CS467 Theory of Computation

Chapter 1. Finite Automata and Regular Language

Deterministic Finite Automata (DFA): a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

(1) Q is a finite set called **states**;
(2) Σ is a finite set called the **alphabets**;
(3) $\delta: Q \times \Sigma \to Q$ is the **transition function**;
(4) $q_0 \in Q$ is the **start state**;
(5) $F \subseteq Q$ is the set of **acceptance states**.

Computation by DFA. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $w = w_1 w_2 \cdots w_n$ be a string with $w_i \in \Sigma$ for all $i \in [1, n]$. Then M **accepts** w if there exists a sequence of states $r_0, r_1, \cdots, r_n$ in Q such that

(1) $r_0 = q_0$;
(2) $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0,1, \cdots, n-1$;
(3) $r_n \in F$.

Nondeterministic Finite Automata (NFA): a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

(1) Q is a finite set called **states**;
(2) Σ is a finite set called the **alphabets**;
(3) $\delta: Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the **transition function**, where $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$;
(4) $q_0 \in Q$ is the **start state**;
(5) $F \subseteq Q$ is the set of **acceptance states**.

Computation by NFA. Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA and let w be a string. Then M **accepts** w if we can write w as $y_1 y_2 \cdots y_m$, where $y_i \in \Sigma_\epsilon$ for all $i \in [1, m]$ and a sequence of states $r_0, r_1, \cdots, r_n$ exists in Q such that

(1) $r_0 = q_0$;
(2) $r_{i+1} \in \delta(r_i, y_{i+1})$ for $i = 0,1, \cdots, m-1$;
(3) $r_m \in F$.

Recognize. For a set A, we say that M **recognize** A if $A = \{ l \mid M \text{ accepts } l \}$.

Theorem. Every NFA has an equivalent DFA, and vice versa. i.e., they recognize the same language.
Proof idea. First, construct NFA from DFA is simple. Then, consider how to construct DFA from NFA. Let Q_{DFA} be $\mathcal{P}(Q_{NFA})$, and for any state $q \in Q$, compute its *silently reachable class* $E(q)$ (the states that can be reached via ϵ-transition) Then, construct the following DFA:

(1) $Q' = \mathcal{P}(Q)$;
(2) $\delta'(R, a) = \cup \left\{ E(q) \mid q \in Q \wedge (\exists r \in R)\big(q \in \delta(r, a)\big) \right\}$;
(3) $q_0' = E(q_0)$;
(4) $F' = \{ R \subseteq Q' \mid R \cap F \neq \emptyset \}$.

Regular Language. A language is **regular** iff some finite automata (DFA/NFA) recognizes it.

Regular operators. Let A, B be languages (the subset of Σ^*). We define the following three regular operators.

(1) **Union**. $A \cup B = \{ x \mid x \in A \vee x \in B \}$;
(2) **Concatenation**. $A \circ B = \{ xy \mid x \in A \wedge y \in B \}$;
(3) **Kleene star**. $A^* = \{ x_1 x_2 \cdots x_k \mid k \geq 0 \wedge x_i \in A \}$.

Theorem. The class of regular languages is closed under $\{\cup, \circ, ^*\}$.
Proof idea. (1) parallelly combine two FAs; (2) sequentially combine two FAs; (3) add a new start state and recursively use the FA.

Lemma. The class of regular languages is closed under complementation and intersection.

(1) **Complementation**. $\overline{A} = \Sigma^* - A$;
(2) **Intersection**. $A \cap B = \{x \mid x \in A \wedge x \in B\}$.

Proof idea. (1) simply let $F' = Q - F$ is enough; (2) according to $A \cap B = \overline{\overline{A} \cup \overline{B}}$ and (1).

Regular Expression: Given alphabet Σ, we say that R is a regular expression if R is:

(1) a for some $a \in \Sigma$;
(2) ε (empty character – this is a valid string!);
(3) $\emptyset$ (empty set – no string!);
(4) $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions;
(5) $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, sometimes written as $R_1 R_2$;
(6) (R_1^*), where R_1 is a regular expression.

Note. The corresponding language defined above is: (1) $\{a\}$; (2) $\{\varepsilon\}$; (3) $\emptyset$; (4) $L(R_1) \cup L(R_2)$; (5) $L(R_1) \circ L(R_2)$; (6) $\left(L(R_1)\right)^*$.

Theorem. A language is regular iff some regular expressions describes it.
Proof idea. (1) ($\Longleftarrow$) simply by reduction using properties of FAs; (2) ($\Longrightarrow$) using the idea of Floyd-Warshall algorithm; let i denote q_i, then let $R(i,j,k)$ be the regular expression from i to j and we use the intermediate state not greater than k, then:

$$R(i,j,k) = R(i,j,k-1) \cup R(i,k,k-1)R(k,k,k-1)^*R(k,j,k-1)$$

Therefore, $L(M) = \cup \{R(1,j,n) \mid j \in F\}$.

Lemma. (*the pumping lemma for regular languages*) If A is a regular language, then there is a number p (i.e., **pumping length**) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

(1) $|y| > 0$;
(2) $|xy| \leq p$;
(3) For each $i \geq 0$, $xy^iz \in A$.

Proof idea. Let $p = |Q| + 1$, then use *pigeonhole principle* to derive a cycle of states representing y.

Problems from Formal Language Theory.

(1) **Acceptance problem**. Does a give string belong to a given language?
(2) **Emptiness problem**. Is a given language empty?
(3) **Equality**. Are two given language equal?

Problem and Answers for FAs.

(1) **Acceptance problem**. Given a FA A and a string w, does A accepts w?
Sol. Convert FA to DFA, and deterministically follow w.
(2) **Emptiness problem**. Given a FA A, is the language $L(A)$ empty?
Sol. Convert FA to DFA, and use DFS/BFS to check whether the acceptance states are reachable.
(3) **Equality**. Are two given FAs equal?
Sol. Use the lemma $L(A) = L(B) \Leftrightarrow \left(L(A) - L(B)\right) \cup \left(L(B) - L(A)\right) = \emptyset$. Since the regular language is closed under complementation and intersections, the regular language should be closed under differentiation. Therefore, the problem is reduced to the emptiness problem.

Chapter 2. Context-Free Languages

Context-Free Grammar (CFG): a 4-tuple (V, Σ, R, S), where

(1) V is a finite set called the **variables**;
(2) Σ is a finite set disjoint from V, called the **terminals**;
(3) R is a finite set of **rules**, with each rule being a variable and a string of variables and terminals, that is,
$$V \to (V \cup \Sigma)^*$$
(4) $S \in V$ is the **start variable**.

Derivations. Let u, v, w be the string of variables and terminals, and
$$A \to w \quad \in R$$
Then uAv yields uwv, written $uAv \Rightarrow uwv$. Then we say "u derives v", written $u \Rightarrow^* v$, if

(1) $u = v$, or
(2) there is a sequence $u_1, u_2, \cdots, u_k$ for $k \geq 0$ and $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$.

Context-Free Language (CFL): the language of CFG is $\{ w \in \Sigma^* \mid S \Rightarrow^* w \}$, which is a **context-free language**.

Leftmost Derivations. A derivation of a string w in a grammar G is a **leftmost derivation** if at every step the leftmost variable is the one replaced.

Ambiguity. A string w is derived **ambiguously** in a CFG G if it has two or more different leftmost derivations. Grammar G is **ambiguous** if it generates some strings ambiguously.

