

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 $w^R = au^R$

$(wx)^R = x^R w^R$: 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^*)

Σ^* : countably infinite with non-empty alphabet, enumerate with lexicographic order

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

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

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

$(L_1 L_2)^R = L_2^R L_1^R$: $\forall x (\forall y ((xy)^R = y^R x^R))$ 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

***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 c^*)

***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 $L = \{w \in \{a, b\}^* : \exists x, y (x \neq y \wedge |x| = 2 \wedge |y| = 2 \wedge \text{Substr}(x, w) \wedge \text{Substr}(y, w))\}$

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 $C_0 |-_M C_1 |-_M C_2 |-_M \dots |-_M C_n$

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

***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)

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

***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 $\delta' = \emptyset$

While $\exists Q \in active\text{-}states$ for which δ' has not been computed //computing δ'

For each $c \in \Sigma_M$

Set $new\text{-}state = \emptyset$

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 $A' = \{A \in K' : Q \cap A \neq \emptyset\}$

Extra: when making FSM may start with complement

Regular expressions

Allowable: \emptyset, ε , every element of Σ , if α, β are regex then so are $\alpha\beta, \alpha \cup \alpha\beta, \alpha^*, \alpha^+, \alpha^\dagger, (\alpha)$, 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

***fsmtoregexheuristic*(M : FSM):** remove unreachable states, if no accepting states return \emptyset , 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

***fsmtoregex*(M : FSM): *buildregex*(standardize(M))**

***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 \emptyset

***buildregex*(M : FSM):**

If no accepting return \emptyset , 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 $(R'(p, q) = R(p, q) \cup R(p, \text{rip})R(\text{rip}, \text{rip})^*R(\text{rip}, q)$, 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

***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 $X \rightarrow aY$ create transition from X to Y labeled a , if $X \rightarrow a$ go to $\#$, if $X \rightarrow \varepsilon$ 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:

$a^i b^j c^k; i, j, k \geq 0$, if $i = 1$ then $j = k$: 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

$a^i b^j, i \neq j$: 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

$a^n b^n$: $S \rightarrow aSb | \varepsilon$

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

ww^R (Even palindrome): $S \rightarrow aSa | bSb | \varepsilon$

$\#_a(w) = \#_b(w)$: $S \rightarrow aSb | bSa | SS | \varepsilon$

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

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

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

$a^n b^m c^p d^q : m + n = p + q$: $S \rightarrow aSd | T | U, T \rightarrow aTc | V, U \rightarrow bUd | V, V \rightarrow bVc | \varepsilon$

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 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 or from rules $S \rightarrow SS | \varepsilon$, 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 $S \rightarrow SS$ with $S \rightarrow SS_1 | S_1$ for left branching, then make S_1 do what S originally did

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

Chomsky Normal Form

Normal form: for set C of data is a set of syntactically valid objects, for every element of C there is an equivalent element in F with respect to some set of tasks (possibly except finite exceptions), and at least some tasks are easier to perform on elements of F than C

CNF: RHS has 1 terminal or 2 nonterminals, parsers can use binary trees, exact length of derivations known ($|w| - 1$ applications of nonterminal rules & $|w|$ applications of terminal rules)

Conversion to CNF: remove null rules, remove unit productions ($A \rightarrow B$), remove mixed rules (RHS length ≥ 1 & has terminal), remove long rules (length ≥ 2)

Remove unit productions: pick one $X \rightarrow Y$, remove, for every rule $Y \rightarrow \dots$ make new rule $X \rightarrow \dots$ unless it has already been removed once

Remove mixed rules: make new nonterminal T_a for each terminal a , for all quali-

ifying rules substitute terminals for new nonterminals, then add rules $T_a \rightarrow a$ for all pairs

Remove long rules: chain them, ex: $A \rightarrow BCDE$ becomes $A \rightarrow BX_1, X_1 \rightarrow CX_2, X_2 \rightarrow DE$

Pushdown automaton

Sixtuple: $(K, \Sigma, \Gamma, \Delta, s, A)$, Γ is stack alphabet, Δ is transition relation, finite subset of $(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*) \times (K \times \Gamma^*)$, state, input, pop, state, push

Configuration: element of $K \times \Sigma^* \times \Gamma^*$, initial is (s, w, ε)

Top of stack: leftmost character

Accepting: in accepting state, stack & input empty

Transitions: input/pop/push

$a^n b^n$: push all a , when see b start popping a for every b , for $2n$ on b push 2 a for every a in input

