Proof techniques: construction, contradiction, counterexample, case enumeration, induction, pigeonhole principle, proving cardinality, diagonalization

Languages, Strings

Alphabet: finite set of symbols

String: finite sequence of symbols from alphabet

Replication: $w^0 = \varepsilon, w^{i+1} = w^i w$

Reverse: $w^R = w = \varepsilon$ if $|w| = 0$, else $\exists a \in \Sigma$ and $\exists u \in \Sigma^*$ such that $w = ua$, then define $w^R = au^R$

$\boldsymbol{(wx)^R = x^r w^r}$: induction on $|x|$, base case $|x| = 0$ so $x = \varepsilon$, consider any string x where $|x| = n+1$, then $x = ua$ for some character a and $|u| = n$, so $(wx)^R = (w(ua))^R = (wu)a^R = a(wu)^R = a(u^R w^R) = (au^R)w^R = (ua)^R w^R = x^R w^R$

Language: set of strings (finite/infinite) from alphabet, uncountably infinite number of these (power set of Σ^*)

$\boldsymbol{\Sigma^*}$: countably infinite with non-empty alphabet, enumerate with lexicographic order

$\boldsymbol{L_1 L_2}$: $\{w \in \Sigma^* : \exists s \in L_1(\exists t \in L_2(w = st))\}$

$\boldsymbol{L^*}$: $\{\varepsilon\} \cup \{w \in \Sigma^* : \exists k \geq 1(\exists w_1, w_2, \ldots w_k \in L(w = w_1 w_2 \ldots w_k))\}$ or $L^0 \cup L^1 \cup L^2 \cup \ldots$

$\boldsymbol{L^+}$: LL^* or $L^* - \{\varepsilon\}$ iff $\varepsilon \notin L$ or $L^0 \cup L^1 \cup L^2 \cup \ldots$

$\boldsymbol{(L_1 L_2)^R = L_2^R L_1^R}$: $\forall x(\forall y((xy)^R = y^R x^R))$ from before, then $(L_1 L_2)^R = \{(xy)^R : x \in L_1 \text{ and } y \in L_2\} = \{y^R x^R : x \in L_1 \text{ and } y \in L_2\} = L_2^R L_1^R$

Decision problem: problem to which answer is yes/no or true/false

Decision procedure: answers decision problem

Machine power hierarchy: FSM (regular), PDA (context-free), TM (semi-decidable & decidable)

Rule of least power: use least powerful language suitable for expressing info, constraints or programs on WWW

$\boldsymbol{firstchars(L)}$: $\{w : \exists y \in L(y = cx \wedge c \in \Sigma_L \wedge x \in \Sigma_L^* \wedge w \in c^*)\}$, closed under FIN but not INF (since result is first character *)

$\boldsymbol{chop(L)}$: $\{w : \exists x \in L(x = x_1 c x_2, x_1 \in \Sigma_L^*, x_2 \in \Sigma_L^*, c \in \Sigma_L, |x_1| = |x_2|, \text{and } w = x_1 x_2)\}$

, language where all strings have exact middle character removed, must have had odd length to begin with, closed under FIN but not INF (can get empty set if never odd length)

Extra: to describe language with at least 2 different substrings of length 2 $L = \{w \in \{a, b\}^* : \exists x, y(x \neq y \wedge |x| = 2 \wedge |y| = 2 \wedge Substr(x, w) \wedge Substr(y, w))\}$

Finite State Machines

DFSM Quintuple: $M = (K, \Sigma, \delta, s, A)$, K = finite set of states, Σ = alphabet, δ = transition function from $(K \times \Sigma)$ to K, $s \in K$ = initial state, $A \subseteq K$ = set of accepting states

Configuration: element of $K \times \Sigma^*$, current state and remaining input

Yields relation: $|-_M$, relates 2 configurations if M can move from the first to the second in 1 step, $|-_M^*$ for 0 or more

Computation: finite sequence of configurations for some $n \geq 0$ such that C_0 is an initial configuration, C_n is of the form (q, ε) for some state $q \in K_M$ and $C_0 |-_M C_1 |-_M C_2 |-_M \ldots |-_M C_n$

DFSM will halt in $|w|$ steps: execute computation from C_0 to C_n, each step will consume one character, so $n = |w|$, C_n is either accepting or rejecting configuration, so will halt after $|w|$ steps

Parity: odd if number of 1 is odd for binary string

$\boldsymbol{MinDFSM(M : DFSM)}$:

Initialise classes with an accepting class & non-accepting class
For each class with more than 1 state
For each state and character check which class it goes to
If behaviour differs between states split them
End for
End for
Go through all the classes again until no splitting happens
Each class becomes its own state, transitions already defined above

