18.404: Theory of Computation

Brice Huang

Fall 2016

These are my lecture notes for the Fall 2016 iteration of 18.404, Theory of Computation, taught by Prof. Michael Sipser.

These notes are written in LaTeX during lectures in real time, and may contain errors. If you find an error, or would otherwise like to suggest improvements, please contact me at bmhuang@mit.edu.

Special thanks to Evan Chen and Tony Zhang for the help with formatting, without which this project would not be possible.

The text for this course is Introduction to the Theory of Computation by Michael Sipser.

These notes were last updated 2016-12-18. The permalink to these notes is `http://web.mit.edu/bmhuang/www/notes/18404-notes.pdf`.

Contents

1 September 8 and 13, 2016 **6**

2 September 15, 2016 **7**

2.1 Finite Automata . . . 7

2.2 Non-regular languages . . . 8

3 September 20, 2016 **10**

3.1 Context-Free Languages . . . 10

3.2 Ambiguity . . . 11

3.3 Push-Down Automata . . . 12

3.4 CFLs and PDAs . . . 13

4 September 22, 2016 **14**

4.1 Closure Rules for CFLs . . . 14

4.2 Pumping Lemma for CFLs . . . 15

4.3 Applications of Pumping Lemma for CFLs . . . 15

4.4 Aside: Intersections of Context-Free Languages are not Context-Free 16

4.5 Turing Machines . . . 16

5 September 27, 2016 **18**

5.1 Why Turing Machines? . . . 18

5.1.1 Example: Multitape Turing Machines . . . 18

5.1.2 Example: Nondeterministic Turing Machines . . . 19

5.2 Turing-Enumerators . . . 19

5.3 Church-Turing Thesis . . . 20

6 September 9, 2016 **21**

7 October 4, 2016 **22**

7.1 Undecidability . . . 22

7.2 Diagonalization . . . 22

7.3 Undecidability of A_{TM} . . . 24

7.4 Consequences of the undecidability of A_{TM} . . . 25

7.5 Introduction to Reductions . . . 25

8 October 6, 2016 **27**

8.1 Mapping Reducibility . . . 27

8.2 Non Turing-recognizable languages . . . 28

8.3 Post-Correspondence Problem . . . 29

9 October 13, 2016 **30**

9.1 Administrivia . . . 30

9.2 Review . . . 30

9.3 Post Correspondence Problem . . . 30

9.3.1 Linearly Bounded Automata . . . 31

9.3.2 Reduction of Post-Correspondence Problem . . . 32

10 October 18, 2016 **34**

10.1 Review: Computation History Method . . . 34

10.2 Recursion Theorem . . . 35

10.3 Applications of Recursion Theorem . . . 36

11 October 20, 2016 **37**

11.1 Complexity Theory: Introduction . . . 37

11.2 Complexity Theory and Models . . . 38

11.3 The Class P . . . 39

12 October 25, 2016 **41**

12.1 The class NP . . . 41

12.2 P vs NP . . . 41

12.3 More Examples . . . 42

12.4 Satisfiability Problem . . . 43

13 November 1, 2016 **44**

13.1 Polynomial-Time Reductions . . . 44

13.2 Reduction Example: $3SAT$ and $CLIQUE$ . . . 44

13.3 NP-Completeness . . . 45

13.4 Another Reduction: $3SAT$ and $HAMPATH$ . . . 46

14 November 3, 2016 **47**

14.1 Implications . . . 47

14.2 Proof of Cook-Levin . . . 47

15 November 8, 2016 **50**

15.1 Godel's Letter . . . 50

15.2 Space Complexity . . . 50

15.3 Quantified Boolean Formulas . . . 51

15.4 Word Ladders . . . 52

16 November 10, 2016 54

16.1 The Ladder Problem . . . 54

16.2 Proof of Savitch's Theorem . . . 55

16.3 PSPACE-completeness . . . 56

16.4 $TQBF$ is $PSPACE$-complete . . . 56

17 November 15, 2016 58

18 November 17, 2016 59

18.1 Last Time . . . 59

18.2 Log-space and P . . . 59

18.3 NL-completeness . . . 60

18.4 coNL . . . 62

19 November 22, 2016 63

19.1 Motivation and Outline . . . 63

19.2 Hierarchy Theorems . . . 63

19.2.1 Space Hierarchy Theorem . . . 63

19.2.2 Time Hierarchy Theorem . . . 64

20 November 29, 2016 66

20.1 Natural Intractable Problems . . . 66

20.2 Oracles and Relativization . . . 68

21 December 1, 2016 69

21.1 Oracles and Relativization, continued . . . 69

21.2 Probabilisitic Computation . . . 69

21.3 BPP . . . 70

21.4 Branching Programs . . . 70

22 December 6, 2016 72

22.1 BPs and Arithmetization . . . 72

22.2 Digression: Algebra . . . 72

22.3 Arithmetization . . . 73

23 December 8, 2016 78

23.1 BPP and RP . . . 78

23.2 ISO . . . 78

23.3 Interactive Proofs . . . 78

23.3.1 Example: ISO and non-ISO . . . 79

23.3.2 Formal Definition . . . 79

23.3.3 Analysis of non-ISO interactive proof . . . 80

23.3.4 $\#SAT$, $IP = PSPACE$ . . . 80

24 December 13, 2016 **81**

24.1 Administrivia 81

24.2 Last Time 81

24.3 A polynomial time IP protocol for #SAT 81

24.3.1 Arithmetization 82

24.3.2 The IP Protocol 82

1 September 8 and 13, 2016

I missed these lectures. The content of these lectures can be found in Sections 1.1-1.2 of the text.

2 September 15, 2016

Last time:

- Nondeterminism, NFA;
- $U \circ *$ closures;
- Regular expression $\to$ NFA.

Today:

- DFA $\to$ regular expressions (this shows that DFAs are equivalent to regular expressions);
- Proving non-regularity;
- Context-Free Grammars.

2.1 Finite Automata

We will prove the following theorem:

Theorem 2.1
If A is regular, then $A = L(R)$ for some regular expression R.

Here, $L(R)$ is the set of all strings generated by a regular expression R.

We will prove this by converting a DFA recognizing the language A into a regular expression R such that $L(R) = A$. We first define:

Definition 2.2. A **generalized NFA** (GNFA) is an NFA that allows regular expressions on transition arrows.

We will actually show how to convert a GNFA into a regular expression. Since all DFAs are GNFAs, this proves our theorem.

We start with a slightly weaker construction:

Lemma 2.3
A generalized NFA can be converted into a regular expression, provided that

- There is a single accept state that is not the start state.
- There is a transition from every state to every other state, except that the start state has only out-transitions, and the accept state has only in-transitions.

Proof. We proceed by induction on k, the number of states in G.

Base case: $k = 2$. The resulting GNFA has just two states, the start and end states, joined by a transition labeled r. Then we just take as our regular expression $R = r$.

Induction step: $k > 2$; assume we know how to do the construction for $k - 1$. Consider a state x that isn't the start or accept state. We rip x out of G with all its transitions, leaving a machine with $k - 1$ states.

To repair the damage of removing x: if, in the old DFA, the transitions $q_i \to x$, $x \to x$, $x \to q_j$, $q_i \to q_j$ are labeled by the regular expressions r_1, r_2, r_3, r_4, we $q_i \to q_j$ with $r_1(r_2) * r_3 \cup r_4$ – this way, any string that causes the old GNFA to go from q_i to q_j, directly or via x, causes the new GNFA to also go from q_i to q_j. Performing this process on all pairs of nodes q_i, q_j repairs the damage of removing x. By induction we are done. □

What about languages not of the form described in the lemma? It turns out we can manipulate all generalized NFAs into this special form. To do this, we create two new nodes: a super-start node and a super-accept node. The super-start node has an outgoing ϵ-transition to the old start state and outgoing $\emptyset$-transitions to everywhere else; the super-accept node has incoming ϵ-transitions from the old accept states, and incoming $\emptyset$-transitions from everywhere else. For any other pair of states q_i, q_j, if there is no transition from q_i to q_j, add an $\emptyset$-transition.

Remark 2.4. We can now convert $GNFA \to$ regex $\to NFA \to DFA \to GNFA$, so all of these are equivalent.

2.2 Non-regular languages

The language

$$B = \{w | w \text{ has equal numbers of 0s and 1s}\}$$

is not regular. Intuitively, this is because the count of the difference in the numbers of 0s and 1s can be arbitrarily large, and we can't keep infinitely many states. However, intuition can be wrong – as an example,

$$C = \{w | w \text{ has equal numbers of 01s and 10s}\}$$

is in fact regular.

The point is, "I couldn't think of a finite automaton" isn't enough to prove a finite automaton doesn't exist. We will build a methodology of proving a language cannot be regular by determining a property that all regular languages must satisfy.

Lemma 2.5 (Pumping Lemma)
For any regular language A, there is a number p (the **pumping length**) so that if $s \in A$ and $|s| \geq p$, then we can write $s = xyz$, where:

- $xy^iz \in A$ for any $i \geq 0$;
- $y \neq \epsilon$;
- $|xy| \leq p$.

Informally: any long string can be cut up in such a way that the middle section can be repeated as many times as we want. Or, "all long strings can be pumped."

Remark 2.6. The pumping lemma provides a necessary, *but not sufficient*, condition for a language to be regular. There are non-regular languages that will satisfy the condition of the pumping lemma.

Proof. Suppose the DFA M recognizes A, and let M have k states. Take $p = k + 1$. Any word in A of length $\geq p$ corresponds to a walk of length $\geq p$ among the states of A. This walk has length longer than the number of states of M, so it has to self-loop. Set the contents of the first loop to y.

Note that $|xy| \leq k + 1 = p$ because the only state that xy can visit twice is the node that it ends on. □

Example 2.7

We can use the Pumping Lemma to prove that the language $A = \{0^k 1^k | k \geq 0\}$ is not regular.

Suppose for contradiction that A is regular. Then the Pumping Lemma applies, and A has some pumping length p. The string $s = 0^p 1^p$ is obviously in A. But, the condition $|xy| \leq p$ implies that x, y are entirely in the region of 0s. But, y clearly can't be the pumped string if it consists entirely of 0s.

Example 2.8

We can also prove the language B above is not regular. Suppose for contradiction that B is regular. Then, since 0^*1^* is regular, $B \cap 0^*1^*$ is regular. But this is A, which isn't regular, contradiction.

3 September 20, 2016

Today:

- Context-free languages
- Context-Free Grammar
- PDA
- Ambiguity
- CFG $\to$ PDA

Joke. "How are things going with the homework? It's due Thursday, I hope you've looked at it." - Prof. Sipser

Finite automata (and regular languages) are very limited in their computational ability. Today we study a stronger model.

3.1 Context-Free Languages

Today we will look at context-free languages (CFLs) and their associated automata, the push-down automata. These will turn out to be equivalent, in the same way DFAs and regular expressions are equivalent.

Context-free languages consist of **variables**, including at a **start variable** S, and **terminals** described by a set of rules. There are **rules** in the language (collectively called the **context free grammar**, or CFG) that allow substitution of a variable by a string of variables and/or terminals. To form a word in a CFL, we start with the start variable and apply rules until we are left with only terminals. .

Example 3.1

Consider the CFL with rules $S \to 0S1$, $S \to R$, $R \to \epsilon$. This has variables S, R and terminals $0, 1$.

We can form a word as follows:

$$S \to 0(S)1 \to 0(0(S)1)1 \to 0(0(R)1)1 \to 0011$$

It's not hard to see that this language generates all strings consisting of some number of 0s, followed by the same number of 1s.

Remark 3.2. Recall that the language of strings of the form 0^k1^k is not regular, so context-free languages can recognize languages that regular expressions cannot.

For notational compactness, we can combine rules whose left-hand sides are the same. For example, $S \to 0S1$, $S \to R$ can be written $S \to 0S1|R$.

Example 3.3

Consider the CFL with rules $S \to E + T|T$, $T \to T \times F|F$, $F \to (E)|a$. This has variables E, T, F and terminals $+, \times, (,), a$.

We generate a word:

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

Remark 3.4. Note that this grammar has the precedence of $\times$ over $+$ built in!.

We can represent substitutions in a CFL by a **parse tree**.

Definition 3.5. A context-free grammar is a 4-tuple $G = (V, \Sigma, R, S)$, where

- V is a set of variables;
- Σ is a set of terminals;
- R is a set of rules;
- $S \in V$ is the start variable.

For $u, v \in (V \cup \Sigma)^*$, we write $u \to v$ (read: "u yields v in one step") if we can go from u to v in one substitution step, according to the rules R. We write $u \to^* v$ (read: "u derives v") if we can go from u to v in some number (possibly zero) of substitution steps.

If $u = S$ and $u \to^* v$, we say that G **derives** v.

The language defined by G is $L(G) = \{w|G \text{ derives } w\}$.

3.2 Ambiguity

In many languages, there are multiple different parse trees corresponding to a single string, and these parse trees correspond to two different meanings for the string.

Example 3.6

In English: "I touched the boy with the stick."

Of course, we don't want our languages to be ambiguous – when our languages correspond to computer instructions, we want every string to have a precise, unambiguous meaning.

3.3 Push-Down Automata

It is provable that context-free languages are strictly stronger than languages generated by regular expressions.

The **push-down automaton** (PDA) is related to CFLs like DFAs and NFAs are related to regular expressions.

The PDA, like the NFA, reads an input from left to right and (possibly nondeterministically) updates a finite, internal state as it reads, and outputs "accept" or "reject" when it finishes reading the input. However, the PDA also has an internal memory stack – a data structure supporting two operations:

- "push": writing an element on the top of the stack, pushing the existing elements down one level;
- "pop": reading the top element of the stack and removing it from the stack, causing the remaining elements to rise one level.

Example 3.7

We exhibit a PDA recognizing the language $\{0^k1^k | k \in \mathbb{Z}_{\geq 0}\}$:

Our PDA has two states: pushing and popping. Initially the PDA starts in the pushing state, and pushes each 0 it sees onto the stack. After seeing the first 1, the PDA switches into the popping state. If it ever sees a 0 again, it rejects; else, for every 1 it sees it pops a 0, and it accepts if and only if its stack is empty in the end.

Formally:

Definition 3.8. A PDA is a 6-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$.

- Q, Σ, q_0, F are the same as before.
- Γ is the stack alphabet, the collection of symbols we may write on the stack.
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q, \Gamma_\epsilon)$.

The $\mathcal{P}$ is because the PDA is allowed to be nondeterministic; the PDA transitions according to some element of the output of δ.

Here, Σ_ϵ and Γ_ϵ represent Σ and Γ augmented with the empty string ϵ. We use these in the input because we're allowed to do operations without reading the input or the stack, and in the output because it may not write anything on the stack.

So, the PDA transitions by reading from the input tape, the stack, or both; it uses the transition function δ to get a set of transitions (q, γ), of which it nondeterministically picks one. The first coordinate is the state the machine transitions to, and the second coordinate (if not ϵ) gets pushed onto the stack.

Example 3.9

We exhibit a PDA recognizing the language $\{ww^R | w \in \Sigma^*\}$, where w^R is the reverse of w.

If our machine knew where the midpoint of the input was, it could push every element it reads until it reaches the middle. Then, it switches states, so that for every input symbol it reads it pops a symbol from the stack and compares the two, and rejects if they are not equal. When our machine reaches the end of the input, it accepts if the stack is empty and all comparisons were successful.

Our machine doesn't actually know where the midpoint of the input is – but that's OK, because its nondeterminism lets it guess where the midpoint is. As long as the machine is in the first (pushing) state, after reading each element it nondeterministically chooses to continue or switch into the second (popping) state.

Remark 3.10. We can prove that $\{ww^R | w \in \Sigma^*\}$ can't be recognized by a deterministic PDA. Thus, unlike with DFAs and NFAs, deterministic and nondeterministic PDAs are not equivalent.

3.4 CFLs and PDAs

Theorem 3.11

A language A is a context-free language if and only if $A = L(P)$ for some push-down automaton P.

Proof. We will prove one direction, that any CFL is recognized by some push-down automaton.

Given a context-free grammar G, we will construct a PDA P such that P recognizes the language generated by G. We'll make P simulate the grammar.

Morally: P will guess the sequence of substitutions that generate a word from the rules of G.

First, P writes S on the top of the stack. Then, it repeatedly reads the top element of the stack. If it is a variable v, P nondeterministically guesses all possible substitutions for v and writes them on top of the stack; else, P pops the top element (which is a terminal) and compares it to the next element of the input, and dies if they don't match. P will accept if and only if some thread eventually accepts. $\square$

4 September 22, 2016

Today:

- Non-CFLs;
- Turing machines.

4.1 Closure Rules for CFLs

Recall from last time:

Theorem 4.1
A is a CFL if and only if some PDA recognizes it.

We proved one direction of this theorem last lecture. The other direction is hard, and we won't go over it.

Corollary 4.2
Every regular language is a context-free language.

Proof. Every DFA is also a PDA because we can choose to not use the stack. Therefore, any language that can be recognized by a DFA can also be recognized by a PDA. □

Corollary 4.3
The intersection of a CFL and a regular language is a CFL.

Proof. We can augment the memory of the PDA recognizing the CFL with the DFA recognizing the regular langauge and run both machines in parallel.[1] We accept iff both machines accept. □

Warning. The intersection of two context-free languages is not necessarily a context-free language! We will provide a counterexample later.

