18.404

Rachel Wu

Fall 2017

These are my lecture notes from 18.404, Theory of Computation, at the Massachusetts Institute of Technology, taught this semester (Fall 2017) by Professor Michael Sipser[1].

I wrote these lecture notes in LaTeX in real time during lectures, so there may be errors and typos. I have lovingly pillaged Evan Chen's and Tony Zhang's formatting commands. Should you encounter an error in the notes, wish to suggest improvements, or alert me to a failure on my part to keep the web notes updated, please contact me at rmwu@mit.edu.

This document was last modified 2017-12-08. The permalink to these notes is http://web.mit.edu/rmwu/www.

[1] sipser@mit.edu

Contents

1 September 7, 2017

1.1 Administrivia

- Lectures are Tuesdays and Thursdays, 2:30-4p in 2-190.
- This course is split into two halves: computability and complexity. Computability is essentially a solved problem, but complexity deals with "what is computable, in practice?"

1.2 Finite automata

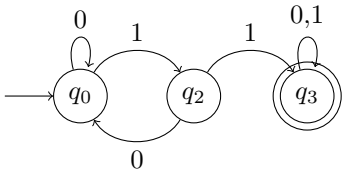


Figure 1: Finite automaton. If the input leads to a double circle, the input is accepted. Otherwise the input is rejected.

Collections of strings in this topic are called **languages**. Let the finite automaton be A. Then, $L(A)$ is the language of A. Here, the set $\{w|w \text{ has substring } 11\}$ is the set of accepted inputs.

Definition 1.1. A **finite automaton** M is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

where Q is a finite set of states, Σ is a set of input symbols, $\delta : Q \times \Sigma \to Q$ is the transition function,[2] q_0 is the starting state, and F is the set of accept states.

Definition 1.2. M **accepts** input $w = w_1, w_2, \ldots, w_n$ (for $w_i \in \Sigma$) if there is a corresponding sequence $r_0, r_1, r_2, \ldots, r_n \in Q$.

Observe that $r_0 = q_0$, since we must start at q_0. Then given some r_i, $r_{i+1} = \delta(r_{i+1}, w_0), \forall 1 \leq i \leq n$. Finally, r_n must be an accepting state. So $L(M) = \{w|M \text{ accepts } w\}$ is the language of M, and M "recognizes" $L(M)$.

Definition 1.3. Language A is **regular** if $A = L(M)$ is the language for some finite automaton M.

Conceptually, a finite automaton can represent anything a finite computer can compute. We now characterize the family of regular languages.

[2] Here, we use δ as $\delta(q_2, 0) = q_i$.

1.3 Regular languages

There are three **regular operations**: union $\cup$, concatenation $\circ$, and star *. We refer to ab as the concatenation of string a and string b. Given two languages A and B, their union is

$$A \cup B = \{w | w \in A \text{ or } w \in B.\} \tag{1.1}$$

Concatenation is the cartesian product of A and B.

$$A \circ B = \{xy | x \in A, y \in B\} \tag{1.2}$$

Star is a unary symbol, applied only on A.

$$A^* = \{w = x_1 x_2 x_3 \dots x_k | x_i \in A, k \geq 0\} \tag{1.3}$$

Note that A^* must contain the empty string, which the professor *emphasizes* is different from the empty set.

We can use regular operations to build up regular expressions. Atomic regular expressions are members of the alphabet, $\Sigma, \epsilon, \emptyset, \Sigma.$, where ϵ is the empty string. Composite regular expressions are $r_1 \circ r_2, r_1 \cup r_2, r^*$, where r_1, r_2 are regular expressions.

Example 1.4

Suppose we have atomic elements 0 and 1. Then $(0 \cup 1)^*$ is the set of all possible strings Σ^*.

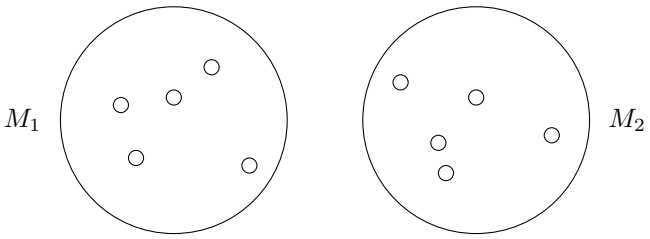
There are some interesting quirks.

- If we take $\emptyset \circ A$, then we get $\emptyset$ since there are no strings in $\emptyset$ to choose as a first element.
- If we want A, we would do $\epsilon A = A$.
- Also, $\emptyset^*$ is the set that has only the empty string.

Theorem 1.5

If A_1 and A_2 are regular languages, so is $A_1 \cup A_2$. That is, the collection of regular languages is closed under union.

Proof. Let finite automata M_1, M_2 recognize A_1, A_2, respectively. We construct $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes $A_1 \cup A_2$.



The naive approach is to feed the input twice, once into each and reset, but we're not allowed to choose the path; we can only feed the input once. Theoretically, we could run the paths in parallel, and if either accepts, we accept.

We construct M as follows.

1. $Q = Q_1 \times Q_2 \{qr | q \in Q_1, r \in Q_2\}$, or the cartesian product of the two automata's states.
2. The transition function is $\delta((qr), a) = \delta_1(q, a)\delta_2(r, a)$
3. The accept states are $F = Q_1 \times F_2 \cup F_1 \times Q_2 = \{fg | f \in F_1 \text{ or } g \in F_2\}$.

□

2 September 8, 2017

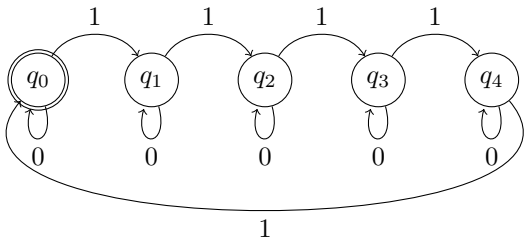
Our TA is Rishad,[3] and office hours are Tuesdays 12:30-2:30p in G6 lounge. Recitations don't cover new material, but we go over problems.

2.1 Practice problems

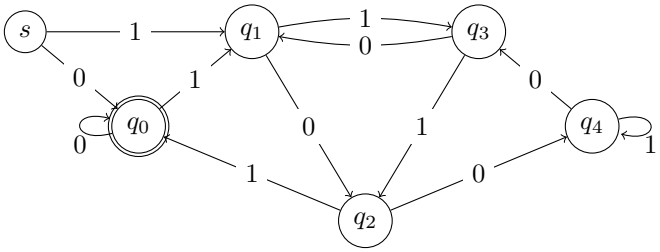
Exercise 2.1. Construct finite automata to show that the following languages are regular. Let $\Sigma = \{0, 1\}^*$.

1. $L = \{w | \#$ of 1s in w is divisible by $5\}$
2. $L = \{w | w$ is divisible by $5\}$

We can solve the first question by counting the number of 1s.



We can perform long division with a state machine as well. For example, $59 = 111011_2$ divided by $5 = 101_2$ is 1101 with a remainder of 100.



For each character we read, we shift left by 1 bit, add the new character, and find the remainder modulo 5. Each state q_i corresponds to the current remainder i.

Question. Will it be useful if I take notes in class? Well you might want to jot down what's confusing... and we're opening a Piazza... but wait OF COURSE IT'S USEFUL TO TAKE NOTES!

Exercise 2.2. What's the regular expression for exactly one 1? 0^*10^* Two 1s? $0^*10^*10^*$ Number of 1s divisible by 5? $(0^*10^*10^*10^*10^*)^* \cup 0^*$

[3] rrrahman@mit.edu

3 September 12, 2017

3.1 Nondeterministic finite automata

Professor Sipser casually remarks to do extra problems if you want a recommendation letter.

Definition 3.1. A **nondeterministic finite automaton** $M = (Q, \Sigma, \delta, q_0, F)$ is a finite automaton where Q, Σ, q_0, F are standard, but

$$\delta = Q \times \Sigma = \mathcal{P}(Q) = \{R | R \subseteq Q\},$$

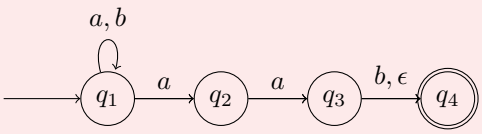
a set of possible states.

There are three equivalent ways to think about NFAs.

1. Computational: fork new threads, and if one thread accepts, we accept
2. Mathematical: tree of possibilities, and if any leaf accepts, the tree accepts
3. Magical: machine guesses a path and is always right

Example 3.2

Consider the following NFA.

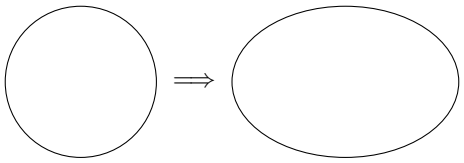


Inputs ab and $aabb$ are rejected, while aa and aab are accepted.

Theorem 3.3

If nondeterministic finite automaton N recognizes A, then A is regular.

Proof. We convert $N = (Q, \Sigma, \delta, q_0, F)$ into an equivalent deterministic finite automaton $M = (Q', \Sigma', \delta', q_0', F')$.



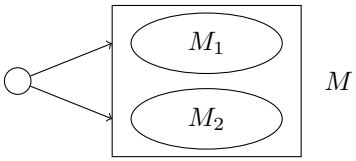
- The new set of states is $Q' = \mathcal{P}(Q)$.
- We obtain the new transition function by following each potential state. For $R \in Q'$ (or $R \subseteq N$) and $a \in \Sigma$, $\delta'(R, a) = \{q | q \in \delta(r, a)\}$ for any $r \in R$.

- The set of accepting states is $F' = \{R \in Q' | R \cap F \neq \emptyset\}$.

□

Now that we have introduced nondeterministic finite automata, we can provide a new proof for closure under $\cup$.

Proof of closure under union. We construct a NFA M with a single new start state that sends two threads, one through each of M_1, M_2.



□

Theorem 3.4

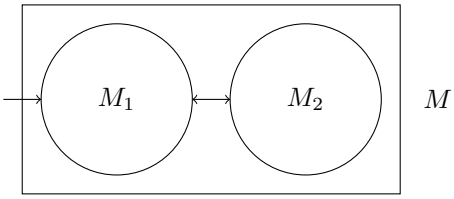
If A_1, A_2 are regular, so is A_1A_2. That is, concatenation is regular.

The naive approach is to first run through M_1, then M_2. However, the machine doesn't know where M_1 ends and M_2 begins. We could break off M_1 too early and not find M_2.

$$w = \underbrace{\epsilon A_1}_{} \underbrace{\qquad\qquad \epsilon A_2}_{}$$

Instead, we use nondeterministic finite automata.

Proof. We prove by construction. At any time, M_1 has the option to switch to M_2 whenever it finds an initial prefix of A_2.



We consider all potential states reached by the input. If at least one potential state is accepted at the end, then we accept.[4]

M accepts if some possible state accepts at the end of input.

□

[4]Put a finger on each potential state and if any of the fingers accepts, we accept. If there are no more paths, we remove the finger.

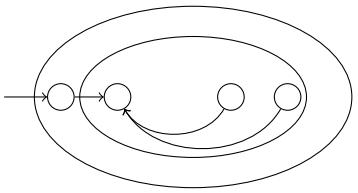
Theorem 3.5

If A_1 is regular, then so is A_1^*.

Proof. The new language is essentially a concatenation of many A_1s.

$$w = \underbrace{\in A_1 \quad \in A_1 \quad \in A_1}_{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad} \quad \dots$$

Every time we reach an empty string, we have the option of returning to the beginning. We make a new start state with an ϵ transition to the original start state.



□

3.2 Regular expressions to finite automata

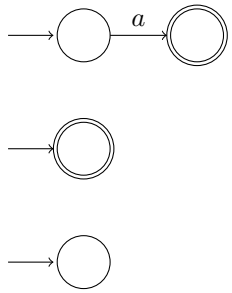
Now that we have proved closure under union, concatenation, and star, we can show that regular expressions generate finite automata.

Theorem 3.6

Every regular expression r generates a regular language.

Proof. If r is atomic, then we construct with the following rules.

$$r = \begin{cases} a \in \Sigma \\ \epsilon \\ \emptyset \end{cases}$$



Otherwise R is composed by regular expressions, for which we use the proven constructions.

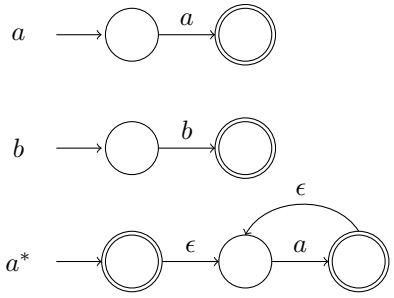
$$r = \begin{cases} r_1 \cup r_2 \\ r_1 r_2 \\ (r_1)^* \end{cases}$$

□

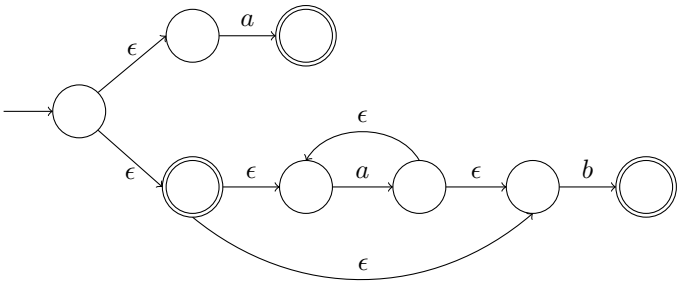
Example 3.7

Convert regular expression $a \cup a^*b$ into a NFA.

We first convert the "primitives."



Now we string them together.

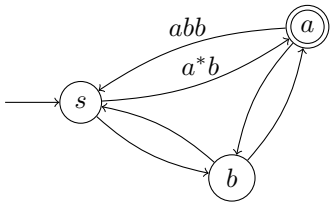


4 September 14, 2017

4.1 Finite automata to regular expressions

Recall that last class, we converted regular expressions to finite automata. Today, we prove that the converse is true as well.

We first introduce the **generalized nondeterministic finite automata** as a tool for this proof. Instead of single words, we accept arbitrary regular expressions on the transitions.



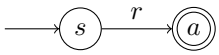
The GNFA also has the following properties:

1. There is a transition from every state to every other state, even if the accepted string is ϵ, except that
2. the start state has no in degrees, and
3. there is a unique accept state with no out degrees.
4. Every state also has a self-loop, possibly empty.

Proposition 4.1

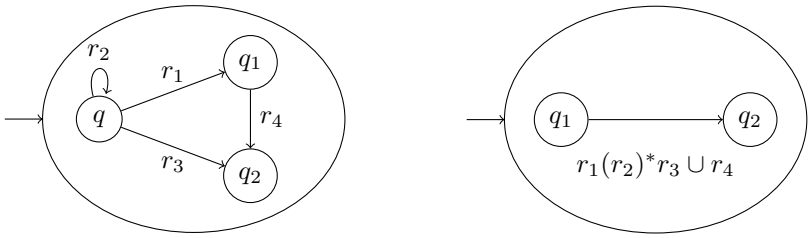
For every $k \geq 2$, we can convert a GNFA with k states to a regular expression R by induction on k.

