Computer Science 121: Introduction to Formal Systems and Computation

Tommy MacWilliam <tmacwilliam@cs.harvard.edu>

December 19, 2011

1 Introduction and Overview

2 September 6: Sets, Relations, Strings, Languages

- sets are defined by their members
  - $A = B$ means that for every x $x, \in A$ iff $x \in B$
- sets can be finite or infinite
  - if A is finite, then its cardinality $|A|$ is the number of elements in A
  - the empty set $\emptyset$ has cardinality 0
- set operations
  - union: $\{a, b\} \cup \{b, c\} = \{a, b, c\}$
  - intersection: $\{a, b\} \cap \{b, c\} = \{b\}$
  - difference: $\{a, b\} - \{b, a\} = \{a\}$
- A and B are disjoint iff $A \cap B = \emptyset$
- power set of $S = P(S) = \{X : X \subseteq S\}$
  - $P(\{a, b\}) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$
  - $|P(S)| = 2^{|S|}$ (provided S is finite)
- function $f : S \to T$ maps each element $s \in S$ to (exactly one) element of T, denoted $f(s)$
  - one-to-one: $s_1 \neq s_2 \implies f(s_1) \neq f(s_2)$
  - onto: for every $t \in T$ there is an $s \in S$ such that $f(s) = t$
  - bijection: one-to-one and onto
- S has (finite) cardinality $n \in \mathcal{N}$ iff there is a bijection $f : \{1, \dots, n\} \to S$
- a k-ary relation on $S_1, \dots, S_k$ is a subset of $S_1 \times \cdots \times S_k$
  - a binary relation on S is a subset of $S \times S$
- a binary relation can be pictured as a directed graph
  - formally, a directed graph G consists of a finite set V of vertices and a set of edges $E \subseteq V \times V$
    - transitive: path from A to C means path from A to B to C

    * symmetric: all edge has corresponding edge in the other direction
    * reflexive: each node has an edge to itself

- symbol: $a, b, \ldots$
- alphabet: finite, nonempty set of symbols, usually denoted by Σ
- string: finite number of symbols "put together"
  - empty string denoted by ε
- $\Sigma^\star$: set of all strings over the alphabet Σ
  - e.g. $\{a, b\}^\star = \{\epsilon, a, b, aa, ab, \ldots\}$
- order for writing strings is lexicographic order (shorter strings first, alphabetical order within strings of same length)
- concatenation of strings written as $x \cdot y$ or just xy
- reversal x^R of a string x is x written backwards

3 September 8: Proofs and DFAs

- a language L over an alphabet Σ is a set of strings over Σ (i.e. $L \subseteq \Sigma^\star$)
  - can be either finite or infinite
- ε is an empty string, and $\emptyset$ is the empty set
  - different than $\{\varepsilon\}$ and $\{\emptyset\}$
- concatenation of languages $L_1L_2 = \{xy : x \in L_1, y \in L_2\}$
  - e.g. $\{a, b\}\{a, bb\} = \{aa, ba, abb, bbb\}$
- Kleene star: $L^\star = \{w_1, \ldots, w_n \ : \ n > 0, w_1, \ldots w_n \in L\}$
  - e.g. $\{aa\}^\star = \{\epsilon, aa, aaaa, \ldots\}$, $\{ab, ba, aa, bb\}^\star$ is all even strings
  - $\emptyset^\star = \{\varepsilon\}$
- proof is a formal argument of the truth of some mathematical statement
  - formal means that successive statements are unambiguous, and could be put into a syntax a machine could check
- hints for writing proofs
  - state the game plan, including proof technique
  - keep the flow linear, and use English to move from step to step
  - use as little new symbolism as possible, and use existing symbolism properly
  - avoid the word "clearly"
  - when the proof is done, clearly state you are done
- pigeonhole principle: if there are more pigeons than pigeonholes and every pigeon is in a pigeonhole, then some pigeonhole must contain at least 2 pigeons

- for any finite sets S and T and any function $f : S \to T$, if $|S| > |T|$ then there exist $s_1, s_2 \in S$ such that $s_1 \neq s_2$ but $f(s_1) = f(s_2)$

- proofs by induction: base case, induction hypothesis (i.e. assume true for n), use induction hypothesis to arrive at definition for $n + 1$

- proofs by contradiction: assume opposite of claim, then deduce that opposite of claim must be false, so claim must be true

4 September 13: Finite Automata

- DFA is a set of states with transitions among the states

- DFA starts at a designated start state and is given some input string. for each symbol in the input string, transition function determines which state to go to next based on current symbol

- formal definition of a finite automaton: 5-tuple $(Q, \Sigma, \delta, q_0, F)$

  - Q: finite set of states
  - Σ: finite alphabet
  - δ: transition function, $Q \times \Sigma \to Q$
  - q_0: start state, $q_0 \in Q$
  - F: set of accept/final states, $F \subseteq Q$

- M accepts a string X if after starting M in the start state with head on the first square, when all X has been read, M ends up in a final state

- if $\delta(p, \sigma) = q$, then if M is in state p and reads symbol $\sigma \in \Sigma$, then M enters state Q

