ECS 120 Lecture Notes
Phillip Rogaway

- Started: Fall 2012 (a TR class), based on notes from many prior years
- Last Revised: Spring 2023 (a MWF class)

Lecture 1.M

Today
- o General comments
- o Three example problems
- o Formal-language theory [which we didn't get to]

Announcements
- o Course information is on the WWW (not Canvas)
- o Quiz 1 on Friday
- o First non-homework not-due on Friday

General comments

Introduce the TAs: **Fatima** and **Sasha**

Running class differently from the way I have in past years:

No graded homeworks.

Discuss why I made this choice, and that fact that there will be homeworks distributed (or should I say non-homeworks?), which the TAs will discuss.

For your grades, I am replacing homeworks and the midterm(s) with

Friday quizzes

About 10 mins. Probably 9 of them.

Read the course-information sheet!

Also different from usual:

Probably the last time I will teach this class

(or, for that matter, teach any class at UCD). Feels sad.
Associated to this, I hope to teach a somewhat more relaxed, less frenetically paced class than I usually do. Ask me lots of questions to ensure that that happens.

Three example problems

1) GENERAL DIOPHANTINE EQUATION – Hilbert's 10th question
Does a polynomial $P(x_1, \ldots, x_n)$ / **Z** have an integer zero? Find it, or say that there is none.

Eg: $9x^2 + 12x^3 = 1000$ *no*, 3 divides LHS, not RHS

$x^2 + 2xy - x^3 = 13$ *no*, factor out an x to give $x(x+2y-x^2) = 13$, whence x must be ±1 or ±13 (we are working over **Z**), all of which force non-integral values for y,

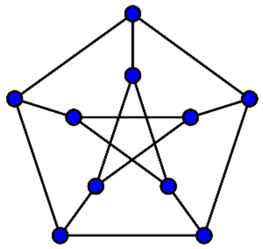
$x^2 + 2xy - x^3 = 8$ *yes*, x=2, y=3

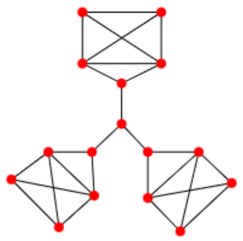
Impossible: [Yuri Matiyasevich 1970] (PhD thesis) (age 22)

Yuri Matiyasevich utilized a method involving Fibonacci numbers in order to show that solutions to Diophantine equations may grow exponentially. Earlier work by Julia Robinson, Martin Davis and Hilary Putnam had shown that this sufficed to show that every computably enumerable set is Diophantine.

2) PERFECT MATCHING

Given a graph of who likes whom, can we match people up into two-people per dorm room such that everyone likes their roommate? Assumes, however unrealistically, the "liking" is symmetric.


Yes


No

Easy: [Jack Edmonds, 1965]. Problem is in **P** (it has a polynomial-time solution). The Blossom algorithm $O(n^2 m)$. Improved to $O(n^{0.5} m)$ by [Micali, Vazirani (1980)

3) PARTITION INTO TRIANGLES

Same as above – but now the graph represents who likes whom, and we ask if we can find an assignment of people to dorm rooms where everyone likes their roommates.

Answer is No in both examples above – see why. Then make up an example where the answer is Yes, by starting with a union of distinct triangles and then adding in some extra ages.

All of the above phrased as **decision** questions. We could also have phrased them as **search** (or compute-the-right-value) questions: find an integer root or determine that none exists; find a perfect matching or determine that non exists; find a partition into triangles or determine that none exists. It doesn't much matter; the problem have essentially the same difficulty.

Search: Compute a "best" solution to some problem
Decision: Just yes/no

We will usually focus on **decision**. Why?

1) Simplifies things
2) Often (but not always), the search question reduces to the decision questions (explored in a homework)

That's all we got to. Some comments that I don't think I got to make follow:

How we will proceed:

- Define some sort of **model of computation**
- Investigate it
- Repeat

Models should be **technology-independent**
Necessary to make a "robust" theory

Topics
0. Automata Theory (warms us up for (1) and (2))
1. Computability (what separates Problem #1 from problems #2 and #3)
2. Complexity theory (what separates Problem #2 and Problem #3)

Lecture 1.W

Today
o Language-theoretic vocabulary
o Operations on strings and languages (except we didn't actually get to languages)

Announcements
o Quiz 1 on Friday
o First non-homework not-due on Friday

Language-theoretic vocabulary

Def: An **alphabet** is a finite nonempty set.
Its elements are called **characters**, or **symbols**.

Usually denote an alphabet using symbols Σ or Γ.

Eg:

- $\Sigma = \{0,1\}$
- $\Sigma = \{1\}$
- $\Sigma = \{a, b\}$
- $\Sigma = \{0,1,\ldots,9\}$
- $\Sigma = \{a,\ldots,z\}$
- $\Sigma =$ ASCII, $|\Sigma|=128$

No:

$\Sigma = \mathbf{N}$
$\Sigma = \varnothing$

Def: A **string** is a finite sequence of characters drawn from some alphabet Σ.
(more formally, a string x can be regarded as a pair
(n, f) where $n \geq 0$ is a number and a function $f{:}[1..n] \rightarrow \Sigma$ names the characters.

Ex: x=010
x=abbabab
x = hello there
$x=\varepsilon$ (the empty string – the unique string of length 0)

Can you tell what is Σ from looking at the string? NO. You only know that it contains at least the characters used in the string.

Strings and decision questions:

```
                 +-------+
                 |       | / accept
x --------->     |   M   |+
                 |       | \ reject
                 +-------+
```

- Strings are how we ask our questions – we encode them as strings.
- Strings are one way of returning answers – say YES or NO …. Or respond with a string that gives more information, like encoding an integer or an approximation of a real number
- Strings are also how we code our programs – a program can be considered a big long string.
- To a computer scientist … the book you are reading is a string
- The DVD you just watched is a string
- The contents of my computer can be encoded in a big long string
- Your genotype is a string – a sequence of characters from a 4-letter alphabet.

Not really an exaggeration to say that theoretical computer scientists think of **everything** of interest as being encoded as a string. But are there important things that **can't** be encodes as a string??

Operations on strings

Strings:

- concatenation. Forms a *monoid* (operator; associative & unit. But no inverse)
  Write as °, but often suppress.
- Squaring, raising to the n. What should we define x^0 to be? Want $x^{a+b}=x^a x^b$
- Equality testing
- Length, | . |
- Substring extraction. Beware of different conventions.
  x[a..b] means, for me, the a-th character until the b-th, inclusive,
  with indexing starting at 1. So abcdefg[1..3] = abc
  But in Python, say, a different convention is used for substrings, with
  abcdefg[1:3] = bc (note: not quite legal python; you would need quotes around abcdefg to make it a string an not a variable)
- Reversal, x^R

Lecture 1.F

Today:
- o Quiz 1
- o Languages and operations on them

Announcements
- o Quiz 1:10 – 1:20 pm. Clear your desk of stuff. Don't sit next to someone you know.

```
          F r o n t   o f   c l a s s

            *     *     *        defn of "next to"
            *    You    *
```

Def: A **language** is a set of strings, each drawn from the same underlying alphabet Σ.

L = {big, black, dog}
$L = \{1, 111, 11111, \ldots\} = \{1^n\text{: is odd}\}$ Not closed under concatenation. It grows
$L = \{\varepsilon, 11, 1111, \ldots\} = \{1^n\text{: is even}\}$. Closed under concatenation
$L = \{\varepsilon\}$ not the same as:
$L = \varnothing$
PRIMES = $\{1^p\text{: } p \text{ is prime}\}$
PRIMES10 = {2,3,5,7,11,13,17,…}
~~L = {all English sentences}~~ ← I don't like this one – not a precise enough description
~~L_w = {all strings that ChatGPT could respond when asked question w.~~

L = {d: d is the millionth digit of the decimal expansion of π
(digit 1000000 of 314159…)} ← I don't mind this
L = {d: d is a digit that appears infinitely often in the decimal representation of pi}

Not langugaes:
ε $\{\{\varepsilon\}\}$ **N** (the set of natural numbers)

Languages can be **finite** or **infinite** (the more "interesting" ones for us are infinite).

Describe the relationship between languages and decision procedures

```
                     +--------+
                     |        | / accept
     x --------->    |   M    |+
                     |        | \ reject
                     +--------+
```

$L = L(M)$ = the set of all strings that M "accepts"

Describe M as partitioning the world into two sorts of strings – those that it **accepts**, and the rest.

THESIS: By understanding **languages** we understand (a lot about) **computation**.

Operations on languages

Languages

- Concatenation, $A\,B = \{xy: x \in A, y \in B\}$, What is $L\,\varnothing$? What is $L\,\{\varepsilon\}$
- $L^2 = L\,L$
- Raise to the n. Definite it recursively: $L^n = L\,L^{n\text{-}1}$
  What should we define L^0 to be?
- Do we have L^{a+b}=$L^a\,L^b$
- L^R.
  *w is a **palindrome** if w=w^R.*
  *L is a **palindromic1** if $L = L^R$.*
  *L is a **palindromic2** if it contains only palindromes*
  same?
-
- More generally, if f is a map from strings to strings,
  we can extend it pointwise to languages,
  $f\,(L) = \{\,f(x): x{\in}L\}$
- L^+
- L^* Kleene closure, *. Definite it both recursively and iteratively.
  We didn't actually get to this on Friday; first thing Monday, I suppose

Lecture 2.M

Today

o Finish operations on languages, including Kleene closure

o Defining the <u>regular</u> languages

If L is a language then

$$L^+ = \{x_1 \cdots x_n : x_i \in L, \quad n > 0$$
$$= L \cup L^2 \cup L^3 \cup \ldots$$

where

$$: ^n = L\, L^{n-1} \quad \text{for } n>0.$$
$$L^1 = \mathrm{L}$$

so $L^+ = \cup_{n \geq 1} \; L^n$

L^+ = The smallest language that contains L and is closed under concatenation

If L is a language then

$L^* = \{x_1 \ldots x_n : x_i \in L, \quad n >= 0\}$. When n=0 we mean ε

$= L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$

= The smallest language that contains ε and L and is closed under concatenation

where $L^n = L\, L^{n-1}$ for n>0.

$L^0 = \{\varepsilon\}$

so $L^* = \cup_{n \geq 0} \; L^n$

T/F: L^* contains ε, but L^+ does not. **False**.
T/F: $\varnothing^* = \varnothing$. **False**: $\varnothing^* = \{\varepsilon\}$.
T/F: L^* is infinite. **False**. What are the exceptions?

Def: A **class** is a collection of languages.

Eg: the **finite** languages.

Formally specified **programming languages**

And the **regular** languages, which we are abut to define**.**

Regular languages

Definition. Let Σ be an alphabet.
The **regular languages** over Σ are the following:

1) $\{\varepsilon\}$, $\varnothing$, and $\{a\}$, where $a\in\Sigma$ are all regular languages;
2) If A and B are regular languages, then so are
    a. $A\,B$
    b. $A \cup B$
    c. A^*

Said compactly: "the smallest set of languages containing ~~$\{\varepsilon\}$,~~ $\varnothing$, and $\{a\}$ and closed under concatenation, union, and star.

Regular languages are conveniently denoted by **regular expressions**:

Def: The **regular expressions** over Σ are the following:

1) ε, $\varnothing$, and $a\in\Sigma$ are all regular expressions;
2) If α and β are regular expressions, then so are
    a. $(\alpha \circ \beta)$
    b. $(\alpha \cup \beta)$
    c. (α^*)

Example: {a, ab} is regular. Explain why.
Example: *Every* finite language is regular. Explain why

Lecture 2.W

Today:
- o The language of a regular expression
- o DFAs

Announcements: - Regrades for Q1 open
- Q2 on Friday

Review definition of regular expressions. Then define the language of a regular expression:

Def: Fix an alphabet Σ.

1) $L(\boldsymbol{\varepsilon})=\{\varepsilon\}$, $L(\boldsymbol{\varnothing})=\varnothing$, and $L(a) = \{a\}$ for all $a\in\Sigma$
2) If α and β are regular expressions, then so are
    a. $L((\alpha \mathbf{o}\ \beta)) = L(\alpha)\, L(\beta)$
    b. $L((\alpha\cup\beta)) = L(\alpha) \cup L(\beta)$
    c. $L((\alpha^*)) = (L(\alpha))^*$

Examples:
$(1\cup2)$ $((1\cup2)^*)$ $((1\cup2)^*) \circ (1^*)$ $(\varnothing\cup\varepsilon)\circ(\varnothing^*)$

Not regular expressions as defined above:

1^* $(0\cup 1)^*$ apple ∪ fig ∪ adam

We establish **conventions** so that the above are regarded as regular expressions: add parenthesis only "as needed", grouping left to right. To resolve ambiguity: group left-to-right;

* binds more strongly than concatenation;

Concatenation binds more strongly than union;

That is, the understood **precedence** is * then o then ∪

Concatenation symbol routinely omitted.

The $L(.)$ is also routinely omitted, eg, $L = (0\cup 1)^*$

$01^* = \{0,01,011,\ldots\}$

apple* ∪ fig* = {fig, figg, apple, figgg, applee, …}

Exampled:

1. All strings over {0,1}

   $\{0,1\}^*$ or $(0 \cup 1)$

2. Strings that have three consecutive a's over an alphabet of a's and b's

   $(a \cup b)^*$ aaa $(a \cup b)^*$

3. Strings that start and end with the same bit – over the binary alphabet

   $0(0 \cup 1)^* 0 \cup 1(0 \cup 1)^* 1$

Hmm, that's not right, need to include 0 and 1:

$0(0 \cup 1)^* 0 \cup 1(0 \cup 1)^* 1 \cup \mathbf{0} \cup 1$

Do we need to include ε? Not clear: does the empty string start and stop with the same character? One might say that the English is ambiguous on that.