Proof. The base case is $k = 2$, where $R = r$.



Now we induct on k. Suppose we have a k-state GNFA and we remove state q, which is neither the start nor accept state.

k-state GNFA $\qquad$ $k-1$-state GNFA



Suppose r_1 pointed from q_1 to q, r_2 looped back to q, r_3 pointed from q to q_2, and r_4 pointed from q_1 to q_2. Then after removing q, we replace the edge from q_1 to q_4 with $r_1(r_2)^*r_3 \cup r_4$. We repeat this for each q_1, q_2.

□

Theorem 4.2

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

Proof. We create a GNFA A from R with the above method. □

4.2 Non-regularity

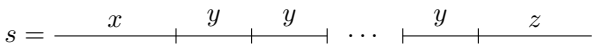
We provide a property that governs all regular expressions, so that if an expression does not obey that property, it is not regular.

Lemma 4.3 (Pumping lemma)

If A is a regular language, there exists a number p (pumping length) such that if $s \in A$ and $|s| \geq p$, then we can write $s = xyz$ where

1. $xy^iz \in A$ for all $i \geq 0$,
2. $y \neq \epsilon$,
3. and $|xy| \leq p$.

That is, any sufficiently long strings in A can be split into 3 pieces, where the middle piece is repeated.



Proof. Given regular language A, consider the DFA for A. By pigeonhole, some state *must* be repeated if the length of the input is longer than the number of states in the machine. If A is finite, then we set p greater than the maximum length string in A. □

Example 4.4

Let $A = \{0^k1^k | k \geq 0\}$, or all strings with a string of 0s, followed by an equal number of 1s. Show that A is not regular.

Proof. Assume for contradiction that A is regular. The pumping lemma gives the pumping length p. Let $s = 0^p1^p$. Since $s \in A$ and $|s| \geq p$, there must exist some partition of $s = xyz$. By (3), y can contain only 0s, but $xyyz$ has more 0s than 1s, so xyz is not in A. Therefore, A is not regular. □

Example 4.5

Let $B = \{w | w \text{ has equal \# of 0s and 1s}\}$. Show that B is not regular.

Proof. Regular languages are closed under intersection, so if B is regular, then $B \cap 0^*1^*$ is regular. However, this intersection is A, which is not regular, so B is not regular. □

4.3 Context-free grammars

Context-free grammars constitute a more powerful computational model, found in compilers and other fields.

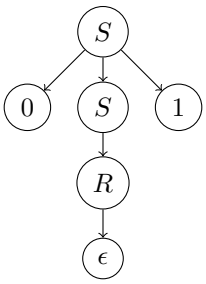
They consist of **substitution rules**, which map **variables** to **terminals**.

Example 4.6

A context-free grammar has symbols S, R with substitution rules

$$S \to 0S1 \quad S \to R \quad R \to \epsilon$$

and terminals on the right side.



This structure is known as a **parse tree**.

Here, $01 \in L(G) = \{o^k 1^k | k \geq 0\}$.

5 September 15, 2017

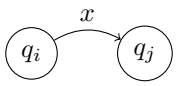
5.1 Reversibility

This proposition is useful for the problem set!

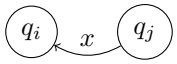
Proposition 5.1

If L is regular, then so is its reverse, $L^{\mathcal{R}}$.

Proof. We prove by construction. Suppose L is recognized by the following machine.



Then $L^{\mathcal{R}}$ is recognized by the machine we construct below.



We flip the arrows for each q_i, q_j pair, convert the start state into the accept state, and create a new start state with transitions to all the old accept states.

□

Proposition 5.2

To test for $L = 1^n$, we require at least n states.

Proof. If we required fewer, there may be a loop before reaching q_{n-1} from q_0. Then we could reach q_{n-1} without traversing the loop, and strings not in $\{1^n\}$ may be accepted. □

5.2 Practice problems

We are so "pumped" to do practice problems.

Example 5.3

Show that the following languages are not regular.

1. $L = \left\{0^i 1^j | i \geq j\right\}$
2. $L = \left\{0^i 1^{2i} | i \geq 0\right\}$
3. $L = \{1^n | n \text{ prime}\}$
4. $L = \left\{0^i 1^j | i \neq j, i \neq 2j\right\}$
5. $L = \left\{x^i y^j z^k | k = i + j\right\}$

If L is regular, we have pumping length p.

1. Consider 0^p1^p. Then
$$x = 0^k, y = 0^{p-k}.$$
However, if we pump 0 times, then we contradict $i \geq j$, so L is not regular.

2. Consider 0^p1^{2p}. Then
$$x = 0^k, y = 0^{p-k}$$
and $xyyz \notin L$.

3. Consider 1^k, where $k \geq p$ and k is prime. Let $y = 1$. Then the string with $2k$ 1s is not in L.

4. Consider something that'll pump from in between to $2j$.

5. Consider $a^pb^pc^{2p}$. Then $y = a$, and $xyyz \notin L$. Alternatively, consider $j = 0$. Then $L \cap a^*b^* = x^iy^{2i}$, which is not regular.

6 September 19, 2017

6.1 Context free languages

Recall our brief foray into context free grammars last week. We continue with a more formal treatment.

Definition 6.1. A **context free grammar** (CFG) G is a 4-tuple (V, Σ, R, S), where

- V is the set of variables,
- Σ is the set of terminals,
- R contains the rules from $V \to \Sigma$, and
- S is the start variable (root).

If $u, v \in (V \cup \Sigma)^*)$, then we say that $u \Rightarrow v$ if we can go from u to v with one substitution, Furthermore, $u \overset{*}{\Rightarrow} v$ if there exists some path from u to v, where u derives v. If $u = S$ is the start variable, then we say that u is a derivation of v.

We also introduce the notation that $A \to X|Y$ means $A \to X$ and $A \to Y$.

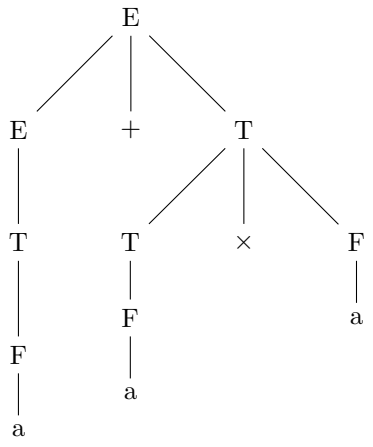
Example 6.2

Consider the following grammar G_2.

$$E \to E + T|T \quad T \to T \times F|F \quad F \to (E)|a$$

The set of terminals is $\{+, \times, (,), a\}$ and the variables are $\{E, T, F\}$.

This grammar looks a lot like arithmetic expressions.



An important thing to note is that grammars bring meaning in their structures. For example, the example arithmetic grammar builds in the dominance of multiplication over addition.

Often, there are multiple parse trees that arise from the same grammar. This fact is known as **ambiguity**.

Definition 6.3. If $A = L(G)$ for some context free grammar G, then A is a **context free language**,

$$L(G) = \left\{ w | S \overset{*}{\Rightarrow} w, w \in \Sigma^* \right\}.$$

6.2 Pushdown automata

Definition 6.4. A **pushdown automaton** (PDA) $B = \{Q, \Sigma, \Gamma, \delta, q_0, F\}$, where

- Q is the finite set of states,
- Σ is the finite input alphabet,
- Γ is the finite stack alphabet, which may be more permissive than Σ,
- δ is the (nondeterministic) transition function,

$$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$$

- q_0 is the start state, and
- $F \subseteq Q$ are the accept states.

Note 6.5. Our pushdown automata are always nondeterministic in this class. Deterministic and nondeterministic PDAs are *not* equivalent in power.

Note 6.6. Before reading any input symbols, the machine pushes the "end of stack" symbol. In this class, we can assume this happens automatically.

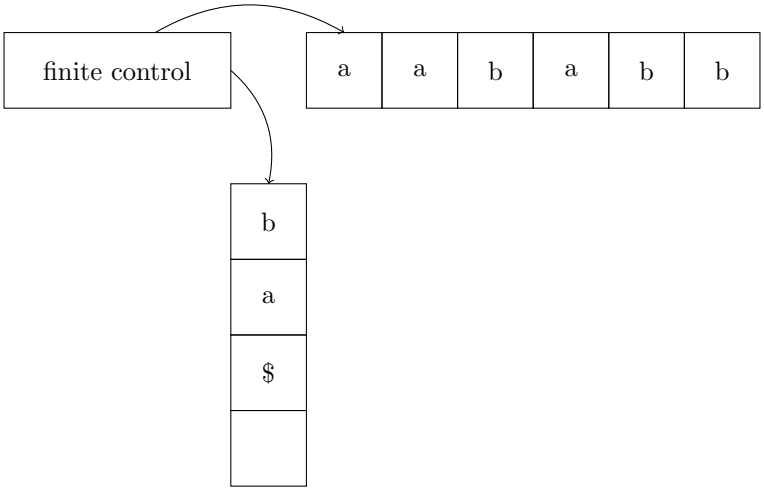


Figure 2: Pushdown automata that reads a's and b's.

We "push" to write a symbol on top of the stack and "pop" to read (and remove) from the top.

Example 6.7
Use pushover automata to recognize $\{a^k b^k | k \geq 0\}$.

1. Read a's and push them onto the stack, until a b is read.
2. For each b read, pop an a.
3. If the stack is empty at the end of input, we accept.

Note that if we encounter an a after a b, then the input is bad anyways.

Example 6.8
Use pushdown automata to recognize $\{ww^{\mathcal{R}} | w \in \{0,1\}^*\}$.

1. Read a symbol and push it onto the stack.
2. Repeat (1). Nondeterministically branch to (3).
3. Read symbol and compare with popped symbol.
4. Repeat until end of input.
5. Accept if stack empty.

This language *cannot* be recognized without nondeterminism.

6.3 Context free grammar to pushdown automata

Time to connect!

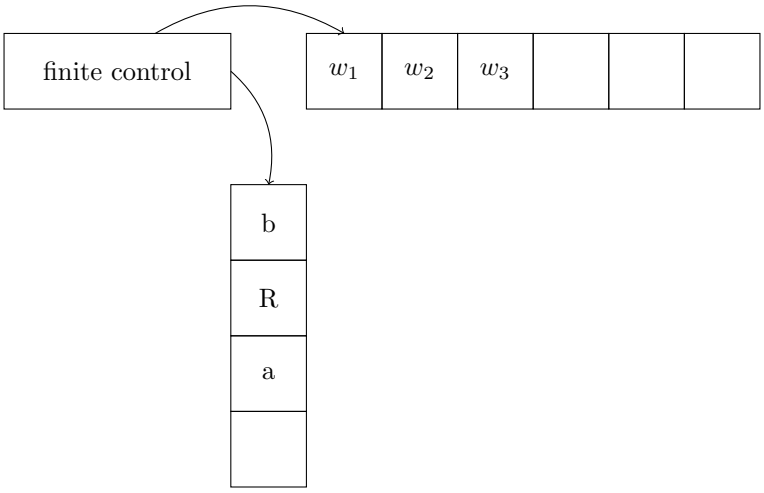
Theorem 6.9
Every context free grammar has an equivalent pushdown automaton.

The converse also holds, though the proof is slightly complicated.

Proof. We convert context free grammar G to pushdown automaton B. Suppose we receive input w.

1. We write the starting symbol S onto the stack.
2. If we have a variable on top, pop and replace.
3. If we have a terminal on top, check directly against w.
4. If there are many options, nondeterministically try them all.
5. Accept at end of input if stack is empty.

For example, suppose that $S \to aRb$. We substitute aRb into the stack.


finite control
w_1
w_2
w_3
b
R
a

□

7 September 21, 2017

7.1 Non context-free languages

We continue our discussion of context-free languages. Theorem 6.9 gives two corollaries.

Corollary 7.1

If A is regular, then A is a CFL.

Corollary 7.2

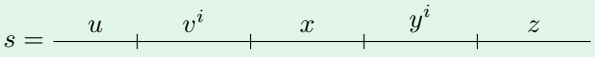
If A is regular and B is a CFL, then $A \cap B$ is a CFL.

CFLs are closed under $\cup, \circ, *$, but not under $\cap$.

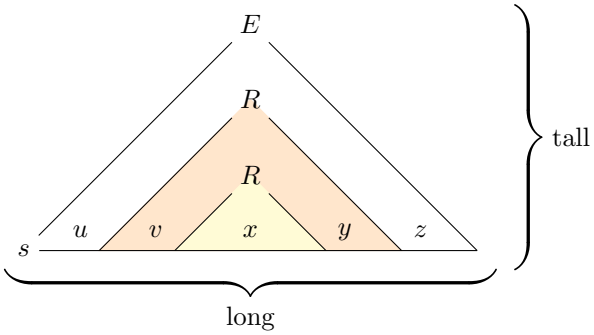
Lemma 7.3 (Pumping lemma for CFLs)

If A is a CFL, there is a pumping length p such that if $s \in A$ and $|s| \geq p$, we can split s into 5 pieces, $s = uvxyz$ where

1. $uv^ixy^iz \in A$ for $i \geq 0$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

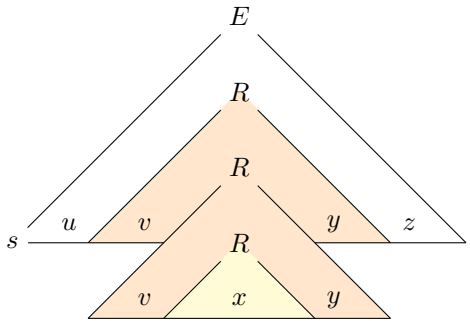


Proof sketch. Let $A = L(G)$, where G is a CFG. Select a long $s \in A$.



The general idea is that the parse tree must be really tall. If so, there must be a really long path, so some variable R must be repeated.

We can take a copy of the vxy-triangle and stick it in again.



We can similarly take out the x-triangle.

Let b be the length of the longest right-hand side of any rule in G (maximum branching factor). If the tree's height is h, then the length of s is at most b^h. Let $p = b^{|V|+1}$, where $|V|$ is the number of variables in G. If $|s| \geq p$, then $h \geq |V| + 1$.

□

Example 7.4

Show that $B = \{a^k b^k c^k | k \geq 0\}$ is a non context-free language.

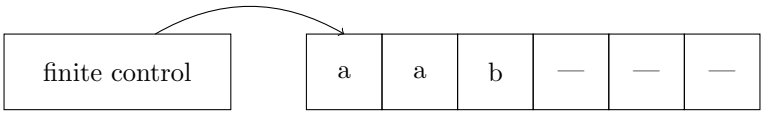
Proof. Assume for contradiction that B is a CFL. The pumping lemma gives pumping length p. Consider $s = a^p b^p c^p$.

$$s = \underbrace{a \ldots}_{} \underbrace{b \ldots}_{} \underbrace{c \ldots}_{}$$

If $s = uvxyz$, then $|xyz| \leq p$, so we can have at most two types of symbols. Therefore, if we pump v and y, we'll have unequal numbers of symbols. □

7.2 Turing machines

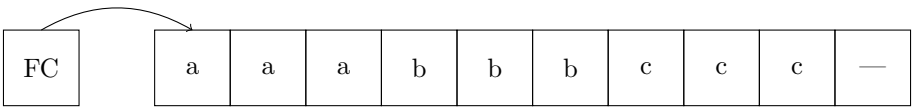
We arrive at the meat of this course. Past the appetizers and veggies.



