M that uses $O(S^2(n))$ space.

Imagine a tableaux of N on w , corresponding to some accepting computation branch. This time, the dimensions are different: the width is $s(n)$, how much space we have and the height is $c^{S(n)}$ for some c . We want to test if there's a tableaux for N on w , but we want to do it deterministically.

Can we fill it in somehow? It's an exponentially big object, and we'll be in trouble if we have to keep it all in memory—we don't have that much memory.

The deterministic machine tries every middle configuration sequentially. (This takes a horrendous amount of time but we only care about space.)



For a start configuration, ask: can you get from top to middle in time $\frac{1}{2}c^{S(n)}$ and from middle to bottom in time $\frac{1}{2}c^{S(n)}$. Now ask this recursively, until we get down to adjacent configurations.

How deep is the recursion going to be? The depth of the recursion is $\log_2 c^{S(n)} = O(S(n))$. What do we have to remember every time we recurse? The working midpoint configurations. For each level of the recursion we have to write down an entire configuration. The configuration takes $S(n)$ space, and each level costs $O(S(n))$ space. Hence the total is $O(S^2(n))$ space. \square

You can implement this in the word-ladder problem: write down a conjecture for the intermediate string. See if can get from/to in half as much time. This is slow slow but runs in relatively small space.

Lecture 17

Thu. 11/8/12

Problem set 5 is out today. It is due after Thanksgiving, so you can think about it while you're digesting.

Last time we talked about

- space complexity
- $\text{SPACE}(s(n))$, $\text{NSPACE}(s(n))$
- PSPACE, NPSPACE
- Savitch's Theorem says that $\text{PSPACE} = \text{NPSPACE}$.

Today we will

- finish Savitch's Theorem.
- Show TQBF is PSPACE-complete.

§1 Savitch's Theorem

Recall the following.

Theorem (Savitch, Theorem 16.13 again): For $s(n) \geq n$,

$$\text{NSPACE}(s(n)) \subseteq \text{SPACE}(s(n)^2).$$

Savitch's Theorem says that if we have a nondeterministic machine, we can convert it to a deterministic machine using at most the square of the amount of time.

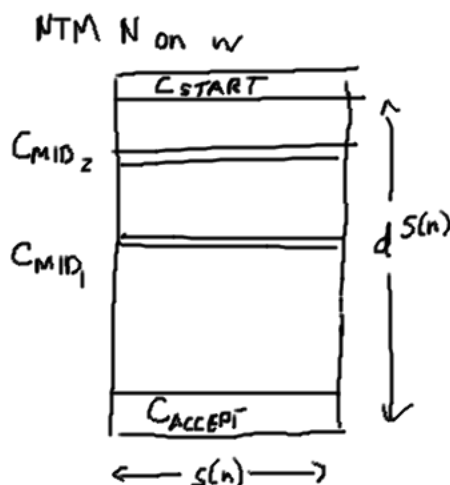
Nondeterminism only blows up space by a square, not an exponential. The proof is not super hard but it is not immediately obvious.

Proof. For NTM N using space $S(n)$ with configurations C_1, C_2 , write $C_1 \xrightarrow{t} C_2$ ("C₁ yields C₂ in t steps") if N can go from C_1 to C_2 in at most t steps. We give a recursive, deterministic algorithm to test $C_1 \xrightarrow{t} C_2$ without using too much space.

We will apply the algorithm to $C_1 = C_{\text{start}}$, $C_2 = C_{\text{accept}}$, and $t = d^{S(n)}$.

We may assume N has a single accepting configuration, by requiring the machine to clean up the space when it is done (just like children have to clean up their room). It puts blanks back, moves its tape head to the left, and only then does it enter the accept state.

The basic plan is to make a recursive algorithm.



We will inevitably have to try all possibilities, but we can do so without using too much space. The machine zooms to the middle, and guesses the midpoint configuration. It tries configurations sequentially one after another, as a candidate for the midpoint—think of it as cycling like an odometer through all possible configurations (of symbols and the tape head). This is horrendously slow, but it can reuse space. Once it has a candidate, it solves 2 problems of the same kind recursively: can we get from the top to the middle in half the time, and once we've found a path to the middle, we ask can we get from the middle to the bottom? Note that in the second half of the procedure, the machine can reuse space from the first half.

The machine continues recursively on the top half, splitting it into two halves and asking whether it can get between the configurations in a quarter of the original time.

The recursion only goes down a logarithmic number of steps, until it gets to $t = 1$. There are on the order of $S(n)$ levels. To check whether one configuration follows another in 1 step, just simulate the machine. How much do we have to write down every time we recurse? We have to write down the candidate for the middle. Each time we recurse we have a new configuration to write down.

We summarize the algorithm below.

On input C_1, C_2, t , do the following.

1. For $t > 1$, for each configuration C_{MID} test if $C_1 \xrightarrow{t/2} C_{\text{MID}}$ and $C_{\text{MID}} \xrightarrow{t/2} C_2$, reusing the space.

Accept if both accept (for some C_{MID}). (Then C_1 can get to C_2 in 2 steps.)

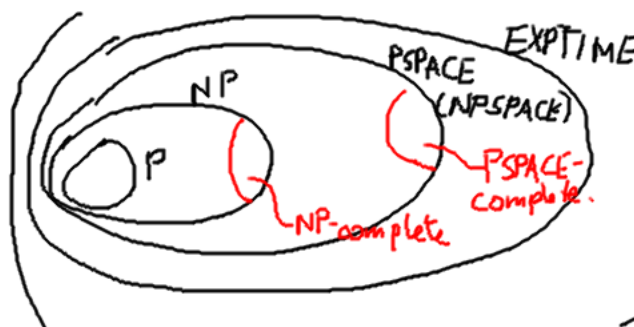
2. If $t = 1$, accept if C_1 can legally yield C_2 in 1 step of N or if $C_1 = C_2$.

The number of levels of recursion is $\log_2 d^{S(n)} = O(S(n))$. Each level requires storing a configuration C_{MID} and uses $S(n)$ space. The total space used is

$$O(S(n))S(n) = O(S^2(n)).$$

□

This tells us $PSPACE = NPSPACE$. Let's draw the picture of the world. (If $S(n)$ is polynomial, $S(n)^2$ is still polynomial.)



Let's move to the second topic for today.

§2 PSPACE-completeness

It is a famous problem whether $P = NP$. We know $NP \subseteq PSPACE$; we can also ask whether $P = PSPACE$. If a language needs polynomial space, can we just use polynomial time? Unbelievably, we don't know the answer to that either. For all we know, the whole picture collapses down to P !

A few (wacky) members of community believe $P = NP$. No one believes $P = PSPACE$. That would be incredible.

What we do have is the notion of NP-complete. There is a companion notion of PSPACE-completeness. Every problem in PSPACE is reducible to a PSPACE-complete problem. This is interesting for some of the same reasons that NP-complete problems are interesting. Showing a problem is PSPACE-complete is even more compelling evidence that outside P , because else $P = PSPACE$. Complete problems for class give you insight for what that space is about, and how hard the problems are.

PSPACE-completeness has something to do with determining who has a winning strategy in a game. There is a tree of possibilities in a game, and a structure to that tree: I win if for every move you make there exists a move I can make such that... This is the essence of what PSPACE is about. While we don't know $P \neq PSPACE$, we do know that P is not equal to the next one up: EXPTIME. You can prove $P \neq EXPTIME$. That is the first time where technology allows us to show something different.

Note there is a tradeoff: more time, less space vs. more space, less time. There are results in these directions, but we won't do them. For instance, there are Savitch's Theorem variants, which trade off time for space. It cuts the recursion at different points.

2.1 Definitions

This should look familiar, but there's one point we have to make clear.

Definition 17.1: We say that B is **PSPACE-complete** if

1. $B \in \text{PSPACE}$.
2. For every $A \in \text{PSPACE}$, $A \leq_P B$.

We are still using polynomial time reducibility. Why polynomial time? It's important to realize if we put PSPACE, that would be stupid. If A is polynomial space reducible to B , what would happen? This is related to the homework due today. The reduction can solve the problem itself and then target it to the right problem.

Thus, every 2 problems in P are polynomial time reducible to one another. Every 2 probs in PSPACE are polynomial space reducible to one another. Every we use polynomial space reducibility, every problem is PSPACE-complete. This is not interesting. *You have to use a reduction less powerful than class you're studying.* A reduction is a transformer of problems, not a solver of problems.

If you have a PSPACE-complete problem, and you can solve it in polynomial time by virtue of some miracle, then every other PSPACE problem can be solved in polynomial time, and we've pulled down all of PSPACE into P.

It's important to understand why we set it up this way!

2.2 TQBF is PSPACE-complete

An example of a PSPACE problem is TQBF (true quantified boolean formulas, where all variables are quantified by \forall 's and \exists):

$$\text{TQBF} = \{ \langle \phi \rangle : \phi \text{ is a true quantified Boolean formula} \}.$$

For instance, $\forall x \exists y (x \vee y)$.

Theorem 17.2: thm:tqbf-pspace-comp TQBF is PSPACE-complete.

The proof will be a recap of stuff we've seen plus 1 new idea.

Proof. 1. $\text{TQBF} \in \text{PSPACE}$: We saw last time that recursing on assignments gives a linear space algorithm (Theorem 16.10).

2. Let $A \in \text{PSPACE}$ be decided by a TM M in space n^k .

We give a polynomial time reduction f from A to TQBF, $f : w \mapsto \phi_w$, such that ϕ_w "says" M accepts w . ϕ_w captures M running on w ; so far this is the same idea as that in the Cook-Levin Theorem.

Consider a tableaux of M on w , with width $S(n)$ and height $d^{S(n)}$. M is deterministic, so there is just 1 possibility for the rows to be the computation history. As in Cook-Levin, we can try to build ϕ_w the same way. That gives us a correct formula.

The difference is that before we were talking about satisfiability. We can just put \exists quantifiers out front to make it a TQBF. This doesn't work; why? How big is the formula?

It's as big as the tableaux, exponentially big! You can't write down an exponentially big formula in polynomial time. We need a shorter formula which expresses the same thing. The ϕ_W from Cook-Levin is too big.

This is why the idea from Cook-Levin by itself is not enough.

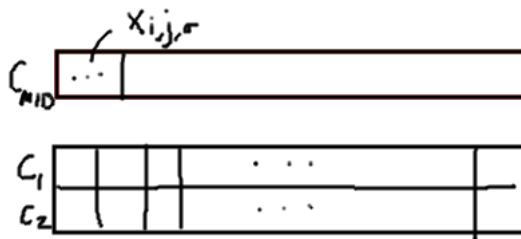
First we solve a more general problem. Let's solve the problem for C_1, C_2, t : give $\phi_{C_1, C_2, t}$ which says $C_1 \xrightarrow{t} C_2$. It will be helpful to talk about any 2 configurations, and being able to go from one to another in a certain amount of time. Even Cook-Levin would give you that: just use C_1 and C_2 in place of the start and end configuration. But this viewpoint allows us to talk about the problem in a different way.

As a first attempt, we can construct $\phi_{C_1, C_2, t}$ saying $C_1 \xrightarrow{t} C_2$ by writing

$$\phi_{C_1, C_2, t} = \exists C_{\text{MID}} (\phi_{C_1, C_{\text{MID}}, t/2} \wedge \phi_{C_{\text{MID}}, C_2, t/2})$$

and constructing subformulas recursively.

Why can we write down $\exists C_{\text{MID}}$? Really it is represented by the configurations of a bunch of variables. It is shorthand for $\exists x_1 \exists x_2 \cdots \exists x_\ell$. If $t = 1$, then $\phi_{C_1, C_2, t=1}$ and we can write the formula by Cook-Levin.



But have we done anything? The semantics of the formula are correct. All this is saying is that we can get from C_1 to C_2 in t steps iff there is some midpoint such that we can get from C_1 to the midpoint in half the time and from the midpoint to C_2 in half the time. (This smells like Savitch's theorem. There is more than meets the eye!) We cut t in half at the expense of creating 2 subproblems. The number of levels of the recursion is fortunately only d . Here $S(n) = n^k$.

We end up with polynomial time steps, but we double the size of the formula each time, so it's still exponential.

We ended up not doing anything! This shouldn't come as a total surprise. We're still only using the \exists quantifier. This is still a SAT-problem! We haven't used the full power of TQBF, which uses \exists and \forall 's.

Now and's and \forall 's are 2 flavors of the same thing. \exists 's are like or's. We're going to get rid of the "and." This looks like cheating but it's not:

$$\phi_{C_1, C_2, t} = \exists C_{\text{MID}} \forall (C_3, C_4) \in \{(C_1, C_{\text{MID}}), (C_{\text{MID}}, C_2)\} (\phi_{C_3, C_4, \frac{t}{2}}).$$

There is a fixed cost out front, and a single new formula at each level, not double formulas, so there is no blowup. We need to show this is legitimate. Note that $\exists C_{\text{MID}}$ stands for a string

that is $O(n^k = S(n))$ long. The same is true of the \forall quantifier. Let's rewrite the \forall in more legal language:

$$\begin{aligned}\forall(C_3, C_4) \in \{(C_1, C_{\text{MID}}), (C_{\text{MID}}, C_2)\}(\phi_{C_3, C_4, \frac{t}{2}}) \\ = \forall C_3 \forall C_4 [(C_3, C_4) = (C_1, C_{\text{MID}}) \vee (C_3, C_4) = (C_{\text{MID}}, C_2) \rightarrow \phi_{C_3, C_4, t/2}]\end{aligned}$$

This is the trick! This was done at MIT by Larry Stockmeyer in his Ph.D. thesis. It is called the Meyer-Stockmeyer Theorem.