- Not all ambiguity can be resolved, these ambiguities are called **inherently ambiguities**. For example, $\{ a^i b^j c^k \mid i = j \vee j = k \}$.
- No effective algorithm for resolving ambiguity, and no effective algorithm for finding out whether a CFG is ambiguous. (*)
- There are standard techniques for writing an unambiguous grammar that help in most cases. (*)

Chomsky Normal Form. A CFG is in Chomsky normal form if every rule is of the form
$$A \to BC; \quad A \to a$$
where a is any terminal and A, B, C are any variables, except that B, C may not be the start variable. In addition, we permit the rule $S \to \epsilon$, where S is the start variable.

Theorem. Any CFL is generated by a CFG in Chomsky normal form.
Proof idea. (1) deal with the start situation, add a new start variable $S_0 \to S$; (2) remove every $A \to \epsilon$ in some order and combine the rules; (3) remove every $A \to B$ by substituting B; (4) replace every $A \to u_1 u_2 \cdots u_k$ where u_i is a terminal or variable with the rules only containing two variables in the right-side.

Theorem. If G is a CFG in Chomsky normal form then any $w \in L(G)$ such that $w \neq \epsilon$ can be derived from the start state in exactly $2|w| - 1$ steps.
Proof idea. $|w| - 1$ steps to generate all variables, and $|w|$ steps to replace variables with terminals.

Pushdown Automata (PDA): a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

(1) Q is a finite set called **states**;
(2) Σ is a finite set called **input alphabet**;
(3) Γ is a finite set called **stack alphabet**;
(4) $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the **transition function**;
(5) $q_0 \in Q$ is the **start state**;
(6) $F \subseteq Q$ is the set of **acceptance states**.

Computation by PDA. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA. We say M accepts input w if w can be written as $w_1 w_2 \cdots w_n$, where each $w_i \in \Sigma_\epsilon$ and sequences of states $r_0, r_1, \cdots, r_m \in Q$ and strings $s_0, s_1, \cdots, s_k \in \Gamma^*$ exist that satisfy the following three conditions:

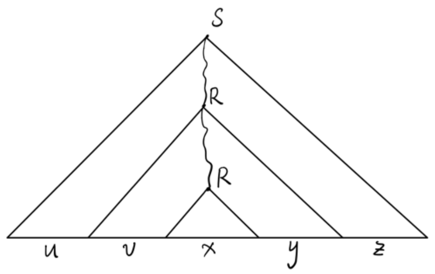
(1) $r_0 = q_0$ and $s_0 = \epsilon$;
(2) For $i = 0,1,2 \cdots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$ (*actually, we are using a stack here*).
(3) $r_m \in F$.

Theorem. A language is context-free iff some PDA recognizes it.
Proof idea. (1) CFG to PDA: write a start symbol onto the stack, rewrite variables on top of the stack (in reverse) according to the rules of CFG, and pop top terminals if it matches the input; (2) PDA to CFG: first modify CFG that only allows one push/pop at one time; then introduce state A_{ij} as "walk from state i to state j in PDA"; and introduce the rules of CFG using two matching elements of the stack.

Theorem. The context-free languages are closed under union, concatenation and Kleene star.
Proof idea. Add new transitions: (1) $S \to S_1 | S_2$; (2) $S \to S_1 S_2$; (3) $S \to S S_1 | \epsilon$.

Theorem. The intersection of a CFL with a regular language is a CFL.
Proof idea. Let $Q = Q_1 \times Q_2$ and combine two transitions (discussions about $\sigma = \epsilon$ and $\sigma \neq \epsilon$).

Lemma. (*the pumping lemma for context-free languages*) If A is a context-free language, then there is a number p (i.e., **pumping length**) where if s is any string in A of length at least p, then s may be divided as $s = uvxyz$ satisfying the conditions:



(1) For each $i \geq 0$, $uv^i xy^i z \in A$;
(2) $|vy| > 0$;
(3) $|vxy| < p$.

Proof idea. Consider the parse tree of the CFL, suppose every node can have no more than b children. If the height of the parse tree is at most h, the length of the string generated is at most b^h. Conversely, if a generated string is at least $b^h + 1$ long, then each of its parse tree must be at least $h + 1$ high. Therefore, we choose the pumping length $p = b^{|V|+1}$. Then for any string $s \in A$ with $s \geq p$, any of its parse trees must be at least $|V| + 1$ high. Consider the parse tree π with the smallest number of nodes. Then according to *pigeonhole principle*, a variable R must appear more than 1 times, and we can divide s into $uvxyz$ as the figure shown.

(1) Replace the subtree of the second R by the subtree of the first R will validate the condition;
(2) If $|vy| = 0$, then the path between two Rs can be eliminated, contradicting the "smallest" of π;
(3) We can always choose R so the last two occurrences of R falls in the bottom $|V| + 1$ "layers".

Theorem. The CFL is NOT closed under intersection or complementation.
Proof idea. (1) $\{a^n b^m c^m\} \cap \{a^m b^m c^n\}$ is not CFL; (2) according to $A \cap B = \overline{\overline{A} \cup \overline{B}}$, if CFL is closed under complementation, then it is closed under intersection, which contradicts (1).

Problems and Answers for PDA/CFG.

(1) **Acceptance problem**. Given a PDA A and a string w, does A accepts w?
Sol. Simulate the process of A with input w.
(2) **Emptiness problem**. Given a PDA A, is the language $L(A)$ empty?
Sol. Convert PDA to CFG; mark all terminals, and mark lhs of every rules if its rhs are all marked; finally check whether the start variable is marked.
(3) **Equality**. Are two given PDAs equal? **Sol**. Undecidable!

Chapter 3. Deterministic Pushdown Automata (*)

Deterministic Pushdown Automata (DPDA): a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

1. Q is a finite set called **states**;
2. Σ is a finite set called **input alphabet**;
3. Γ is a finite set called **stack alphabet**;
4. $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to (Q \times \Gamma_\epsilon) \cup \{\emptyset\}$ is the **transition function**;
5. $q_0 \in Q$ is the **start state**;
6. $F \subseteq Q$ is the set of **acceptance states**.

For every $q \in Q, a \in \Sigma, x \in \Gamma$, exactly one of the values $\delta(q, a, x), \delta(q, a, \varepsilon), \delta(q, \varepsilon, x), \delta(q, \varepsilon, \varepsilon)$ is not $\emptyset$.

Acceptance of DPDA.

1. **Accept**. If a DPDA enters an accept state after it has read the last input symbol of an input string, then it accepts that string;
2. **Reject**. In all other cases, it rejects that string. It could be one of the following cases:
   a. The DPDA reads the entire input but does not enter an accept state when it is at the end, or
   b. The DPDA fails to read the entire input string, usually caused by: DPDA tries to pop an empty stack, or DPDA makes an endless sequence of ϵ – input moves without reading any new inputs.

Deterministic Context-Free Languages (DCFL): The language of a DPDA is a DCFL.

Chap 4. The Church-Turing Thesis

(Deterministic) Turing Machine ((D)TM). A 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where Q, Σ, Γ are finite and

1. Q is a set of **states**;
2. Σ is the **input alphabet** not containing the **blank symbol** $\sqcup$;
3. Γ is the **tape alphabet** where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$;
4. $\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the **transition function**;
5. $q_0 \in Q$ is the **start state**;
6. $q_{accept} \in Q$ is the **accept state**;
7. $q_{reject} \in Q$ is the **reject state**.

Configuration. A configuration of a TM consists of

1. The current state q;
2. The current tape contents uv (there is only $\sqcup$ after v);
3. The current head location is <u>the first symbol of v</u>.

We use uqv to represent the configuration.

Special Configurations.