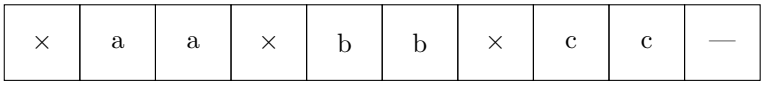
The finite control can read from and write to an infinite tape with two-way head motion. Beyond the input, the tape has infinite blank symbols —. The machine can accept anywhere by entering the unique q_{accept} state.

Example 7.5

Create a Turing machine that recognizes B from example 7.4.



1. Scan right until we encounter a blank symbol —. Check if input $\in a^*b^*c^*$, and reject if not.
2. Return to left end.
3. Scan right, crossing off an a, b, c.



4. Keep crossing off sets of a, b, c until we run out of some symbol, and accept if all symbols have been crossed off. Reject otherwise.

Definition 7.6. A **Turing machine** $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where

- Q are the states,
- Σ is the input alphabet,
- Γ is the tape alphabet ($\Sigma \subseteq \Gamma$),
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, and
- $q_0, q_{\text{accept}}, q_{\text{reject}}$) are the initial, accept, and reject states.

The transition function takes in a state and reads a symbol, to return a new state, a symbol to write, and either left or right motion. For example, $\delta(q_1, b) = (q_2, d, L)$ means to switch to state q_2, write a d, and move left.

Once a TM has started running, it can either accept, reject by halting, or reject by looping.

Definition 7.7. For a Turing machine M, the language of M is

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

- If $A = L(M)$, then M recognizes A, and A is **Turing-recognizable**.
- If M always halts on any input, then M is a **decider**.
- If $A = L(M)$ for a decider, then A is **decidable**.

8 September 22, 2017

8.1 CFL closure under reversal

To reverse a CFG, we can reverse the outputs of each variable.

Example 8.1

The production rule $S \to aTb$ reverses to $S^{\mathcal{R}} \to bT^{\mathcal{R}}a$.

8.2 Practice problems

We are "pumped" to do pumping lemma problems again...

Example 8.2

Prove that the following languages are not context-free.

1. $L = \left\{0^i 1^j 2^{\max(i,j)} | i, j \geq 0\right\}$
2. $L = \{ww | w \in \Sigma^*\}$

1. Consider the word $0^p 1^p 2^p$. Since the length of the center string $|vwy| \leq p$, v and y must contain two characters (1 character cannot be pumped up). If v and y contain 0 and 1, then 2 is no longer the maximum of them. If v and y contain 1 and 2, then we cannot pump down.
2. Consider the word $0^p 1^p 0^p 1^p$. Similar argument as the problem set.

Example 8.3

Consider a Turing machine that can only reset (cannot move arbitrarily left). Show that a Turing machine is equivalent to a reset Turing machine.

Proof. Normal TM simulates reset TM by moving all the way to the left. To simulate a normal TM with a reset TM,

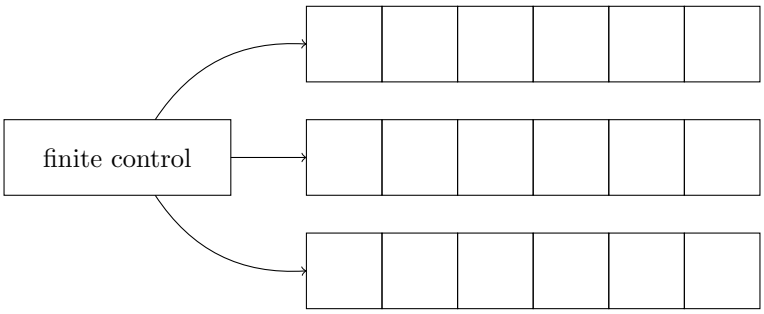
1. mark the original state and reset,
2. shift all symbols to the right by transitioning to the state and writing the previous state,
3. reset, and find the marked state.

□

9 September 26, 2017

9.1 Turing machine variants

We continue on our epic journey through Turing machine land. Today's first destination is the **multi-tape Turing machine**.



Theorem 9.1

A is Turing-recognizable if and only if $A = L(M)$ for some multi-tape Turing machine M.

Proof. Single-tape to multi-tape is trivial. One is a special case of many.

In the other direction, we want to convert multi-tape M into single-tape S. We concatenate the input strings in M into S and delimit the tapes with a special symbol #.

The transition function is

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

We remember the multiple heads with special symbols, so we can think of them as virtual heads.

$\dot{a}$	#	a	$\dot{b}$	$\cdots$	#	$\dot{b}$	a	#	—

We summarize below.

1. Format S's tape into k blocks.
2. For each step of M, scan across the tape to determine the virtual heads (symbols under dots).
3. Scan to update according to M's rules. If we run out of empty tape, shift right.
4. If M enters q_{accept} or q_{reject}, S follows suit.

□

Our next stop is the **non-deterministic Turing machine**. At any given point, there may be multiple possible transitions. If any state accepts, the machine accepts.

The transition function is

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

Theorem 9.2

A is Turing-recognizable if and only if $A = L(M)$ for some non-deterministic Turing machine M.

Proof. Deterministic to non-deterministic is again trivial.

We would like to convert non-deterministic Turing machine N into deterministic Turing machine M. For every input w, M keeps track of every single possible thread. If a thread splits, we make a copy at the end.

□

Our final stop is the **enumerator**, which is a Turing machine with a printer. The generator starts with no input and an empty roll of paper. We define the language of E as

$$L(E) = \{w | E \text{ prints } w, \text{ started on blank input}\}.$$

Theorem 9.3

A is Turing-recognizable if and only if $A = L(E)$ for some enumerator E.

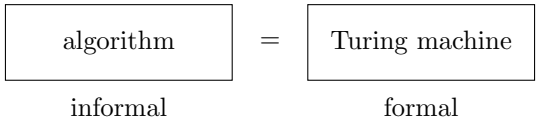
Proof. Given enumerator E, we first construct a recognizer M for $L(E)$. On input w, simulate E. If E prints w, accept. Otherwise, reject by looping (waiting forever on a hopeless printer).

In the other direction, we construct an enumerator E given M. Run M for k steps on $s_1, \ldots, s_k \in \Sigma^*$ for each k. Print s if M accepts s.

□

9.2 Church-Turing thesis

After seeing so many Turing machines, we find that actually, they're all equivalent in computing power! In modern day terms, we can see that different programming languages—Java, Python, Haskell—can all do what every other language can do. To convert between languages, we just write an interpreter.



We take a trip down memory avenue. In 1900, Hilbert published a list of 23 problems for the next century. Of those, we discuss the 10th:

> Is there an algorithm that can answer if a polynomial has integral solutions?

Initially, this question was fuzzy since we did not have a clear definition for what an "algorithm" could do. Later, people redefined an algorithm as solvable by a Turing machine. Let

$$D = \{p | p \text{ is a multivariable polynomial with integer solutions}\}.$$

In 1970, it was proven that there is no such algorithm—D is not Turing-decidable. However, D is Turing-recognizable.

10 September 28, 2017

10.1 Decision problems

For any object B (automaton, grammar, graph, string, polynomial, etc.), we denote B's binary string encoding as $\langle B \rangle$, and we use $\langle B_1, B_2, \dots \rangle$ to denote a set of reasonable encodings.

Theorem 10.1

Let $A_{\text{DFA}} = \{\langle B, w \rangle | B \text{ is a DFA that accepts } w\}$. A_{DFA} is decidable.

Proof. Take Turing machine decider M. On input s, Test if $s = \langle B, w \rangle$ for some B, w. Reject if not. Now simulate B on w. If B is in an accepting state at end of w, accept. Else reject. □

Theorem 10.2

Take Turing machine P. Let $A_{\text{NFA}} = \{\langle B, w \rangle | B \text{ is a NFA that accepts } w\}$. A_{NFA} is decidable.

Proof. On input $\langle B, w \rangle$, Convert NFA B to DFA D. Run M from above on input $\langle D, w \rangle$. Accept if M accepts, and reject otherwise. □

Theorem 10.3

Let $E_{\text{DFA}} = \{\langle B \rangle | B \text{ is a DFA}, L(B) = \emptyset\}$. E_{DFA} is decidable.

Proof. Take Turing machine T. On input $\langle B \rangle$, choose your favorite graph search algorithm and see if there's a path to any accept state.

Mark the start state q_0 and repeat until nothing new is marked. Mark every state with an arrow from a previously marked state. If any accept state is marked, reject. Accept if no accept states are marked. □

Theorem 10.4 (DFA equivalence)

Let $EQ_{\text{DFAs}} = \{\langle A, B \rangle | A, B \text{ are DFAs}, L(A) = L(B)\}$. EQ_{DFAs} is decidable.

Proof. Take Turing machine S. On input $\langle A, B \rangle$, let F be the symmetric difference of $L(A), L(B)$,

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

Construct DFA C, where $F = L(C)$. Run Turing machine T on C to test if $F = \emptyset$. Accept if T accepts, and reject otherwise. □

Theorem 10.5

Let $A_{\text{CFG}} = \{\langle G, w\rangle | \text{CFG } G \text{ generates } w\}$. A_{CFG} is decidable.

Proof. Take Turing machine H. On input $\langle G, w\rangle$, convert G to Chomsky normal form.[5] Try all derivations of length $2|w| - 1$. Accept if any string is equal to w, and reject otherwise. □

Corollary 10.6

Every CFG is decidable.

Proof. Let G be the grammar for CFL A. Take Turing machine M_G. On input w, run H on $\langle G, w\rangle$. Accept if H accepts, and reject otherwise. □

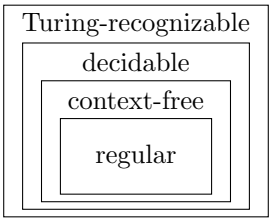


Figure 3: Hierarchy of languages

Theorem 10.7

Let $E_{\text{CFG}} = \{\langle G\rangle | L(G) = \emptyset\}$. E_{CFG} is decidable.

Proof. Take Turing machine J. On input $\langle G\rangle$, mark off all terminals. Mark variable T goes to uvw, which are all marked, for each variable T. Repeat until no more new variables. reject if the start variable is marked, and accept otherwise. □

Example 10.8

Let

$$S \to 01T0|RT01$$
$$T \to 0R|T0$$
$$R \to 0.$$

We mark R, then T, then S.

[5] All rules are of the form $A \to BC|a$, where $a \neq \epsilon$, except for the start state.

Theorem 10.9 (CFG equivalence)

Let $EQ_{\text{CFG}} = \{\langle G_1, G_2\rangle | G_1, G_2 \text{ are equivalent CFGs}\}$. EQ_{CFG} is *not* decidable or recognizable.

Theorem 10.10 (Universal Turing machine)

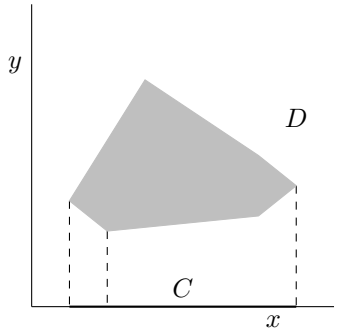
Let $A_{\text{TM}} = \{\langle M, w\rangle | M \text{ accepts } w\}$. A_{TM} is recognizable.

Proof. Take Turing machine U. On input $\langle M, w\rangle$, run M on w. Accept if M accepts, and reject otherwise. Therefore, A_{TM} is Turing-recognizable. □

10.1.1 Notes on problem set 2

Question 10.11. Let C be a language. Prove that C is Turing-recognizable iff a decidable language D exists such that

$$C = \{x | \exists y \in \{0,1\}^* \text{ where } (\langle x, y\rangle \in D)\}.$$



That is, every Turing-recognizable language

$D \to C$ is easy and $C \to D$ is hard. D is a collection of pairs of strings, and C is a projection of D.

$C = \{p | \text{polynomial } p \text{ has solution in integers.}\}$.

11 October 3, 2017

11.1 Turing machine decidability

Recall our hierarchy of languages from figure 3. Last lecture, we proved that A_{TM} is Turing-recognizable. Today, we prove that it is not decidable.[6]

Theorem 11.1

Let $A_{\text{TM}} = \{\langle M, w\rangle | M \text{ accepts } w\}$. A_{TM} is *not* decidable.

Proof. Assume for contradiction that A_{TM} is decidable by Turing machine H. On input $\langle M, w\rangle$, we accept if M accepts w and reject if M does not accept w (reject by halting or looping).

We use H to make Turing machine D. On input $\langle M\rangle$, simulate H on $\langle M, \langle M\rangle\rangle$, where the latter is a description of M.

"In fantasy land, this sounds like theoreticians gone wild"—sipser

Accept if H rejects, and reject otherwise. D on $\langle M\rangle$ accepts if and only if $\langle M\rangle$ on M doesn't accept. Now let $M = D$. D on $\langle D\rangle$ accepts if and only if $\langle D\rangle$ on D doesn't accept. A bunny was just born. □

We can visualize the proof as a diagonalization problem. Let 1 represent accept and 0, reject.

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	...
M_1	1	0	1	0	
M_2	1	1	1	1	
M_3	0	0	0	0	
$\vdots$					

D on $\langle M_i\rangle$ reverses the the effect of M_i on $\langle M_i\rangle$, so it reverses the diagonals of the list. However, if D is a Turing machine, it is also on this list, and it cannot flip itself.

Theorem 11.2

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

Proof. Let M_1, M_2 recognize $B, \overline{B}$ respectively. We construct decider M. On input w, run both machines in parallel, since $B \cup \overline{B} = \Sigma^*$. Accept if M_1 accepts and reject if M_2 accepts. □

[6] There is some proofy fluff in the next section, which may or may not be useful for you. You can also read that first for cuteness (literally!)

Theorem 11.3

$\overline{A_{TM}}$ is not Turing-recognizable.

Proof. This follows from theorems 11.1 and 11.2. □

Theorem 11.4

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

Proof. Assume for contradiction that Turing machine R decides HALT_{TM}. If M runs forever, we are done. Otherwise, run M on w and return the result. Then A_{TM} is decidable, which we know not to be true. Therefore, no such R exists. □

11.2 Diagonalization method

Today we introduce some math to prove more decidability theorems.

Definition 11.5. Two sets A, B have the same cardinality if there exists a bijection $f : A \to B$.

Definition 11.6. Set A is countable if $|A| = |\mathbb{N}|$.

Example 11.7

Let $\Sigma = \{0, 1\}$. Then $\Sigma^* = \{\epsilon, 0, 1, 00, 01, \ldots\}$ is countable.

Just count them. By corollary, all Turing machines are countable since we can encode any Turing machine M as a binary string $\langle M\rangle$.