How big is this formula? We start off with an exponential number of steps $d^{S(n)} = d^{n^k}$, so the number of recursions is $O(n^k)$. Each adds order $O(n^k)$ stuff out front, so the size of the formula is n^{2k} . Its size is polynomial, but it does have a squaring. \square

We see in both Savitch's Theorem 16.13 and Theorem 17.2 the following concept.



Recursion using middle configurations makes things polynomial, not exponential!

In fact, the proof Theorem 17.2 implies Savitch's Theorem: It could have been a nondeterministic Turing machine and the proof still works! Hence, very nondeterministic NPSPACE computation can be reduced to TQBF.

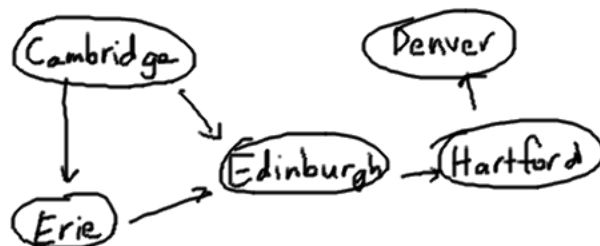
If a nondeterministic machine is reduced to TQBF, there is a squaring. Note TQBF can be done in linear space: A deterministic machine goes through all assignments, and solves TQBF in linear time. This gives a different proof of Savitch's Theorem.

2.3 PSPACE-completeness and games

PSPACE-complete problems can look like games. TQBF doesn't look like a game, but we'll see it really does.

We'll see other PSPACE-complete problems that are more strictly "games." My son, all he does is Xbox. There is a kind of game we used to play before Xbox, called geography. Choose some starting city, like Cambridge. Two players take turns. You have to pick a place whose first letter starts with the same letter Cambridge ends with. Edinburgh ends with H. I can pick Hartford, you Denver, I pick Raleigh, and so on. The first person who gets stuck loses. One more rule: no repetitions.

We can model this game as a graph. All cities are nodes.



Arrows correspond to legal moves. Let's abstract the game, and forget the labels. We take turns picking some path through the graph. It has to be simple: no repeats. If you get stuck somewhere with no place to go you lose. Depending on how you play, you might win or lose. The question is, if you play optimally, who wins? Given one of these graphs, which side has the win? We'll show this problem is PSPACE-complete by reducing TQBF to this problem.

Lecture 18

Thu. 10/11/12

Last time we showed TQBF is PSPACE-complete, analogous to how SAT was complete for NP.

Today we'll talk about

- generalized geography
- games
- log space: L and NL

§1 Games: Generalized Geography

Recall our generalized geography: Boston goes to New York City, Newark, etc., Newark goes to Kalamazoo, etc.

One important class of PSPACE problems are these games: Given a an initial configuration of a game, the moves allowed, and a rule for who has won, which player has the upper hand? If both sides play the best possible strategy, who will win? Many of these problems are in PSPACE.

We'll look at an example, generalized geography, and show that deciding who has a winning strategy is a PSPACE-complete problem.

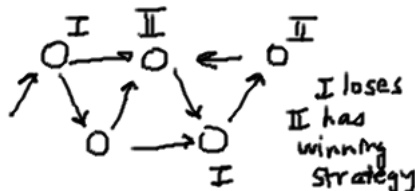
In generalized geography, we give a bunch of geographical names, for instance cities; each city called out has to start with the letter that the previous one ended with. The starting person picks Boston; the second player has to pick a place starting with *N*. Say Newark.

First person start with K , Kalamazoo. The person who gets stuck because there is no place to move to loses. You can draw a graph that shows the possible moves.

Abstracting, we erase the names and just remember the graph. Two players I and II take turns picking nodes of a simple path. The first one unable to move loses. Let

$$GG = \{ \langle G, a \rangle : \text{Player I has a winning strategy (forced win) in } G, \text{ starting at } a \}.$$

Here's an example.



In general, figuring out who has winning strategy is not so easy: it is PSPACE-complete. The proof is nice: it reveals connections between quantifiers and games.

Theorem 18.1: GG is PSPACE-complete.

Proof. Like showing a problem is NP-complete, we start off with a problem we already know to be PSPACE-complete. We have to show two things.

1. $GG \in PSPACE$. (This is easy, a straightforward recursive algorithm.)
2. $TQBF \leq_P PSPACE$. (This is the interesting fun part.)

To make sense of this reduction, we look at the TQBF problem in a different way, as a game.

Let ϕ be a quantified Boolean formula, for instance

$$\phi = \exists x_1 \forall x_2 \exists x_3 \forall x_4 [\Psi].$$

We know how to test whether this is true: Calculate and see if it works out. Put this aside for a moment.

Let's create a game for the formula. There are two players: one of them is called \exists and the other is called \forall . This is how you play the game. The players take a look at a formula. Start with \exists 's turn. \exists gets to pick the value of x_1 . Then it is \forall 's turn. \forall gets to pick the value of the next variable x_2 , and so forth. (There may be several variables in row with the same quantifier, but we can always throw in dummy variables so they alternate.)

\exists pick values of \exists variables, \forall pick values of \forall variables.

The two players have opposing desires. \exists is trying to pick values of variables to make the formula true at the end, to make variables satisfy the formula. \forall is trying to do the opposite: make the formula false.

\exists wins if then chosen values of x_1, \dots, x_4 satisfy Ψ and \forall wins if the chosen values don't satisfy Ψ .

I don't know if this game will be a big seller. However, it is a valid game: each player is trying to achieve an objective, and at end, we know who won.

Who has the winning strategy?

The cool thing is that we've already run into this problem. *This is exactly the same as the TQBF problem.* \exists has a winning strategy exactly when it is a true quantified boolean formula. What does it mean to have winning strategy? It means that under optimal play, the \exists player can make the formula true. In other words, *there exists* some move, such that *no matter* what the for all player does for x_2 , *there exists* some move x_3 ... Whether \exists has a winning strategy is the the same as the truth value of the formula: whether there exists some value, such that for all...

This is just a different view of the truth value.

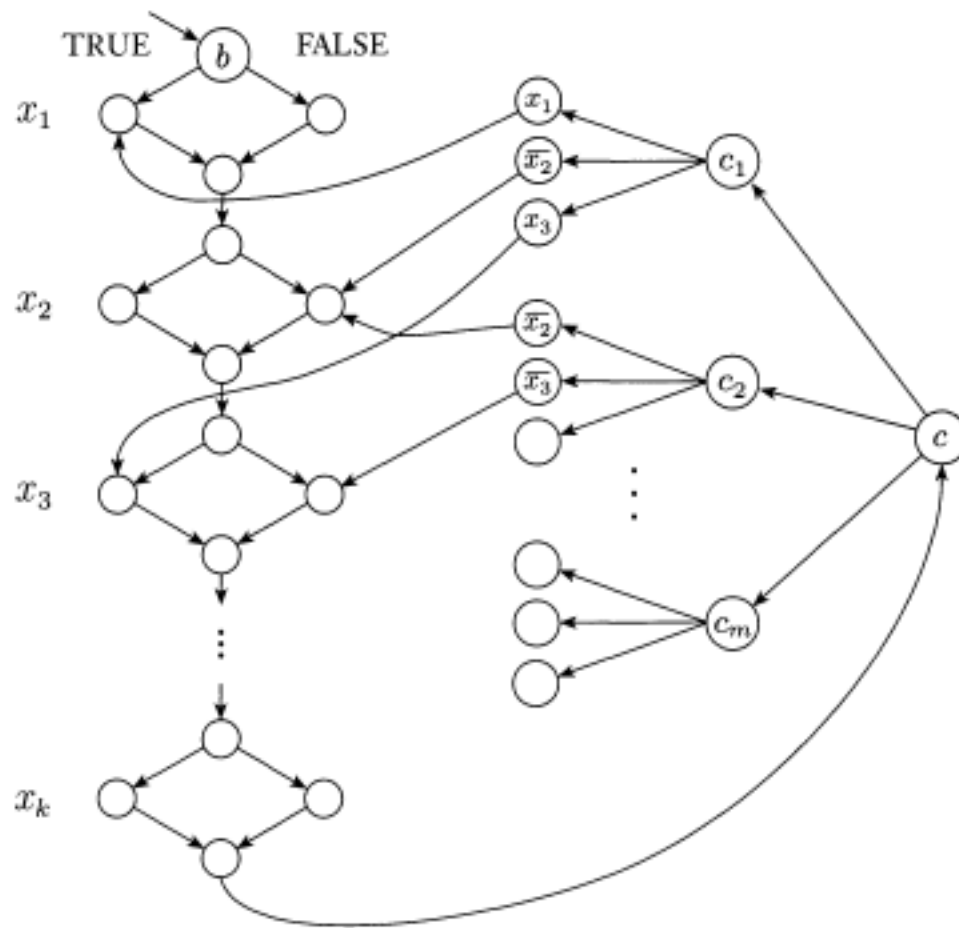
With this representation, we can reduce from TQBF to GG, i.e., show $\text{TQBF} \leq_P \text{GG}$. The technique is reminiscent of SAT reductions: We make gadgets, etc. The way you put them together, though, is different because there is a dynamic game component to it. Playing the game simulate playing the formula game. The gadgets work a little differently.

We send

$$\begin{array}{l} \phi \mapsto \langle G, a \rangle \\ \exists, \forall \quad I, II \end{array}$$

Player I will be like \exists and player II will be \forall .

The graph will have a sequence of diamonds. Let's look at a fragment and think about how it proceeds. \forall starts at the top. \forall has no choice. \exists player has a choice. Now \forall has a choice. This simulates the choice of the variables. The first diamond is the gadget for x_1 , the second for x_2 . (Figure from book)



If \forall appeared twice in a row, then we wouldn't have an extra node, which just served to switch whose turn it is.

After the diamond at the very bottom, all truth values for variables have been chosen. Let's assume going left corresponds to T and right corresponds to F. In the variable game, the game is over. In generalized geography, we're not finished because we want to arrange more structure—an endgame—so that the \exists player wins iff the formula is satisfied.

There is one node for each of the clauses. The \forall player picks a clause. The \forall player is claiming, or hoping, that the clause is unsatisfied. (We can assume it is in CNF, just like we reduced SAT to 3SAT.) \forall is trying to demonstrate that the formula not satisfied, by picking the unsatisfied clause. “You didn't win because clause 2 is not satisfied.”

(The one who tells the truth is going to be the one who ultimately wins.) Each clause points to all its literals. Psychologically, \forall claims c_1 not satisfied. \exists says it is satisfied because x_1 is true. Now it's the moment of truth. It is \forall 's turn. The positive literal is connected to true side of the construct. Negated variables get connected to false side. \exists claims “this clause is satisfied by x_1 .” If \exists was right, and earlier the game had gone through the true side of x_1 , then the \forall player can't move. If the \forall player is right, then play went down the other way, \forall can move, and now \exists is stuck.

All these nodes and arrows are laid down before the play begins. We build gadgets up front, one for each variable, and lay down nodes for each clause. We connect the nodes corresponding to literal in the clauses left or right depending on whether they are positive and negative. Playing the generalized geography game is just playing the formula game. There is a winning strategy exactly when the counterpart in the formula game has winning strategy. \square

This shows TQBF is PSPACE-complete, and hence probably a hard problem.

Similar results have been proven for natural games: the game of Go is played on a 19×19 board; 2 players have 2 colors of stones, each trying to surround the other person's stones. Determining which side has a winning strategy in Go from some preset board configuration is PSPACE-hard: you can reduce GG to the Go problem. There are structures in Go which correspond to moving through the GG configuration, and playing Go game corresponds to GG. There are 2 remarks in order. We actually have to generalize Go: 19×19 finite problem; we consider a $n \times n$ board. All results are asymptotic. (If we only considered 19×19 , then the problem is just a big table lookup.)

Go is at least PSPACE-hard. Whether it's in PSPACE depends on details on how the set game up. A special rule might let the game go on for a very long time, so this depends on details of the definition of game. PSPACE-hardness has been proven for other games $n \times n$ checkers, and $n \times n$ chess (which is less natural).

We now shift to a different set of classes, still in space complexity.

§2 Log space

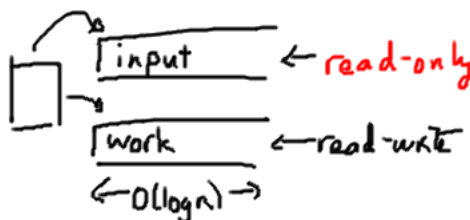
Instead of talking about polynomial space, we'll talk about a whole different regime, called log space. It has its own associated complexity classes and natural problems.

We look at $\text{SPACE}(\log n)$ and $\text{NSPACE}(\log n)$. We have space bounds that have size less than the problem. In order to make sense of this, we need to introduce a different model. Just by reading the entire input, the machine use space n . That is no sensible way to talk about log space. Instead, we allow the machine to read the entire input, but have a limited amount of work space.

Thus we consider a multitape Turing machine, with a

1. input (read-only) tape, and a
2. work (read/write) tape.

We will only count the space used on the work tape. The input given for free.



We can talk about $\log n$ -bounded work tapes. There will be an assumed constant factor allowed.

Definition 18.2: Define

$$\begin{aligned} L &= \text{SPACE}(\log n), \\ \text{NL} &= \text{NSPACE}(\log n), \end{aligned}$$

Why $O(\log n)$? Why not $O(\sqrt{n})$, etc?