Balanced parentheses: push opening, pop opening for every closing

wcw^R : push everything before c , after pop matching

Even palindrome: like above, but nondeterministically decide where the middle is instead of using c ($\varepsilon/\varepsilon/\varepsilon$)

$a^m b^n : m \neq n$: start with equal, if stack & input empty reject, if leftover detected in either stack or input clear & accept, still need to be sure of order

$\neg a^n b^n c^n$: out of order, $i \neq j, j \neq k$ (last two is just unequal a, b, c)

Deterministic: iff Δ_M contains no pairs of transitions that complete with each other & whenever M is in accepting configuration never forced to choose between accept & continue (via ε -transition with no popping)

Reduce nondeterminism: use # as bottom of stack marker & \$ as end of string marker

CFG to PDA top down: 2 states, start at p, q accepting, from p to q push start symbol, loop on q for each rule LHS and push RHS, for each terminal pop

CFG to PDA bottom up: 2 states, start at p, q accepting, from p to q pop start symbol, loop on p for each rule pop reverse RHS and push LHS (reduce), for each terminal push (shift)

Pumping lemma for CF

Show CF: create CFG, PDA, use closure

Definition: $\exists k \geq 1 (\forall \text{strings } w \in L, \text{ where } |w| \geq k (\exists u, v, x, y, z (w = uvxyz, |v| \leq k, |xy| \leq k, \forall q \geq 0 (uv^q xy^q z \text{ is in } L))))$

Closures: union (CFG with 2 possible paths), concatenation (above), Kleene star (let start repeat or ε), reverse (convert to CNF, flip nonterminal rules), interection with R, difference with R

Intersection: try $a^n b^n c^n \cap a^m b^n c^n$, get $a^n b^n c^n$, 2 originals CF but result not, but intersection of CF & R is CF (do it same way as intersection of R, works since only 1 stack needed)

Complement: $a^n b^n c^n$

Difference: $\neg L = \Sigma^* - L$, Σ^* is CF, if closed under difference would be closed under complement, but just proved latter is false, but works for CF & R ($L_1 - L_2 = L_1 \cap \neg L_2$)

$w w$: intersect with R, so $L' = L \cap a^* b^* a^* b^*$ to restrict form, then use pumping lemma on L' , if L is CF then L' must be CF but it is not

Deterministic CF Closures: complement but not union & intersection

Proof that nondeterministic CFL exists: complement of $a^i b^j c^k : i \neq j \vee j \neq k$ is $a^n b^n c^n \cup$ out of order, but then intersecting with $a^* b^* c^*$ returns $a^n b^n c^n$, so original is CF but not DCF

Turing machine

Sixtuple: $(K, \Sigma, \Gamma, \delta, s, H)$, Σ does not contain \square , Γ must contain \square & have Σ as subset, $H \subseteq K$ is set of halting states, δ is transition function $(K - H) \times \Gamma$ to $K \times \Gamma \times \{\rightarrow, \leftarrow\}$

Transition action: go to next state, write, move head

Convert $a^i b^j, 0 \leq j \leq i$ to $a^n b^n$: mark a with \$, find next b , if found mark with # & search for next pair, else write # & search for unmarked a , finally replace \$ to a & # to b

Configuration: $K \times ((\Gamma - \{\square\})\Gamma^*) \times \Gamma \times (\Gamma^*(\Gamma - \{\square\})) \cup \{\varepsilon\}$, state, active tape up to scanned square, scanned square, all active tape after scanned square

Start head position: before input

Macros: M_x writes x , L/R moves left/right once, h halts, y accepts, n rejects, $x \leftarrow a, b$ label in transition means if read either then assign to variable x & continue, if part of L/R then find first one & store, L/R_x find first x to left/right of current

Convert $w \in \{1\}^*$ to w^3 : mark each 1 with #, go to end, put 2 #, repeat until mo 1, then convert all # to 1

Convert $u \square w$ to uw : start from middle blank, for each character in 2nd half store in variable, write blank in original spot, write variable to the left

$a^n b^n c^n$: for each a mark, then mark corresponding b & c , if no more a verify no leftover b or c , also verify order of characters

wcw : look at each character from 1st half & compare to corresponding 2nd half using variables

$b^* a (a \cup b)^*$: at least 1 a , b can come anywhere, so just scan for a

Computing: TM computes a function iff for all w in alphabet & w is an input on which f is defined $M(w) = f(w)$, otherwise it does not halt, output of function is everything after head

Computable/Recursive: iff TM exists that computes it and always halts

Copy machine: for each character store in variable, write blank, $R_{\square} 2x$ (skip

middle blank & get to end of 2nd part), write variable, go back to start of 1st part & write variable back