Example 11.8

The set of rationals

$$\mathbb{Q} = \left\{ \frac{p}{q} \mid p, q \in \mathbb{N} \text{ and in lowest terms} \right\}$$

is countable.

Write the fractions in a table and count the diagonals, skipping seen values.

Theorem 11.9

$\mathbb{R}$ is not countable.

Proof. Suppose we could count them. Then sneaky pesky real numbers keep sneaking in like cockroaches. But I prefer bunnies.

There are too many bunnies and we can't count them all, because they keep having babies while we line them up. □

Now that we good on math, we can return to our Turing machines above.

12 October 5, 2017

Happy birthday bae!

12.1 Reducibility

Last class, we reduced the acceptance problem A_{TM} to the halting problem HALT_{TM}.

Definition 12.1 (Reducibility)**.** If A can be reduced to B, then a solution to B implies a solution to A.

Its contrapositive is also true: a lack of solution to A implies a lack of solution to B. In this class, we will usually use this latter fact.

Example 12.2

We can reduce the problem of earning a living to the problem of finding a job. If a person cannot earn a living, then we assume that he/she could not find a job.

Theorem 12.3

Let $E_{TM} = \{\langle M\rangle | M \text{ is a TM and } L(M) \neq \emptyset\}$. E_{TM} is undecidable.

Proof. Assume for contradiction that Turing machine R decides E_{TM}. We construct Turing machine S that decides A_{TM} using R. Create a modified Turing machine N_w which rejects all inputs $\neq w$ and runs as normal otherwise. Run R on N_w. If $L(N_w) = \emptyset$, then we reject. Otherwise, we accept, and A_{TM} is decidable. Therefore, E_{TM} cannot be decidable. □

Definition 12.4. A function $f : \Sigma^* \to \Sigma^*$ is computable if there is a Turing machine F where for all inputs w, F on w halts with $f(w)$ on the tape.

Definition 12.5 (Mapping reducibility)**.** For languages A, B, A is **mapping-reducible** to B if there exists a computable function $f : A \to B$. where for all w, $w \in A$ iff $f(w) \in B$. We use notation $A \leq_m B$.

A is no harder than B.

Theorem 12.6

If $A \leq_m B$ and B is decidable, then A is decidable.

Proof. Let R decide B. We construct S that decides A. On input w, compute $f(w)$ and run R on $f(w)$. Give the same answer as R. □

The same holds for Turing-recognizable (with essentially the same proof).

Theorem 12.7

$A_{TM} \leq_m \text{HALT}_{TM}$.

Proof. Let $f(\langle M, w\rangle) = \langle M', w\rangle$, which maps A_{TM} to HALT_{TM}. We construct M', where on input w, halt if M accepts w and loop if M halts and rejects w. □

Now we show that $\overline{A_{TM}} \leq_m E_{TM}$. We construct $f(\langle N, w\rangle) = \langle N_w\rangle$, where $\langle N, w\rangle$ decides if N accepts w and $\langle N_w\rangle$ decides if N_w is empty.

Since $\overline{A_{TM}}$ is not recognizable, then E_{TM} is also not recognizable.

Theorem 12.8

Let $EQ_{TM} = \{\langle M, N\rangle | M, N \text{ are TMs and } L(M) = L(N)\}$. Neither EQ_{TM} nor its complement are Turing-recognizable.

Proof. We reduce $\overline{A_{TM}}$ to both EQ_{TM} and $\overline{EQ_{TM}}$. Let $f : \langle M, w\rangle \to \langle M_1, M_2\rangle$.

First we show that $\overline{A_{TM}} \leq_m EQ_{TM}$.

- On input x, M_1 always rejects.
- On input x, M_2 rejects if $x \neq w$ and runs M on w otherwise.

Now we show that $\overline{A_{TM}} \leq_m \overline{EQ_{TM}}$.

- On input x, M_1 always accepts.
- On input x, M_2 accepts if $x \neq w$ and runs M on w otherwise.

□

"I shouldn't joke around when you're confused. It's not nice"—sipser

13 October 12, 2017

13.1 Quiz tips

We have a quiz in a week, so professor provides the following tips.

- If A is reducible to B, then

$$\begin{array}{rcl} B \text{ solvable} & \to & A \text{ solvable, and} \\ A \text{ unsolvable} & \to & B \text{ unsolvable.} \end{array}$$

- To show that B is undecidable, reduce A_{TM} to B. Use decider for B to build decider for A_{TM}.
- To show that B is not recognizable, show that $\overline{A_{\text{TM}}} \leq_m B$. Give a computable $f : \Sigma^* \to \Sigma^*$.
- A handy fact: $A \leq_m B$ if and only if $\overline{A} \leq_m \overline{B}$.

13.2 Computation history method

Definition 13.1. A **linearly bounded automaton** (LBA) is a Turing machine whose tape is the same size as the input.

Definition 13.2. A **configuration** of a Turing machine (or LBA) is a triple (q, p, t) of state, head position, and tape contents.

Theorem 13.3

Let $A_{\text{LBA}} = \{\langle B, w\rangle |$ LBA B accepts $w\}$. A_{LBA} is decidable.

Proof. The number of configurations for a tape of length n is

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

So on input $\langle B, w\rangle$, run B on w for N steps. If B has accepted, then accept. If B has rejected, then reject. Otherwise, B is still running, so reject. □

Definition 13.4. A **computation history** is a sequence of configurations.

Definition 13.5. An **accepting computation history** for Turing machine M on w is the sequence of configurations $c_1, c_2, \ldots, c_l$ that M goes through when accepting w.

Theorem 13.6

Let $E_{\text{LBA}} = \{\langle B\rangle | B$ is an LBA and $L(B) = \emptyset\}$. E_{LBA} is undecidable.

Proof. We reduce A_{TM} to E_{LBA}. Let R decide E_{LBA}. Then we can construct S deciding A_{TM}.

On input $\langle M, w\rangle$, we construct LBA $B_{\text{M,w}}$ which checks if its input x is an accepting computation history for M on w. If M accepts w, then $L(B)$ contains a single string, but if M does not accept w, then $L(B) = \emptyset$.

If R decides whether $L(B)$ is empty, then S can decide A_{TM}. □

13.3 Post-Correspondence problem

Suppose we have a set of dominoes. We have a "match" in an arrangement of dominoes if the top row of dominoes and the bottom row are the same.

$$\left\{ \begin{bmatrix} aa \\ aba \end{bmatrix}, \begin{bmatrix} ab \\ aba \end{bmatrix}, \begin{bmatrix} ba \\ aa \end{bmatrix}, \begin{bmatrix} abab \\ b \end{bmatrix} \right\}$$

For example, we can manually make a match.

$$\left\{ \begin{bmatrix} ab \\ aba \end{bmatrix} \begin{bmatrix} aa \\ aba \end{bmatrix} \begin{bmatrix} ba \\ aa \end{bmatrix} \begin{bmatrix} aa \\ aba \end{bmatrix} \begin{bmatrix} abab \\ b \end{bmatrix} \right\}$$

Formally, suppose we have a set of dominoes

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

Given the sequence $i_1, i_2, \dots, i_l \in P$ of dominoes, we have a match if

$$u_1 u_2 \dots u_l = v_1 v_2 \dots v_l.$$

Definition 13.7. We define the PCP language as

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

Theorem 13.8

PCP is undecidable.

Proof. Reduce A_{TM} to PCP using the computation history method. Assume for contradiction that Turing machine R decides PCP. Then we can construct S deciding A_{TM}.

On input $\langle M, w \rangle$, construct a PCP instance $P_{M,w}$, whose match must start with

$$\begin{bmatrix} u_1 \\ v_1 \end{bmatrix} = \begin{bmatrix} \# \\ \#q_0 w_1 w_2 \dots w_n \# \end{bmatrix}.$$

For each tape symbol $a \in \Gamma$, we add domino

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

If $\delta(q, a) = (r, b, R)$—state r, writes b, moves right—then we add

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

□

14 October 17, 2017

14.1 Quiz tips, ctd.

The professor provides several examples of the computation history method.

1. Build LBA $B_{M,w}$ which accepts only accepting computation history for M on w.
2. Build PCP problem $P_{M,w}$ where a match is the only accepting computation history.
3. To show
$$D = \{\langle p \rangle \mid p \text{ has integer solutions}\}$$
is undecidable, build polynomial (for example)
$$p_{M,w}(x_1, x_2, \ldots, x_9) = 5x_1^2 - 22x_1x_2^7x_3 + \cdots + 425 = 0$$
where the only solution x_1 is an accepting computation history.

14.2 Undecidability, ctd.

We present one last undecidability proof.

Theorem 14.1

Let
$$\text{ALL}_{\text{PDA}} = \{\langle D \rangle \mid D \text{ is a PDA and } L(D) = \Sigma^*.\}$$
ALL_{PDA} is undecidable.

Proof. We reduce A_{TM} to ALL_{PDA} by the computation history method. For contradiction, assume that Turing machine R decides ALL_{PDA}. We build Turing machine S that decides A_{TM}.

On input $\langle M, w \rangle$,

1. Build PDA $D_{M,w}$ which accepts all $x \neq$ accepting computation history for M on w.
2. At each step in the computation history, accept if illegal move.
3. Run R on $\langle D_{M,w}$. If R says that $L(D_{M,w}) = \Sigma^*$, then reject (no accepting computation history). Otherwise accept.

□

14.3 Recursion

There is a philosophical question: can you make something that is more complicated than yourself?

Computationally, yes.

Theorem 14.2 (Recursion theorem)

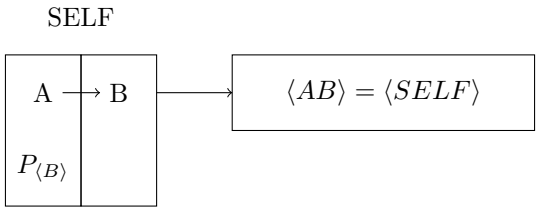
There is a Turing machine "SELF" which prints (leaves on the tape) a description of "SELF" on any input.

Lemma 14.3

There is a computable function $q : \Sigma^* \to \Sigma^*$ where $\forall w, q(w) = \langle P_w \rangle$ where P_w prints w.

Proof of lemma. The function simply says "print w." □

Proof of recursion theorem. Now we construct "SELF."



A should be $P_{\langle B \rangle}$, which leaves $\langle B \rangle$ on the tape. B computes q of the tape contents $\langle B \rangle$, which is $\langle P_{\langle B \rangle} \rangle = \langle A \rangle$. We print these before tape contents. □

> "You think this is all theory, but there is at least one application in the real world—maybe even two!"—sipser

Example 14.4

We implement recursion in the fancy programming language, English.

> Print out two copies of the following, 2nd one in quotes.
>
> "Print out two copies of the following, 2nd one in quotes."

Theorem 14.5

For any Turing machine T, there is a Turing machine R where for all w, $R(w) = T(\langle R, w \rangle)$.

In English, mumble jumble.

We can use "get own description" when writing Turing machines.

The real world application? Computer viruses.

> "Self-documenting code"—txz

We provide a new proof for A_{TM} is undecidable. Assume that H decides A_{TM}. We build Turing machine R, which on w:

1. Retrieve its own description $\langle R \rangle$.
2. Run H on $\langle R, w \rangle$.
3. Accept if H says R doesn't accept w, and reject if R accepts w.

Theorem 14.6 (Fixed point theorem)
Given computable $f : \Sigma^* \to \Sigma^*$, there is some Turing machine R where for all w, R accepts w iff $f(\langle R \rangle)$ accepts w.

Proof. We construct Turing machine R, which on input w,

1. Retrieve its own description $\langle R \rangle$.
2. Compute $\langle S \rangle = f(\langle R \rangle)$.
3. Simulate S.

□

Now we can show that $\text{ALL}_{\text{TM}} \not\leq_m \overline{\text{ALL}_{\text{TM}}}$.

15 October 19, 2017

15.1 Complexity

Today, computability is largely considered a solved problem, with little active research.

However, besides what is *computable*, we are also interested in *how hard* a problem may be to solve. **Complexity** is the study of how difficult problems are. There is much active research in complexity, and the second half of this course will focus on these topics.

We introduce asymptotic notation as a way to evaluate the difficulty of solving problems.

Definition 15.1. We use Big-O notation to denote

$$O(f(x)) = cf(x)$$

for some fixed constant c.

Theorem 15.2

Let

$$A = \left\{a^k b^k | k \geq 0\right\}.$$

A is decidable by a 1-tape Turing machine M that uses at most cn^2 steps for all inputs of length n for some fixed constant c.

Proof. We build Turing machine M. On input w,

1. Scan w to determine if $w \in a^*b^*$.
2. Repeat until done:
   (a) Scan and cross off a single a and a single b per round.
   (b) If one symbol finishes first, reject.
3. Accept.

Step 1 takes $O(n)$ steps. Step 2 takes $O(n)$ steps for $O(n)$ iterations, so the entire algorithm completes in $O(n^2)$. □

But we can do better! If we cross off every other character and keep track of parity, then we obtain an $O(n \log n)$ solution.

With two tapes, however, we *can* attain $O(n)$. Just cross off both tapes at the same time.

This should worry us. It would be a shame if all of complexity theory depended on which type of Turing machine we chose. Fortunately, we can bound this dependency, so we continue our analysis with a 1-tape Turing machine.

Definition 15.3. A Turing machine M runs in time $t(n)$, where $t : \mathbb{N} \to \mathbb{N}$, if for all w of length n, M on w halts within $t(n)$ steps.

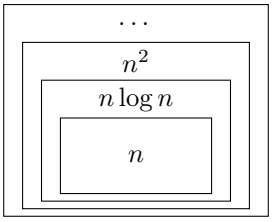


Figure 4: Hierarchy of $TIME$ problems.

Definition 15.4. We define a new problem

$$TIME(t(n)) = \{A \mid A \text{ is decidable by a TM in } O(t(n))\}.$$

Note 15.5. We can only attain $O(n)$ on regular languages.

Theorem 15.6

If a fixed-tape multi-tape Turing machine decides B in time $t(n)$, then $B \in TIME(t^2(n))$ on a 1-tape Turing machine.

Proof. We have $O(t(n))$ copies of the original tape, each of length $t(n)$, so we take $t(n)^2$ steps. □

In fact, any conversion between two reasonable deterministic machines is polynomial in t.

Proposition 15.7 (Polynomial equivalence)

Converting between any two reasonable deterministic models can only change the running time from $t(n)$ to $t^k(n)$.

Definition 15.8 (Class P)**.** We define class P as

$$P = \cup_k TIME(n^k) = TIME(\text{polynomial in } n).$$

There are several implications.

- Class P is invariant for reasonable deterministic models.
- Polynomial time vaguely corresponds to "practically" solvable.

Example 15.9

Let

$$\text{PATH} = \{\langle G, s, t\rangle | \exists s \rightsquigarrow t \in G\}$$

where G is a directed graph.

Let $n = |\langle G, s, t\rangle|$. On $\langle G, s, t\rangle$:

1. Mark s.
2. For each marked node, scan G and mark its neighbors.
3. Repeat until no new nodes marked.
4. Accept if t marked. Reject otherwise.