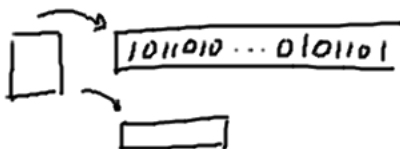
$\log n$ is a natural amount of space to provide. It is just enough to keep track of a pointer into the input. Constant $\log n$, for instance $7 \log n$, that's enough to keep track of 7 pointers into your input.

 Using \log space, a machine can keep track of a finite number of pointers.

We'll do a couple of examples.

Example 18.3: We have that the set of palindromes is in \log -space.

$$\{ww^R : w \in \{0,1\}^*\} \in L.$$



The machine zigzags back and forth on the input. It can't make any marks on the input, only keep track of stuff on the work tape. This is still good enough. The machine keeps track of how many symbols are already matched off; a fixed number of pointers enable this. For instance, it could record that it has already matched off 3 symbols, and is now looking at the 4th on the left or right. The machine uses a \log -space work tape.

We're considering machines with separate read-only input. The input may be enormous: for example, input from a CD-ROM or DVD-rom, onto your small little laptop. The laptop doesn't have enough internal memory to store all of it.

A better analogy is that the read-only input tape is the Internet, huge. You can only store addresses of stuff and probe things. What kinds of problems can you solve, if you have just enough memory to write down the index of things?

Example 18.4: path:nl We have

$$\text{PATH} = \{\langle G, s, t \rangle : G \text{ has a directed } s, t \text{ path}\} \in \text{NL}.$$

A nondeterministic machine can put a pointer on the start node, then nondeterministically choose one of the outgoing edges from the start node. It remembers only the current node it moved to. It forgets where it came from. The machine repeats. The machine jumps node by node nondeterministically, and accepts if it hits t .

The machine enough space to remember a node, which is logarithmic space, and also space to count how many nodes it has visited, so it can quit if it has visited too many vertices.

Can we solve PATH deterministically in log-space? Consider an enormous graph written down over the surface of the US, with trillions and trillions of nodes. Can you with 20 friends (or however many facebook friends you have), each just keeping track of where you are (a pointer into a location), operating deterministically, figure out whether you can get from some location to another? You can communicate by walkie-talkie (or by Internet).

Nobody knows the answer. Whether PATH is solvable deterministically (PATH $\in L$?) is an unsolved problem.

In fact the L vs. NL problem is open just as P vs. NP is open. There are NL-complete problems. If you can solve any of them in L, then you bring all of NL to L. PATH turns out to be complete for NL. We'll start to prove that.

§3 $L, NL \subseteq P$

Before that, let's look at the connection between L, NL, and the classes we've already seen.

Theorem 18.5: $L \subseteq P$.

Proof. If $A \in L$ and TM M decides A using $O(\log n)$ space, we have to show there is a deterministic machine that solves A in polynomial time.

How many configurations does the machine have? This tells us how long the machine can go for. Fix the input w . A configuration of M on w is $(q, h_1, h_2, \text{work tape contents})$. We don't include w because it is read-only. The number of configurations is

$$|Q| \cdot n \cdot d \log n \cdot \underbrace{c^{d \log n}}_{n^k} = O(n^\ell).$$

for some k, ℓ . No configuration can repeat, because no looping is allowed. Since the machine can have at most polynomial number of configurations, it runs in polynomial time. We get $A \in P$. \square

The following is trickier.

Theorem 18.6: thm:nl-p $NL \subseteq P$.

The previous proof would only give $NL \subseteq NP$. To get a deterministic polynomial time algorithm we need to construct a different machine.

Proof. Given a NL TM N , we convert it to an equivalent polynomial-time TM M .

How many configurations does N have? When we count the number of configurations, it doesn't matter if the machine is deterministic or not! A configuration is simply a snapshot. M takes all configurations and writes them down, but there's only polynomially many. $M =$ "on w ,

1. Write all configurations of N on w . We will treat these as the nodes of a graph, called the configuration graph.
2. Put an edge from one configuration c_1 to another c_2 when c_1 leads to c_2 in one step.

Now we have a big graph of all possible configurations. We have c_{start} and c_{finish} (we can assume there is a single accepting configuration, that the machine clears the work tape and moves its head to the left). Now we test if there is a path from the starting to the accepting configuration.

If there is a path, the nondeterministic machine accepts its input. The path gives a sequence of configurations that the nondeterministic machine goes on some path from start to accept.

Conversely, if the machine does accept, there has to be a path of configurations from start to accept, so there is a sequence of edges go from start to accept.

A polynomial time machine can do this test because it's the PATH problem! Depth or breadth first search works fine. This answers whether the NL machine accepts the input.

□

Lecture 19

Thu. 11/15/12

Last time we talked about...

- GG is PSPACE-complete
- L and NL

We reduced from TQBF to GG to show GG is PSPACE-complete. Then we turned our attention to a different regime: what if we consider logarithmic space instead of polynomial space? Log space is enough to give you pointers into the input. This has a certain power which we can describe; it fits in nicely into our framework.

Today we'll talk about

- NL-completeness
- NL=coNL (this differs from what we think is true for NP)

Recall that $L = \text{SPACE}(\log n)$ and $NL = \text{NSPACE}(\log n)$. We have a nice hierarchy:

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE.$$

We don't know whether these containments are proper. We can show that PSPACE and NL are different (and will eventually do so), so not everything in the picture collapses down. Most people believe that these spaces are all different; however, we don't know adjacent inclusions are proper.

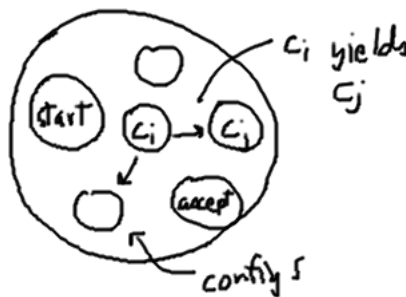
However, $NL = \text{coNL}$ shows that surprising things do happen, and we do have unexpected collapses.

First let's review a theorem from last time.

Theorem (Theorem 18.6): $NL \subseteq P$.

Proof. For a NL-machine N , a configuration of N on w is (q, p_1, p_2, t) . The number of configurations of N on w is polynomial in n where $n = |w|$ (w is fixed). The computation graph is the graph where

- nodes are configurations, and
- edges show how N can move.



Here is a polynomial time algorithm that simulates N . “On input w ,

1. Construct the computation graph.
2. Test if there is a path from start to accept (using any polynomial time algorithm for PATH).
3. Accept if yes and reject if no.”

□

§1 L vs. NL

Now we turn our attention to L vs. NL. We'll show that the situation is analogous to the situation of P vs. NP. How much space deterministically do we actually need for a NL problem? We can do it with polynomial space but that's pretty crude. We can do much better.

We have using Savitch's Theorem that

$$\text{NL} = \text{NSPACE}(\log n) \subseteq \text{SPACE}(\log^2 n)$$

We stated Savitch's Theorem for space bounds $\geq n$; with space bounds of $\geq \log n$ the same argument goes through. No one knows whether we can reduce the exponent, or whether $\text{L}=\text{NL}$.

(We will show that $\text{SPACE}(\log n)$ is provably different from $\text{SPACE}(\log^2 n)$, using the hierarchy theorem 20.1. When we increase the amount of space/time, we actually get new stuff. But maybe some other argument could show $\text{NL} \subseteq \text{SPACE}(\log n)$.)

We will show that there are NL-complete problems, an example of which is PATH. If you can solve PATH or any other NL-complete problems in deterministic log space, then it brings down everything with it to L. We'll show everything in NL is reducible to the PATH problem. This shouldn't be a surprise because it's what we did in the previous theorem: whether a machine accepts is equivalent to whether there's a path. We'll just need to define NL-completeness in the appropriate way and then we'll be done by the argument given in the $\text{NL} \subseteq \text{P}$ theorem.

Definition 19.1: B is NL-complete if

1. $B \in \text{NL}$
2. Every NL-problem is *log-space* reducible to B : for every $A \in \text{NL}$, $A \leq_L B$.

We need to define what it means to be log-space reducible. We have to be careful because the input is roughly n , and the output is roughly n . we don't want to count the output of machine in the space bound. The input and output should be kept separate.

Definition 19.2: A **log-space transducer** is a Turing machine with 3 tapes,

1. input tape (read only),
2. work tape (read-write), and
3. output tape (write only),

such that the space used by the work tape is $O(\log n)$ with n the size of the input.

We say that $f : \Sigma^* \rightarrow \Sigma^*$ is **computable in log-space** if there is a log-space transducer that on input w , which leaves $f(w)$ on the output tape.

We don't use polynomial reducibility because once we have polynomial time we can solve NL problems. The reducer can figure out whether a string is in A , then direct it to a point in B . It would not change the problem, just get the answer and dump it in B . If we used polynomial reducibility, everything would be NL-complete except ϕ and Σ^* .

We need a reduction that the L machine could compute. If we used polynomial reduction, an L machine couldn't necessarily make the reduction. But if we log space reduction, then an L machine can compute the reduction.

We have the following analogous theorem.

Theorem 19.3: If $A \leq_L B$ and $B \in L$ then $A \in L$.

Why doesn't the same argument for P work for L? It's a bit tricky because we can't write all of the output of f on a L-machine.

Proof. The algorithm for A is the following. "On w ,

1. Compute $f(w)$.
2. Test if $f(w) \in B$.
3. Accept or reject accordingly.

But we can't write $f(w)$!

There's a trick that fixes this. We run B without having $f(w)$ available. Every time we need a bit of w , we run the whole reduction, throw away all the output except the bit we're looking for, and plug that into the machine.



Recomputation allows us to get by with logarithmic memory.

□

Proposition 19.4: If $A \leq_L B$ and $B \leq_L C$ then $A \leq_L C$.

Proof. Use the same idea, doing computation on the fly.

□

Now let's turn to NL-completeness.

§2 NL-completeness

Theorem 19.5: PATH is NL-complete.

Proof. We have to show

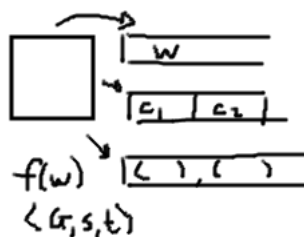
1. $\text{PATH} \in \text{NL}$: We already proved this (Example ??).

2. For $A \in \text{NL}$, $A \leq_L \text{PATH}$. We give a generic reduction. Say that the NL-machine N decides A . We give the reduction f .

Given w , let $f(w)$ be $\langle G, s, t \rangle$ where G is the computation graph for N on w , s is C_{start} , and t is C_{accept} (again, we assume N cleans up its tape at the end, so that there is just one accept state). Testing whether there is a path from C_{start} to C_{accept} is an instance of the PATH problem. We have $w \in A$ iff there is a path from s to t .

This machine does the right thing. We have to show we can do the reduction in log space, i.e., f is log-space computable.

$f(w)$ is supposed to be a description of the nodes and edges, and which is the starting and ending node. Split the work tape into 2 pieces, representing 2 configurations of N on w , say C_1 and C_2 .



We'll go through all possible pairs of configurations sequentially, just like an odometer. For each possibility of C_2 look at all possibilities of C_1 . We cycle through all possible pairs of configurations, testing whether C_1 legally yields C_2 in 1 step according to the rules of N . If so, take the pair and output an edge between them. The whole thing takes log-space, because writing C_1, C_2 takes log space.

This proves f is a log-space computable function, so the reduction takes log-space. □

Note that the output depends on w . How? Which configurations lead from others—it might seem like these depend only on machine. But $f(w)$ should depend on w . The start configuration doesn't depend on w , and doesn't have w built in. When you look at whether you can transition from C_1 to C_2 , they have head positions as part of the configuration. In order to see whether C_1 leads to C_2 we have to see what's in the cell that the head is at. Thus the edges of the graph do depend on w .

For homework, you need to show other problems are NL-complete. To show other problems are NL-complete, we reduce PATH to show they are also NL-compute, just like we reduced 3SAT.

Let's move on to this amazing problem.

§3 NL=coNL

Let's look at the picture. 20 years ago we thought $NL \neq coNL$, with L in the intersection, much like we still think the picture for P vs. NP still looks like this. However, actually $NL = coNL$.

Theorem 19.6: $NL = coNL$.

Proof. A reducible to B exactly when \overline{A} reducible to \overline{B} , so all we need to do is show $\overline{PATH} \in NL$. How do we give a NL-algorithm that recognize the nonexistence of a path?

What could we guess, so that if accept at the end, there's no path?

Perhaps we could guess a cut. But writing down a cut requires more than log-space.

The algorithm is very nonobvious. This was a prize-winning paper.

We'll give the algorithm in pictures. We have our graph G , with starting and ending nodes s and t . The idea came a little out of left field. The guy's advisor asked: if you're trying to solve problems and you're given information for free, what happens? What happens if you're given for free *the number of nodes you can get to from s* ?

We first give a NL-algorithm for the \overline{PATH} , given the number of nodes reachable from s . Let R be the set of nodes reachable from s . Let c be the size of R .



Our algorithm goes through all nodes of G one by one. Every time it gets to a new node, it guesses whether the node is reachable. If it guesses the node is reachable, it will prove it's right by guessing the path. (If can't find the path, that branch dies.) If the node is reachable, some branch will guess the right path and then move on. We keep track of how many reachable nodes we've found. When we're done, if the count equals c , then we've guessed right all the way along; we've found all the reachable ones. All the ones that we've guessed to be nonreachable are really nonreachable. If t wasn't guessed, t is nonreachable!