- size of a DFA defined by number of states, not edges

- formal definition of computation: $M = (Q, \Sigma, \delta, q_0, F)$ accepts $w = w_1 w_2 \cdots w_n \in \Sigma^\star$ if there exist $r_0, \ldots, r_n \in Q$ such that

  - $r_0 = q_0$
  - $\delta(r_i, w_{i+1}) = r_{i+1}$ for each $i = 0, \ldots, n - 1$
  - $r_n \in F$

- a language is called regular if some finite automaton recognizes it

- more formal definition:

  - inductively define $\delta^\star : Q \times \Sigma^\star \to Q$ by $\delta^\star(q, \epsilon) = q$, $\delta^\star(q, w\sigma) = \delta(\delta^\star(q, w), \sigma)$
  - intuitively, $\delta^\star(q, w) =$ state reached after starting in q and reading the string w
  - M accepts w if $\delta^{\star 9} q_0, w) \in F$

5 September 15: Nondeterministic Finite Automata

- a deterministic computation is one in which the machine is in a single state and knows exactly what the next state will be

  - in a nondeterministic machine, several choices for a given state may exist

- an NFA can have multiple transitions to different states for the same input symbol

  - in this case, machine makes multiple copies of itself, then each branch continues computing independently

- can also have a transition for ε, such that machine transitions without a need for input
- formal definition of a nondeterministic finite automaton: 5-tuple $(Q, \Sigma, \delta, q_0, F)$
  - Q: finite set of states
  - Σ: finite alphabet
  - δ: transition function, $Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$
  - $q_0 \in Q$: start state
  - $F \subseteq Q$: set of accept states
- $N = (Q, \Sigma, \delta, q_0, F)$ accepts $w \in \Sigma^\star$ if we can write $w = y_1 y_2 \cdots y_m$ where each $y_i \in \Sigma \cup \{\varepsilon\}$ and there exist $r_0, r_1, \ldots r_m \in Q$ such that:
  - $r_0 = q_0$
    * machine must begin at start state
  - $r_{i+1} \in \delta(r_i, y_{i+1})$ for each i
    * next state r_{i+1} only allowable if transition function takes current state r_i to r_{i+1} given the next input y_{i+1}
  - $r_m \in F$
    * acceptability defined if current state is in the set of final states
- NFA accepts w if there is at least one accepting computational path on w
  - number of paths can grow exponentially with w, because machine keeps copying itself
- for every NFA N, there exists a DFA M such that $L(M) = L(N)$
  - where $L(N)$ denotes the language accepted by N
  - states of M are the sets of states in N, or $M = \mathcal{P}(N)$
    * i.e. if branch goes to q_1 and q_2 simultaneously, introduce a new node $\{q_1, q_2\}$
  - final states of DFA are all states that contain final state of NFA
  - states that are unreachable by NFA must be defined in DFA as dead states (i.e. using $\emptyset$)
- NFAs allow us to easily represent strings that begin with $aaba$, strings that end with $aaba$, etc.
- regular language is one that can be represented by a DFA/NFA
- class of regular languages is closed under
  - union: $L_1 \cup L_2 = \{x \mid x \in A \vee x \in B\}$
    * proof: new start state with ε-transitions to start states of L_1 and L_2
  - concatenation: $L_1 \circ L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}$
    * proof: ε-transitions from final states of L_1 to start state of L_2
  - Kleene star: $L_1^\star = \{x_1 x_2 \cdots x_k \mid k \geq 0, x_1 \in L_1\}$
    * proof: new start state that is an accept state, ε-transition to original start state, ε-transitions from accept states to original start state
  - complement: $\overline{L_1}$
  - intersection: $L_1 \cap L_2$

6 September 20: Regular Expressions

- subset construction says any n-state NFA can be represented as a 2^n-state DFA
- regular expressions represent languages as strings
  - $L((R_1 \circ R_2)) = L(R_1) \circ L(R_2)$, $L(((a^\star) \circ (b^\star))) = \{a\}^\star \circ \{b\}^\star$
  - $L(\cdot)$ called semantics of the regular expression
- R is a regular expression if it has the form a, ε, $\emptyset$, $(R_1 \cup R_2)$, $(R_1 \circ R_2)$, or $(R_1^\star)$
- $(0 \cup 1) = \{0\} \cup \{1\}$
- $R \cup \emptyset = R$ and $R \circ \varepsilon = R$
- precedence order: *, then $\circ$, then $\cup$
  - $a \cup bc^\star = (a \cup (b \circ (c^\star)))$
- can use Σ to represent any symbol in the alphabet, so $\Sigma^\star a$ represents all strings ending in a
- using closure properties of regular languages, language is regular if it can be represented by a regular expression
- regular expressions ε (empty string) and $\emptyset$ (language containing no strings) are different

7 September 22: Regular Languages and Countability

- for every regular language L, there is a regular expression R such that $L(R) = L$
- GNFA: have transitions labelled by regular expressions, one start and accept state, and exactly one transition between states
- for every NFA N, there is an equivalent GNFA G
- for every GNFA G, there is an equivalent regular expression R
- constructing GNFAs:
  - rip: remove a state q_r (other than q_{start} and q_{accept})
  - repair: for every two states $q_i \notin \{q_{accept}, q_r\}, q_j \notin \{q_{start}, q_r\}$, let $R_{ij}, R_{ir}, R_{rr}, R_{rj}$ be regular expressions on transitions $q_i \to q_j$, $q_i \to q_r$,$q_r \to q_r, q_r \to q_j$
  - then in GNFA, put $R_{ij} \cup R_{ir} R_{rr}^\star R_{rj}$ on the transition $q_i \to q_j$
  - essentially, look at paths from q_i to q_j containing q_r, then construct regular expression such that single arrow from q_i to q_j accomplishes the same thing

