Reading Notes of
Elements of the Theory of Computation

Tianyi Cui

November 19, 2011

Introduction

Fundamental questions in computer science answered by theory of computation:

- What is an algorithm?
- What can and what cannot be computed?
- When should an algorithm be considered practically feasible.

The theory of computation is the mathematical abstractions of computers, but its origin is even before the advent of the electronic computer.

> It is based on very few and elementary concepts, and draws its power and depth from the careful, patient, extensive, layer-by-layer manipulation of these concepts – just like the computer.

1 Sets, Relations, and Languages

1.1 Sets

Power set: 2^A, the collection of all subsets of set A.

Partition of set A, subset of 2^A whose elements are nonempty and disjoint when contain all elements of A.

1.2 Relations and functions

a and b are called the *components* of the ordered pair (a, b).

The *Cartesian product* of two sets.

Ordered tuples: ordered triples, quadruples, quintuples, sextuples...

n-ary relation: unary, binary, ternary...

The domain, image, *range*, of function; one-to-one + onto = bijection; inverse.

1.3 Special types of binary relations

The relation $R \in A \times A$ is called a *directed graph*.

Properties of relations: reflexive, symmetric, antisymmetric, transitive.

Equivalence relation: r, s, t. Partial order: r, a, t. Total order.

1.4 Finite and infinite sets

Call two sets *equinumerous* if there is a bijection between them.

Finite (equinumerous with $\{1, 2, \ldots n\}$, infinite, countably infinite (equinumerous with $\mathbf{N}$), countable, uncountable.

1.5 Three fundamental proof techniques

The Principle of Mathematical Induction: Let A be a set of natural numbers such that (1) $0 \in A$, and (2) for each natural number n‘\$n\$‘, if $\{0, 1, \ldots, n\} \subseteq A$, then $n + 1 \in A$. Then $A = \mathbf{N}$.

The Pigeonhole Principle: if A and B are finite sets and $|A| > |B|$, then there is no one-to-one function from A to B.

The Diagonalization Principle: Let R be a binary relation on a set A, and let D, the diagonal set for R, be $\{a : a \in A \text{ and } (a, a) \notin R\}$. For each $a \in A$, let $R_a = \{b : b \in A \text{ and } (a, b) \in R\}$. Then D is distinct from each R_a. Lemma: the set $2^{\mathbf{N}}$ is uncountable.

1.6 Closures and algorithms

The *reflecxie transitive closure* of a directed graph.

The *rate of growth* of a function f on $\mathbf{N}$.

The proof of correctness of the Floyd algorithm: define *rank of a path* as the biggest index among its intermediate nodes, and prove that after the jth iteration, all path with rank less than or equal to j will be found.

Closure property: Let D be a set, let $n \geq 0$, and let $R \subseteq D^{n+1}$ be a $(n+1)$-ary relation on D. Then a subset B of D is said to be *closed under* R if $b_{n+1} \in B$ whenever $b_1, \ldots, b_n \in B$ and $(b_1, \ldots, b_n, b_{n+1}\} \in R$. Any property of the form "the set B is closed under relation $R_1, R_2, \ldots, R_m$" is called a *closure property* of B.

The minimal set B that contains A and has property P is unique if P is a closure property defined by relations on a set D while $A \subseteq D$. Then we call B the *closure* of A under the relation $R_1, \ldots, R_m$.

Inclusion property: unary closure (take $n = 0$ in definition).

Any closure property over a finite set can be computed in polynomial time (see ex1.6.9).

1.7 Alphabets and languages

Here is the *mathematics of strings of symbols.*

symbol: any object, but often only common characters are used.

alphabet: a finite set of symbols.

string: finite sequence of symbols from the alphabet, which has *length*, operation of *concatenation*$(\circ)$, *substring*, *prefix*, *suffix*, s^n, operation of *reversal*(s^R) defined

language: any set of strings over an alphabet Σ, that is, any subset of Σ^*. It might be able to be enumerated *lexicographically*. It has *complement* $(\overline{L})$, *concatenation of languages*, *Kleene star* (the set of all strings obtained by concatenating zero or more strings from it). We write L^+ for LL^*, which is the *closure* of L under the function of concatenation.

1.8 Finite representations of languages

This section discusses how to use *regular expressions* to represent languages.

A *regular expression* is the representation of language using empty set, characters in alphabet, concatentaion (symbol usually omitted), function of union (the *or* operator in regex), star, and parentheses. We can define the function $\mathcal{L}$ from regular expressions to lanuages, whose range is called the class of *regular languages.*

A *language recognition device* is an algorithm that is specifically designed to answer questions of the form "is string w a member of L?".