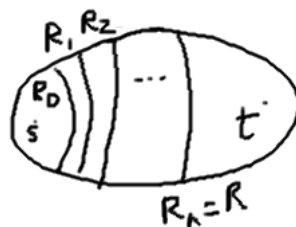
Now we've reduced the problem to computing c . We compute it using the same technique. We layer the graph into R_0, R_1, R_2, \dots where

$$R_i = \text{nodes reachable from } s \text{ by path with length } \leq i.$$

Note

$$R_0 \subseteq R_1 \subseteq \dots \subseteq R_m = R$$

because m is the maximal possible number of steps you need to reach any node. Let $C_i = |R_i|$.



We will show how to compute C_{i+1} from C_i . Then we can throw away the previous C -value. So we can get the count of nodes reachable in any number of steps, and we're done.

We need a way of testing whether a node is in R_{i+1} : nondeterminism will either get the value correctly, or that branch of the nondeterminism will fail. Some branch will have guessed everything correctly along the way.

Each time we're testing whether a node v is in R_{i+1} , we go through all the nodes, guessing which are in R_i and which are not. If we guess it's in R_i , we prove it is in by guessing a path of length at most i . We keep a count of the number of nodes in R_i and make sure it equals C_i at the end. Along the way we check if any of these nodes connect to v .

Now iterate over $i = 0, \dots, m - 1$. □

Lecture 20

Tue. 11/20/12

Last time we showed:

- PATH is NL-complete
- NL=coNL

Today we'll talk about the time and space hierarchy theorems.

§0 Homework

Problem 3:

Show a language is in L. If you just try to do it with your bare hands, it's a mess. But if you use the methodology we talked about in lecture, it's a 1-2 line proof. Don't just dive in but use a technique we introduced to make it simpler.

Problem 6:

Here the satisfiability problem is made to be easier: The clauses have at most 1 negated literal per clause (for instance, $(\bar{x} \vee y_1 \vee \dots \vee y_k)$), and that negated literal cannot appear anywhere else. This turns out to be solvable in NL, and be NL-complete. As a hint, $(\bar{a} \vee b)$ is equivalent to $a \rightarrow b$. Thus we can rewrite $(\bar{x} \vee y_1 \vee \dots \vee y_k)$ as

$$(x \rightarrow (y_1 \vee \dots \vee y_k)).$$

This suggests you think of the problem as a graph which represents the formula in some way. The nodes are the clauses, and have an edge going from $(x \rightarrow y_1 \vee \dots \vee y_k)$ to a clause containing y_1 on the left-hand-side of an implication. If x is true, one of y_1, \dots, y_k is true; then following an edge we get to one of the y_k . Think of this as a connectivity problem in a graph.

For the reduction, we want to reduce the graph to the restricted satisfiability problem. We can just reduce from graphs that don't have any cycles in them. Reduce a path problem to the satisfiability problem, using a construction inspired by the above. The construction requires the starting graph not to have cycles. You have to remove the cycles because they cause problems. The acyclic path problem is still NL-complete; this is in the textbook. Use the technique of level graphs, explained below.

To show

$$\text{PATH} \leq_L \text{acyclic-PATH},$$

take your graph and convert it to a bunch of copies of the vertex set, where an edge from a to b now goes from a in one layer to b in the next layer down. There are no backward-going edges so there are no cycles. But if we had an edge from s to t , there is still an edge from s to t in modified graph.

§1 Space hierarchy

We've wrapped up the basics on complexity classes for time and space. We'll now talk about a pair of theorems that relate to both time and space. The hierarchy theorems have a very simple message. With respect to time and space (let's think of time for the moment), and if you have a certain amount of time you're allowing the machine, then if you increase the time, you'd expect there's more stuff the machine you could do (say n^3 instead of n^2). For this question the answer is known: if you give the machine a little more time or space, it can do more things.

In particular, the theorem tells you there are decidable languages that are not in P.

So far we have

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE.$$

Even $L \stackrel{?}{=} P$ is open, and $P \stackrel{?}{=} PSPACE$ is open. However, $L, PSPACE$ are provably different, so we can't have both $L = P$ and $P = PSPACE$.

There are separations out there, which we don't know how to prove. The general belief is that all of these are separate. We can actually prove something stronger. We have by Savitch's Theorem that

$$NL \subseteq \text{SPACE}(\log^2 n) \subset PSPACE,$$

the inclusion proper by Space Hierarchy. We know $NL \neq PSPACE$, but nothing stronger is known.

Theorem 20.1 (Space Hierarchy Theorem): [thm:space-hierarchy](#) For functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$ where

1. f is **space constructible**: it can be computed in $f(n)$ space. (This is a technical condition that all normal functions will satisfy.)
2. $g(n) = o(f(n))$,

then there is a language

$$A \in \text{SPACE}(f(n))$$

with

$$A \notin \text{SPACE}(g(n)).$$

(Note $g(n) = o(f(n))$ means $g(n) < cf(n)$ for any constant $c > 0$, if you make n large enough. In other words, $f(n)$ dominates $g(n)$ for large enough n .)

We will find some language A in $\text{SPACE}(f(n))$ and not in $\text{SPACE}(g(n))$, to prove this.

For instance take $g(n) \sim n^2$ and $f(n) \sim n^3$: we can do something in n^3 space that we can't do in n^2 space.

The space hierarchy theorem has a slightly easier proof than the time hierarchy theorem.

What are we going to do? I'll tell you what we're not going to do. It would be nice if the language was some nice language, understandable as a string manipulation, with f as a parameter somewhere. Rather, it will be an artificial concocted language designed specifically to meet the conditions that we want; we won't be able to understand it simply otherwise. Later on we'll find more natural languages that take a lot of space.

The machine operates in space $f(n)$, and by design, makes sure its language can't be done in less space. *It simulates all smaller space machines and acts differently from them.* Really it amounts to a diagonalization. We build something different from everything in some list.

Let's review diagonalization. To prove \mathbb{R} is uncountable, given a list of real numbers, we make a number differing from everything in the list by at least one digit (Theorem 8.7). To show A_{TM} is undecidable, we make a machine that looks at what M_i does on $\langle M_i \rangle$ and does the opposite (Theorem 8.10). Its language D is new thing, and can't be on the list of all possible Turing machines, a contradiction.

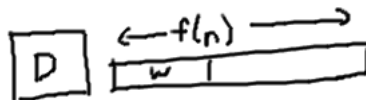
Our proof is similar in spirit. Think of M_i as the machines that operate in space $g(n)$ where $g(n) = o(f(n))$, the small space machines. D does something different from what each M_i does, so D can't be a small space machine.

However, D is decidable. Testing whether the M_i accept their input takes small space. Our language is decidable in space just little more. We have to be careful in our analysis just to show D can decide the diagonal in just a little more space; by construction it can't do the tests in small space, but can do it in more space $f(n)$.

Proof. We give a Turing machine (decider) D where $A = L(D)$ and D is a decider running in space $O(f(n))$. This gives $A \in \text{SPACE}(f(n))$. Our algorithm for D will show that D is not solvable in smaller space, $A \notin \text{SPACE}(g(n))$.

Our first try is the following. Let $D =$ "on input w (of length n):

1. Compute $f(n)$ and mark off $f(n)$ tape cells. (If the machine ever needs to use more space, then reject.)¹²



2. If $w \neq \langle M \rangle$ for some TM M , then reject. If $w = \langle M \rangle$, continue; D will try to be different from M .
3. Run M on w and do the opposite (this is an oversimplification; we have to make some adjustments)."

Modulo a little engineering, this is our description of D . Conceptually, this is the whole proof.

But we might not finish, M might take more space than we allocated, in which case D ends up rejecting. Is that a problem? We only have an obligation to be different from the small-space machines. We'll be able to run small spaces to completion. Our language is different from what those languages are.

This is a bit of a cheat. There are 2 critical flaws in the argument. I claimed that if M 's computation doesn't fit in the space, I don't have to worry about it. That's not true. It could be the machine uses lots of space on small input, but on large input, it uses space $o(f(n))$. We have $g(n) > f(n)$ for a particular w (but not asymptotically)—we had one chance to be different from that machine, and we've blown it. No one tells us the constant factor. This problem seems more serious!

We want to run M on a bigger w . We don't what w we need, but big enough so the asymptotics kick in. Thus we'll pad it in all possible ways; we'll have infinitely many chances to be different.

We change the above as follows. let's strip off trailing 0's and see if the remainder is a Turing machine. We could have a string with billions of 0's, run on some relatively small Turing machine. Let D = "on input w (of length n):

1. Compute $f(n)$ and mark off $f(n)$ tape cells. (If the machine ever needs to use more space, then reject.)
2. If $w \neq \langle M \rangle 0^*$ for some TM M , then reject. If $w = \langle M \rangle 0^*$, continue; D will try to be different from from M .

¹²We use the technical condition that $f(n)$ can be computed in $f(n)$ space; the machine needs to understand how much extra space it got in order to do something new to it. There is a counterpart to the theorem: we can construct gaps in hierarchy where nothing new from g up to f , by constructing f so complicated, that we can't compute f in f space. This is the gap theorem. There is one gap you can describe easily; log-log-space. There is a gap between constant space and log-log-space. Nothing nonregular is in $o(\log \log n)$ space.

3. Run M on w and do the opposite.”

This allows D to run M on very long inputs.

This solves one problem. But it's possible that M on w goes forever. It can only do so in a particular way: using a small amount of space. If M blindly simulates, it is going to loop. The amount of time it can take is exponential in the amount of space. Thus, we run a counter to count up the amount of time a machine can run without getting into a loop, on that amount of space. It's just $2^{f(n)}$.

The counter takes a constant factor more space; put the counter out to the right, or think of it running on a separate track below. If we exceed the time bound, then reject. Using asymptotics, for large enough n , we will run to completion on some input and be different.

Let D = “on input w (of length n):

1. Compute $f(n)$ and mark off $f(n)$ tape cells. (If the machine ever needs to use more space, then reject.)
2. If $w \neq \langle M \rangle 0^*$ for some TM M , then reject. If $w = \langle M \rangle 0^*$, continue; D will try to be different from M .
3. Run M on w and do the opposite.

(a) Reject if exceeds $2^{f(n)}$ time.”

□

Constructibility works down to $\log n$ (we have to work with the special model for sublinear space).

§2 Time Hierarchy Theorem

The issue of the overhead becomes more of a concern.

Theorem 20.2: thm:time-hierarchy If f is **time-constructible** and $g(n) = o\left(\frac{f(n)}{\log n}\right)$, then there exists $A \in \text{TIME}(f(n))$ and $A \notin \text{TIME}(g(n))$.

In the interest of time (pun intended) we'll just sketch the proof. The idea is the same.

Proof. Let D = “on input w of length n ,

1. Compute $f(n)$. Start counter (“timer”) to $f(n)$.
2. If $w \neq \langle M \rangle 0^*$ then for some TM M , reject.
3. Run M on w and do the opposite (provided it runs within the time on the counter).

We have to be careful. Every time we do a step, we refer back to M . The overhead, if we're not careful, will be bad. We only have an extra factor of $\log n$ to work with. We extend our tape alphabet so that every tape cell has enough space to write 2 symbols. We'll keep the description of M on the tape: Like checking out book from the library, we'll take M and carry it with us. More complicated is the counter.

M is constant size thing. The counter is not constant in size; it grows with n , hence is logarithmic in size. This contributes $\log n$ overhead. \square

Lecture 21

Tue. 11/27/12

Last time we talked about hierarchy theorems. If we allow a bit more time or space, then there are more things we can do.

Today we'll talk about

- natural intractable problems
- Relativization, oracles

§1 Intractable problems

Definition 21.1: Define

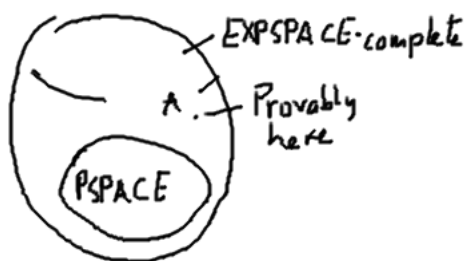
$$\begin{aligned}\text{EXPTIME} &= \bigcup_k \text{TIME}(2^{n^k}) \\ \text{EXPSPACE} &= \bigcup_k \text{SPACE}(2^{n^k}).\end{aligned}$$

(Think of it as $2^{\text{poly}(n)}$.)

The hierarchy theorems show that there are things we can do in exponential time that we can't do in polynomial time, and the same is true for space. We have proper inclusions.

$$P \subset \text{EXPTIME}, \quad \text{PSPACE} \subset \text{EXPSPACE}.$$

We found $A \in \text{EXPSPACE} \setminus \text{PSPACE}$. This was a language that the hierarchy machine produced for us. It decides in such a way that makes it provable different. A is by design not doable in polynomial space, because it diagonalizes over all polynomial space machines.



But A is an unnatural language; it has no independent interest; it just came out for sake of proving the hierarchy theorem.

We'd like to prove some more natural language is in $\text{EXPSPACE} \setminus \text{PSPACE}$. To do this we turn to completeness.

We'll introduce an exponential space complete problem, in the same spirit as our other complete problems. Everything in the class reduces to it. It cannot be in polynomial space because otherwise $\text{PSPACE} = \text{EXPSPACE}$. Because A is outside PSPACE , the classes are different, and the exponential space complete problem must also be outside.