Example 15.10 (Hamiltonian path)

Let

$$\text{HamPath} = \{\langle G, s, t\rangle \mid \exists \text{ Hamiltonian path } s \rightsquigarrow t \in G\}$$

where G is a directed graph.

It is unknown whether we can find a Hamiltonian path in P.

16 October 24, 2017

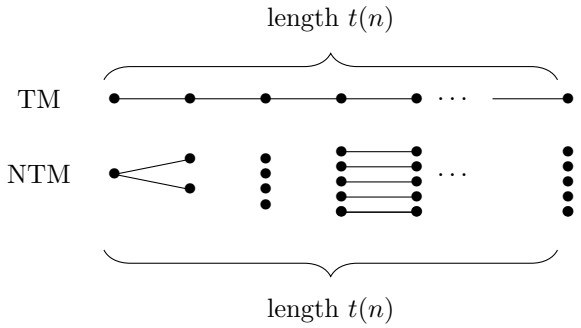
16.1 Nondeterministic complexity

Problems like Hamiltonian path can be verified in polynomial time, but we cannot check if a graph *does not* have a Hamiltonian path.

Definition 16.1. A nondeterministic Turing machine runs in time $t(n)$ if for all inputs of length n, the Turing machine uses at most $t(n)$ steps on every thread.

Accordingly, $NTIME$ is defined as

$$NTIME = \{A \mid A \text{ is decidable by a NTM in } O(t(n))\}.$$



Definition 16.2 (Class NP)**.** Class NP is

$$NP = \cup_k NTIME(n^k)$$

or "nondeterministic polynomial time."

> "You will make me very, *very* upset if you call it 'not-polynomial time.' —sipser

Theorem 16.3
Hamiltonian path is in NP.

Proof. We give a nondeterministic decider for Hamiltonian path. On input $\langle G, s, t\rangle$, we nondeterministically write a sequence of nodes $v_1, v_2, \ldots, v_m$ of nodes, where m is the number of nodes. Check that

- start and ends are correct: $s = v_1, t = v_m$,
- every edge is valid: $v_i \to v_{i+1}$ for all i,
- and there are no repeated v_i.

Accept if all true, and reject otherwise. □

Theorem 16.4 (Composite numbers)

Let the composite numbers be the set

$$C = \{x | x \in \mathbb{N}; x = yz, y, z \in \mathbb{N}_{\geq 1}\}.$$

C is in NP.

Proof. On input x, nondeterministically guess $y \leq x$. Test if $y > 1, y < x$, and if y evenly divides x. Accept if so, and reject otherwise. □

In lay terms, P is the class of languages whose membership can be "tested" quickly, while NP is the class of languages that can be "verified" quickly.

And... we come to the wonderful world of certificates. We can verify NPness with a short certificate of membership.

16.2 P vs. NP

Our professor has an entire folder dedicated to wacky P vs. NP proof correspondences.

Theorem 16.5

If A is a context-free language, then A is NP.

Proof. It is easy to show that A is in NP. We convert A to Chomsky normal form, so that all derivations are length $2k - 1$.

We construct a NTM for A. On input w, nondeterministically guess a derivation of length $2|w| - 1$ for w. Accept if valid, reject otherwise.

Now we show that A is in P. We use dynamic programming. Suppose start state $S \to TU$. Then we verify that T goes to the left half of w, and U goes to the right half. Consider all substrings from $w_i \dots w_j$. Then we just fill in the table with the derivations of each substring.

□

> "If you're not here, come back—come back!!!"—sipser
>
> "Now we're going to add in the secret ingredient to dynamic programming: don't be stupid."—sipser

Example 16.6 (SAT)

A boolean formula is **satisfiable** if there exists an assignment of variables that renders the formula true.

$$SAT = \{\phi | \phi \text{ is a boolean formula that is satisfiable}\}$$

SAT is in NP.

Proof. Just guess the assignment. □

17 October 31, 2017

17.1 Polynomial time reducibility

Definition 17.1. A is polynomial time reducible to B ($A \leq_p B$) if A is mapping reducible to B ($A \leq_m B$) and the reduction is computable by a Turing machine that runs in polynomial time.

Theorem 17.2

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

This is the same proof as decidable.

Proof. Suppose Turing machine decides B in polynomial time. We construct Turing machine S that decides A in polynomial time. On input w, compute $f(w)$ and test if $f(w) \in B$ with R. Return the same result. □

Now, we can show that every language $A \in NP$ is polynomial time reducible to SAT.

Definition 17.3 (Conjunctive normal form)**.** A literal is x or $\overline{x}$ and a clause is an `OR` of literals. A formula is in **conjunctive normal form** (cnf) if it is an `AND` of clauses.

For example,

$$\phi = (a \vee b \vee \bar{c} \vee \bar{d}) \wedge (\bar{b} \vee c) \wedge \cdots \wedge (t \vee o \vee \bar{n} \vee y)$$

is in conjunctive normal form. To satisfy a cnf formula, at least one literal from each clause must be true. Every boolean formula can be converted to a cnf formula, but not necessarily in polynomial time.

Example 17.4 (3-SAT)

A k-cnf formula is a cnf formula in which every clause has exactly k literals. 3-SAT is the problem

$$\{\langle \phi \rangle \mid \phi \text{ is satisfiable 3-cnf formula}\}.$$

For example, $(a \vee b \vee c) \wedge (a \vee \bar{b} \vee d)$ is a 3-cnf formula.

Example 17.5 (k-clique)

A k-clique can be defined as

$$\{\langle G, k \rangle \mid G \text{ is undirected graph with } k \text{ fully connected nodes}\}.$$

The k-clique problem is in NP—we just provide the clique.

Theorem 17.6

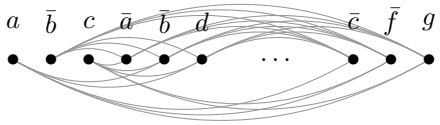
If clique is in P, then 3-SAT is in P.

Proof. We show that 3-SAT $\leq_p$ clique.

We convert 3-SAT problems to clique problems. Suppose we have formula

$$(a \vee \bar{b} \vee c) \wedge (\bar{a} \vee \bar{b} \vee d \wedge \cdots \wedge (\bar{c} \vee \bar{f} \vee g).$$

We create a node for each literal.



We draw edges according to the following rules.

1. We do not connect two literals in the same clause.
2. We do not connect any literal with its complement.
3. We draw all other edges.

We have constructed G, where k is the number of clauses.

If the formula is satisfiable, then G has a k-clique. Suppose we have a satisfying assignment. Then we select the satisfied literal from each clause. These literals form a k clique because complements cannot be connected, and each literal comes from a single clause.

In the other direction, we could not have picked 2 literals from the same clause, because they are not connected. Furthermore, there are exactly k literals from k clauses. Finally, literals selected must be true, and there are no contradictions. □

Theorem 17.7

B is NP-complete if

1. $B \in$ NP, and
2. $\forall A \in$ NP, $A \leq_p B$.

Theorem 17.8

3-SAT is NP complete.

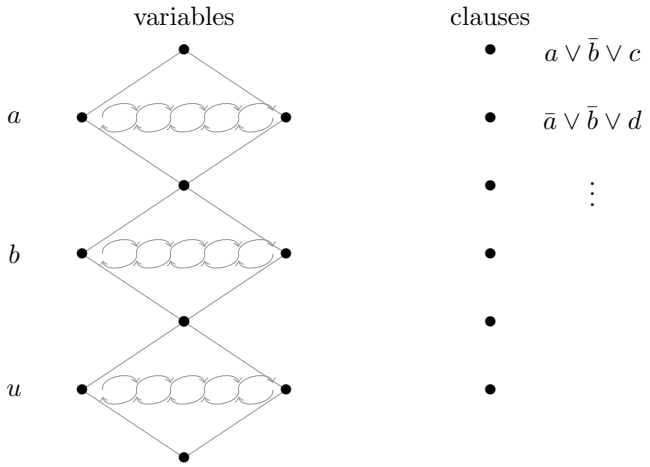
Theorem 17.9

3-SAT is reducible to Hamiltonian path.

Proof. Consider the same boolean formula from above,

$$(a \vee \bar{b} \vee c) \wedge (\bar{a} \vee \bar{b} \vee d \wedge \cdots \wedge (\bar{c} \vee \bar{f} \vee g).$$

We construct Hamiltonian path gadgets. For each variable, we draw these diamond paths.



Note the following.

1. Each clause must be visited at least once, but needs not be visited for each node.
2. Each variable is either true or false. If the variable is true, then it must be true for all the clauses in which it is selected as the token literal. Therefore, the path must be moving in the same direction (same truth value).
3. If any clause is not selected for a variable, we just take the swirls to the next part.

□

18 November 2, 2017

We begin by going over a problem in the problem set.

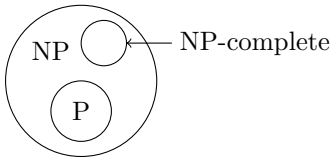
Example 18.1
Prove that $\{a, b, c, p \mid a^b \equiv c \mod p\} \in P$.

The key is to keep taking numbers modulo p.

> "Two words—dynamic programming."—sipser, about problem 2, problem set 4.

18.1 NP-completeness

Imagine you worked super hard to prove whether $P \stackrel{?}{=} NP$, and then you realize that there are harder problems in NP. You'd be really sad. But not if you chose an NP-complete problem.

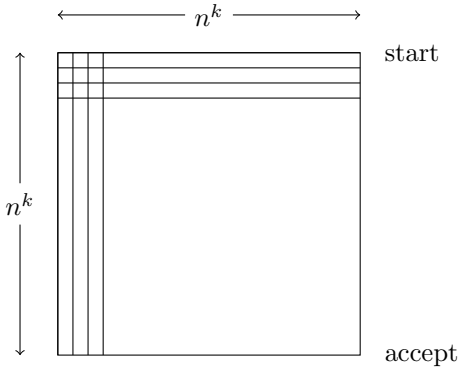


Theorem 18.2 (Cook-Levin theorem)
If $SAT \in P$, then $NP = P$. That is, SAT is NP-complete.

Proof. It is clear that $SAT \in NP$. We guess the assignment.

Now let $A \in NP$ be decided by NTM M in n^k time. We show that $A \leq_p SAT$ by giving reduction f where $f(w) = \phi_w, w \in A \iff \phi_w \in SAT$.

We create a n^k by n^k tableau for M on w, where each row is a configuration for M. Without loss of generality, if the machine accepts before n^k steps, it stays at the same state.



In other words, this is a computation history for one thread. Within n^k steps, the machine can only accept n^k of the tape, so this tableau contains the possible configurations.

ϕ_w "says" that M accepts w, so ϕ_w is the logical representation that M accepts w. Equivalently, a tableau exists for M on w.

Let

$$\phi_w = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}.$$

In each cell, we have boolean variables $x_{ij\sigma}$ where $i, j \in \{1, \ldots, n^k\}$ and $\sigma = Q \cup \Gamma$. For example, $x_{3,4,a} = 1$ means that row 3, column 4 has the symbol a.

ϕ_{cell} says that each cell has at most 1 symbol.

$$\phi_{\text{cell}} = \bigwedge_{i,j \in \{1,\ldots,n^k\}} \left[\underbrace{\left(\bigvee_{\sigma \in Q \cup \Gamma} x_{ij\sigma} \right)}_{\text{at least one symbol}} \wedge \underbrace{\left(\bigwedge_{\sigma,\tau \in Q \cup \Gamma, \sigma \neq \tau} \overline{x_{ij\sigma}} \vee \overline{x_{ij\tau}} \right)}_{\text{at most one symbol}} \right]$$

Each cell has at least one symbol selected, and between every pair of cells at the same i, j, only one symbol is selected.

Furthermore, ϕ_{start} says that the top row is the start configuration.

$$\phi_{\text{start}} = x_{1,1,q_0} \wedge x_{1,2,w_1} \wedge x_{1,3,w_2} \ldots$$

Likewise, ϕ_{accept} says that the bottom row has an accept state somewhere.

$$\phi_{\text{accept}} = \bigvee_{j \in \{1,\ldots,n^k\}} x_{n^k j q_{\text{acc}}}$$

Finally, q_{move} says that each row follows from the previous. Consider each 2 by 3 "neighborhood." We can check whether these transitions are legal, and we can enumerate all legal neighborhoods.

$$\bigwedge_{i,j \in \{1,\ldots,n^k\}} \underbrace{\bigvee_{abcdef} x_{i-1,j,a} x_{i,j,b} x_{i+1,j,c} x_{i-1,j+1,d} x_{i,j+1,e} x_{i+1,j+1,f}}_{i,j \text{ neighborhood is valid}}$$

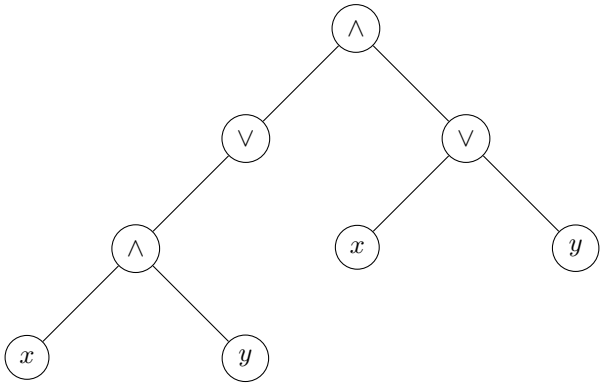
where a, b, c, d, e, f come from legal neighborhoods.

a	b	c
d	e	f

The entire proof works in size $O(n^{2k})$.

□

Suppose $\phi = ((x \wedge y) \vee z) \wedge (x \vee y)$. We want to convert $\phi \to \psi$, which is an equivalent 3-cnf.



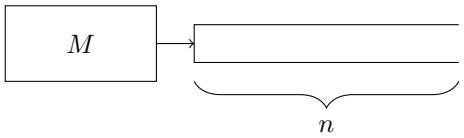
Let $b_4 = x \wedge y$.

x	y	b_4
0	0	0
0	1	0
1	0	0
1	1	1

19 November 7, 2017

19.1 Space complexity

Definition 19.1. Turing machine M in $f(n)$ space ($f : \mathbb{N} \to \mathbb{N}$) if M is a decider and M on w uses at most $f(n)$ tape cells on all inputs of length n.



We can define two new problems,

$$SPACE(f(n)) = \{A \mid \text{some TM } M \text{ decides } A \text{ with } O(f(n)) \text{ space}\}$$
$$NSPACE(f(n)) = \{A \mid \text{some NTM } M \text{ decides } A \text{ with } O(f(n)) \text{ space}\}$$

Correspondingly, we can define two new classes of problems.

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

Theorem 19.2

We assume that $f(n) \geq n$.

1. $TIME(f(n)) \subseteq SPACE(f(n))$
2. $SPACE(f(n)) \subseteq TIME(2^{O(f(n))}) = \bigcup_k TIME(c^{f(n)})$

Proof. The proofs are simple.

1. In $f(n)$ steps, we can reach at most $f(n)$ space.
2. With $f(n)$ cells, there are at most $2^{O(f(n))}$ configurations (before we start looping, and these are deciders). Therefore, we take at most $2^{O(f(n))}$ steps.