8 September 27: Non-Regular Languages

- an alphabet Σ is finite by definition, so $\Sigma^\star$ is countably infinite (and $\mathcal{P}(\Sigma^\star)$ is uncountable)
- for every alphabet Σ, there exists a non-regular language over Σ
- language like $0^n 1^n$ is not regular, because machine must remember number of 0s seen
- approach to proving non-regularity: prove a general property for all regular languages, then show the language does not have it

- pumping lemma: if L is regular, then there is a number p such that for every string $s \in L$ of length at least p, s can be divided into $s = xyz$, where $y \neq \varepsilon$ and for every $n \geq 0$, $xy^n z \in L$
  - p is the number of states in the smallest DFA
  - division $s = xyz$ satisfies $|xy| \leq p$ and $|yz| \leq p$
  - each string contains a section that can be repeated any number of times with the resulting string remaining in the language
- definition of pumping lemma: $s = xyz$ satisfies:
  - for each $i \geq 0$, $xy^i z \in A$
  - $|y| > 0$ (aka $y \neq \varepsilon$)
  - $|xy| \leq p$
- pumping lemma essentially says there is some sequence of states that takes string to a potentially repeating sequence of states, then through a sequence to a final state
- using the pumping lemma: proof by contradiction
  - suppose L is regular, so L has a pumping length > 0
  - look at what strings are accepted, then show you can never have $s = xyz$
- example: application of the pumping lemma on the language $\{0^n 1^n \mid n \geq 0\}$
  - let $s = 0^p 1^p$. consider 3 cases:
  - y has only 0s. then $xy^2 z$ has more 0s than 1s, so PL does not hold
  - y has only 1s. then $xy^2 z$ has more 1s than 0s, so PL does not hold
  - y has both 0s and 1s. $xy^2 z$ must contain some 1s before 0s, so PL does not hold
  - therefore, this language is not regular, because every regular language can be pumped
- example: application of the pumpin lemma on the language $\{w \mid w \text{ has an equal number of 0s and 1s}\}$
  - let $s = 0^p 1^p$. by condition 3, $|xy| \leq p$, so y cannot contain any 1s
  - therefore, this language is not regular because it cannot be pumped

9 September 29: DFA Minimization and Context-Free Grammars

- minimizing DFAs
  - states p and q are distinguishable if there is a string w such that $\delta^\star(p, w)$ and $\delta^\star(q, w)$ is final
  - divide M into equivalence classes of final and non-final states
  - break up equivalence classes: if p, q are in the same equivalence class but $\delta(p, \sigma)$ and $\delta(q, \sigma)$ are not equivalent for some $\sigma \in \Sigma$, then p and q must be in different classes
  - when all states are separated, form a new, finer equivalence relation, and repeat
- context-free grammar: set of generative rules for strings
  - more powerful method of describing languages than DFAs
- using grammars: write down start variable, find a variable that is written down and replace, repeat until no steps remain
  - $A \to 0A1$, $A \to B$, $B \to \#$ generates 000#111

- all strings that can be possibly generated comprise the language of the grammar

- sequence of steps taken to generate a string from a grammar called a derivation
  - $L(G_1)$ denotes the language generated by the grammar G_1
- can abbreviate several rules with $A \to 0A1 \mid B$ (as opposed to $A \to 0A1, A \to B$)
  - example: $S \to aSb \mid SS \mid \varepsilon$ generates strings of properly nested parentheses
- formal definition of a CFG 4-tuple: $G = (V, \Sigma, R, S)$
  - V: finite set of variables
  - Σ: finite set of terminals
  - R: finite set of rules, each of the form $A \to w$ for $A \in V$ and $w \in (V \cup \Sigma)^\star$
  - S: start variable, where $S \in V$
- derivations: for α, $\beta \in (V \cup \Sigma)^\star$:
  - $\alpha \Rightarrow_G \beta$ if $\alpha = uAv, \beta = uwv$ for some $u, v \in (V \cup \Sigma)^\star$ and rule $A \to w$
  - $\alpha \Rightarrow_G^\star \beta$ (α yields β) if there is a sequence $\alpha_0, \ldots, \alpha_k$ for $k \geq 0$ such that $\alpha_0 = \alpha$, $\alpha_k = \beta$, and $\alpha_{i-1} \Rightarrow_G \alpha_i$
- tips for designing CFGs
  - many CFLs are simply the union of simpler CFLs, so construct easier CFGs, then mergy them
  - if language is regular, then first construct a DFA, then convert the DFA to a CFG
  - if machine needs to remember how many of a symbol exist (like $0^n 1^n$), then rules in the form $R \to uRv$ will come in handy
  - think about CFGs recursively, and place variables where recursive structures can appear
- string is derived ambiguously if grammar derives the same string in different ways
  - some languages are inherently ambiguous, such that they can only be generated using an ambiguous grammar
- a CFG is in Chomsky normal form if every rule is in the form $A \to BC$ or $A \to a$
  - first, add a new start variable, then eliminate rules in the form $A \to \varepsilon$, then eliminate rules in the form $A \to B$, patching up grammar so it generates same language