The language is a describable language. We can write it in an intelligible way. It's a toy language. There are languages that people are more interested in that have completeness properties. Our problem will illustrate the method, which we care about more than the results. This is like the Post Correspondence Problem. Other languages are less convenient to work with.

Definition 21.2: A language is **intractable** if it is provably outside of P .

Example 21.3: Here's a problem that mathematicians care about. Remember that we talked about number theory: we can write down statements.

Consider a statement of number theory with quantifiers and statements involving only $+$. Chapter 6 gives an algorithm for testing whether such statements are true or false. It's a beautiful application of finite automata. The algorithm is very slow; it repeatedly involves converting a NFA to a DFA, which is an exponential blowup. The algorithm runs in time $2^{2^{\dots}}$, a tower whose length is about the length of the formula. It can be improved to double exponential.

Is there a polynomial time algorithm? No, it's complete for double exponential time; it provably cannot be improved. We'll give the flavor of how these things go, by giving an exponential problem that's more tailored to showing it's complete. That's the game plan.

1.1 EQ_{REX}

First we consider a simpler problem.

Definition 21.4: Define

$$\text{EQ}_{\text{REX}} = \{ \langle R_1, R_2 \rangle : R_1, R_2 \text{ are regular expressions and } L(R_1) = L(R_2) \}.$$

This can be solved in polynomial space (it's a good exercise). We can convert regular expressions to NFAs of about the same size. Thus we can convert the problem to testing whether two NFA's are equivalent. We'll look at the complementary problem, the inequivalence problem, show that is in PSPACE.

We show $\overline{\text{EQ}_{\text{REX}}}$ is in NPSPACE and use Savitch's Theorem 16.13. The machine has to accept if the strings are not equivalent. We'll guess the string on which they give a different answer.

If one machine is in an accepting state on one and the other machine not in an accepting state on any possibility, we know the machines are not equivalent, and we can accept.

Does this also show the inequivalence problem is in NP? Why not? Why can't we use the string as the witness, that's accepted by one machine to another? The mismatch could be a huge string that is not polynomially long. The first string on which differ could be exponentially long.

To use polynomial space, we modify our machine so it guesses symbol by symbol, and simulates the machine on the guessed symbols.

A variant of this problem is not in PSPACE.

For a regular expression R , let $R^k = \underbrace{R \cdots R}_k$. Imagining k is written down as a binary number, we could potentially save a lot of room (save exponential space) by using exponentiation. We'll talk about regular expressions with exponentiation.

Definition 21.5: Define

$\text{EQ}_{\text{REX}\uparrow} = \{\langle R_1, R_2 \rangle : R_1, R_2 \text{ are regular expressions with exponentiation and } L(R_1) = L(R_2)\}.$

Definition 21.6: We say B is EXPSPACE-complete if

1. $B \in \text{EXPSPACE}$.
2. for all $A \in \text{EXPSPACE}$, $A \leq_P B$.¹³

We show the following.

Theorem 21.7: $\text{EQ}_{\text{REX}\uparrow}$ is EXPSPACE-complete.

Proof. First we have to show $\text{EQ}_{\text{REX}\uparrow}$ is in EXPSPACE. If we have *regular* regular expressions, we know it's in polynomial space; using the Savitch's Theorem trick we argued at the beginning of class that it's doable in polynomial space. For regular expressions with exponentiation, expand each concatenation. This blows up the expression by at most an exponential factor. Now use the polynomial algorithm in the exponentially larger input. The claim follows immediately.¹⁴

¹³Why polynomial time reduction, not polynomial space reduction? Reductions are usually doable in log space; they are very simple transformations relying on repeated structure. Cook-Levin could be done in log-space reduction. If weaker reductions already work, there's no reason to define a stronger one.

¹⁴If we allow complements in the expression, we're in trouble. The algorithm doesn't work for complements. If we have complementation we have to repeatedly convert NFA's to DFA's to make everything work out.

Now we show $\text{EQ}_{\text{REX}\uparrow}$ is EXPSPACE -complete. Let $A \in \text{EXPSPACE}$ be decided by TM M in space 2^{n^k} . We give a reduction f from A to $\text{EQ}_{\text{REX}\uparrow}$ sending w to $f(w) = \langle R_1, R_2 \rangle$, defined below.

Let Δ be the computation history alphabet. Let

- R_1 be just all possible strings over some alphabet, Δ^* , and
- R_2 be all strings except rejecting computation histories for M on w .

If M rejects w , there is a rejecting computation history. Then R_2 will be all strings except for that one string, and the regular expressions will not be equivalent, $R_1 \neq R_2$.

If M accepts w , then there are no rejecting computation histories, and $R_1 = R_2$.

How big are R_1 and R_2 allowed to be? They have to be polynomial in the size of w . How big can R_2 be? w already has exponential space, so the string is double-exponentially big. The challenge is how to encode: how to represent the enormous objects even though you yourself are very small.

We construct R_2 as follows. R_2 is supposed to describe all the junk: every string which fails to be a computation history (because it's scribble). We look at all the possibilities that R_2 can fail; we have to describe all failure modalities. We'll write

$$R_2 = R_{\text{bad-start}} \cup R_{\text{bad-reject}} \cup R_{\text{bad-compute}}.$$

The beginning is bad, the end is bad, or somewhere along the line we made a mistake moving from one configuration to the next.

A computation history looks like

$$C_{\text{start}} \# C_1 \# \cdots \# C_{\text{reject}}.$$

The configurations are 2^{n^k} big, because we have to write down the tape of the machine. Assume they are padded to the same length, so $C_{\text{start}} = qw_1 \cdots w_n \sqcup \cdots \sqcup$.

$R_{\text{bad-start}}$: We describe $R_{\text{bad-start}}$ as all words which don't have first symbol q_0 , or 2nd symbol w_1 , and so forth, so everything that doesn't start with $qw_1 \cdots w_n \sqcup \cdots \sqcup$. To start, let

$$R_{\text{bad-start}} = (\Delta - q_0)\Delta^* \cap \Delta(\Delta - w_1)\Delta^* \cap \Delta^2(\Delta - w_2)\Delta^* \cap \cdots \cap \Delta^n(\Delta - w_n)\Delta^* \cap \cdots$$

(Technically we have to write out $\Delta - q_0$ as a union. This is shorthand. It's not a regular expression as we wrote it, but we can easily convert it.) Now we have to deal with the blanks. This is a little of a pain. Naively we have to write down an exponential number of expressions $\Delta^i(\Delta - \sqcup)\Delta^*$. We do a bit of regular expression hacking. We let

$$\begin{aligned} R_{\text{bad-start}} &= (\Delta - q_0)\Delta^* \cap \Delta(\Delta - w_1)\Delta^* \cap \Delta^2(\Delta - w_2)\Delta^* \cap \cdots \cap \Delta^n(\Delta - w_n)\Delta^* \\ &\quad \cap \Delta^{n+1}(\Delta \cup \varepsilon)^{2^{n^k} - (n+1)}(\Delta - \sqcup)\Delta^* \cap 2^{n^k}(\Delta - \#)\Delta^*. \end{aligned}$$

(Any string that starts with $n + 1$ to 2^{n^k} symbols followed by a non-blank is a bad starting string.) Note that 2^{n^k} can be written down with $n^k + 1$ bit

$R_{\text{bad-reject}}$: Let

$$R_{\text{bad-reject}} = (\Delta - q_{\text{rej}})^*.$$

$R_{\text{bad-compute}}$: For $R_{\text{bad-compute}}$, we need to describe all possible errors that can happen, $\Delta^*(\text{error})\Delta^*$. An error means we have a bad window; we have an incorrect window def following abc in the same position. Thus we let

$$\bigcup_{abcdef \text{ illegal window}} \Delta^*(abc\Delta^{2^{n^k}-2}def)\Delta^*.$$

Note that this is a constant-size union independent of n ; it is at most size $|\Delta \cup Q|^6$.

We're done! R is a polynomial time regular expression with exponentiation. \square

We proved this language is not in PSPACE, hence not in P, hence truly intractable.

Can we use the same method to show the satisfiability problem is not in P ? That would show $P=NP$. There is a heuristic argument that shows this method will not solve the P vs. NP problem. This has to do with oracles.

The moral of the story is that this method, which is very successful in showing a language outside of P, is not going to show SAT is outside of P.

§2 Oracles

Sometimes we want to think of a Turing machine that operates normally, but is allowed to get a certain free language. The machine is hooked up to a black box, the oracle, which is going to answer questions whenever the machine decides to ask one, about whether a string is in the language.

Definition 21.8: An **oracle** for a language A is a machine (black box) that answers questions about what is in A for free (in constant time/space).

M^A is a TM with access to an oracle for A .

Let P^A be the languages decidable in polynomial time with oracle A , and define NP^A in the languages decidable in nondeterministic polynomial time with oracle A .

Let's look at some examples. A handy oracle is an oracle for SAT.

Example 21.9: P^{SAT} is the class of languages that you can solve in polynomial time, with the ability to ask whether any expression is in SAT.

Because SAT is NP-complete, this allows you to solve any NP problem:

$$NP \subseteq P^{\text{SAT}}.$$

Given a language in NP, first compute a polynomial reduction to SAT, and then ask the oracle whether the formula is true.

We also have

$$\text{coNP} \subseteq P^{\text{SAT}},$$

because P^{SAT} , a deterministic class, is closed under complement.

This is called computation relative to SAT. The general concept is called **relativization**.

Whether

$$\text{NP}^{\text{SAT}} \stackrel{?}{=} P^{\text{SAT}}$$

is open. However, we do know the following.

Theorem 21.10: For some A ,

$$P^A = \text{NP}^A.$$

For some other B ,

$$P^B \neq \text{NP}^B.$$

We'll prove the first fact, and then see the important implications.

Proof. Let $A = \text{TQBF}$ (or any PSPACE-complete problem). Now because TQBF is in PSPACE, the machine can answer the question without the oracle, we can eliminate the oracle.

$$\text{NP}^{\text{TQBF}} \subseteq \text{NPSPACE} \stackrel{\text{Savitch}}{=} \text{PSPACE} \subseteq P^{\text{TQBF}},$$

the last because TQBF is PSPACE-complete. □

Here is the whole point of why this is interesting.

Suppose we can prove $P \neq \text{NP}$ using essentially the technique for the first $\frac{2}{3}$ of the lecture: hierarchy theorem and a reduction. At first glance that's possible. But diagonalization at its core is one machine simulating another machine, or a variety of different machines.

Notice that simulation arguments would still work in the presence of an oracle. We give both the simulating machine and simulated machine the same oracle; the proof goes through. The simulating machine can also ask the same oracle.

Suppose we have a way of proving $P \neq \text{NP}$ with simulating. Then we could prove $P^A \neq \text{NP}^A$ for every oracle A . But this is false! We know $P^A = \text{NP}^A$ for certain oracles! This simple-minded approach doesn't work.



A solution to $P \stackrel{?}{=} \text{NP}$ cannot rely on simulating machines alone, because if it did, by relativization the proof would show that the same is true with any oracle.

Lecture 22

Thu. 11/29/12

Last time we talked about

- $\text{EQ}_{\text{REX}\uparrow}$ is EXPSPACE-complete.

- Oracles

We gave an example of a provably intractable language, and concluded the same technique can't be used to prove $P \stackrel{?}{=} NP$ (relativization).

Today we'll look at a different model of computation that has important applications. We allow Turing machines to access a source of randomness to compute things more quickly than we might otherwise be able to do. We'll talk about

- Probabilistic computation and BPP
- Primality and branching programs

§1 Primality testing

We'll use primality testing as an example of a probabilistic algorithm.

Let

$$\text{PRIMES} = \{p : p \text{ is a prime number in binary}\}.$$

We have $\text{PRIMES} \in \text{coNP}$ (easy). We can write down a short proof in elementary number theory that $\text{PRIMES} \in \text{coNP}$. A big breakthrough in 2002 showed $\text{PRIMES} \in P$.

We'll give a probabilistic, polynomial-time algorithm for PRIMES. We'll just sketch the idea, without going through the details. It is probabilistic in the sense that for each input the running time is polynomial, but there is a small chance that it will be wrong.

We need the following.

Theorem 22.1 (Fermat's little theorem): For any prime p and a relatively prime to p ,

$$a^{p-1} \equiv 1 \pmod{p}.$$

This comes from the abstract algebra fact that if you raise the element of a finite group to the size of the group you get the identity.

For example, if $p = 7$ and $a = 2$, then $2^6 = 64 \equiv 1 \pmod{7}$.

In contrast, if you take $p = 9$, $a = 2$, then $2^8 \equiv 256 \equiv 4 \not\equiv 1 \pmod{9}$. We have just given a proof that 9 is not a prime number: 9 does not have a property that all prime numbers are. However, this proof does not tell you what the factors are. (So primality testing may not help you do any factoring.)

Suppose $a^{p-1} \pmod{p} \neq 1$ for p not prime. This gives an easy test for primality. Unfortunately, this is false. An example is $p = 561 = 3 \cdot 11 \cdot 17$. We have $2^{560} \equiv 1 \pmod{561}$.

We're going to look at something which is still false, but closer to being true. Suppose for p not prime, $a^{p-1} \pmod{p} \neq 1$ for most $a < p$. This would not necessarily give a polynomial time algorithm, because it might give the wrong answer. But you can pick random a 's; each time you pick an a , you have a 50-50 chance of getting a result which is not 1.

To test if p is a prime number, test a hundred random a 's. If you run 100 times and fail, the number is probably prime. But this is also false. For 561, it fails for all a relatively prime to p . This test ignores Carmichael numbers, which masquerade for primes.