Proposition 4.4
The union of two context-free languages is context-free.

Proof. Rename variables so that the two langauges have distinct variables; then, merge the two languages' grammars and define a new start variable, which can be substituted with either language's start variable. □

[1]Formally, we create a new machine whose states are the Cartesian product of the DFA's states and the PDA's states, and whose interactions with the memory stack depend only on the PDA-coordinate of the current state.

Remark 4.5. Since CFLs are closed under union but not intersection, they are not closed under taking complements; this is because if CFLs are closed under union and complements, they must be closed under intersection by de Morgan's laws.

4.2 Pumping Lemma for CFLs

There is an analogue of the Pumping Lemma for context-free languages:

Lemma 4.6 (Pumping Lemma for CFLs)

For any context-free language A, there exists a p (its **pumping length**) such that if $s \in A$ and $|s| \geq p$, then we can write $s = uvxyz$ such that

- $uv^ixy^iz \in A$ for all $i \geq 0$;
- $vy \neq \epsilon$;
- $|vxy| \leq p$.

The one-line version of the proof is: a big word must have a big parse tree, which must have a big path, which must contain two identical nodes. We pump by splicing the upper node's subtree onto the lower node.

Proof. Let $s \in A$ be long, and G be a CFG by A.

If s is sufficiently long, its parse tree will have a path (from root to leaf) longer than the number of variables in the grammar. This path must, then, have some variable twice. Call this variable R; we use R_1 to denote the instance of R on this path farthest from the root, and R_2 the instance of R on this path second farthest from the root.

Let x be the part of S belonging to the subtree of R_1; let v, y be the parts of S belonging to the subtree of R_2 which are to the left and right of the subtree of R_1. Finally, let u, z be the parts of S to the left and right of the subtree of R_2.

We can pump v and y by copying the subtree of R_2 and pasting it in place of the subtree of R_1, yielding a string $uvvxyyz$. We can repeat this operation to pump v and y again.

To un-pump v and y, we copy the subtree of R_1 and paste it in place of the subtree of R_2, yielding a string uxz.

Finally, we give a bound for p. Let b be the maximal branching factor of the tree, which is the length of the largest right-hand side of a rule. So, if h is the height of the parse tree, then $|s| \leq b^h$. Therefore, $p = b^{\# \text{ variables}+1}$ guarantees that the parse tree has a height at least # variables + 1, and the pumping lemma holds.

The remaining conditions are easy to check. □

4.3 Applications of Pumping Lemma for CFLs

Example 4.7

We will show that $D = \{a^k b^k c^k | k \geq 0\}$ is not context-free.

Assume for sake of contradiction that D is context-free. Then, D satisfies the Pumping Lemma for CFLs, with some pumping length p. Consider the string $s = a^p b^p c^p$; the Pumping Lemma asserts that it can be written as $uvxyz$ such that v, y are pumpable.

By the Pumping Lemma, the central segment vxy has length $\leq p$. But, this means that vxy can encompass at most two of the three symbols a, b, c; so, no matter how we write $s = uvxyz$, at least one of a, b, c will get completely left out of the pumping. Thus uv^2xy^2z has an unequal number of as, bs, and cs and is not in D, contradicting the Pumping Lemma.

Example 4.8

Let $E = \{ww | w \in \Sigma^*\}$. We will show E is not context-free.

Assume for sake of contradiction that E is context-free, with pumping length p. We can see that $s = 0^p 1^p 0^p 1^p$ cannot be pumped.

Example 4.9

Let $F = \{w | w$ has equally many as, bs, and cs$\}$.

Suppose for contradiction that F is context-free. The intersection of a context-free language and a regular language is context free. So, consider

$$F \cap a^* b^* c^*.$$

This is D, which we showed before is not context-free!

4.4 Aside: Intersections of Context-Free Languages are not Context-Free

The languages $D_1 = \{a^i b^i c^j | i, j \geq 0\}$ and $D_2 = \{a^i b^j c^j | i, j \geq 0\}$ are both context-free. But,

$$D_1 \cap D_2 = \{a^k b^k c^k | k \geq 0\}$$

is D, which we showed above is not context-free.

4.5 Turing Machines

Like a finite automaton, a **Turing machine** has a finite control and memory, and an input tape. In addition to the operations of a DFA, the Turing machine has the following abilities:

- The Turing machine can write on its tape (and overwrite the input);
- The Turing machine's head can move in both directions;
- The Turing machine's tape is infinitely long, and initially contains blanks at all spaces past the end of the input;

- Unlike the DFA, which accepts if and only if it is in an accept state when it reaches the end of the input, the Turing machine accepts if and only if it ever reaches an accept state. When it reaches an accept state it stops immediately.

Note that it doesn't make sense to speak of the end of a Turing machine's input, because the Turing machine can itself write on the tape.

Example 4.10

We show how a Turing machine can recognize $D = \{a^k b^k c^k | k \geq 0\}$.

The Turing machine starts at the beginning and scans through the input, checking that all a's come before b's and all b's come before c's. It returns to the beinning of the input and overwrites the first a, b, and c it sees with a symbol x. It then returns back to the beginning, and overwrites the next a, b, and c with a symbol x. It continues in this manner, and accepts if and only if the as, bs, and cs run out simultaneously.

We now formally define a Turing machine:

Definition 4.11. A Turing Machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ such that

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\},$$

where $\Sigma \subset \Gamma$.

When a Turing machine runs, three outcomes may happen. It may:

- halt and accept;
- halt and reject;
- not halt and run forever (this is called "looping").

We consider both the second and third outcome to be rejection, but the distinction between these outcomes will be important.

Definition 4.12. If M is a Turing machine and $A = \{w | M \text{ recognizes } w\}$, we say that $A = L(M)$ (read: A is the language of M), and A is **Turing-recognizable**.

Sometimes we want to deal with Turing machines that always halt, so we have special terminology for these machines:

Definition 4.13. A Turing machine that halts on all inputs is called a **decider**, and its language is called a **Turing-decidable** language.

In particular, a decider must explicitly reject all words not in its language.

5 September 27, 2016

This time:

- Variants of Turing Machines;
- Church-Turing Thesis.

We will use Turing machines as a model of a general-purpose computer.

5.1 Why Turing Machines?

If we add a second stack to a pushdown automaton, its power increases.

It turns out, if we modify the features of a Turing machine a number of different ways – for example, adding a second input tape, or even making it nondeterministic – the power of a Turing machine remains unchanged. A large number of computational models can be shown to be equivalent to Turing machines.

As an analogy: we have a large number of programming languages today – Python, Java, C++, FORTRAN, even assembly code. These languages look very different, but we can prove that their computing power is the same, in the sense that the set of mathematical problems solvable by these programming languages is the same, by constructing compilers that compile one language into another.

5.1.1 Example: Multitape Turing Machines

Suppose we have a Turing machine with multiple input tapes, with the following usage. The input is written in one tape, while the other tapes start blank; all the tapes are read/write, and the machine has a head on each tape. Each step, the machine reads the locations corresponding to all its heads, updates state, and moves each head left or right.

Theorem 5.1

A language A is Turing-recognizable if and only if some multitape Turing Machine recognizes A.

Proof. A language recognizable by an ordinary TM is clearly recognizable by a multitape TM, so one direction is obvious.

Suppose a language A is recognized by a multitape TM S. We will find an ordinary TM S that recognizes A by **simulating** M.

S will divide its tape into segments ("virtual tapes"), such that each segment represents a tape of M.

S has to remember the head location of each of M's heads. It does this by expanding the alphabet of M along with a dotted version of each symbol.

As an example: if M has tapes aba and $bbbb$ with heads on the first a and third b, S stores

$$\#\dot{a}ba\#bb\dot{b}b\#.$$

To simulate one step of M, S scans the entire tape, decides what M would do[2], and then scans the tape again, taking the same action on each virtual tape that M would take.

We need to be careful: what happens if one of the heads of M moves too far to the right, outside the bounds of the virtual memory allocated for it on S's tape? If this happens, we shift things on S's tape to the right to make additional space.

Finally, if M accepts or rejects, S does the same thing. $\square$

5.1.2 Example: Nondeterministic Turing Machines

Turing machines transition according to a function

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\},$$

which takes as input a state and an input symbol, and which outputs a state, a write symbol, and a direction to move.

Analogously, **nondeterministic Turing machines** transition according to a function

$$\delta : Q \times \Gamma \to \mathcal{P}Q \times \Gamma \times \{L, R\}.$$

For each triple (state, write symbol, direction) in the output, the nondeterministic Turing machine starts a new thread and takes that action. If any thread eventually accepts, the nondeterministic Turing machine accepts.

Theorem 5.2

A language A is Turing-recognizable if and only if some nondeterministic TM recognizes A.

Proof. A language recognizable by an ordinary TM is clearly recognizable by a nondeterministic TM, so one direction is obvious.

Suppose we have a nondeterministic TM N. We will use a deterministic TM M to simulate N.

Like before, we use segments of M's tape as virtual memory (which we expand as necessary) for the tapes of the threads of N, and augment the alphabet of N with dotted symbols to track the location of the head on each virtual tape. Each time M passes through its tape, it performs one step on each virtual tape. If a thread t branches into multiple threads $t_1, \ldots, t_k$, M pushes contents of its tape to the right to allocate space on its tape for these threads, and creates these threads by copying the tape for t.

If one thread of N ever accepts, M halts and accepts. $\square$

5.2 Turing-Enumerators

A Turing-Enumerator is a Turing machine E that

- always starts on a blank tape; and

[2]since M has finitely many tapes, S can remember the symbols at M's heads.

- has the additional ability to print strings under program control.

Definition 5.3. The language of an enumerator E, written $L(E)$, is the set of strings that E outputs.

Definition 5.4. We say a language A is **Turing-enumerable** if $A = L(E)$ for some enumerator E.

Theorem 5.5

A language A is Turing-recognizable if and only if is Turing-enumerable.

Proof. Let E be a Turing-enumerator. We construct a Turing machine M that recognizes $L(E)$.

We simply simulate E. Suppose M is given input w. M runs E; whenever E prints a string x, M accepts if $w = x$, and otherwise continues.

Conversely, let M be a Turing machine. We construct a Turing enumerator E that generates $L(M)$.

Naively, we could try to simulate M. E could feed all possible inputs into M:

$$x = \epsilon, 0, 1, 00, 01, 10, 11, \dots,$$

and output exactly the inputs M accepts. But, this runs into a serious problem: if M doesn't halt on, say, 00, it won't process anything past 00.

We fix this by time-sharing: let $s_1, s_2, s_3, \dots$ be an enumeration of the possible inputs of M. E starts a thread running M on s_1, runs it for a step, then starts a thread running M on s_2, runs all active threads for a step, starts a thread running M on s_3, runs all active threads for a step, and so on. Whenever a thread accepts, E prints the thread's input, and when a thread rejects, it dies. □

5.3 Church-Turing Thesis

The **Church-Turing Thesis** states that any computable algorithm is computable by a Turing machine.

Hilbert's Tenth Problem asked: given a polynomial equation $P(x_1, x_2, \dots, x_n) = s$, is there an algorithm that finds a solution $(x_1, \dots, x_n)$ *in integers* that solves it? It turns out this problem is undecidable, but without the notion that algorithms are equivalent to Turing machines, there was not even a hope of solving this problem.

Joke. "Hilbert gave a list of 23 problems to challenge mathemmaticians for the next century. Kind of like homework... except you had a hundred years." - Prof. Sipser

6 September 9, 2016

I missed class today. Today's lecture content can be found in Section 4.1 of the text.

7 October 4, 2016

From last time, recall that the problem

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

is Turing recognizable. Last time we went over some decision problems and constructed Turing deciders that decided them.

7.1 Undecidability

Today, we talk about undecidability – problems that Turing machines cannot decide.

Decidability is stronger than recognizability, because for a language L to be decidable the Turing machine that recognizes L must also reject explicitly anything not in L.

Our main theorem will be:

Theorem 7.1
A_{TM} is undecidable.

To prove this theorem, we will use a **diagonalization** argument. We first have to build this machinery:

7.2 Diagonalization

We motivate this discussion with Cantor's question of whether there are different sizes of infinity – a question that came long before Turing machines.

What does it mean for two sets have the same size? If the sets are finite, we can just count them. But what do we do for infinite sets?

We say two infinite sets are the same size if we can pair off elements from the two sets such that all elements from both sets get paired. Formally:

Definition 7.2. A function $f : A \to B$ is:

- **injective** (one to one) if $f(x) \neq f(y)$ whenever $x \neq y$;
- **surjective** (onto) if for all $y \in B$ there exists some $x \in A$ such that $f(x) = y$;
- **bijective** if injective and bijective.

Definition 7.3. Two sets A, B have the same size if there exists a bijective function $f : A \to B$.

Definition 7.4. A set A is **countable** if it has the same size (according to the previous definition) as the set $\mathbb{N} = \{1, 2, 3, \dots\}$.

This definition leads to somewhat surprising consequences:

Example 7.5

Let $E = \{2, 4, 6, \ldots,\}$ be the set of even numbers. Even though E is clearly a proper subset of $\mathbb{N}$, there is a bijective function from $\mathbb{N}$ to E given by $f(x) = 2x$.

Example 7.6

Let Σ^* be the set of binary strings:

$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}.$$

Σ^* also has the same size as $\mathbb{N}$: we just order elements of Σ^* by length, and among all the strings of the same length by lexicographic ordering. We put this list and a list of $\mathbb{N}$ side by side and pair things off.

Example 7.7

Let $\mathbb{Q}^+ = \{\frac{m}{n} | m, n \in \mathbb{N}\}$ be the set of positive rational numbers. We can put the natural numbers in a table:

$$\begin{array}{ccccc} \frac{1}{1} & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots \\ \frac{2}{1} & \frac{2}{2} & \frac{2}{3} & \frac{2}{4} & \cdots \\ \frac{3}{1} & \frac{3}{2} & \frac{3}{3} & \frac{3}{4} & \cdots \\ \frac{4}{1} & \frac{4}{2} & \frac{4}{3} & \frac{4}{4} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

We can list these numbers by going along the downward-left diagonals:

$$\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{2}{2}, \frac{3}{1}, \frac{1}{4}, \frac{2}{3}, \frac{3}{2}, \frac{4}{1}, \ldots.$$

We then put this list and the natural numbers side by side. This is a bijective map!

It seems like a lot of sets have the same size as $\mathbb{N}$, and we might think that all infinite sets can be somehow bijectively mapped to $\mathbb{N}$. But:

Theorem 7.8

$\mathbb{R}$, the set of real numbers, is not countable.

Proof. Suppose for contradiction that R is countable. Let's say, for example, the map $f : \mathbb{N} \to \mathbb{R}$ is:

$\mathbb{N}$	$\mathbb{R}$
1	2.7181281828...
2	3.1415926...
3	0.3333333...
4	0.143295321...

We'll construct some element $x \in \mathbb{R}$ that isn't mapped to an element of $\mathbb{N}$.

We want x to disagree with $f(1)$ in the first digit after the decimal point. So, we pick x's first digit after the decimal point to be $6 \neq 7$.

We want x to disagree with $f(2)$ in the second digit after the decimal point. So, we pick x's second digit after the decimal point to be $5 \neq 4$.

Similarly, we pick x's third digit after the decimal point to be $2 \neq 3$, and x's fourth digit after the decimal point to be $7 \neq 2$.

So, for all $n \in \mathbb{N}$, this x we constructed disagrees with $f(n)$ at the nth position after the decimal. It therefore does not equal any real number in our list.

This contradicts that f maps some natural number to all real numbers. □

7.3 Undecidability of A_{TM}

Let $\mathcal{L}$ be the set of all languages over Σ.

Theorem 7.9

$\mathcal{L}$ is uncountable.

Proof. Same proof. $\mathcal{L}$ is the power set of Σ^*, the set of all binary strings. By diagonalization, the power set of a countable set is not countable. □

Theorem 7.10

The set of all Turing Machines is countable.

Proof. Turing Machines can be encoded in a finite string, and the set of finite strings is countable. □

Since the set of all languages is larger than the set of all TMs, there must exist *some* languages that is not Turing recognizable.

Now, we show something stronger:

Theorem 7.11

A_{TM} is undecidable.

Proof. Assume for contradiction that some Turing Machine H decides A_{TM}. This means that running H on input $\langle M, w\rangle$ accepts if M accepts w, and rejects *by halting* if M doesn't accept w (which M may do by explicitly rejecting or looping).

We use H to construct a Turing Machine D (for "diagonalization"):

D, on input a string representation of a Turing Machine $\langle M\rangle$, runs H on $\langle M, \langle M\rangle\rangle$, and accepts M iff H rejects. That is, D accepts $\langle M\rangle$ iff M rejects $\langle M\rangle$.

Now, what happens if we run D on $\langle D\rangle$? The above reasoning shows: D accepts $\langle D\rangle$ iff D rejects $\langle D\rangle$. This is a contradiction. □

Morally, here's what's happening: we enumerate Turing machines $M_1, M_2, \ldots$ and make a table for whether M_i accepts the string representation of M_j, say:

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\ldots$
M_1	*acc*	*acc*	*acc*	$\ldots$
M_2	*rej*	*rej*	*rej*	$\ldots$
M_3	*acc*	*rec*	*acc*	$\ldots$
M_4	*rec*	*acc*	*rej*	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

.