10 October 4: Pushdown Automata

- given a context free grammar G, parse tree describes how to interpret a string x
- regular grammars generate exact the regular languages
  - a CFG is right-regular if any occurrence of a nonterminal in a rule is the rightmost symbol
- converting a DFA to a regular grammar
  - variables are states
  - $\delta(P, \sigma) = R$ becomes $P \to \sigma R$
  - if P is accepting, add rule $P \to \varepsilon$

- pushdown automata composed of finite automaton + pushdown store
  - pushdown store is a stack of symbols which the machine can read/alter only at the top
  - transitions in the form $(q, \sigma, \gamma) \mapsto (q', \gamma')$: if in state q reading σ and γ on top of stack, replace γ with γ' and enter state q'
  - stack provides additional memory beyond finite amount available in control
  - equivalent in power to CFGs, and some languages are easier to express in a particular way
- when symbol is written onto the stack, all other variables shift downward
- PDA accepts a string if computation starts in start state with head at beginning of string and stack empty and ends in a final state with all input consumed
  - if no transition matches both the input and stack, PDA dies
- example PDA for $0^n 1^n$: for every 0, push a 0 onto the stack, and for every 1, pop a 0 off the stack. if stack is non-empty when input remains, then do not accept, else accept
- transition function includes current state, input symbol read, and variable at the top of the stack
- special variable $ used to signify an empty stack
- notation $a, b \to c$ signifies that machine reading a from input may replace the symbol b on the top of the stack with a c
  - if $a = \varepsilon$, machine can transition without reading any symbols
  - if $b = \varepsilon$, machine can transition without popping anything
  - if $c = \varepsilon$, machine can transition without pushing anything
- example: PDA for even palindromes
  - $(q, a, \varepsilon) \mapsto (q, a)$
  - $(q, b, \varepsilon) \mapsto (q, b)$
  - $(q, \varepsilon, \varepsilon), (r, a, a), (r, b, b) \mapsto (r, \varepsilon)$
- formal definition of a PDA: $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$
  - Q: states
  - Σ: input alphabet
  - Γ: stack alphabet
  - δ: transition function, where $Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to P(Q \times (\Gamma \cup \{\varepsilon\}))$
  - q_0: start state
  - F: set of final states
- the class of languages recognized by PDAs is the CFLs
  - for every CFG G there is a PDA M with $L(M) = L(G)$
  - for every PDA M there is a CFG G with $L(G) = L(M)$
  - therefore, PDAs are equivalent in CFGs
- proof that every CFL is accepted by some PDA

  - general idea: derivation is simply a sequence of substitutions, and each step of a derivation yields some intermediate string. non-determinism used to guess which variable to substitute. so, store symbols starting with the first variable in the intermediate string on the stack
  - put \$ on the stack. if top of stack is variable A, nondeterministically select one of the rules for A and substitute A by the string on the RHS of rule. if top of stack is terminal a, read next symbol and compare; if match repeat, if not, die. if top of stack is \$, enter accept state if all input has been read
  - let $G = (V, \Sigma, R, S)$. create a generalized PDA that can push strings onto the stack, via $(q, a, b) \mapsto (r, cd)$
  - corresponding PDA has 3 states: start state, loop state, final state
  - transitions:
    * start by putting $S\$$ on the stack, then go into q_{loop}: $\delta(q_{start}, \varepsilon, \varepsilon) = \{(q_{loop}, S\$)\}$
    * remove a variable from the top of the stack, replace it with a corresponding right hand side: $\delta(q_{loop}, \varepsilon, A) = \{(q_{loop}, w)\}$ for each rule $A \to w$
    * pop a terminal symbol from the stack if it matches the next input symbol: $\delta(q_{loop}, \sigma, \sigma) = \{(q_{loop}, \varepsilon)\}$ for each $\sigma \in \Sigma$
    * go to accept state if stack contains only \$: $\delta(q_{loop}, \varepsilon, \$) = \{(q_{accept}, \varepsilon)\}$

- proof that for every PDA there is a CFG
  - modify PDA so that there is a single accept state, all accepting computations end with an empty stack, and in every step, push or pop a symbol (but not both)
  - variables: A_{pq} for every two states p, q of M
  - goal: A_{pq} generates all strings that can take M from p to q, beginning and ending with the empty stack
  - rules:
    * for all states p, q, r, $A_{pq} \to A_{pr}A_{rq}$
    * for states p, q, r, s and $\sigma, \tau \in \Sigma$, $A_{pq} \to \sigma A_{rs} \tau$ if there is a stack symbol γ such that $\delta(p, \sigma, \epsilon)$ contains (r, γ) and $\delta(s, \tau, \gamma)$ contains (q, ϵ)
    * for every state p, $A_{pp} \to \epsilon$
  - start variable: $A_{q_{start}q_{accept}}$

