Administrivia
* Book: Introduction into the Theory of Computation
  - 0, 1, 2, 3, 4, 5, 7 (not 6 or 8)
- Instructor: Dr. Don Sheehy (sheehy)
- Office Hours: Tu, Wed 11-12 @ EB2 3280
- Parts of Course
  - Automata
  - Computability
  - Complexity

Computational taxonomy is a better term for complexity

$2^i - 6k - 4h$

Chapter 0

Terms
- Sets: Unordered collection elements, defined by membership.
  - Operations
    + Union: $A \cup B$, $\bigcup_{A \in x} A$
    + Intersection: $A \cap B$, $\bigcap_{A \in x} A$
    + Difference: $A \setminus B$ ← no total b/c not commutative
    + Symmetric Difference: $A \triangle B = (A \setminus B) \cup (B \setminus A)$
      - Basically XOR
    + Power Set: Set of all subsets. $\text{Pow}(X) = \{Y : Y \subseteq X\}$
      - Includes empty set
  - Relations
    + Subset: $\subseteq$ or $\supseteq$
    + Strict Subset: $\subset$ or $\supset$
    + Equality: $=$
  - Other Things
    + Cartesian Product: $A \times B = \{(a, b) : a \in A, b \in B\}$
    + Cardinality: "Size of set." $|A|$
    + Combinations: ...
      Let X be a set & $n \in \mathbb{N}$
      $\binom{X}{2} = \{Y : Y \subseteq X \wedge |Y| = n\}$

Set Notation
Enumeration: {<elements>}
Builder: {<expr> : <... ⇒ bool>}

- Alphabet: A finite set of symbols. (Normally Σ)
- Strings: Sequences of symbols (elements) in some alphabet, must be finite.
- Language: A set of finite strings (over some alphabet), can be infinite.
- Proposition: True or false statement.
- Predicate: Function from something to boolean.
- Relation: Binary predicate (i.e. domain is 2-tuple & codomain is bool)
  - Equivalence Relation Properties
    + Reflexive: xRx = True
    + Symmetric: $xRy = yRx$
    + Transitive: $xRy \wedge yRz \Rightarrow xRz$

- Quantifiers: Allow us to make statements about existence & number. Turn predicates into propositions.
  - Universal: $\forall \Rightarrow$ every
  - Existential: $\exists \Rightarrow \geq 1$

In homework, tests, & papers, use full sentences & statements. Don't rely on symbols.

- Formal Implication: A implies B. $A \Rightarrow B$.

A	B	$A \Rightarrow B$
T	T	T
T	F	F
F	T	T
F	F	T

The same as $A \wedge \neg B$.

- Proof: Convincing logical argument. Proofs are for people & go into enough detail for the reader to be thoroughly convinced, a "level of comfort".
  - Each step is justified
- Theorem: Proposition that has been formally proven.
- Lemma: Small subproof used to help prove a theorem.

Tips for Proofs
- Convince yourself the statement is true/false.
- Alternate b/w trying to prove or disprove.
- Prove a simpler thing.

Tips for Understanding Proofs
- Remove hypotheses/requirements to see how the proof breaks.
- Lower the proof, make sure you can prove things glossed over.
- Do high level description of proof

Proofs

There are three big proof styles here

- Construction: Build up something from knowns.
- Contradiction: Assume the contrapositive of the proposition & reach a contradiction.
- Induction: Prove some small example & use that to build up proofs for all others.

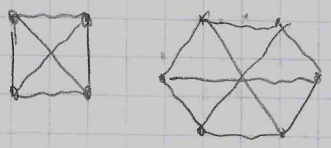
Construction w/ Graphs

A graph is a pair of sets (V, E) where $E \subseteq \binom{V}{2}$.

For $v \in V$, $\deg(v) = |\{e \in E : v \in e\}|$.

Prove that for all n even & $n > 2$, $\exists G \in (V, E)$ where $|V| = n$ s.t. $\forall v \in V$ $\deg(v) = 3$

$\neg \exists \Leftrightarrow \forall \neg$



Pattern!

$V = \{0, \dots, n-1\}$

$E = \{\{i, (i+1)\%n\} : i \in \{0, \dots, n-1\}\} \cup \{\{i, i+n/2\} : i \in \{0, \dots, \frac{n}{2}-1\}\}$

Induction w/ MU Game

Relevant rule of MU: $Mx \to Mxx$ for some string x of symbols from $\Sigma = \{M, I, U\}$

Prove $\exists$ derivation $MI \to MI^{2^n}$ $\forall n \in \mathbb{Z}$ $n \geq 0$

When $n = 0$, $MI = MI^{2^0}$

When $n > 0$, we assume true for $n-1$. One application of rule one

$MI^{2^{n-1}} \to MI^{2^{n-1}} I^{2^{n-1}} = MI^{2^n}$

Chapter 1: Regular Languages

In a string Σ means any symbol.

Notation

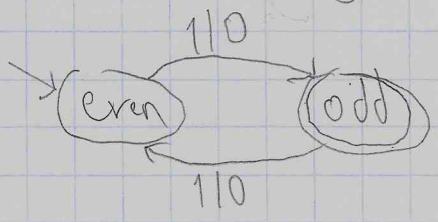
- a^i = a repeated i times
- $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$
- ε = empty string
  - $\Sigma = \varnothing$ has ε in this class

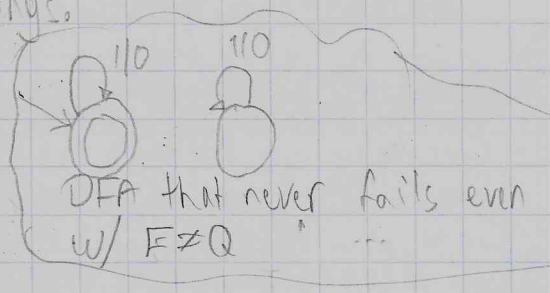
Deterministic Finite Automata (DFA) / Finite State Machine (FSM)

It is deterministic b/c each node has a single unique output. Finite b/c there are a finite number of nodes.

We normally represent this using a state diagram. Circles are states. Double circles are valid end states. Arrows w/ labels represent transitions b/w states w/ input given their label.

DFA for validating odd-length binary strings.





Def: DFA

We formally define a DFA as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set (states)
- Σ is an alphabet
- δ: is a transition function.
  $\delta: Q \times \Sigma \to Q$, $\delta(q, a) = q'$ state
  (q: state, a: symbol)
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ accept states ← *set

Give states meaning names

* This is awfully mechanical & lends itself to programming.

Since transition functions can be complex, we often draw a state diagram or transition table.

states \ symbols	Ø	1
0	E	E
E	0	0

Languages + DFAs

Given an alphabet Σ & a DFA M, the language $L(M)$ L defines is

$L(M) = \{ s \in \Sigma^* \mid M \text{ accepts } s \}$

Def: DFA accepts string

Let $M = (Q, \Sigma, \delta, q_0, F)$, $w = w_1 w_2 \dots w_n$ $w_i \in \Sigma$

Iff $\exists$ sequence of states $r_0, r_1, \dots, r_n \in Q$ s.t.

1) $r_0 = q_0$
2) $r_{i+1} = \delta(r_i, w_{i+1})$: for $i = 0, \dots, n-1$
3) $r_F \in F$

* Note: Even tho we use existential quantifiers, if the DFA matches a string, there is only one sequence b/c it is deterministic.

Def: DFA accepts/recognizes language
Iff $L(M) = A$

Def: Regular Languages
A language where there is some DFA recognizes it.

* Even if a language is regular, a subset of it might not be. Σ^* is regular!

Regular Operations / Operations on Regular Languages

- Union: Normal set union
- Concatenation: $A \circ B = \{ ab \mid a \in A, b \in B \}$
- Star: $A^* = \{ x_1 x_2 \dots x_k \mid k \geq 0, x_i \in A \}$

Although intersection does have all the properties of regular operations, it's not useful enough to be included.

Thm: Regular languages are closed under union.

Given A & B are regular languages. $A = L(M_A)$, $B = L(M_B)$.

To union, we basically take the product & transform the rest.

You can do optimizations

$Q = Q_A \times Q_B$

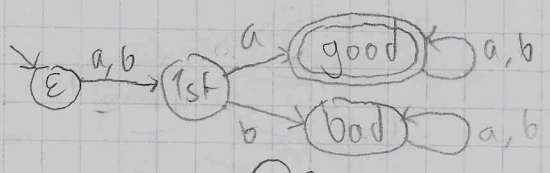
$\delta((q_A, q_B), s) = (\delta_A(q_A, s), \delta_B(q_B, s))$

$F = \{ (q_A, q_B) : q_A \in F_A, q_B \in F_B \}$

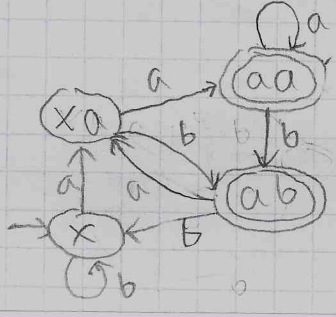
$q_0 = (q_{A0}, q_{B0})$

Practice

$\Sigma = \{a, b\}$
$A = \{ x a y \mid x \in \Sigma, y \in \Sigma^* \}$



$\Sigma = \{a, b\}$
$A = \{ x a y \mid x \in \Sigma^*, y \in \Sigma \}$



Non-Deterministic Finite Automata (NFA)

A non-deterministic finite automata is just like a DFA except that the transition function is no longer a function. In other words, you can have multiple arrows w/ shared symbols out of a state. Additionally we have epsilon arrows, which you can take for free at any time.