H is supposed to be able to compute the entries of this table. Now, we ask: where does D fall on this table? On input $\langle M_j\rangle$, the response of D is the opposite of the diagonal entry $M_j, \langle M_j\rangle$. Then, the diagonal entry $D, \langle D\rangle$ disagrees with itself, so D can't be in this table.

7.4 Consequences of the undecidability of A_{TM}.

Theorem 7.12

If A and $\overline{A}$ are both Turing-recognizable, then A is decidable.

Proof. Let Turing Machines R, S recognize A and $\overline{A}$. We construct a Turing Machine T that decides A as follows:

Given input w, we run R, S in parallel on w. Since R and S recognize A and $\overline{A}$, the one that will accept w will terminate. If R accepts, T accepts; if S accepts, T rejects. □

Corollary 7.13

$\overline{A_{TM}}$ is not Turing-recognizable, since if it were, A_{TM} would be Turing-decidable.

7.5 Introduction to Reductions

We will use the undecidability of one language L_1 to establish the undecidability of another language L_2. We do this by showing that if L_2 is decidable, then L_1 is decidable. This general technique is called **reduction**.

Let

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

We will show HALT_{TM} is undecidable by showing that if it is decidable, then A_{TM} is decidable.

Theorem 7.14

HALT_{TM} is undecidable.

Proof. Assume for contradiction that we have a TM R that decides HALT_{TM}. We construct a TM S that decides A_{TM}.

On input $\langle M, w\rangle$, S runs R on $\langle M, w\rangle$. If R rejects (i.e. running M on w doesn't halt), S rejects. If R accepts (i.e. running M on w halts), S runs M on w (because this is now guaranteed to halt), and accepts iff M accepts. $\square$

8 October 6, 2016

8.1 Mapping Reducibility

Recall that A_{TM} is not decidable. We proved this last time with a diagonalization argument.

We also saw that HALT_{TM} is not decidable. We proved this by showing that if HALT_{TM} is decidable, then given black-box access to a decider for HALT_{TM} we can decide A_{TM}.

We say that we have **reduced** A_{TM} to HALT_{TM}.

Generally, we say that we **reduce** problem A to problem B if, given access a black-box solver for B, we can solve A with a machine of the same power. We'll give a more precise definition later.

Reductions are useful because they show that if we can do B, then we can do A. Often we care more about the contrapositive: if we cannot do A, we cannot do B. Reductions are a powerful way to prove that a problem B can't be solved.

Example 8.1

Let

$$E_{TM} = \{\langle M\rangle : M \text{ is a TM, and } L(M) \text{ is empty}\}.$$

We will prove that E_{TM} is undecidable.

Suppose E_{TM} is decidable – that is, there is a decider R that, on input a description $\langle M\rangle$ of a Turing Machine M, decides whether M accepts no inputs. We will use R to construct a Turing Machine S that decides A_{TM}.

S works as follows: on input $\langle M, w\rangle$, where M is a Turing Machine and w is a string, S constructs the Turing Machine M_w, which on input x, erases the tape, writes w on the tape, and simulates M on w.

Note that M_w behaves the same way on all inputs. We use R to test if $L(M_w)$ is empty. If it is, we reject; else, we accept.

Since the existence of R lets us solve A_{TM}, R cannot exist.

We now define mapping reducibility formally.

Definition 8.2. Say a Turing Machine M computes $f : \Sigma^* \to \Sigma^*$. If on any input w, M halts with $f(w)$ on the tape, we say f is **computable**.

Definition 8.3. A language $A \in \Sigma^*$ is **mapping reducible** to $B \in \Sigma^*$ (denoted $A \leq_m B$), if there exists a computable function f such that $w \in A$ iff $f(w) \in B$. In this case, we say f is the **reducing function** from A to B.

Theorem 8.4

If $A \leq_m B$ and B is decidable (resp. recognizable), then A is decidable (resp. recognizable).

Proof. Let R be the Turing machine that decides (resp. recognizes) B, and let f be the computable map $A \to B$. We construct a decider (resp. recognizer)

S for A: on input w, S computes $f(w)$ and runs R on $f(w)$. S accepts iff R accepts. $\square$

Corollary 8.5

If $A \leq_m B$ and A is not decidable (resp. recognizable), then B is not decidable (resp. recognizable).

8.2 Non Turing-recognizable languages

Recall from last time that if $A, \overline{A}$ are both Turing-recognizable, then A is decidable, and that this implies $\overline{A_{TM}}$ is not Turing-recognizable.

Theorem 8.6

If $A \leq_m B$, then $\overline{A} \leq_m \overline{B}$.

Proof. If there exists a computable function f such that $w \in A$ iff $f(w) \in B$, then it is also true that $w \in \overline{A}$ iff $f(w) \in \overline{B}$. $\square$

Example 8.7

We will show that $A_{TM} \leq_m \overline{E_{TM}}$ by exhibiting a reducing function f.

On input $\langle M, w\rangle$, f constructs the Turing machine M_w, which, on any input, erases the tape, writes w on the tape, and simulates M on it. f returns $\langle M_w\rangle$.

Since $M_w \in \overline{E_{TM}}$ iff it accepts some input, which occurs iff M accepts on w (i.e. $\langle M, w\rangle \in A_{TM}$), this is a reduction from A_{TM} to $\overline{E_{TM}}$.

As a corollary, since $A_{TM} \leq_m \overline{E_{TM}}$, we have $\overline{A_{TM}} \leq_m E_{TM}$. Since $\overline{A_{TM}}$ is not recognizable, neither is E_{TM}.

Let

$$EQ_{TM} = \{\langle M, N\rangle : M, N \text{ are TMs}, L(M) = L(N)\}.$$

Theorem 8.8

Neither EQ_{TM} nor $\overline{EQ_{TM}}$ is Turing-recognizable.

Proof. We will show (a) $\overline{A_{TM}} \leq_m EQ_{TM}$ and (b) $\overline{A_{TM}} \leq_m \overline{EQ_{TM}}$.

(a) we can equivalently show $A_{TM} \leq_m \overline{EQ_{TM}}$. Our reducing function f works as follows:

On input $\langle M, w\rangle$, f creates two Turing machines M_1 and M_2. M_1, on any input, rejects; M_2, on any input, simulates M on w and accepts iff M accepts. f returns $\langle M_1, M_2\rangle$.

Observe that $\langle M_1, M_2\rangle \in \overline{EQ_{TM}}$ iff M_2 accepts on some input, which occurs iff M accepts on w – that is, $\langle M, w\rangle \in A_{TM}$. This is a reduction from A_{TM} to $\overline{EQ_{TM}}$.

(b) we can equivalently show $A_{TM} \leq_m EQ_{TM}$. Our reducing function g works as follows:

On input $\langle M, w \rangle$, g creates two Turing machines M_1 and M_2. M_1, on any input, accepts; M_2, on any input, simulates M on w and accepts iff M accepts. f returns $\langle M_1, M_2 \rangle$.

Observe that $\langle M_1, M_2 \rangle \in EQ_{TM}$ iff M_2 accepts all inputs, which occurs iff M accepts on w – that is, $\langle M, w \rangle \in A_{TM}$. . This is a reduction from A_{TM} to EQ_{TM}.

□

8.3 Post-Correspondence Problem

Suppose we have a finite number of types of cards, each with a top row containing some string and a bottom row containing some string. The **Post-correspondence problem** asks: is there a way to choose and order some of these cards (possibly more than one of each type) such that the string we get from reading the cards' top rows is the same as the string we get from reading the cards' bottom rows?

Formally, language A_{pcp}, the language of solutions to the Post-correspondence problem is

$$A_{pcp} = \left\{ \begin{bmatrix} a_1 \\ b_1 \end{bmatrix}, \begin{bmatrix} a_2 \\ b_2 \end{bmatrix}, \dots, \begin{bmatrix} a_k \\ b_k \end{bmatrix} | \exists i_1, \dots, i_n \text{such that} a_{i_1} \dots a_{i_n} = b_{i_1} \dots b_{i_n} \right\}.$$

We will show next time that this language is not Turing-recognizable.

9 October 13, 2016

Today:

- Post Correspondence Problem
- Computation History Method

9.1 Administrivia

There's a midterm on Thursday, October 27, during class hours. The midterm will be in Walker. The test is open-book and open-notes; laptops are allowed, provided wi-fi is turned off.

9.2 Review

Last time we showed a lot of problems B were undecidable by reducing A_{TM} to B. We argued that given access to a B-solver we can solve A_{TM}; but, because we know A_{TM} is unsolvable, it's impossible for a B-solver to exist.

We defined mapping reducibility; morally, A is mapping reducible to B if we have a function f that converts any question about A to a question about B. Mapping reduction is a type of reduction.

9.3 Post Correspondence Problem

Up until now, the problems we've shown are undecidable are about machines. It's not that surprising that problems about Turing machines aren't decidable, because Turing machines aren't powerful enough to decide questions about themselves.

But, there are problems completely unrelated to Turing machines that aren't decidable.

One problem we discussed earlier is Hilbert's Tenth Problem of testing whether there are integer solutions to polynomial equations. This was proved undecidable by a (very complex) reduction from A_{TM}.

Joke. "When I was a graduate student at Berkeley, there's an entire course dedicated to doing that one reduction. That would be fun, but we don't have time for it in this course." - Prof. Sipser

Today we discuss the **Post Correspondence Problem** (PCP): given a set of 2-row cards

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

for strings $u_1, \dots, u_k, v_1, \dots, v_k$, does there exist a sequence of cards (possibly including more than one of a type) such that when placed in order, the concatenated top string equals the concatenated bottom string?

Example 9.1

Given the set

$$P = \{c_1 = \begin{bmatrix} \text{aa} \\ \text{aba} \end{bmatrix}, c_2 = \begin{bmatrix} \text{ab} \\ \text{aba} \end{bmatrix}, c_3 = \begin{bmatrix} \text{ba} \\ \text{a} \end{bmatrix}, c_4 = \begin{bmatrix} \text{abab} \\ \text{b} \end{bmatrix}.$$

Then, the sequence c_2, c_1, c_3, c_1, c_4 works, so the Post-Correspondence Problem's answer on this P is yes.

To reduce this from A_{TM}, we first need some machinery.

9.3.1 Linearly Bounded Automata

Definition 9.2. A **Linearly Bounded Automaton** is a Turing Machine, except with a finite input tape. The machine's head is not allowed to go off the left or right ends of the input.

Analogously to A_{TM} and E_{TM}, we can define A_{LBA} and E_{LBA}:

Definition 9.3.

$$A_{LBA} = \{\langle M, w\rangle | \text{LBA } M \text{ accepts } w.\} E_{LBA} = \{\langle M\rangle | \text{LBA } M \text{ accepts nothing.}\}$$

It turns out that unlike A_{TM}, A_{LBA} is decidable! But E_{LBA} is still undecidable, and that proof will have a useful idea.

Theorem 9.4

A_{LBA} is decidable.

Proof. The key idea is, the LBA has finitely many internal states and finite memory, so it has finitely many possible configurations. If it runs for too long, it must have been in some state twice, which means it is looping.

Formally: we say a *configuration* of an LBA is a triple (q, p, t), where q is a state of the LBA's control, p is the head location, and t is the contents of the tape.

Say the input w has length $|w| = n$. There are $|Q|$ states, n head positions, and $|\Gamma|^n$ tape contents, so it has

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

configurations.

Thus the following algorithm decides A_{LBA}: on input $\langle M, w\rangle$, run M on w for $|Q| \cdot n \cdot |\Gamma|^n$ steps. If M accepts, accept; else, reject. □

Joke. "Maybe people don't program anymore. They just do big data." - Prof. Sipser

Before proving E_{LBA} is undecidable, we define:

Definition 9.5. An **accepting computation history** is a sequence of configurations $c_0, \ldots, c_{acc}$ of a TM M on w. The accepting computation history is only defined when M accepts on w.

We think of this as a series of snapshots of M as it computes on w.

We format a configuration (q, p, t) by writing the state q immediately to the left of the head position. For example, we format the configuration

$$\begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline\end{array}$$

with state q_6 and the head at 2 as

$$\begin{array}{|c|c|c|c|c|}\hline 1 & q_6 & 2 & 3 & 4 \\ \hline\end{array}.$$

We format an accepting computation history with configurations $c_0, c_1, \dots, c_{acc}$ as

$$\#c_0\#c_1\# \dots \#c_{acc}\#.$$

Theorem 9.6

E_{LBA} is undecidable.

Proof. Assume E_{LBA} is decidable by a Turing machine R. We will build a TM S deciding A_{TM}.

S is going to build an LBA $B_{M,w}$ (with M, w built in) that accepts its input if it is a valid accepting computation history for M on w.

Thus, if M doesn't accept on w, then $B_{M,w}$ accepts nothing. Else, it accepts exactly one input, namely the accepting computation history of M on w.

Building $B_{M,w}$ isn't hard. For each transition $c_i \to c_{i+1}$, $B_{M,w}$ crosses off corresponding locations in c_i and c_{i+1} to check if c_{i+1} properly updates from c_i.

Thus, S builds $B_{M,w}$ and passes the description of $B_{M,w}$ to R. If R says that $L(B_{M,w}) = \emptyset$, then M does not accept w, so S rejects. Else there exists an accepting computation history for M on w, so M accepts w, and S accepts. □

The reason this works is that it's a lot easier for $B_{M,w}$ to check a computation than to run it.

Joke. "It's a lot easier to check a computation than to do it. It's like checking a proof! It's a lot easier to check homework than to do it. You don't have to be smart to be a homework grader. Oh, I hope there are none of them here." - Prof. Sipser

9.3.2 Reduction of Post-Correspondence Problem

Theorem 9.7

PCP is undecidable.

Proof. We will show that A_{TM} reduces to PCP.

Assume there exists a TM R that decides PCP. We will build a TM S that decides A_{TM}.

To decide whether M accepts w, S will design $P_{M,w}$, a set of PCP cards emulating a computation of M on w.

We let the first card be

$$\begin{bmatrix} \# \\ \#q_0w_1,\dots,w_n\# \end{bmatrix}.$$

We design the cards such that this card has to go first.

For each $a \in \Gamma \cup \#$, we have the card

$$\begin{bmatrix} a \\ a \end{bmatrix}.$$

If $\delta(q,a) = (r,b,R)$, we add the card

$$\begin{bmatrix} qa \\ br \end{bmatrix}.$$

The idea is that the cards build a computation history, so some sequence of cards leads to a match if and only if there is an accepting computation history. For example, if our input is 101 and the machine in state q_0 reads 1, and transitions to q_7 and writes 2, we have the cards

$$\begin{bmatrix} \# \\ \#q_0101\# \end{bmatrix} \begin{bmatrix} q_01 \\ 2q_7 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} \# \\ \# \end{bmatrix}$$

So, the cards emulate state transitions, where the top row is always one step behind the bottom row.

Say the bottom row reaches an accept state. We want the top row to catch up to the bottom row. To do this: for each $a \in \Gamma$, we include cards

$$\begin{bmatrix} q_{acc}a \\ q_{acc} \end{bmatrix} \text{ and } \begin{bmatrix} aq_{acc} \\ q_{acc} \end{bmatrix},$$

so after the bottom row reaches an accept state the top row can catch up one character at a time. Finally we include

$$\begin{bmatrix} q_{acc}\#\# \\ \# \end{bmatrix}$$

to let this terminate.

There's just one issue left: this set of cards has a trivial match, namely $\begin{bmatrix} a \\ a \end{bmatrix}$. This isn't a serious problem, because we can add some "junk" to make this work – how this is done was not covered in lecture. □

10 October 18, 2016

Reminder that the midterm is next Thursday, October 27, during class at Walker. Covers material through today.

Today we discuss the Recursion Theorem.

10.1 Review: Computation History Method

Consider the problem

$$ALL_{PDA} = \{\langle B \rangle | B \text{ is a PDA}, L(B) = \sigma^*\}.$$

This problem asks whether a given PDA B accepts all strings.

Remember that E_{PDA} was decidable. Curiously, this isn't true for ALL_{PDA}!

Theorem 10.1

ALL_{PDA} is undecidable.

Proof. Assume for contradiction that a TM R decides ALL_{PDA}. We will build a TM S that decides A_{TM}.

On input $\langle M, w \rangle$, we will design a PDA $B_{M,w}$ that accepts all strings which are *not* accepting computation histories on W.

Let's say we input $x = \#C_1\#C_2\# \dots \#C_{acc}$ into B. We want B to accept everything that is not an accepting computation history because it's easier to find a mistake than to check that everything's OK.

So, B nondeterministically picks a C_i and puts it on the stack, and then compares it to C_{i+1}. These two snapshots should be the same except near the head position, where it's updated according to the rules.

There's just one bug: if we put C_i onto the stack and pop it to compare it to C_{i+1}, it's reversed when we pop it. We can fix this by reversing all the even-numbered snapshots C_i.

Now, we use R to check if $B_{M,w} = \sigma^*$; if yes, S rejects because there is no accepting computation history, and else S accepts. □

Remark 10.2. Why can't we create $B_{M,w}$ with the accept and reject states reversed, so $B_{M,w}$ accepts only accepting computation histories? Because in the positive formulation, $B_{M,w}$ has to check *all* transitions. Say $B_{M,w}$ checks that C_2 follows from C_1. Now we can't back up and reread C_2 to check the transition from C_2 to C_3. In the negative formulation, we get around this by just checking that one transition fails.

