Theory of Computation (CS 46)

Sets and Functions

We write 2^A for the set of subsets of A.

Definition. A set, A, is countably infinite if there exists a bijection from A to the natural numbers.

Note. This allows us to enumerate A, using the order from the bijection.

Definition. A set is countable if it is finite or countably infinite. It is uncountable otherwise.

Theorem. If A is a subset of a countable set, then A is countable.

Theorem. $\mathbf{N} \times \mathbf{N}$ is countable.

Corollary. A countable union of countable sets is countable.

Programs

Definition. A program is a finite string (ordered set of symbols) over a finite alphabet.

Note. This is distinct from the process associated with the program, which may be infinite.

Theorem. The number of programs is countable.

Proof. The number of strings of length k is finite, and the set of all programs is the union of this countable number of countable sets.

Theorem. The number of functions from the natural numbers to {0,1} is uncountably infinite.

Corollary. There are more functions than possible programs. Thus, some functions are not computable.

Languages

Definition. An alphabet (Σ) is a finite set of symbols. A string is a finite sequence from an alphabet. The set of all strings over an alphabet is denoted Σ^*.

- $|w|$ is the length of the string w.
- a^k = aa … a (k times)

Definition. A language is a subset of Σ^*.

The following operations are defined on languages:

- $L_1 + L_2$ is the union of the two languages
- $L_1 \bullet L_2 = \{l_1 l_2 \mid l_1 \in L_1 \text{ and } l_2 \in L_2\}$ is the concatenation of two languages.
- $L_1^* = \cup L_1^i$ for all $i \geq 0$.

Definition. A regular expression over the alphabet Σ is a string over $\{(,), \varnothing, \varepsilon, +, *\} \cup \Sigma$ obtained as follows:

- $\varnothing$, ε are regular expressions
- If $a \in \Sigma$ then a is a regular expression.
- If α, β are regular expressions then so are $(\alpha\beta)$, $(\alpha+\beta)$, α^*.
- Nothing else is a regular expression

Regular expressions correspond to regular languages by $L(\varnothing) = \varnothing$, $L(x) = \{x\}$, and the operations act as expected.

Definition. A regular language is any language that can be represented by a regular expression.

Finite Automata

Definition. A deterministic finite automaton is a 5-tuple, $(K, \Sigma, \delta, s, F)$, where K is the (finite) set of states, Σ is a (finite) alphabet, $\delta: K \times \Sigma \rightarrow K$ is the transition function, s is the start state, and $F \subset K$ is the set of final states. A string is accepted if, starting at s, the automaton transitions so that when the string has been reduced to an empty string, the automaton is in a state of F.

Note. Any future actions of an automaton are determined by the remaining input string and the current state; this, all "memory" must be included in the current state. (This suggests that a finite automaton has a finite amount of memory.

Definition. A non-deterministic finite automaton is a 5-tuple, $(K, \Sigma, \delta, s, F)$, where K is the (finite) set of states, Σ is a (finite) alphabet, $\delta: K \times \Sigma \rightarrow K$ is the transition relation, s is the start state, and $F \subset K$ is the set of final states. (There may be more than one state (or 0 states) which is the result of some pair.) A string is accepted if any possible run of the string ends in some state in F.

Note. We may also consider a non-deterministic finite automaton's transition function as from subsets of K and letters to other subsets of K.

Theorem. A language is regular if and only if it is accepted by a finite automaton.

Theorem. The set of languages accepted by deterministic finite automata equals the set of languages accepted by non-deterministic finite automata.

Pumping Lemma. If L is regular, then there exists $N > 0$ such that when $|w| > N$ and $w \in L$ then we may write $w = xyz$ such that $|xy| < N$ and $xy^iz \in L$ for all $i \geq 0$.

Proof (sketch). Let N be the number of states (in the associated deterministic finite automaton). Then, in reading the string, the automaton must pass through some state twice. Let y be the part of the string falling between the first time that state is seen and the second time that state is seen. Then, that part may be removed or repeated without affecting if the string is accepted.