One approach to defining interesting classes of languages: explain some sort of **machine**, and consider the languages **accepted** by that machine.

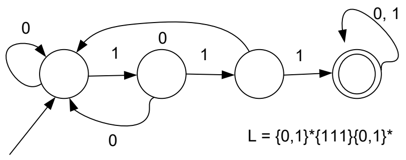
DFAs

x → <u>BOX *M*</u> → 1 (if x in L) or
for deciding *L* 0 (if x not in L)

Example of a first kind of machine – a **DFA**.
Deterministic Finite Automata (singular: automaton).

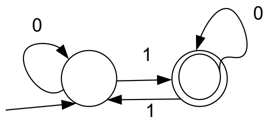
Describe the “meaning” of each state – “I’ve just seen a …”
Introduce $L(M)$ notation.

$L = \{w\in\{0,1\}^*$: w contains a “111”$\}$
$= \{0,1\}^*\{111\}\{0,1\}|^*$
$= (0\cup1)^*111(0\cup1)^*$



Can we make due with fewer than four states?

$L(M) = \{w\in\{0,1\}^*$: w is a binary string containing an odd number of 1’s$\}$



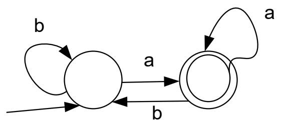
Lecture 2.F

Today: o Q2
o More practice with DFAs and regular expressions
o Formal definition for DFAs and their language

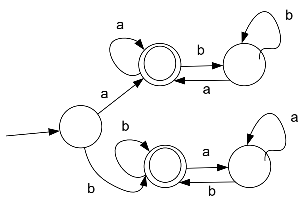
Announcements:
- Drop deadline is today

Different ways to look at $L(M)$:
1. Decision making device
2. Generating device
3. Not a device—just talk about the existence of paths

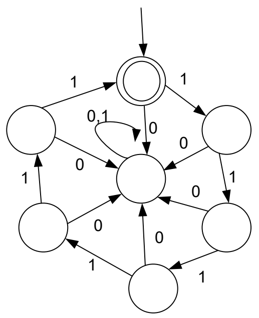
$L(M) = \{w\in\{a,b\}^*$: w ends in an “a”$\}$



$L(M) = \{w\in\{a,b\}^*$: w starts and ends with the same character$\}$. What about ε ?? Say it **doesn’t** start and end with the same character.



$L(M) = \{1^n : 6|n\}$. Assume an alphabet of {1}.   Assume an alphabet of {0,1}.



We did this with 7 states. Is it possible to do it with 6?

$L(M) = \{1^n$: *n* is dividible by *6* or *7*$\}$. Assume an alphabet of {1}.   How many states will you need?

Getting formal:

Def: A DFA is a 5-tuple $M=(Q, \Sigma, \delta, q_0, F)$ is a five-tuple where

- Q is a finite non-empty set ("states")
- Σ is an alphabet ("input alphabet")
- $\delta: Q \times \Sigma \to Q$ is a function ("the transition function")
- $q_0 \in Q$ is a state ("the start state")
- $F \subseteq Q$ ("the accept states")

How to define if *M* **accepts** *x*?

Def: M accepts *x* if $\delta^*(q_0\, x) \in F$

Where we define $\delta^* : Q \times \Sigma^* \to Q$ by extending δ to all **strings** as follow

$\delta^*(q, \varepsilon) = q$

$\delta^*(q, ax) = \delta^*(\delta(q, a), x)$

Can you think of a different way to do it?   Draw picture.

$$q \xrightarrow{a} r \xrightarrow{\quad x \quad} s$$

Proposition: $\delta^*(q, xy) = \delta^*(\delta^*(q, x), y)$

Proof: by induction on $|x|$. Omit, or HW exercise.

Basis: x=ε: $\delta^*(q, y) = ?$ $\delta^*(\delta^*(q, \varepsilon), y) = \delta^*(q, y)$
Inductive step: Suppose $\delta^*(q, x y) = \delta^*(\delta^*(q, x), y)$ if $|x|=n$. Show it's true if $|x|=n+1$. Write $x=aw$ where $|w|=n$:

$\delta^*(q, x y) = \delta^*(q, aw y)$
$= \delta^*(\delta(q, a), w y)$
$= \delta^*(\delta^*(\delta(q, a), w), y)$ inductive step
$= \delta^*(\delta^*(\delta(q, a), w), y)$
$= \delta^*(\delta^*(\delta(q, a), w), y)$
$= \delta^*(\delta^*(q, a w), y)$
$= \delta^*(\delta^*(q, x), y)$

Lecture 3.M

Today
- o Review definition of a DFA
- o Some closure properties of DFA-acceptable languages

Announcements:
- o This week's quiz will be **online** – available Thursday noon to Friday noon

DFA definition

Repeat the definition of a DFA as a 5-tuple $M=(Q, \Sigma, \delta, q_0, F)$, and review definition of $L(M)$ for a DFA M. Language is DFA-acceptable if there is a DFA for which it is the language.

The sort-of-arbitrariness of definitional choices: suppose we make the state set $[0..n\text{-}1]$. State 0 is the start state. Then a DFA would be a machine $M = (n, \Sigma, \delta, F)$ with δ: $[n] \times \Sigma \rightarrow [n]$. We would define L(M) with little change, and the DFA-acceptable languages wouldn't change at all. You could summarize and say that there is very little significance as to whether we make Q an arbitrary set or a specific set associated to the size of the machine. On the other hand: suppose we allowed the state set to be infinite. Then everything changes! Now there will be a DFA for any language. Changes everything … and in a way that makes the notion nearly meaningless.

Sometimes not clear at all. Like suppose I let the machine eat the input by going left or right, backing up. Doesn't change anything, but not at all obvious.

Closure Properties

Def: A language L is DFA-acceptable if there exists a DFA M s.t. $L=L(M)$
Def: A class is a set of languages
Write **DFA** for the class of DFA-acceptable languages.

We want to understand this class.

Closed under

	DFA	**Regular**
complement	yes	?
union	yes	yes
intersection	yes	?
symmetric difference	yes	?
concatenation	?	yes
Kleene closure (*)	?	yes
Reversal	?	Yes

Fill in table.

- ➔ For **union**: do the **product construction**. First an informal description, then a formal description. Then an example, maybe $L(M)=\{ 1^n \colon 3|n \text{ or } 5|n \}$. Assume an alphabet of $\{1\}$.
- ➔ For **intersection**: modify construction.
- ➔ Alternative for intersection $(A \cap B)^c = (A^c \cup B^c)^c$
- ➔ Difference $A \oplus B = (A - B) \cup (B - A)$

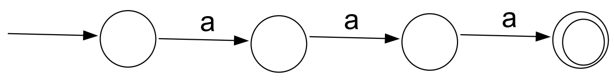
Lecture 3.W

Today
- o NFAs: notion/definition, closure properties, and the subset construction

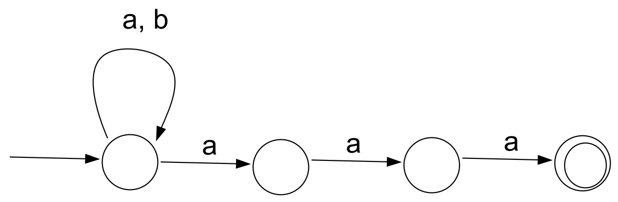
Announcements
- - Q3 to open tomorrow at noon

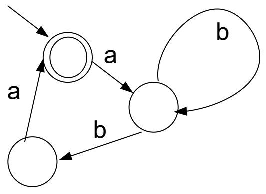
NFAs as machines with "defects" – either multiple outgoing arrows labelled by a given character … or no outgoing arrows labelled by a given character. Also, epsilon-arrows.

What language **should** we say this accepts?



Say the alphabet is {a,b}. Asymmetry of definition: accepts when there exists an accepting path; reject when there is no accepting path. More examples:



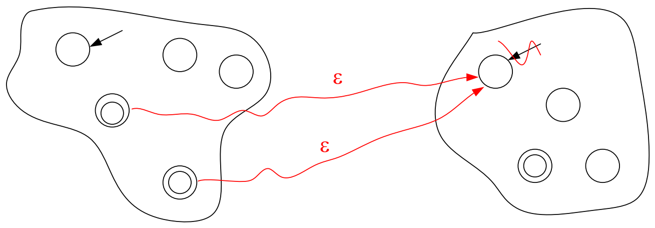


After introducing machines, reopen closure properties. **Concatenation** closure property. What would be cool: Could do concatenation by just "concatenating" the machines, **union** by just "unioning" them, **star** with back arrows and a new state to handle the epsilon-arrows. Described all of these constructions and argue correctness. All before giving a formal definition.

	DFA	**Regular**	**NFA**
complement	yes	?	?
union	yes	yes	yes
intersection	yes	?	?
concatenation	?	yes	yes
Kleene closure (*)	?	yes	yes
Reversal	?	yes	yes

If you want to understand something,
find multiple different-looking ways to characterize it.
Works quite generally across math and CS …

Show constructions for all of the new yes entries in the table:
Union, concatenation, *.
For *, maybe do it incorrectly first, letting students discover the problem.



If we had such ε-arrows, Iwe could show closure under concatenation and *, too.
Why don't we just **define** a new kind of machine that permits such arrows.
As long as we're relaxing things like this, let's drop the "exactly *one symbol from each element of Σ, too" rule*, allowing multiple arcs out of a state with a given label – or no arcs out of state with a given label.

Example: Let's do one more design, but with these new rules

- ➔ $L(M) = \{ x \in \{a,b\}^*: x$ ends in 'aabbaab'$\}$
- ➔ $L(M) = \{ 1^n: 3|n \text{ or } 5|n \}$. Assume an alphabet of $\{1\}$.
- ➔ Now assume an alphabet of $\{0,1\}$.
- ➔ $L(M) = \{w \in \{a,b\}^*: |w| \geq 4$ and w starts and ends with the same **pair** of characters$\}$

Lecture 3.F

Today

- o NFAs: formal definition
- o The subset construction (DFAs and NFAs accept the same languages)
- o Regular languages are again one in the same

Announcements

- Q3 over. Wow, you all do stuff last minute!

Formalization:
Def: An NFA $M = (Q, \Sigma, \delta, q_0, F)$ is … where
$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$
Book writes this Σ_ε

Definition:
An NFA $M = (Q, \Sigma, \delta, q_0, F)$ **accepts** x
if $\exists\ a_1, \ldots, a_n \in \Sigma \cup \{\varepsilon\}$ and $\exists\ q_1, \ldots, q_n \in Q$ where
$x = a_1 \ldots a_n$ and
$q_i \in \delta(q_{i-1}, a_i)$ for all $1 \leq i \leq n$ and
$q_n \in F$.

M **accepts** x if $\delta^*(q_0, x) \cap F \neq \varnothing$

$L(M) = \{x \in \Sigma^*: M \text{ accepts } x\}$

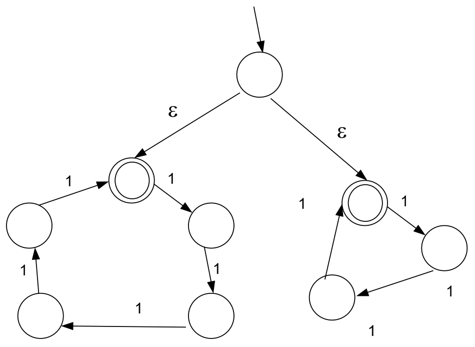
A language L is NFA-acceptable if there exists an NFA that accepts it

NFA = all the NFA-acceptable languages.

Alternative: define δ^*: $Q \times \Sigma^* \to \mathcal{P}(Q)$

- For $S \subseteq Q$, let $\mathbf{E}(S)$ = the smallest set containing S such that $q \in S \rightarrow \delta(q,\varepsilon) \in S$.
- Allow δ to act on sets instead of states by saying that $\delta'(S, a) = \cup q \in S \quad \delta(q,a)$
- Define

$$\delta^*(q, x) = \begin{cases} \mathbf{E}(\{q\}) & \text{if } x = \varepsilon \\ \mathbf{E}(\delta'(\mathrm{E}(\{q\}), a)) & \text{if } x = ay \text{ for some } a \in \Sigma, y \in \Sigma^* \end{cases}$$



How to think about nondeterministic computation?

1. **Existence of paths.** Think in terms of **paths**: if there **exists** an x-labeled path from the start state to a final state, you **accept**. If there exists no such paths, you **reject**. This view isn't very procedural. Could you actually fashion an NFA and run it on a string?
2. **LEDs** ← Given M, Could you build an NFA, of about the same size as M, from flip flops and LEDs that would take an x and decide if $x \in L(M)$? *YES* Proof associated to the **subset construction**.

T/**F**: If there exists a path from the start state to a non-final state, you reject.

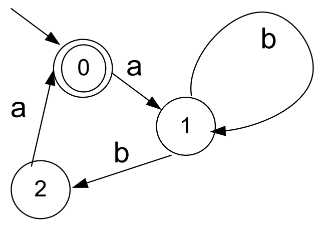
Can you "run" an NFA?

Theorem: **DFA** = **NFA** That is, every NFA-acceptable language is DFA-acceptable.

Step 1: get rid of e-transitions. Illustrate how.
(What does it mean no ε-arrows? $\delta(q, \varepsilon)=\varnothing$ for all $q \in Q$.)

Step 2: subset construction. Describe using LED-model. Product construction needs two fingers, but subset construction needs an unbounded number of fingers which you can spread across the board …) Figure out the correct final-state

Do **example** of subset construction

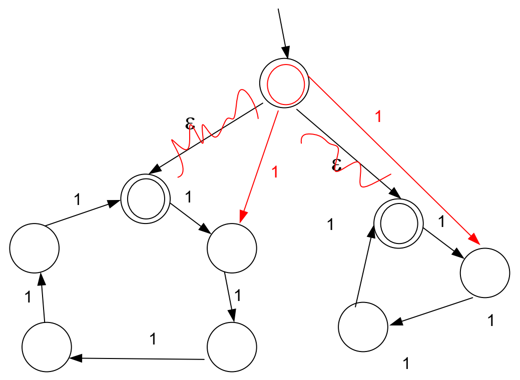


Maybe illustrate subset construction with

➔ (a,b)* ab
➔ $L(M)= \{1^n: 2|n \text{ or } 3|n \}$. Assume an alphabet of {0,1}.