□

It directly follows from (1) that $P \subseteq PSPACE$.

Theorem 19.3

$NP \subseteq PSPACE$.

Proof. We begin with two observations.

First, $SAT \in PSPACE$. Given a formula ϕ, we cycle through assignments on the tape and verify. Therefore, $SAT \in SPACE(n)$.

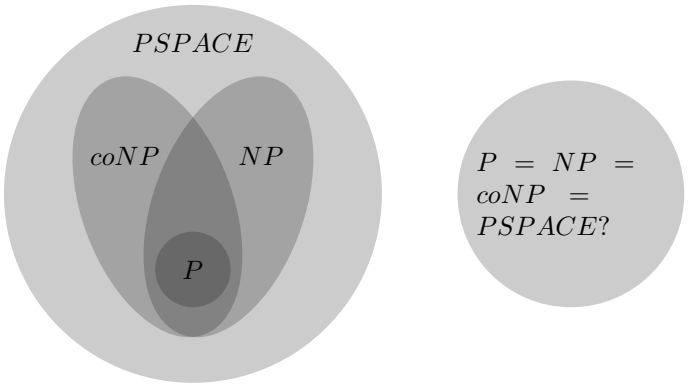
Second, if $A \leq_p B$ and $B \in PSPACE$, then $A \in PSPACE$. The polynomial time reduction converts from B to A.

We know that $SAT \in PSPACE$. For all $A \in NP$, $SAT \leq_p A$, so $A \in PSPACE$. □

Definition 19.4. *coNP* is the set of complements of NP problems,

$$coNP = \left\{ A \mid \overline{A} \in NP \right\}.$$

We claim that $coNP \in PSPACE$. We think our world view looks like the following, though everything would collapse down to one space if $P = PSPACE$.



Example 19.5 (Quantified boolean formulas)

Each variable in the boolean formula is quantified with either ∃ or ∀. For example,

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

The former is `TRUE` and the latter is `FALSE`.

This is a generalized form of SAT, which only has ∃ quantifiers on each variable.

Theorem 19.6

Let

$$TQBF = \{\langle \phi \rangle \mid \phi \text{ is a } \texttt{TRUE} \text{ QBF}\}.$$

$TQBF \in PSPACE$.

We recursively set $x = 0, x = 1$ and figure out the rest. This takes polynomial space.

Example 19.7

We can play word ladders!

```
WORK
PORK
PORT
SORT
SOOT
SLOT
SLAT
SLAY
PLAY
```

English is ill-defined mathematically, so we consider a DFA instead.

Theorem 19.8

Let

$$LADDER_{DFA} = \{\langle B, w, x\rangle \mid B \text{ is DFA}, \exists y_1, \ldots, y_k \in L(B)\}$$

where $y_1 = w$, $y_k = x$, and each y_i, y_{i+1} differs in 1 place.

We claim that $LADDER_{DFA} \in NPSPACE$.

Nondeterministically, we write out each word we're looking at, make a transition, and forget the old word. After a certain number of steps (amount of combinations possible), we shut down a thread if it hasn't found the target.

Deterministically, we solve this problem recursively. For each recursion, we halve the possible ladder height. We try every possible legal word for the "middle" word. The base case is a ladder of 1 step, which is easy to check.

On $\langle B, w, x, t\rangle$,

1. If $t = 1$, then check if w, x differ in 1 place.
2. Otherwise, try all possible y,

This takes n^2 memory.

20 November 9, 2017

20.1 Recursive PSPACE proofs

Recall that

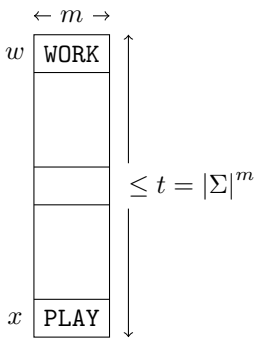
$$LADDER_{DFA} = \{\langle B, w, x\rangle \mid B \text{ is a DFA, exists ladder in } B \text{ from } w \text{ to } x\}.$$

We can show that $LADDER_{DFA} \in PSPACE$.

Proof. Let the bounded ladder problem be

$$BL = \{\langle B, w, x, t\rangle \mid \text{ladder of length } t\}.$$

1. If $t = 1$, test whether $w, x \in L(B)$ and w, x differ at at most 1 symbol.
2. If $t > 1$, for each y of length $|w|$, recursively run BL on $\langle B, w, y, t/2\rangle$ and $\langle B, y, x, t/2\rangle$. Accept if both accept for some y.
3. Reject if no accepting y is found.



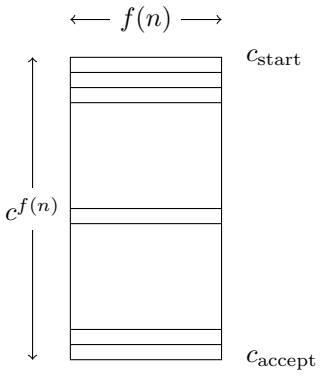
We analyze the space complexity. Each recursive level stores y in $O(n)$ space. There are $\log_2 t$ levels, so the total space is $O(n \log t)$.
To solve the original $LADDER$ problem on $\langle B, w, x\rangle$, we run BL on $\langle B, w, x, t\rangle$, where $t = d^n, d = |\Sigma|$.

□

Theorem 20.1 (Savitch's theorem)
For $f(n) \geq n$, $NSPACE(t(m)) \leq SPACE(f(m)^2)$.

This theorem implies that $PSPACE = NPSPACE$.

Proof. We convert NTM N to Turing machine M. Consider the tableau for N on w. We assume that the machine cleans up and parks its head at the left, so there is only one accepting configuration.



For configurations c_i, c_j, we write that $c_i \xrightarrow{k} c_j$ if c_i can yield c_j in at most k steps.
N accepts w if $c_{\text{start}} \xrightarrow{t} c_{\text{accept}}$ where $t = c^{f(n)}$.
Recursively test if $c_i \xrightarrow{k} c_j$ for each configuration on $f(n)$ space. For each c_{mid}, test if $c_i \xrightarrow{k/2} c_{\text{mid}}$ and $c_{\text{mid}} \xrightarrow{k/2} c_j$.
The base case is $k = 1$, at which point we check of $c_i \to c_j$ via N's rules.
There are $\log c^{f(n)} = O(f(n))$ levels, each of which is $f(n)$.

□

Definition 20.2. Language B is $PSPACE$-complete if

1. $B \in PSPACE$ and
2. for every $A \in PSPACE$, $A \leq_p B$.

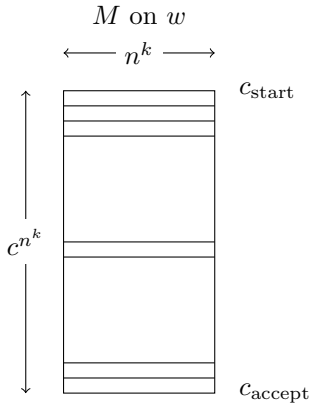
Theorem 20.3
$TQBF$ is $PSPACE$-complete.

Proof. We've already shown that $TQBF \in PSPACE$.

Let $A \in PSPACE$ be decided by Turing machine M in space n^k. We show that $A \leq_p TQBF$.

We give a reduction $f : w \to \phi_w$ where $w \in A$ if and only if ϕ_w is true. That is, ϕ_w "says" M accepts w.

Let's draw another tableau.



For configurations c_i, c_j, we construct $\phi_{c_i,c_j,k}$ which "says" that $c_i \xrightarrow{k} c_j$,

$$\phi_{c_i,c_j,k} = \exists c_{\text{mid}}[\phi_{c_i,c_{\text{mid}},k/2} \wedge \phi_{c_{\text{mid}},c_j,k/2}]$$

when $k > 1$. We directly check configurations when we reach $\phi_{c_i,c_j,1}$.

Unfortunately, this doesn't work. There are $O(n^k)$ levels, but at each level, we double the number of formulas. Fortunately, we're very close!

$$\phi_{c_i,c_j,k} = \exists c_{\text{mid}} \forall (c_\alpha, c_\beta) \in \{(c_i, c_{\text{mid}}), (c_{\text{mid}}, c_j)\} [\phi_{c_\alpha,c_\beta}].$$

Or equivalently,

$$\forall c_\alpha, c_\beta[((c_\alpha, c_\beta) = (c_i, c_{\min})) \vee ((c_\alpha, c_\beta) = (c_{\min}, c_j))].$$

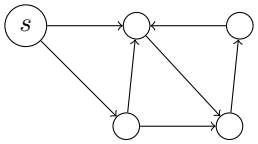
□

21 November 14, 2017

21.1 PSPACE-complete games

Consider the "geography" game. We name off countries, where you must name a country starting with the opponent's country's last letter, no repeats. Eventually, we'll start looping.

But under optimal conditions, who has an advantage? This question is $PSPACE$-complete (unproven, but can be shown by induction).



A **winning strategy** is a forced win under optimal play.

Theorem 21.1

Let $GG = \{\langle G, s\rangle \mid \text{player 1 has a winning strategy in geography game.}\}$. GG is $PSPACE$-complete.

Proof. We can show that $GG \in PSPACE$ by induction.

Now we show that $TQBF \leq_p GG$. Let's play a formula game. Suppose we have a TQBF

$$\phi = \exists x_1 \forall x_2 \exists x_3 [(x_1 \vee x_2) \wedge (\overline{x_2} \vee x_3) \wedge \dots]$$

where the inside is a cnf formula. There are two players: $\exists$ and $\forall$. Each player gets to pick an assignment for his corresponding variables. Player $\exists$ wants the formula to be true, while $\forall$ wants the formula to be false.

If ϕ is true as a formula, then $\exists$ has a winning strategy, and if ϕ is false, then $\forall$ has a winning strategy.

$\exists$ has a winning strategy iff $\phi \in TQBF$.

□

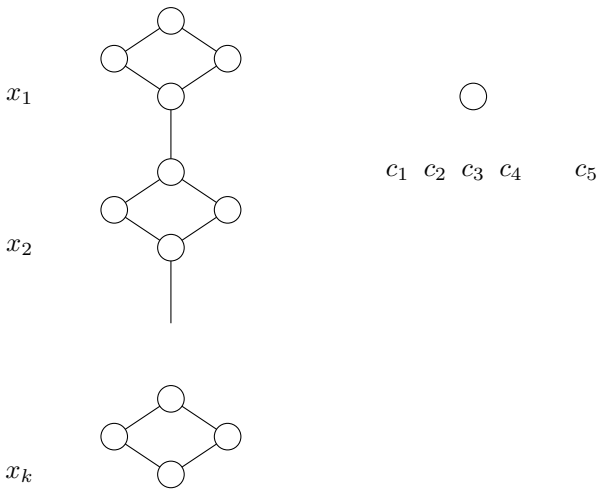
Example 21.2

Consider the TQBF

$$\exists x_1 \forall x_2 \exists x_3 \dots \forall x_k [(x_1 \vee \bar{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \overline{x_4} \wedge \dots \wedge ()].$$

We construct a generalized geography game.

Without loss of generality, assume that there are alternating $\exists$ and $\forall$ quantifiers. (We can add in dummy variables.)

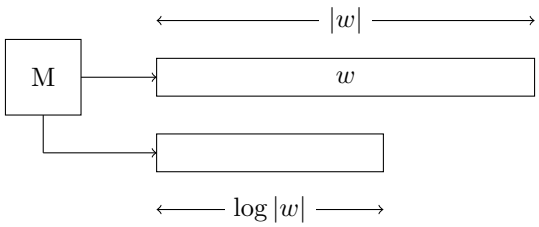


We assign left to true and right to false. At each turn, one player will take a left or right on a diamond and assign a variable.

If $\forall$ wins, some clause was not satisfied, so $\forall$ can pick the unsatisfied clause. Then $\exists$ must select the offending literal (if $\forall$ lied). If $\exists$ is correct, then he can keep playing (nodes have not been visited). Otherwise, $\exists$ is stuck.

21.2 Logarithmic space

We shift to a 2-tape model. There is a read-only input tape of length n and a working tape of size $O(\log n)$.



Definition 21.3. Let L be the class of $SPACE(\log n)$ and let NL be $NSPACE(\log n)$.

Logarithmic space is exactly enough space to write a pointer into the input.

Example 21.4

The language of palindromes

$$\left\{ww^{\mathcal{R}} \mid w \in \Sigma^*\right\} \in L.$$

The path problem

$$\{\text{PATH} = \{\langle G, s, t\rangle | \exists s \rightsquigarrow t \in G\}\} \in NL$$

For palindromes, we can check symbol for symbol, at both ends. For path, we nondeterministically guess paths and keep a count of how many steps we've taken.

"If we're nondeterministic, we don't even need friends!"—sipser

Proposition 21.5

Not worth being a theorem. $L \subseteq P$.

Proof. Let a configuration of M on w be a tuple (q, p_1, p_2, t), where q is the state, p_1, p_2 are the head locations, and t is the work tape. There are

$$|Q| \cdot n \cdot O(\log n) \cdot c^{O(\log n)}$$

configurations possible.

In a L-space machine M, we can't run for more than a polynomial number of steps before repeating configurations. □

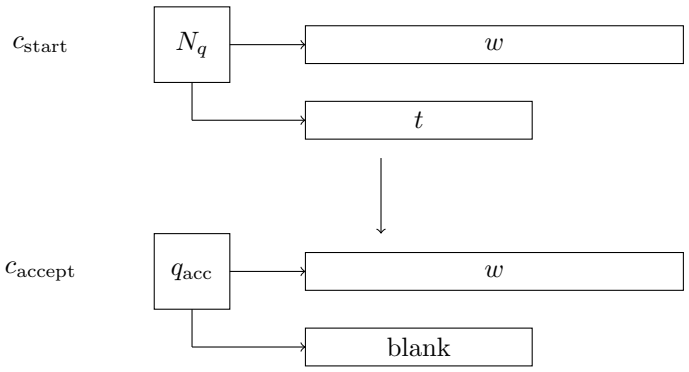
Proposition 21.6

$NL \subseteq P$.

Proof. Let $A \in NL$ be decided by NTM M in $O(\log n)$ space. We convert NTM N (log space) to TM M (polynomial space).

For M on w:

1. Construct the **computation graph** for N on w, where the nodes are configurations of N on w and the edges are valid configuration transitions. (We add edge $c_i \to c_j$ if c_i yields c_j.)
2. Test if there exists a path from the start configuration to an accepting configuration.
3. Accept if yes, reject if no.



□

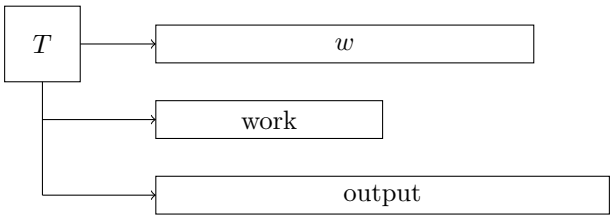
22 November 16, 2017

22.1 NL-completeness

Definition 22.1. B is **NL-complete** if

1. $B \in NL$ and
2. $\forall A \in NL, A \leq_L B$, where $\leq_L$ stands for log-space reduction.