But let's assume our heuristic is true.

Then this test works. Let's write the algorithm down. Here is a probabilistic algorithm assuming the heuristic. "On input p ,

1. Pick $a_1, \dots, a_k < p$ at random. (k is the amplification parameter, which allows us to adjust the probability of error.)
2. Compute $a_i^{p-1} \pmod{p}$ for each i .
3. Accept if all results equal 1, and reject any result is not 1."

With our assumption, if p is prime,

$$P(\text{accept}) = 1.$$

If we have a prime number we always get 1 by Fermat's little theorem. But if p is composite, then the probability is going to be small (under the false assumption)

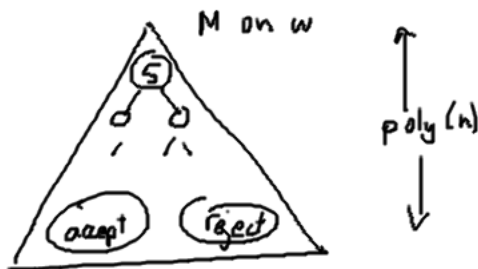
$$P(\text{accept}) \leq 2^{-k}.$$

It's like flipping a coin each time you pick an a . This is our motivating example for making our definition.

§2 Probabilistic Turing Machines

We set up a model of computation—probabilistic Turing machines—which allows us to talk about complexity classes for algorithms like this.

Definition 22.2: A **probabilistic Turing machine** is a type of NTM where we always have 1 or 2 possible moves at each point. If there is 1 move, we call it a **deterministic** move, and if there are 2 moves, we call it a **coin toss**. We have accept or reject possibilities as before.



We consider machines which run in time $\text{poly}(n)$ on all branches of its computation.

Definition 22.3: For a branch b of M on w , we say the probability of B is

$$P(b) = 2^{-\ell}$$

where ℓ is the number of coin toss moves in b . We have

$$P(M \text{ accepts } w) = \sum_{b \text{ accepting branch}} P(b).$$

This is the obvious definition: what is the probability of following b if we actually tossed coins at each coin toss step? At each step there is $\frac{1}{2}$ chance of going off b .

The machine will accept the input with certain probability. Accept some with 99%, 0%, 2%, 50%. We want to say that the probability does the right thing on every input, but with small probability of failing (the error).

Definition 22.4: For a language A , we say that probabilistic TM M **decides A with error probability ε** if for $w \in A$,

$$P(M \text{ accepts } w) \geq 1 - \varepsilon.$$

If $w \notin A$, then

$$P(M \text{ rejects } w) \geq 1 - \varepsilon$$

(i.e., it accepts with small probability, $P(M \text{ accepts } w) \leq \varepsilon$.)

For instance if a machine accept with 1% error, then it accept things in the language with 99% probability.

There is a forbidden behavior: the machine is not allowed to be unsure, for instance accept/reject an input with probability $\frac{1}{2}$. It has to lean overwhelmingly one way or another way. How overwhelming do you want to be? We have a parameter k , which we can apply universally to adjust the error possibility. By repeating an algorithm many times, we can decrease error.

Lemma 22.5 (Amplification lemma): For a probabilistic Turing machine M with error probability ε , with $0 \leq \varepsilon < \frac{1}{2}$, any any polynomial $p(n)$, there is a probabilistic Turing machine M' equivalent to M and has error probability $2^{-p(n)}$.

Not only can we get the error probability small, we can get the probability decreasing rapidly in terms of n .

Proof sketch. M' on w runs M on w $\text{poly}(n)$ times and outputs the majority answer. □

This motivates the following important definition of a complexity class.

Definition 22.6: Define

$$\text{BPP} = \left\{ A : \text{some probabilistic poly-time TM decides } A \text{ with error probability } \frac{1}{3} \right\}.$$

BPP stands for **bounded probabilistic polynomial-time**.

Here, bounded means bounded below $\frac{1}{2}$. The $\frac{1}{3}$ looks like an arbitrary number, but it doesn't matter. Once you have a TM you can make the probability $\frac{1}{10^{100}}$ is you want. All you need about $\frac{1}{3}$ is that $\frac{1}{3} < \frac{1}{2}$.

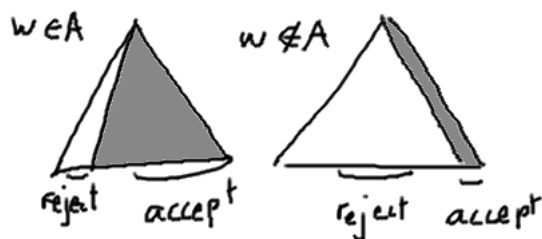
We can prove $\text{PRIMES} \in \text{BPP}$ by souping up the algorithm we described appropriately. Now we know $\text{PRIMES} \in \text{P}$. Obviously $\text{P} \subseteq \text{BPP}$. (A P-algorithm gives the right answer with error 0.) We still don't know $\text{P} \stackrel{?}{=} \text{BPP}$.

In fact most people believe $\text{P} = \text{BPP}$, because of pseudorandomness. If there was some way to compute a value of the coin toss in a way that would act as good as a truly random coin toss, with a bit more work one could prove $\text{P} = \text{BPP}$. A lot of progress has been made constructing pseudo-random generators, but they require assumptions such as $\text{P} \neq \text{NP}$.

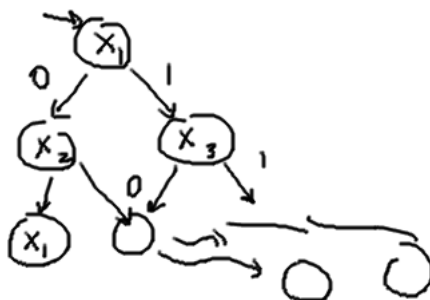
§3 Branching programs

We turn to a bigger example of a problem in BPP that has a beautiful proof. It has an important idea that turned out to be revolutionary in complexity theory.

(A useful picture to keep in mind is the following. Fig 3.)



We need to define our problem.



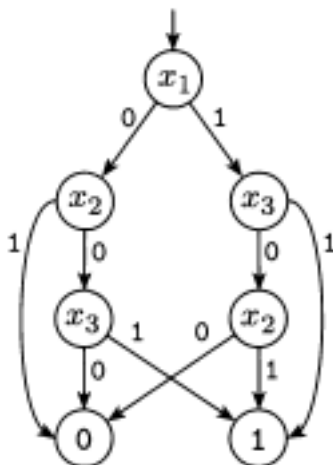
Definition 22.7: A **branching program** (BP) is a directed graph labeled with variable names (possibly repeated) such that the following hold.

- Every node has a label and has 2 outgoing edges 0 and 1, except for two special nodes at the end.
- The 2 special nodes are 0 and 1. (Think of them as the output.)

- There is a special start node, and no cycles.

To use a branching program, make an assignment of the variables to 0's and 1's. Once you have the assignment, put your finger on the start node. Look at the variable at the node. Read the variable's value, and follow 0 or 1 out. An assignment of variables will eventually take you to an output node 0 or 1; that is the output of the program.

Here is a branching program. It computes the exclusive or function.



We want to test whether two different-looking branching programs are equivalent: whether they compute the same function.

Definition 22.8: Define the equality problem for BP's by

$$\text{EQ}_{\text{BP}} = \{ \langle B_1, B_2 \rangle : B_1, B_2 \text{ are BP's and compute some function} \}.$$



This is in coNP: when two BP's are not equivalent, then we can give an assignment on which they differ.

$$\text{EQ}_{\text{BP}} \in \text{coNP}.$$

In fact it is coNP-complete. There's not much more we can say without radical consequences to other things.

We consider a special case that disallows a feature that our first BP has. We disallow reading the same variable twice on any path. Once we've read x_1 , we can't read x_1 again.

Definition 22.9: In a **read-once BP**, each x_i can appear at most once on a path from the start to the output.

Let's look at the problem

$$\text{EQ}_{\text{ROBP}} = \{ \langle B_1, B_2 \rangle : B_1, B_2 \text{ are read-once BP's and compute some function} \}.$$

This is in coNP, but it's not known to be complete. (It is not known to be P, but known to be in BPP. It would probably not be coNP-complete.)

Our main theorem is the following.

Theorem 22.10: $\text{EQ}_{\text{ROBP}} \in \text{BPP}$.

Our first approach is to run the 2 BP's on random inputs. But that's not good enough to give a BPP algorithm: we can only run on polynomially many out of exponentially many input values, and see if they ever do something different. But you can construct branching programs B_1 and B_2 that agree everywhere except at 1 place. They are obviously not equivalent. But if you run them on random input, the chance of finding that disagreement is low. Even if you run polynomially many times, you're likely not to see the disagreement, and you would think they're not equivalent.

We need to make the chance of finding the disagreement at least $\frac{1}{2}$, or some fixed value greater than $\frac{1}{2}$.

Instead we'll do something totally crazy. Instead of setting the x_i 's to 0's and 1's, we'll set them to other values: 2, 3's, etc. What does that mean? The whole problem is to define it. We extend in some algebraic way to apply to nonboolean input, and a single difference gets magnified into an overwhelming difference.

This is worth doing, because the math ideas behind the proof is important.

We'll give a taste of the proof, and finish it next time.

Now x_1 could be given the value 2. We'll blend 0's and 1's together. It uses the following important technique, called **arithmetization**. We want to convert a Boolean model of computation into arithmetic operations that simulate boolean ones.

For instance consider \wedge, \vee . We want to simulate these using arithmetic operations that operate on boolean variables the same way. We want to use $+$, \times but get the same answer.

$$\begin{aligned} a \wedge b &\rightarrow ab \\ \neg a &\rightarrow 1 - a \\ a \vee b &\rightarrow a + b - ab. \end{aligned}$$

Our first step is to convert the branching programs, write it out in terms of and's, or's, and negations. We express the program as a circuit in terms of and's and or's. Then we convert to $+$'s and \times 's, so that the program still simulate faithfully when given boolean inputs, but now has meaning for nonboolean inputs. That's the whole point. There is analysis that we have to work through, but this sets the stage.

Lecture 23

Thu. 10/11/12

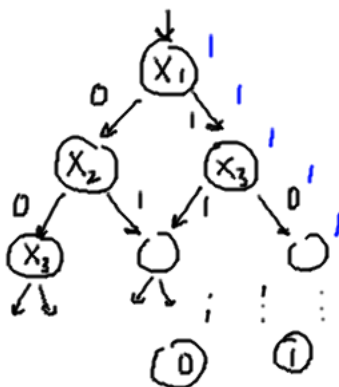
We are going into more advanced topics. Last time we talked about

- probabilistic computation
- BPP

Today we'll see that

- $\text{EQ}_{\text{ROBP}} \in \text{BPP}$.

Unlike PRIMES, this is not known to be in P . A read-once branching program looks like this. (Ignore the blue 1's for now.)



§0 Homework

Problem 1:

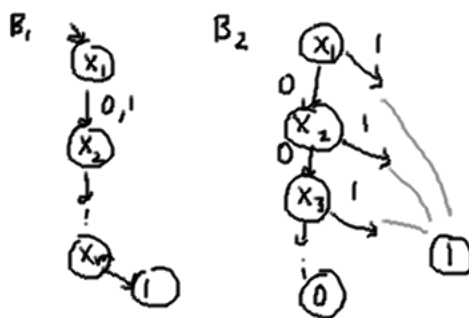
Using padding, we can relate different unsolved problems: EXP vs. NEXP to P vs. NP.

Problem 2:

This is on nondeterministic time hierarchy. It has a short answer, but you have to see what's going on in the proof to see why it doesn't work. There is a nondeterministic time hierarchy, but you need a fancier method of proof, to overcome this problem. A famous paper by Steve Cook shows how to overcome it.

§1 EQ_{ROBP}

In the figure, B_1 is the constant-1 branching program. The only way B_2 can output 0 is if everything is 0. It computes the OR function. B_1 and B_2 almost compute the same function; they agree everywhere except the place where everything is 0.



If we run the programs on polynomially many input, the chance that we land on the single input where they differ is exponentially small. We need exponentially many of them to have a shot. The straightforward algorithm for testing equivalence is not going to run in polynomial time.

The key idea is to run on other numbers, using arithmetization. This technique is also used in error-correcting codes and other fancy things.

We will simulate \wedge, \vee with $+, \times$.

$$a \wedge b \rightarrow ab$$

$$\bar{a} \rightarrow 1 - a$$

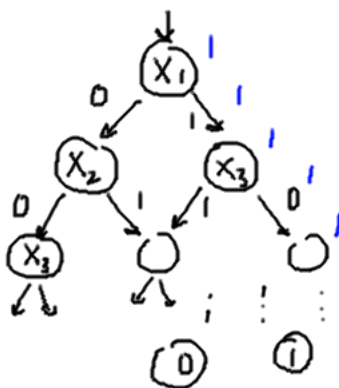
$$a \vee b \rightarrow a + b - ab \quad \rightarrow a + b \text{ if } a, b \text{ not both } 1.$$

(For $a \vee b$, we can use $a + b$ if a, b are never both 1.)

We first rerepresent branching program with and's and or's.

Let's think about running the branching program slightly differently. It is a boolean evaluation: we give a boolean assignment to every one of the nodes and edges. Put a 1 on every node and edge that you follow and 0 on everything you don't.

Every path starts at x_1 , so we assign the node with 1. Let's say $x_1 = 1$; we write 1 on the edge going to x_3 and 0 on the edge x_2 to say we didn't go that way. Put 1 on x_3 . Let's say x_3 is 0. Then we put 1 on the 0-path from x_3 . Everything else is 0.