A *language generator* is the description of the way of generating members of a language.

The relation between the above two types of finite language specifications is another major subject of this book.

2 Finite Automata

2.1 Deterministic Finite Automata

DFA is computer with no memory; input a string, output indicate whether it's acceptable.

DFA definition: quituple $M = (K, \Sigma, \delta, s, F)$, where K is a finite set of *states*, Σ is an alphabet, $s \in K$ is the *initial state*, $F \subseteq K$ is the set of *final states*, and δ the *transition function*, is a function from $K \times \Sigma$ to K.

A *configuration of a DFA* is any elements of $K \times \Sigma^*$. For two configuration (q, w) and (q', w'), then $(q, w) \vdash_M (q', w')$ if and only$\vdash_M^*$ if $w = aw'$ for some symbol $a \in \Sigma$, and $\delta(q, a) = q'$. We say that (q, w) **yields** (q', w') **in one step**. Denote the reflexive transitive closure of $\vdash_M$by $\vdash_M^*$, $(q, w) \vdash_M^* (q', w')$ is read, (q, w) **yields** (q', w').

A string $w \in \Sigma^*$ is said to be *accepted* by M if and only if there is a state $q \in F$ such that $(s, w) \vdash_M^* (q, e)$. *The language accepted* by M, $L(M)$ is the set of all strings accepted by M.

2.2 Nondeterministic Finite Automata

NFAs are simply a useful *notational generalization* of finite automata, as they can greatly simplify the description of these automata. Moreover, every NFA is equivalent to a DFA.

NFA definition: quituple $M = (K, \Sigma, \Delta, s, F)$ where Δ the *transition relation*, is a subset of $K \times (\Sigma \cup \{e\}) \times K$. Each triple $(q, u, p) \in \Delta$ is called a *transition* of M.

Two finite automata M_1 and M_2 are *equivalent* if and only if $L(M_1) = L(M_2)$. For each NFA, there is an equivalent DFA.

2.3 Finite Automata and Regular Expressions

The class of languages accepted by DFA or NFA, is the same as the class of *regular languages* – those that can be described by regular expressions.

The class of regular languages are the closure of certain finite languages under the language operations of union, concatenation, and Kleene star. We can prove similar closure properties of the class of languages accepted by finite automata: union, concatenation, Kleene star, complementation (exchange the final and nonfinal states), intersection (represented as complementation and union). Therefore, a language is regular *only if* it is accepted by a finite automaton.

The other part, a language is regular *if* it is accepted by a finite automaton, can be proved by constructing a regular expression from every NFA. The way is to find all paths between initial state to some final state, then union them.

2.4 Languages that are and are not Regular

Showing languages are regular: use regular expressions or automata and operations on languages.

Theorem: Let L be a regular language. There is an integer $n \geq 1$ such that any string $w \in L$ with $|w| \geq n$ can be rewritten as $w = xyz$ such that $y \neq e$, $|xy| \leq n$, and $xy^iz \in L$ for each $i \geq 0$.

2.5 State Minimization

In language $L \subseteq \Sigma^*$, string $x, y \in \Sigma^*$. x and y are *equivalent with respect to* L, denoted $x \approx_L y$, if for all $z \in \Sigma^*$, the following is true: $xz \in L$ if and only if $yz \in L$.

Let M be a DFA, the two strings $x, y \in \Sigma^*$ are *equivalent with respect to* M, denoted $x \sim_M y$, if they both drive M from s to the same state.

If $x \sim_M y$, then $x \approx_{L(M)} y$. So $\sim_M$ is a **refinement** of $\approx_{L(M)}$.

Let $L \subseteq \Sigma^*$ be a regular language. Then there is a DFA with precisely as many states as there are equivalence classes in $\approx_L$ that accepts L. – *Can be constructively proved.*

Corollary: A language L is regular if and only if $\approx_L$ has finitely many equivalence classes.

Algorithm for state minimization: continuely refine the relation $\equiv = \approx / \sim$, initially $\equiv_0 = \{F, K - F\}$.

2.6 Algorithms for Finite Automata

Expnential: NFA to DFA, NFA to regex, decides whether two regex or NFA are equivalent.

Polynomial: regex to NFA, DFA to minimal DFA, decides whether two DFA are equivalent.

Two DFA are equivalent if and only if *their standard automata are identical.*

If L is a regular language, then there is an algorithm which, given $w \in \Sigma^*$, tests whether it is in L in $\mathcal{O}(|w|)$ time using DFA. If using NFA, the time complexity should be $\mathcal{O}(|K|^2|w|)$.