Def: NFA

We formally define a NFA as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

Q is a finite set (states)

Σ is an alphabet

δ is a transition function

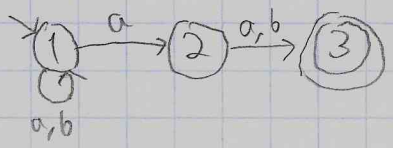
$\delta: Q \times \Sigma_\varepsilon \to P(Q)$ ← power set (returns set of states)

$\Sigma = \Sigma \sqcup \{\varepsilon\}$ ← disjoint union

$q_0 \in Q$ is the start state ← only one (not super important)

$F \subseteq Q$ accept states

Example:



	a	b
1	{1,2}	{1}
2	{3}	{3}
3	∅	∅

We'll show it later, but all NFAs can be transformed into DFAs. They're just kinda ugly (normally).

Def: NFA accepts string

Let w be a string $w = y_1 \cdots y_m$ $y_i \in \Sigma_\varepsilon$ & M be a NFA $M = (Q, \Sigma, \delta, q_0, F)$.

M accepts w iff

$\exists r_0, \ldots, r_m \in Q$ s.t.

$r_0 = q_0$ &

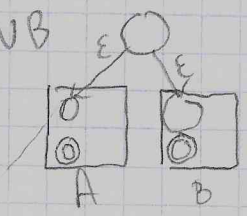
$r_{i+1} \in \delta(r_i, y_{i+1})$ &

$r_m \in F$

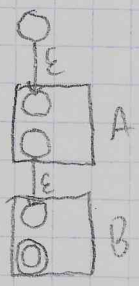
Regular Operations

If you believe NFAs & DFAs are equivalent (they are & we'll show it). It becomes really easy to prove the regular operations. We'll show just w/ diagrams.

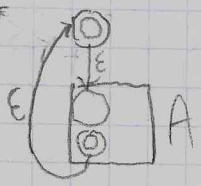
- Union: $A \cup B$



- Concatenation: $A \circ B$



- Star: A^*



Converting NFAs to DFAs

Although NFAs feel more powerful than DFAs, they're actually equivalent, Meaning all NFAs can be converted to DFAs.

The method is very similar to proving union w/ DFAs; the states for your DFA become sets of states from your NFA.

Construction: NFA → DFA

Let N be a NFA $N = (Q, \Sigma, \delta, q_0, F)$.
$\delta: Q \times \Sigma_\varepsilon \to P(Q)$

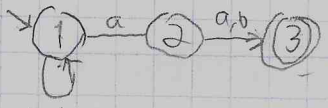
Let D be a DFA $D = (P(Q), \Sigma, \delta', q_0', F')$
$\delta': P(Q) \times \Sigma \to P(Q)$
$\delta'(R, a) = \bigcup_{q \in R} \delta(q, a)$, $R \subseteq Q$, $R \in P(Q)$

$q_0' = \{q_0\}$ ← need sets

$F' = \{R \subseteq Q \mid R \cap F \neq \emptyset\}$ ← or $\exists x \in R$ s.t. $x \in F$
$R \in P(Q)$ too

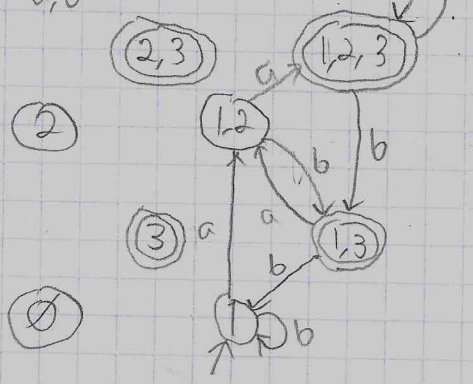
Example:

NFA:



it's a cube. not all are reachable

↳ DFA:



Handling ε-arrows w/ NFA → DFA

To handle ε-arrows, we need to complicate our NFA definition a bit.

We define a new operation E on a set of states R s.t.
$E(R) = \{q_k \in Q \mid q_k$ is reachable from R by zero or more ε-arrows$\}$
We will use this on DFAs from NFAs but not NFAs themselves.

Using E, we now redefine $q_0' = \{q_0\}$ & δ' for a DFA from a NFA:
Let N be a NFA $N = (Q, \Sigma, \delta, q_0, F)$.
Let D be a DFA from N $D = (P(Q), \Sigma, \delta', E(\{q_0\}), F')$

$\delta: Q \times \Sigma_\varepsilon \to P(Q)$
$\delta': P(Q) \times \Sigma \to P(Q)$
$\delta'(R, a) = \bigcup_{q \in R} E(\delta(q, a))$

Recall $F' = \{R \subseteq Q \mid R \cap F \neq \emptyset\}$

DFA → NFA (The Easy One)

Let D be a DFA $D = (Q, \Sigma, \delta, q_0, F)$.
Let N be a NFA from D $N = (Q, \Sigma, \delta', q_0, F)$

Define δ' s.t. $\delta'(q, a) = \{\delta(q, a)\}$.

That's it! Although trivial, symmetry is an important part of equivalence.

Proving $L(N) = L(D)$

We & the book won't do formal proofs here, but hopefully you're convinced by being able to convert any NFA → DFA & vice versa.

Regular Expressions

* Note: The regular expressions we discuss here are formal & thus not equivalent to programming language's regex. (Especially PCRE which technically can't be regular expressions.) We won't follow traditional regex syntax as such.

Regular expressions are essentially string-like shorthands for NFAs.

To express regular expressions, we first add meta-characters to the alphabet.
$\Sigma \cup \{\text{'U'}, \text{'o'}, \text{'('}, \text{')'}, \text{'*'}, \text{'ε'}, \text{'ø'}\}$

A string is a regular expression if has the following syntax
1) a (for some $a \in \Sigma$) original
2) ε
3) $\varnothing$
4) $(R_1 \cup R_2)$
5) $(R_1 \circ R_2)$
6) (R_1^*)

} Where R_1 & R_2 are regular expressions

Precedence: $* > \circ > \cup$
Concatenation ($\circ$) is elided

We now define the semantics for all these cases.
1) $L(`a') = \{a\}$ (for some $a \in \Sigma$) original
2) $L(`\varepsilon') = \{\varepsilon\}$
3) $L(`\varnothing') = \varnothing$
4) $L(`(R_1 \cup R_2)') = L(R_1) \cup L(R_2)$
5) $L(`(R_1 \circ R_2)') = L(R_1) \circ L(R_2)$
6) $L(`R_1^*') = L(R_1)^*$

'$R \circ \varnothing$' defines the same language as '$\varnothing$' b/c you demote all the accept states in the LHS & make them point to the start state of the RHS.

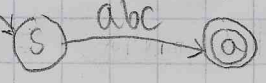
Regular Expressions → NFAs (via [illegible])

We know how to do an NFA trivially for cases 1, 2, & 3.
Cases 4, 5, & 6 are applications of regular operations on regular expressions.
Thus, regular expressions can be converted to NFAs. & define regular languages

NFAs → Regular Expressions (via GNFAs + simplification)

The core idea of this is compacting a NFA until we have just a single regular expression.

We call the compacted NFAs <u>generalized non-deterministic finite automata (GNFA)</u>.



↑ GNFA for 'abc'

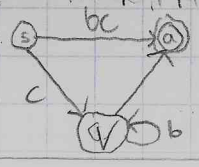
GNFAs have one start state s (no in-arrows)
& one accept state a (no out-arrows).

Each arrow (ordered pair of states) has a regular expression, expressed formally as
$\delta : (Q \setminus \{a\}) \times (Q \setminus \{s\}) \to R$ (regular expressions)

If we don't draw an arrow, we assume the empty set

<u>Example: δ of GNFA</u>

GNFA:



δ:

	s	q
a	abc	c
q	c	b

Def: GNFA accepts string

Let N be a GNFA & w be a string $w = w_1 w_2 \dots w_m$ (substrings, NOT characters).

N accepts w iff $\exists r_0, \dots, r_m \in Q$ s.t.

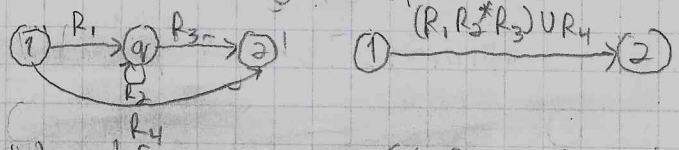
$r_0 = s$

$r_m = a$

$w_{i+1} \in L(\delta(r_i, r_{i+1}))$ ← the substring is accepted by the language b/w those two states r_i, r_{i+1}.

* Note: We are splitting string w into arbitrary substrings. This includes as many empty strings as you want.

Now we work on simplifying GNFAs. We do this by only focusing on one state q & handling all "paths" thru it.



We define a new δ' from δ that does not use δ.

$$\delta'(1,2) = \delta(1,2) \cup (\delta(1,q)\,\delta(q,q)^*\,\delta(q,2))$$

* gracefully handles empty set

Since we can always apply this rule until we only have s & a & this operation doesn't change the language (not proved here), we have shown you can convert a NFA to a regular expression.

Thm: Regular Expressions define Regular Languages & Woo! A third method

The class of languages described by regular expressions is the regular languages.

Chapter 2: Context Free Languages

We'll start w/ an example of a non-regular language & a proof of it. Later we'll get into context free grammars (CFGs).

Pf: Motivation

Let $A = \{ s \mid s \text{ has equal \# of 0's \& 1's} \}$ be a language.
Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA.

We want to show $L(M) \neq A$ (for all M).

Suppose M recognizes A. It therefore accepts $s = 1^{|Q|}0^{|Q|}$.

By the pumping lemma, there is some repeated states since $|s| = 2|Q| > |Q|$. We can therefore remove some repeated states. Since a DFA is deterministic, we know the repeated state has the same "input" symbols, so we remove an arbitrary amount of 1s (or 0s WLOG).

This gives us $s' \in L(M)$ where $s' = 1^{|Q|-a}0^{|Q|}$, which is not a string in A.