Number of states $\geq$ equivalence classes in L: suppose it is less than equivalence classes, then by pigeonhole principle there must be at least 1 state that contains strings from 2 equivalence classes, but then future behaviour on these two strings will be identical, which is not consistent with the fact that they are in different equivalence classes

NDFSM Quintuple: replace δ with Δ, transition relation, finite subset of $(K \times (\Sigma \cup \{\varepsilon\})) \times K$

NDFSM vs DFSM: can enter configuration with input symbols left but no move available (halt without accepting), can enter configuration from which 2 or more competing transitions available (ε-transition, more than 1 transition for single input character)

$\boldsymbol{eps(q)}$: $\{p \in K : (q, w) |-_M^* (p, w)\}$, closure of $\{q\}$ under relation $\{(p, r)$: there is a transition $(p, \varepsilon, r) \in \Delta\}$, to calculate initialise $result = \{q\}$, add all transitions $(p, \varepsilon, r) \in \Delta$ where $p \in result, r \notin result$ to $result$, then return $result$

$\boldsymbol{ndfsmtodfsm(M : NDFSM)}$:

Compute $eps(q)$ for each state q, $s' = eps(s)$ (initial state)
Set $active\text{-}states = \{s'\}$ (set of set of states) and $\delta' = \varnothing$
While $\exists Q \in active\text{-}states$ for which δ' has not been computed //computing δ'
For each $c \in \Sigma_M$
Set $new\text{-}state = \varnothing$
For each state $q \in Q$
For each state $p : (q, c, p) \in \Delta$
Set $new\text{-}state = new\text{-}state \cup eps(p)$
End for
Add $(Q, c, new\text{-}state)$ to δ', if $new\text{-}state \notin active\text{-}states$ insert it
End for
End for
End while
Set $K' = active\text{-}states$ and $A' = \{A \in K' : Q \cap A \neq \varnothing\}$

Extra: when making FSM may start with complement

Regular expressions

Allowable: $\varnothing, \varepsilon$, every element of Σ, if α, β are regex then so are $\alpha\beta, \alpha \cup \beta, \alpha^*, \alpha^+, (\alpha)$, no actual need for rules for ε and α^*

Order of operations: Kleene star, concatenation, union (high to low)

FSM & Regex equivalence: can create FSM to accept regex (do so for each rule), algorithm exists for other way

$\boldsymbol{fsmtoregexheuristic(M : FSM)}$: remove unreachable states, if no accepting states return $\varnothing$, if start state part of loop create new start state s & connect s to original start via ε-transition, if multiple accepting states create new accepting state & connect old ones to new with ε-transition, if only 1 state return ε, rip out all states other than start & accept, return regex

$\boldsymbol{fsmtoregex(M : FSM)}$: $buildregex(standardize(M))$

$\boldsymbol{standardize(M : FSM)}$: remove unreachable states, create start & accepting if needed (from heuristic), if multiple transitions exist between 2 states collapse, create any missing transitions with $\varnothing$

$\boldsymbol{buildregex(M : FSM)}$:

If no accepting return $\varnothing$, if only 1 state return ε
While states exist that are not start or accepting
Select some state rip
For every transition from p to q
If both p & q are not rip compute new transition $(R'(p, q) = R(p, q) \cup R(p, rip)R(rip, rip)^* R(rip, q)$, either direct or via rip), then remove rip
End for
End while
Return final regex

Regular grammar

Quadruple: (V, Σ, R, S), V = rule alphabet, both nonterminals & terminals, Σ = terminals, $\subseteq V$, R = finite set of rules, LHS single nonterminal, RHS is ε, single terminal or single terminal + single nonterminal, S = nonterminal

Regular Languages & Grammar equivalence: construction, both ways

$\boldsymbol{grammartofsm(G)}$: create separate state for each nonterminal, start state is S, if rule exist with single terminal RHS create new state labeled #, for each rule $X \to aY$ create transition from X to Y labeled a, if $X \to a$ go to #, if $X \to \varepsilon$ mark as accept, mark state # as accept, if undefined (state, input) pairs remaining point them to dead state & add loops to dead state for each character

Nonregular languages **Number of regular languages**: countably infinite, upper bound is number of FSM/regex, lower bound is every element of a^+ as its own language

Every finite language is regular: union them all

Show that L is regular: finite, FSM, regex, finite equivalence classes, regular grammar, closure theorems

Closure properties: union, concatenation, Kleene star, complement (swap accept & not, need all transitions explicitly, dead states & DFSM), intersection $(\neg(\neg L_1 \cup \neg L_2)$, De Morgan's law), difference $(L_1 \cap \neg L_2)$, reverse (turn start to accept, create new start connected by ε to accepting states, flip transitions), letter substitution ($letsub(L_1) = \{w \in \Sigma_2^* : \exists y \in L_1 \wedge w = y$ except every character c of y is replaced by $sub(c)\}$, sub = function from Σ_1 to Σ_2^*)