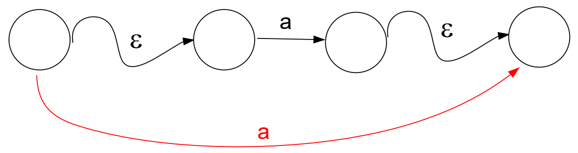
Convert this to a **DFA.** Represent as a **table** and then draw what the table indicates as a DFA.

Now: what to do about ε-arrows?
Let's **eliminate them** before we do the subset construction as defined above.
Example:



More generally,

1) exhaustively add shortcuts:
2) Eliminate the ε-arrows



3) Adjust the final states:
   Finalize any state if there is a ε-path from it to a final state in the original NFA

In this way, no more ε-arrows.

How to write this down formally? Many students have problems with this sort of thing. Here, we need a way to capture the set of states reachable from q by ε-arrows. This can be done as follows:

- Given a DFA $M = (Q,\Sigma,\delta,q_0,F)$ and a state $q \in Q$, let $E(q)$ be the smallest subset of Q such that (1) $q \in E(q)$, and (2) if $s \in E(q)$ then $\delta(s, \varepsilon) \subseteq E(q)$.
- Now, for (3): Let the new final states F' of the machine $M' = (Q,\Sigma,\delta,q_0,F')$ we are constructing from $M = (Q,\Sigma,\delta,q_0,F')$ be $F' = \{q \in Q\colon E(q) \cap F \neq \varnothing\}$.
- Extend E and δ to sets: $E(S) = \cup_{s \in S}\ E(s)$, $\delta(S,a) = \cup_{s \in S}\ \delta(s,a)$,
- For (1): $\delta(q,a) = E(\delta(E(q),a))$

Lecture 4.M

Today
   o Go over Q3
   o Regular languages are the DFA/NFA-acceptable languages
   o Proving that a language is not minimal … or not reguar

Are the regular languages NFA-acceptable? YES,
Answer (1): describe two proofs: (a) how to convert a regular expression into an NFA acceptable languages. (b) Or the more abstract proof that the NFA-acceptable languages include $\varnothing$, {ε},{a} and are closed under union, concatenation, and *. The regular languages are the smallest such set.

Are the NFA-acceptable languages regular? Also YES

Theorem: Every NFA-acceptable language is regular.

Hardest theorem so far. Really the first tricky proof. Show how to do it, by example, on

L = {binary encoding of numbers divisible by 3}

Conversion procedure:

1. First eliminate all e-arrows by protocol already described. Henceforth assume none.
2. Add new start state (no arrows into it) and new final state (no arrows out of it) and definalize every other final state.
3. Repeatedly: select a state q other than the newly added start and final state, and KILL(q)
4. Whenever you create parallel arcs with labels x and y, combine them to (x u y)

To KILL(q):

For each pair p to q to r with the p→q transition alpha, and q→ q labeled beta and q→ r labelled gamma, make a p→r arrow (alpha)(beta *)(gamma). When all such shortcuts added, eliminate state q.

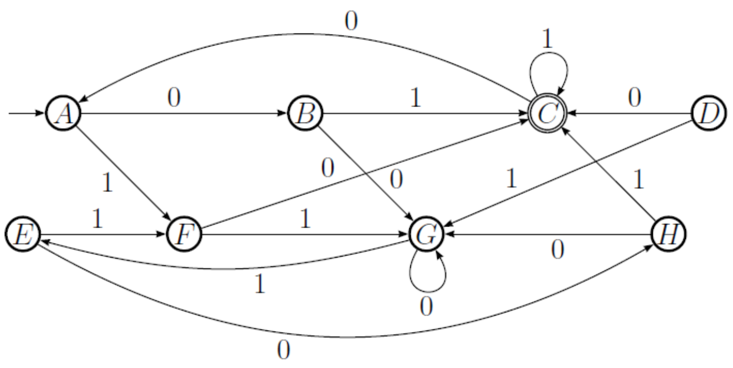
Big parentheses -- not to be covered this year: Minimizing DFAs and Myhill-Nerode

Remember PS1:

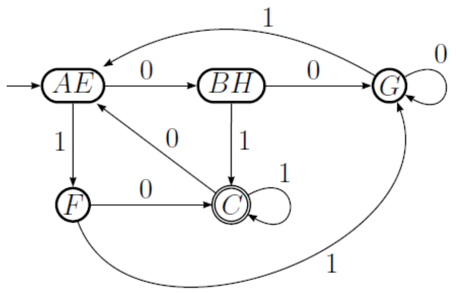
Problem 5. Fix a DFA $M = (Q, \Sigma, \delta, q_0, F)$. For any two states $q, q' \in Q$, let us say that q and q' are *equivalent*, written $q \sim q'$, if, for all $w \in \Sigma^*$ we have that $\delta^*(q, w) \in F \Leftrightarrow \delta^*(q', w) \in F$. Here δ^* is the extension of δ to Σ^* defined by $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, ax) = \delta^*(\delta(q, a), x)$.

(a) Prove that $\sim$ is an equivalence relation.

(b) Suppose that $q \sim q'$ for distinct q, q'. Describe, first in plain English and then in precise mathematical terms, how to construct a smaller (=fewer state) DFA M' that accepts the same language as M.

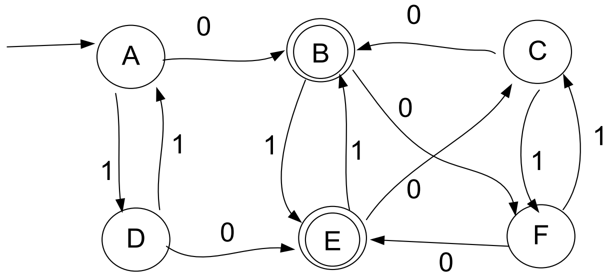


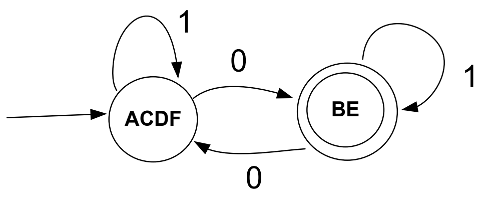
	B	C	E	F	G	H
A	≠	≠		≠	≠	≠
B		≠	≠	≠	≠	
C			≠	≠	≠	≠
E				≠	≠	≠
F					≠	≠
G						≠



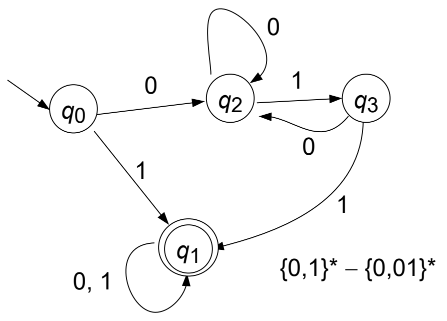
Another example (if needed):

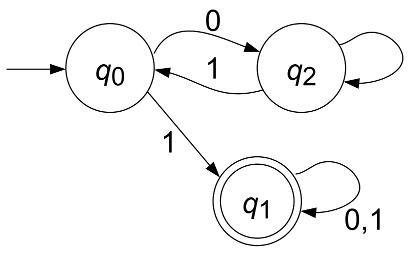
	B	C	D	E	F
A	≠			≠	
B		≠	≠		≠
C				≠	
D				≠	
E					≠



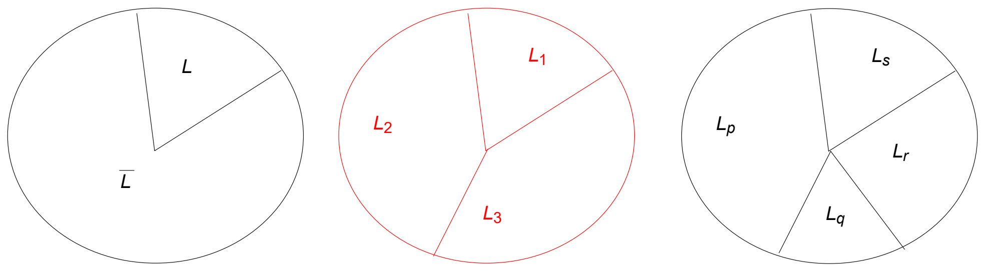


And more examples:





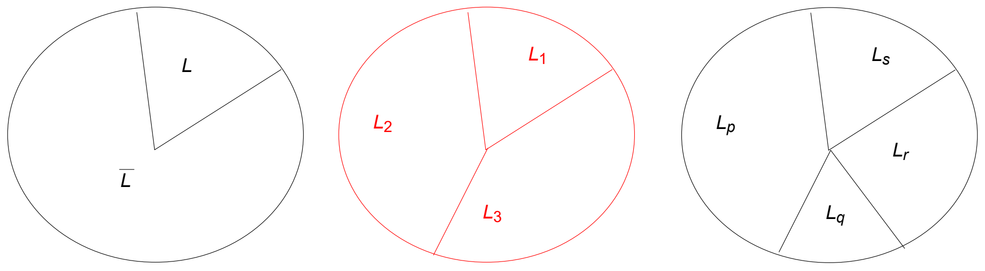
Every DFA partitions the universe of strings into L and its complement…



Partition 2 Partition L Partition M

Let $M = (Q,\Sigma,\delta,q_0,F\,)$ be a DFA. For simplicity, assume it has no unreachable states.
DFA M partitions Σ^* into **two sets**: $L(M)$ and its complement (**Partition 2**).
It **also partitions it into** $\boldsymbol{n}$**=|**$\boldsymbol{Q}$**| sets** – strings that take you to each state $q \in Q$: $L_q=\{x\in\Sigma^*: \delta^*(q_0,x)=q\}$.
Partition M. How do these two partitions related to one another? The second is a refinement of the first.
Block L is just the union of blocks L_q where q is final; block L^C is the union of blocks L_q where q is nonfinal.

End of class last time was explaining something about regular languages



Partition 2 Partition L Partition M

Let $M = (Q,\Sigma,\delta,q_0,F\,)$ be a DFA. For simplicity, assume it has no unreachable states.
DFA M partitions Σ^* into **two sets**: $L(M)$ and its complement (**Partition 2**).
It **also partitions it into** $\boldsymbol{n}$**=|**$\boldsymbol{Q}$**| sets** – strings that take you to each state $q \in Q$: $L_q=\{x\in\Sigma^*: \delta^*(q_0,x)=q\}$.
Partition M. How do these two partitions related to one another? The second is a refinement of the first.
Block L is just the union of blocks L_q where q is final; block L^C is the union of blocks L_q where q is nonfinal.
Finally, M – indeed $L=L(M)$ – partition Σ^* in the following interesting way:

For any language L, define $\sim$ by $x \sim y$ iff $\forall z$, $xz \in L$ **iff** $yz \in L$.

Proposition: $\sim$ is an equivalence relation.

How does partition L related to partition M? Partition M is a refinement of Partition L.

What is happening in our minimization procedure is to combine all the blocks of Partition M until each of these is a block of partition L. The resulting DFA is the unique. What's the DFA corresponding to partition L?

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

$Q = [x]$
$\delta([x], a) = [xa]$
$q_0 = [\varepsilon]$
$F = \{[x]:\ x\in L\}$

Forced to define things this way. Must check that it is actually well-defined.

Myhill-Nerode Theorem (1958): L is regular iff $\sim$ has a finite number of equivalence classes. This number of equivalence classes is the number of states in a smallest DFA for L.
The structure of that DFA is unique (that is, the minimum-state DFA is unique up to the naming of states).

Gives one way of showing that a language L is not regular: show that $\sim_L$ has infinitely many blocks. $L = \{a^n b^n: n \geq 0\}$. Claim: $[a^n]$ and $[a^m]$ are distinct if $n \neq m$. Why? Let $z=b^n$. Then $a^n z \in L$ but $a^m z \notin L$.

Another way to say essentially the same thing: consider the infinitely many strings $s_i = a^i$.
If, for any two of them, $\delta^*(q_0,a^i)= \delta^*(q_0,a^j)$, $i \neq j$, then we have a problem: $\delta^*(q_0,a^i b^i) = \delta^*(q_0, a^j b^i)$, but one is in L and the other is not. This can be generalized:

Proposition: Suppose L is a language and $s_1, s_2,\ldots$ are distinct strings. Suppose for all s_i, s_j there exists x such that one of $s_i\, x$, $s_j\, x$ is in L and the other is not. Then L is not regular.

End Parentheses.

Lecture 4.W

Today:
o Proving DFAs minimal
o Proving languages not-regular
Announcements:
- Dog day?
- Q4 on Friday. Back to in-class, is my assumption. Preferences?
- One more day on regular languages

Minimality of DFAs
Consider the language L = $\{1^{6n}: n >= 1\}$. Unary alphabets. What's the **smallest** DFA (= fewest states) accepting this? Draw the 6-state DFA that does. Claim: there is no smaller DFA that accepts the same language.