This is a contradiction. □

Pumping Lemma & Proving Non-Regular Languages

Def: Pumping a String

A string $s \in A$, where A is some language, can be pumped if s can be decomposed into substrings

$$s = xyz$$

where

$$|y| > 0$$

s.t.,

$$L(xy^*z) \subseteq A \leftarrow y \text{ can be repeated or absent}$$

In other words, we can add/remove repetitions. There can be many different repetitions; we can pump any one but only one at a time.

Def: Pumping Lemma (Don-style)

If A is a regular language, all sufficiently long strings $s \in A$ can be pumped.

We will normally use the pumping lemma as the contrapositive to prove non-regular languages. In other words, we say
"If there exists sufficiently long string $s \in A$ that cannot be pumped, then A is non-regular."

Def: Pumping Lemma (Book-style)
For all regular languages A,
there exists a number p s.t.
for all $s \in A$ where $|s| \geq p$,
s can be decomposed into $s = xyz$ s.t.
(i) $xy^iz \in A$ for all $i \geq 0$,
(ii) $|y| > 0$, &
(iii) $|xy| \leq p$.

This $\forall$ & $\exists$ back & forth is like a game ($\forall$ = opponent, $\exists$ = you).

* Note: Given a finite language A, you won't be able to pump any string in A, but you can just choose p larger than every other string in the language, so it's not a problem.

Proof: Pumping Lemma
Let A be a regular language.
By the definition of regular languages, there exists a DFA M s.t. $L(M) = A$ & $M = (Q, \Sigma, \delta, q_0, F)$.
Let $p = |Q|$.
Let $s \in A$ s.t. $|s| \geq p$.

Because M accepts s, $\exists r_0, \dots, r_{|s|}$ s.t.
$r_0 = q_0$,
$r_{i+1} = \delta(r_i, s_{i+1})$ for $i \in \{0, \dots, |s|-1\}$, &
$r_{|s|} \in F$.

By the pigeon-hole principle, among $r_0, \dots, r_{|s|}$ there exists some repetition $r_j = r_k$, $j < k \leq p$ s.t.
$s = xyz$,
x takes us $r_0, \dots, r_j$,
y takes us $r_j, \dots, r_k$, &
z takes us $r_k, \dots, r_{|s|}$.

This means
$|x| = j$,
$|y| = k - j > 0$ (b/c $j < k$), &
$|xy| = j + (k-j) = k$
thus showing (ii) & (iii).

We now show (i), i.e. $xy^iz \in A$ for all $i \geq 0$,
x takes M r_0 to r_j,
y^i takes us r_j to r_k, & ← since $r_j = r_k$ by definition, so y^0 still works.
z takes us r_k to $r_{|s|}$.

Example: Palindromic Strings

Let P be a language s.t. $P = \{$ palindromic strings$\}$ where the alphabet $\Sigma = \{0, 1\}$. In other words

$P = \{w \in \Sigma^* \mid w = w_1 \cdots w_n \quad w_1 \cdots w_{n/2} = w_n \cdots w_{n/2}\}$.

Suppose for contradiction that P is regular.

By the pumping lemma (PL), there exists p s.t. all $s \in P$ w/ $|s| \geq p$ can be pumped.

Let $s = xyz$ s.t.
- $|y| > 0$,
- $|xy| \leq p$, &
- $xyyz \in P$. ← in fact $xy^iz \in P$ for $i \geq 0$.

Suppose
$s = 0^p 1 0^p$
where
- $xy = 0^p$ ✓
- $x = 0^{p-i}$ ✓
- $y = 0^i$ where $i > 0$, &
- $z = 10^p$.

We now show $xyyz$ is not in the language as
$xyyz = 0^p 0^i 1 0^p = 0^{p+i} 1 0^p$
is not a palindrome for $i > 0$.

Therefore $xyyz \notin P$, thus reaching a contradiction ↯ showing P is not a regular language.

Context Free Grammars (CFGs)

Context free grammars are our first formal language more powerful than regular languages

Def: Context Free Grammar (CFG)

A Context free grammar G is a 4-tuple $G = (V, \Sigma, R, S)$ where
- V is a finite set of variables, $\} V \cap \Sigma = \emptyset$
- Σ is a finite set of terminals, $\}$
- R is a finite set of substitution rules or productions that look like
  Variable $\to (V \cup E)^*$
- $S \in V$ is the start variable.

Normally, multiple simple production rules for the same variable are joined on one line by a pipe |. For example

$R \to a$
$R \to b$

can be written as

$R \to a \mid b$.

Def: Yielding Strings

We say string1 yields string2 iff we can go from string1 to string2 in one production, i.e. given rule $A \to w \in R$ then uAv yields uwv.

If s1 yields s2, we write
$s1 \Rightarrow s2$.
If s1 can yield s2 after multiple productions, we write
$s1 \overset{*}{\Rightarrow} s2$. (s1 yields star s2).

Def: String in a CFG

Let G be a CFG s.t. $G = (V, \Sigma, R, S)$.
Let w be a string.

$w \in L(G)$ iff $S \overset{*}{\Rightarrow} w$.

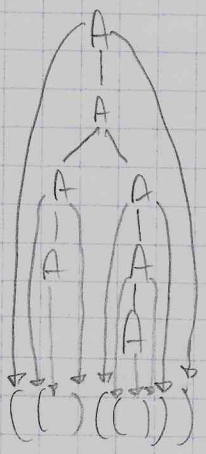
We call the sequence of productions used to go from S to w the derivation.

Def: Parse Tree

Parse trees are an easier way to write derivations, allowing you to track the "history" of derivation.

For example, given CFG $G = (V, \Sigma, R, S)$, here is the parse tree for (()(()))

Even if you have different parse trees, you can accept the same string.



Def: Context Free Language

A context free language is a language a CFG can generate

Def: Ambiguous Grammars

CFG G is ambiguous if there exists a $w \in G$ such that there are two different leftmost derivations. In other words, there are multiple parse trees.

The leftmost derivation is where you always resolve the leftmost variable. This just eliminates uninteresting decisions on order from mattering.

Context Free Languages vs Regular Languages

We have already shown CFGs can generate non-regular languages. For example

$A \to (A) \mid AA \mid \varepsilon$

is the language of matched parens, which is not regular.

Can we show all regular languages can be described by a CFG?

Proofs: Regular Languages $\subset$ Context Free Languages

The trick here is that all trivial (1 character) languages can be expressed by a CFG & CFGs support the regular operations.

CFGs can trivially define all trivial languages.

For all of the subsequent sections, suppose you have CFGs

$G_1 = (V_1, \Sigma, R_1, S_1)$ &
$G_2 = (V_2, \Sigma, R_2, S_2)$

where

$V_1 \cap V_2 = \emptyset$.

We find a new G for each operation.

Union: $L(G) = L(G_1) \cup L(G_2)$

Let $G = (\{S\} \cup V_1 \cup V_2, \Sigma, R_1 \cup R_2 \cup \{S \to S_1 \mid S_2\}, S)$.

Concatenation: $L(G) = L(G_1) \circ L(G_2)$

Let $G = (\{S\} \cup V_1 \cup V_2, \Sigma, R_1 \cup R_2 \cup \{S \to S_1 S_2\}, S)$.

Star: $L(G) = L(G_1)^*$

Let $G = (\{S\} \cup V_1, \Sigma, R_1 \cup \{S \to S_1 \mid SS \mid \varepsilon\}, S)$.

Pushdown Automata (PDA)

CFGs, like regular expressions, so far seem great for generating strings but pretty rubbish at testing for matches. We would like something like DFAs/NFAs for easy & efficient membership testing. Pushdown automata are how we'll do this.

We'll use the following syntax for describing rules for Pushdown automata. (PDA)
<char>, <pop> → <push>.

There are 5 main types of operations PDAs can perform
- $x, y \to z$ (standard)
- $x, \varepsilon \to z$ (don't pop)
- $x, y \to \varepsilon$ (don't push)
- $\varepsilon, y \to z$ (ignore input)
- $x, \varepsilon \to \varepsilon$ (do nothing w/ input)

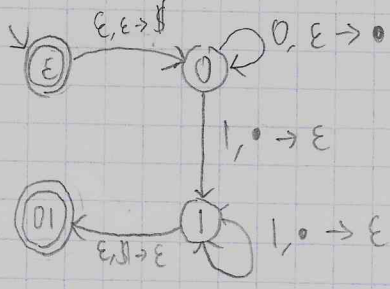
PDAs can be NFAs by just never pushing or popping.

Def: Pushdown Automata (PDA)
We define a PDA M to be
$P = (Q, \Sigma, \Gamma, \delta, q_0, F)$
where
Q is the set of states,
Σ is the alphabet,
Γ is the stack alphabet,
δ is the transition function where
$\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$, ← power set b/c non-deterministic, $Q \times \Gamma_\varepsilon$ b/c it considers & modifies both its stack & state.

$q_0 \in Q$ is the start state, &
$F \subseteq Q$ is the accept states.

We often draw pushdown automata w/ state diagrams, similar but more complicated for DFAs / NFAs.

Example:
We define a PDA P for a non-regular language L where $L = \{0^n 1^n \mid n \geq 0\}$.
Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where
$\Sigma = \{0, 1\}$
$\Gamma = \{\$, \bullet\}$, &
Q, δ, q_0, & F are given by the following state diagram



Here • acts like a count of n & $\$$ acts like a ticket in & out.

Def: PDA accepts string
A PDA P accepts $w \in \Sigma^*$ iff
$\exists w_1, \ldots, w_m \in \Sigma_\varepsilon$ where $w = w_1, \ldots, w_m$,
$\exists r_0, \ldots, r_m \in Q$ (sequence of states), &
$\exists s_0, \ldots, s_m \in \Gamma^*$ (sequence of stacks)
such that
$r_0 = q_0$,
$r_m \in F$,
The star is there b/c we can have many things on the stack.
(continued on next page.