Algorithm. We may convert any DFA to a regular expression. Let R(i, j, k) be the regular expressions that allow the automaton to pass from q_i to q_j through no state of number higher than k. Then, $R(i, j, k+1) = R(i, j, k) + R(i, k+1, k)R(k+1, k+1, k)^*R(k+1, j, k)$. The full regular expression is $R(q_{start}, q_{final}, |K|)$. (In order to make a single final state, add an ε-transition from every state in F to a new final state.)

Phrase Structure and Context-Free Grammars / Pushdown Automata

Definition. A phrase structure grammar is $G = (V, T, R, S)$, where $T \subset V$ is the set of terminals, $V - T$ is the set of non-terminals, $R: (V^* - T^*) \rightarrow V^*$ is a set of production rules and $S \in V - T$ is the start symbol.

Definition. A context-free grammar is a phrase structure grammar in which $R: V - T \rightarrow V^*$ (all productions are from a single non-terminal).

Definition. Let $G = (V, T, R, S)$ be a grammar. Let $\alpha, \beta \in V^*$. Then α directly derives β if $\alpha = \alpha_f\alpha_m\alpha_l$ and $\beta = \alpha_f\beta_m\alpha_l$ and $(\alpha_m \rightarrow \beta_m) \in R$. (The substrings may be empty.) derives is the reflexive, transitive closure of directly derives.

Definition. If $G = (V, T, R, S)$ is a grammar then the language generated by G is given by $L(G) = \{w \in T^* \mid S \text{ derives } w\}$.

Definition. A context free grammar is <u>right linear</u> if all productions are of the form $A \rightarrow wB$ or $A \rightarrow w$, where $w \in T^*$, $A, B \in V - T$.

Theorem. Right linear grammars generate precisely the regular languages.

Definition. A CFG is in Graibach Normal Form if $R: (V - T) \rightarrow T(V - T)^*$. (Every rule sends a non-terminal to a single terminal and a string of non-terminals.)

Note. Any context-free language can be generated by a grammar in Graibach Normal Form.

Definition. A <u>pushdown automaton</u> is a finite automaton with a stack, so that transitions are determined in part by the top of the stack and may affect the stack. Formally, $M = (K, \Sigma, \Gamma, \Delta, s, F)$ where K is a set of states, Σ is the alphabet, Γ is the set of stack symbols, $\Delta: K \times (\Sigma + \{\varepsilon\}) \times \Gamma^* \rightarrow K \times \Gamma^*$ is a relation, s is the starting state and F is the set of final states. A pushdown automaton accepts if it is in an accepting state with an empty stack when the string terminates.

Note. It is equivalent if the automaton accepts if it has an empty stack when the string terminates or if it is in a final state when the string terminates.

Note. Pushdown automata are generally non-deterministic. In fact, the set of languages accepted by deterministic automata are strictly contained in the set of languages accepted by pushdown automata.

Theorem. The set of languages accepted by pushdown automata is the set of context free languages.

Definition. A grammar in Chomsky normal form has $R: (V - T) \rightarrow ((V - T)^2 + T)$. All rules have either two non-terminals or one terminal on the right hand side.

Theorem. Most CFG's may be transformed into Chomsky normal form (possibly losing ε).

Note. A grammar in Chomsky normal form may be parsed in $O(n^3)$ steps.

Pumping Lemma II. If L is a context-free language there exists $K > 0$ such that if $w \in L$ and $|w| > K$ we may write $w = uvxyz$, $|vy| \geq 1$, $|vxy| < K$ such that $uv^ixy^iz \in L$ for all $i \geq 0$.

Proof (sketch). Let $K = 2^{|V - T| + 1}$. Consider the parse tree of w. The height of this parse tree is greater than $|V - T| + 1$, which is the number of non-terminals. Thus, there is at least one path with a repeated non-terminal. Choose the terminal with lowest "next-to-last" appearance (A). Suppose the last A yields and the previous A yields vAy. Then, we find that S yields uAz which yields uv^nxy^nz for any n.