Proof. Suppose for contradiction that some 5-state DFA accepted L. Consider the 6 strings
ε, 1, 11, 111, 1111, 11111, and the 6 corresponding states
$\delta^*(q_0,\varepsilon)$, $\delta^*(q_0,1)$, $\delta^*(q_0,11)$, $\delta^*(q_0,111)$, $\delta^*(q_0,1111)$, $\delta^*(q_0,111111)$.
By the PHP, some two of these must coincide. Do example of possibilities, like

$\delta^*(q_0, 1^2) = \delta^*(q_0, 1^4)$. Add two more 1s to each ....
How many cases? C(6,2) = 6*5/2=15. But can do it systematically. If
$\delta^*(q_0, 1^a) = \delta^*(q_0, 1^b)$ for some 0 <= a < b <=5 then look at
$\delta^*(q_0, 1^a 1^{6-n}) = \delta^*(q_0, 1^b 1^{6-b}) = \delta^*(q_0, 1^6)$
not in F in F

The existence of not-regular langauges

Some languages are not regular:
$L = \{a^n\ b^n\text{: } n >= 0\}$

Another approach, have seen before: pigeonhole principle. Review how it works. Now generalize:

The pumping lemma

Let L be a regular language. Then there exists a number p ("the pumping length") such that
$\forall s \in L, |s| \geq p,$
$\exists\ xyz,\ s = xyz,\quad |y| \geq 1,$
$\forall i\ | \geq 0,\quad x\, y^i\, z \in L.$

Give the usual proof …

Lecture 4.F

Today:
o Q4
o Practice showing languages not regular
o Maybe start decision procedures involving regular languages

Announcements:
- Dog day next Wednesday!

Pumping Lemma from last time
$(\forall\ L$, regular$)$
$(\exists\ N)$
$(\forall\ s \in L, |s| \geq N)$
$(\exists\ x, y, z$ s.t. $s = xyz, y \neq \varepsilon,\quad |xy| \leq N)$
// Add this side condition later, but then strengthen it as below
$(\forall\ i \geq 0)$
$x\, y^i\, z \in\ L$

Strong form of PL (stronger still than Sipser's version)

($\forall$ L, regular)
    ($\exists$ N)
        ($\forall$ $s_1\ s\ s_2 \in L$, $|s| \geq N$)
            ($\exists$ x, y, z s.t. $s = s_1\ xyz\ s_2$, $y \neq \varepsilon$)
                ($\forall$ $i \geq 0$)
                    $x\,y^i\,z \in L$

Gives you complete control of what "window" gets pumped: choose any string in L and choose any length-N windows within that string, and you can be sure that the y-portion lives within that window.

0) $L = \{a^n b^n\text{: } n \geq 0\}$.   Prove by PL. Use to motivate strengthened form of PL.
1) $L = \{x \in \{a, b\}^*\text{: } x$ has an equal number of a's and b's$\}$ Then give a totally different proof, using closeure properties.
2) $L = \{ww\text{: } w \in \{a, b\}^*\}$.   Let s = $a^p b\ a^p b$.   (Show too that choices like a^p don't work)
3) $L = \{xy\text{: } x,y \in \{a,b\}^*, |x|=|y|\}$.   ← is **regular**
4) $L = \{x\#y \in \{0,1\}^*\text{: } x$ and y encode binary numbers with y one more than $x\}$. No leading 0's. $\{0\#1, 1\#10, 10\#11, 11\#100, \ldots\}$.   Pumping lemma.

Lecture 5.M

Today:
- o Some more practice with showing languages NOT regular
- o Decision procedures involving regular languages

Announcements:
- Dog day on Wednesday.
- Last lecture on regular languages
-

A. Go over pumping lemma statement – basic form and then strengthened form
Emphasize use is for showing languages NOT regular, not for showing languages regular. Because it gives a property of regular languages, not a characterization of them.

If L is regular then L has property P
The "pumping property" – a sort of **periodicity** property.
This periodicity property of a language: "all sufficiently long strings **pump**".
When does a string s "pump"? It **pumps** if s can be partitioned into pieces $s = xyz$, $|y|>0$, and all "pumped up" $x\,y^{\text{something}}\,z$ remain in L – and $x\,z$ is in L, too.

T/**F**: if L has property P, the pumping property, it is regular.

Not the only way to show a language is not regular. Let's revisit
$L = \{x \in \{a, b\}^*: x$ has an equal number of a's and b's$\}$
We proved it not-regular using the PL. Now do it from closure properties. Assume for contradiction that L is regular. Then $L \cap a^*b^*$ would be, too, because regular languages are closed under intersection. But this is just $\{a^n b^n: n \geq 0\}$, which we know to be not-regular.

How about

More examples:

- $L = \{x \in \{a,b\}^*: x$ has an equal number of a's and b's$\}$ Prove by intersecting with $a^* b^*$.
- $L = \{a^i b^I: i \leq I\}$. 1) PL: $s = a^p b^p$. Pump up within the a-portion. 2) Closure properties: $L^R = \{b^I a^i: i \leq I\}$. Homomorphism: $L^H = \{a^I b^i: i \leq I\}$. Intersect L and L^H: $L^\cap = \{a^i b^i: i \geq 0\}$.
- $L = \{w \in \{0,1\} : w$ is *not* a palindrome $\}$. Use closure under regular languages. hen what we did last time.
- $L = \{w \in \{0,1\}: w$ has an equal number of 01's and 10's$\}$. NO! $0^* \cup 1^* \cup (0^+1^+0^+1^+)^+ \cup (1^+0^+1^+0^+)^+$
- $L = \{w \in \{0,1,2\}: *:$ w has an equal number of 01's and 10's$\}$. First, intersect with (012)*, leaving
  $L' = \{(012)^n (012)^n: n \geq 0)$. Now use PL on the string $s = (012)^p (012)^p$.
- $L = \{xy: x,y \in \{a,b\}^*, |x|=|y|\}$. ← is **regular**
- $L = \{x\#y \in \{0,1\}^*: x$ and y encode binary numbers with y one more than $x\}$. No leading 0's. $\{0\#1, 1\#10, 10\#11, 11\#100, \ldots\}$. Le

Decision questions:

1) Given a DFA M, is $L(M) = \varnothing$?
2) Given a DFA M, $L(M)=\Sigma^*$?
3) Given an NFA M, $L(M)=\Sigma^*$?
4) Given a regular expression α, is $L(\alpha)=\Sigma^*$?
5) Given a DFA M and a word w, is $w \in L(M)$?
6) Given a DFA M, is $L(M)$ infinite? Same question for regular expressions.

Lecture 5W

Dog Day!!

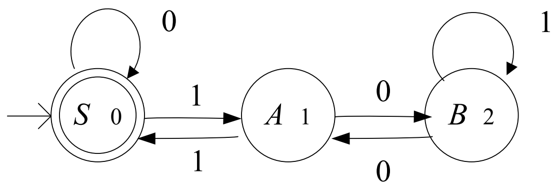
Today:

o A couple more decision procedures
o A whirlwind look at grammars

Decision procedures

A. Given DFA M1 and M2, is L(M1)=L(M2)?
B. Not a decision procedure: given two DFA, either determine that they accept the same language or find a string that distinguishes them: it is in the language of one but not the other.
C. Given regular expressions α_1 and α_2, is $L(\alpha_1) = L(\alpha_2)$?

Grammar-based characterization of the regular languages



$S \to 0S$ **Variables** (or **symbols** or **non-terminals**): S, A, B. **Start symbol**: S **Terminals**: 0, 1
$S \to 1A$ **Rules** (or **productions**) : 7 listed.
$A \to 0B$ Each has a **left-hand-side** and a **right-hand side**
$A \to 1S$ Whole thing is called a **grammar** $G = (V, \Sigma, R, S)$
$B \to 0A$ If we're being totally general, R is any finite subset of $(V \cup \Sigma) \times (V \cup \Sigma)$
$B \to 1B$ This grammar very simple: for each rule, the LHS
$S \to \varepsilon$ is a variable and the RHS a terminal and then a variable; or ε
Such a grammar is called a **right-linear grammar**

Or, a bit more compactly:

$S \to 0S \mid 1A \mid \varepsilon$
$A \to 0B \mid 1S$
$B \to 0A \mid 1B$

Proposition: The regular languages are exactly the languages that have right-linear grammars.

In a context-free grammar (CFG) the right-hand side can mix terminals and variables arbitrarily. The left-hand side is still a variable.

Lecture 5F

Today:
 o go over Q5
 o finish grammars

Announcements:
 o Please indulge my curiosity & sign the attendance sheet today

Last time: described (general, unrestricted) grammars. Looked at a special kind of grammar, a right-linear grammar, and we explained why they exactly characterize the regular languages.
  Given a DFA we turned it into a right-linear grammar
  Given a right-linear grammar, easy to turn it into an NFA

Definition: An (unrestricted) grammar is a 4-tuple $G = (V,\Sigma,R,S)$ where

- V is a finite set ("variables")
- Σ is an alphabet ("terminals")
- R is a finite subset of $(V \cup \Sigma)^* \, V \, (V \cup \Sigma)^* \times (V \cup \Sigma)^*$ ("rules" or "productions")
- $S \in V$ (the "start symbol")

For prettier discourse, we write $A \rightarrow \alpha \in R$, or just $A \rightarrow \alpha$, to mean that $(A, \alpha) \in R$.

Right-linear grammar: LHS: variable RHS: terminal variable or emptystring
CFG: LHS: variable RHS: string of terminals and variables
Unrestricted grammar: LHS & RHS are strings of terminals and non-terminals. But LHS should contain at least one non-terminal

$S \rightarrow \varepsilon$
$S \rightarrow a\,S\,b$

Show how to **derive** strings with this grammar, and that its language is

$L(G) = \{a^n b^n : n \geq 0\}$

- Another example: $L(G) = \{a^n b^N : N > n \geq 0\}$

$S \rightarrow A\,B$
$A \rightarrow a\,A\,b \mid \varepsilon$
$B \rightarrow b \mid bB$

Show derivations for strings, and corresponding **parse trees**.

Conventions: A,B,C,S,T … variables
 a, b, c, …, 0, 1, #, (,), … terminals
 x, y, α, β, strings of variables and terminal – **sentential forms**

Given a CFG G= (V,Σ,R,S), define a relation ⇒ on $(V \cup \Sigma)^*$ (**sentential forms**), the **yields,** or **yields in one step,** relation**:**

$x\, A\, y \Rightarrow x\, \alpha\, y$ if $A \rightarrow \alpha \in R$

Let $\Rightarrow^*$ Let (* over the ⇒; can't do in word; henceforth ➔ be the **reflexive-transitive closure** of ⇒. (Formally define this.

$L(G) = \{x \in \Sigma^*\colon S \Rightarrow^* x\}$ – the language of the CFG G.

L is **context free** if L=$L(G)$ for some CFG G.

A string w is **ambiguously derived** if there are two parse trees (or two leftmost derivations) for it. A CFG G is **ambiguous** if it there is some string that is ambiguously derived.

Ambiguity usually regarded as a "bad" property of a CFG (and a worse property of a CFL). Dangling **if-then-else** problem.
A grammar being ambiguous is usually consider a defect of the grammar. Dangling if problem:
    if B **then if** B' **then** S1 **else** S2

A grammar being ambiguous is usually consider a defect of the grammar. Dangling if problem:
    if B **then if** B' **then** S1 **else** S2

Make two parse trees for this.

Exercise: design a CFG for

- $L(G) = \{a^n b^N\colon N > n \geq 0\}$

$S \rightarrow A\, B$
$A \rightarrow a\, A\, b \mid \varepsilon$
$B \rightarrow b \mid bB$

Exercise: What is the language of the following CFG:

$S \rightarrow \varepsilon \mid AA$
$A \rightarrow AAA \mid a \mid b\, A \mid A\, b$

Claim: it's $\{w \in \{a,b\}^*\colon w$ has an even # of a's$\}$

L is **context free** if L=$L(G)$ for some CFG G.

A machine characterization of CFLs: **PDAs**:

Draw a picture: stacks grow "up".
Formal definition: $M = (Q,\Sigma,\Gamma,\delta,q_0,F)$ where Q is a finite set, Σ,Γ are alphabets, $q_0 \in Q$, $F \subseteq Q$, and

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

Informally describe operation of machine for $L = \{0^n1^n: n \geq 0\}$. Then draw the picture.

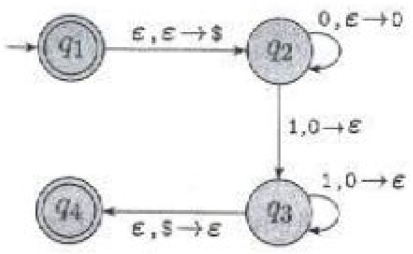
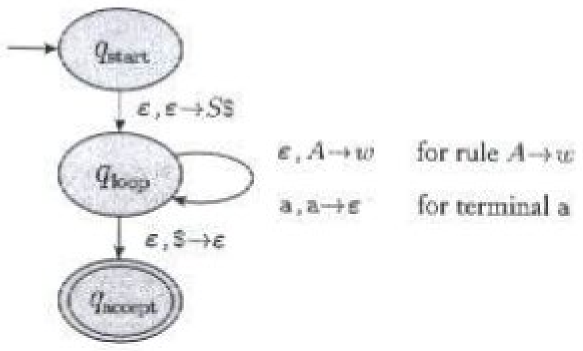


FIGURE 2.15
State diagram for the PDA M_1 that recognizes $\{0^n1^n | n \geq 0\}$

Theorem: The PDAs accept exactly the context-free languages.

Proof. We will do only one direction: every CFL $G = (V,\Sigma,R,S)$ has a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ that accepts it.



Example: Convert the grammar you came up with for

$L(G) = \{a^nb^N: N > n \geq 0\}$

$S \rightarrow A\ B$
$A \rightarrow a\ A\ B \mid \varepsilon$
$B \rightarrow b \mid bB$

into an equivalent PDA.

Thm: There exists an efficient algorithm to decide if a string x in in the language of a CFG G.

Called the CYK (Cocke–Younger–Kasami (CYK) algorithm, circa 1965-1970). A lovely dynamic programming algorithm. Not going to cover this year.

The pumping lemma

A tool for showing a grammar **not** context free.