$(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ for all $i \in \{0, \ldots, n-1\}$ where
$s_i = at$ & $s_{i+1} = bt$,
$a, b \in \Gamma_\varepsilon$, &
$t \in \Gamma^*$.

a is old top, b is new top, & t is old stack.

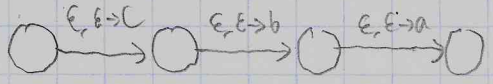
* Note: We don't actually care about the final state of the stack. If we actually care, we can use "tickets" like we did in the example earlier.

As you can see, it is a lot like an NFA accepting a string, except you must provide/have a valid sequence of states as well.

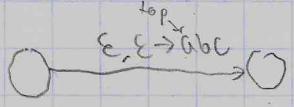
CFG → PDA

To help us, we're going to use some informalisms / gadgets.

Notation: Pushing multiple Things
We can always push multiple things onto the stack via

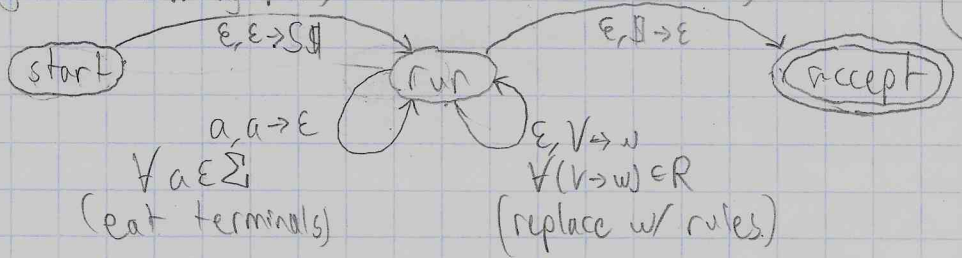


Let's shorten that to



We now convert a CFG G & PDA P where
$G = (V, \Sigma, R, S)$
and
$P = (Q = \{\text{start, run, accept}\}, \Sigma, \Gamma = \Sigma \cup V \cup \{\#\}, q_0 = \text{start}, F = \{\text{accept}\})$.

We construct P by the following state diagram.



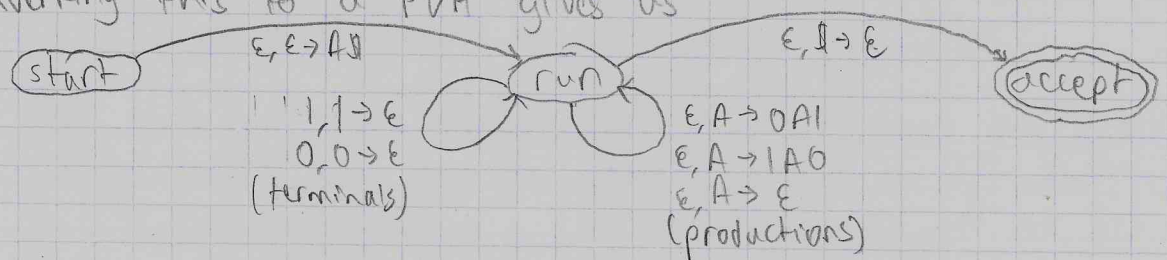
Every PDA can be optimized into 3 states: start, run, accept

Example:

Let G be a CFG defined by the following productions.

$A \to 0A1 \mid 1A0 \mid \varepsilon$

Converting this to a PDA gives us

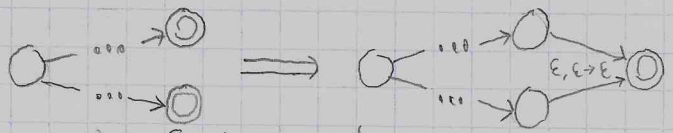


This method of transforming a CFG to a PDA is both easy to implement & does a leftmost derivation.

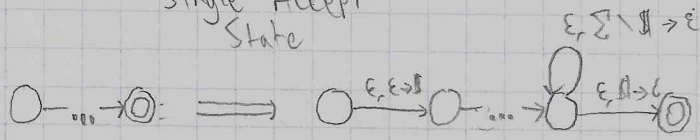
PDA → CFG

For the construction, we only consider PDAs w/ certain properties & then show that all PDAs can be made to have these properties. We require the following properties

i) Single accept state. Acheived via epsilon arrows from previous accept states to single accept state.
ii) Empty stack before accept state. Acheived via "stack dumping" w/ the "ticket" right between previous accept state & single accept.
iii) Never push & pop simultaneously. Acheived via splitting arrow w/ epsilon arrow to push after pop.



Single Accept State



Empty Stack

Never Push & Pop Simultaneously

* Note: This proof is a lot like dynamic programming & graph traversals.

Proof:

We define a CFG G such that $L(G) = L(P)$ for some PDA P.

We do this by defining variables A_{pq} for all $(p, q) \in Q \times Q$. We say $A_{pq} \Rightarrow^* w$ iff

P goes from state p to state q w/ an empty stack. ← We use this b/c throwing in an A_{pq} won't change the stack

We now define production rules (on the next page).

We handle the self-referential case. **Unit Case**

$\forall p \in Q,\ A_{pp} \Rightarrow \varepsilon$

We now handle the case where there is some path w/ an empty stack between. **Extend Case**

$\forall p, q, r \in Q,\ A_{pr} \Rightarrow A_{pq} A_{qr}$

We now handle the case where we push, do something, & then pop. **Bridge Case**

$\forall p, q, r, s \in Q,\ t \in \Gamma,\ a, b \in \Sigma_\varepsilon$,
where
$(q, t) \in \delta(p, a, \varepsilon)$ ← push
$(r, \varepsilon) \in \delta(s, b, t)$ ← pop
$A_{pr} \Rightarrow a A_{qs} b$.

We know this works b/c A_{qs} says that starting at q & ending at s doesn't change the stack.

T. Let the rules above be R.

Thus PDA $G = (\{A_{pq} \mid (p, q) \in Q \times Q\}, \Sigma, R, A_{start, accept})$

Going Beyond Context Free Languages (CFLs)

Thm:
A language is context free iff a PDA recognizes it.

The proof for some languages not being context free is similar to the pumping lemma for regular languages.

The intuition comes from looking for possible repetitions in the parse tree.

If we have some sufficiently long path in the parse tree ($\geq |V|$), then we know there is some repeated variable, meaning their is some loop by which the variable repeats itself. Thus we can "pump" this repetition. This repetition also "locks down" the language in a way where diverging paths are forever independent.

Thm: Pumping Lemma for CFLs
If A is a context free language, then
there exists a p such that
for all $w \in A$ where $|w| \geq p$,
there exists substrings u, v, x, y, z

Thm: Intersection of CFLs is not Context Free
Let A & B be context free languages. $A \cap B$ may not be context free.

Proof:
Let $A = \{a^i b^i c^* \mid i \geq 0\}$ & $B = \{\varepsilon, a^* b^i c^i \mid i \geq 0\}$ be CFLs.
Let $C = \{a^i b^i c^i \mid i \geq 0\}$ be a non-context free language.

$C = A \cap B$. □

Thm: Complement of CFL is not Context Free
Let A be a CFL. A^c may not be context free.

Proof:
Suppose complement is closed under CFLs. Let A & B be CFLs.

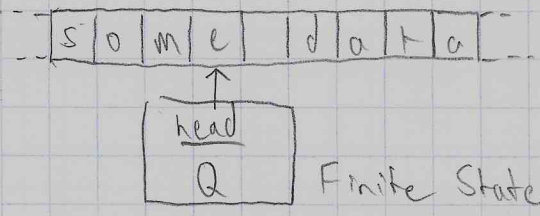
$(A^c \cup B^c)^c$ must be a CFL b/c union is closed & we assume complement is closed.

Recall $(A^c \cup B^c)^c = A \cap B$ by DeMorgan's Laws. Since $A \cap B$ is not closed under CFLs.

Therefore, complement is not closed. □

Chapter 3: Turing Machines

A Turing machine is fundamentally an infinitely long tape w/ a read/write head at a particular cell on the tape. The tape is controlled by a finite state machine. Basically, you have finite code but infinite memory. The head can only move one cell at a time.



It's like brain fuck!

* Note: A Turing machine gets its input as initial state on the tape.

The head starts at an "initial" position

Def: Turing Machines (TM)

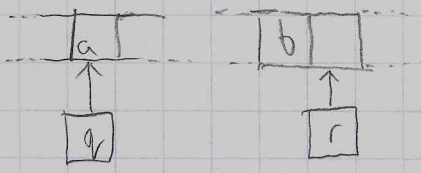
Let T be a Turing machine s.t.

$$T = (Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{reject})$$

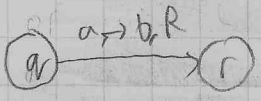
where

Q is a set of states,
Σ is an input alphabet,
$\Gamma \subseteq \Sigma$ is on tape alphabet, ← $\Gamma \subseteq \Sigma$ b/c input given as data
$q_{start} \in Q$ is the start state,
$q_{accept} \in Q$ is the accept state,
$q_{reject} \in Q$ is the reject state,
$\delta: Q \times \Gamma \to Q \times \Gamma \times \{R, L\}$
$\delta(q, a) = (r, b, R)$

↑ the tape normally has a "blank" symbol.



or

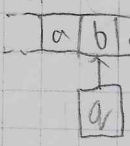


* Note: We always have to move left or right & consume or write a characters. We can add "gadgets" to fix these (e.g. move left then right or consume any symbol & write the same symbol). Since these capabilities don't add anything, make proofs harder, & may add ambiguity, we won't formally use those.

For a Turing machine to terminate, it must explicitly reach the accept or reject state. Otherwise, it is in an infinite loop / halting.

We often want to succinctly represent the configuration of a Turing machine (i.e. its tape, head location, & state).