Turing Machines

Definition. A Turing Machine is $M = (K, \sum, \Delta, s, H)$ where K is a set of states, $H \subseteq K$ is the set of halting states, $\sum$ is the alphabet, which includes > (the left-end marker), _ (the blank symbol), $\Delta: (K - H) \times \Sigma \rightarrow K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$ which sends the current state and symbol being read to a new state and either symbol or move left/right. Note that $\delta(q, >) = (q, \rightarrow)$ for all $q \in K - H$.

Note. One way to describe complex Turing machines is by linking simple Turing machines with arrows that may depend on the symbol seen.

Theorem. A Turing Machine that decides languages may be simulated by a 2-stack PDA.

Proof. Consider popping the input onto one stack and moving the top symbol from one stack to the other to move the tape head. After dealing with the infinite blanks at the end and the left-end marker, this PDA can do anything the Turing machine does.

(Also, a 3-tape Turing Machine can simulate a 2-stack PDA by having one tape for the input and one for each stack.)

Extensions and Other Definitions. (All are equivalent to one-tape Turing machines. All simulations, except for the non-deterministic one, can be done in polynomial time.)

- Instead if having a left-end marker, the Turing machine might always halt in a non-accepting state if it falls off the left end of the tape.
- A tape that is infinite in both directions.
- The ability to write and move in the same step (this might decrease the number of states slightly).
- Two-Track Turing Machine: There is a second space below the input (always exactly lined up) that can be written on. (This could be simulated by a regular Turing machine using the alphabet $\Sigma \times \Sigma$.)
- Two-Tape Turing Machine. There are two tapes, and the read/write head may look anywhere on each. (This may be simulated by a 4-track Turing machine, where the first and third tracks have the contents of tape 1 and 2, and the second and fourth tracks have markers for the current location of the heads. Then, each computation would begin by identifying what each head would be seeing, and going from there.)
- Two-Head Turing Machine: Two heads on the same tape.
- Two-Dimensional Tapes
- Random Access Turing Machines: The machine may jump to any square on the tape. This machine makes use of one infinite tape and various registers that hold integers. (This, too, may be simulated by a one-tape Turing Machine, in polynomial time – both in the number of states and in the running time.)
    - Since Random Access Turing Machines have both random access and possibly infinite length words, they are more powerful than computers.
- Non-deterministic Turing Machines: These machines accept if any possible sequence of states accepts/halts.
    - Simulation by a Deterministic 3-Tape Turing Machine: Let k be the maximum number of options for $\delta(q, a)$ over all q, a. Let the first tape hold the original input to M (so that it does not get destroyed.) Let the second tape hold the current simulation of M. Let the third tape hold stings over an alphabet with k letters. This machine goes through all the possible strings over the alphabet with k letters, in order of increasing length, pausing after each is written, so that the second tape may copy the input to M, and simulate M on the second tape, making the choices designated by the string on the third tape, This machine halts if and only if M halts on any path.

Definition. Let M be a Turing Machine with $H = \{y, n\}$. M <u>accepts</u> an input if it halts in state y. M <u>rejects</u> an input if it halts in state n. M <u>decides</u> a language, L, if M accepts whenever $w \in L$ and rejects otherwise. (M *always* halts.) A language decided by a Turing machine is called <u>recursive</u>.

Note. The alphabet of the language and the Turing machine are not the same – the Turing machine alphabet must contain _ and >, and may contain other symbols as well.

Definition. A language is <u>recursively enumerable</u> if there exists a Turing machine that halts on an input if and only if it is in the language.

Theorem. Any recursive language is recursively enumerable.

Proof. Make the "no" state a non-halting state.

Theorem. The complements of recursive languages are also recursive.

Proof. Reverse the "yes" and "no" states.

Theorem. Recursive and recursively enumerable languages are closed under union, concatenation, and Kleene star,

Definition. A function, f, is recursive if there is a Turing machine that, when given an input, w, halts with f(w) on its tape.

Definition. A primitive recursive function can be created through composition and recursive definition from the basic functions of the zero function ($f(n_1, \ldots, n_k) = 0$), the j^{th} identity function ($f(n_1, \ldots, n_k) = n_j$), and the successor function ($f(n) = n+1$). A primitive recursive predicate is a function that takes on only 0 and 1.