1. **Initial configuration (start configuration)** of M on input w is the configuration $\epsilon q_0 w$.
2. **Accepting configurations** of M are the configuration $uq_{accept}v$;
3. **Rejecting configurations** of M are the configuration $uq_{reject}v$;
4. **Halting configurations** includes the accepting configurations and rejecting configurations.

Configurations in transitions. Let $a, b, c \in \Gamma, u, v \in \Gamma^*$ and $q_i, q_j \in Q$.

1. If $\delta(q_i, b) = (q_j, c, L)$, then uaq_ibv yields uq_jacv;
2. If $\delta(q_i, b) = (q_j, c, R)$, then uaq_ibv yields $uacq_jv$.

Special cases occur when the head is at one of the ends of the configuration.

1. For left-hand end, the configuration q_ibv yields q_jcv is the transition moves left (i.e., we prevent the machine from going off the left-hand end of the tape); and it yields cq_jv when transition moves right.

(2) For right-hand end, the configuration uaq_i is equivalent to $uaq_i \sqcup$ because we assume that blanks follow the part of the tape represented in the configurations.

Computation by TM. M accepts w if there are sequence of configurations $C_1, C_2, \cdots, C_k$ such that

(1) C_1 is the start configuration of M on w;
(2) Each C_i yields C_{i+1};
(3) C_k is an accepting configuration.

Note. $C_1, C_2, \cdots, C_{k-1}$ can not be halting configurations. <u>TM stops immediately when in a halting configuration</u>.

Turing-recognizable. A language is **Turing-recognizable** if some TM (of any type) **recognizes** it.
(Turing-)decidable. A language is (**Turing-)decidable** if some TM (of any type) **decides** it.
Note. Here "of any type" means TM, multi-tape TM and NTM. *Recognize* only needs "accept"; *decide* needs both "accept" and "reject", no "loop" is allowed.

Multi-tape Turing Machine. A multi-tape TM M has several tapes:

(1) Each tape has its own head for reading and writing;
(2) The input is initially on tape 1, with all other tapes being blank;
(3) The transition function is $\delta: Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$, where k is the number of tapes, and
$$\delta(q_i, a_1, \cdots, a_k) = (q_j, b_1, \cdots, b_k, L, R, \cdots, L)$$
means that if M is in state q_i and head 1 through k are reading symbols a_1 through a_k, the machine goes to state q_j, writes symbols b_1 through b_k, and directs each head to move left, or right, or stay put as specified.

Theorem. Every multi-tape TM has an equivalent single-tape TM.
Proof idea. The tape can be multiplexed, i.e.,

[tape 1, 0][tape 2, 0]…[tape k, 0] [tape1, 1][tape2, 1]…[tape k, 1] … [tape 1, n][tape 2, n]…[tape k, n]

Nondeterministic Turing Machine (NTM). The transition function for an NTM has the form
$$\delta: Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$
And the computation of a nondeterministic Turing machine is a tree whose branches correspond to different possibilities for the machine. If *some* branch of the computation leads to the acceptance state, the machine accepts its input.
Note. Here we should notice that NTM only merge the branches by "once …, accept", and it cannot merge the branches by "once …, reject" or "if all …, …"!

Theorem. Every NTM has an equivalent DTM.
Proof idea. Use 3-tape TM (input tape, simulation tape and address tape). Simulate each possible branch in i steps and increase i step-by-step. If one branch accepts, then accept; otherwise, it may loop forever.

Enumerator. An enumerator is a Turing machine with an attached printer. The Turing machine can use that printer as an output device to print strings. Every time the Turing machine wants to add a string to the list, it sends the string to the printer.

Theorem. A language is Turing-recognizable iff some enumerator enumerates it.
Proof idea. ($\Longleftarrow$) Let E be an enumerator, and TM M checks the output of E, once w appears on the output of E, then accept; ($\Longrightarrow$) Let $s_1, s_2, \cdots$ be a list of all possible strings in Σ^*. Enumerate $i = 1,2,3, \cdots$, and run M for i steps on each input $s_1, s_2, \cdots, s_i$; if any computation accepts, then print out the corresponding s_j.

Church-Turing Thesis. Intuitive notion of algorithms = Turing machine algorithms.

Hilbert's 10th Problem.

$$D = \{\, p \mid p \text{ is a polynomial with integer coefficients and with an integral root} \,\}$$

$$D_1 = \{\, p \mid p \text{ is a polynomial on a single variable } x \text{ with integer coefficients and with an integral root} \,\}$$

Theorem. Both D and D_1 are Turing-recognizable.
Theorem (*Yuri Matijasevič, Martin Davis, Hilary Putnam, and Julia Robinson, 1970*). D is undecidable.
Theorem. D_1 is decidable.
Proof idea. There is a bound of x in D_1.

Chapter 5. Decidability

Decidable problems concerning regular languages.

(1) $A_{DFA} = \{\, \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \,\}$ (simulate);
(2) $A_{NFA} = \{\, \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \,\}$ (convert to NFA/NTM, and simulate);
(3) $A_{REX} = \{\, \langle R, w \rangle \mid R \text{ is a regular expression that generates } w \,\}$ (convert to FA, and simulate);
(4) $E_{DFA} = \{\, \langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset \,\}$ (traverse the diagram of FA);
(5) $EQ_{DFA} = \{\, \langle A, B \rangle \mid A, B \text{ are DFAs and } L(A) = L(B) \,\}$ (Use the properties of REXs and call E_{DFA}).

Decidable problems concerning context-free languages.

(1) $A_{CFG} = \{\, \langle G, w \rangle \mid G \text{ is a CFG that generates } w \,\}$ (Convert to Chomsky normal form and check all the derivations of $2|w| - 1$ steps; special check for $|w| = 0$ is needed);
(2) $E_{CFG} = \{\, \langle G \rangle \mid G \text{ is a CFG and } L(A) = \emptyset \,\}$ (Mark the terminals and repeatedly mark the lhs variable of rule whose rhs are all marked; finally check the start variable);
(3) Every CFL is decidable. (Any CFL is be generated by Chomsky normal form CFG, and call A_{CFG}).

Note. $EQ_{CFG} = \{\, \langle G_1, G_2 \rangle \mid G_1, G_2 \text{ are CFGs and } L(G_1) = L(G_2) \,\}$ is NOT decidable.

Functions. Let $f: A \to B$ be a function.

(1) f is **one-to-one** if $f(a) \neq f(a')$ whenever $a \neq a'$;
(2) f is **onto** if for every $b \in B$, there is an $a \in A$ with $f(a) = b$.
(3) A and B are same size if there is a one-to-one, onto function $d: A \to B$, then d is a **correspondence**.

Cantor's Theorem. A is countable iff it is either finite or has the same size as $\mathbb{N}$.
Theorem. $\mathbb{R}$ is uncountable.
Corollary. Some languages are not Turing-recognizable.
Proof. Fix an alphabet Σ, then Σ^* is countable, and the set of all TMs is countable (every TM can be identified with a string $\langle M \rangle$). But the set of all languages over Σ is uncountable. Therefore, there must exists langauges that can not be recognized by TM, that is, not TM-recognizable.

Theorem. A_{TM} is Turing-recognizable, but not Turing-decidable.
Proof. To prove Turing-recognizable, we construct a **universal Turing machine** U on $\langle M, w \rangle$:

1. Simulate M on w;
2. If M enters its accept state, then accept; if it enters its reject state, reject.

To prove "not Turing-decidable", we use the **diagonalization method**, assume there is a decider H for A_{TM}. Then we construct D on $\langle M \rangle$, where M is a TM.

1. Run H on input $\langle M, \langle M \rangle \rangle$;
2. Output the opposite of what H outputs, that is, if H accepts, then rejects; and if H rejects, then accept.