Definition 22.2. A **log-space transducer** is a 3-tape Turing machine with a read-only input tape (length n), a read/write work tape ($\log n$), and a write-only output tape (unrestricted length).



Definition 22.3. We say that $A \leq_L B$ if $A \leq_m B$ and the reduction is log-space computable by a log-space transducer.

Theorem 22.4

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

Proof. For A on w, compute $f(w)$ and run the decider for B on $f(w)$.

We might not have space to store $f(w)$, so we store instead a pointer into $f(w)$. For each bit of $f(w)$ we wish to ingest, restart the transducer and obtain the bit in question.

□

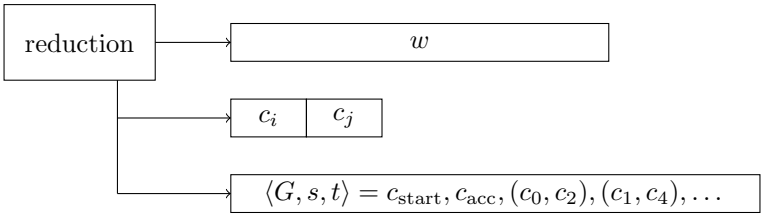
> "You have to recompute the bit multiple times... I think I'm recomputing the proof multiple times at this point."—sipser

Theorem 22.5

PATH is NL-complete.

Proof. PATH is clearly in NL.

For all $A \in NL$, we show that $A \leq_L$ PATH. Suppose NTM N decides A. Then we define $f : w \to \langle G, s, t \rangle$, where $w \in A$ if and only if there exists a $s \rightsquigarrow t$ path. G is a computation graph for N on w.



We go through every pair of configurations (c_i, c_j) and print out all valid configurations. The configurations are log space because $A \in L$.

□

22.2 NL = coNL

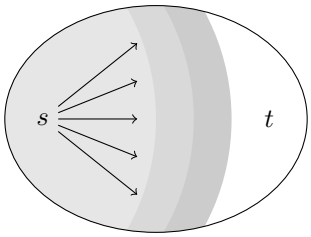
The $\mathsf{NP} \stackrel{?}{=} \mathsf{coNP}$ question is an open problem. However, there *is* an interesting result for NL.

Theorem 22.6

Let $\mathsf{coNL} = \left\{\overline{A} \mid A \in \mathsf{NL}\right\}$. $\mathsf{NL} = \mathsf{coNL}$.

Proof. We show that $\overline{\mathsf{PATH}} \in \mathsf{NL}$.

Let R_i be the nodes reachable from s within i steps, and let $c_i = |R_i|$.



On $\langle G, s, t, c\rangle$:

1. For every node v, nondeterministically pick a path $s \rightsquigarrow v$ or declare that v is unreachable (each thread assigns reachable or not).
2. For every v that is reachable, there will be some thread that finds the right path for everything.
3. Check if the count of reachable is equal to c. Accept if yes, and t is unreachable.

We can test if $t \in R_i$, given some c_i. Starting a 0, we look at all nodes reachable within 0, 1, ... steps, each step building off the previous, until we reach $|V|$ steps. This is essentially BFS. □

23 November 17, 2017

23.1 Practice problems

We reviewed the past few lectures and do practice problems.

Example 23.1

A directed graph G is **strongly connected** if $\forall u, v \in V, \exists (u, v), (v, u) \in E$. Let

$$\mathsf{SC} = \{\langle G \rangle \mid G \text{ is strongly connected}\}.$$

Show that SC is NL-complete.

Solution. $\mathsf{SC} \in \mathsf{NL}$ since we can non-deterministically select vertices u, v and check if there exists $(u, v), (v, u) \in E$.

Now we give a reduction from PATH. On input $\langle G, s, t \rangle$:

1. For all nodes $v \in V$, add edge $v \to s$ and edge $t \to v$ in G'.
2. If G' is strongly connected, then for every u, v, there exists a path from $u \rightsquigarrow v$ via $u \to s \to t \to v$. Thus, there must exist a $s \rightsquigarrow t$ path in G.

In the reverse direction, if G' is strongly-connected, then any edges in $G' \setminus G$ cannot be used in a $s \rightsquigarrow t$ path. Therefore, there must exist a path from $s \rightsquigarrow t$ in G.

Example 23.2

Show that $\mathsf{BIPARTITE} \in \mathsf{NL}$.

Solution. A graph is bipartite if there are no odd cycles. Since $\mathsf{NL} = \mathsf{coNL}$, we show that $\mathsf{BIPARTITE} \in \mathsf{coNL}$. Nondeterministically select a vertex $v \in V$ and select a path from v. Keep track of the path length until $|V|$. If we ever return to v and the length is odd, accept.

24 November 28, 2017

24.1 Hierarchy theorem

The professor mentions that the hierarchy theorem is a proof by diagonalization.

There are some solvable problems that are not decidable in P, but may be decidable, given more time.

24.2 Exponential space

Today we show that there is a *natural* problem we can solve in exponential space, but not with polynomial space. That is, we *might* care about it; it's not too contrived.

Example 24.1

Let $EQ_{REX} = \{\langle R_1, R_2\rangle \mid L(R_1) = L(R_2)\}$ be the equality problem for regular expressions.

This language is in PSPACE

Proof. First convert R_1, R_2 to NFAs B_1, B_2 (linear space increase). Since PSPACE = NPSPACE, we can use the corresponding NFAs. Now we guess the w where B_1, B_2 disagree. □

We can make the problem a bit harder. Let

$$R^k = \underbrace{R \circ R \circ \cdots \circ R}_{k}$$

be an "enhanced" regular expression, where k is written in binary.

Theorem 24.2

Let $EQ_{REX\uparrow} = \{\langle R_1, R_2\rangle \mid L(R_1) = L(R_2)\}$ be the equality problem for enhanced regular expressions. $EQ_{REX\uparrow} \notin PSPACE$.

Definition 24.3. EXPSPACE$= \bigcup_k$ SPACE(2^{n^k}).

Definition 24.4. EXPTIME$= \bigcup_k$ TIME(2^{n^k}).

Definition 24.5. B is EXPSPACE-complete if

1. $B \in$ EXPSPACE and
2. $\forall A \in$ EXPSPACE, $A \leq_p B$.

The hierarchy theorem shows that

$$\text{PSPACE} \subsetneq \text{EXPSPACE}$$
$$\text{P} \subsetneq \text{EXPTIME}.$$

"If you got lost, I haven't done anything, so feel free to ask a question about nothing."—sipser

Theorem 24.6

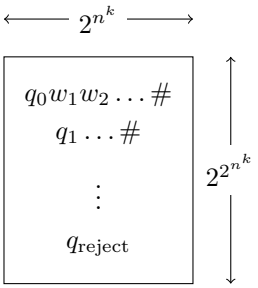
$EQ_{REX\uparrow}$ is exponential-space complete.

Proof. $EQ_{REX\uparrow} \in$ EXPSPACE since it is in PSPACE.

Now let $A \in$ EXPSPACE be decided by Turing machine M in space 2^{n^k}. We give a reduction $f : w \to \langle R_1, R_2 \rangle$ such that $w \in A$ if and only if $L(R_1) = L(R_2)$.

Let $L(R_1)$ be all strings Δ^*, and let $L(R_2)$ be all strings except for a rejecting computation history for M on w.

The tape is 2^{n_k} long, so the computation history is *really* big.



$$R_2 = R_{\text{bad,start}} \cup R_{\text{bad,move}} \cup R_{\text{bad,reject}}$$

If M accepts w, there is no rejecting computation history, so $L(R_2) = \Delta^* = L(R_1)$.

We need a smol regular rexpression, so let's begin!

$$R_{\text{bad,start}} = s_0 \cup s_1 \cup s_2 \cdots \cup s_n \cup s_{\text{blanks}} \cup s_{\#}$$

where each s_i describe all strings that mess up at location i. For example, $s_0 = \Delta - q_0\Delta^*$, or all strings that don't start at q_0, and $s_1 = \Delta - \Delta w_1 \Delta^*$, etc. The general term is $s_n = \Delta^n(\Delta - w_n)\Delta^*$.

Unfortunately, there are too many. Fortunately, we write them all as

$$\Delta^{n+1}(\Delta \cup \epsilon)^{2^{n^k}-(n+1)}(\Delta - w)\Delta^*$$

We use a similar expression for all strings that end wrong.

Now we weed out the bad moves (e.g. Tony dancing). Recall the 3 by 2 window we used for the Cook-Levin construction.

$$\Delta^* \bigcup_{\text{illegal windows}} [abc\Delta^{2^{n^k}-3}def]$$

where $2^{n^k} - 3$ is the distance between c and d in the computation history as a string.

□

25 November 30, 2017

25.1 Oracles

Definition 25.1 (Oracles)**.** For any language A, a Turing machine M with an **oracle** for A (M^A) can answer "is x in A" for free.

So

$$P^A = \{B \mid B \text{ solvable in polynomial time by some } M^A\}$$

Since SAT is NP-complete,

$$\mathsf{NP} \subseteq \mathsf{P}^{\mathsf{SAT}} \qquad \mathsf{coNP} \subseteq \mathsf{P}^{\mathsf{SAT}}.$$

Theorem 25.2

For $A = \mathsf{TQBF}$, $\mathsf{NP}^{\mathsf{A}} = \mathsf{P}^{\mathsf{A}}$

Proof. TQBF is PSPACE-complete, so

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

Since TQBF is solvable in PSPACE, every time we call the oracle, we can also just solve it ourselves. □

Suppose we provide a proof where A simulates B, and we give both machines an oracle C. The proof still works the same, since A can call C on its own; it does not depend on the fact that B has an oracle. This is known as **relativization**.

Prof. Sipser's interpretation of oracle relativation is that we cannot prove $\mathsf{P} = \mathsf{NP}$ by mere simulation.

25.2 Probabilistic complexity

Often, practical algorithms do not require an exact answer; they can produce effective approximations with much less effort.

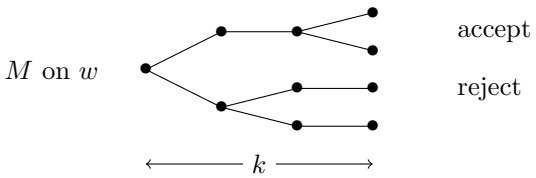
Example 25.3

Suppose we want to know, "who will be elected the next President of the United States?" We could make everyone vote, but we could also poll a sample. There are potential issues:

- We could be wrong! The sample could be bad.
- We could be systematically choosing more from one group than the other group.

But that's okay! Error is inherent to these models.

Definition 25.4. A **probabilistic Turing machine** M is a NTM where at every point, there are either 1 or 2 successors (next steps).



For thread b, let $\Pr\{b\} = 2^{-k}$ where k is the number of coin tosses on b. Then

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

For language A, M decides A with error probability ϵ for $0 \leq \epsilon \leq 1/2$ if

- for $w \in A$, $\Pr\{M \text{ accepts } w\} \geq 1 - \epsilon$, and
- for $w \notin A$, $\Pr\{M \text{ rejects } w\} \geq 1 - \epsilon$.

Definition 25.5 (Class BPP)**.** We define class BPP as

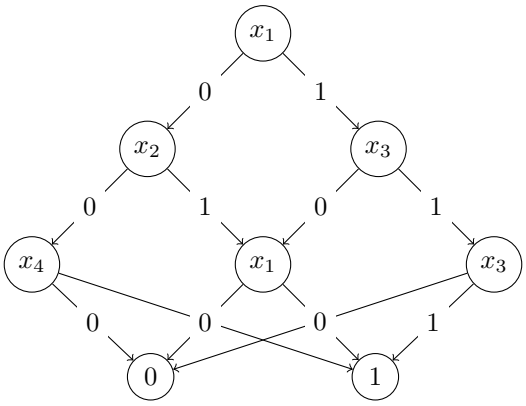
$$\mathsf{BPP} = \{A \mid \text{ some probabilistic TM decides } A \text{ with } \epsilon = 1/3\}.$$

Lemma 25.6 (Amplification lemma)

If a probabilistic polynomial-time Turing machine M decides A with error probability $\epsilon < 1/2$, there is some probabilistic polynomial-time Turing machine N deciding A with error probability δ for any $\delta > 0$.

25.3 Branching programs

Definition 25.7 (Branching program)**.** On some input (e.g. $\langle x_1, x_2, x_3, \dots \rangle = \langle 0, 1, 1, \dots \rangle$), **branching program** P is a decision tree computes a function $f : \{0,1\}^n \to \{0,1\}$.



Theorem 25.8

Let

$$\mathsf{EQ}_{\mathsf{BP}} = \{\langle B_1, B_2 \rangle \mid B_1, B_2 \text{ compute same language}\}$$

where B_1, B_2 are branching programs. $\mathsf{EQ}_{\mathsf{BP}} \in \mathsf{coNP}$.

Definition 25.9. In a **read-once branching program**, every path queries a variable at most once (note above that x_1 and x_3 appear twice).

The equivalence problem for read-once branching programs is in BPP.

Theorem 25.10

Let

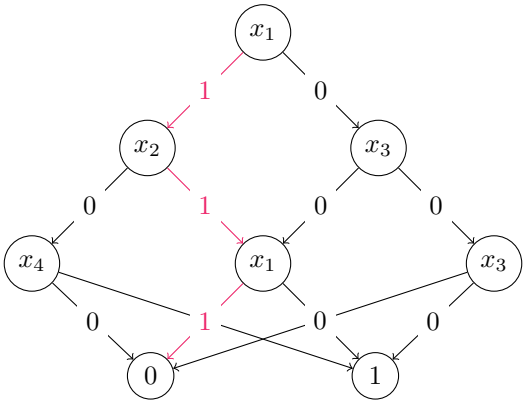
$$\mathsf{EQ}_{\mathsf{ROBP}} = \{\langle B_1, B_2\rangle \mid B_1, B_2 \text{ compute same language}\}$$

where B_1, B_2 are *read-once* branching programs. $\mathsf{EQ}_{\mathsf{ROBP}} \in \mathsf{BPP}$.

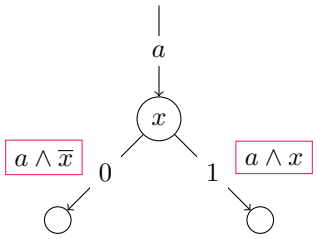
Proof. Suppose we have two ROBPs B_1, B_2.

We feed in an input and check the output.

Find a path from x_1 to an output and mark that path: assign every edge on the path to 1 and all other edges to 0



Now suppose we arrive at the following intersection.

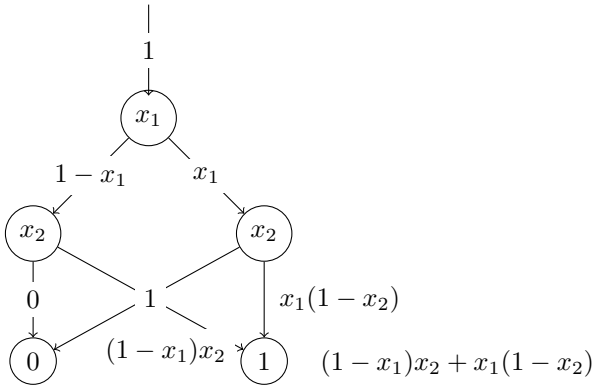


We take the left path if $a \wedge \overline{x}$ and the right path if $a \wedge x$.