Definition. The minimalization of a function is the smallest such number such that a predicate holds (or 0 if none exists). A function is minimalizable if there is always another number that will make the predicate true. A function is μ-recursive if it can be obtained from the basic functions by composition, recursive definition, and minimalization of minimalizable functions.

Universal Turing Machines

Definition. A Universal Turing Machine is a Turing Machine that takes in an encoding of any other Turing Machine and input and simulates that Turing machine on the input.

Example. One way to design this is to encode each state as qxxx (where xxx is in binary, with enough initial 0's on some that all encodings of states have the same length), each letter in the alphabet as axxx (with special encodings for >, _, →, and ←), and each transition in the form (old state, input, new state, output). Then, M_U has 3 tapes. It copies the description, "M", onto its second tape and shifts "w" to the beginning of the first tape. It writes the current state on the third tape. For each move, it scans the current input (which is taking up multiple tape squares) and current state, finds the corresponding move, and executes it, changing both the first tape and the state on the third tape if necessary.

Theorem. Let K = {x | x is not the encoding of a Turing Machine or x is the encoding of the Turing Machine, M, such that M does not halt on x}. K is not recursively enumerable.

Proof. Suppose K is recursively enumerable. Then there is some Turing machine, M*, such that M* halts on the language K. If M* ∈ K, then M* halts on "M*". Since "M*" is a valid encoding of a Turing machine, M* must not halt on "M*" if it is in K. On the other hand, if M* is not in K, then M* must halt on itself. However, then "M*" is in K. This is a contradiction. So K is not recursively enumerable.

Theorem. Let H_1 be the set of all Turing Machines such that M halts on "M". Then, H_1 is recursively enumerable but not recursive.

Proof. The universal Turing Machine halts on "M""M" if and only if "M" ∈ H_1. Thus a machine that accepts H_1 is one that copies the encoding of M and runs the universal Turing machine on it. Hence, it is recursively enumerable. However, its complement (K above) is not recursively enumerable. Since recursive languages are closed under complement, H_1 is not recursive.

Theorem. Let H = {"M""w" | M halts on w}. H is recursively enumerable but not recursive.

Proof. The universal Turing Machine halts on H. However, $H_1H_1 \subset H$. If H were recursive, H_1H_1 (and thus H_1) would be recursive. Since it is not, H is not recursive.

Corollary. Recursively enumerable languages are not closed under complement or intersection.

Undecidability and the Halting Problem

Definition. A problem or language is <u>undecidable</u> if there is no Turing Machine that decides it. (Not semidecides! This machine must halt on all inputs.) This means that there is no general algorithm to deal with this problem.

Definition. Let L_1 and L_2 be languages in Σ^*. A <u>many-one reduction</u> from L_1 to L_2 is a recursive function $\tau: \Sigma^* \rightarrow \Sigma^*$ such that $x \in L_1$ if and only if $\tau(x) \in L_2$. If there is a many-one reductions from L_1 to L_2, we write $L_1 \leq_M L_2$.

Theorem. If $L_1 \leq_M L_2$ and L_1 is not recursive, then L_2 is not recursive.

Proof. Suppose L_2 is recursive. Then there exists M_2 that decides L_2. Let R be the Turing machine that computes τ. Then the composition of R and M_2 decides L_1. Since L_1 is undecidable, this leads to a contradiction. So L_2 is not decidable.