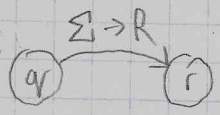
Notation: Configuration of TM



is written as "aqbcd" where $Q \cap \Gamma = \emptyset$.

You have to provide these as a "transcript" of the computation

Notation: Eliding Read/Write
When we want to not read anything (i.e. read & write same thing), we elide the write & put Σ for the read.



Def: TM accepts string
A Turing machine M accepts string $w \in \Sigma^*$ iff
$\exists c_0, \ldots, c_n$
s.t.
c_0 is the start config for w,
c_i yields c_{i+1} for $i \in \{0, \ldots, n-1\}$, &
c_n is an accepting config.

Def: Accept/Reject Config
An accepting config is one where state = q_{accept}.

A rejecting config is one where state = q_{reject}.

Def: Language of TM
Let M be a Turing machine.

$L(M) = \{ w \in W^* \mid M \text{ accepts } w \}$. ← not reject or halting

Def: Halting for a TM
A TM which is in a configuration such that it never reaches an accepting or rejecting state is called halting. ← This is really hard to detect.

Def: Decider
A decider is a TM that either always accepts or rejects, never halts.

Def: Turing Recognizable Language
A language is T-recognizable if $\exists$ TM that recognizes it.

Def: Turing Decidable
A language is T-decidable if $\exists$ TM that decides it.

Note: All turing decidable languages are turing recognizable.

It's often easy to think about TMs having variables. This works as long as your "variables" have finite possibilities.

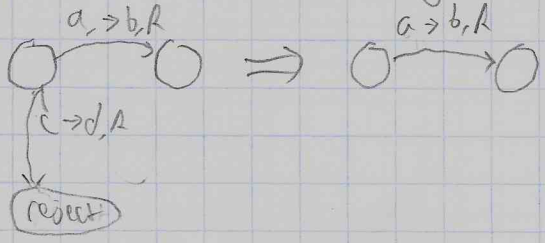
Sugaring

Let's make Turing machines slightly easier to use w/o changing their power.

Let's introduce the concept of not writing anything. You can simulate this w/ our old turing machine by reading & writing the same thing.

Let's introduce the concept of skipping over a cell (not reading or writing anything). You can simulate this by just reading & writing the symbol, no matter what you get.

We will often elide specific transitions to the reject state & assume unhandled symbols go to the reject state.



We can also have multiple tapes. Since this is more convoluted to prove that it doesn't make the TM more powerful, we'll start a new section.

Simulating k-tapes w/ One Tape

You could alternatively prove this w/ a more complex δ & "Q" if you had multiple elements in a single cell.

With a k-taped Turing machine (kTM), the transition function δ is defined as $\delta: Q \times \Gamma^k \to Q \times \Gamma^k \times \{R, L, S\}^k$, where super k means a homogenous k-tuple (i.e. $\Gamma^3 = \Gamma \times \Gamma \times \Gamma$).

To simulate a regular TM, we rely on special "marked" tape symbols & a special "move over" machine.

Given a kTM, w/ tape symbols Γ, we define a new TM w/ $a, b, \dot{a}, \dot{b} \in \dot{\Gamma}$ tape symbols that uses the following steps to run all the tapes

i) Prepare the tapes. (Put a marked blank on all non-prime tapes).
ii) Scan to read all heads.
iii) Use δ to find k actions $\in (\Gamma \times \{L, R, S\})$.
iv) Scan to update all tapes.

Whenever a tape wants to grow beyond it's current bounds, we push

everything to the left/right by one. We won't prove this machine exists here, but it should be pretty easy to think about.

Non-Deterministic Turing Machines

Non-deterministic TMs (NTMs) are as powerful as normal TMs. NTMs transition function δ is defined as $\delta: Q \times \Gamma \to P(Q \times \Gamma \times \{L, R\})$.

Simulating a NTM w/ a TM relies on us being able to have multiple tapes. Basically, the Turing machine starts w/ an input

+DDO: Finish me

Hilbert's 10th Problem

Problem:
Give a algorithm to decide if a given polynomial if a given polynomial (of n-variables) has integer roots.

Abstractly, we can plug every integer in & evaluate the polynomial to see if one evaluation results in 0.

This means we can (pretty trivially) recognize the language of polynomials w/ integer roots. (i.e. it is Turing-recognizable.)

But, can it be decided?

For 1-variable polynomials, yes b/c we can determine an upper & lower bound, giving us a finite search space.

For n-variable polynomials, it is not decidable, only recognizable.

The Church-Turing Thesis

Informally, the thesis is that the informal idea of an algorithm is equivalent to a Turing machine algorithm (& Church's Lambda calculus algorithm).

In this class / in general, there a 3 different "levels" at which you can describe a Turing Machine

- Formally $(Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{stop})$
- Implementation level (pseudo-code, code, recipe).
- High level (just prose)

If this is true, we've found "the model of computation" & we can't compute some things.

Decision Problems

For the purpose of using Turing machines to reason about computation, we encode the problem into dealing w/ strings & languages. Then we can use TMs to reason about the decidability of the problem.

This is called making the problem a decision problem.

Chapter 4: Decidability

We will use the framework of Turing Machines from chapter 3 to determine if problems are decidable or not.

If it's undecidable, maybe change the problem?

We will motivate this using the acceptance problem.

The Acceptance Problem (for Regular Languages)

Let $A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts string } w\}$ be a language. Is A_{DFA} a decidable language.

($\langle B, w\rangle$: encoding of DFA B & string w)

B & w can be any DFA & w can be any string. We are writing a TM which simulates a DFA.

In other words, a TM can simulate a DFA by reading an encoding of the DFA. Further, the TM can do it in a finite amount of time.

Heres a sketch of a TM that decides A_{DFA}:

1) The TM validates the DFA & TM encoding
2) Read in B & w to simulate B.
3) When the simulated DFA accepts, accept. Otherwise reject.

This is intuitively decidable b/c B is guaranteed to run in a finite amount of time (b/c it is given a finite string).

This also works for an NFA (A_{NFA}) b/c we can convert a NFA to DFAs. We just do that first & then run the same DFA. ← We write A_{NFA}

Likewise for regular expressions (A_{REX}) by doing regex → NFA → DFA.

Let $E_{DFA} = \{\langle A\rangle \mid A \text{ is a DFA \& } L(A) = \emptyset\}$ be the set of DFAs A that do not accept any strings.

Here is a sketch of the TM:

1) The TM reads the DFA as a graph & does a search for accept states.
2) If the TM traverses an accept node, accept. If the TM traverses no new nodes after a traversal, reject.

Since there a finite number of traversals, the problem is decidable/possible to do in finite time.

There is a much harder/unintuitive example. Determining the equality of DFAs is decidable. In other words
$EQ_{DFA} = \{\langle A, B\rangle \mid A = B \text{ i.e. } L(A) = L(B)\}$
is a decidable language. This isn't intuitive b/c the naive way would be to check all strings.

TO do this, we must recall the regular operations on DFAs: union, intersect, & complement.

TO decide EQ_{DFA}, we create a DFA C s.t. C = A symmetric difference B = $(A \cap B^c) \cup (A^c \cap B)$
We know we can do this by the regular operations. Then, we just check $\langle C \rangle \in E_{DFA}$, since E_{DFA} is decidable & we can do $C = (A \cap B^c) \cup (A^c \cap B)$ in finite time, EQ_{DFA} is decidable.

Acceptance Problems for Context-Free Languages

Def: Chomsky Normal Form
Chomsky normal form is a restricted form of CFGs that guarantees that no derivation of string w will be longer that $|w|+1$.

All rules are of the form
$T \to AB$ $\quad A, B \in R$
$T \to a$ $\quad a \in \Sigma$
$S \to \varepsilon$

Is $A_{CFG} = \{\langle G, w\rangle \mid G \text{ is a CFG that accepts string } w\}$ decidable? That is, is it possible to simulate/run a CFG in finite time.

It is and the strategy is this:
1) Rewrite the CFG G in Chomsky normal form, ← There is an algorithm which does this
2) Iterate through all possible derivations less than $|w|+1$ & check if the derivation works.
3) If we find a derivation that works, accept. If we can't, reject.

This follows the strategy of designing a TM that recognizes a language ↓ limiting it's search space to be finite.

Cantor's Diagonalization
A method of defng different "sizes" of infinity.
Russel's Parador

Russel's parador is where the collection of all sets is not a set.

Suppose $\mathcal{A} = \{\text{all sets}\}$ is a set. Let $D = \{s \in \mathcal{A} : s \notin s\}$ be the set of sets that do not contain themselves. Is $D \in D$?

If $D \in D$, then it shouldn't be in D by def

If $D \notin D$, then it should be in D by def.

Thus, $\mathcal{A}$ cannot be a set. ← Since having $\mathcal{A}$ is what broke everything.

Def: Enumerable
An infinite set S is enumerable if you can count the elements. Formally iff there exists a bijection b/w S & N (natural numbers), it is enumerable.

What are some enumerable sets?

- Natural Numbers: $\mathbb{N}$
- Set of finite strings: Σ^*
- Set of TMs.
  - Proof: We describe TMs by encoding them into strings. $\langle TM \rangle \in \Sigma^*$, so $\langle TM \rangle$ is enumerable.

Not all sets are enumerable tho!

- Real Numbers: $\mathbb{R}$
- Set of all languages

Thm:

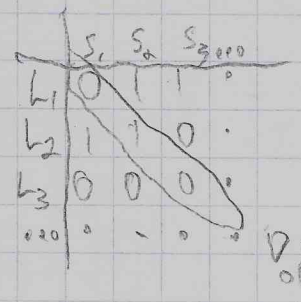
The set of all languages are not enumerable.

Pf:

Suppose for contradiction that we have $L_1, L_2, L_3, \ldots$ (i.e. the bijected/enumerated languages).