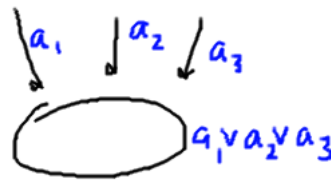


The advantage is that we can write a boolean expression corresponding to what 1's we write down. Suppose we have a node x_i .

We need a boolean expression to say which edge we went down. On the right side we'll put $a \wedge x_i$. Why does that make sense? The only way we'll go down that path is if we went through x_i and $x_i = 1$. On the 0 edge we write $a \wedge \bar{x}_i$.



This tells us how to label the edges. How do we label the nodes? Suppose a_1, a_2, a_3 label edges going to a node. Then we label the node with $a_1 \vee a_2 \vee a_3$. The start gets 1 and the output is the value of the 1 node (a = output).

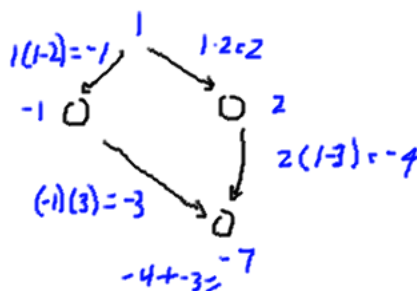


Now let's redo it with $+$ and \times instead of \vee and \wedge . There are no cycles; the path can enter every node in exactly one way. Thus we never have more than 1 a_i set to 1. Thus for the "or" we don't need the correction term, and we can just add, $a + b$.



Using the arithmetization, we can assign non-Boolean values, and there is perhaps some nonsensical result that comes out. Remember that we wrote down the branching program for parity (exclusive or), for instance, $x_1 \oplus x_2$. Have you ever wondered what $2 \oplus 3$ is? Probably not. Let's plug in $x_1 = 2$ and $x_2 = 3$ into the arithmetized version of this branching program.

Let's see what happens. Plug in 1 at the start node. If we assigned $x_1 = 0$ and $x_2 = 1$, everything work out as before. But now can give values even if we don't have 0's and 1's coming in. We get the following values.



We get $2 \oplus 3 = -7$. (We haven't discovered some fundamental fact about exclusive or. There's no fundamental meaning to this.)

Let's summarize. Originally we thought of running a BP as following some path. That way of thinking doesn't lend itself to arithmetization. Instead of thinking about taking a path, think about evaluating a branching program by assigning values to all nodes by the procedure, and looking at 1 node. There is no path, but this way of thinking is equivalent. We can look at the value on the output node even if the input nodes didn't have 0/1 values coming in.

If we had a different branching program representation of the same boolean formula (say, xor), would we get different value? No. As you will see from the coming proof, if we have a different representation that is still read-once, and it agrees on the boolean case, then it agrees on the non-boolean case. This is not true with a general branching program!

As an example, if we flip x_1, x_2 in the xor program we get the same value for $2 \oplus 3$.

Finally, here is the probabilistic algorithm.

Proof. Let $M = \text{"on } \langle B_1, B_2 \rangle$,

1. Randomly select non-Boolean values for x_1, \dots, x_m from the finite field $\mathbb{F}_q = \{0, 1, \dots, q-1\}$ where q is prime (this is modular arithmetic modulo q). Choose $q > 3m$.
2. Compute B_1, B_2 (arithmetized) on x_1, \dots, x_m .
3. Accept if we get the same output value. Reject if we do not."

Now we have to prove this works.

We claim the following.

1. If B_1, B_2 are equivalent then $P(M \text{ accepts}) = 1$. (If they agree then they agree on boolean values. We'll prove they agree even on nonboolean values.)
2. If B_1, B_2 are not equivalent then $P(M \text{ rejects}) \geq \frac{2}{3}$.

We prove statement 1. This is the hard part.

We take the input variables and keep them as variables x_i ; do the calculation symbolically. We'll write down expressions like $x_1, 1 - x_1$, and so forth. At every step we're multiplying things like x_i or $(1 - x_i)$, or adding together terms. At the output node 1 we have some polynomial in x_1, \dots, x_m .

Evaluating B_1, B_2 symbolically in the arithmetized version, we get polynomials P_1, P_2 on x_1, \dots, x_m . These polynomials have a very nice form: They all look like products of x_i 's and $(1 - x_i)$'s, added up, for instance

$$\begin{aligned} & x_1(1 - x_2)x_3x_4(1 - x_5) \cdots x_m \\ & + (1 - x_1)x_2x_3(1 - x_4) \cdots \\ & + \cdots \end{aligned}$$

In each summand, we never get same variable appearing more than once because of the read-once property. How do we know we get every variable appearing once? We can always pad out a branching program by adding missing variables, to turn it into a “read exactly once” branching program.

Both P_1, P_2 look like this. Why is it nice? *It's the truth table of the original program on Boolean values.* The summands give the rows of the Boolean truth table. If two BP's agree in the Boolean world, they have the same truth table and hence the same polynomial, and agree everywhere.

(There is an exponential number of rows, but this doesn't matter: when we run the algorithm, we don't calculate the polynomial, which takes exponential time. We get a specific value of the polynomial by plugging in values and computing things on the fly.)

Part 2 uses a magical fact about polynomials.

Lemma 23.1: If $P(x)$ is a nonzero polynomial of degree at most d , then $P(x)$ has at most d zeros. (This is true in any field, in particular \mathbb{F}_q .)

The probabilistic version is the following: if you pick $x \in \mathbb{F}_q$ at random, $\text{Prob}(P(x) = 0) \leq \frac{d}{q}$. Lemma 2 is the multivariate version.

Lemma 23.2 (Schwartz-Zippel): **lem:schwartz-zippel** If $P(x_1, \dots, x_m)$ is nonzero, each x_i has degree at most d , and you pick $x_i \in \mathbb{F}_q$ randomly, then

$$\text{Prob}[P(x_1, \dots, x_m) = 0] \leq \frac{md}{q}.$$

This is proved from the single-variable case by induction.

Remember we had 2 polynomials P_1, P_2 ? Let's look at the difference $P := P_1 - P_2$. If the branching programs are not equivalent, then the difference of the polynomials is nonzero. That nonzero polynomial has few roots. P is zero in very few places, so P_1, P_2 agree in very few places. When we run the arithmetization of P_1, P_2 , we're unlikely to get the same value coming out. It's very likely we'll get different values coming out, and very likely we'll reject.

For our $P = P_1 - P_2$, what is d ? Every variable appears once. Hence $d = 1$. m is the number of variables, and $q > 3m$, so the probability is at most $\frac{1}{3}$. The chance we got an agreement in P_1, P_2 is at most $\frac{1}{3}$. The chance we get disagreement is at least $\frac{2}{3}$. \square



Though arithmetization—converting boolean formulas to a polynomial and then running on randomly selected nonboolean formula—we can magnify the chance that a probabilistic algorithm works.

This is a nice probabilistic algorithm. We'll use this method again in the last two lectures, where we'll prove amazing results about satisfiability using interactive proof systems.

Lecture 24

Thu. 12/6/12

The final exam is Wednesday December 19, 9-12 in Walker. It is open book, notes, and handouts. It covers the whole semester with emphasis on the 2nd half. It is a 3-hour version of the midterm with some short-answer questions.

Handout: sample questions.

Last time we showed $\text{EQ}_{\text{ROBP}} \in \text{BPP}$ and today we'll talk about

- Interactive Proofs
- IP

§1 Interactive proofs

1.1 Example: Graph isomorphism

We'll move into the very last topic, an amazing idea: the interactive proof system. It's a probabilistic version of NP, the same way BPP is a probabilistic version of P. Another amazing thing is that it goes against the intuition of NP we built up during the term: If a problem is in NP, it has short certificates, so a prover can convince a verifier about a certain fact, membership in the language.

Using the idea of interactive proof, a prover can convince a verifier about a certain fact even though there are no short certificates.

We'll start this off with a famous example: testing whether or not graphs are isomorphic:

$$\text{ISO} = \{ \langle G_1, G_2 \rangle : G_1, G_2 \text{ graphs}, G_1 \equiv G_2 \}.$$

Two graphs are **isomorphic** iff we can match up the nodes so that edges go between corresponding nodes. It is clear $\text{ISO} \in \text{NP}$: just give the matching. It is one of the rare (combinatorial) problems in NP that is neither known to be in P nor NP-complete. Almost every other problem is either known to be in P or NP-complete, except for bunch of problems related to number theory.

The graph isomorphism is the most famous such problem. There is a huge literature trying to prove it one way or other, with no success yet. Define $\text{NONISO} \in \overline{\text{ISO}}$.

Is $\text{NONISO} \in \text{NP}$? Not known. It seems one has to astronomically search through all permutations to determine non-isomorphism. Is there a short certificate, or do you essentially have to go through same process again?

There is way for you to convince me of the fact provided you have sufficient computational power at your disposal. Here is a whimsical version of interactive proof system: The prover has unlimited computational power but is not trustworthy. The verifier checks the prover. The prover is like army of slaves, also called graduate students. The verifier is the king, sometimes called the professor. The grad students (slaves) stay up all night, and have unlimited computational power. The professor only operates in probabilistic polynomial time. The professor has a research problem: are these graphs isomorphic? The grad students get to work, with their fancy computers. They find: Yes, they're isomorphic! The professor knows that grad students basically honest folks, but they have lots of other things worry about, like XBox. The prof needs to be convinced, and be sure what answer is. If the grad students say yes, the professor says: convince me, and the grad students give the isomorphism. Suppose the grad students say the graphs are nonisomorphic. The professor asks for a proof.

There is a simple protocol they can go through with the professor to convince this professor that the graphs are non-isomorphic. This was established back in mid-1980's. Laszlo Babi, a leading expert in graph isomorphism, was flabbergasted.

Both the professor and students have access to the 2 graphs. The professor takes 2 graphs, turns around secretly, chooses one of G_1, G_2 at random, and randomly permutes the vertices. The professor asks, "Is the graph I picked G_1 or G_2 ?" If the grad students can answer reliably, then they must be nonisomorphic. If they are isomorphic, it could have come from either one, and there is no way to tell which one the prof picked; the best thing one can do is guess. If the graphs really were different, the students can use a supercomputer to guess which one professor the picked: The graph can only be isomorphic to one of G_1, G_2 .

The professor does this several times. If the students can answer the question 100 times correctly in a row, either they are legitimately doing the protocol, or they're incredibly lucky.

In fact, interactive proof systems can show formulas are unsatisfiable. The proof is more complicated. This gives all of coNP doable with interactive proof system!

We know $\text{ISO} \in \text{NP}$, but we don't know whether $\text{NONISO} \in \text{NP}$. But we can convince someone of non-isomorphism if we have enough computational power. We extend NP to a bigger class, where we can convince a verifier of membership in languages beyond NP.

Interactive proof systems play a big role in cryptography: here the prover is limited in some way, but has special information (a password), and has to convince someone that he has the password without revealing the password.

1.2 Formal model of interactive proofs

We write down a formal model.

Definition 24.1: Let P be a prover with unlimited computational power. Let V be a verifier with probabilistic polynomial time computational power. Let $(P \leftrightarrow V)$ be an interaction

where P and V exchange polynomially many messages (both given input w) until V says accept or reject.

We say that $A \in \text{IP}$ if there are V and P where for every w , $w \in A$,

$$\text{Prob}[(P \leftrightarrow V) = \text{accept}] \geq \frac{2}{3}$$

and for $w \notin A$, for every \tilde{P} ,

$$\text{Prob}[(\tilde{P} \leftrightarrow V) = \text{reject}] \geq \frac{2}{3}.$$

To show a language is in IP, we set up a verifier and prover. For every string in the language, working together, the prover gets the verifier to accept with high probability. If the string is not in language, then no matter what prover you choose (\tilde{P} is cheating prover trying to make the verifier accept when she shouldn't), rejection is the likely outcome.

Theorem 24.2: $\text{NONISO} \in \text{IP}$.

Proof. We write the NONISO protocol with this setup in mind. On input $\langle G_1, G_2 \rangle$,

V: Choose G_1 or G_2 at random. Then randomly permute and send result to P .

P: Replies: which G_i did V choose?

Repeat twice.

V: Accept if P is correct both times.

Reject if P is ever wrong.

If $G_1 \neq G_2$ then

$$\text{Prob}[(V \leftrightarrow P) \text{ accepts}] = 1$$

The honest prover can tell which G_i the verifier picked by detecting whether it is isomorphic to G_1 or G_2 .

The honest prover only in play when $\langle G_1, G_2 \rangle$ is in the language. Now the sneaky prover steps in: I'll take a shot at it. If $G_1 \neq G_2$, then the sneaky prover (pretending $G_1 \equiv G_2$) can't do anything, can only guess. The probability it guesses right twice is $\frac{1}{4}$.

Thus if $G_1 \equiv G_2$, then for any \tilde{P} ,

$$\text{Prob}[(V \leftrightarrow \tilde{P}) \text{ accepts}] \leq \frac{1}{4}.$$

This shows $\text{NONISO} \in \text{P}$. □

Proposition 24.3: $\text{NP} \subseteq \text{IP}$.

$\text{BPP} \subseteq \text{IP}$

Proof. For $\text{NP} \subseteq \text{IP}$, the prover sends the certificate to the verifier. This is just a 1-way conversation. The verifier checks the certificate.

For BPP, the verifier doesn't need the prover. The verifier can do all by his or her lonesome self. □

§2 IP=PSPACE

Now we'll prove the amazing theorem. This blew everything away when it came out, I remember that.