Therefore,

$$D(\langle M \rangle) = \begin{cases} \text{accept}, & \text{if } M \text{ does not accept } \langle M \rangle \\ \text{reject}, & \text{if } M \text{ accepts } \langle M \rangle \end{cases}$$

But consider D itself, we have

$$D(\langle D\rangle) = \begin{cases} \text{accept}, & \text{if } D \text{ does not accept } \langle D\rangle \\ \text{reject}, & \text{if } D \text{ accepts } \langle D\rangle \end{cases}$$

There comes a contradiction – What will D output on input $\langle D\rangle$?
Therefore, A_{TM} is not Turing-decidable.

co-Turing-recognizable. A language is **co-Turing-recognizable** if it is the complement of a Turing-recognizable language. For example, $\overline{A_{TM}}$ is co-Turing-recognizable.

Theorem. A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.
Proof idea. ($\Longrightarrow$) Use decider to recognize; ($\Longleftarrow$) Suppose A and $\overline{A}$ is recognized by TM M_1 and M_2, then run M_1 and M_2 parallelly to decide whether accept or reject.

Corollary. $\overline{A_{TM}}$ is not Turing-recognizable.
Proof idea. Otherwise, A_{TM} should be decidable using the previous theorem.

Chapter 6. Reducibility

Undecidable problems for language theory.

(1) $HALT_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$
Sol. Assume R decides $HALT_{TM}$, then we will exhibit a TM S on input $\langle M, w\rangle$ which decides A_{TM}:
  a. Run R on input $\langle M, w\rangle$;
  b. If R rejects, then reject;
  c. Then simulate w on M; if M accepts, then accept; if M rejects, then reject.

(2) $E_{TM} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$
Sol. Assume R decides E_{TM}. First, we construct M_1 on input x as follows.
  a. If $x \neq w$, then reject;
  b. If $x = w$, then run M on w and accept if M does.
Then we construct a TM S on input $\langle M, w\rangle$, which decides A_{TM}, as follows.
  a. Construct M_1 using M;
  b. Run R on input $\langle M_1\rangle$;
  c. If R accepts, then reject; if R rejects, then accept.

(3) $REGULAR_{TM} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$
Sol. Assume R decides $REGULAR_{TM}$. First, we construct M_2 on input x as follows.
  a. If x has the form 0^n1^n, then accept;
  b. Otherwise, run M on w and accept if M does.
Then we construct a TM S on input $\langle M, w\rangle$, which decides A_{TM}, as follows.
  a. Construct M_2 using M;
  b. Run R on input $\langle M_2\rangle$;
  c. If R accepts, then accept; if R rejects, then reject.

(4) $EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$
Sol. Assume R decides EQ_{TM}. We construct a TM S on input $\langle M\rangle$, which decides E_{TM}, as follows.
  a. Run R on input $\langle M, M^*\rangle$, where M^* is a TM that rejects all inputs;
  b. If R accepts, then accept; if R rejects, then reject.

Computation histories. Let M be a TM and w an input string. An **accepting computation history** for M on w is a sequence of configurations $C_1, C_2, \cdots, C_l$ where C_1 is the start configuration of M on w, C_l is an accepting configuration of M, and each C_i legally follows from C_{i-1} according to the rules of M.

A **rejecting computation history** for M is defined similarly, except that C_l is a rejecting configuration.

Linear Bounded Automata (LBA). An LBA is a TM wherein the tape head isn't permitted to move off the portion of the tape containing the input. If the machine tries to move its head off either end of the input, the head stays where it is.

Theorem. $A_{LBA} = \{ \langle M, w \rangle \mid M \text{ is an LBA that accepts } w \}$ is decidable.
Proof. Notice that the total number of configurations is qng^n suppose M has q states, g symbols and a tape of length n. Therefore, we just need to simulate M for at most qng^n steps; if it doesn't halt, it must be looping.

Theorem. $E_{LBA} = \{ \langle M \rangle \mid M \text{ is an LBA and } L(M) = \emptyset \}$ is undecidable.
Key idea. Construct a LBA B that only accepts the accepting computation history for M on w.
Proof. (by contradiction) Assume R decides E_{LBA}. Given a TM M and a string w, we can construct a LBA B on input x that works as follows.

(1) Breaks up x according to the delimiters into strings $C_1, C_2, \cdots, C_l$;
(2) Determines whether $C_1, C_2, \cdots, C_l$ is an accepting computation history for M on w.

Therefore, the following S decides A_{TM}. S on input $\langle M, w \rangle$:

(1) Construct an LBA B from M and w using previous methods;
(2) Run R on input $\langle B \rangle$;
(3) If R rejects, then accept; if R accepts, then reject.

Theorem. $ALL_{CFG} = \{ \langle G \rangle \mid G \text{ is a CFG and } L(G) = \Sigma^* \}$ is undecidable.
Key idea. Construct a CFG G that doesn't generate the accepting computation history for M on w.
Proof. (by contradiction) Assume R decides ALL_{CFG}. Given a TM M and a string w, we can construct a PDA D on input x and then convert it to CFG G. We want to construct a CFG G that generate all strings

(1) that do not start with C_1; or
(2) that do not end with an accepting configuration; or
(3) in which C_i does not properly yield C_{i+1} under the rule of M.

Therefore, we construct the PDA D as follows:

(1) D starts by non-deterministically branching to guess which of the three conditions to check.
(2) The first and the second are straightforward;
(3) The third branch accepts if some C_i does not properly yield C_{i+1}.
    a. It scans the input and non-deterministically decides that it has comes to C_j.
    b. It pushes C_i onto the stack until it reads the delimiter;
    c. Then D pops the stack to compare with C_{i+1}; they are almost the same except that around the head position, where the difference is dictated by the transition function of M;
    d. D accepts if there is a mismatch or an improper update.

A minor problem is that when D pops C_i off the stack, it is in reverse order. (Here notice that we are not able to store all the configurations due to the limited space in PDA). A trick is to define the computation histories as follows: $C_1; C_2^R; C_3; C_4^R; \cdots; C_l^{(R)}$. Therefore, we can directly compare the input and the top of the stack.
Therefore, the following S decides A_{TM}. S on input $\langle M, w \rangle$:

(1) Construct a PDA D using previous methods and convert it to CFG G;
(2) Run R on input $\langle G \rangle$.
(3) If R rejects, then accept; if R accepts, then reject.

Computable Functions. A function $f: \Sigma^* \to \Sigma^*$ is a computable function if some Turing machine M, on every input w, halts with $f(w)$ on its tape.

Mapping reducible. Language A is **mapping reducible** to language B, written $A \leq_m B$, if there is a computable function $f: \Sigma^* \to \Sigma^*$, where for every $w \in \Sigma^*$

$$w \in A \Leftrightarrow f(w) \in B$$

The function f is called the **reduction** from A to B.

Theorem. If $A \leq_m B$ and B is decidable, then A is decidable.
Proof idea. Using the definition of mapping reducible.
Corollary. If $A \leq_m B$ and A is undecidable, then B is undecidable.

Theorem. $A_{TM} \leq_m HALT_{TM}$.
Proof idea. Construct M' and when M rejects w, M' enters a loop.

Theorem. If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable.
Proof idea. Using the definition of Turing-recognizable.
Corollary. If $A \leq_m B$ and A is not Turing-recognizable, then B is not Turing-recognizable.

Theorem. EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable.
Proof idea. First show that $\overline{A_{TM}} \leq_m EQ_{TM}$. Construct F on input $\langle M, w \rangle$:

1. Construct the following machines M_1, M_2: M_1 rejects any input; M_2 accepts an input if M accepts w.
2. Output $\langle M_1, M_2 \rangle$.