$\forall$ CFLs L
 $\exists$ a number p
  $\forall\ s \in L,\ |s| \geq p$
   $\exists\ u,v,x,y,z\quad s=uvxyz,\ |vy|\geq 1,\ |vxy| \leq p,$
    $\forall\ i \in \mathbf{N},\quad uv^ixy^iz \in L.$

Example:

- $L = \{a^i\, b^i\, c^i\!: \ i \geq 0\}$ is **not** context free. Draw picture, work out cases.

Corollary: the context free languages are not closed under complement.
Reason: L^c **is** context-free.

- $L = \{ww\!: w \in \{a,b\}^*\}$ is **not** context free. Draw picture, work out cases.
  Here, try $s = 0^p1\ \ 0^p1$ and explain that it does not work.
  Then use $s = 0^p1^p\ \ 0^p1^p$

More closure properties:

	Reg	**CFL**
complement	yes	no
union	yes	yes
intersection	yes	no
concatenation	yes	yes
Kleene closure (*)	yes	yes
Reversal	yes	yes

- Counterexample to **closure under intersection**: $\{a^i\, b^i\, c^i\!: i\geq 0\} = \{a^i\, b^i\, c^j\!: i,j\geq 0\ \} \cap \{a^i\, b^j\, c^j\!: i,j\geq 0\ \}$

Prop: the intersection of a regular language and a context-free language is context free.

Lecture 6M

o Review of Grammars
o Turing machines

Review of grammars: A grammar as a 4-tuple $G = (V,\Sigma,R,S)$. Different kind of rules give different kinds of grammars. Context-free grammars: lefthand side a single variables, right-hand sides are arbitrary; Unrestricted: lefthand sides are arbitrary except for having to contain at least one variable, righthand sides arbitrary; right-linear: lefthand side a single variable, righthand side a terminal then a variable, or else the empty string. Corresponding to the CFL, r.e. languages (as we will define soon); and regular languages. Notion of a derivation and of a parse tree. Notion of ambiguity. There exists a machine characterization: PDA/NPDAs. Simple languages are not context free: $\{a^n\, b^n\, c^n$: n\ge 0$\}$. (Interestingly, the complement of this language is CF, so the CFLs are not closed under complement.)

Turing machines

1936, Alan Turing (born 1912). “On computable numbers, with an application to the Entscheidungsproblem”. David Hilbert, 1928. “asks for an algorithm that takes as input a statement of a first-order logic (possibly with a finite number of axioms beyond the usual axioms of first-order logic) and answers "Yes" or "No" according to whether the statement is universally valid. By the completeness theorem of first-order logic, a statement is universally valid if and only if it can be deduced from the axioms, so the Entscheidungsproblem can also be viewed as asking for an algorithm to decide whether a given statement is provable from the axioms using the rules of logic.

“In 1936 and 1937, Alonzo Church and Alan Turing, respectively, published independent papers showing that a general solution to the Entscheidungsproblem is impossible.”

Draw picture and introduce basic vocabulary

Def: A **Turing Machine** (TM) $M = (Q,\Sigma,\Gamma,\delta,q_0,q_A,q_R)$ is a 7-tuple with

- Q a finite set (“states”)
- Σ an alphabet (“input alphabet”)
- Γ an alphabet (“the tape alphabet”), $\Sigma \subseteq \Gamma$, $\square \in \Gamma - \Sigma$
- δ: $Q \times \Gamma \to Q \times \Gamma \times \{L,R\}$ (“transition function”)
- $q_0, q_A, q_R \in Q$, $q_A \neq q_R$ (“start state”, “accept state”, “reject state”)

Lecture 6W

Today

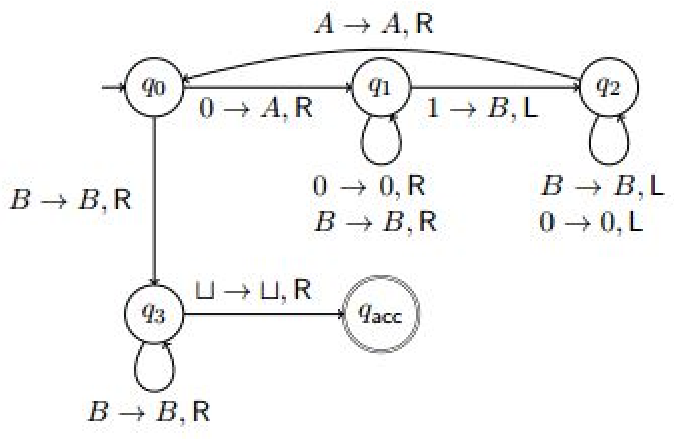
o Turing machines: example and formalization

Reminders

- Quiz on Friday. In person. Grammars and TM basics. Please plan to stay for the whole class. 50 mins – make yourself suffer!!
- Wasn't there someone with a flute?

Review machine informally, first.

Let's make a machine to accept $L = \{0^n1^n\text{: } n \geq 1\}$



Everything unspecified: to q_R

// Let's change the language to $L = \{a^nb^n\text{: } n \geq 1\}$ to make it more mneumonic
// which tape-alphabet character is replacing what input-alphabet character.

Step through operation on some subset of: 01, 0011, 11, 011, 001.
Write machine as a 7-tuple, with a table for the transition function.

Formalizing when the machine accepts:

A **configuration** is an element of $\Gamma^* \times Q \times \Gamma^*$
Write $\alpha\, q\, \beta$ instead of (α, q, β). Convention: reading the first character that's to the right of the state. If nothing is there, it's a blank.

Defining the yields-in-one step relation on configurations

$\alpha\, p\, b\, \beta \vdash \alpha\, c\, q\, \beta$	if $\delta(p, b)=(q, c, R)$	char on right; move right
$\alpha\, p \vdash \alpha\, c\, q$	if $\delta(p, \square)=(q, c, R)$	nothing on right; move right
$\alpha\, a\, p\, b\, \beta \vdash \alpha\, q\, a\, c\, \beta$	if $\delta(p, b)=(q, c, L)$	char on left and right; move left
$\alpha\, a\, p \vdash \alpha\, q\, a\, c$	if $\delta(p, \square)=(q, c, L)$	nothing on right, char on left, move left
$p\, b\, \beta \vdash q\, c\, \beta$	if $\delta(p, \square)=(q, c, L)$	char on right, nothing left, move left left end of tape is a like a wall.
$p \vdash q\, c$	if $\delta(p, \square)=(q, c, L)$	nothing on right, nothing on left, move left

Alternate when tape is two-way infinite

$p\, b\, \beta \vdash q\, \square\, c\, \beta$	if $\delta(p,\square)=(q,c,L)$	char on right; nothing on left; move left
$p \vdash q\, \square\, c$	if $\delta(p,\square)=(q,c,L)$	nothing on right; nothing on left; move left

Lecture 6F

Today
- o Quiz 6. I'd prefer you to stay after! Why??
- o Turing-decidable and Turing-acceptable languages

Last time: A
Given a TM M = , we defined a relation from $M = (Q,\Sigma,\Gamma,\delta,q_0,q_A,q_R)$ we defined a relation $\vdash$ on **configurations**, elements of $\Gamma^* \times Q \times \Gamma^*$ that capture everything relevant about where you're at in a computation.

How do you run M on some string x. You start in configuration $C_0 = (\varepsilon, q_0, x)$. Then you go to the next configuration, C_1, where $C_0 \vdash C_1$. And so on, producing a sequence of configurations $C_0 \vdash C_1 \vdash C_2 \vdash \ldots \vdash C_n \ldots$ We say that $C_0 \vdash^* C_n$.

$\vdash^*$ is the reflexive-transitive closure of $\vdash$. Define this in two ways. First way: as named: Given any relation $\vdash \subseteq A \times A$, can define $\vdash^*$ by insisting that $x \vdash^* x$, and by asserting that whenever $x \vdash^* y$ and $y \vdash^* z$ implies $x \vdash^* z$. That is, the smallest relation containing ~ but also reflexive and transitive.

Alternatively:
$C \vdash^* C'$ if there exists $C_0, C_1, \ldots, C_n$ s.t. $C_0 = C$ and $C_n = C'$ and $C_i \vdash C_{i+1}$ for all $0 \le i < n$.

An alternative definitional approach: define a function next(C) that, given a configuration C, returns the next configuration from C – or, if C is accepting or rejecting, returns NULL.

Key definitions:
An **accepting** configuration: $\alpha\, q_A\, \beta$ for some α, β
A **rejecting** configuration: $\alpha\, q_R\, \beta$ for some α, β
A **halting** configuration: an accepting configuration or a rejecting configuration
M **accepts** *x* if $q_0 x \vdash^* C$ for some accepting configuration *C*
M **rejects** *x* if $q_0 x \vdash^* C$ for some rejecting configuration *C*
M **halts** on input *x* if *M* accepts *x* or *M* rejects *x*
M **loops** or **diverges** on input *x* if *M* neither accepts nor rejects *x*

Talk about the two "ways" that *M* might loop: repeating a configuration, or never repeating any. The latter is in some sense "worse" because it is possible to "see" that you are repeating a configuration – but it might not be possible to know that you are producing a sequence of new configurations that will never repeat.

$L(M) = \{x \in \Sigma: M \text{ accepts } x\}$

M **accepts** *L* if
  $x \in L \iff M$ accepts *x*

L is **Turing acceptable (or r.e.)** if there exists a TM *M* that *accepts L*.

M **decides** *L* if
  $x \in L \Rightarrow M$ accepts *x*
  $x \notin L \Rightarrow M$ rejects *x*

L is **Turing-decidable** if there exists a TM *M* that *decides L*.

Ask some T/F questions:
- If M decides L then M accepts L yes
- If M accepts L then M decides L no
- Every r.e. language is recursive no
- Ever recursive language is r.e. yes
- If M decides L then M always halts yes
- If L is decidable and L(M)=L then M always halts no

Lecture 7M
Today
 o Quiz-6 front-pages
 o Review of recursive/r.e.
 o Alternative models of computation & the Church-Turing thesis
 o Reading from a children's book
  (*Natural Wonders Every Child Should Know* (1912) (Edwin Tenny Brewster)
  https://web.cs.ucdavis.edu/~rogaway/classes/120/spring14/brewster.pdf

How powerful are Turing Machines?

The short answer: very powerful.
In some way, at least as powerful as general-purpose computers.

Souped-up machines:

1. Extra tracks
2. Two-way infinite tapes
3. Extra heads
4. Extra tapes
5. Two-dimensional tapes
6. RAMs
7. Unrestricted grammars
8. NTM

Church-Turing Thesis

That which a TM can compute coincides with **Our intuitive notion of what <u>is</u> computable**

Or, said differently, TMs compute all and only what is "effectively calculable".
"A function is said to be 'effectively calculable' if its values can be found by some purely mechanical process. Although it is fairly easy to get an intuitive grasp of this idea, it is nevertheless desirable to have some more definite, mathematically expressible definition. Such a definition was first given by Gödel at Princeton in 1934... These functions were described as 'general recursive' by Gödel... Another definition of effective calculability has been given by Church... who identifies it with lambda-definability. The author [i.e. Turing himself] has recently suggested a definition corresponding more closely to the intuitive idea... It was stated above that 'a function is effectively calculable if its values can be found by a purely mechanical process.' We may take this statement literally, understanding by a purely mechanical process one which could be carried out by a machine... The development of these ideas leads to the author's definition of a computable function, and to an identification of computability [in Turing's precise technical sense] with effective calculability. It is not difficult, though somewhat laborious, to prove that these three definitions are equivalent. [4]" -- Turing'S PhD thesis, 1939

We will use the Church-Turing thesis **implicitly** by saying that when you describe a procedure in a way that I deem a clear algorithmic description, you have described something Turing-computable. And more strongly, too: if you describe an effective computation that decides something, you've shown it Turing-decidable; if you describe an effective computation that recognizes something, you've shown it Turing-recognizable.

Digital Modeling Thesis

That which a TM can compute coincides with **That which a general-purpose digital computer can compute (with unlimited time and memory)**

Lecture 7W

Today

- o A bit more of the book we were reading
- o Arguments for/against the Church-Turing thesis
- o Undecidability of A_{TM}

Announcements:

- Q7 opens tomorrow noon, closes Friday noon
- Fatima is sick and cancelled her discussion section.
  I will be holding office hours 3:30 – 5:00 pm (3009 Kemper)

Arguments for/against the Church-Turing thesis

Briefly review souped-up models. There are also paired-down models, including:

1. 2-counter machine
2. 2-tag systems (Emil Post, 1943; Turing-Complete shown by Wang (1963), Cocke & Minsky (1964); Woods 2006)
3. Rule 110 (Cook, 2004)

Example: 2-tag machine [Emil Post, 1943] (Turing-complete: Wang 1963, 1964)
http://www.ini.uzh.ch/~tneary/WoodsNeary-FOCS06.pdf

Alphabet Σ, rules $R: \Sigma \to \Sigma^*$

At each step you replace the leftmost two characters of the input word w with $w[3:] \,||\, R(w[1])$. Machine **halts** if 0 or 1 symbols left on tape. Initial input $a_1 \dots a_n$ is encoded as $a_1\, a_1 \dots a_n\, a_n$. We can say that it **accepts** if it transforms w to 1 and **rejects** if it halts without accepting. Amazing claim: that this is again Turing-equivalent.

Restate the Church-Turing thesis.

Arguments for

Simulation results
Test of time
Diversity of
equivalent models

Arguments against (or at least coloring our interpretation of the Thesis)

Lack of probabilism
Lack of "body"
Super-Turing computation ("hypercomputation"), like
oracle machines or the
Blum-Shum-Smale model:
(a RAM but the registers can store reals and you can perform arithmetic on them. Can compare reals.)
No graphics
No internet
No clock
Can't ignore time or memory
ML / ChatGPT argument

Undecidability of A_{TM}

Theorem [Turing 1936; Church 1936; Cantor (diagonalization) 1891]
$A_{TM} = \{<M, w>$: TM M accepts $w\}$ is **not** decidable.