Some More Undecidable Problems

- *Theorem.* Given a Turing Machine, M, the question of whether M halts on the empty tape is undecidable.
    - *Proof.* Let M_w be the machine that, when started on the empty tape, write w on its tape and then simulates M. M_W halts on the empty tape if and only if M halts on w. Since "M" "w" may be transformed into "M_w" by a Turing machine, we see that deciding if M halted on the empty tape would decide the halting problem. Since that is impossible, we may not decide if M halts on the empty tape either.
- $L(M) = \Sigma^*$ is undecidable.
- $L(M_1) = L(M_2)$ and $L(G_1) = L(G_2)$
- Given M, the question of whether $L(M) = \varnothing$ is undecidable.
- Whether M ever reaches a particular state or writes a particular symbol.
- *Post Correspondence Problem:* Given $\{(u_1, v_1) \ldots, (u_n, v_n)\}$, is there any way to arrange the pairs so that $u_{j1} \ldots u_{jn} = v_{j1} \ldots v_{jn}$?
- For any context-free grammar, G, is $L(G) = \Sigma^*$. (The Post Correspondence Problem can be reduced to this.)

Definition. A property of recursively enumerable languages is <u>trivial</u> if it is satisfied by none of the recursively enumerable languages or all of the recursively enumerable languages.

Rice's Theorem. Every non-trivial property of recursively enumerable languages is undecidable.

Proof. Let Q be a non-trivial property of recursively enumerable languages. Let L_Q = {"M" | L(M) satisfies Q}. Without loss of generality, assume $\varnothing$ does not satisfy Q. Suppose M_L semi-decides L_Q. Let τ("M""w") = "T_{MwL}", the machine that, given any input, runs M on w and M_L on the input. Then, the input to T_{MwL} is in L_Q if and only if M halts on w. So this is a reduction from the halting problem to L_Q.

Definition. A grammar is <u>context-sensitive</u> if every rule $u \rightarrow v$ has $|u| \leq |v|$.

Definition. An in-place acceptor (linear bounded automaton) is a Turing machine that never moves more than one space beyond the length of the original input.

Theorem. Linear bounded automata accept any context-sensitive grammar.

Proof. Simulate a backward derivation of any input by checking if the input has been transformed into S and, if not, non-deterministically choosing a position in the remaining string and a rule to un-apply. If any list of rules halts with S on the string, the input was in the language.

Theorem. Any language accepted by a linear bounded automaton is context-sensitive.

Proof. Apply the proof that a phrase structure grammar can simulate a Turing machine. To remove ε-productions, attach symbols to the end-markers, so that the endmarker is replaced by the symbol (and not the empty string).

Chomsky Hierarchy (all inclusions are strict)

Name(s)	*Acceptor*	*Example*
Regular (right linear grammar)	Finite automata	
Context-Free Grammar	Pushdown automata	$a^n b^n$
Context-Sensitive Grammar	Inplace Acceptor	$a^n b^n c^n$
Recursive	Turing Machine (decides)	
Recursively Enumerable	Turing Machine (semi-decides)	Halting

Note. Context-free and regular languages that contain the empty string are not technically context-sensitive, but close enough.

Undecidable Questions

	Regular	**D-CFL**	**CFL**	**CSL**	**Recursive**	**Recursively Enumerable**
Is $w \in L$?	D	D	D	D	D	U
Is $L = \varnothing$?	D	D	D	U	U	U
Is $L = \Sigma^*$?	D	D	U	U	U	U
Is $L_1 = L_2$?	D	?	U	U	U	U
Is $L_1 \subseteq L_2$?	D	U	U	U	U	U
Is the complement the same type?	D (always yes)	D	U	?	D (always yes)	U

Complexity

Whether a problem can be solved in polynomial time generally does not depend on what system is solving the problem. The most important exception is that non-deterministic Turing machine (probably) cannot be simulated by other machines in polynomial time. (Also, sometimes problems can be solved in time polynomial in the size of the number, but not in the length of the input (binary vs. unary, for example).)

Definition. P = {L | L can be decided by a Turing Machine in time polynomial in the length of the word}.

Theorem. P is closed under union, intersection, Kleene star, concatenation, and complement.

Note. If a language is semi-decidable in a fixed number of steps, it is decidable in one more step than that (since we may answer "no" if the answer "yes" has not come before that time).

Definition. NP = {L | L can be decided by a non-deterministic Turing machine in a number of steps polynomial in the length of the words}.

Theorem. NP is closed under union, intersection, concatenation, and Kleene Star.

Definition. coNP = $\{L \mid L^C \in NP\}$.