Then we show that $\overline{A_{TM}} \leq_m \overline{EQ_{TM}}$, which is equal to $A_{TM} \leq_m EQ_{TM}$. Construct G on input $\langle M, w \rangle$:

1. Construct the following machines M_1, M_2: M_1 accepts any input; M_2 accept an input if M accepts w.
2. Output $\langle M_1, M_2 \rangle$.

Thus, according to the previous theorem, EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable.

Chapter 7. Complexity Theory

Time Complexity. Let M be a deterministic Turing machine that halts on all inputs. The **running time** or **time complexity** of M is the function $f: \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

Big-$\mathcal{O}$ Notation. Let $f, g: \mathbb{N} \to \mathbb{R}^+$ be two functions. Say that $f(n) = \mathcal{O}(g(n))$ if positive integers c, n_0 exist such that for every integer $n \geq n_0$,

$$f(n) \leq cg(n)$$

When $f(n) = \mathcal{O}(g(n))$, we say that $g(n)$ is an **upper bound** for $f(n)$, or more precisely, that $g(n)$ is an **asymptotic upper bound** for $f(n)$, to emphasize that we are suppressing constant factors.

- n^c for $c > 0$ is a **polynomial bound**;
- $2^{(n^\delta)}$ for $\delta > 0$ is an **exponential bound**.

Small-*o* Notation. Let $f, g: \mathbb{N} \to \mathbb{R}^+$ be two functions. Say that $f(n) = o(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

In other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists, where $f(n) < cg(n)$ for all $n > n_0$.

Time Classes. Let $t: \mathbb{N} \to \mathbb{R}^+$ be a function. Define the **time complexity class** $\text{TIME}(t(n))$ to be the collection of all languages that are decidable by an $\mathcal{O}(t(n))$ time Turing machine.

Theorem. Every language that can be decided in $o(n \log n)$ time on a single-tape Turing machine is regular.

Theorem. Let $t(n)$ be a function with $t(n) \geq n$. Then every $t(n)$ time multi-tape Turing machine has an equivalent $\mathcal{O}(t^2(n))$ single-tape Turing machine.
Proof idea. Simulate an TM M with k tapes by a single tape S; S store the tape in this order:

$$tape_{1,0}, tape_{2,0}, \cdots, tape_{k,0}, tape_{1,1}, tape_{2,1}, \cdots, tape_{k,1} \cdots \cdots$$

And S keeps track of the locations into the head by writing a tape symbol with a dot above it to mark the place where the head is on that tape would be.

Nondeterministic machine running time. Let N be a nondeterministic Turing machine that is a decider. The **running time** of N is the function $f: \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.

Theorem. Let $t(n)$ be a function with $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{\mathcal{O}(t(n))}$ time deterministic single-tape Turing machine.
Proof idea. Simulate an NTM N by a DTM D of three tapes (input tape, simulation tape, address tape). D try all possible branches of N's nondeterministic computation. If D ever finds the accept state on one of these branches, it accepts. Suppose every node in the tree can have at most b children, the total number of leaves in the tree is at most $b^{t(n)}$, and the total running time is $t(n)b^{t(n)} = 2^{\mathcal{O}(t(n))}$. Then convert 3-tape D to a single-tape TM, $\left(2^{\mathcal{O}(t(n))}\right)^2 = 2^{\mathcal{O}(t(n))}$.

P. P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$P = \bigcup_{k \in \mathbb{N}} \text{TIME}(n^k)$$

(1) P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine.
(2) P roughly corresponds to the class of problems that are realistically solvable on a computer.

Reasonable encodings. We use $\langle \cdot \rangle$ to indicate a reasonable encoding of one or more objects into a string.

- Note that *unary encoding* of n using n 1s is exponentially larger than the standard binary encoding of n, hence not reasonable.
- A graph can be encoded either by listing its nodes and edges, i.e., its adjacency list, or its adjacency matrix, where the (i,j)th entry is 1 if there is an edge from node i to node j and 0 if not.

Classic P problems.
(1) $PATH = \{\langle G, s, t \rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\} \in P$ (DFS);
(2) $RELPRIME = \{\langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime}\} \in P$ (Euclid algorithm);

Theorem. Every context-free language is a member of P.
Proof idea. Convert the context-free language into a context-free grammar in Chomsky normal form. Let w be an input string and $n = |w|$. For every $i \leq j \leq n$ we will compute

$$\text{table}(i,j) = \text{the collection of variables that can generate the substring } w_i w_{i+1} \cdots w_j$$

Update the table in correct order, then we only needs to check whether $S \in \text{table}(1, n)$.

Verifier. A **verifier** for a language A is an algorithm V, where

$$A = \{ w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c \}$$

We measure the time of a verifier only in terms of the length of w, so a polynomial time verifier runs in polynomial time in the length of w. A language A is polynomial verifiable if it has a polynomial time verifier. The string c in the above definition is a **certificate**, or **proof**, of membership in A. For polynomial verifiers, the certificate has polynomial length in the length of w.

NP. NP is the class of languages that have polynomial time verifiers.

Theorem. A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.
Proof idea. ($\Longrightarrow$) Non-deterministically select the certificate c and run the verifier on $\langle w, c \rangle$; ($\Longleftarrow$) V on input $\langle w, c \rangle$: first simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step; then if this branch of N's computation accepts, accept; otherwise, reject.

Nondeterministic time complexity classes.

$$\text{NTIME}(t(n)) = \{ L \mid L \text{ is a language decided by an } \mathcal{O}(t(n)) \text{ time nondeterministic Turing machine} \}$$

Corollary.

$$NP = \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k)$$

Classic NP Problems.

(1) **Hamilton path**. A Hamilton path in a directed graph G is a directed path that goes through each node exactly once.

$$HAMPATH = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph with a Hamilton path from } s \text{ to } t \}$$

Theorem. $HAMPATH \in NP$. (The certificate is the Hamilton path from s to t)

(2) **The clique problem**. A clique in an undirected graph is a subgraph, wherein every two nodes are connected by an edge. A k-clique is a clique that contains k nodes.

$$CLIQUE = \{ \langle G, k \rangle \mid G \text{ is an undirected graph with a } k - \text{clique} \}$$

Theorem. $CLIQUE \in NP$. (The certificate is the subgraph of k nodes)

(3) **The subset sum**.

$$SUBSET - SUM = \{ \langle S, t \rangle \mid S = \{x_1, x_2, \cdots, x_k\} \text{ and for some } \{y_1, y_2, \cdots, y_l\} \subseteq S, \text{we have } \sum_{i \in [l]} y_i = t \}$$

Theorem. $SUBSET - SUM \in NP$. (The certificate is $\{y_1, y_2, \cdots, y_l\}$)

Intuitive for P and NP.

(1) P = the class of languages for which membership can be decided quickly;
(2) NP = the class of languages for which membership can be verified quickly;
(3) $P \subseteq NP$, but $P = NP$ or $P \subset NP$?

Chapter 8. NP-Completeness

Satisfiability Problem. The satisfiability problem is to test whether a Boolean formula is satisfiable, i.e.,

$$SAT = \{ \langle \varphi \rangle \mid \varphi \text{ is a satisfiable Boolean formula} \}$$

Where, Boolean variables are assigned to TRUE(1) or FALSE(0); Boolean operations are AND, OR and NOT; a Boolean formula is an expression involving Boolean variables and operations; a Boolean formula is satisfiable if some assignment makes the formula evaluate to 1.