In fact, the positive formulation of $B_{M,w}$ can't possibly work, because if it works it'll imply that E_{PDA} is undecidable.

Remark 10.3. Why can't we get around this by having the input string repeat each C_i twice: $\#C_1\#C_1\#C_2\#C_2\# \dots$. This now lets us check all the transitions, but there's no guarantee that the two copies of each snapshot are actually the same.

Joke. It looks like we just resurrected the proof of the false theorem! Which isn't possible, because the theorem is false.

10.2 Recursion Theorem

If we have a factory that makes cars, it feels like the factory has to be more complex than the car. And if we wanted to make a second-order factory that makes factories, we'd expect the second-order factory to be more complex than the factory. So, it feels like we can't make machines that can make themselves.

Somewhat counterintuitively, this is actually possible! We'll show how to make a Turing Machine M that prints itself (that is, when we turn on M, it halts with a description of itself on the tape).

Lemma 10.4

There is a computable $q : \Sigma^* \to \Sigma^*$ where for every w, $q(w) = \langle P_w \rangle$, where P_w is a Turing machine that prints w and halts.

Proof. Output $\langle P_w \rangle$ where $P_w =$ "print w". □

Theorem 10.5

There exists a Turing Machine $SELF$ which prints itself.

Proof. $SELF$ proceeds in two stages A and B. We let $A = P_{\langle B \rangle}$. This gets run first, so when B is run, $\langle B \rangle$ is on the tape.

Now, why don't we just define $B = P_{\langle A \rangle}$? This doesn't work because this makes a circular definition. Since A has B in its own code, B can't have A in its own code.

So, we have B figure out what A is. B uses q on the tape contents to get $P_{\text{tape contents}} = P_{\langle B \rangle} = A$. We combine this with the tape contents to get $\langle AB \rangle = SELF$. □

By packing this process into a single function, we can give Turing machines a primitive operation for referencing their own code!

Example 10.6

Let the language be English. We want a command in English equivalent to "print this sentence" – but just saying "print this sentence" is cheating, because English has a "this" feature, which self-references the sentence; a Turing machine doesn't have a native command to print its own code.

Let's say we command: print blah. Someone obeying this command prints: blah.

Let's say we command: print two copies of the following, the second one in quotes: "blah". Someone obeying this command prints: blah "blah".

Let's say we command: print two copies of the following, the second one in quotes: "print two copies of the following, the second one in quotes". Someone obeying this command prints: print two copies of the following, the second one in quotes: "print two copies of the following, the second one in quotes".

Hooray.

Theorem 10.7

For any TM T, there is a TM R such that for every w, $R(w) = T(w, \langle R \rangle)$.

Proof. Analogous to before. There's a Turing machine with stages A, B, T, with $A = P_{\langle BT \rangle}$, $B = P_{\langle A \rangle}$, $T = \langle ABT \rangle$. □

Morally: It's not only possible for a TM to print out a copy of itself, but also to do computations on its string representation. R is a compiler for T.

Joke. This is a really valuable thing to be able to do. Knowing yourself is useful sometimes. - Prof. Sipser.

This gives us a new, one-line proof that A_{TM} is undecidable.

Proof. Assume H decides A_{TM}.

We construct a Turing machine R: on input w, get own description $\langle R \rangle$ and runs H on $\langle R, w \rangle$. R sees what H does, and does the opposite. This is an immediate contradiction. □

10.3 Applications of Recursion Theorem

Computer viruses use the Recursion Theorem all the time, to transmit their code in the hope of infecting other computers.

Definition 10.8. A **minimal TM** is the TM with the shortest among all equivalent TMs, and

$$MIN_{TM} = \{M | M \text{ is a minimal TM}\}.$$

Theorem 10.9

MIN_{TM} is not Turing-recognizable.

We will use the Recursion Theorem to get a very simple proof.

Proof. Assume MIN_{TM} is Turing-recognizable. We construct a Turing Machine R.

R, on input w, gets is own description $\langle R \rangle$. Since MIN_{TM} is Turing-recognizable, it has an enumerator.

Run the enumerator on MIN_{TM} until some TM S with length bigger than $\langle R \rangle$ appears. R then simulates S on w. S can't be minimal if R is shorter than it and outputs the same result. This is a contradiction. □

11 October 20, 2016

Today we transition into complexity theory.

11.1 Complexity Theory: Introduction

Computability theory asks which problems is decidable – as long as a Turing machine can decide the machine in a finite amount of time we're happy.

In complexity theory, we're only concerned with langauges we *already know* are decidable, and we care about the amount of computational resources needed to solve them.

In particular, we care about how long a Turing machine takes to solve the problem, as a function of the input length.

Definition 11.1. We say $f(n) = O(g(n))$ if there exists c such that $f(n) < cg(n)$ for all sufficiently large n.

Definition 11.2. We say $f(n) = o(g(n))$ if for all c, $f(n) < cg(n)$ for all sufficiently large n.

Example 11.3

Let $A = \{a^k b^k | k \geq 0\}$. We claim there is a one-tape TM that decides A on inputs of length n in at most $O(n^2)$ steps, for fixed c.

The algorithm is as follows:

- Scan the input and check if it is of the form a^*b^*. If not, reject. Else return to the beginning.
- Repeat until all a's crossed off or all b's crossed off:
  - Scan and cross off first a and first b. Then return to the beginning.
- If a's and b's are all crossed off, accept. Else reject.

We can do a time analysis: we do $O(n)$ iterations, each of which take $O(n)$ steps, for a total runtime of $O(n^2)$.

Example 11.4

Now we ask: in the previous problem, can we do better? Can we get $o(n^2)$? Consider this algorithm:

- Scan the input and check if it is of the form a^*b^*. If not, reject. Else return to the beginning.
- Repeat until all a's crossed off or all b's crossed off:
  - Scan and check that the parity of the number of non-crossed as equals the parity of the number of non-crossed bs. If not, reject. Else return to the beginning.
  - Cross off every other a and every other b.
- If a's and b's are all crossed off, accept. Else reject.

Morally, each pass through checks a digit in the binary representation of the number of a's and number of b's. This achieves $O(n \log n)$! Yay.

Can we do even better? Can we decide A in $O(n)$ steps? It turns out the answer is no for single-tape Turing machines.

But, it's easy to see that two-tape Turing machine can solve this in $O(n)$ steps!

11.2 Complexity Theory and Models

This presents a problem. Recall that in computability theory, lots computational models are equivalent in the sense that they can decide the same problems – so, no matter what model we use, we get the same decidable problems. We say that computability theory is **model-independent**.

But, we just showed that complexity theory is dependent on the model! A two-tape Turing machine can decide A in $O(n)$, while a one-tape Turing machine can't.

We fix this by showing that in some sense, the complexity difference between computational models doesn't matter *too much*.

We choose a one-tape deterministic TM as our basic model.

Definition 11.5. For a function $t : \mathbb{N} \to \mathbb{N}$, let

$$TIME(T(n)) = \{B | \text{B decidable by a one-tape TM in } O(t(n)) \text{ steps.}\}$$

The intuition is that as the time bound $t(n)$ increases, the number of languages in $TIME(t(n))$ grows.

Now, we'll prove a result that says that the runtime of an algorithm doesn't depend too much on the model.

Theorem 11.6

If B is decidable in time $t(n)$ on a multitape TM, then it is decidable in the $O(t^2(n))$ on a one-tape TM.

Proof. Recall how we simulated a multitape TM M on a single-tape TM S: we take the contents of our M's tapes and place them sequentially on S's tape, delimited by dividers #.

The combined tapes of M have length at most $O(t(n))$, so the content of S's tape is at most $O(t(n))$. A simulation of a step in M takes at most $O(t(n))$ steps on S, because in the worst case it has to push the entire contents of the tape to the right by 1. Since we simulate $O(t(n))$ steps of M, the simulation takes at most $O(t^2(n))$ steps. $\square$

Definition 11.7. Two computational models are **polynomially related** if they can simulate each other by at most a polynomial increase in runtime $t(n) \mapsto t^k(n)$.

It turns out, most reasonable deterministic computational models of Turing machines are polynomially related.

11.3 The Class P

Definition 11.8. The class P is the class of all problems that can be solved in time polynomial in the length of its input. In notation:

$$P = \bigcup_k TIME(n^k).$$

Why is this a nice thing to consider?

- Since reasonable computational models are polynomially related, P doesn't depend on the model! A problem is solvable in polynomial time in one model iff it is solvable in polynomial time in another.
- P is also roughly corresponds to the set of problems that can be practically solvable.

Example 11.9

Let

$$PATH = \{\langle G, s, t\rangle | G \text{ is a directed graph with path from } s \text{ to } t\}.$$

We decide this problem as follows. On input $\langle G, s, t\rangle$:

- Scan through input and mark s.
- Repeat until nothing new is marked:
  - Scan G and mark all y where (x, y) is an edge and x is already marked.
- If t is marked, accept. Else reject.

The repeated step is at most $O(n^2)$, so this algorithm terminates in $O(n^3)$.

In practice, we don't care too much about the exact time bound, as long as it is polynomial.

Example 11.10

A **Hamiltonian path** from s to t in a directed graph G is a path from s to t that hits all nodes once. We define

$$HAMPATH = \{\langle G, s, t\rangle | G \text{ is a directed graph with Hamiltonian path from } s \text{ to } t\}.$$

We could just try all paths, and this approach clearly shows this problem is decidable. But, this takes exponential time.

Is there a smarter way to solve this? It turns out that this is an open problem, which is equivalent to the **P vs NP conjecture**.

12 October 25, 2016

Today we look at NP, the P vs NP problem, and satisfiability.

12.1 The class NP

Recall last time that the path problem is in P, but the Hamiltonian Path problem is not – it's not known whether we can compute a Hamiltonian Path problem in polynomial time.

Still, in some sense the Hamiltonian Path problem is easy. If someone has found a Hamiltonian Path on a graph, he can convince you that a Hamiltonian Path exists by just showing you the path. Even though Hamiltonian Paths are hard to compute, they are easy to check.

There are problems that don't have this property: for example, there isn't a way to easily prove that a graph has no Hamiltonian Path.

We will formally define this "easy checkability" property.

Definition 12.1. Say a nondeterministic Turing Machine M runs in tine $t(n)$ if every branch of M's computations halts within $t(n)$ steps for all inputs of length n.

Definition 12.2. For a function $t : \mathbb{N} \to \mathbb{N}$, let

$NTIME(T(n)) = \{B | \text{B decidable by a nondeterministic single-tape TM in } O(t(n)) \text{ steps.}\}$

Definition 12.3. The class NP is the class of all problems that can be solved by an NTM in time polynomial in the length of its input. In notation:

$$NP = \bigcup_k NTIME(n^k).$$

Problems that are easily checkablem are in NP. The intuition is, if there is a polynomial-length certificate that a problem is solvable (e.g. that a Hamiltonian Path exists), an NTM can find and check it in polynomial time.

Example 12.4

We will show that $HAMPATH \in NP$ by exhibiting an NTM M deciding $HAMPATH$. On input $\langle G, s, t\rangle$, M does the following:

- Let m be the number of nodes in G, with nodes $v_1, \ldots, v_m$.
- Nondeterministically write a permutation π on $\{1, \ldots, m\}$.
- Check if $v_{\pi(1)}, \ldots, v_{\pi(m)}$ is a Hamiltonian path – that is, check that $v_{\pi(1)} = s, v_{\pi(m)} = t$, and the edge $\left(v_{\pi(i)}, v_{\pi(i+1)}\right)$ exists in G.

12.2 P vs NP

To reiterate:

- P is the set of languages where we can test membership easily.

- NP is the set of languages where we can verify membership easily, given a short (polynomial-length) certificate.

Clearly P is a subset of NP, because if we can test membership easily, we can verify membership easily without a certificate.

Whether $P = NP$ is an open problem. If $P = NP$, then:

- Any problem that can be verified easily can also be solved easily;
- Search is never necessary – a problem that can be solved by searching for a certificate can also be done without the search.

Conversely, if $P \neq NP$, there are problems for which search is unavoidable.

We think that $P \neq NP$, but this isn't known.

In fact, the NTM formulation of NP is equivalent to the certificate formulation. If an NTM can decide a problem in polynomial time, a computation history of the NTM is a valid certificate. Conversely, if a certificate exists, an NTM can solve the problem by nondeterministically searching for it.

12.3 More Examples

Example 12.5

Consider the language

$$COMPOSITES = \{x | x \text{ is a binary number}, x = yz, y, z > 1.\}$$

It's trivial to show $COMPOSITES \in NP$. Any nontrivial factor of a composite number x is a certificate that x is composite.

It was recently shown that $COMPOSITES \in P$, but this algorithm is much, much harder.

Example 12.6

Recall that all CFLs can be written in Chomsky Normal Form, and have derivations of length $2|w|-1$. This was how we showed that membership in a CFLs is decidable.

This also shows that membership in a CFLs is in NP. The derivation of a word is the certificate.

In fact, membership in a CFL is in P. We show this with a dynamic programming algorithm. We make the subproblem table:

w_1	w_2	w_3	$\dots$	w_{n-1}	w_n	
			$\dots$			w_1
$-$			$\dots$			w_2
$-$	$-$		$\dots$			w_3
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$-$	$-$	$-$	$\dots$	$-$		w_n

The cell (w_i, w_j) contains all possible derivations of $w_i w_{i+1} \dots w_j$. We compute this table from the bottom-left up: cells (w_i, w_i)correspond to terminals. We compute a cell (w_i, w_j), for $i < j$, as follows: for all $k \in \{i, \dots, j-1\}$, we check if we can combine the derivations for (w_i, w_k) and (w_{k+1}, w_j) by a valid rule, and write down all possible combinations in (w_i, w_j).

12.4 Satisfiability Problem

A **Boolean formula** consists of Boolean variables $v_1, \dots, v_n$ and operators $\vee$, $\wedge$, $\neg$.

A Boolean formula is **satisfiable** if we can assign values to the variables such that the formula evaluates to True.

The **Boolean satisfiability problem** is:

$$SAT = \{\langle \phi \rangle | \phi \text{ is a satisfiable Boolean formula.}\}$$

It's obvious that $SAT \in NP$ because an assignment of the variables that makes ϕ evaluate to True is a certificate.

It's unknown whether $SAT \in P$. In fact:

Theorem 12.7 (Cook-Levin)
If $SAT \in P$, then $P = NP$.

Proof. Next time! □

Joke. "It's [SAT] kind of the granddaddy of all problems in NP!" - Prof. Sipser

13 November 1, 2016

Today we will talk about polynomial reductions and NP-completeness.

13.1 Polynomial-Time Reductions

We introduce the analogue of mapping reductions in complexity theory:

Definition 13.1. A language A is **polynomial-time reducible** to a language B (written $A \leq_p B$) if $A \leq_m B$ by a reduction function f computable by a polynomial-time TM.

Like before, we think polynomial-time reductions as question transformers, which transform a question about A to a question about B.

Theorem 13.2
If $A \leq_p B$ and $B \in P$, then $A \in P$.

Proof. If Turing machine R decides B in polynomial time, Turing machine S can decide A as follows: on input w, compute $f(w)$ (in polynomial time) and use R to test whether $f(w) \in B$. Give the same answer as R. □

13.2 Reduction Example: $3SAT$ and $CLIQUE$

We define some components of a Boolean formula:

- A **variable** is an unknown with value True or False (e.g. x);
- A **literal** is a variable or the negation of a variable (e.g. $\overline{x}$);
- A **clause** is a disjunction of literals (e.g. $x_1 \vee x_2 \vee \overline{x_3}$);
- A Boolean formula in **conjunctive normal form (CNF)** is a conjunction of clauses (e.g. $(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3)$.

If every clause in a conjunctive normal form has k literals, we say it is in in k-conjunctive normal form. Then:

Definition 13.3.

$$3SAT = \{\phi | \phi \text{ is a satisfiable 3-CNF formula}\}.$$

$3SAT$ is in NP because a valid assignment of variables is a certificate. It's unknown whether $3SAT$ is in P.

Definition 13.4. In a graph G, a k-clique is a set of k nodes such that all pairs of these nodes are directly connected by an edge.

Definition 13.5.

$$CLIQUE = \{\langle G, k\rangle | G \text{ is an undirected graph with a } k\text{-clique}\}.$$

$CLIQUE$ is also in NP because a valid set of k nodes is a certificate. It's unknown whether $CLIQUE$ is in P.

Surprisingly, although we don't know whether $3SAT$ or $CLIQUE$ is in P, we do know:

Theorem 13.6

If $CLIQUE \in P$, then $3SAT \in P$.

Proof. We will show that $3SAT \leq_p CLIQUE$ by exhibiting a reduction f.

We have a 3-CNF boolean formula, say:

$$\phi = (a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee \overline{b} \vee d) \wedge \cdots \wedge (\overline{a} \vee b \vee z).$$

For ϕ to evaluate to true, all the clauses have to evaluate to true; so, within each clause, at least one literal must evaluate to true.

We want f to convert this to a $CLIQUE$ problem $\langle G, k \rangle$. We do this as follows. To make G:

- We make a node for every instance of a literal. If a literal appears multiple times we make make multiple nodes for that literal. So, for example, in the above example, we make nodes for $a, \overline{b}, \overline{c}, \overline{a}, \overline{b}, d, \ldots, \overline{a}, b, z$.
- We connect all pairs of nodes, except:
  - Two nodes whose corresponding literals are part of the same clause, and
  - Two nodes whose corresponding literals are complementary (i.e. x and $\overline{x}$).