Pumping theorem: $\exists k \geq 1(\forall \text{strings } w \in L, \text{where} |w| \geq k(\exists x, y, z(w = xyz, |xy| \leq k, y \neq \varepsilon, \forall q \geq 0(xy^q z \text{ is in L}))))$

Using closure to prove nonregular: assume it is regular, use closure with known regular language, result is known nonregular language, so it must be nonregular, ex: $\#_a(w) = \#_b(w)$, intersect with a^*b^* to get $a^n b^n$, intersection & complement most useful

Octal divisible by 7: 0, 7, 16, 25, ..., true only if sum of digits divisible by 7, so states in FSM are mod 7, is regular

Extras:

$\boldsymbol{a^i b^j c^k; i, j, k >= 0}$, if $\boldsymbol{i = 1}$ then $\boldsymbol{j = k}$: can change conditions to $i \neq 1 \vee j = k$, all strings with length ≥ 1 is pumpable, so need to intersect with ab^*c^*, so $i = 1$ guaranteed and results in $ab^j c^k; j, k > 0 \wedge j = k$, then use pumping theorem

$\boldsymbol{a^i b^j, i \neq j}$: must use $a^k b^{k+k!}$, if only use $a^k b^{k+1}$ can just pump 2 a at a time to skip the equal part, $y = a^p$ for some nonzero p, pump in $\frac{k!}{p}$ times (must be integer because $p < k$), get $k + (\frac{k!}{p})p = k + k!$, alternatively prove that the complement is not regular using closure $(\neg L = a^n b^n \cup \{\text{out of order}\}$, intersect with a^*b^* to get $a^n b^n)$

Context-Free Languages

Rewrite system: list of rules & algorithms for applying them, match LHS of some rule against some part of working string & applies it, loop until told to stop, takes system & initial string as input, if given grammar & start symbol will generate language

CFG: no restriction on RHS but LHS must still have 1 nonterminal

CFG Quadruple: (V, Σ, R, S), R is finite subset of $(V - \Sigma) \times V^*$

CFL: Language is context-free iff generated by some CFG

Recursive: RHS can generate own LHS

Self-embedding: recursive but also includes terminals on both sides afterwards, if not true then must be regular (not necessarily vice versa)

Concatenation: use 2 nonterminals together in RHS, generate each separately

$\boldsymbol{a^n b^n}$: $S \to aSb|\varepsilon$

Balanced parentheses: $S \to \varepsilon|SS|(S)$

$\boldsymbol{ww^R}$: $S \to aSa|bSb|\varepsilon$

$\boldsymbol{\#_a(w) = \#_b(w)}$: $S \to aSb|bSa|SS|\varepsilon$

Arithmetic: $S \to E + E|E * E|(E)|id$

$\boldsymbol{(a^n b^n)^*}$**(each region can have different $\boldsymbol{n}$)**: $S \to MS|\varepsilon, M \to aMb|\varepsilon$

$\boldsymbol{a^n b^m : n \neq m}$: $S \to A, B$ (more a than b, more b than a), $A \to a, aA, aAb$ (at least one extra generated), $B \to b, Bb, aBb$

Removing unproductive rules: mark every nonterminal as productive & every terminal as productive, if RHS all productive then mark LHS productive, loop until no changes, remove unproductive

Removing unreachable rules: mark start as reachable & all others as not, if LHS reachable mark all nonterminals in RHS as reachable, loop until no changes, remove unreachable

Parse trees: leaf nodes are terminals except ε, root is start, all others are nonterminals

Branching factor: longest RHS length

Generative capacity: weak (set of strings), strong (set of parse trees)

Ambiguity: multiple parse trees for 1 string, can come from different operator first in arithmetic orfrom rules $S \to SS|\varepsilon$, even if only 1 S needed can generate multiple & nullify

Reducing ambiguity: remove null rules, recursive rules with symmetric RHS & ambiguous optional postfix (ex: dangling else, can attach to inner or outer if)

Removing null rules: mark all nonterminals in null rules as nullable, if any rule has nullable RHS then mark LHS as nullable, then for all modifiable rules (RHS has at least 1 nullable) add new rules containing variant without nullable, remove null rules

Removing symmetric rules: force branching to 1 direction, replace $S \to SS$ with $S \to SS_1|S_1$ for left branching, then make S_1 do what S originally did

Operator order: if parsed first (ex: in arithmetic, $E \to E + T$) then lowest priority