11 October 6: Closure Properties and Non-CFLs

- CFLs are the languages accepted by PDAs
- CFLs are closed under union, concatenation, Kleene star, and intersection with a regular set
  - intersection proof: if L_1 is context-free and L_2 is regular, then construct a PDA with state set $Q_1 \times Q_2$ that keeps track of computation of both M_1 (a PDA) and M_2 (a DFA)
  - intersection between two context-free languages is not necessarily context-free
  - complement of a CFL is not necessarily context-free
- pumping lemma for CFLs: if L is context-free, then there is a number p such that any $s \in L$ of length at least p can be divided into $s = uvxyz$ where:
  - $uv^ixy^iz \in L$ for every $i \geq 0$
  - $|vy| > 0$ (both v and y cannot be empty)
  - $|vxy| \leq p$

- pumping lemma for CFLs essentially says that string can be divided into 5 parts, where parts 2 and 4 can be pumped
- proof of pumpinig lemma
  - since RHS of rules in CFGs have a bounded length, long strings must have tall parse trees
  - tall parse tree must have a path with a repeated nonterminal
    - let $p = b^m + 1$, where b is the max length of RHS of rule, and m is the # of variables
    - suppose T is the smallest parse tree for a string $s \in L$ of length at least p. then
    - let h be the height of T. $b^h \geq p = b^m + 1$, so $h > m$, therefore path of length h in T has a repeated variable
- example: application of pumping lemma to $a^n b^n c^n$
  - if v and y consist of only one type of symbol, then there are no longer equal numbers
  - if v and y contain more than one type of symbol, then symbols are out of order

12 October 13: General CF Recognition

- converse of CFL pumping lemma is false, because some non-context-free languages satisfy conclusion of pumping lemma
- top-down CFG to PDA construction
  - start by putting start variable on the stack
  - remove variable from top of stack and replace it with corresponding RHS
  - pop a terminal symbol from the stack if it matches the next input symbol
  - go to accept state if stack contains only $
- bottom-up CFG to PDA construction
  - start by putting $ on the stack
  - shift input symbols on the stack
  - reduce RHS on the stack to corresponding LHS
  - accept if stack contains just start variable and $
- context-free recognition: given some CFG, determine if $w \in L(CFG)$
  - could construct PDA from CFG, then run PDA on w
  - could also brute-force by checking all parse trees of height up to some upper limit
  - improvement: transform CFG into CNF, then use dynamic programming
- recall grammar is in CNF if every rule is in the form $X \to YZ$ or $X \to \sigma$
  - to convert, eliminate all non-CNF rules in this order: all ε-rules, unit rules (in the form $X \to Y$), long rules (in the form $X \to ab$), terminal-generating rules (in the form $X \to a$ where $a \notin V^\star$ and $|a| > 1$)
  - dynamic programming can be used if grammar is in CNF to make CF recognition algorithm run in polynomial, not exponential, time

13 October 18: Turing Machines and Simulations

- Turing machines are similar to finite automata, but have unlimited and unrestricted memory, and can do everythin a computer can do
- tape (which is infinite) initially contains input string, blank everywhere else
  - machine can also write to tape, and to read it, needs to move head back onto it
  - machine computes until it produces output, which can either be accept or reject output
- formal definition of a Turing machine: 7-tuple (Q, Σ, Γ, δ, q_0, q_{accept}, q_{reject})
  - Q: set of states
  - Σ: input alphabet not containing the blank symbol
  - Γ: tape alphabet, where $\in \Gamma$ and $\Sigma \subseteq \Gamma$
  - δ: transition function, $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
  - $q_0 \in Q$ is the start state
  - $q_{accept} \in Q$ is the accept state
  - $q_{reject} \in Q$ is the reject state, where $q_{accept} \neq q_{reject}$
- state of Turing machine called a configuration, and C_1 yields C_2 if C_1 can go to C_2 in a single step
  - if uaq_ibv yields uq_iacv, then we have $\delta(q_i, b) = (q_j, c, L)$
  - if uaq_ibv yields $uacq_jv$, then we have $\delta(q_j, c, R)$
  - Turing machine accepts input w if there exists a sequence such that C_0 is start and C_k is accept
- language is Turing-recognizable if some TM either accepts, rejects, or enters a loop
- language is Turing-decidable if some TM either accepts or rejects, without entering a loop

14 October 20: The Church-Turing Thesis

- all TM variabnts are equivalent in power, that is, they recognize the same class of languages (aka robust)
- multitape TM has multiple independent tapes and heads
  - every multitape TM has an equivalent single tape TM: concatenate tapes onto single tape, separated with delimiter symbol, simulate multiple heads by marking symbols
  - since multi-tape TM is equivalent to a TM, it must be true that a language is Turing-recognizable if and only if a multi-tape Turing machine recognizes it
- transition function for non-deterministic TM has form $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times L, R)$
  - still does not increase power, since every non-deterministic TM has an equivalent deterministic Turing machine
  - can simulate non-deterministic by having deterministic TM try each branch of the non-deterministic
  - view computation as a tree, then breadth-first search because DFS could lead to infinite computation, therefore missing solution
- Turing-recognizable languages also called recursively enumerable
  - enumerator is TM with attached printer, which it uses as an output device