We set k equal to the number of clauses.

It remains to show that this reduction is correct – that is, ϕ is in $3SAT$ iff $f(\phi) = \langle G, k \rangle$ is in $CLIQUE$.

Suppose ϕ is satisfiable, so there is an assignment that satisfies ϕ. This assignment satisfies every clause, so in this assignment every clause has at least one true literal.

We pick one true literal in every clause. We claim that the corresponding nodes in G are a k-clique. All the edges between these nodes are present because the corresponding literals are in different clauses and are consistent. We picked k nodes, so we have a k-clique.

Conversely, suppose G has a k-clique. We take nodes in the k-clique and set the corresponding literals to true. We never run into the issue of having to set x and $\overline{x}$ to true, because the nodes for x and $\overline{x}$ aren't connected and can never be in the same clique. This sets at least one literal in each clause to true, so we have a valid assignment. □

13.3 NP-Completeness

Next time, we will show:

Theorem 13.7

Every language in NP is poly-time reducible to $3SAT$.

We already showed that $3SAT \leq_p CLIQUE$, so in fact, this shows:

Corollary 13.8
Every language in NP is poly-time reducible to $CLIQUE$.

This says, in some sense, that $3SAT$ and $CLIQUE$ are the hardest problem in NP. If any problem in NP is not in P, then $3SAT$ is not in P.

This notion of the "hardest problem in NP" motivates the following definition:

Definition 13.9. B is **NP-complete** if:

- $B \in NP$;
- For every $A \in NP$, $A \leq_p B$.

13.4 Another Reduction: $3SAT$ and $HAMPATH$

Recall that

$$HAMPATH = \{\langle G, s, t\rangle | G \text{ has a Hamiltonian path from } s \text{ to } t\}.$$

Theorem 13.10
$3SAT \leq_p HAMPATH$.

Proof. Darn. Doing diagrams in real time is hard. This resource (or the text) explains the reduction well: `https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/sat.pdf`.

Essentially:

- We convert a Boolean formula to a chain of "gadgets" such that each gadget represents a variable.
- Each gadget can be traversed in two directions, one corresponding to setting it to true, and one corresponding to setting it to false.
- We wire in clause-nodes into the their constituent variables' gadgets, so that they're only reachable when the variable's gadget is traversed in the direction that sets the clause to true.
- Then, we have a Hamiltonian path iff all the clause-nodes are reachable, which occurs iff we can choose a direction to traverse each gadget (i.e. a boolean setting for each variable) that makes the Boolean formula true.

□

14 November 3, 2016

Today we prove the celebrated Cook-Levin Theorem:

Theorem 14.1
SAT is NP-Complete.

14.1 Implications

Last time, we showed $3SAT \leq_p CLIQUE, HAMPATH$. In recitation, we will show $SAT \leq_p 3SAT$.[3] In the text, we show $HAMPATH \leq_p UHAMPATH$, the Hamiltonian Path problem on undirected graphs.

Together, this shows:

Corollary 14.2
$SAT, 3SAT, CLIQUE, HAMPATH, UHAMPATH$ are all NP-complete.

Thus, if any of these problems are in P, then $P = NP$; conversely, if $P \neq NP$, then none of these problems are in P.

14.2 Proof of Cook-Levin

We need to show two things:

1. $SAT \in NP$;
2. For any $A \in NP$, $A \leq_p SAT$.

(1) is easy because a valid assignment of variables to a boolean formula ϕ is a certificate that $\phi \in SAT$. So, we focus on (2).

Morally, all this proof is doing is simulating a Turing machine with logic gates. With this simulation, we can simulate a TM that verifies a problem $A \in NP$ given a certificate with a boolean formula.

Let $A \in NP$ be any language. Say that A is decided by an NTM M in time n^k. We will exhibit a polynomial-time reduction to SAT. Let our reducing function be f, where $f(w) = \phi_w$, such that $w \in A$ iff ϕ_w is satisfiable.

The idea is, ϕ_w "says" that M accepts w.

We construct ϕ_w as follows. Say that a thread of M accepts w, and let $c_1, c_2, \dots, c_{n^k}$ be the computation history of this thread of M (if M terminates before n^k steps, just let the remaining c_i be the final state). Then, we say an "accepting tableau of M on w" is:

[3] The proof is also in the text, or here: `http://web.mit.edu/~neboat/www/6.046-fa09/rec8.pdf`

$\dots c_1 \dots$
$\dots c_2 \dots$
$\dots c_3 \dots$
$\vdots$
$\dots c_{n^k} \dots$

Since M runs in time n^k, each line has length at most n^k. We set each line to have width n^k and pad the right with blanks as needed.

We want ϕ_w to be satisfiable iff an accepting tableau of M on w exists.

For each cell (i,j) in the tableau, we make $|Q|+|\Gamma|$ boolean variables (think of these as "lights" in the cell) $x_{i,j,\sigma}$, for $\sigma \in \Gamma \cup Q$. If our tableau is an accepting computation history, what has to be true?

1. Each cell must have exactly one light turned on;
2. The top row is the starting configuration

We therefore let

$$\phi_w = \phi_{cell} \wedge \phi_{start} \wedge \phi_{inside} \wedge \phi_{accept},$$

where these variables, described below, correspond to the above conditions.

(1) ϕ_{cell}

ϕ_{cell} says that exactly one variable $x_{i,j,\sigma}$ is true for each i,j. Thus, for particular i,j, we want:

$$\bigvee_{\sigma \in \Gamma \cup Q} x_{i,j,\sigma}$$

to be true, because at least one of the $x_{i,j,\sigma}$ has to be true. Likewise we want

$$\bigwedge_{\substack{\sigma \neq \tau \\ \sigma,\tau \in \Gamma \cup Q}} \left(\overline{x_{i,j,\sigma} \wedge x_{i,j,\tau}}\right),$$

because we can't have two $x_{i,j,\sigma}$ simultaneously be true. Thus for particular i,j we want

$$\bigvee_{\sigma \in \Gamma \cup Q} x_{i,j,\sigma} \wedge \bigwedge_{\substack{\sigma \neq \tau \\ \sigma,\tau \in \Gamma \cup Q}} \left(\overline{x_{i,j,\sigma} \wedge x_{i,j,\tau}}\right).$$

Moreover we want this for all i,j, so we set

$$\phi_{cell} = \bigwedge_{i,j} \left(\bigvee_{\sigma \in \Gamma \cup Q} x_{i,j,\sigma} \wedge \bigwedge_{\substack{\sigma \neq \tau \\ \sigma,\tau \in \Gamma \cup Q}} \left(\overline{x_{i,j,\sigma} \wedge x_{i,j,\tau}}\right) \right)$$

(2) ϕ_{start}

ϕ_{start} says that the top row of the tableau is the start configuartion. The starting configuration is $q_0 w_1 w_2 \dots w_{n^k-1}$, so

$$\phi_{start} = x_{1,1,q_0} \wedge x_{1,2,w_1} \wedge x_{1,3,w_2} \wedge \cdots \wedge x_{1,n^k,w_{n^k-1}}.$$

(3) ϕ_{accept}

ϕ_{accept} says that the bottom row of the tableau contains the accept state. That is:

$$\phi_{accept} = \bigvee_{1 \le j \le n^k} x_{n^k j q_{acc}}.$$

(4) ϕ_{inside}

ϕ_{inside} says that the interior of M follows the rules of M.

We say a 2 row by 3 column "window" of the tableau is "legal" if it obeys the rules of M. For example, if we see the window

a	q_2	b
a	c	q_3

and we know that when the TM is in state q_2 looking at a b, it writes a c and moves right, then this window is legal.

We can see that a tabeleau is valid iff all windows are legal.

Remark 14.3. How do we know we don't have a row with two q_i's? There is only one q_i in the first row, and the 2×3 window checker enforces that each q_i gives rise to only one q_i in the next row.

Remark 14.4. In fact, this is why a 2×2 window is insufficient. Here, it's possible for a q_i to give rise to a q_i to its down-left and to its down-right.

Remark 14.5. Since there is only one q_i marker per row, this also ensures that we don't multiple transitions in a row, because transitions can only occur when the q_i is involved.

Then, we define

$$\phi_{inside} = \bigwedge_{i,j} (2 \times 3 \text{ window at } (i,j) \text{ is legal}),$$

where the Boolean formula could be explicitly written if we actually felt like it.

Conclusion

This is a reduction of A to SAT. This uses a certificate of length $(|Q| + |\Gamma|)\, n^{2k}$, and is thus polynomial. $\square$

15 November 8, 2016

15.1 Godel's Letter

Godel showed that in any mathematical system, there are statements at are true but not provable.

A corollary of this is that the problem of deciding whether a mathematical statement is true is undecidable; the problem of deciding whether a mathematical statement is provable (which is stronger than true) is also undecidable.

But, the problem of deciding whether a mathematical statement is provable with a proof of length $\leq n$ is decidable – a machine can simply exhaustively search all strings of length $\leq n$.

Godel asked Von Neumann: if $\phi(n)$ is the optimal time a machine can check whether a proof of length $\leq n$ exists, how quickly does $\phi(n)$ grow? If it grows sufficiently, slowly, we can replace the work of mathematicians with machines. This was, in some sense, a statement of whether $P = NP$ many years before the problem was formulated.

15.2 Space Complexity

We define these classes of problems analogously to their time-complexity counterparts:

Definition 15.1. For a function $s : \mathbb{N} \to \mathbb{N}$:

$$SPACE(s(n)) = \{A | \text{Some (deterministic) TM decides } A \text{ and runs in space } O(s(n))\}.$$

Definition 15.2. For a function $s : \mathbb{N} \to \mathbb{N}$:

$$NSPACE(s(n)) = \{A | \text{Some NTM decides } A \text{ and runs in space } O(s(n))\}.$$

Analogously to P and NP, we define:

Definition 15.3. The class $PSPACE$ is the class of all problems that can be solved by a TM in space polynomial in the length of its input. In notation:

$$PSPACE = \bigcup_k SPACE(n^k).$$

And:

Definition 15.4. The class $NPSPACE$ is the class of all problems that can be solved by an NTM in space polynomial in the length of its input. In notation:

$$NPSPACE = \bigcup_k NSPACE(n^k).$$

First, note the obvious relation:

Theorem 15.5

For any function $t : \mathbb{N} \to \mathbb{N}$, $TIME(t(n)) \subseteq SPACE(t(n))$.

Proof. It takes one unit of time to write on one unit of space. A machine running for x steps uses at most space x. $\square$

The reverse relation is a lot less clear:

Theorem 15.6

For any function $t : \mathbb{N} \to \mathbb{N}$, $SPACE(s(n)) \subseteq TIME(2^{O(s(n))}) = \bigcup_{c>1} TIME(c^{s(n)})$.

Proof. This is similar to the proof for LBAs. If our tape is bounded by size $s(n)$, a Turing machine restricted to this tape can run for time at most $2^{O(s(n))}$ before cycling. $\square$

In fact:

Theorem 15.7

$NP \subset PSPACE$.

Proof. We first claim $SAT \subset PSPACE$.

To solve SAT in polynomial space, we just exhaustively guess all possible assignments of variables[4] and test if they satisfy the formula.

Then, for any $A \in NP$, we first reduce A to SAT in polynomial time (and thus, polynomial space), and then solve SAT in polynomial space. $\square$

15.3 Quantified Boolean Formulas

We discuss a generalization of SAT.

Definition 15.8. A **Quantified Boolean Formula** (QBF) is a Boolean formula preceded by logical quantifiers[5], which is true if the formula, taken as a logical statement, is true.

Example 15.9

The QBF

$$\forall x \exists y\, [(x \vee y) \wedge (\overline{x} \vee \overline{y})]$$

is true. If $x = 1$, making $y = 0$ makes the Boolean formula true, and if $x = 0$, making $y = 1$ makes the Boolean formula true.

Example 15.10

The QBF

$$\exists y \forall x\, [(x \vee y) \wedge (\overline{x} \vee \overline{y})]$$

is false. (Convince yourself of this!)

This shows that the order of the quantifiers matters.

[4]We overwrite previous guesses to not use exponential space. To track which assignments we have already guessed, we guess in lexicographic order.

[5]$\forall$, meaning "for all," and $\exists$, meaning "there exists."

Definition 15.11. The problem of whether a QBF is true is

$$TQBF = \{\phi | \phi \text{ is a true QBF}\}.$$

Remark 15.12. $TQBF$ is a generalization of SAT, because making all the quantifiers $\exists$ recovers SAT. It isn't known whether $TQBF \in P$.

In fact, it isn't know whether $TQBF \in NP$.

Theorem 15.13

$TQBF \in PSPACE$.

We prove this by providing the following algorithm:

Algorithm 15.14

We simply test recursively. On input ϕ: if $\phi = \exists x_i \phi'$, we substitute $x_i = 1$ and $x_i = 0$ into ψ, and return true iff ϕ' is true on at least one of these substitutions; if $\phi = \forall x_i \phi'$, we substitute $x_i = 1$ and $x_i = 0$ into ψ, and return true iff ϕ' is true on both substitutions.

Since we can test substitutions sequentially and overwrite previous substitutions, this works in polynomial space.

15.4 Word Ladders

Consider the problem of **word ladders**, where we try to get from one word to another by changing only one letter at a time. For example:

$$\text{LEAD} \to \text{MEAD} \to \text{MELD} \to \text{MOLD} \to \text{GOLD}.$$

More formally:

Definition 15.15. The problem $LADDER_{DFA}$ is defined as:

$$LADDER_{DFA} = \{\langle B, s, t\rangle | B \text{ is a DFA, } \exists \text{ ladder } u_1, u_2, \dots, u_k \in L(B) \text{ from } s \text{ to } t\}.$$

Theorem 15.16

$LADDER_{DFA} \in NPSPACE$.

We prove this with the following nondeterministic algorithm:

Algorithm 15.17

Starting at s, nondeterministically change one symbol at a time and check if the new word is in $L(B)$. If it isn't, stop, and otherwise continue. If we ever reach t, we accept.

Each thread of this algorithm clearly uses polynomial space. To make this a decider, we just have to make sure no thread of this algorithm runs forever.

If an accepting thread exists, an accepting thread that doesn't visit the same word twice exists. So, since the number of words in $L(B)$ of length $|s|$ is at most $|\Sigma|^{|s|}$, we can terminate a thread when it runs for longer than this many steps, since such a thread must have visited a word twice.

In fact, $LADDER_{DFA} \in PSPACE$! This is a consequence of Savitch's Theorem:

Theorem 15.18 (Savitch's Theorem)

$PSPACE = NPSPACE$.

In a sense, this says that $P = NP$ has already been solved, in the space-universe.

Attempted Proof of Savitch's Theorem. Suppose we have an NTM N which decides a problem in $NPSPACE$. We want to simulate the computation of N on an input w with a deterministic TM.

A naive approach is to DFS the tree of threads of N's computation on w; whenever N makes a nondeterministic decision, we visit the threads of this computation in some canonical order.

This doesn't work! If a thread of N's computation on w runs in n^k space, its computation can have $2^{O(n^k)}$ before it stops. To DFS this computation tree we have to write down this sequence of states and backtrack when we hit a dead end. But, we can't write down an exponential number of states. Darn.

So, how do we prove this theorem? Next time... □

16 November 10, 2016

Today, we prove three theorems:

- $LADDER_{DFA} \in PSPACE$;
- Savitch's Theorem;
- $TQBF$ is $PSPACE-complete$.

In fact, we will see that these proofs are, in some sense, the same proof.

16.1 The Ladder Problem

Last time, we showed that $LADDER_{DFA} \in NPSPACE$, by starting at s, overwriting one letter at a time nondeterministically, accepting if we reach t, and rejecting if a branch runs too long.

Now we want to do this deterministically. We saw last time that a depth first search approach doesn't work because to keep track of how to backtrack, we have to write down an exponential number of states.

Say the length of s is n, and let our alphabet have size d. What's the maximum number of steps in a word ladder from s to t? This is at most d^n, the number of words of length n.

If we can get from s to t in d^n steps, there exists a string mid such that we can get from s to mid, and mid to t, each in $\frac{1}{2}d^n$ steps.

Then, there exists a string mid_1 such that we can get from s to mid_1, and from mid_1 to mid, in $\frac{1}{4}d^n$ steps. Likewise, there is a string mid_2 such that we can get from mid to mid_2, and mid_2 to t, in $\frac{1}{4}d^n$ steps.

This inspires the following algorithm:

Algorithm 16.1

We solve the subproblem of checking whether we can get from S to T in at most C steps. To solve this:

- If $C = 1$, check if $S \to T$ is a valid transition.
- Else, let MID iterate over all d^n strings of length n. For each value of MID, check whether we can get from S to MID and MID to T in $C/2$ steps. If for some MID, both checks succeed, accept.

We run this algorithm starting with $S = s, T = t, C = d^n$.

Proposition 16.2

This algorithm uses polynomial space.

Proof. If our recursion goes K layers deep, we need to remember K values of MID. But, the max recursion depth we need is

$$\log_2 d^n = n \log_2 d = O(n).$$

Each value of MID requires n symbols to remember, and the computations in each recursive layer require $O(n)$ space, so each recursive layer uses $O(n)$ space. Thus, the total space we need is $O(n^2)$. $\square$

This proves:

Theorem 16.3
$LADDER_{DFA} \in PSPACE$.