We define a diagonal language that is not in the enumerated languages.

$D = \{ s_i \in \Sigma^* \mid s_i \notin L_i \}$ ← valid b/c we can enumerate Σ^*



Thus, for every L_j,
if $s_j \in L_j$, then $s_j \notin D$
if $s_j \notin L_j$, then $s_j \in D$.
Thus D is a new language not in our enumeration.

We can play this game repeatedly adding new diagonal languages D.

Thus, L is not enumerable.

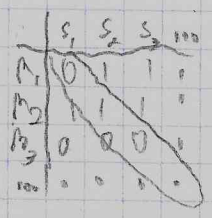
Thm:

There exist languages that are not turing recognizable.

A quick & dirty proof is that TMs are enumerable, but languages are not. Thus $|\{\text{languages}\}| >> |\{\text{all TMs}\}|$.

Pf:

We do another diagonalization. We can enumerate both strings & TMs to make a table.



We define a new language $D = \{ s_i \in \Sigma^* \mid TM_i \text{ does not accept } s_i \}$. That is, the language D is not recognized by any TM.

Acceptance of Turing Machines

This is like A_{DFA}, A_{NFA}, A_{REX}, A_{CFG} we did yesterday. However, unlike those, A_{TM} is undecidable.

Thm:

$A_{TM} = \{ \langle M, w \rangle \mid w \in L(M) \}$ is not decidable.

To prove this, we do a diagonalization by defining a language of TMs that don't accept their encoding.

Pf:

Suppose for contradiction that A_{TM} is decidable.

$J = \{ \langle M \rangle \mid \langle M \rangle \notin L(M) \}$ ← Set of languages that don't accept themselves. Recall $A_{TM} = \{ \langle M, w \rangle \mid w \in L(M) \}$

Let S be a TM that decides A_{TM}.
Let T be the opposite of S. We define T as

$T =$ "
On input $\langle M \rangle$
- Run S on $\langle M, \langle M \rangle \rangle$
- If S accepts, reject. Otherwise, accept
"

We know, by definition $L(T) = J$.

We now ask, is $\langle T, \langle T \rangle \rangle \in A_{TM}$? Equivalently, does T accept itself?

If yes, then S accepts $\langle T, \langle T, \langle T \rangle \rangle \rangle$, so T rejects $\langle T \rangle$. Meaning T rejects itself. Contradiction!

If no, then S rejects $\langle T, \langle T \rangle \rangle$, so T accepts $\langle T \rangle$. Meaning T accepts itself. Contradiction!

Therefore, A_{TM} is not decidable.

Is TM recognizable? Yep! Given input $\langle M, w \rangle$, you run M on w & see if it works. This relies on TMs being able to emulate other TMs, which is true, but we haven't proven. This is called universal computation.

Thm: Decidability & Complements
A language A is decidable iff
A is recognizable
& $\bar{A}$ is recognizable.

Pf: $\Rightarrow$
If A is decidable, then A is also recognizable by definition.

If A is decidable, then $\exists$ TM M which decides A. We define a new TM M' which runs M & then accepts when M rejects & rejects when M accepts.
Thus, M' decides $\bar{A}$, so $\bar{A}$ is recognizable.

Pf: $\Leftarrow$
Suppose A & $\bar{A}$ are both recognizable.
Let M_A & $M_{\bar{A}}$ be TMs that accept A & $\bar{A}$ respectively.

We define a new TM M which decides A. We define M to run M_A & $M_{\bar{A}}$ in parallel &
accept when M_A accepts
& reject when $M_{\bar{A}}$ accepts.

Thus A is decidable.

To prove something is not decidable, we normally use contradiction, following the below form.
Suppose some problem is decidable.
Use this problem's decider to create a decider for a language/problem that we know is not decidable.
Contradiction!

Post Correspondance Problem (PCP) ← Named after dude

We are given a set of dominoes

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\}$$

An example of a less direct reduction.

The goal is to find a sequence $i_1, \ldots, i_n$ s.t.

$t_{i_1} \circ t_{i_2} \circ \cdots \circ t_{i_n} = b_{i_1} b_{i_2} \circ \cdots \circ b_{i_n}$ (concatenation)

Example:

$\left\{ \left[\frac{aa}{b}\right], \left[\frac{b}{aa}\right], \left[\frac{a}{b}\right] \right\}$ has no solution b/c no domino starts (or ends!) w/ same symbol.

Example:

$\left\{ \left[\frac{a}{abbb}\right], \left[\frac{bb}{b}\right] \right\}$ is solved by 1, 2, 2, 2

PCP is recognizable b/c you can test all sequences.

Somewhat surprisingly, this problem is undecidable in general.

pf: ...

We prove PCP_Γ is undecidable by reducing A_{TM} to PCP. This reduction is done by supposing TM decides PCP & using that to decide A_{TM}.

How? We construct an instance of PCP that will have match iff M accepts w.

Assume that one domino is the start domino (resolved later) & the TM never moves off left of tape (use move over machine)

Our constructed instance of PCP is

$$\left\{ \left[\frac{\#}{\#\, q_0 w_1 \cdots w_n \#}\right] \right\} \cup \left\{ \left[\frac{a}{a}\right] \mid \forall a \in \Gamma \right\} \leftarrow \text{unchanged}$$

start → $\cup \left\{ \left[\frac{q a}{b r}\right] \mid \delta(q, a) = (r, b, R) \right\}$ ← move right

(can get rid of by adding symbols) $\cup \left\{ \left[\frac{c q a}{r c b}\right] \mid \forall c \& \forall \delta(q, a) = (r, b, L) \right\}$ ← move left

$$\cup \left\{ \left[\frac{\#}{\#}\right], \left[\frac{\#}{\sqcup \#}\right] \right\} \leftarrow \text{next configuration}$$

$$\cup \left\{ \left[\frac{a\, q_{accept}}{q_{accept}}\right], \left[\frac{q_{accept}\, a}{q_{accept}}\right], \left[\frac{q_{accept} \# \#}{\#}\right] \mid \forall a \in \Gamma \right\} \leftarrow \text{accept}$$

Only dominoes that allow you to "catch up"

Chapter 5: Reducibility

Let's start out defining problems, solutions, & instances formally b/c they can be easy to mix up sometimes b/c of how we formally define things.

- Problem: A formal language we want to recognize.
  - Normally frame difficulty w/ $\Omega(f)$ to say no solution can do better than runtime f. ($\Omega(f)$: lower bound)
- Solution: A Turing machine that accepts the language of a problem.
  - Normally frame performance w/ $O(f)$ to say our solution never does worse than runtime f. ($O(f)$: upper bound)
- Problem Instance: A specific string from the problem language.

Let's talk about a big problem, the halting problem. The halting problem asks "Can we look at a TM & see if it is possible for it to loop forever? Formally,

✡ $HALT_{TM} = \{\langle M, w\rangle \mid M \text{ halts (either accepts or rejects) } w\}$

is not a recognizable language. To prove this we will assume this is true & use its decidability allows other problems, known to be undecidable, to be decidable, reaching a contradiction.

This pattern is called reducing one problem to an instance (or set of instances) of another. In other words, we are transforming problems into different (but similar) problems. This is extremely similar to subsets.

Thm: Reduction & Decidability

Let A & B be problems.

If A reduces to B (in a sense $A \leq B$), then

A undecidable $\Rightarrow$ B undecidable &

B decidable $\Rightarrow$ A decidable.

Framing this using sets:

A decidable $= \forall x \in A$ x is decidable.

A undecidabe $= \exists x \in A$ x is not decidable

A reduces to $B \Leftrightarrow A \leq B$

$\exists x \in A$ s.t. x is not decidable $\Rightarrow \exists x \in B$ s.t. " " &

$\forall x \in B$ x is decidable $\Rightarrow \forall x \in A$ x is decidable.

We use this concept of reducibility to show that $HALT_{TM}$ is undecidable.

Thm:

$HALT_{TM}$ is undecidable.

Pf: $HALT_{TM}$ reduces to A_{TM}
Suppose $HALT_{TM}$ is decidable & is decided by TM A.

We define a TM S that decides A_{TM} using A.

"On input $\langle M, w \rangle$
1) Run R on $\langle M, w \rangle$
2) If R rejects $\langle M, w \rangle$ (i.e. $\langle M, w \rangle$ loops forever), reject.
3) Otherwise, simulate M on w b/c you know that M will terminate on input w.
"

We have thus defined a decider for A_{TM}, which we know is impossible. Thus $HALT_{TM}$ is undecidable.

Thm:

$E = \{\langle M \rangle \mid L(M) = \emptyset\}$ be the set of Turing machines that accept no strings.

E_{TM} is undecidable.

Pf: E_{TM} reduces to A_{TM}
Suppose that TM R decides E_{TM}.

We define a new TM S that decides A_{TM} using R:

"On input $\langle M, w \rangle$
1) Construct a new TM M'
"M' = "On input x
1) If $x \neq w$, then reject
2) Otherwise, accept w if M accepts
"
2) Run R on M' to determine if $L(M') = \emptyset$. (By the definition of M', R accepts $M' \Rightarrow M$ accepts w & R rejects $M' \Rightarrow M$ does not accept w.)
3) If R accepts, accept. Otherwise, reject.
"

Define but don't run

By the way M' is defined, we know $L(M') = \begin{cases} \emptyset & \text{if } M \notin L(M) \\ \{w\} & \text{if } M \in L(M) \end{cases}$

We have thus defined a decider for A_{TM}, which we know is impossible. Thus $HALT_{TM}$ is undecidable.

Thm:

$Regular_{TM} = \{\langle M \rangle \mid M \text{ is a TM where } L(M) \text{ is regular}\}$ be the set of Turing machines that define regular languages.

Pf: $Regular_{TM}$ reduces to A_{TM}
Suppose that TM R decides $Regular_{TM}$.