Theorem 24.4: IP=PSPACE.

What does this mean? Take the game of chess, some game where you can test in polynomial space which side has a forced win. It takes an ungodly amount of time to go through the search tree, but in relatively small space you can show (say,) white has a forced win. There is probably no short certificate, but if Martians with supercomputers have done all computations, they could convince mere probabilistic time mortals like us that white has a forced win without us going through the entire game tree.

We'll prove a weaker version, $\text{coNP} \subseteq \text{IP}$. This was discovered first, contains pretty much all the ideas, and is easier to describe. The full proof of IP=PSPACE is in the textbook.

It's enough to work with satisfiability, show the prover can convince the verifier that a formula is not satisfiable.

The amazing thing is the technique. We'll use arithmetization as we did before.

2.1 Aside

Every few months I get a email or letter claiming to prove $P=NP$. The first thing I look is whether $P=NP$. If the person claims $P=NP$, I don't even look at it. It is probably some horrible algorithm with accompanying code.

I tell them, then you can factor numbers. Here's a website with various numbers known to be composite, where no one knows the factorization. Just factor one of them. That usually shuts them up, and I never hear from them again.

If $P \neq NP$, then almost without exception, their proof goes like this. They claim, clearly any algorithm for SAT, etc. has to operate in the following way... Then they give a long analysis that shows it has to be an exponential algorithm. The silly part is the "clearly." That's the whole point. How do you know you can't do something magical; plug the input through a Fourier transform and do some strange things, and have the answer pop out.

You have to prove no such crazy algorithms exist.

The cool thing about the IP protocol is that it does something crazy and actually works.

2.2 $\text{coNP} \subseteq \text{IP}$

Proof of $\text{coNP} \subseteq \text{IP}$. For a formula ϕ let $\#\phi$ be the number of satisfying assignments of ϕ . Note $\#\phi$ will immediately tell you whether ϕ is satisfiable.

Define **number-SAT** (sharp-SAT) by

$$\#\text{SAT} := \{ \langle \phi, k \rangle : \#\phi = k \}.$$

This is not known to be in NP. (It would be in NP for small k . However, if there are exponentially many satisfying assignments, naively we'd need an exponential size certificate.)

However, we show

$$\#SAT \in \text{IP}.$$

We'll set up a little notation. Fix ϕ . Let

$$\phi(x_1, \dots, x_m) = \begin{cases} 0, & \text{unsatisfying} \\ 1, & \text{satisfying} \end{cases}$$

Let

$$T() = \sum_{x_i \in \{0,1\}} \phi(x_1, \dots, x_m).$$

Note $T() = \#\phi$ is the number of satisfying assignments. Add 1 every time satisfying, 0 if not satisfying assignments.

Define

$$T(x_1, \dots, x_j) = \sum_{x_i \in \{0,1\}, i > j} \phi(x_1, \dots, x_m).$$

We are presetting some of the values of the formula, and counting the number of satisfying assignments subject to those presets. Thus

$$T(x_1, \dots, x_j) = \#\phi_{x_1, \dots, x_j}$$

where $\phi_0 = \phi$ with $x_1 = 0$, $\phi_{01} = \phi$ with $x_1 = 0, x_2 = 1$, and so forth. In particular, since we assign values to all of the x_i values, $T(x_1, \dots, x_n)$ is 0 or 1.

We have the following relations.

$$\begin{aligned} T() &= \#\phi \\ T(x_1, \dots, x_m) &= \phi(x_1, \dots, x_m) \\ T(x_1, \dots, x_j) &= T(x_1 \dots x_j 0) + T(x_1 \dots x_j 1). \end{aligned}$$

To see the last equation, note the number of satisfying assignments with x_1, \dots, x_j is the sum of the number of satisfying assignments additionally satisfying $x_{j+1} = 1$ and the number of satisfying assignments additionally satisfying $x_{j+1} = 0$, because one of these has to be true.

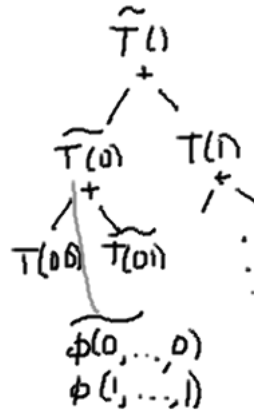
We set up the #SAT protocol. (Our first version will have a little problem, as we will see.) Suppose the input is $\langle \phi, k \rangle$. The prover is supposed to make the verifier accept with high probability.

0. P: Sends $T()$, V checks $k = T()$. (Reject if things don't check out.)
1. P: Sends $T(0)$ and $T(1)$. V checks that $T() = T(0) + T(1)$.
2. P: Sends $T(00), T(01), T(10), T(11)$. V checks $T(0) = T(00) + T(01)$ and $T(1) = T(10) + T(11)$. (This is exponential, which is a problem. But humor me.)
- \vdots

m . P: Sends $T(0 \dots 0), \dots, T(\underbrace{1 \dots 1}_m)$. V checks $T(\underbrace{0 \dots 0}_{m-1}) = T(\underbrace{0 \dots 0}_{m-1}0) + T(\underbrace{0 \dots 0}_{m-1}1), \dots,$
 $T(\underbrace{1 \dots 1}_{m-1}) = T(\underbrace{1 \dots 1}_{m-1}0) + T(\underbrace{1 \dots 1}_{m-1}1).$

$m + 1$. V checks $T(0 \dots 0) = \phi(0 \dots 0), \dots, T(1 \dots 1) = \phi(1 \dots 1)$, and accepts if all these are equal.

Think of this as a tree.



This algorithm might seem trivial, but it's important to understand the motivations.

An honest prover sends correct input. Suppose we have a dishonest prover: If k is wrong, the prover tries to convince the verifier to accept anyway. The prover sends wrong value for $T()$.

This is like asking a kid questions, trying to ferret out a lie. One lie lead to other lies. (But to the kid things may look locally consistent...) There must be a lie on at least one of two branches. At least one lie must propagate down at each step, all the way to a lie at the bottom, which the verifier catches.

The only problem is the exponential tree. You can imagine trying to do something probabilistic. Instead of following both branches, let's pick a random branch to follow. You're a busy parent. You can't check out all possible things your kid is saying. Pick one. Choose one branch. But you want a high probability of detecting the cheating. If you a pick random branch, with 50-50 chance, as soon as you get off the lying side we get to the honest side, prover is saved. The prover thinks, "You're not going to catch me now," and behaves honestly all the way down.

The dishonest prover should only make the verifier accept with low probability.

Instead we pick non-boolean values. We arithmetize the whole setup, and reduce to one randomly chosen non-boolean case. We only have to follow a single line of these non-boolean values down. Again we rely on the magic of polynomials.

If the prover lied, then in almost all of the non-boolean values we could pick, there will be a lie. A lie leads to another lie almost certainly. The rest of the protocol is set up in

terms of arithmetization. Arithmetize everything and everything just works. We finish next time. \square

Lecture 25

Tue. 12/11/2012

Last time we talked about

- interactive proofs
- IP.

Today we'll finish the proof of $\text{coNP} \subseteq \text{IP}$.

A prover with unlimited computational power tries to convince a verifier that a string is in the language. For a string in the language, the prover will convince the verifier with high probability. For a string not in the language, that prover, or any other prover, will fail with high probability.

The big result is $\text{IP} = \text{PSPACE}$; we prove a weaker form $\text{coNP} \subseteq \text{IP}$. (It was around half a year before Adi Shamir got trick to go from $\text{coNP} \subseteq \text{IP}$ to $\text{IP} = \text{PSPACE}$.)

§1 $\text{coNP} \subseteq \text{IP}$

Last time we introduced an exponential protocol for $\#\text{SAT}$, a coNP -hard problem.

This protocol doesn't use the full power of IP. It is a one-way protocol, like NP: the verifier doesn't send the prover any questions.

Using arithmetization, we find a polynomial that faithfully simulates the expression when we plug in 0's and 1's. The degree of the polynomial is not too big.

$$\begin{aligned} a \wedge b &\rightarrow ab \\ \bar{a} &\rightarrow 1 - a \\ a \vee b &\rightarrow a + b - ab \\ \phi &\rightarrow P_\phi(x_1, \dots, x_m) \end{aligned}$$

The total degree of the polynomial will be at most the length n of ϕ : when we combine two expressions the degrees will at most be the sum.

Instead of reducing the verification of one T -value to two T -values, we reduce it to one T -value but one that is non-boolean. The formulas will have other values when you plug in

other values.

$$\begin{array}{c}
 T() = k \\
 \downarrow \\
 T(?) \\
 \downarrow \\
 T(?, ?) \\
 \downarrow \\
 T(?, ?, ?)
 \end{array}$$

We arithmetize P . P looks just like it looks before, but instead of using the formula, we use the polynomial that represents the formula:

$$T(x_1, \dots, x_i) = \sum_{x_{i+1}, \dots, x_m \in \{0,1\}} P_\phi(x_1, \dots, x_m).$$

If we preset to 0's and 1's, we get the same value because the polynomial agrees with the boolean formula on boolean values.

If we preset nothing, there is no change: $T()$ is the number of satisfying assignments. Everything is added up over booleans. If we set everything, we have possibly non-boolean values, and

$$T(x_1, \dots, x_m) = P_\phi(x_1, \dots, x_m).$$

We now give the protocol. This is where the magic happens. We'll work over some finite field \mathbb{F}_q , where $q > 2^n$. The reason we make it so big is that k can be a value between 0 and 2^n . We will have wraparound issues if we use a field that can't represent all these possible values.

0. P sends $T()$. V checks $k = T()$.
1. P sends $T(z)$ as a polynomial in the variable z . (More formally, P sends the coefficients. Note that the degree in z is at most $|\phi|$. Each number has at most m bits, and there are at most $|\phi| + 1$ coefficients. Of course calculating this is difficult, but that's okay: this is the prover. The grad students work hard. They don't get paid for their work beyond their stipend, which is polynomial, so doesn't matter. They send an answer which is polynomial.)

V checks $T(0)$ and $T(1)$ are correct by checking

$$T() = T(0) + T(1).$$

Note the nice thing is that *one object* allows us to get two values. This will prevent the blowup.

V sends a random $r_1 \in \mathbb{F}_q$. The prover now has to show $T(r_1)$ is correct.

2. P sends $T(r_1, z)$ as polynomial in z . (P convinces V that $T(r_1)$ is correct.)

V checks $T(r_1) = T(r_1, 0) + T(r_1, 1)$.

\vdots

m . P sends $T(r_1, \dots, r_{m-1}, z)$ as a polynomial in z . V checks $T(r_1 \cdots r_{m-1}) = T(r_1 \cdots r_{m-1}, 0) + T(r_1 \cdots r_{m-1}, 1)$. V chooses random $r_m \in \mathbb{F}_q$.

$m + 1$. V checks $T(r_1, \dots, r_m) = P_\phi(r_1, \dots, r_m)$, and if so, accepts.

How does the verifier check at the end stage it's okay? Plug it into P_ϕ .

$$\begin{array}{c}
 T() = k \\
 \downarrow \\
 T(r_1) \\
 \downarrow \\
 T(r_1, r_2) \\
 \downarrow \\
 \vdots \\
 \downarrow \\
 T(r_1, \dots, r_m) \equiv P_\phi(r_1, \dots, r_m)
 \end{array}$$

The honest prover will make the verifier accept with probability 1. Just follow the protocol: send the correct polynomials.

The verifier says, "Convince me the polynomial is right by convince me it works on some random element."

Why does this work? Why are we using polynomials? Let's see what happens when the prover tries to lie. If k is wrong the verifier will reject with high probability. In order to preserve any hope of making the verifier accept, the prover has to lie. If $T()$ is a lie, then one of $T(0)$, $T(1)$ has to be wrong. But these came from the same polynomial, by plugging in 0, 1. So the polynomial is wrong. We evaluate that wrong polynomial at a random input. But 2 low-degree polynomials agree in only a small number of locations, because a polynomial of low degree has only a small number of roots. Every place where the polynomial is 0 becomes a zero of the difference polynomials (between the actual and claimed polynomial).

$T(r_1)$ doesn't necessarily have to be a lie. But that's very unlikely: a small number of agreements (roughly n) out of exponentially many possibilities.

$T(r_1)$ is almost certainly wrong. The dishonest prover tries to convince the verifier that $T(r_1)$ is right. The prover again has a chance again of getting lucky: if the verifier picks a place where the incorrect and correct polynomial agree. But every step, it's hard to succeed. Almost certainly $T(r_1)$ incorrect forces $T(r_1 r_2)$ incorrect, and so forth.

1.1 Analysis of protocol

If $\langle \phi, k \rangle \in \#SAT$, then

$$\text{Prob}(V \leftrightarrow P \text{ accepts}) = 1.$$

If $\langle \phi, k \rangle \notin \#SAT$, then by the Schwartz-Zippel Lemma 23.2, for any prover \tilde{P} ,

$$\text{Prob}(V \leftrightarrow \tilde{P} \text{ accepts}) \leq m \cdot \frac{\deg P_\phi}{q} = m \frac{n}{2^n} = \frac{\text{poly}(n)}{2^n}.$$

The prover has m chances to get lucky. If it gets lucky, it follows the original protocol: just send the correct values all the way down. The probability of getting lucky at one stage is the degree of the polynomial divided by the size of the field q . This is small.

This shows $\#SAT \in \text{IP}$, and hence $\text{coNP} \subseteq \text{IP}$.

§2 A summary of complexity classes