Yay.

16.2 Proof of Savitch's Theorem

Theorem 16.4 (Savitch's Theorem)
For a function $s : \mathbb{N} \to \mathbb{N}$ such that $s(n) \geq n$,

$$NSPACE(s(n)) \subseteq SPACE((s(n))^2).$$

Given a problem A, decided by an NTM N in space $O(s(n))$, we will construct a deterministic TM M that runs in space $O(s^2(n))$.

N has a tape of length $s(n)$. If N accepts on input w, there is an **accepting tableau** of configurations – a table where each row is a configuration of N, the first row is C_{start}, the starting configuration of N on w, and the last row is an accepting configuration C_{accept}.

For sake of simplicity, we can say C_{accept} is unique. Otherwise we can just loop over all possible values of C_{accept} with a little more space.

How many steps can an accepting computation run for? The tape has length $s(n)$, so there are $d^{s(n)}$ possible configurations. So, if an accepting computation exists, there is an accepting tableau with at most $d^{s(n)}$ rows.

This motivates this algorithm, which is similar to the ladder algorithm:

Algorithm 16.5
We solve the subproblem of checking whether we can get from a configuration C_i to a configuartion C_j in at most T steps. To solve this:

- If $T = 1$, check if $C_i \to C_j$ is a valid transition.
- Else, let C_{mid} iterate over all d^n strings of length n. For each value of C_{mid}, check whether we can get from C_i to C_{mid} and C_{mid} to C_j in $T/2$ steps. If for some C_{mid}, both checks succeed, accept.

We run this algorithm starting with $C_i = C_{start}, C_j = C_{accept}, C = d^{s(n)}$.

How much space does this algorithm use?

If our recursion goes K layers deep, we need to remember K values of C_{mid}. But, the max recursion depth we need is

$$\log_2 d^{s(n)} = s(n) \log_2 d = O(s(n)).$$

Each recursive layer uses $O(s(n))$ space. Thus, the total space we need is $O(s(n)^2)$.

This proves Savitch's Theorem! $\square$

16.3 PSPACE-completeness

We've showed that $P \subseteq NP \subseteq PSPACE$. Analogously to NP-completeness, we can define a condition for $PSPACE$-completeness:

Definition 16.6. A problem B is in $PSPACE$ if:

- $B \in PSPACE$;
- For every $A \in PSPACE$, $A \leq_p B$.

Remark 16.7. The $\leq_p$ in the above definition is a *polynomial-time* reduction. We don't want it to be a polynomial-space reduction – if we did, all problems in $PSPACE$ would be $PSPACE$-complete, which isn't interesting!

16.4 $TQBF$ is $PSPACE$-complete

In a way analogous to the proof of Cook-Levin, we will prove:

Theorem 16.8
$TQBF$ is $PSPACE$-complete.

We already know that $TQBF \in PSPACE$. Thus, it suffices to show that for all $A \in PSPACE$, $A \leq_p TQBF$.

We could try the Cook-Levin approach: we make d boolean variables for each cell in an accepting computation tableau, and write down a Boolean formula that says that a valid computation tableau exists.

Unfortunately, this doesn't work! If $A \in PSPACE(n^k)$, the accepting computation tableau can be d^{n^k} rows long. We can't hope to, in polynomial time, write a formula this big.

Instead, we create variables $\phi_{C_i,C_j,t}$, which say, "we can transition from C_i to C_j in at most t steps."

We can recursively define

$$\phi_{C_i,C_j,t} = \exists C_{mid} \left[\phi_{C_i,C_{MID},t/2} \wedge \phi_{C_{MID},C_j,t/2}\right].$$

When $t = 1$, it's not hard (details omitted) to write down a formula for $\phi_{C_i,C_j,t}$.

What is our recursion depth? Computation tableau depths are bounded by d^{n^k}, so our recursive depth is

$$\log_2 d^{n^k} = n^k \log_2 d = O(n^k).$$

This, by itself, doesn't work yet – we get an exponential number of subproblems! And this shouldn't be surprising, because we haven't used any $\forall$ operators – Remember that TQBFs with only $\exists$ operators are the same as SAT.

We fix this as follows: we write

$$\phi_{C_i,C_j,t} = \exists C_{mid} \forall_{(C_i',C_j')\in\{(C_i,C_{MID}),(C_{MID},C_j)\}} \phi_{C_i',C_j',t/2}.$$

Now, our recursive branching factor is 1, so we have a total of $O(n^k)$ subproblems.

This is a polynomial reduction! □

17 November 15, 2016

I missed class today. Today's lecture content can be found in Sections 8.3-8.5 of the text. Sketches of some results are in the is in the "Last time" section of the following lecture.

18 November 17, 2016

18.1 Last Time

We finished our discussion of polynomial space. We showed that GG is $PSPACE$-complete.

We introduced the language classes

$$L = SPACE(\log n)$$

and

$$NL = NSPACE(\log n),$$

which are algorithms that use only logarithmic space. To talk about sublinear space, we had to introduce a new model of computation: the input tape is now read-only, and we have a separate, shorter read/write work tape.

Example 18.1

The language $\{0^k 1^k | k \geq 0\}$ is in L.

Example 18.2

The language $PATH$ is in NL.

Whether $L = NL$ is an open question.

18.2 Log-space and P

Theorem 18.3

$L \subseteq P$.

Proof. Given a TM M that runs in log space, let $A = L(M)$.

Let a configuration of M on w, where $|w| = n$ be (q, p_1, p_2, t), where q is the machine's state, p_1, p_2 are the head positions on the input tape and work tape, and t is the contents of the work tape. Say the work tape has $|\Gamma| = d$ symbols in its alphabet. How many configurations are there? There are:

$$|Q| \cdot n \cdot O(\log n) \cdot d^{O(\log n)} = |Q| \cdot n \cdot O(\log n) \cdot poly(n) = poly(n).$$

□

In fact, we can show something stronger:

Theorem 18.4

$NL \subseteq P$.

Proof. Let A be a language decided by an NL-machine N. We want to give a polynomial time algorihtm for A.

This proof is a bit trickier. We can't explore the whole tree of the computation of N on w. By the previous argument, this tree has height $poly(n)$, but because of branching, the tree's size is exponential. Darn.

Instead, we will construct a computation graph G for N on w:

- The nodes of G are configurations of N on W;
- There is an edge from c_i to c_j if N can go from c_i to c_j in one step;

We test if there is a path from c_{start} to c_{accept} and accept iff a path exists.

We established before that there are polynomially many configurations, so G can be generated in polynomial time. Testing whether a path exists can be done in polynomial time (by DFSing the graph, for example), so this algorithm works in P. □

Here, we trade NL's nondeterministicness for P's ability to use more than logarithmic space. It's not known whether we can deterministically simulate a computation by an NL-machine in log space – this is equivalent to asking whether $L = NL$.

Remark 18.5. What do we do if there are multiple accept states? The simplest solution is to assume that N, after reaching any accept state, clears the working tape, moves both heads to the left, and then really accepts. Then, there's only one possible accept state.

Remark 18.6. In fact, Savitch's Theorem works for $s(n) \geq \log n$ (exercise: check the details), so $NL \subset SPACE(\log^2 n)$. But, simulating an NL machine in $\log^2 n$ space turns out to use more than polynomial time. It's not known whether we can simulate an NL machine in both polynomial time and $\log^2 n$ space.

18.3 NL-completeness

Hopefully this definition looks familiar:

Definition 18.7. A language B is **NL-complete** if:

- $B \in NL$;
- for all $A \in NL$, $A \leq_L B$.

What does the $\leq_L$ mean? Why not just use $\leq_P$?

We don't want to use polynomial reduction in this definition! Since $NL \subset P$, any problem in NL is polynomial-time reducible to any other problem in NL; the reduction function can just find the answer.[6] So, to define NL-completeness we actually need a new notion of reduction:

[6]This is the same reason $PSPACE$-completeness is defined with polynomial-time reductions, and not polynomial-space reductions.

Definition 18.8. A **log space transducer** is a Turing machine with three tapes: a read-only input tape of length n, a read/write work tape of length $O(\log n)$, and a write-only output tape of unbounded length.[7]

Definition 18.9. A problem A is **log-space reducible** to a problem B, denoted $A \leq_L B$, if there is a reduction function f, computable by a log-space transducer, such that $w \in A$ iff $f(w) \in B$.

Like before, we have the notion that $A \leq_L B$ means A is at most as hard as B. The following theorem formalizes this:

Theorem 18.10

If $A \leq_L B$ and $B \in L$, then $A \in L$.

Proof. Let a TM R decide B in L. We will make TM S that decides A in L.

Let's first try this the naive way. On input w, S runs the log-space transducer f from A to B, and writes $f(w)$. R then decides B on $f(w)$.

...oh wait, we can't actually write $f(w)$ because the work tape has logarithmic length. Welp.

We patch this as follows: we run R *first*, without computing $f(w)$. Every time R goes to read some symbol (say, the ith symbol) in $f(w)$, we start up the transducer from scratch (without having it actually print anything) and wait for it to compute the ith symbol of $f(w)$. It passes this symbol to R.

Using this procedure we can get away with not actually storing $f(w)$ anywhere, so R runs in log space. Yay. $\square$

...zzz honestly this proof feels like a last-minute patch to a buggy software that needed to be shipped right now.

Theorem 18.11

$PATH$ is NL-complete.

Proof. We have to show two things: $PATH \in NL$, and $A \leq_L PATH$ for all $A \in NL$. We proved the first part last time.

So, suppose $A \in NL$ is decided by an NL-machine N. We will reduce A to $PATH$.

As before, assume N has a single accepting configuration. Our reduction function f works as follows: given input w to A, f will compute $\langle G, s, t\rangle$ such that G is the computation graph of N on w, s is the start state of this computation, and t is the accept state of N. Then, $w \in A$ iff G has a path from s to t – that is, if $PATH$ accepts $\langle G, s, t\rangle$.

Now, we just have to show we can compute this reduction in log space! We systematically cycle through all pairs (c_i, c_j) (think: like an odometer!) and check whether $c_i \to c_j$ is a valid transition; if it is, we print out (c_i, c_j) on the output.

This is a log space reduction! $\square$

[7]Though, the output is bounded by the fact that the machine has to halt. In particular, the machine has $poly(n)$ configurations, so the model implicitly limits the output length to $poly(n)$.

18.4 coNL

We define the language $coNL$ as the complement of NL:

$$coNL = \{A | \overline{A} \in NL\}.$$

Theorem 18.12

$NL = coNL$.

Proof. It suffices to prove that $\overline{PATH} \in NL$. This is equivalent to $PATH \in coNL$, and because $PATH$ is NL-complete, the theorem follows.

We will construct an NL machine that accepts on $\langle G, s, t\rangle$ iff t is not reachable from s.

[I didn't follow this construction in lecture. Will fill in ASAP.] □

Remark 18.13. The analogous assertion for NP, that $NP = coNP$, is not known.

19 November 22, 2016

Today, we will discuss the time and space hierarchy theorems.

19.1 Motivation and Outline

Recall that

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE.$$

We don't know whether $P = NP$. In fact, we don't even know if $P = PSPACE$, though we strongly believe this isn't in P.

In fact, we haven't proven that *any* decidable language is outside P.

We'll show that if we give a machine slightly more computation time, it can solve more problems. This will imply that there are problems solvable in exponential time not solvable in polynomial time.

Analogously, we can show that if we can give a machine more computation space, it can solve more problems. Along the way, we can show $L \neq PSPACE$.

This presents an interesting situation: we don't know whether $P = L$, or whether $P = PSPACE$. But, the fact that $L \neq PSPACE$ means that, at least, *both* questions can't be answered in the affirmative.

In fact, we can show $NL \neq PSPACE$ – this is the strongest result of this form we can prove. This holds because $NL \subseteq SPACE(\log^2 n)$ by Savitch's Theorem, and $SPACE(\log^2 n) \subsetneq PSPACE$ by the Space Hierarchy Theorem, which we will prove today.

19.2 Hierarchy Theorems

19.2.1 Space Hierarchy Theorem

The Hierarchy Theorems allow us to say things like:

$$TIME(n^2) \subsetneq TIME(n^3)$$

and

$$SPACE(n^2) \subsetneq SPACE(n^3).$$

Formally:

Theorem 19.1 (Space Hierarchy Theorem)
For $f : \mathbb{N} \to \mathbb{N}$ (satisfying some technical condition, described later), if $g(n) = o(f(n))$, then

$$SPACE(g(n)) \subsetneq SPACE(f(n)).$$

In other words, there is a language A such that:

- A is decidable in $O(f(n))$ space, and
- A is not decidable in $o(f(n))$ space.

Proof of Space Hierarchy Theorem. We will just exhibit a Turing machine D deciding A. This Turing machine has to ensure two things:

- D runs in $O(f(n))$ space;
- A cannot be decided in less space – that is, D behaves differently than all machines that use $o(f(n))$ space.

D will operate as follows:

On input w, D first computes $f(n)$ and goes to the right $f(n)$ spaces on its tape. It writes a # sign $f(n)$ spaces to the right on the tape. D resets its head to the left of the tape. If it ever hits the # sign on its computation, it automatically rejects; this ensures that D never uses more than $f(n)$ space.

If $w \neq \langle M \rangle$ for any Turing machine M, D rejects. Otherwise, D simulates M on W:

- If M accepts, D rejects;
- If M halts and rejects, then D accepts.

Thus, D accepts exactly the descriptions $w = \langle M \rangle$ of TMs M that halt in $f(n)$ space, and which reject w. This ensures that D behaves differently than all smaller Turing machines! □

The technical condition mentioned before is that f has to be space-constructible:

Definition 19.2. $f : \mathbb{N} \to \mathbb{N}$ is **space-constructible** if some $O(f(n))$ space TM can compute the function $a^n \to a^{f(n)}$ and $f(n) \geq \log n$.[8]

In particular, if f is really small the Space Hiearchy Theorem fails. For example, if $f(n) = \log\log\log n$ and $g(n) = 1$, f and g can recognize the same set of languages – namely, the regular languages.

There are some bugs with the above description of D, but these can be resolved with a little tinkering. For example, the machine M might use $o(f(n))$ space with a big constant factor, so it takes more than $f(n)$ space for small inputs. We fix this by making D accept the descriptions $w = \langle M \rangle 10^*$ of TMs M that halt in $f(n)$ space.

Another bug: what happens if M loops on its finite memory? We fix this by terminating D after $2^{f(n)}$ transitions.

Remark 19.3. What happens if we pass in $w = \langle D \rangle$? It turns out there's no contradiction: there's some overhead to the simulation, so when D tries to simulate itself and the simulated D marks off the $f(n)$th space, it will use more than $f(n)$ spaces in the tape of the original D, causing the original D to reject.

19.2.2 Time Hierarchy Theorem

Definition 19.4. A function $f : \mathbb{N} \to \mathbb{N}$ is **time-constrictible** if some $O(f(n))$ time TM can compute $a^n \to a^{f(n)}$.

[8]Here, a^n denotes the string of n instances of the symbol a.

Theorem 19.5 (Time Hierarchy Theorem)
For time constructible $f(n)$, if $g(n) = o\left(f(n)/\log f(n)\right)$, then

$$TIME(g(n)) \subsetneq TIME(f(n)).$$

Remark 19.6. Note that unlike the Space Hierarchy Theorem, g has to be smaller by at least a log factor. This is an artifact of the proof – nobody knows if it's actually necessary.

Proof of Time Hierarchy Theorem. We take the same approach as the Space Hierarchy Theorem. We construct a Turing Machine D that operates as follows:

On input w, D computes $f(n)$. D will reject if it uses more than $f(n)$ time.

If $w \neq \langle M \rangle 10^*$, D rejects.

Else, w simulates M on w. If M accepts, D rejects. If M halts and rejects, D accepts.

The reason for the $\log f(n)$ factor is that D needs $O(\log f(n))$ space to store the time countdown. □

20 November 29, 2016

20.1 Natural Intractable Problems

We can define the class of problems taking exponential time and space:

Definition 20.1. The class $EXPTIME$, problems taking at most exponential time, is defined by

$$EXPTIME = \bigcup_k TIME\left(2^{n^k}\right).$$

Definition 20.2. The class $EXPSPACE$, problems taking at most exponential space, is defined by

$$EXPSPACE = \bigcup_k SPACE\left(2^{n^k}\right).$$

We know, from the Hierarchy Theorems, that $P \subsetneq EXPTIME$ and $PSPACE \subsetneq EXPSPACE$. To prove the hierarchy theorems, we constructed an unnatural-looking language that was in one class but couldn't possibly be in the other.

We will construct a natural language that can be solved in exponential time, but not in polynomial time.

Definition 20.3. Let R be a regular expression. The **Regular Expression with exponentiation** ($REX\uparrow$) R^k is a concatenation of k strings, all matching the regular expression R.

We define the problem:

$$EQ_{REX\uparrow} = \{\langle R_1, R_2\rangle | R_1,\, R_2 \text{ are } REX\uparrow\text{s such that } L(R_1) = L(R_2).\}$$

We will prove:

Theorem 20.4
$EQ_{REX\uparrow} \in EXPSPACE \setminus PSPACE$.