Assume for contradiction that this language is (was? were?) decidable

Machine M_1
$<M, w>$ … **Accept** if M accepts w
Reject if M does **not** accept w

Machine M_2
$<M>$ … **Accept** if M accepts $<M>$
Reject if M does **not** accept $<M>$

Machine M_3
$<M>$ … **Accept** if M does **not** accept $<M>$
Reject if M accepts $<M>$

Consider what happens if we feed this machine its own description
Machine M_3
$<M_3>$ … **Accept** if M_3 does **not** accept $<M_3>$ *absurd*
Reject if M_3 accepts $<M_3>$ *absurd*

Contradiction! We conclude that our original assumption, that the language was decidable, is in error.

This is a deep and interesting proof. Think about it! Teach it to your mom. Teach it to your nieces/nephew. A child can totally understand it (adults have more trouble).

Lecture 7F
Today
- o Four-possibilities theorem
- o Classification guesses
- o Turing-computable functions

Announcements:
- Next week is reductions. Be caught-up and ready to think!!

Prop: The recursive languages are closed under complement.

Def: If L^c is r.e. we say that L is **co-r.e.**

A_{tm}: **re?** co-r.e.? neither

$L = \{<M>: L(M) = \emptyset\}$ co-r.e.

Machine M_L to accept L:

```
The machines input is <M>.
Let $w_0, w_1, \ldots$ be an enumeration of all strings (over some understood alphabet)
for n = 1 to infinity do
    Run M on each $w_0, w_1, \ldots\ w_n$ for n steps
    If any of these accept by that time: ACCEPT   // the input <M>
```

Proposition: If L is r.e. and co-r.e. then L is recursive.

Proof: Let M1 and M2 be machines that accept L and L^c.

```
To decide L, on input x:
    For n = 1 to infinity do
        Run M1 on x for n steps. If M1 accepts within this time: ACCEPT
        Run M2 on x for n steps. If M2 accepts within this time: REJECT
```

Observation: This procedure, even though it looks like it might potentially run forever, does not: it always terminates, and with a correct decision on x's presence in L.

Four-Possibilities Theorem

Corollary: Every language L is either decideable, r.e but not co-r.e., co-r.e. but not re, or neither r.e. nor co-r.e.

Draw a picture. Ask if it is "to scale". Point out that there are uncountably many languages, but only countably many r.e. (or co-r,e, or decidable. languages.

Corollary: A_{TM} is **r.e. but not decidable**
\overline{A_{TM}} is **co-re. but not decidable**

Lecture 8M

Today
- o Classification guesses
- o Turing-computable functions
- o Many-one-reductions and their properties

Announcements:
- This Friday quiz online; next Friday in person; following Friday is the final!!

First review the 4-possiblities theorem and the two interesting languages we can place in that picture: ATM and its complement.

Classification guesses

A_{TM} = {<*M*, *w*>: TM *M* accepts *w*}

NONEMPTY = {<*M* >: TM *M* accepts *some* string *w*}

CFLALL = {<*G*>: CFGs *G* can generate all strings, L(G)=Sigma*}

// need a fact that we didn't show but that I'd like students to know: that there's an algorithm to decide if x is in the language of a CFG G. Even in the strong sense, that there exists an algorithm M such that M decides of <G,x> if x in L(G). Not obvious. Lookup Cocke–Younger–Kasami algorithm (CYK or CKY).

CFLEQ = {<G_1, G_2>: CFGs G_1 and G_2 accepts the same language}

FINITE = {<*M* >: TM *M* accepts only finitely many strings}
Using the undecidability of A_{TM} or its complement to show that other languages are undecidable, not r.e., or not co-r.e.

Many-one reductions.

First we will need a notion of a **Turing-Computable function**

A function *f*: $\Sigma^* \to \Sigma^*$ is **Turing-computable** (or just **computable**) if there exists a TM M such that, for every $x \in \Sigma^*$
$(\varepsilon, q_0, x) \vdash^* ((\varepsilon, q_A, f(x))$.

A function that is not computable is **uncomputable**.

Example: Increment, $<x,y> \mapsto x+y$, , $<x,y> \mapsto xy$, etc.

And example of an interesting function that is **not** Turing-computable

Let $B(n)$ = the maximum number of 1's that a halting *n*-state TM can print on a tape before halting. Need to fix alphabet. "Busy Beaver Function"

Or this variant:
Let $\underline{b}(n)$ = the maximum number of steps that a halting *n*-state TM can run in when applied to an initially blank tape.

Or this variant
Let β (<*n*, *m*>) = the maximum number of steps that a halting *n*-state TM can run in when applied to an input containing *m* characters.

All such functions are not Turing-computable.

Proof that β is not Turing-computable: Suppose to the contrary that $\beta(n,m)$ were Turing-computable. Let's use it to decide Atm. Given $<M, x>$, run M on x for β (#states-in-M, $|x|$) steps. If it accepts within that time, **accept**. If it rejects within that time, **reject**. If it hasn't halted within that time, *again* **reject** (because M will never halt). \
This says something pretty interesting about computation: that a function can grow so rapidly that this fact alone makes it uncomputable.

Key definition: One of the two most interesting definitions in the course.

$A \leq_m B$ if a Turing-computable function f such that $x \in A$ iff $f(x) \in B$.

(Remind the students what a T-C function is.)

Draw picture.

Proposition: $\leq_m$ is reflexive and transitive.

Proposition: If $A \leq_m B$ then
A recursive $\Leftarrow$ B recursive
A r.e. $\Leftarrow$ B r.e.
A co-r.e. $\Leftarrow$ B co-r.e.

Give proofs.

Lecture 8W
Today
o Using many-one reductions

Announcements:
- This Friday quiz online; next Friday in person; following Friday is the final
- Review session Wednesday June 7, 7-9 pm, sound ok? Could be Tuesday, could be earlier.

Repeat definition of $\leq_m$ of and proposition about how decidability and acceptability propagate.

Corollary: If $A \leq_m B$ then
A not recursive $\Rightarrow$ B not recursive
A not r.e. $\Rightarrow$ B not r.e.
A not co-r.e. $\Rightarrow$ B not co-r.e.

Strategy:
To show that a language
- L not r.e., reduce a not-r.e. language to it – say \overline{A_{TM} } $\leq_m L$
- L is not co-r.e., reduce a not co-r.e. language to it – say $A_{TM} \leq_m L$
- L is not decidable, do *either* of the above.

Let's go back:

FINITE = {<*M*>: TM *M* accepts a finite language}

Claim: this langauge is undecidable. In fact, it's neither r.e. nor co-r.e.

- overline{A_{TM}} $\leq_m$ FINITE //so FINITE is not r.e.

<M, w> $\mapsto$ *<M'>* *computed by a T-C f such that*
M doesn't accept *w* $\Rightarrow$ $L(M')$ is finite
M accept *w* $\Rightarrow$ $L(M')$ is infinite

```
M' on input x:
    Run M on w
    If M accepts w, then accept
    reject
```

If *M* doesn't accept *w* then $L(M') = \varnothing$, which is finite.
If *M* accept *w* then then $L(M') = \Sigma^*$, which is infinite.

- $A_{TM} \leq_m$ FINITE –so FINITE is not co-r.e.

<M, w> $\mapsto$ *<M'>* *computed by a T-C f such that*
M accepts *w* $\Rightarrow$ $L(M')$ is finite
M doesn't accept *w* $\Rightarrow$ $L(M')$ is infinite

```
M' on input x:
    Run M on w for |x| steps.
    If it accepts within that amount of time, reject
    Otherwise, accept
```

Now if *M* accepts *w* then it does so in some number of steps *N,* and $L(M')$ = all strings of length less than *N,* a **finite** set.

If *M* does **not** accept *w* then $L(M') = \Sigma^*$, an **infinite** set.

BTHP = {<M>: TM M halts on blank tape}

Prove that $A_{\text{TM}} \leq_m$ BTHP.

NONEMPTY = {<M>: TM M accepts *some* string w}

r.e. but not recursive.

- $A_{\text{TM}} \leq_m$ NONEMPTY – so FINITE is not co- r.e.

<M ,w> $\mapsto$ M' by a T-C. function f:

M accepts w $\Rightarrow$ M' that accepts some string w , i.e., $L(M') \neq \varnothing$
M !accepty w $\Rightarrow$ M' that accepts no string w , i.e., $L(M') = \varnothing$

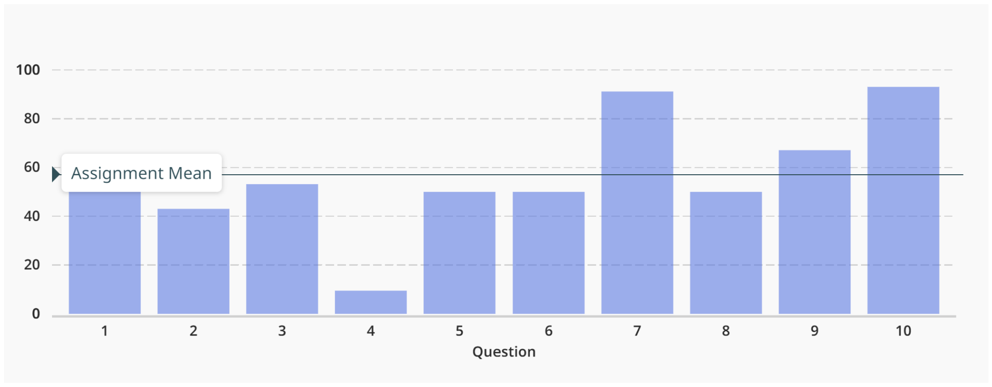
```
M' on input x:
        Run M on w.
        If M accepts w, then accept x
        reject x
```

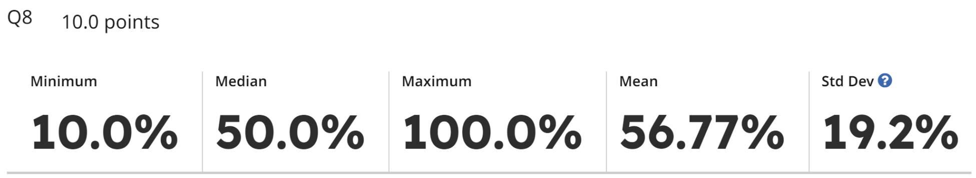
If M accepts w then $L(M') = \Sigma^*$, so $L(M') \neq \varnothing$
If M !accept w then $L(M') = \varnothing$

Lecture 8F

Today
o More reductions and undecidable problems (last computability lecture)

Announcements:
o Review session Tues 6/6 6pm 66 Rslr

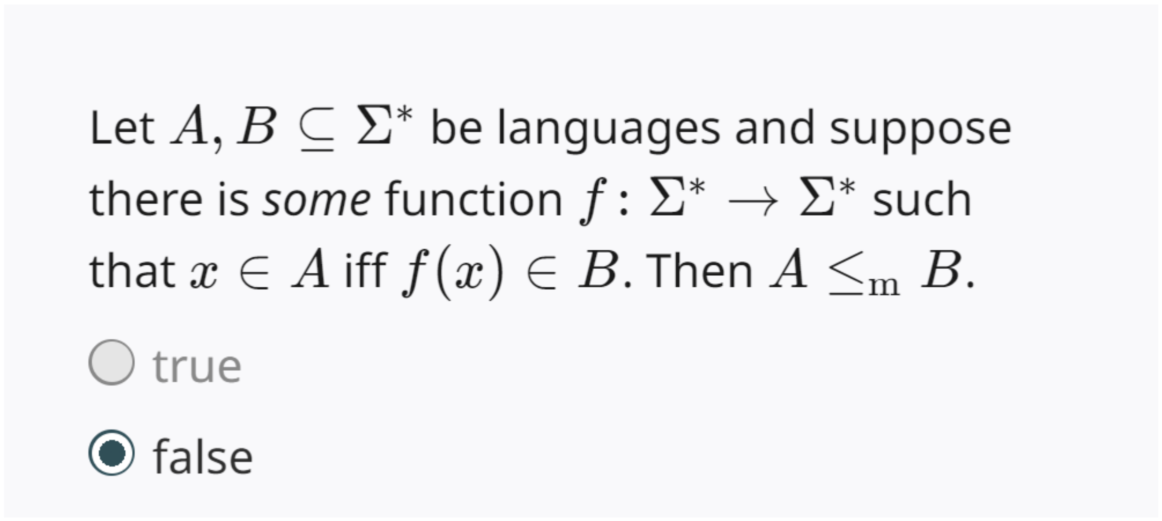



Q8 10.0 points

Minimum	Median	Maximum	Mean	Std Dev
10.0%	50.0%	100.0%	56.77%	19.2%

The above is totally auto-graded, 0-or-1 for each problem, but I might go back and give partial credit on a couple of the select-all questions.

Problem 4 – 9% correct on a T/F question!


Let $A, B \subseteq \Sigma^*$ be languages and suppose there is *some* function $f : \Sigma^* \to \Sigma^*$ such that $x \in A$ iff $f(x) \in B$. Then $A \leq_m B$.

true [✓ false]

VIRUS = {*P*: *P* is a C-program that spawns a shell; does a "`cd /; rm -rf *`" }

It is r.e. Guess it's not decidable. So the guess amounts to saying it's **not** co-r.e.