- has a blank input tape, then runs and prints out strings in the language

- a language is Turing-recognizable iff some enumerator enumerates it
  - M: run E, and every time E outputs a string, compare to w
  - E: for every $i \to \infty$, run M for i steps on each possible input $s_i \in \Sigma^\star$
- algorithm is a simple series of steps for carrying out a single task
- Hilbert problems: solution can easily be recognized, but not easily determined
- Church-Turing Thesis: intuitive notion of algorithms is equivalent to Turing machine algorithms
  - Church's λ-calculus and Turing's machines are equivalent
- algorithm describes a process for solving a problem at a high level, such that tape-level TM descriptions are no longer necessary

15 October 25: Decidability, Universal Machines

- can now assume some reasonable computational models, such that implementation of machine does not determine decidability of language
- Turing's thesis says that a TM can be constructed for any computatable algorithm, so don't need to prove that a TM can be made
- decidable, computable, and recursive mean the same thing
- recognizable, recursively enumerable (aka r.e.) mean the same thing
- language is recognizable if there is a TM that will halt in accept state on strings in language, but will not necessarily halt on strings not in the language
- language is decidable if TM will answer yes or no after some amount of computation
- enumerators: TM without an input, but immediately starts computing and spits out strings
- decidable problems
  - A_{DFA}: test if DFA accepts an input string (prove by constructing TM to simulate DFA)
  - A_{NFA}: test if NFA accepts an input string (prove by converting NFA to DFA)
  - A_{REX}: test if regex accepts an input string (prove by converting regex to NFA)
  - E_{DFA}: test if language is empty (prove by searching for reachable final states)
  - EQ_{DFA}: test if languages of two DFAs are equal (prove by constructing symmetric difference)
  - A_{CFG}: test if CFG generates an input string (prove by converting to CNF and running for $2n-1$ steps)
  - E_{CFG}: test if language is empty (prove by marking terminal variables and trying to generate them)
- every CFL is decidable because A_{CFG} is decidable, and we can construct a machine that runs A_{CFG} decider on an input
- regular $\subseteq$ context-free $\subseteq$ decidable $\subseteq$ Turing-recognizable

16 October 27: Undecidability

- there exists a universal TM U such that when U is given $\langle M, w\rangle$ for any TM M and input w, U produces the result of running M on w
- proof that undecidable languages exist
  - every recursive language is decided by a TM
  - there exist countably many TMs
  - there exist uncountable many languages
- properties of recursive languages
  - if a language is recursive, then it is also r.e.
  - if a language is recursive, then so is its complement
  - a language is recursive iff both it and its complement are r.e.
- unsolvable problem: $A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$
  - in general, determining whether a TM M accepts an input string w
- A_{TM} is recognizable, because we can simulate M and accept if M accepts and reject if M rejects
  - relies on the existence of the universal TM U
  - corrolary: $HALT_{TM}$ (does M halt on the input w) is Turing-recognizable
- proof that A_{TM} is undecidable
  - suppose there exists a decider H that accepts if M accepts w and rejects if M does not accept w
  - now, construct a TM D that takes as input $\langle M\rangle$, runs H on $\langle M, \langle M\rangle\rangle$, then accepts if H rejects and rejects if H accepts
  - now, run D on $\langle D\rangle$. D accepts if D does not accept $\langle D\rangle$ and rejects if D does not accept $\langle D\rangle$
  - this is a contradiction (because D rejects $\langle D\rangle$ in the case when D accepts $\langle D\rangle$, so D and H cannot exist, so A_{TM} is undecidable
- a language is decidable if it is Turing-recognizable and co-Turing-recognizable
  - latter means that complement of the language is Turing-recognizable
- because A_{TM} is undecidable, $\overline{A_{TM}}$ must not be Turing-recognizable (because A_{TM} is recognizable and undecidable)

17 November 1: Reductions

- $HALT^{\varepsilon}_{TM}$: does a TM halt on the empty string, is also undecidable
  - $HALT_{TM}$ is undecidable for any input, because input simply represents initial state of the TM tape
- for any property X, a set S is co-X if $\overline{S}$ has the property X
  - non-Turing-recognizable languages: $\overline{A_{TM}}$, $\overline{HALT_{TM}}$
- A_{finite} is undecidable because we can reduce A_{TM} to A_{finite}

- construct $M^\star$ that simulates M and accepts if M accepts, loops forever if not

- one language reduces to another if we can use a black box for L_2 to build an algorithm to L_1
  - function f is computable if there is a TM that on an input w, halts with just $f(w)$ on the input tape

- a mapping reduction is a computable function $f : \Sigma_1^\star \to \Sigma_2^\star$ such that for any $w \in \Sigma^\star$, $w \in L_1$ iff $f(w) \in L_2$
  - notation is $L_1 \leq_m L_2$
  - if L_2 is decidable, then so is L_1, and if L_1 is undecidable, then so is L_2

- every nontrivial property of r.e. languages is undecidable
  - Rice's Theorem: if P is a subset of the class of r.e. languages and both P and $\overline{P}$ are both nonempty, then the language deciding if a string has the property P is undecidable
  - therefore, $L(M) = \emptyset$, $L(M) =$ regular, and $|L(M)| = \infty$ are all undecidable