We make the following definition:

Definition 20.5. A problem B is $EXPSPACE$-complete if:

- $B \in EXPSPACE$;
- For all $A \in EXPSPACE$, $A \leq_p B$.

We will show:

Proposition 20.6
$EQ_{REX\uparrow}$ is EXPSPACE-complete.

Since the Hierarchy Theorem says that $EXPSPACE \neq PSPACE$, this proposition will imply that $EQ_{REX\uparrow} \in EXPSPACE \setminus PSPACE$.

So, how do we make an $EXPSPACE$ algorithm for $EQ_{REX\uparrow}$?

We'll first make an $NPSPACE$ algorithm for EQ_{REX}. Let R_1, R_2 be regular expressions. These regular expressions have corresponding NFAs N_1, N_2; the size of these NFAs bounds the size of a string that is accepted by one but not the other. Thus, an $NPSPACE$ algorithm deciding EQ_{REX} can nondeterministically guess all sufficiently short strings; if some thread finds a string that matches one of R_1, R_2 but not the other, it rejects, and otherwise it accepts.

But, by Savitch's Theorem $NPSPACE = PSPACE$, so in fact we have a $PSPACE$ algorithm for EQ_{REX}.

Since EQ_{REX} is in $PSPACE$, by repetition $EQ_{REX\uparrow}$ is in $EXPSPACE$. Thus it suffices to prove:

Proposition 20.7

$EQ_{REX\uparrow}$ is $EXPSPACE$-hard.

Proof. Let $A \in EXPSPACE$, decided by a TM M in space 2^{n^k}. We will give a polynomial-time reduction $f : A \to EQ_{REX\uparrow}$.

On input w, f has to compute $f(w) = \langle R_1, R_2 \rangle$. f will set R_1 to Σ^*, the set of all strings, and R_2 to Σ^* except a rejecting computation history of M on w.

So, what does a rejecting computation history of M on w look like? It's a string $c_1\#c_2\#\dots\#c_{rej}\#$. We will pad each c_i with spaces at the end, so it is exactly 2^{n^k} cells long.

We construct $R_2 = R_{bad-start} \cup R_{bad-reject} \cup R_{bad-move}$. These are all the ways a string can fail to be a rejecting computation history: a string can fail to start at the start state, fail to end at the reject state, or fail a transition somewhere in the middle.

Let the start configuration be $c_1 = q_0 w_1 w_2 \dots w_n$, followed by $2^{n^k} - n - 1$ blanks. We define s_0 as all strings that don't start with a q_0:

$$s_0 = (\Sigma - s_0)\Sigma^*.$$

Likewise, we define s_1 as all strings that don't have a w_1 in the second position:

$$s_1 = \Sigma(\Sigma - w_1)\Sigma^*,$$

and so on, to

$$s_n = \Sigma^n(\Sigma - w_n)\Sigma^*.$$

We define s_{blank} as the strings in which one of the $n+1$th through 2^{n^k} th positions are not blank:

$$s_{blank} = \Sigma^{n+1}(\Sigma \cup \epsilon)^{2^{n^k}-(n+1)}(\Sigma - blank)\Sigma^*.$$

Finally, we set $s_\#$ to the strings in which the $2^{n^k}+2$th position is not $\#$:

$$s_\# = \Sigma^{2^{n^k}+1}(\Sigma - \#)\Sigma^*.$$

Then, we can construct

$$R_{bad-start} = s_0 \cup s_1 \cup \cdots \cup s_n \cup s_{blank} \cup s_{\#}.$$

We can define $R_{bad-reject} = (\Sigma - q_{rej})^*$.

Finally, we define $R_{bad-move}$ as follows. Recall that in the proof of the Cook-Levin theorem, we showed that an accepting computation tableau is valid iff every 3×2 window in the tableau is valid. Thus:

$$R_{bad-move} = \Sigma^* \left(\bigcup_{\text{bad window (abc,def)}} abc\Sigma^{2^{n^k}-2} def \right) \Sigma^*$$

□

Joke. "I hate the word 'clearly' – it's so dangerous. It's clear to me, that's what I know." - Prof. Sipser

It follows that $EXP_{REX\uparrow}$ is in $EXPSPACE \setminus PSPACE$.

20.2 Oracles and Relativization

We've shown a reasonable decidable langauge that is provably outside P. This proof rested on the diagonalization method used to prove the Hierarchy Theorems.

Now, we discuss a philosophical question: can we use a similar procedure to show that satisfiability is not in P? We will show that an "unadorned" diagonalization method won't work.

Recall that a diagonalization method is really a simulation – the simulating machine simulates the simulated machine and behaves differently from it.

Given a language A, we say an **oracle** for A is a black-box that answers, for any w, whether $w \in A$ for free.

Joke. "We call it an oracle because it's magical. But you don't have to go up to the mountain top!" - Prof. Sipser

We use M^A to denote a TM, equipped with an oracle for the language A. We can define P^A as the set of languages which can be decided in polynomial time by TMs equipped with an A-oracle.

In particular, note that because SAT is NP-complete, we have $NP \subseteq P^{SAT}$ and $coNP \subseteq P^{SAT}$.

We can likewise define NP^A as the set of languages which can be decided in nondeterministic polynomial time with the help of an A-oracle. $coNP^A$ is defined similarly.

We will complete this discussion next time.

21 December 1, 2016

21.1 Oracles and Relativization, continued

We begin with the following surprising theorem:

Theorem 21.1

There exists an oracle A such that $P^A = NP^A$. There is also an oracle B such that $P^B = NP^B$.

Proof. We will only prove the first claim.

We claim that $P^{TQBF} = NP^{TQBF}$. Clearly $P^{TQBF} \subseteq NP^{TQBF}$. Observe that:

$$NP^{TQBF} \subseteq NPSPACE = PSPACE \subseteq P^{TQBF}.$$

The first inclusion is true because an NPSPACE machine can replace every call to the TQBF oracle with a direct computation. The middle equality follows from Savitch's Theorem. The last inclusion follows from TQBF being PSPACE-complete.

Thus $NP^{TQBF} = P^{TQBF}$ and we are done. □

Now, why does this mean that we can never use a diagonalization argument to show $P \neq NP$?

Note that if a machine M simulates a machine N, then for any oracle A, M^A can also simulate N^A – whenever N calls oracle A, M can also call A. If we could use diagonalization to show $P \neq NP$, diagonalization would also show $P^A \neq NP^A$ for all A. But, the above theorem shows this is impossible.

21.2 Probabilisitic Computation

We consider a model of computation in which Turing machines have access to a source to randomness. It appears that in this model, Turing machines can do more than they could in deterministic models.

We can model NTMs (in the non-probabilistic context) as machines where in each step, the machine has 1 or 2 possible next moves.[9].

In the probabilistic model, if the machine M has 2 possible next moves, we say M flips a fair coin to choose which move it picks.

Then, any outcome that occurs after k coin tosses occurs with probability 2^{-k}. On input w, we say that

$$\Pr[M \text{ accepts } w] = \sum_{\text{accepting branches } b} \Pr[b].$$

[9]If an NTM has a branching factor larger than 2, we can decompose the branching into a tree.

Example 21.2

Suppose we have a nondeterministic machine M for SAT that, on input a boolean formula ϕ, probabilistically guesses all assignments of variables. If there is exactly one assignment of variables that satisfies ϕ, the probability that M correctly decides SAT is exponentially small.

21.3 BPP

Ideally, we want our probabilistic Turing Machines to get the right answer with high probability. This motivates the following definition:

Definition 21.3. A probabilisitc TM M decides a language A with error probability ϵ ($\epsilon > 0$) if for all w, M on w gives the right answer with probability $\geq 1 - \epsilon$.

Once we have this, we can define a class of languages that can be solved by probabilistic TMs with high probability:

Definition 21.4. The class BPP (for "Bounded Probabilistic Poly-Time") is:

$$BPP = \{A | A \text{ is decided by a probabilistic poly-time TM with error probability } \frac{1}{3}\}.$$

Remark 21.5. In a slightly different class of languages, the *runtime* of the machine can also be probabilisitic – there are some branches that may run for a really long time. This class is called ZPP.

The $\frac{1}{3}$ in the definition of BPP is actually completely arbitrary. If a probabilistic TM has error probability $\frac{1}{3}$, we can bring the error probability down to any $\epsilon > 0$ by running this machine a bounded number of times and taking the majority vote of the results.

$P \subseteq BPP$ is trivial – every program in P is also in BPP because we can just not use randomness.

$NP \subseteq BPP$ is not known.

21.4 Branching Programs

We introduce a new model of computation. In a **branching program**, we have a set of nodes, two of which are labeled $0, 1$, and the remaining of which are labeled by variables $x_1, x_2, \dots, x_n$ (possibly with repetition). Each node except $0, 1$ has two outgoing edges, which are labeled $0, 1$.

On input some bit-string $b_1 b_2 \dots, b_n$, the branching program starts at b_1. Whenever the program arrives at a node x_i, it looks at the bit b_i and walks on the outgoing edge labeled with b_i.

Definition 21.6. We say branching programs B_1, B_2 are **equivalent** if they compute the same function. We define:

$$EQ_{BP} = \{\langle B_1, B_2 \rangle | B_1, B_2 \text{ are equivalent branching programs}\}.$$

In the homework, we will prove:

Proposition 21.7

EQ_{BP} is $coNP - complete$.

We won't be able to prove things like $EQ_{BP} \in BPP$, because this would immediately collapse all of $NP, coNP$ into BPP. Instead, we aim for a more modest result:

Definition 21.8. A **Read-Once Branching Program** (ROBP) is a Branching PRogram that can read a variable at most one time on each path. EQ_{ROBP} is the language of pairs $\langle B_1, B_2 \rangle$ of equivalent ROBPs B_1, B_2.

Theorem 21.9

$EQ_{ROBP} \in BPP$.

This proof is a bit esoteric, and will take all of next lecture.

For now, let's try something crude: suppose we guess an arbitrary input w and run B_1, B_2 on w. If the two programs return different results, we can conclude they're different. But, if they return the same result, we have to keep trying.

It's possible to make ROBPs that disagree on only one input.[10] So, if we take this approach, we might have to try an expected exponential number of times to find the disagreeing input.

Instead of random input, we will expand the input alphabet. So, instead of inputting 0s and 1s, we'll also allow input of other integers; using algebra, we can extend the computation of the ROBPs to make sense of these inputs. We argue that if two ROBPs disagree on the boolean inputs in at least one place, they disagree in the generalized inputs in most places. Then, we can do random sampling on the generalized inputs!

[10]For example, the program that always returns 1, versus the program that computes the logical or of all the input bits.

22 December 6, 2016

22.1 BPs and Arithmetization

We will show how to decide EQ_{ROBP} with high probability.

Without the read-once condition, this language is coNP-complete (proof: on pset). With the read-once condition, we will show how to decide this language in BPP.

Although the read-once condition formally means that every variable is read *at most* once on a branch, we will, for now, take it to mean that every variable is read *exactly* once on a branch. Formally:

$$EQ_{ROBP} = \{\langle B_1, B_2\rangle | B_1,\, B_2 \text{ are ROBPs and } B_1 \equiv B_2\}$$

Recall from last time, the naive solution: on input $\langle B_1, B_2\rangle$ we evaluate B_1, B_2 on some random boolean $x_1 x_2 \dots x_n$, accept if they always agree, and reject if they ever disagree. The problem with this approach is that B_1, B_2 might disagree on an exponentially small number of inputs, so we have to run an exponential number of iterations to reject with confidence.

We can do better by extending the ROBP to non-boolean inputs.

22.2 Digression: Algebra

Recall this fact from high school algebra:

Lemma 22.1
A polynomial $p(x) = a_d x^d + a_{d-1}x^{d-1} + \cdots + a_1 x + a_0$ of degree $\le d$ has $\le d$ roots, unless it is the zero polynomial, in which case it is zero everywhere.

As a consequence:

Corollary 22.2
If $p_1(x), p_2(x)$ are polynomials of degree $\le d$, then either $p_1(x) = p_2(x)$ everywhere, or $p_1(x) = p_2(x)$ at at most d points.

Proof. Consider the polynomial $p = p_1 - p_2$. □

This motivates our strategy for deciding EQ_{ROBP}: we extend B_1, B_2 to polynomials in a way that the polynomials emulate the Boolean formulas on Boolean inputs (encoded as 0, 1). On the generalized set of inputs, the above Corollary implies that B_1, B_2 either agree in very few places, or agree everywhere.

Actually: computing on real numbers $\mathbb{R}$ is hard, so we'll compute on $\mathbb{F}_q$, the integers modulo q (for some prime q we will decide later). It turns out (we won't prove this) the above results on polynomials work in $\mathbb{F}_q$[11] as well. This gives us:

[11] In fact, on any field $\mathbb{F}$.

Lemma 22.3

If $p(x)$ has degree $\le d$ and is not the zero polynomial, then for a random $r \in \mathbb{F}_q$,

$$\Pr[p(r) = 0] \le \frac{d}{q}.$$

Proof. This just says that any p has $\le d$ roots. □

We'll use the multivariate version of this lemma, which we won't prove:

Lemma 22.4 (Schwartz-Zippel Lemma)

For a nonzero polynomial $p(x_1, \dots, x_m)$, with degree $\le d$ in each variable, and uniformly random $r_1, \dots, r_m \in \mathbb{F}_q$:

$$\Pr[p(r_1, \dots, r_m) = 0] \le \frac{md}{q}.$$

22.3 Arithmetization

We will simulate $\wedge$ and $\vee$ with the arithmetic operations $+$ and $\times$. We simulate:

- $a \wedge b$ with $a \times b$;
- $\overline{a}$ with $1 - a$;
- $a \vee b$ with $a + b - ab$ (the $-ab$ is so $1 \vee 1 = 1$).
- If we know that a, b are not both 1, we can simulate $a \vee b$ with $a + b$. (This is called the "disjoint or.")

Let's do an example. The ROBP in Figure 1 computes the XOR function $x_1 \oplus x_2$,

Given an input x_1, x_2, we write 1s on all nodes and edges in the path the ROBP takes, and 0s everywhere else. Figure 2 shows this labeling when $x_1 = 1, x_2 = 0$.

We can write Boolean formulas for these labels in terms of x_1, x_2, as shown in Figure 3.

In general, if node x_i is labeled with an expression b, the 1-edge coming out of x_i is labeled $b \wedge x_i$, and the 0-edge is labeled $b \wedge \overline{x_i}$.

Also, if a node has incoming edges $b_1, b_2, \dots, b_i$, the node is labeled $b_1 \vee b_2 \vee \dots \vee b_i$. Moreover, these edges are disjoint, so these or's are actually disjoint or's.

The branching program accepts if we hit the node labeled 1, so in fact the entire branching program just computes the expression at the 1 node.

We claim that this recursive labeling has the following nice property:

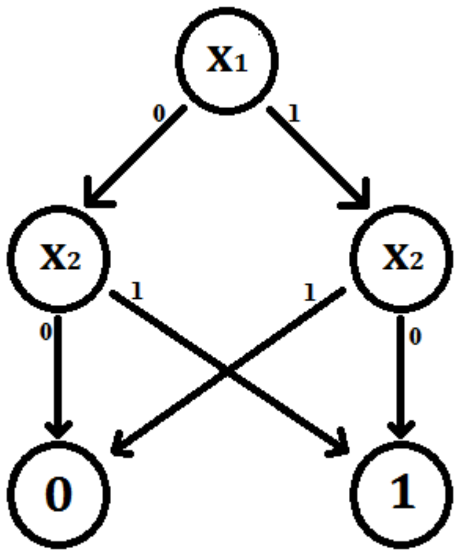

Figure 1: An ROBP computing $x_1 \oplus x_2$

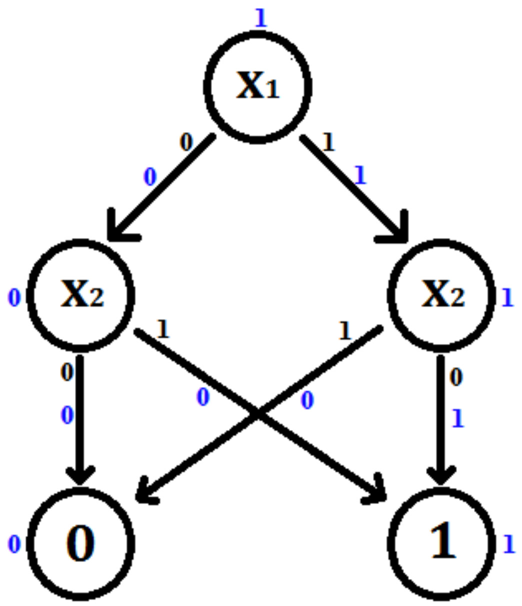

Figure 2: Labels (in blue) when $x_1 = 1, x_2 = 0$.