Polynomial time computable. A function $f: \Sigma^* \to \Sigma^*$ is a **polynomial time computable function** if some polynomial time Turing machine exists that halts with just $f(w)$ on its tape, when started on any input w.

Polynomial time mapping reducible. Let $A, B \subseteq \Sigma^*$, then A is **polynomial time mapping reducible**, or simply **polynomial time reducible**, to B, written $A \leq_P B$, if a polynomial time computable function $f: \Sigma^* \to \Sigma^*$ exists, where for every w,

$$w \in A \Leftrightarrow f(w) \in B$$

The function f is called the **polynomial time reduction** of A to B.

Theorem. If $A \leq_P B$ and $B \in P$, then $A \in P$.

3SAT. A **literal** is a Boolean variable or a negated Boolean variable; a **clause** is several literals connected with $\vee$s; a Boolean formula is in **conjunctive normal form**, called a **cnf-formula**, if it comprises several clauses connected with $\wedge$s; a Boolean formula is a 3cnf-formula if all the clauses have three literals. Let

$$3SAT = \{\langle\varphi\rangle \mid \varphi \text{ is a satisfiable 3cnf} - \text{formula}\}$$

Theorem. $3SAT \leq_P CLIQUE$.
Proof idea. Transform each clause to 3 nodes representing literals; then connect nodes between different triples except that x_i and $\overline{x_i}$.

Definition. A language B is NP-complete if it satisfies two conditions:

(1) B is in NP;
(2) Every A in NP is polynomial time reducible to B (if merely satisfies this condition, then B is **NP-hard**).

Theorem. If B is NP-complete and $B \in P$, then $P = NP$.
Proof idea. Reduce every NP problem to B and call the solver.

Theorem. If B is NP-complete and $B \leq_P C$ for some C in NP, then C is NP-complete.
Proof idea. Reduce every NP problem first to B, then reduce B to C.

Theorem (*Cook, Levin*). SAT is in NP-complete.
Proof idea. First it is easy to see that SAT is in NP.

Let N be an NTM that decides a language A in time n^k for some $k \in \mathbb{N}$. We show that $A \leq_P SAT$. We construct a **tableau** for N on w, which is an $n^k \times n^k$ table whose rows are the configurations of the branch of the computation of N on input w. The first row of the tableau is the starting configuration of N on w, and each row follows the previous one according to N's transition function. A tableau is **accepting** if any row of tableau is an accepting configuration. Every accepting tableau for N on w corresponds to an accepting computation branch of N on w. Thus the problem of determining whether <u>an accepting tableau for N on w exists</u>.

Define cell as the matrix elements, then four conditions: $\varphi_{cell}, \varphi_{start}, \varphi_{accept}, \varphi_{move}$.

(1) Every cell must contains (and only contains) a character;
(2) The first row is the starting configuration;
(3) There exists a row which is an accepting configuration (a q_{accept} exists in the tableau);
(4) Each row of the tableau corresponds to a configuration that legally follows the preceding row's configuration according to N's rules by ensuring each 2×3 windows is legal, where **legal** means the window does not violate the actions specified by N's transition function.

Corollary. $3SAT$ is NP-complete.
Proof idea. Use tricks to convert SAT to $3SAT$.
Corollary. $CLIQUE$ is NP-complete.
Proof idea. According to $3SAT \leq_P CLIQUE$.

Classic NP-complete problems.

(1) **Vertex cover**. If G is an undirected graph, a vertex cover of G is a subset of the nodes where every edge of G touches one of those nodes.

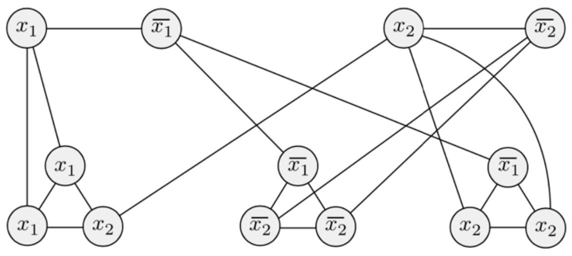
$$VERTEX-COVER = \{\langle G,k\rangle \mid G \text{ is an undirected graph that has a } k-\text{node vertex cover}\}$$

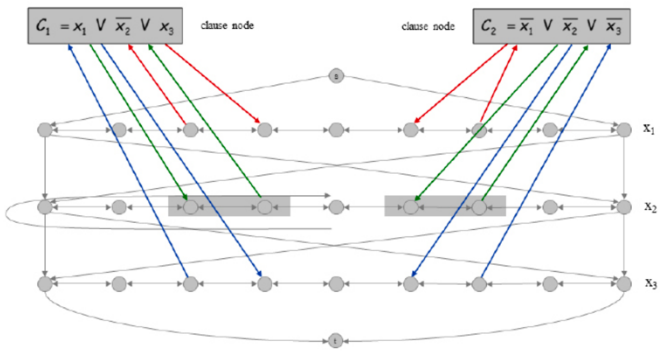
Theorem. $VERTEX-COVER$ is NP-complete.

Proof idea. Reduce $3SAT$ to $VERTEX-COVER$. We construct G as the figure shown, then

$$\varphi \in 3SAT \Leftrightarrow \langle G, m+2l\rangle \in VERTEX-COVER$$

$\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.




$C_1 = x_1 \vee \overline{x_2} \vee x_3$
clause node
clause node
$C_2 = \overline{x_1} \vee \overline{x_2} \vee \overline{x_3}$
s
x1
x2
x3
t

(2) **Hamilton path**.

Theorem. $HAMPATH$ is NP-complete.

Proof idea. Reduce $3SAT$ to $HAMPATH$. Construct G to have $2n$ Hamilton cycles. Traverse path i from left to right will set $x_i = 1$; otherwise set $x_i = 0$. For each clause, add a node and 6 edges (notice the direction!). Add an edge $t \to s$ to finish the cycle.

(3) **Undirected Hamilton path**. Hamilton path problem in an undirected graph.

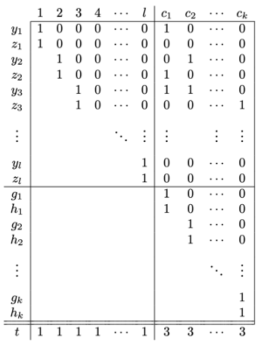
Theorem. $UHAMPATH$ is NP-complete.

Proof idea. Reduce $HAMPATH$ to $UHAMPATH$. Replace each node u by $u_{in} \leftrightarrow u_{mid} \leftrightarrow u_{out}$. Replace each edge $u \to v$ by $u_{out} \leftrightarrow v_{in}$.

(4) **The subset-sum problem**.

Theorem. $SUBSET-SUM$ is in NP-complete.

Proof idea. Reduce $3SAT$ to $HAMPATH$. Given $3SAT$ instance with n variables and k clauses, form $2n+2k$ decimal integers, each of $n+k$ digits, as illustrated in the right figure. No carries possible.



	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

$\varphi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \ldots) \wedge \cdots \wedge (\overline{x_3} \vee \ldots \vee \ldots)$.

Chapter 9. Space Complexity

Space Complexity. Let M be a DTM that halts on all inputs. The **space complexity** of M is the function $f: \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n. If the space complexity of M is $f(n)$, we also say that M runs in space $f(n)$. If M is an NTM wherein all branches halt on all inputs, we define its space complexity $f(n)$ to be the maximum number of tape cells that M scans on any branch of its computation for any input of length n.