- proof of Rice's Theorem
  - suppose that $\emptyset \notin P$. pick any $L_0 \in P$ and say $L_0 = L(M_0)$
  - define $f(\langle M \rangle) = \langle M' \rangle$ where M' is a TM that simulates M on ε, and if M halts, simulates M_0 on input w
  - because $HALT_{TM}^{\varepsilon}$ is undecidable, so is L_P

18 November 3: Undecidable Problems and Unprovable Theorems

- reduction is a means of converting one problem into another such that a solution to the second is a solution to the first

- $HALT_{TM}$ is undecidable because if we have a decider for $HALT_{TM}$, we also have a decider for A_{TM}

- proof by contradiction that $HALT_{TM}$ is undecidable
  - assume that the TM R decides $HALT_{TM}$. we can use R to construct S, a TM that decides A_{TM} (which we know cannot exist because A_{TM} is undecidable)
  - construct S by running R on input. if R rejects, reject, because this implies an infinite loop, which is not an acceptance. if R accepts, then we can simulate M, because accepting implies it will halt.
  - if M accepts, then accept, and if M rejects, reject
  - S clearly decides A_{TM}, but because A_{TM} is undecidable, then R cannot exist

- undecidable problems
  - E_{TM}: test if TM accepts any strings
  - $REGULAR_{TM}$: test if language accepted by a TM is regular
  - EQ_{TM}: test if languages accepted by two TMs are equal
  - EQ_{CFG}: test if languages generated by two CFGs are equal
  - test if intersection of two CFGs is empty
  - test if language generated by CFG is $\Sigma^\star$
  - test if language generated by one CFG is the subset of another CFG

- function is computable if there is some TM M that on every input w, halts with just $f(w)$ on the tape
  - often take the form of machine transformations, such that f returns the encoding of a new TM M' based on input $\langle M \rangle$
- formal definition of a mapping reduction: a language A is mapping reducible to a language B, written as $A \leq_m B$, if there is a computable function $f : \Sigma^\star \to \Sigma^\star$ where for every w, $w \in A \iff f(w) \in B$. f is called the reduction of A to B
  - if $A \leq_m B$ and B is decidable, then A is decidable
  - if $A \leq_m B$ and A is undecidable, then B is undecidable
  - if $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable
  - if $A \leq_m B$ and A is not Turing-recognizable, then B is not Turing-recognizable
- WATCH THIS LECTURE BECAUSE THE PROBLEMS ARE BANANULARS

19 November 8: Computational Complexity

- formula is a well-formed string over an alphabet of variables, relations, and quantifiers
- universe describes the values variables can be assigned
  - universe together with an assignment is called a model
  - for some model M, a theory of M, written $Th(M)$, is a collection of true sentences
- $Th(N, +)$ is decidable
  - $\forall x \exists y : [x + x = y]$ is a true statement in this model
- $Th(N, +, \times)$ is undecidable
  - reduce to $HALT^{\varepsilon}_{TM}$. M halts on ε iff $P_M = \exists n$ such that M halts on ε after n steps
- Godel's Incompleteness Theorem: some true statement must be unprovable
- a TM has a running time $t : \mathcal{N} \to \mathcal{N}$ iff for all n, $t(n)$ is the maximum number of steps taken by M for all inputs of length n
  - generally expressed as functions of n
  - $TIME(t)$ is the class of languages that can be decided by some multitape TM with running time $\leq t(n)$
- speeding up by a constant factor is the equivalent of throwing more hardware at a problem
  - too sensitive to multiplicative constants, so we instead study growth rate
- $g = O(f)$ if there exist $c, n_0 \in \mathcal{N}$ such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$
  - $O(n^k)$, or polynomial time, considered "fast"
  - $\Omega(k^n)$, or exponential time, considered "slow"
- $g = o(f)$ iff for every $\varepsilon > 0$, $\exists n$ such that $g(n) \leq \varepsilon \cdot f(n)$ for all $n \geq n_0$
- $f = \Theta(g)$ iff $f = O(g)$ and $g = O(f)$
- lower-order terms in a polynomial don't matter to growth rate
- $\log_a x = \Theta(\log_b x)\ \forall a, b > 1$

20 November 10: Polynomial Time

- asymptotic analysis describes the running time of an algorithm on large inputs
- little-o also defined as $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- every multitape TM that runs in $t(n)$ has an equivalent single-tape TM that runs in $t^2(n)$
- running time of a nondeterministic TM is time it takes for worst-case branch
  - every nondeterministic TM that runs in $t(n)$ has an equivalent deterministic TM that runs in $2^{O(t(n))}$
- brute force techniques often result in exponential running times
- $PATH \in P$, where $PATH$ is the problem of finding a path between two nodes in an undirected graph
  - depth-first search is a polynomial-time algorithm
- determining if two numbers are relatively prime is in P via Euclid's algorithm
- using dynamic programming, every CFL is in P because we have shown that all CFLs are decidable
- P is model-independent (for a reasonable computational model), such that changing model of computation will not remove a problem from P