$A_{TM} \leq_m$ VIRUS

$<M, w> \mapsto P$ by a T-C f
M accepts $w \Rightarrow$ P spawns a shell and does the dreaded command;
M !accept $w \Rightarrow$ P doesn't spawn a shell and do the dreaded command

Have *P* emulate the running of *M* on *w*.
As it does so, your program does **not** spawn a shell a do the dreaded command.
If *M* accepts *w*, then have *P* spawn a shell and do the `cd /; rm -rf *`

`REG = ` {<*M*>: *M* is a TM and *L*(*M*) is regular }
`FINITE` $\leq_m$ `REG`

$<M> \mapsto <M'>$
$L(M)$ is finite $\Rightarrow$ $L(M')$ is regular
$L(M)$ is infinite $\Rightarrow$ $L(M')$ is not regular

```
M' on input x:
  if $x \notin \{a^n b^n : n \geq 0\}$, REJECT
  Parse x as $a^n b^n$
  // dovetail to try to find n strings in L(M):
       For i = 1 to infinity do
            Run M on each string of length at most i for i steps
            If find n different strings accepted, then ACCEPT
```

If $L(M)$ is finite, say $|L(M)|= N$, then $L(M') = \{a^n b^n : N \geq n\}$
which is a **finite** set, and therefore a **regular** language

If $L(M)$ is infinite, $L(M')$ is $\{a^n b^n : n \geq 0\}$, which is **not** a regular language

OMIT proof for 2023, but mention this language,
classify it, and possibly hint as to tht eproof. CFGALL = {<*G*>: *G* is a CFG and *L*(*G*)=Σ* }

co-r.e. Not r.e. $!A_{TM} \leq_m$ CFGALL

$<M,w> \mapsto$ G by a T-C function f
If *M* doesn't accept w then L(G) = S*

If M does accept w then $\exists\, x$ that's **not** in $L(G)$ – ie, $L(G)$ has a "hole"

Given M and w, we will define a language ACC-COMP-WORDS$_{M,\,w}$. The langauge is the set of all strings

$$C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_t$$

with each $C_i \in (Q \cup \Gamma)^*$ where

- $C_0 = q_0\, w\, \Box^{\,i}$ for some i
- Each $|C_i| = |C_{i+1}|$
- Each $|C_i| \vdash |C_{i+1}|$
- C_t is an accepting configuration

Then

- !ACC-COMP-WORDS is CF
- Given $<M,w>$, we can produce a CFG G for ! ACC-COMP-WORDS
- If M accepts w then there is some element in ACC-COMP-WORDS, so ! ACC-COMP-WORDS$\neq\emptyset$
- If M !accepts w then there is no element in ACC-COMP-WORDS, so ! ACC-COMP-WORDS$=\Sigma^*$

If you know that CFGALL is undecidable: CFGALL $\leq_m$ CFGEQ

Additional undecidable problems:

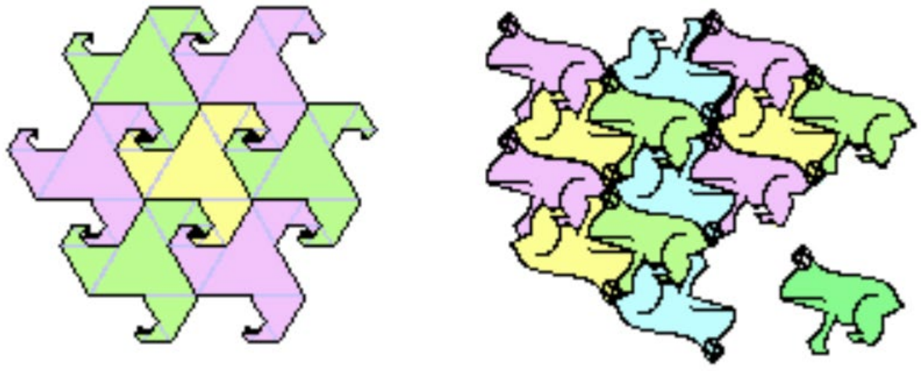
DIOPHANTINE
Instance: a polynomial $P(x_1, \ldots, x_n)$ / $\mathbf{Z}$
Question: Does it have an integer root? That is, a setting of each variable to an integer value where the whole polynomial comes out to be 0.

PCP (Post Correspondence Problem)
Instance: binary strings $x_1, \ldots, x_n$; $y_1, \ldots, y_m$
Question: does there exists incices i1, …, is; j_1, …, j_t such that
x_{i_1}, …, x_{i_s} = y_{j_1}, …, y_{j_t}

Note the different format for writing problem. Could also have written as
{<x_1, …, x_n, y_1, … y_m>: blah } but as the conditions get complicated, this is just a bit too cumbersome.

TILING PROBLEM
Instance: A number n and a set of n "tile types" $(L_1, T_1, R_1, B_1), \ldots, (L_n, T_n, R_n, B_n)$ each of these subscripted variables a number, thought of as the "colors" of the left, top, right, and bottom of 1x1 tiles.
Question: Can the plane [or the upper quadrant of the plane] be properly tiled using only tiles of these types. A proper tiling gives neighboring tiles the **same** color on the common edge.

A sort of generalization of geometric tiling of the plane, like



Lecture 9W

Today: o Complexity Theory:
The classes P, NP

Announcements: o Review session Tues 6/6 6pm 66 Rslr

The Class P

Definition: For *M* a TM and *x* a string,
$\text{TIME}_M(x)$ = # of steps that *M* takes on input *x*, or ∞ if *M* diverges on *x*.

Definition: A language $L \in \mathbf{P}$ if there's a TM *M* and a polynomial *poly* such that *M* decides *L* and $\text{TIME}_M(x) \leq poly(|x|)$.

Polynomial-time and the class **P** is routinely understood as formalization of that which has a "practical algorithm." This certainly isn't an exact formalization of that concept, insofar as there are practical algorithms (Simplex is the most famous) that do not run in polynomial time, while there are impractical algorithms that do. Nonetheless, it has worked surprisingly well.

One reason we like **P** as a mathematical stand-in for practicality is the robustness of this measure.

Polynomial-time modelling thesis
Reasonable models of computation are *equivalent* up to a polynomial slowdown.

To the extent that this thesis is true, **P** is the same no matter what reasonable model of computation we assume (RAM model, TM, etc.)

The polynomial-time modelling thesis is increasingly debatable by the emergence of **quantum computers**. There are still "toys", from what I understand, not able to solve any practical problems any better than conventional computers. Will this remain so? It is unclear. In any case, it is possible to change the model of computation to capture polynomial time on a quantum computer, which is formalized with the class **QP**. If a problem is in **QP** (or its optimization counterpart) but not ostensibly in **P** you might worry about using it as a basis for cryptographic hardness.

Some examples of languages in **P**: A DFA M accepts a given string w. Two DFAs accept the same languages/ A CFG accepts a given string. G is a connected graph. G is bipartite

Not obviously in **P**: Two regular expressions denote the *same* language. A Boolean formula ϕ is **satisfiable** (=there is some way of setting its variables so that it comes out true). A graph G is **3-colorable** (there is some way to paint its vertices using only three colors so that no adjacent vertices get the same color).

When we describe problems like these, it is often convenient to switch notations, as illustrated here:

GSAT
INSTANCE: A Boolean formula ϕ.
QUESTION: Is ϕ satisfiable (that is, is there a way to set its variables to 0/1 so that the formula evaluates to 1?

Just a just different notation than writing GSAT = $\{<\phi>: \phi$ is a satisfiable Boolean formula$\}$. The advantage is that we have more lines of text without being inside the open set notation. Also, we stop pounding on the reader with the fact that everything must ultimately be string-encoded if we are feeding it to some model of computation.

The class NP

Def. A language $L \in$ **NP** if it has a polynomial-time **verifier**, which is a TM V such that

- $x \in L \Rightarrow V(x, c)$ accepts for some c.
- $x \notin L \Rightarrow V(x, c)$ rejects for all c
- for some polynomial *poly*, $\text{TIME}_V(x, c) \leq poly(|x|)$

The string c is a **certificate** that establishes that $x \in L$. You can think of it as a succinct proof of the fact. Why is it succinct? Well, the TM V would not even have time to read the certificate if it had super-polynomial length, whence there is no loss in generality in just assuming that its length is polynomially bounded. For this reason, we sometime call c a **succinct** certificate.

You should notice that **P** $\subseteq$ **NP**. Why? Because the algorithm M that decides a language $L \in$ **P** already verifies membership in polynomial time without needing any certificate.

Explain that GSAT $\in$ **NP**: the certificate could be the satisfying assignment. The certificate for graph 3-colorability, G3C, could describe a valid coloring. The certificate for two regular expressions denoting the same language? Well, that one doesn't seem to have one. It would appear that that language, REGEQ, is not in **NP**.

It seems that NP is more than P – that some problems are just easier to verify than to decide. For example, GSAT and G3C are language that seem to be in the difference. But this is not known. It is considered the biggest open problem in computer science whether **P** $\subsetneq$ **NP**, as most people believe, or if, instead, **P** = **NP**.

It is a practical question, too. If you have a problem that you'd really like to solve, but you can't find a polynomial-time solution to, how do you know if there is no polynomial-time solution or if instead, you're just not clever enough? When should you stop working on trying to find a polynomial-time algorithm and decide, instead, that the problem probably doesn't admit one.

Lecture 9F
Today: o Polynomial-time reductions, $A \leq_p$ and NP-Completeness
Announcements: o Review session Tues 6/6 6pm 66 Roessler

Polynomial-time reductions

Polynomial-time computable function: A function f is polynomial-time computable if there is a polynomial poly and a TM M that computes f where $\text{TIME}_M(x) \leq \text{poly}(|x|)$.

Polynomial-time reductions

Def: Let A and B be languages over the same alphabet. We say that A polynomial-time reduces to B, written $A \leq_p B$, if there's a polynomial-time computable function f such that $x \in A$ iff $f(x) \in B$.

In other words, this is just a many-one reduction except that we insist that the reduction itself (i.e., the function mapping A-instances to B-instances and not-in-A-instances to not-in-B-instances) needs to be computable in polynomial time.

Mirroring what we did before,

Prop: If $A \leq_p B$ and $B \in$ **P** then $A \in$ **P**.
Or, taking the contrapositive, if $A \leq_p B$ and $A \notin$ **P** then $B \notin$ **P**.

You can interpret this as saying that B is at least as hard as A.

Prop: If $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$.

That is, polynomial-time reductions are transitive. So are many-one reductions. Indeed the argument is the same, except that we add in the observation that the composition of polynomials is a polynomial.

The notion of NP-completeness

Here is one of the most beautiful and important definitions in all of computer science.

Def [Cook 1971, Levin 1973]: A language L is **NP-complete** if

1. $L \in \mathrm{NP}$, and
2. For all $A \in \mathrm{NP}$, $A \le_{\mathrm{p}} L$.

In other words, L is a **hardest** language in **NP.**

The second condition is in our definition is called **NP-hardness**. We are saying that L is at least as hard as any problem in NP.

Prop: If L is NP-complete and $L \in \mathbf{P}$ then $\mathbf{P{=}NP}$.

Interpreting this, a way to evidence that L does **not** have a polynomial-time decision procedure is to show that L is NP-complete. If you do this, you are showing that anyone who can provably solve L in polynomial time has proven that **P=NP**, resolving what is arguably the biggest open question in computer science. They would have proven not only that L has a polynomial-time solution, but so does GSAT, G3C, TSP, … . If you think that those well-studied problem do not have a polynomial-algorithm, then you are obliged to think that L doesn't, either.

Proof: Give it, employing the idea that the composition of a polynomial-time computable reduction and a polynomial-time decision procedure gives a polynomial-time decision procedure. Because the composition of polynomials is polynomial.

A pictorial explanation on the above definition, showing the hardest languages in NP

Theorem [Cook, Levin]: There exists an NP-Complete language. In fact, CIRCUIT is NP-complete.

CIRCUIT
INSTANCE: A Boolean circuit C with designated input wires $X_1, \ldots, X_n$ and a single designated output wire Y.
QUESTION: Is there a 0/1 choice for input wires $X_1, \ldots, X_n$ of C that makes the output wire Y come out 1?

Proof: next time.

Let's just imagine that we've proven this – assume it. How can we now show that some other language L is NP-complete?

1) Show that L NP.
2) show that $A \le_{\mathrm{p}} L$ for some NP-complete language L.

One you believe that CIRCUIT is NP-Complete, we can start showing that lots of other languages are by reducing known NP-Complete languages to them. It gets easier as you go, to, because you have more problems to start from. But it also gets harder, as you have more choices for starting points.

GSAT
INSTANCE: A Boolean formula ϕ.
QUESTION: Is ϕ satisfiable (that is, is there a way to set its variables to 0/1 so that the formula evaluates to 1?

3SAT
INSTANCE: A Boolean formula ϕ in 3CNF: the formula is the AND of "clauses"; each clause is the OR of three "literals"; each literal is a variable or it's completement; the variables used in each clause are distinct.
QUESTION: Is ϕ satisfiable?

It is generally advantageous to have the problem you're reducing from be more structured / constrained; this makes it easier to use it as the starting point of a reduction.

Lecture 10M
Today: o Showing some languages NP-Complete
Announcements: o Review session: Tues 6/6 @ 6pm in Roessler66
o Final: Fri 6/8 @ 10:30 am in (sid%2)? Giedt1003: Roessler66

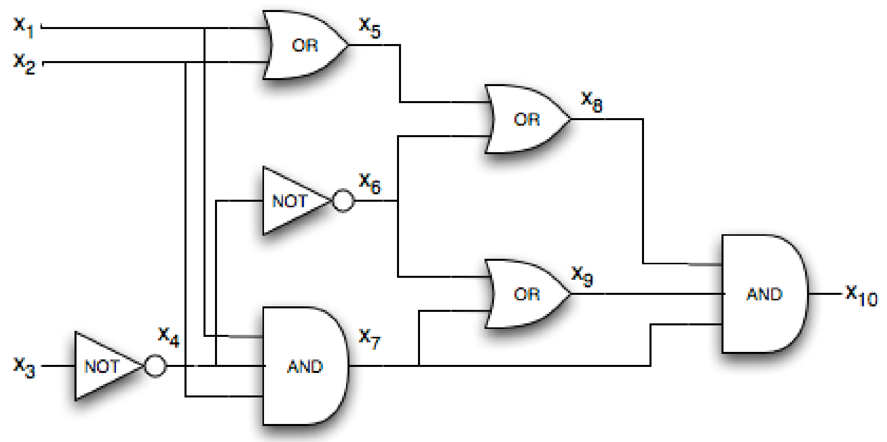
Review: P, NP, many-one reductions, NPC.
NP-Completeness as a way to evidence that a decidable problem is hard – that it *can't* be solved in a reasonable amount of time.