Space complexity classes. Let $f: \mathbb{N} \to \mathbb{R}^+$ be a function.

$$\text{SPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } \mathcal{O}(f(n)) \text{ space DTM}\}$$
$$\text{NSPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } \mathcal{O}(f(n)) \text{ space NTM}\}$$

Examples. $SAT \in \text{SPACE}(n)$, $\overline{ALL_{NFA}} \in \text{NSPACE}(n)$ (non-deterministically guess input for 2^q times where q is the number of state in NFA, them mark all the possible states; if no accept state is marked, then accept).

Theorem (*Savitch*): For any function $f: \mathbb{N} \to \mathbb{R}^+$, where $f(n) > n$ (actually $f(n) > \log n$ is enough),

$$\mathrm{NSPACE}(f(n)) \subseteq \mathrm{SPACE}(f(n))$$

Proof idea.

(1) Construct CANYIELD on input c_1, c_2 and t; CANYIELD accepts iff congifuration c_1 can yield configuration c_2 in no more than t steps (just enumerate the middle configuration c_m and divide the process into two subprocess CANYIELD($c_1, c_m, t/2$) and CANYIELD($c_m, c_2, t/2$).

(2) Modify N so that when it accepts, it clears its tape and moves the head to the leftmost cell – thereby entering a configuration c_{accept}. Let c_{start} be the start configuration of N on w. We select a constant d so that N has no more than $2^{df(n)}$ configurations using $f(n)$ tape, where $n = |w|$.

(3) Then, M on input w: Output the result of CANYIELD($c_{start}, c_{accept}, 2^{df(n)}$). Then M uses space

$$\mathcal{O}\left(\log 2^{df(n)} \cdot f(n)\right) = \mathcal{O}(f^2(n))$$

PSPACE. PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine. In other words,

$$\mathrm{PSPACE} = \bigcup_{k \in \mathbb{N}} \mathrm{SPACE}(n^k)$$

NPSPACE. Similarly,

$$\mathrm{NPSPACE} = \bigcup_{k \in \mathbb{N}} \mathrm{NSPACE}(n^k)$$

Theorem. PSPACE = NPSPACE. (by Savitch's Theorem)

Lemma. A machine which runs in time t can use at most t space.
Corollary. $P \subseteq PSPACE$ and $NP \subseteq NPSPACE = PSPACE$.

Theorem. $P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$
Proof idea.Let $f: \mathbb{N} \to \mathbb{R}^+$ satisfy $f(n) > n$, then a TM uses $f(n)$ space can have at most $f(n) \cdot 2^{\mathcal{O}(f(n))}$ configurations (state(constant) * header * tape). Therefore, it must run in time $f(n) \cdot 2^{\mathcal{O}(f(n))}$. Hence,

$$PSPACE \subseteq EXPTIME = \mathrm{PSPACE} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}\left(2^{n^k}\right)$$

PSPACE-Completeness. A language B is PSPACE-completeness if

(1) B is in PSPACE, and

(2) Every $A \in PSPACE$ is polynomial time reducible to B. (if merely satisfies this condition, then B is **PSPACE-hard**)

Remark. Why not polynomial space reduction? Then every $A \in PSPACE$ is polynomial space reducible to B, where B is a language with $\emptyset \neq B \neq \Sigma^*$.

TQBF problem. A Boolean formula contains Boolean variables, the constant 0 and 1, and the Boolean operators AND, OR and NOT. The universal quantifiers $\forall$ in $\forall\varphi$ means that φ is true for every value of x in the universe; the existential quantifiers $\exists$ in $\exists\varphi$ means that φ is true for some value of x in the universe. Boolean formulas with quantifiers are **quantified Boolean formulas**. For such formula, the universe is $\{0,1\}$. When every variable of a formula appears within the scope of some quantifier, the formula is **fully quantified**. A fully quantified Boolean formula is always either true or false. The TQBF problem is to determine whether a fully quantified Boolean formula is true or false, i.e.,

$$TQBF = \{\langle\varphi\rangle \mid TQBF \text{ is a true fully quantifier Boolean formula}\}$$

Theorem. $TQBF$ is PSPACE-complete.
Proof idea. First, it's easy to see that $TQBF \in PSPACE$.

Let A be a language decided by a TM M in space n^k. We need to show $A \leq_P TQBF$. Using two collections of variables c_1, c_2 and $t > 0$, we define a formula $\varphi_{c_1,c_2,t}$. If we assign c_1 and c_2 to actual configurations, the formula is true iff M can go from c_1 to c_2 in at most t steps. Then we can let φ be the formula

$$\varphi_{c_{start},c_{accept},h}$$

Where $h = 2^{dn^k}$, where d is chosen so that M has no more than 2^{dn^k} configurations on an input of length n. We can use the methods of Cook-Levin Theorem to encode each configuration with $\mathcal{O}(n^k)$ variables. Then,

$$\varphi_{c_1,c_2,t} = \exists m_1 \left[\varphi_{c_1,m_1,\frac{t}{2}} \wedge \varphi_{m_1,c_2,\frac{t}{2}}\right]$$

This requires exponential time, but actually we can define

$$\varphi_{c_1,c_2,t} = \exists m_1 \forall (c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\} \left[\varphi_{c_3,c_4,\frac{t}{2}}\right]$$

Therefore, the size of $\varphi_{c_1,c_2,t}$ is $\log t$, bounded by $\log 2^{dn^k} = n^{\mathcal{O}(k)}$.

The formula game. Let $\varphi = \exists x_1 \forall x_2 \exists x_3 \cdots Q x_k [\psi]$. We associate a game with φ.

(1) Two players, Player A and Player E, take turns selecting the values of the variables $x_1, \cdots, x_k$;
(2) Player A selects values for the variable bounded to $\forall$;
(3) Player E selects values for the variable bounded to $\exists$.
(4) At the end, if ψ is true, then Player E wins; otherwise, Player A wins.

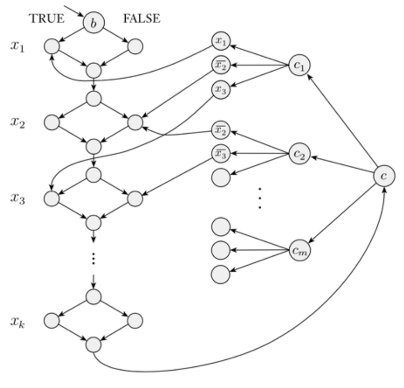
Let

$$FORMULA-GAME = \{\langle\varphi\rangle \mid \text{Player E has a winning strategy for the formula game associated with } \varphi\}$$

Theorem. $FORMULA-GAME$ is PSPACE-complete.
Proof idea. Show that $TQBF \leq_P FORMULA-GAME$.

Generalized Geography. Two players take turns to name cities from anywhere in the world. Each city chosen must begin with the same letter that ended the previous city's name. Repetition isn't permitted. The game starts with some designated starting city and ends when some player can't continue and thus loses the game.

(1) Take an arbitrary directed graph with a designated start node;
(2) Player I starts by selecting the designated start node;
(3) Then the players take turns picking nodes that form a simple path in the graph;
(4) The first player unable to extend the path loses the game.



$\exists x_1 \forall x_2 \cdots \exists x_k [(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3} \vee \cdots) \wedge \cdots]$.

Let

$$GG = \{\langle G, b\rangle \mid \text{Player I has a winning strategy for the geography game played on graph } G \text{ starting at } b\}$$

Theorem. GG is PSPACE-complete.
Proof idea. Show that $FORMULA-GAME \leq_P GG$. Construct f as the figure above shown.