21 November 15: NP

- Hamiltonian path through a graph is a path that goes through every node exactly once
  - easy to verify solution to $HAMPATH$ by simply checking path
  - however, determining if $HAMPATH$ exists in a graph cannot be done in polynomial time
- compositeness is a similarly easily-verifiable problem, because we can simply multiply the given p and q
- a verifier for a language A is an algorithm V where $A = \{w \mid V$ accepts $\langle w, c\rangle$ for some string c
  - verifier essentially returns true if input is a member of the language A
- NP is the class of problems verifiable in polynomial time
  - a language is in NP iff it is decided by some nondeterministic polynomial time TM
- we can convert a nondeterministic polynomial time verifier to an NTM
  - NTM nondeterministically selects a string c then runs V on certificate
- problems in NP
  - $CLIQUE$: subgraph in which every pair of nodes is connected
    * certificate: nodes in the clique
    * verifier: test if nodes are in G, have the appropriate number, and if subgraph contains all edges connecting nodes
  - $SUBSET - SUM$: is there a collection in a set such that elements sum to a given value
    * certificate: elements in the subset
    * verifier: test if all elements are in set and sum to desired target

- TSP (traveling salesman problem): there exists a tour of all cities of length $\leq$ some target
  - nondeterministic strategy: write does a sequence of cities for $\leq n^2$, then trace through the tour and check length (in $\leq n$)
- Hamiltonian circuit: special case of TSP, path that touches each node exactly once and returns to start
- Eulerian circuit: path passes through every edge exactly once and returns to start
- boolean satisfiability: determine assignment of variables satisfying a boolean formula
  - certificate: assignment of variables
  - verifier: test if resulting statement is true or false

- P is a subset of NP, but it is currently unknown if P and NP are equal
- a string for which a verifier accepts is called a certificate
- $2 - SAT \in P$ because we can utilize implications if there are only 2 literals/clause, which will take $O(n^2)$ steps

22 November 17: NP-Completeness

- a problem is NP-complete if it is in NP and all problems in NP reduce to it
  - therefore, if we can solve any NP-complete problem in polynomial time, we can solve all problems in NP in polynomial time (such that $P = NP$)
- Cook-Levin Theorem: $SAT \in P$ iff $P = NP$
- polynomial time reducibility is the analog to mapping reducibility for undecidable problems
  - if $A \leq_P B$ and $B \in P$, then $A \in P$
- example reduction: $3SAT$ to $CLIQUE$
  - construct a graph where each variable in boolean formula becomes a node
  - connect all nodes except contradictory variables (i.e., x and $\overline{x}$) and variables in the same triple
  - formula is satisfiable iff there exists a k-clique, where k is the number of clauses in the $3SAT$
    - at least one literal must be true in every clause
    - no two nodes in clique can be in the same clause, because nodes in same clause are unconnected
    - clique cannot contain a contradiction, because contrary variables are not connected
- if some language B is NP-complete and $B \in P$, then $P = NP$
- if B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete
- example reduction: $3SAT$ to $VERTEX - COVER$
  - $VERTEX - COVER$: are there k vertices such that at least one endpoint of every edge is covered
  - create node for each unique variable in the formula and its complement, then connect them (i.e. connect x_i to $\overline{x_i}$), forming a dumbbell for each unique variable
  - now, construct triangle of nodes for each clause, where each node represents a literal in the clause. connect each literal to the corresponding node created in the previous step (i.e. connect triangle node for x_i to the dumbbell node for x_i)
- example reduction: $VERTEX - COVER$ to $CLIQUE$
  - construct G^c, or the graph formed by removing all edges, then connecting all originally-unconnected nodes
  - G has a k-cover iff G^c has a $|G| - k$ clique

23 Cook-Levin Theorem

- SAT is NP-complete
  - clearly in NP, because certificate consisting of variable assignments is easily verifiable
  - computation can be represented using a tableau, where each cell on the tape is a cell in a column
  - each row can be computed from the previous using a circuit
  - processor in a computer is a circuit, so everything can be reduced to SAT
  - QED by CS124
- more NP-complete reductions
  - $INDEPENDENT-SET$: set of vertices such that no two are adjacent
    - certificate: set of vertices in independent set
    - reduction from $3SAT$
      - add a node for each variable in each clause, forming a triangle of connected nodes
      - node in one triangle can be connected to a node in another triangle if the variables are negations of each other
      - we can have at most one vertex per triangle (and hence per clause) because triangle is connected
      - we cannot have contradictory assignments, because contradictions are also connected
  - $MAX-CLIQUE$
    - certificate: set of vertices in clique
    - reduction from $INDEPENDENT-SET$
      - any independent set in G is a clique in the complement of G, or G^c
  - $MIN-COVER$
    - certificate: set of vertices composing cover
    - reduction from $INDEPENDENT-SET$
      - if I is an independent set in G, then $V-I$ is a vertex cover in G
      - similarly, if C is a vertex cover in G, then $V-C$ is an independent set in G

24 Appendix A: Closure Properties

⟨blitza⟩	$\star$	ϕ	$\circ$	$\cup$	$\cap$	$\overline{L}$
regular	✓	✓	✓	✓	✓	✓
CF	✓	✓	✓	✓	×	×
recursive	✓	✓	✓	✓	✓	✓
r.e.	✓	×	✓	✓	✓	×