Binary +1: go to end, flip all successive 1s at the end to 0s, then behind all the flipped bits write 1

$f(w) = ww$: copy, then shift 2nd part back

Busy beaver functions: $S(n)$ is max steps executed by TM (with tape alphabet blank & 1, halt on blank tape) with n non-halting states when started on blank tape before halting, $\Sigma(n)$ is max number of 1s left on tape by TM with n non-halting state before halt

Simulating multi-tape TM on single tape: $O(n \times (|w| + n)) = O(n^2)$ assuming that $n \geq w$, n is number of passes

Nondeterministic quintuple: $(K, \Sigma, \Gamma, \Delta, s, H)$, Δ is subset of $((K - H) \times \Gamma) \times (K \times \Gamma \times \{\rightarrow, \leftarrow\})$

2 of at least 1 letter: nondeterministically select 1 letter, then see if there are two of them

Week 10:

Simulating real computer:

Tapes: memory, program counter, address register, accumulator, current instruction opcode, input, output

Formats: memory is series of (address, value) pairs separated by delimiters, instruction is series of instruction number, 4-bit opcode & address (no need to separate last 2)

Algorithm: Move input string to input file, initialise PC to 0, loop (scan memory for instruction matching PC, copy opcode to current instruction, copy address to address register, PC+1, scan memory for address in address register (operand), if load copy memory to accumulator, if add add operand to accumulator, if jump copy address register to PC, etc)

Performance: $O(n^3)$, $O(n^6)$ on single tape

Encoding TM:

States: number them from 0 to $|K| - 1$ in binary, use smallest number of digits that fits, 0 for start, prefix with y for accept, n for reject, h for halt, q else

Tape alphabet: prefix with a , same numbering as states, 0 is blank

Transitions: (state, input, state, output, move), special case for start state that accepts: (q0)

Final: list of transitions

Universal TM: on input $\langle M, w \rangle$, U must halt iff M halts on w , if M is deciding/semideciding then must also accept/reject accordingly, if M computes function then $U(\langle M, w \rangle)$ must equal $M(w)$

Running UTM:

Tapes: M 's tape, $\langle M \rangle$ (program to run), encoding of M 's state (program counter)

Initialisation: copy $\langle M \rangle$ to program tape, write 1st state to state tape

Simulation: scan program for quintuple that matches current (state, input) pair, perform associated action, if no matching quintuple halt, finally report result

Church-Turing Thesis: all formalisms powerful enough to describe everything we think of as computational algorithm are equivalent

Week 11

TM Universe Definition: only strings which meet syntactic requirements of the language definition, set of syntactically valid strings is in D

Complement: with respect to universe of the same syntactic form, ex: complement of $\langle M, w \rangle$: TM M halts on input w is still in the form $\langle M, w \rangle$, but simply does not halt

Halting problem is SD: run the TM, then accept, if no halt will never accept

Halting problem is not D: contradiction, if decidable there will be TM to do so (if halt accept, else reject), but if $M_H(\langle Trouble, Trouble \rangle)$ is run ($Trouble(x)$: if M_H accepts $\langle x, x \rangle$ then loop forever, else halt) then if M_H returns true $Trouble$ loops, but if return false then $Trouble$ does halt

If H is in D all SD would be in D: there will be semideciding TM, if H in D there will be oracle O to decide it, then we can run O on $\langle M_L, w \rangle$, if O accepts (will halt), then run, if it accepts then accept, else reject, if will not halt then reject

Set of CFL is proper subset of D: CFLs decidable, so subset, & at least 1 language is decidable but not CF ($a^n b^n c^n$)

There are languages not in SD: countably infinite number of SD (enumerate TM) over nonempty alphabet, but uncountably infinite languages (power set)

SD extra example: email of someone who will respond to message in group

SD not closed under complement: if can semidecide both it and complement then can decide it, would be in D, contradiction

$\neg H$ is not in SD: H is in SD but not D, if $\neg H$ in SD then H would be in D, but it's not, so can't be in SD

Languages not in D: H , $\neg H$, does M halt on empty tape, is there any string on which M halts, does M accept all strings, do 2 TMs accept same language, is language accepted by TM regular

Rice's theorem: no nontrivial property of SD languages is decidable, nontrivial is one that is not simply true for all languages or false for all

Using Rice's theorem: specify property P, show that domain is SD, show it is nontrivial

Is language regular: is $\{\langle M \rangle : L(M) \text{ is regular}\}$ in D, P is true if L is regular, false otherwise, domain is SD since it is languages accepted by TM, nontrivial because a^* is regular but $a^n b^n$ is not