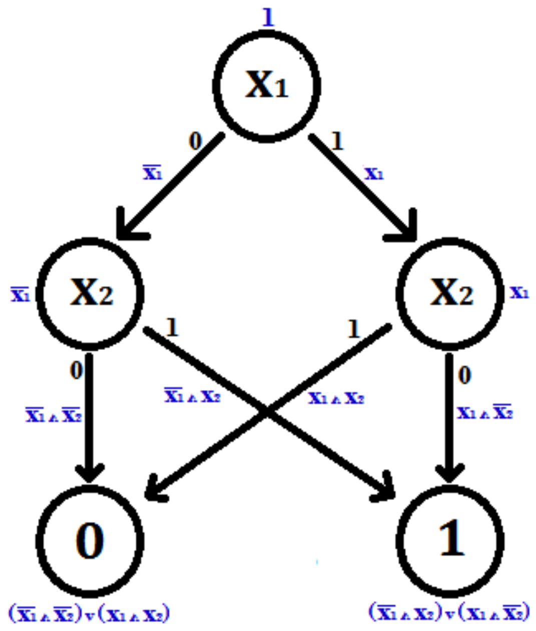
1
X1
0
1
X2
X2
0
1
1
0
0
1
$\overline{x_1}$
x_1
$\overline{x_1}$
x_1
$\overline{x_1} \wedge \overline{x_2}$
$\overline{x_1} \wedge x_2$
$x_1 \wedge \overline{x_2}$
$x_1 \wedge x_2$
$(\overline{x_1} \wedge \overline{x_2}) \vee (x_1 \wedge x_2)$
$(\overline{x_1} \wedge x_2) \vee (x_1 \wedge \overline{x_2})$

Figure 3: Boolean formulas for labels (in blue)

Lemma 22.5

The 1 node of this ROBP is labeled with a logical or of clauses of the form

$$y_1 \wedge y_2 \wedge \cdots \wedge y_n,$$

where for each i, y_i is x_i or $\overline{x_i}$.

Proof. We choose to write all labels in conjunctive normal form, by always distributing logical ands over logical ors. Then, each clause in a node or edge's label corresponds to a path from the root to the node or edge; a x_i indicates that the path included the 1 out-edge of a node x_i, and a $\overline{x_i}$ indicates that the path included the 0 out-edge of a node x_i.

Since each path from the root to the 1 node must read each variable exactly once, the lemma follows. □

Now, we can replace the Boolean expressions with our arithmetical extensions! Since our or's are all disjoint or's, we choose to arithmetize them all as just addition. Figure 4 shows the arithmetization of our example.

The extended branching program now computes the polynomial at the node labeled 1.

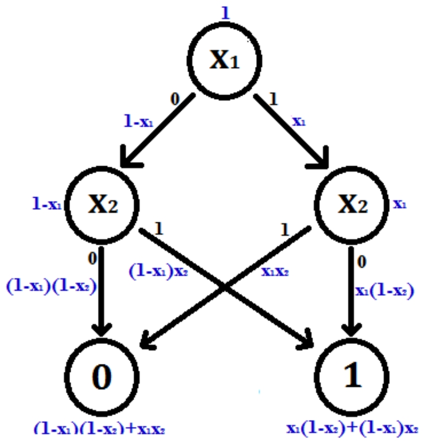

Figure 4: Arithmetization of labels

Example 22.6

At the risk of being repetitive:

We see that the ROBP for the operation $a \oplus b$ computes the Boolean expression

$$x_1 \oplus x_2 = (x_1 \wedge \overline{x_2}) \vee (\overline{x_1} \wedge x_2).$$

This gets arithmetized as

$$x_1 \oplus x_2 = x_1(1 - x_2) + (1 - x_1)x_2.$$

On inputs $x_1, x_2 \in \{0, 1\}$, this emulates the Boolean expression. For example,

$$0 \oplus 1 = (0)(1 - 1) + (1 - 0)(1) = 1.$$

But, this is perfectly well-defined on other integer inputs! For example,

$$2 \oplus 3 = (2)(1 - 3) + (1 - 2)(3) = -7.$$

This equation doesn't "mean" anything mathematically. But that isn't the point – this is a well-defined, polynomial extension of the XOR function, which lets us use properties of polynomials in our BPP algorithm.

Joke. "Now, before we get carried away here, we didn't discover some fundamental truth of the universe, that $2 \oplus 3 = -7$. " - Prof. Sipser

Lemma 22.5 yields the following corollary about the arithmetization:

Corollary 22.7

The 1 node of the arithmetized ROBP is labeled with a sum of products of the form

$$y_1 y_2 \cdots y_n,$$

where for each i, y_i is x_i or $1 - x_i$.

Proposition 22.8

Let p_1, p_2 be the polynomials computed by the arithmetizations of B_1 and B_2. Then, $B_1 \equiv B_2$ iff $p_1 = p_2$.

Proof. By Corollary 22.7 given the polynomial p computed by a ROBP B, we can read off the truth table of B. Consequently, the polynomial computed by B depends only on B's truth table, and not how B is structured on an ROBP. So, $B_1 \equiv B_2$ implies $p_1 = p_2$.

If $B_1 \not\equiv B_2$, they differ on some Boolean input, so obviously $p_1 \neq p_2$. $\square$

Proposition 22.8 implies that the polynomials computed by B_1, B_2 have degree n. Schwarz-Zippel then implies the following:

Proposition 22.9

Let $q > 3n$. If $B_1 \not\equiv B_2$, on a uniformly random input $r \in \mathbb{F}_q$, the polynomials P_{B_1} and P_{B_2} computed by the arithmetizations of B_1, B_2 agree with probability $< \frac{1}{3}$.

Thus, our algorithm is: on input $\langle B_1, B_2 \rangle$, pick a prime $q > 3n$ and a random $r \in \mathbb{F}_q$. Compute the evaluations of r on the polynomials P_{B_1} and P_{B_2}. If they are equal, accept; else, reject.

If $B_1 \equiv B_2$, this algorithm is correct all the time. If $B_1 \not\equiv B_2$, Proposition 22.9 implies that we accept with probability $< \frac{1}{3}$.

Remark 22.10. The polynomial P_B corresponding to an ROBP B takes an exponential time to write down, but this is fine - our algorithm just computes the evaulation of P_B on the fly. Given input r, we can compute all the labels numerically rather than symbolically – we never need to explicitly write P_B down!

23 December 8, 2016

23.1 BPP and RP

Recall from last time, that

$$BPP = \{A | A \text{ is decided by a probabilistic poly-time TM with error probability } \frac{1}{3}\}.$$

But, it turns out that a lot of BPP algorithms make errors only on one side. For example, our algorithm for EQ_{ROBP} from last time, on $\langle B_1, B_2 \rangle$, always returns the right answer if B_1, B_2 are equivalent, and makes a mistake with small probability if B_1, B_2 are not equivalent.

We have a name for problems solvable by such algorithms:

Definition 23.1. *RP is the set of languages A such that for some prob. poly-time TM M:*

- *If $w \in A$, M accepts w with probability $\geq \frac{2}{3}$;*
- *If $w \notin A$, M rejects w with probability 1.*

23.2 ISO

Given undirected graphs A, B, we say A is **isomorphic** to B ($A \equiv B$) if there's a bijection between A's vertices and B's vertices that respects adjacency (i.e. the graphs "look the same" when we match corresponding vertices).

We can define the problem:

$$ISO = \{\langle A, B \rangle | A \equiv B\}.$$

This is clearly an NP problem – if A, B are isomorphic, the bijective mapping is the certificate.

Is ISO in P? This isn't known. What's interesting is that it isn't known whether ISO is NP-complete, either – usually, problems in NP can either be shown to be in P, or shown to be NP-complete.

23.3 Interactive Proofs

Intuition: probabilistic algorithms are to P as interactive proofs are to NP.

The basic framework of interactive proofs is an interaction between a "prover" and a "verifier." The prover has unlimited computational resources, and the verifier is bounded by probabilistic polynomial time. The verifier wants the answer to a question and assigns the question to the prover.

But, the verifier doesn't trust the prover. The prover gets the answer and needs to convince the verifier of the answer, in a way that the verifier can check, with high confidence, in probabilistic polynomial time. A fraudulent prover shouldn't be able to convince the verifier of a wrong answer.

23.3.1 Example: ISO and non-ISO

Given two isomorphic graphs A, B, can the prover convince the verifier that the graphs are isomorphic? Yes – the prover just tells the verifier the mapping.

What if A and B are not isomorphic? The provers can convince the verifier with the following protocol:

- The verifier secretly flips a coin and picks one of A and B. He copies the graph he picked and scrambles the vertices (by a uniformly random permutation) into an isomorphic graph C, which he gives to the prover.
- The prover tells the verifier whether C is isomorphic to A or B.
- If the prover answers correctly, the verifier accepts; else the verifier rejects.

If A and B are not isomorphic, the prover can always complete the protocol. But, if A and B *are* isomorphic, C is isomorphic to A *and* B.[12]

So, if a fraudulent prover tries to convince the verifier that A and B aren't isomorphic, he can only complete the protocol half the time. If they repeat this protocol n times, and A and B are isomorphic, the fraudulent prover only completes the protcol with probability $\frac{1}{2^n}$.

By repeating the protocol enough times, the verifier can be sure, with arbitrarily high confidence, that the prover is telling the truth.

23.3.2 Formal Definition

We now formally define an interactive proof system:

Definition 23.2. An **interactive proof system** is an interaction between a prover P, a probabilistic TM with unlimited computation, and a verifier V, a probabilistic polynomial-time TM. Given input w to P and V, P and V exchange a polynomial-sized number of pessages until V outputs accept or reject.

Definition 23.3. The problem class IP is defined as the set of problems A such that for some V:

- If $w \in A$, $\Pr[P \leftrightarrow V \text{ accepts } w] \geq \frac{2}{3}$ for some P;
- If $w \notin A$, $\Pr[\tilde{P} \leftrightarrow V \text{ accepts } w] \leq \frac{1}{3}$ for *any* $\tilde{P}$.

The intuition for this definition is: if $w \in A$, an hoenst prover makes the verifier accept with high probability; if $w \notin A$, even a fraudulent prover can only make the verifier accept with low probability.

As before, the probability thresholds of $\frac{2}{3}$ and $\frac{1}{3}$ are arbitrary. By repeated, independent runs, we can amplify these probabilities arbitrarily close to 1 and 0.

Note that $NP, BPP \in IP$ trivially – $NP \in IP$ because the prover just provides the certificate, and $BPP \in IP$ because we don't need the prover.

[12]with identical probability distributions, because C is generated by a uniformly random permutation

23.3.3 Analysis of non-ISO interactive proof

If the graphs are not isomorphic, a genuine prover can make the verifier accept with probability 1.

If the graphs are isomorphic, a fraudulent prover can make the verifier accept with probability at most $\frac{1}{2}$. If we repeat the protocol twice, a fraudulent prover can make the verifier accept with probability at most $\frac{1}{4} < \frac{1}{3}$. Thus this is a protocol in IP.

23.3.4 $\#SAT$, $IP = PSPACE$

I had to leave class early today. In the rest of class we stated without proof that $IP = PSPACE$ and discussed the $\#SAT$ problem, which we claim is in IP.

The rest of the lecture content is in Section 10.4 of the text, from the discussion of $IP = PSPACE$ through the Proof Idea that $\#SAT \in IP$. The discussion on $\#SAT$ is summarized in next class's "Last Time" section.

24 December 13, 2016

24.1 Administrivia

The final exam is 3 hours, and will be open book / notes / handouts. It covers the whole course, but will focus on the 2nd half of the course.

24.2 Last Time

We define $\#\phi$ as the number of satisfying assignments of the Boolean formula ϕ, and

$$\#SAT = \{\langle \phi, k\rangle | k = \#\phi\}.$$

We introduce a function T that counts the number of satisfying assignments $x_1 \dots x_n$ with a specified prefix. Formally, for a formula ϕ on $x_1 \dots x_n$, we define

$$T(a_1 \dots a_i) = \sum_{a_{i+1},\dots,a_n \in \{0,1\}} \phi(a_1 \dots a_n).$$

We have, by definition, the recurrence relation $T() = \#\phi$, $T(a_1 \dots a_i) = T(a_1 \dots a_i 0) + T(a_1 \dots a_i 1)$.

An exponential-time protocol is as follows:

- P sends $T()$ to V, and V checks that $k = T()$;
- P sends $T(0)$ and $T(1)$ to V, and V checks that $T() = T(0) + T(1)$;
- P sends $T(00), T(01), T(10), T(11)$ to V, and V checks that $T(0) = T(00) + T(01)$, $T(1) = T(10) + T(11)$; and so on, until
- P sends $T(0 \dots 0)$ through $T(1 \dots 1)$, and V checks the corresponding relations.
- Finally, V checks that $T(0 \dots 0) = \phi(0 \dots 0)$, thorugh $T(1 \dots 1) = \phi(1 \dots 1)$.

This works: if a fraudulent prover claims that $k = \#\phi$ when this is not true, he has to lie in the first stel; this lie must be justified by a lie in the second step, which must be justiifed by a lie in the third step, and so on. When we get to the last step, the verifier will catch the lie.

24.3 A polynomial time IP protocol for #SAT

Of course, the exponential protocol is silly – IP algorithms aren't allowed to use an exponential-size interaction, and the verifier might as well just skip to the last step and do it himself. Moreover, this protocol doesn't use the power of interactive proofs. There's no randomness, and the verifier isn't saying anything back to the prover.

We'll use the exponential protocol to motivate the actual solution, which will be polynomial-length.

24.3.1 Arithmetization

We will use arithmetization on the boolean formulas – we simulate the operations $\wedge, \vee, \neg$ with the operations:

- $a \wedge b \to ab$
- $\overline{a} \to 1 - a$
- $a \vee b \to a + b - ab$.

This transforms ϕ to a polynomial p_ϕ of degree $\le |\phi|$, such that on inputs in $\{0, 1\}$ p_ϕ emulates ϕ. We work in integers modulo q, where $q > 2^m$ is a prime.

We define the generalized T in exactly the same way:

$$T(a_1 \dots a_i) = \sum_{a_{i+1}, \dots, a_n \in \{0,1\}} p_\phi(a_1 \dots a_m).$$

Note that when $a_1, \dots, a_i \in \{0, 1\}$, the generalized T is the same as the original T! Our arithmetized framework is a faithful simulation of the original problem. But, this function is also well-defined for any integers $a_1, \dots, a_i$!

Joke. "You can't put a 2 into an and gate! Lights flash, smoke comes out, and the program crashes." - Prof. Sipser

24.3.2 The IP Protocol

Here's the intuition for our procedure:

In our exponential procedure, we used $T(0)$ and $T(1)$ to justify $T()$, then $T(00), T(01), T(10), T(11)$ to justify $T(0)$ and $T(1)$, and so on, down through a tree of justifications. The problem with the tree is that it blows up exponentially.

Instead of taking in strings of bits $x_1 \dots x_m$, our arithmetized p_ϕ takes in input sequences of *integers* $r_1 \dots r_m$. Then, we justify $T()$, with high confidence, with just *one* value $T(r_1)$, justify $T(r_1)$ with just one value $T(r_1 r_2)$, and so on. Our tree doesn't blow up exponentially, so we're happy.

Our protocol is as follows: on input $\langle \phi, k \rangle$:

- P sends $T()$ to V; V checks that $k = T()$.
- P sends $T(z)$ *as a polynomial in* z to V.[13] V computes $T(0)$ and $T(1)$ and checks that $T() = T(0) + T(1)$; V selects a *random* $r_1 \in \mathbb{F}_q$ to P and asks P to convince it that the polynomial $T(z)$ is correct at $z = r_1$.
- P sends $T(r_1 z)$ as a polynomial in z to V. V computes $T(r_1 0)$ and $T(r_1 1)$ and checks that $T(r_1) = T(r_1 0) + T(r_1 1)$; V selects a random $r_2 \in \mathbb{F}_q$ and asks P to convince it that $T(r_1 z)$ is correct at $z = r_2$.
- P sends $T(r_1 r_2 z)$ as a polynomial in z, and so on.

[13] P can do this by formally evaluating T, with a variable in place of x_1.

Why does this work? If the P is honest, then V can check everything in polynomial time, and P always does the right thing, so V accepts.

Let's suppose we have a fraudulent prover $\tilde{P}$, and tries to convince V that $\langle \phi, k \rangle \in \#SAT$, for some wrong value of k.

In the first step, the value $\tilde{P}$ gives, $\tilde{T()}$, must be a lie.

So, in the second step, the $\tilde{T}(z)$ P gives must be such that at least one of $\tilde{T}(0)$ and $\tilde{T}(1)$ is wrong. So, $\tilde{T}(z)$ must be wrong!

We're evaluating $\tilde{T}(z)$ at a random place r_1. Remember that two low-degree polynomials can agree at very few points. So, at r_1, the evaluation of $\tilde{T}(z)$ probably disagrees with the evaluation of the actual $T(z)$ at r_1.

Then, the $\tilde{T}(r_1 z)$ P gives must be be such that at least one of $\tilde{T}(r_1 0)$ and $\tilde{T}(r_1 1)$ is wrong. So, $\tilde{T}(r_1 z)$ must be wrong.

Therefore, with high probability a lie at one stage of the protocol forces a lie at the next stage, and the verifier can catch the lie at the end.

Of course, the fraudulent prover $\tilde{P}$ might get lucky. If at some stage, the evaluation of the polynomial $\tilde{T}(r_1 \dots r_i z)$ that $\tilde{P}$ gives agrees with the evaluation of the true polynomial $T(r_1 \dots r_i z)$ at r_{i+1}, $\tilde{P}$ can be honest from that point forward and fool the verifier. But, a probability computation shows that this happens with low probability.

Joke. "If the prover sends something ridiculous, like 'vote for Trump,' the verifier can obviously just reject! " - Prof. Sipser

Since the fraudulent prover can make V accept with only low probability, we are done.