Last time: we claimed that CIRCUIT is NPC, which is the Cook-Levin theorem. For now, let's just assume that. Then:

1) GSAT is NP-Complete
2) 3SAT is NP-Complete
3) G3C is NP-Complete
4) CLIQUE is NP-Complete [discussion section]

All of these are easily seen to be in NP.
To show GSAT is NP-Complete,
show that CIRCUIT $\leq_p$ GSAT

by example:



would have the corresponding formula

$$\varphi = x_{10} \wedge (x_4 \Leftrightarrow \neg x_3)$$

$$\wedge (x_5 \Leftrightarrow (x_1 \vee x_2))$$

$$\wedge (x_6 \Leftrightarrow \neg x_4)$$

$$\wedge (x_7 \Leftrightarrow (x_1 \wedge x_2 \wedge x_4))$$

$$\wedge (x_8 \Leftrightarrow (x_5 \vee x_6))$$

$$\wedge (x_9 \Leftrightarrow (x_6 \vee x_7))$$

$$\wedge (x_{10} \Leftrightarrow (x_7 \wedge x_8 \wedge x_9))$$

To show that 3SAT is NP-Complete,
sthat CIRCUIT $\leq_p$ 3SAT.

First replace everything with 2-input NAND gates. Then we need a "gadget" – a way to somehow write a NAND gate as a piece of 3CNF formula. You probably know how to do with for 3DNF – do it by example. For 3CNF, you can do something analogous. You end up with:

A	B	C	Is C the NAND of A and B? 0 = no, 1 = yes
0	**0**	**0**	**0**
0	0	1	1
0	**1**	**0**	**0**
0	1	1	1
1	**0**	**0**	**0**
1	0	1	1
1	1	0	1
1	**1**	**1**	**0**

`Nand(A,B,C)=`$(A \text{ or } B \text{ or } C)(A \text{ or } \overline{B} \text{ or } C)(\overline{A} \text{ or } B \text{ or } C)(\overline{A} \text{ or } \overline{B} \text{ or } \overline{C})$
`is satisfiable` **`iff`** `C = A NAND B`

Idea: the first clause is satisfiable except for the first red row above; the second clause is satisfiable except for the second red row above; etc. It is like and'ing together the four clauses cl1, cl2, cl3, cl4

A	B	C	cl1	cl2	cl3	cl4
0	0	0	0	1	1	1
0	0	1	1	1	1	1
0	1	0	1	0	1	1
0	1	1	1	1	1	1
1	0	0	1	1	0	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	0

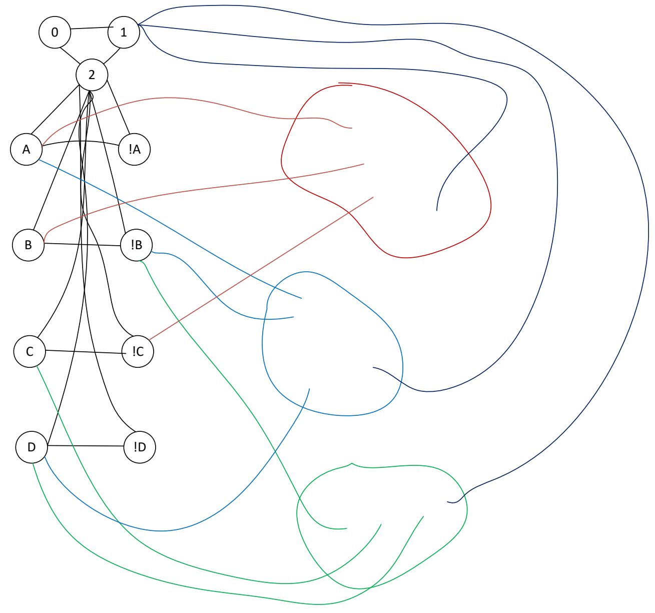
Reduction: Given the circuit of NAND gates, each wire labelled with a variable, include four clause Nand(var1,var2,var3) or each gate. You will also need to conjoin Y which, to do in 3CNF, we can accomplish as (Y or A or B) (Y or A or B) (Y or A,B) (Y or !A or !B)

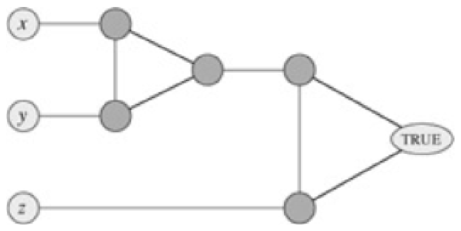
G3C

INSTANCE: A graph G
QUESTION: Can the vertices of G be painted with 3 colors such that no adjacent vertices are given the same color.

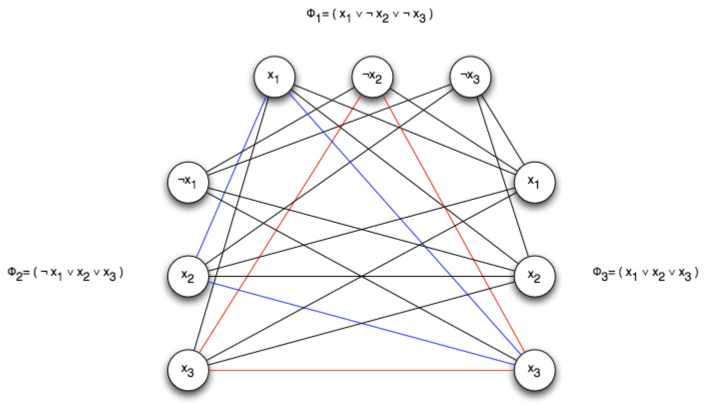
Reduction: Start with a formula, say: (A or B or !C) (A or !B or D)(!B or C or D).
Go through reasoning to try to "compile" it into a graph where the graph is going to be 3-colorable iff the formula is satisfiable.

(A or B or !C) (A or !B or D) (!B or C or D)





CLIQUE -- maybe save this one for discussion section
Instance: A graph G and a number k ≤ |V(G)|
Question: Does G have a clique of size k?


$\Phi_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$
$\Phi_2 = (\neg x_1 \vee x_2 \vee x_3)$
$\Phi_3 = (x_1 \vee x_2 \vee x_3)$
x_1
$\neg x_2$
$\neg x_3$
$\neg x_1$
x_2
x_3
x_1
x_2
x_3

Proving the Cook-Levin Theorem

CIRCUIT is NP-Complete. Prove it. Sketch. Let L be in NP. Then there exists a polynomial p such that …

This stuff here is all hardwired into the circuit   This stuff here is all "floating"

```
q0  a   b   a   b   …   b   a   #  0   1   0   1    …   1

c   p   b   a   b   …   b   a   #  0   1   0   1    …   1

r   a   d   a   b   …   b   a   #  0   1   0   1    …   1

……………………………………………………………………………………………………………

……………………………………………………………………………………………………………

……………………………………………………………………………………………………………

 c  a   1   a   0   …   b   a   b  0   qa   0   1    …   1

                              |
                              |
                         Output wire
```

The output wire is preceded by logic to capture that there is a q_a somewhere in the final row. Logic to enforce that the right value of x is hardwired in at the beginning, q0 as the initial state, one character on each cell for the certificate. Then, logic to capture that each cell is computed correctly from the cell above, to the left, and to the right.

Lecture 10W

Today:
- o Awards
- o IP and ZK
- o Living, light or heavy   //Milan Kundera, *The Unbearable Lightness of Being*

Announcements: o Final: Fri 6/8 @ 10:30 am in (sid%2)? Giedt1003: Roessler66

Awards:

1) P-probably-isn't-closed-under-homomorphism award (PPICUH) If P is closed under homomorphism than P=NP. Use the NP-completeness of SAT and replace every the 0/1 certificate with an equal length string of just 0s. What about: if P=NP then P is closed under homomorphism. Let L in P, h a homomorphism. Given a string y in h(L) a verifier Vcan determine if there is an x in L such that x in L and h(x)=y, whence, if P=NP, we can construct an algorithm that finds such an x or determines that none exists.
2) Discord awards: 30-or-more posts | set it up| every posted your lecture notes
3) Perfect-attendance prize
4) Q9-art award

IP and ZK

IP enlarges NP in to way: we allow interaction between the prover and the verifiers, and we allows the verifier to be probabilistic. It is okay if it has a tiny chance of error (over its coins).

```
  x            alpha_1            x
P   <---------------------------  V
                beta_1             R
    ---------------------------->

                   …

               alpha_n
    <---------------------------
               beta_n
    ---------------------------->

                                   V(x, alpha_1, beta_1, …, alpha_n, beta_n, R) → accept or reject
```

IP: L is in IP if there exists a verifier V such that:

1) For some prover P, for every x in L, Pr[(P<-->V) (x) accepts] $> 1 - 2^{100}$
2) For all provers P*, for every x not in L, Pr[(P<-->V) (x) accepts] $< 2^{100}$

ZK: adds in a condition that all that V see from an interaction with the honest prover is a sample from a distribution that it could generate on its own, simply by assuming that x is in L.

PSPACE: the set of languages that can be decided in polynomial SPACE. Huge. Very hard to think of decidable languages that need a super-polynomial amount of space. Example: Showing that a language is not decidable. Deciding if generalized-game problems (eg, generalizing chess or go) are a win for white, black, or neither.

Theorem [Shamir, 1992]. IP = PSPACE

Theorem [a zillion people, including me]   IP = ZKIP   (=PSACE)

Zero-Knowledge IP: things that can be proven where you don't reveal anything except for the fact itself. Formalization is based on simulatability, capturing the idea that you learn nothing from an interaction that establishes that x is in L if that which you obtain is just a random sample from a distribution that you yourself could generate by just ASSUMING that x is in L.

Something easier: NP is in ZKIP

Proof: describe the standard proof for G3C, in the envelope model.

[proof]

Living, light or heavy

There is something wonderfully game-like in an enterprise like investigation IP or ZK. The starting point is not the real world; it is a flight of fancy. And the pursuit isn't really supposed to lead you to something practical, either; if that should happen, it's more or less by accident, and, in the view of many theorists, an unfortunate turn. These things are and were for fun. Unapologetically.

To be clear: not many theoretical computer scientists lose sleep over alien arrival. The are our playthings. Maybe they are also our metaphor for power. Perhaps because the computer scientists' fantasy is that ultimate power is computational prowess. For who exactly wins in such a world?

Theoretical computer science (TCS) was once, and a piece remains, an arena for fanciful games. And if the choice is between playing these games verses helping Facebook or Chevron do their shit, then please, play the games.

But at some point, for me, it no longer felt like enough. life playing intellectual games.

It felt inappropriate because the world felt so totally on the brink of collapse. You probably don't need *me* to tell you the environment is in crisis. That we're in the middle of the sixth mass extinction event the planet has witnessed, and the first one cause by one species effectively attacking almost all of the rest.

The other day I saw an article in the Guardian with some snippets of commencement addresses. The one that caught my eye was someone named Patton Oswalt, a comedian. He told the kids:

> To the graduating class of 2023, I say three words: *you poor bastards*.
>
> Democracy's crumbling, truth is up for grabs, the planet's trying to kill us, and loneliness is driving everyone insane.

> I breezed into a world full of trivia and silliness and fun. You are about to enter a hellscape where you will have to fight for every scrap of your humanity and dignity. You do not have a choice , be anything but extraordinary. Those are the times you're living in right now.

It all seemed funny, accurate, and reasonable – until he says that everyone has to be extraordinary. Because not everybody gets to be extraordinary; that's just the nature of being *extra*ordinary.

I think that one feels constantly pushed these days in multiple directions these days. I do, at least, and I think others feel it, too.

1) In one direction is the draw of *conventionalism*. Doing what everyone else is doing. No need to think. Get a good paying job, make money, buy a house, get married, whatever people around you seem to want or expect. to want.

2) On the one hand there is this desire to, well, *save the world*. To find a *meaningful* life. To genuinely work to be *extra*ordinary. You start an NGO, or find a great one to work at. You become a prominent writer or artist or scientist. You use your perch to remake the world—or at least some piece of it. It is not easy. It is a life lived *heavy*.

3) But then, alas, you decide that there's no real chance of (2), that your efforts can't meaningfully change the world. You start to think that your best bet is to just try to *live lightly*. You want to carve out a life dominated by friendships, love, sex, food, travel, recreation, fun. Life is short. And when the heaviness intrudes, as it will, you do your best to push it aside.

Living life too lightly does not work. It will feel empty. It excessively foregrounds self. But living a life too heavy also does not work. It collapses from the weight. You become embittered and humorless. Somehow one must find a balance.

The last couple of weeks I kept remembering fragments of a book I read when I was about your age: a novel by a Czech author Milan Kundera, *The Unbearable Lightness of Being*. I remember the book having a strong impact on me. The title itself is a paradox in the spirt of *zero-knowledge*, no? If something is genuinely knowledge, how can it be zero? If one's way of being is light, how could it be unbearable?

I don't really remember Kundera's take on these questions, but maybe my recalling the novel in these last weeks is a sign that I internalized Kundera's ideas, but that they are relevant now, and that I am in need of reminding. So, as the last thing for today, and for our course, I am going to invite anyone interested to read (or re-read) this charming book, and to have a discussion on it. I promise to be in my **Zoom** room on **9/9 at 9 pm**, ready and eager to discuss with anyone who wants the book (not the film!) ***The Unbearable Lightness of Being***.

Kind wishes,
Phillip Rogaway