We define a new TM S that decides A_{TM} using R to reach a contradiction.

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

1) Construct a new TM M'

M' = "On input x

1) If $x = 0^n 1^n$ accept
2) Otherwise, run M on w & accept if M accepts w

"

2) Run R on M' to determine if $L(M') = \Sigma^*$ (i.e. $w \in L(M)$) or if $L(M') = \{0^n 1^n\}$ (i.e. $w \notin L(M)$)

" 3) If R accepts, accept. Otherwise, reject.

By definition $L(M') = \begin{cases} \{0^n 1^n\} & \text{if } w \notin L(M) \\ \{0,1\}^* & \text{if } w \in L(M) \end{cases}$

In future don't put proofs inside their theorem. It causes ridiculous indentation

We have thus defined a decider for A_{TM}, which we know is impossible. Thus Regular_{TM} is undecidable.

Thm:

Let $EQ_{TM} = \{\langle M_1, M_2 \rangle : L(M_1) = L(M_2)\}$ be the language of equal TMs.

EQ_{TM} is undecidable.

Pf:

Suppose TM R decides EQ_{TM}.

We define a new TM S that decides E_{TM}. (Reduce EQ_{TM} to E_{TM}).

S = "On input $\langle M \rangle$

1) Run R on $\langle M, M' \rangle$ where M' rejects everything.
2) If R accepts, accept. Otherwise reject.

"

We have thus defined a decider for E_{TM}, which we know is impossible, so EQ_{TM} is undecidable. □

Computation History

Recall that if we have a TM M, we say M accepts w iff $\exists$ configurations $C_1, \ldots, C_n$ s.t. C_1 yields C_{i+1} & C_n is accepting configuration. We call this an accepting computation history.

By looking at computation histories, we can better reason about decidability. In particular, it can help us define looping

Restricted TMs

Ref: Linear Bounded Automaton (LBA)
A linear bounded automaton is a TM which cannot grow/write outside the size of the input tape.

This restricts our potentially infinite set of configurations to be finite, however it can be arbitrarily large.

Why is it linear bounded? The size of your memory grows linearly w/ input.

How many configurations can a LBA be?
There can be qng^n configurations
where
n is input length,
q is the number of states, &
g is the number of tape symbols.

$A_{LBA} = \{ \langle M, w \rangle \mid M$ is LBA that accepts string $w\}$ is decidable. Since an LBA has qng^n different configurations, we run the LBA M for qng^n steps.

Why? After qng^n steps, we know M has repeated at least one configuration, so it is stuck in a loop.

Thm:
Let $E_{LBA} = \{ \langle M \rangle \mid M$ is an LBA where $L(M) = \emptyset \}$.

E_{LBA} is not decidable.

Pf:
We show E_{LBA} is undecidable by reducing A_{TM} to E_{LBA}.

Given $\langle M, w \rangle$ we construct an LBA B s.t. $L(B) \neq \emptyset$ ($\langle B \rangle \in E_{LBA}$), iff M accepts w.

LBA B is given a computation history & checks
1) c_1 is a start config ($q_{start} w_1 \dots w_k$)
2) c_i yields c_{i+1}
3) c_n is an accepting config ($\dots q_{accept} \dots$).
Thus $L(B) = \{$accepting computational histories for $M, w\}$.

Assume R decides E_{LBA}. We define a TM S that decides A_{TM}.
$S =$ "On input $\langle M, w \rangle$
1) Write $\langle B \rangle$ on the tape
2) Run R on $\langle B \rangle$.
3) If R accepts, reject b/c there is no computation history that can result in acceptance. If R rejects, accept.
".

We cannot decide A_{TM}, so E_{LBA} is undecidable. □

Chapter 6: Complexity Theory

How do we give fine (grained) results on problems? (Unlike solutions we analyzed in CSC 316.)

Example:
$A = \{0^k 1^k \mid k \geq 0\}$.

We define a TM M to decide A

M = "On input w
1) Scan tape. If 0 comes after 1, reject.
2) Repeat while tape has 0's & 1's
   i) Scan tape. Cross off 1 zero & 1 one.
3) If anything is left, reject. Otw accept.
".

$T(M) = 2n + (n/2)(2n) + n = 3n + n^2$

Runtime is a specific time, not how long it took to run.

Def: Running Time / Time Complexity
Running time is a function $f: \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of steps needed for inputs of length n.

of a solution is
The most common way to measure running time using worst-case asymptotic analysis. Recall:
- Big O ($g \in O(f)$ or $g = O(f)$) $\approx g \leq f$
- Big Omega ($g \in \Omega(f)$ or $g = \Omega(f)$) $\approx g \geq f$
- Big Theta ($g \in \Theta(f)$ or $g = \Theta(f)$) $\approx g = f$

Now that we have a way to measure the running time of solutions to a problem, we can define time complexity for problems.

Def: Time Complexity Class
Let $t: \mathbb{N} \to \mathbb{N}$ be the running time of a function.
We define the time complexity class of running time T as
$TIME(t(n)) = \{L : \exists TM \text{ that decides } L \text{ w/ running time } O(t(n))\}$

Note: A language can belong to multiple time complexity class. That is $TIME(n \log n) \subseteq TIME(n^2)$

Example:
Let's decide $A = 0^k 1^k$ more efficiently to show $A \in TIME(n \log n)$

We define TM M to decide A

M = "on input w

$2n$ 1) Scan tape. If 0 follows 1, reject.

$\log_2 n$ 2) While there are still 0's & 1's on the tape, repeat

$2n$ i) Scan to check if total # of 0's & 1's is odd (one is even & one is odd). If so, reject.

$2n$ ii) Scan tape, cross out every other 0 & every other 1.

n 3) If nothing is left, accept. O/w reject.

"

The running time of M is $2n + (\log_2(n))(2n+2n) + n$

$2n + (\log_2 n)(2n+2n) + n = 3n + 4n\log n$

Different models of computation that are equivalent in power can give different running times.

Therefore $A \in TIME(n \log n)$.

We could do linear time $O(n)$ if we use 2 tapes. You just write all the zeros to the 2nd tape. Then you move the head over the first & second tape simultaneous to count them.

Thm: Multitape TMs

If A is decided by a $O(t(n))$ time multitape TM then $A \in TIME(t(n)^2)$ by a 1-tape TM.

Problem Classes

$TIME(\log n) \subseteq P$ b/c $TIME(\log n) \subseteq TIME(n)$

Def: P

P is the class of languages that have polynomial time solutions. Formally,

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

To show $A \in P$, it is sufficient to give a polynomial time algorithm.

Why? No operation a more sophisticated TM can do requires more than polynomial time to do. (For example, multitape TMs just take $t(n)^2$ time.)

We will now define NP (non-deterministic polynomial time). We need more tools tho.

Def: Verifier

A verifier for a language A is an algorithm V s.t.

$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some } c\}$

c is a "hint" or more likely certificate of the computation.

This is often much simpler than producing a solution. Recall our homework question about hints. $|c| = \text{poly}(|w|)$ (i.e. $|c| \leq n^{|w|}$)

Verifiers allow us to precisely define NP problems A "turn" non-deterministic TMs into deterministic TMs.

Def: NP

NP is the class of languages which have a polynomial time verifier.

Thm: NP & NTMs

A language is in NP iff it is decided by a poly-time NTM.

In other words,

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

Pf:

Let V be a $O(n^k)$ verifier for $A \in NP$.

We define a poly-time NTM that recognizes A

N = "On input w w/ length n,
1) Nondeterministically select a string c of length at most n^k.
2) Run V on $\langle w, c \rangle$.
3) If V accepts, accept. Otherwise, reject.
"

"takes constant time" bc you pick correct one immediately.

In summary
P = Solvable in poly-time
NP = Verifiable in poly-time.

There are some problems outside of NP.

Thm:
$P \subseteq NP$.

Pf:

Create a verifier by just solving the problem & ignore the certificate.

What about the other way? In other words, does $P \supseteq NP$? We can't find problems you can verify in poly-time & definitely can't solve in poly-time.

Recall we can simulate an NTM w/ a TM in $2^{O(t(n))}$ time.

therefore

$$NP \subseteq \underbrace{\bigcup_k TIME(2^{(n^k)})}_{EXPTIME}$$

If a language can be verified, it can be decided bc verifiers are deciders & can be simulated w/ TMs

Polynomial Time Reductions

Like reductions about decidability but for runtime.

This works by transforming instances of problem A to instances of problem B in polynomial time. Then, if $B \in P \Rightarrow A \in P$. The contrapositive also holds, $A \notin P \Rightarrow B \notin P$. Both of these proofs work by solving B w/ A.
"A polynomial-time reduces to B" $\Leftrightarrow$ A is not harder than B"

We also write, for A polynomial-time reduces to B,
$A \leq_p B$.
This notation makes the transitivity & assymetry clear:
$A \leq_p B \wedge B \leq_p C \Rightarrow A \leq_p C$. $A \leq_p B \not\Rightarrow B \leq_p A$ ← If P=NP, this is true

There's a special language SAT ∈ NP that is the "hardest" problem in NP.
In other words,
$\forall A \in NP \quad A \leq_p SAT$

If we solve SAT in polynomial time, then P=NP.

Let's formally define poly-time reducability.

We define a function/mapping from instances of A to B
$f: \Sigma^* \to \Sigma^*$ s.t. $w \in A \Leftrightarrow f(w) \in B$.

Def: Poly-time Computable
Mapping f (above) is poly-time computable iff $\exists$ TM M that takes input w & halts in $O(n^k)$ steps w/ output f(w).

Def: Poly-Time Reducible
A is poly-time reducible to B (i.e. $A \leq_p B$) iff $\exists$: poly-time computable
$f: \Sigma^* \to \Sigma^*$ s.t. $w \in A \Leftrightarrow f(w) \in B$.