Chapter 10. The classes L and NL

Machine model for L/NL. A Turing machine with two tapes: a read-only input tape and a read/write work tape; On the input tape, the input head can detect symbols but not change them. The input head must remain on the portion of the tape containing the input. The work tape may be read and written in the usual way. Only the cells scanned on the work tape contribute to the space complexity of this type of Turing machine.
Note. For sublinear space bounds, we use only this two-tape model.

L. L is the class of languages that are decidable in logarithmic space on a deterministic Turing machine.

$$L = \text{SPACE}(\log n)$$

NL. NL is the class of languages that are decidable in logarithmic space on a nondeterministic Turing machine.

$$NL = \text{NSPACE}(\log n)$$

Examples.

(1) $\{ 0^k 1^k \mid k \geq 0 \} \in L$ (Only a counter is needed);

(2) $PATH \in NL$ (Only need to record the current position, and only need to simulate for at most m steps);

Configuration (re-defined). If M is a Turing machine that has a separate read-only input tape and w is an input, a **configuration of *M* on *w*** is a setting of the state, the work tape, and the positions of the two tape heads. The input w is not a part of the configuration of M on w.

If M runs in $f(n)$ space and w is an input of length n, the number of configurations of M on w is $n2^{\mathcal{O}(f(n))}$. Now, when $f(n) \geq \log n$, we still have that the time complexity of a machine is at most exponential in its space complexity. Savitch's theorem can be extended to the sublinear space case provided that $f(n) \geq \log n$.

Log space transducer. A **log space transducer** is a TM with a read-only input tape, a write-only output-tape and a read-write work tape. The head on the output tape cannot move leftward, so it cannot read what it has written. The work tape may contain $\mathcal{O}(\log n)$ symbols.

Log space computable function. A log space transducer M computes a function $f: \Sigma^* \to \Sigma^*$, where $f(w)$ is the string remaining on the output tape after M halts when it is started with w on its input tape. We call f a **log space computable function.**

Log space reducible. Language A is log space reducible to language B, written $A \leq_L B$, if A is mapping reducible to B by means of a log space computable function f.

NL-complete. A language B is NL-complete if

(1) $B \in NL$, and

(3) Every $A \in NL$ is log space reducible to B. (if merely satisfies this condition, then B is **NL-hard**)

Remark. Why not polynomial time reduction? Then every $A \in NL$ is polynomial space reducible to B, where B is a language with $\emptyset \neq B \neq \Sigma^*$.

Theorem. If $A \leq_L B$ and $B \in L$, then $A \in L$.

Proof idea. $f(w)$ may be too large to fit within the log space bound! Therefore, suppose the log space reduction function is f, and B is decided by a TM $M_B \in L$. We build a TM M_A for A.

(1) M_A computes individual symbols of $f(w)$ as required by M_B;

(2) M_A keeps track of where M_B's input head would be on $f(w)$;

(3) Every time M_B moves, M_A restarts the computation of f on w from the beginning and ignores all the output except for the desired location of $f(w)$.

Only a single symbol of $f(w)$ needs to be stored at any point, in effect trading time for space.

We further analyze the space complexity for each part.

(1) f: obviously $\log n$;

(2) M_B: M_B uses the space of $\mathcal{O}(\log|f(w)|)$, where we only know that $|w| = n$; notice that f is a log space halted transducer, therefore its output is bounded by its time complexity, which is $|f(w)| = |w|2^{\mathcal{O}(\log|w|)}$. Therefore, M_B only uses the space of $\mathcal{O}(\log|w|) = \mathcal{O}(\log n)$.

Therefore, we only use $\mathcal{O}(\log n)$ in M_A, thus $A \in L$.

Corollary. If any NL-complete language is in L, then $L = NL$.
Proof. Say NL-complete language A is in L. Then every language in NL can be reduced to A. Since both A and log space transducer, the language should also be in L. Therefore, $L = NL$.

Theorem. $PATH$ is NL-complete.
Proof idea. First, we have proved that $PATH \in NL$. We just need to show that $PATH$ is NL-hard. For any language A in NL, say NTM M decides A in $\mathcal{O}(\log n)$ space. Given an input w, we construct $\langle G, s, t\rangle$ in log space, where G is a directed graph that G contains a path from s to t iff M accepts w.

1. Nodes of G are the configurations of M on w;
2. For configurations c_1 and c_2 of M on w, the pair (c_1, c_2) is an edge of G if c_2 is one of the possible next configurations of M starting from c_1;
3. Node s is the start configuration of M on w;
4. M is modified to have a unique accepting configuration, which is node t.

The above reduction operates in log space: there is a log space transducer T that outputs $\langle G, s, t\rangle$ on input w:

1. List the nodes of G: each node is a configuration of M on w and can be represented in $c \log n$ space (since the configurations are at most $|w|2^{\mathcal{O}(\log|w|)}$, therefore we can use $\log\left(|w|2^{\mathcal{O}(\log|w|)}\right) = \mathcal{O}(\log|w|) = \mathcal{O}(\log n)$ space to represent each node). T sequentially goes through all possible strings of length $c \log n$ and tests whether each is a legal configuration of M on w, and output those that pass.
2. List the edges of G: T tries all pairs (c_1, c_2), tests whether each is a legal configuration of M on w. Those that do are add to the output tape.

Corollary. $NL \subseteq P$.
Proof. $\forall A \in NL, A \leq_L PATH$; As any TM uses space $f(n)$ runs in time $n2^{\mathcal{O}(f(n))}$, a log space reducer also runs in polynomial time; and $PATH \in P$. Therefore A is polynomial time reducible to $PATH$ (transducer polynomial time, $PATH$ polynomial time). Therefore $NL \subseteq P$.

Theorem (*Immerman, Szelepscnyi*). $NL = coNL$.
Proof. Consider $\overline{PATH} = \{\langle G, s, t\rangle \mid \text{There is no path from } s \text{ to } t \text{ in } G\}$. Since $\overline{PATH}$ is already coNL-complete (because $PATH$ is NL-complete), we just need to show that $\overline{PATH} \in NL$ then we can derive $NL = coNL$. Suppose G has m nodes in all (represented by $[m]$). Let c be the number of nodes in G that are reachable from s. Consider the input $\langle G, s, t, c\rangle$ first. The machine M works as following. (Process I)

1. Initialize $\theta = 0$, for every node $u \in [m]$, M non-deterministically guess if u is reachable from s.
    a. If $u = t$ and the guess is YES, reject;
    b. If $u \neq t$ and the guess is YES, verify the guess: Guessing a path of length at most m from s to u.
        i. If the verifying passes: $\theta = \theta + 1$;
        ii. If the verifying fails: reject.
2. If $\theta = c$, accept; otherwise, reject.

A_i is defined as the collection of nodes that are at a distance of i or less from s. Then $A_0 = \{s\}, A_i \subseteq A_{i+1}$. Let $c = |A_i|$, then $c = \theta_m$. Obviously $\theta_0 = 1$, we will calculate θ_{i+1} from θ_i. (Process II)

1. Initialize $c_{i+1} = 0$. For every node v, repeat: $\theta_i' = 0$
    a. For every node $u \in G$, guess whether $u \in A_i$; If YES, verify the guess by guessing the path of length at most i from s to u.
        i. If verifying passes, $\theta_i' = \theta_i' + 1$; and test if $(u, v) \in G$; if YES, $\theta_{i+1} = \theta_{i+1} + 1$ and return (try another v); otherwise, return (try another u).
        ii. Otherwise, reject.
    b. If $\theta_i' \neq \theta_i$, reject.
2. Output θ_{i+1}.

Repeat Process II m times to get $c = \theta_m$, and call Process I on input $\langle G, s, t, c\rangle$.

Final Complexity Classes.

$$L \subseteq NL = coNL \subseteq P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$$