We can convert boolean expressions to arithmetic expressions.

$$a \wedge b \to ab \qquad \overline{a} \to 1 - a \qquad a \vee b \to (a + b) - ab$$

Consider the following branching program, which computes XOR.



The formula at the 1 output is the function that this branching program evaluates! But now we are not limited to assigning boolean values to the variables. We can just as easily say $x_1 = 2, x_2 = 3$ and compute that "$2 \oplus 3 = 7$."

□

26 December 1, 2017

26.1 Review

Definition 26.1 (Class RP)**.** For language A, we say that A is in class RP if

1. for $w \in A$, $\Pr\{M \text{ accepts } w\} \geq 2/3$, and
2. for $w \notin A$, $\Pr\{M \text{ rejects } w\} = 0$.

Example 26.2

Here are some open problems.

- $\mathsf{P}^{\mathsf{SAT}} \stackrel{?}{=} \mathsf{NP}^{\mathsf{SAT}}$
- $\mathsf{NP}^{\mathsf{SAT}} \stackrel{?}{=} \mathsf{coNP}^{\mathsf{SAT}}$
- $\mathsf{P} \stackrel{?}{=} \mathsf{BPP}$

Example 26.3

$\mathsf{BPP} \subseteq \mathsf{PSPACE}$.

Proof. We simply run every possible path and keep track of how many accept. There are at most 2^k paths of length k, so we require $\log 2^k = k$ bits to keep track of which path we're on. Accept if and only if $\geq 2/3$ of the paths accept. □

Example 26.4

$\mathsf{MIN\text{-}FORMULA} \in \mathsf{coNP}^{\mathsf{SAT}}$.

Proof. We can decide $\overline{\mathsf{MIN\text{-}FORMULA}}$ as follows. On input ϕ:

1. Non-deterministically select a formula ψ shorter than ϕ with the same variables.
2. Use the SAT oracle to determine if $\psi \equiv \phi$. Accept if so. Reject otherwise.

□

Example 26.5

$\mathsf{RP} \in \mathsf{NP}$.

Proof. Certificate is any accepting path, since for $w \notin A$, we will never accept. □

Example 26.6

If $\mathsf{SAT} \in \mathsf{BPP}$, then $\mathsf{SAT} \in \mathsf{RP}$.

Proof. Let M be the BPP solver for SAT. We construct the RP solver R. On input ϕ:

1. Run M on ϕ.
2. If M rejects, reject. If M accepts, try an assignment for each x_i, a la problem 3 from the pset.
3. Check the assignment at the end.

□

27 December 5, 2017

27.1 Branching programs is BPP

We continue the proof from the last lecture.

Here are two cute theorems.

Theorem 27.1

Let $p(x) = a_1x^d + a_2x^{d-1} + \cdots + a_dx + a_{d+1}$. If non-zero $p(x)$ has degree d, then p has at most d roots.

Proof by easy induction.

Theorem 27.2

Given p_1, p_2 of degree d, then p_1 and p_2 agree on at most d places.

Let $p = p_1 - p_2$. p has at most d roots, so they can agree at most d places.

We do arithmetic in some finite field $\mathbb{F}_q$.

Theorem 27.3

For $p \neq 0$ of degree d and a randomly selected $x \in \mathbb{F}_q$,

$$\Pr\{p(x) = 0\} \leq \frac{d}{q}.$$

Theorem 27.4 (Schwartz-Zippel lemma)

Let $p(x_1, x_2, \ldots, x_m)$ be a polynomial of degree d in each x_i. For random values $(x_1, \ldots, x_m) \in \mathbb{F}_q^m$,

$$\Pr\{p(x_1, x_2, \ldots, x_m) = 0\} \leq \frac{md}{q}.$$

Another easy proof by induction.

The intuition is that if p is not zero everywhere, then it is zero rarely.

We return to branching programs. We convert ROBP B_1, B_2 into equivalent polynomials p_1, p_2 that simulate the branching programs.

1. If $B_1 \equiv B_2$, then $p_1(x) = p_2(x)$ for all x.
2. If $B_1 \not\equiv B_2$, then $\Pr\{p_1(\vec{x}) = p_2(\vec{x})\}$ for random $\vec{x} \in \mathbb{F}_q^m$ is small.

Each polynomial p is a sum of products of every x_i term, positive or negated. For example,

$$p = x_1(1 - x_2)(1 - x_3)\ldots(1 - x_m) + (1 - x_1)x_2x_3\ldots x_m + \ldots$$

Not every variable need appear, but no variable appears more than once in each clause. Here, $d = 1$ and m is the number of variables. So let $q \geq 3m$ (for proving the BPP bound). Then

$$\Pr\{p(x_1, x_2, \ldots, x_m) = 0\} \leq \frac{md}{q} = \frac{m}{q} \leq \frac{1}{3}.$$

28 December 7, 2017

28.1 Interactive proofs method

We introduce the **interactive proofs method** to prove an interesting result about graph isomorphism. We have a **verifier** V that runs in probabilistic polynomial time and a **prover** P that is computationally unlimited.

Both P and V see input w. They exchange n^k messages, and then V ouputs `accept` or `reject`.

Definition 28.1. $\Pr\{V \leftrightarrow P \text{ accepts}\}$ is the probability that V accepts w while interacting with P.

Definition 28.2 (Class IP)**.** $A \in \mathsf{IP}$ if there is a V and P where for every w:

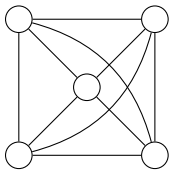
- If $w \in A$, then $\Pr\{V \leftrightarrow P \text{ accepts } w\} \geq 2/3$
- If $w \notin A$, then for every $\tilde{P}$. $\Pr\left\{V \leftrightarrow \tilde{P} \text{ accepts } w\right\} \leq 1/3$.

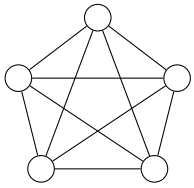
The former P is known as an **honest prover**, while the latter is known as the **crooked prover** (it cannot accept no matter how hard it tries).

> "I'm old. I'm just a bounded probabilistic time. But you're my research group, and you're young! You're unlimited computationally—you're unconstrained in the amount of effort you put in. You can spend all night on the problem."—sipser

28.2 Graph isomorphism

Given graphs A, B, are they isomorphic?





Let ISO be the problem

$$\mathsf{ISO} = \{\langle A, B\rangle \mid A = B\}.$$

ISO is in NP, but we don't know if it is NP-complete. We can provide a mapping as a certificate.

It is also unknown if $\overline{\mathsf{ISO}} \stackrel{?}{=} \mathsf{NP}$.

A recent result is that $\mathsf{ISO} \in \mathsf{TIME}(n^{\log^k n})$, which is quasi-polynomial time.

Theorem 28.3
$\overline{\mathsf{ISO}} \in \mathsf{IP}$.

Proof. We give a protocol for $\overline{\mathsf{ISO}}$. On input $\langle A, B\rangle$:

1. V: pick A or B at random and randomly permute. Send result C to prover (repeat twice).
2. P: determine if $C \equiv A$ or $C \equiv B$ and report back to the verifier.
3. V: accept if both correct. Reject otherwise.

If $A \not\equiv B$, then $\Pr\{V \leftrightarrow P\} = 1$, since the honest prover will always obtain the right answer. On the other hand, if $A \equiv B$, then for every $\tilde{P}$, $\Pr\{V \leftrightarrow P\} \leq 1/4$, within the bounds. □

28.3 IP = PSPACE

Suppose we were to play a game of chess, and wanted to determine if white has a winning strategy. For us, we could only walk down the computation tree...

Theorem 28.4
IP = PSPACE.

...But if we had a computationally unlimited prover, then we could determine the answer without walking the exponentially-large tree. That's cool!

We prove a slightly weaker result, since it's easier.

Proposition 28.5
coNP ⊆ coNP.

Definition 28.6. #-SAT $= \{\langle \phi, k \rangle \mid \phi$ has exactly k satisfying assignments$\}$.

#-SAT is coNP-hard since $\overline{\text{SAT}} \leq_p$ #-SAT (test if there are 0 satisfying assignments $\phi \to \langle \phi, 0 \rangle$).

Given $\phi(x_1, \ldots, x_m)$, we define

$$T(a_1, \ldots, a_i) = \sum_{a_{i+1}, \ldots, a_m \in \{0,1\}} \phi(a_1, \ldots, a_m).$$

We preset the first i values and determine the number of satisfying assignments, given the existing values. Note the following properties.

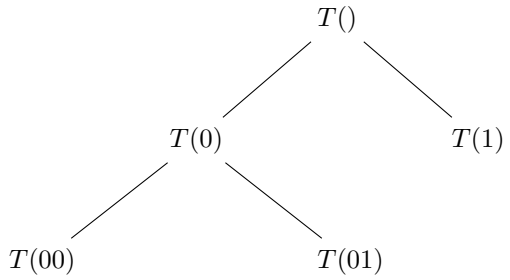
- $T()$ with no presets is equal to the number of satisfying assignments.
- $T(a_1, \ldots, a_m) = \phi(a_1, \ldots, a_m)$.
- $T(a_1, \ldots, a_i) = T(a_1, \ldots, a_i, 0) + T(a_1, \ldots, a_i, 1)$.

Now we can provide the protocol for #-SAT. On input $\langle \phi, k \rangle$:

1. P: send $T()$. V: verify that $k = T()$.
2. P: send $T(0), T(1)$. V: verify that $T() = T(0) + T(1)$.
3. P: send $T(00), T(01), T(10), T(11)$. V: verify that $T(0) = T(00) + T(01)$, etc., and so on.

   ...

4. P: send $T(a_1, \ldots, a_m)$ for each $a_1, \ldots, a_m \in \{0,1\}^m$. V: verify.
5. V: verify that $T(a_1, \ldots, a_m) = \phi(a_1, \ldots, a_m)$ for each $a_1, \ldots, a_m \in \{0,1\}^m$.



$\vdots$

$T(0, \ldots, 0)$

$\phi(0, \ldots, 0)$

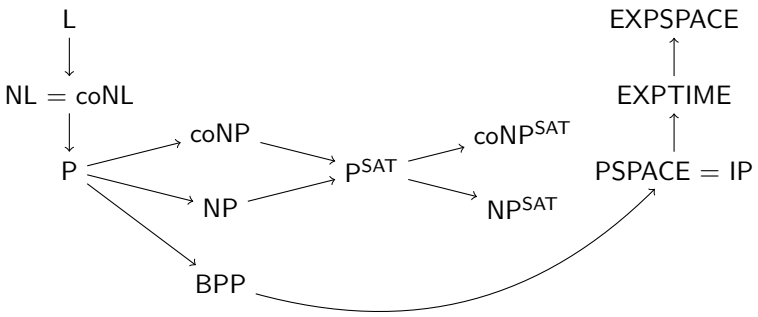
A lie at the root propagates down all the way to the leaves, which the verifier can catch on its own.

29 December 8, 2017

29.1 Final review

Let's look at all the complexity classes.

solid line is $\subseteq$, dashed line is $\subsetneq$.



There are also many open problems. If your solution "solves" an open problem, the solution if probably wrong.

$$\mathsf{P} \stackrel{?}{=} \mathsf{NP} \to \mathsf{SAT} \stackrel{?}{\in} \mathsf{P} \qquad \mathsf{P}^{\mathsf{SAT}} \stackrel{?}{=} \mathsf{NP}^{\mathsf{SAT}}$$
$$\mathsf{P} \stackrel{?}{=} \mathsf{NP} \cap \mathsf{coNP} \qquad \mathsf{P} \stackrel{?}{=} \mathsf{PSPACE}$$
$$\mathsf{BPP} \stackrel{?}{=} \mathsf{P} \qquad \mathsf{L} \stackrel{?}{=} \mathsf{NL}$$
$$\mathsf{NP} \stackrel{?}{=} \mathsf{coNP} \to \mathsf{SAT} \stackrel{?}{\in} \mathsf{coNP}$$

We move on to techniques to show membership.

L
- To show that $A \in \mathsf{L}$, we provide a log-space Turing machine for A, in which we can store pointers and counters.
- We can also show that $A \subseteq B$ for $B \in \mathsf{L}$.

NL
- To show that $A \in \mathsf{NL}$, we provide a log-space NTM for A or $\overline{A}$.
- We can reduce from PATH or BIPARTITE.

NP
- To show that $A \in \mathsf{NP}$, we provide a NTM for A.
- Provide a polynomial-length certificate for a verifier.
- Reduce from SAT, 3-SAT, etc.

Example 29.1

Let UNIQUE-SAT be the problem

$$\mathsf{UNIQUE\text{-}SAT} = \{\phi \mid \phi \text{ has exactly one satisfying assignment.}\}$$

Show that $\mathsf{UNIQUE\text{-}SAT} \in \mathsf{P}^{\mathsf{SAT}}$.

Query both assignments for each variable. Reject if both 0 and 1 satisfy some x_i.

Now we give recipes for showing completeness using reductions.

NL-complete	• Show that $A \in \mathsf{NL}$ or $\overline{A} \in \mathsf{NL}$. • Show that $B \leq_L A$ for NL-complete B, e.g. PATH, $\overline{\text{PATH}}$, E_{NFA}, EQ_{NFA}, or strongly-connected subcomponents.
NP-complete	• Show that $A \in \mathsf{NP}$. • Show that $B \leq_L A$ for NP-complete B, e.g. SAT, 3-SAT. • For 3-SAT reduction, design gadgets for the variables of the clauses and relate them to ϕ. Enforce consistency. Argue that $\phi \in$ 3-SAT $\leftrightarrow w \in A$.
coNP-complete	• Show that $A \in \mathsf{coNP}$. • Show that $B \leq_L A$ for coNP-complete B, e.g. EQ_{BP}.
PSPACE-complete	• Show that $A \in \mathsf{PSPACE}$. • Give a direct reduction, or show that $B \leq_L A$ for NP-complete B, e.g. TQBF, A_{LBA}, GG, EQ_{REX}. • For
EXPSPACE-complete	• Show that $A \in \mathsf{EXPSPACE}$. • Show that $B \leq_L A$ for NP-complete B, e.g. $\text{EQ}_{\text{REX}\uparrow}$.

Example 29.2

Show that 2-SAT is NL-hard.

Proof. We show that $\overline{\text{PATH}} \leq_L$ 2-SAT.

For edge $(x, y) \in G$, we add clause $(\overline{x} \vee y)$. Since we travel away from s and towards t, we map $s \to s$ and $t \to \overline{t}$. If there exists a $s \rightsquigarrow t$ path, then all variables along that path are forced to be true until $\overline{t}$, at which we reach contradicting assignments for t. By similar argument, if there does not exists a $s \rightsquigarrow t$ path, then there is a satisfying assignment.

□