Thm:
If $B \in P$ & $A \leq_p B$, then $A \in P$.

(You can reduce to way, way harder problems.)

Pf:

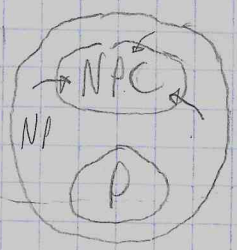
Let M be polytime TM that decides B.
Let f be the polytime reduction to B.

Define a polytime TM f that decides B
S = "On input w
1) Compute f(w).
2) Run M on f(w).
..." □

(The function $f: \Sigma^* \to \Sigma^*$ doesn't need to be a bijection (& often isn't).)

Def: NP-Complete (NPC)
A language B is NP-complete iff
i) $B \in NP$
ii) $\forall A \in NP \quad A \leq_p B$ ← wack!

Here's the idea:



Thm:
If B is NPC & $B \leq_p C$ where $C \in NP$, then $C \in NPC$.

This works by using the transitivity of $\leq_p$.

Pf:
$B \in NPC \quad \forall A \in NP \; A \leq_p B \wedge B \leq_p C \quad \therefore C \in NPC$
$\Rightarrow B \in NP \quad \Rightarrow \forall A \in NP \; A \leq_p C$
$\Rightarrow C \in NP$

The first NP-Complete problem was SAT (satisfiability).

Problem: SAT

Try to make boolean expression (variables, operations, & values) true by changing the variables. It is satisfiable if there does exist some combination of variables that makes it true.
Variables: True (1), False (0)
Operations: AND ($\wedge$), OR ($\vee$), NOT ($\neg$ or $\overline{var}$)

Thm: Cook-Levin Theorem
SAT is NP-Complete & thus $SAT \in P \Leftrightarrow P = NP$.

Pf:

We first show $SAT \in NP$.
The certificate is the assignment. You can trivially evaluate a boolean expression in poly-time.

We now show $\forall A \in NP \; A \leq_p SAT$.
To do this, for every A we need to build a boolean formula.

$A \in NP \Rightarrow \exists$ NTM M_A that decides A in $O(n^k)$.
Iff $w \in A$, then there is an accepting computation history for M_A on input w.

We know the size of the configuration history is $O(n^k)$ steps b/c if it took n^k steps, it could only grow the tape by ≤ 1 each time.

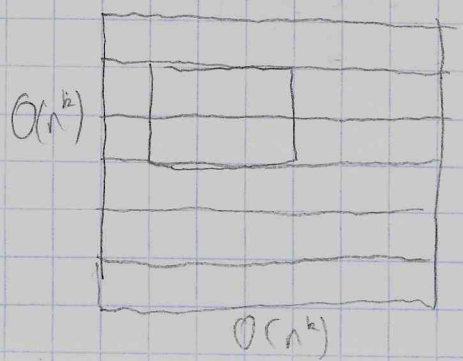
$n + \sum_{1}^{n^k} i$

$n + (n^k)(n^k+1)/2$

$\frac{1}{2} n^{2k} + \frac{1}{2} n^k + n$ ← largest possible

$O(n^k)$ (polynomial time)

We write the configuration history as a tableau



we can check chunks of these to see if they are consistent w/ the TMs rules. (2×3 chunks) We can define a boolean formula for these chunks & we know there will be $O(n^k)$ of them & we convert individually in $O(n^2)$, so we do the conversion in poly time.

To convert the tableau to a boolean expression, each cell $c_{i,j} \in Q \cup \Gamma \cup \{\#\}$, we define a variable

$$x_{i,j,s} = \begin{cases} \text{true} & \text{if } c_{i,j} = s \\ \text{false} & \text{o/w} \end{cases}$$

We write the boolean expression as

$$\phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$$

ϕ_{cell} is true iff there is exactly one symbol per cell.

ϕ_{start} is true if the start configuration is valid.

ϕ_{move} is true iff all moves are valid

ϕ_{accept} is true iff the final configuration is accepting

$$\phi_{cell} = \bigwedge_{1 \le i,j \le n^k} \left(\bigvee_s x_{i,j,s} \right) \wedge \bigwedge_{s,t} \left\{ \overline{x_{i,j,s} \wedge x_{i,j,t}} \right\} \quad O(n^k) \text{ in length}$$

for all cells — at least one symbol — at most one per cell

$$\phi_{start} = x_{1,1,q_{start}} \wedge x_{1,2,w_1} \wedge x_{1,3,w_2} \wedge \cdots$$

$$\phi_{accept} = \bigvee_{1 \le i,j \le n^k} \left(x_{i,j,q_{accept}} \right)$$

accept is somewhere

See book for ϕ_{move} in detail. It's complex. It checks that every 2×3 window that follows.

For example, this is a legal window for $\delta(q_1, b) = (q_2, c, L)$.

a	q_1	b
q_2	a	c

for A to SAT

We have thus show we have a polynomial time reduction, so $A \le_p SAT$.

Thus SAT is NP-complete. □

Def: Conjunctive Normal Form (CNF)

A boolean formula is in CNF iff it is of the form

$C_1 \wedge C_2 \wedge \ldots$

where C_i is a clause of the form

$(\text{literal} \vee \text{literal} \vee \ldots)$

Basically, CNF is an AND of ORs.

Disjunctive normal form has 3 literals

Def: 3-CNF

A formula is in 3-CNF if each clause has 3 literals.

Note: $x_1 \vee x_2 - x_1 \vee x_2 \vee x_3$

Problem: 3SAT

The set of satisfiable boolean equations in 3-CNF form.

Formally

$3SAT = \{\phi : \phi \text{ is satisfiable 3CNF formula}\}$.

Trivially $3SAT \leq_p SAT$ by the identity polynomial map.

More surprisingly, $SAT \leq_p 3SAT$. This is hard to prove directly, but you could prove 3SAT is NP-Complete using a similar method as you did SAT.

Def: Independent set from Graph

An independent set is a subset of vertices w/ no edges b/w them. $IS = \{\langle G, k\rangle \mid G \text{ has an independent set of size } k\}$.

Problem: Independent Set (IS)

Given a graph, does it have an independent set of size k?

We show IS in NP-Complete by showing 3SAT reduces to independent set (IS).

Pf: IS is NP-Complete

We define a poly-time reduction f from instances of 3SAT to IS.

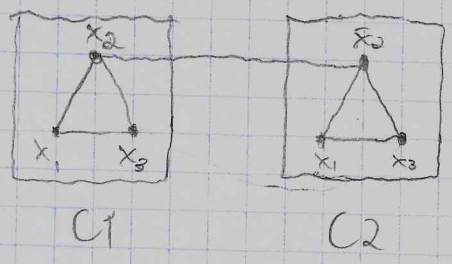
Given a boolean formula ϕ w/ k-clauses, we find a graph $f(\phi)$ where we want k-size independent set.

We draw groups of 3 vertices for each clause, where each term corresponds to a vertex. Picking a vertex corresponds to making that term true

Since we don't want to pick multiple terms w/in a clause, (not useful to do) we draw edges b/w all vertices in a 3-group.

Since we can't pick vertexes that correspond to a term & a not of itself, we draw edges b/w vertices & their not-counterpart.

For example, take $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$ gives us
(C_1 C_2)



Let's think about complements of languages now.

Def: coNP
$\text{coNP} = \{A \mid \bar{A} \in NP\}$. This the complement of NP but languages.

Thm: P w/ NP & coNP
P is a subset of NP ∩ coNP. That is $P \subseteq (NP \cap coNP)$.

Basically, coNP are problems where you refute a solution in polynomial time

Pf:

First we show $A \in P \Rightarrow A \in NP$. Let the verifier V for A just be the poly-time decider M for A. In other words, we need no certificate, we just solve it.

Now we show $A \in P \Rightarrow (A \in coNP \Leftrightarrow \bar{A} \in NP)$. Let the verifier for A just be the poly-time decider M for A w/ the accept & reject states swapped.

Def: NP-Hard
Let A be a language. A is NP-Hard iff
$\forall A \in NP: A \leq_p B$.

NP-Complete except doesn't have to be in NP

By the definition of NP-Hard, all NP-Complete problems are NP-Hard.

A lot of problems we try to solve are optimization problems. How do we frame those as decision problems? Normally, they (can) end up being NP-Hard.

Problem: Max-Clique
Given an input graph G, find the largest clique w/in the graph.

Thm:
Max-Clique is NP-Hard.

Pf:

The NPC problem k-clique reduces to Max-Clique. This relies on the fact that if a k-clique exists, then a (k-1)-clique exists.

Define the verifier for k-clique using Max-Clique

M = " On input $\langle G, k \rangle$
1) Return $k \leq$ Max-Clique(G).
"

Therefore, k-clique $\leq_p$ Max-Clique, so Max-Clique is NP-Hard

Space Complexity

We always talk about time complexity, but often space is important.

Def: PSPACE
Let PSPACE = $\{ A \in \Sigma^* \mid \exists$ TM that decides A in $O(n^k)$ space$\}$.

Thm:
P $\subseteq$ PSPACE b/c each step can only add at most one more cell to the tape. In P steps you cannot take more than PSPACE.

Thm:
PSPACE = NPSPACE b/c each step in an NTM-simulating TM only takes polynomial space & you don't need to store memory b/w steps.

PSPACE is really a lot of space. Let's limit it.

Def: LSPACE
Let LSPACE = $\{ A \in \Sigma^* \mid \exists$ TM that decides A in $(\log n)^k$ space$\}$.

Thm:
LSPACE = NLSPACE. This is though to prove. Read the book! :)

Cryptography

Somewhat unsurprisingly, cryptography cares a lot about time complexity. They want codes that are easy to verify but hard to solve/break.

One example of a problem is factoring, which is used to break RSA-encryption. We know factoring $\in (NP \cap coNP)$, but we don't think factoring $\in P$.