Definition. EXP = {L | L can be decided in exponential time in the length of the input}.

Fact. If NP ≠ coNP, then P ≠ NP.

Theorem. A problem is in NP if and only if there is a succinct certificate; in other words, any possible solution is polynomial in length and can be verified in polynomial time.

Theorem. Given any problem, there is a polynomial optimization algorithm if and only if there is a polynomial yes-no algorithm for a fixed number.

Definition. Let $L_1, L_2 \subseteq \Sigma^*$. $L_1 \leq_p L_2$ if there is a polynomial time computable function $\tau: \Sigma^* \rightarrow \Sigma^*$ such that $x \in L_1$ if and only if $\tau(x) \in L_2$. τ is called a polynomial time reduction from L_1 to L_2.

Definition. L is NP-Complete if L is in NP and $L' \leq_p L$ for all $L' \in NP$.

Problem. Let a set of clauses, F, each clause with only "or" in it, be given. F is satisfiable if there is some assignment of truth values to the variables that makes each clause true (that is, at least one variable is true in each clause) – note that clauses contain both x and x^c. (This is called conjunctive normal form.)

Theorem (Cook Levin). Satisfiability is NP-Complete.

Proof. Let $M = (\{q_1, \ldots, q_s = y\}, \{x_1, \ldots, x_n\}, \Delta, q_1, \{q_{s-1}, q_s\})$ be a one-tape Turing Machine that decides some language in p(n) time. Let $w = x_{j1}\ldots x_{jn} \in \Sigma^*$. Modify M so that it runs exactly p(|w|) steps. Using the variables C_{ijt} (cell i of the tap contains x_j at the t^{th} step), S_{kt} (M is in state q_k at time t), and H_{it} (M is scanning square i at time t), create the following clauses.

- M is scanning exactly one square at each time
- Each configuration has exactly one symbol in each cell at each time (we only care about the first p(n) cells, since M does not get beyond there).
- Each configuration has exactly one state.
- At most one cell is modified in each step.
- The change of state, tape, and head position is allowed by Δ.
- The machine begins in the correct initial configuration.
- The machine ends in state y.

Since these clauses are satisfiable if and only if w is in L(M) and the clauses can be made in polynomial time, this is a polynomial time reduction from an arbitrary NP problem to Satisfiability, and Satisfiability is NP-Complete.

Some More NP-Complete Problems

- 3-coloring: Can a graph be colored with 3 colors such that no two adjacent nodes have the same color?
- Traveling Salesman Problem: In a complete, weighted graph, find the shortest patch that passes through each vertex only once.
- Hamiltonian Cycle

- Bounded Tiling Problem: Can a square be tiled according to tiles with certain rules?
    - General Tiling Problem: Tiling the first quadrant. This is undecidable, since tiles may be used to simulate a Turing machine, so that a tiling exists if and only if the machine fails to halt.
- Equivalence of regular expressions without Kleene star.
- Dominating Set: Finding a set of vertices such that every other node in the graph is adjacent to a vertex in the set.

Definition. An approximation algorithm is one that finds a solution to an optimization problem such that |value(optimal) – value(solution found)| / value(optimal) < ε for some fixed ε.

- Algorithms may exist for none, some or all ε.

Theorem. There is no general approximation algorithm for the Traveling Salesman Problem. (It is inapproximable.)

Proof. If there were, we could use it to solve a Hamiltonian cycle problem (add edges of large enough weight to a Hamiltonian cycle problem – if there is a small enough possible path, then there is a Hamiltonian cycle.

Definition. Let M be a Turing Machine where all computations halt in exactly p(|w|) steps and each step has exactly two (not necessarily distinct) choices. M is a Monte Carlo Turing Machine if at least ½ of M's computations for any x it will accept answer yes.

Definition. RP = {L | there exists a Monte Carlo Turing Machine for L}.

Theorem. The set of all composites is in RP.

Definition. ZPP = RP ∩ coRP.

We know that P ⊆ ZPP ⊆ RP ⊆ NP ⊆ EXP. We do not know which of these containments are strict. (At least one is.)