3 Context-Free Languages

3.1 Context-Free Grammars

The concepts of *laguage recognizer* and *language generator*.

A **context-free grammar** $G = (V, \Sigma, R, S)$, where V is an alphabet, $\Sigma \subseteq V$ is the set of *terminals*, $R \subseteq (V - \Sigma) \times V^*$ is the finite set of *rules*, and $S \in V - \Sigma$ is the *start symbol*.

Member of $V - \Sigma$ is called *nonterminals*. For any $A \in V - \Sigma$ and $u \in V^*$, we write $A \to_G u$ whenever $(A, u) \in R$. For any strings $u, v \in V^*$, we write $u \Rightarrow_G v$ if and only if there are strings $x, y \in V^*$ and $A \in V - \Sigma$ such that $u = xAy$, $v = xv'y$, and $A \to_G v'$. The relation $\Rightarrow_G^*$ is the reflexive transitive closure of $\Rightarrow_G$. Finally, $L(G)$ the *language generated* by G, is $\{w \in \Sigma^* : S \Rightarrow_G^* w\}$; we also say that G *generates* each string in $L(G)$. A language L is said to be a **context-free language** if $L = L(G)$ for some context-free grammar G.

When the grammar to which we refer is obvious, we write $A \to w$ and $u \Rightarrow v$ instead of $A \to_G w$ and $u \Rightarrow_G v$.

We call form $w_0 \Rightarrow_G w_1 \Rightarrow_G \cdots \Rightarrow_G w_n$ a *derivation* in G of w_n from w_0, its *length* or *steps* is n.

All regular languages are context free. We can directly construct the rules of a CFG using DFA's transition function: $R = \{q \to ap : \delta(q, a) = p\} \cup \{q \to e : q \in F\}$.

3.2 Parse Trees

Show the derivation of a CFG in a tree, called *parse tree*, which has *nodes*, each node carries a *label*, there are nodes called *root* and *leaves*. By concatenating the labels of the leaves from left to right, we obtain the derived string of terminals, which is called the *yield* of the parse tree.

Two derivations are identical except for two consecutive steps, during which the same two nonterminals are replaced by the same two strings but in opposite orders in the two derivations. The derivation in which the leftmost of the two nonterminals is replaced first is said to precede the other, written $D_1 \prec D_2$.

Two derivations D and D' are *similar* if the pair (D, D') belongs in the reflexive, symmetric, transitive closure of $\prec$. We also have *leftmost derivation* and *rightmost derivation*.

Leftmost derivation: we write $x \overset{L}{\Rightarrow} y$ if and only if $x = wA\beta$, $y = w\alpha\beta$, i.e. the leftmost nonterminal must be replaced.

Usually, we can disambiguate an ambiguous CFG, unless the language is *inherently ambiguous*.

3.3 Pushdown Automata

A *pushdown automaton* is a sextuple $M = (K, \Sigma, \Gamma, \Delta, s, F)$, where K is a finite set of *states*, Σ is an alphabet (the *input symbols*), Γ is an alphabet (the *stack symbols*), $s \in K$ is the *initial state*, $F \subseteq K$ is the set of *final states*, and Δ, the *transition relation*, is a finite subset of $(K \times (\Sigma \cup \{e\}) \times \Gamma^*) \times (K \times \Gamma^*)$.

$((p, a, \beta), (q, \gamma)) \in \Delta$, then when M is in state p with β at the top of the stack, may read a from input, replace β by γ on the top of the stack, and enter state q.\vdash

3.4 Pushdown Automata and Context-Free Grammars

Pushdown automaton is exactly what is needed to accept arbitrary context-free languages.

Construct a pushdown automaton from a context-free language.

Let $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, where Δ contains the following transitions: (1) $((p, e, e), (q, S))$, (2) $((q, e, A), (q, x))$ for each rule $A \to x$ in R, (3) $((q, a, a), (q, e))$ for each $a \in \Sigma$.

Construct a context-free language from a pushdown automaton.

Simple pushdown automaton: whenever $((q, a, \beta), (p, \gamma))$ is a transition of the pushdown automaton and q is *not* the start state, then $\beta \in \Gamma$, and $|\gamma| \leq 2$. The machine always consults its topmost stack symbol and replaces it either with e, or with a single stack symbol, or with two stack symbols. We can construct an equivalent simple pushdown automaton from any pushdown automaton. And then construct CFG.

3.5 Languages that are and are not Context-Free

The context-free languages are *closed* under union, concatenation, and Kleene star. But not closed under intersection or complementation.

The intersection of a context-free language with a regular language is a context-free language. (The Cartesian product of state set of two automaton.)

The fanout of G, denoted $\phi(G)$, is the largest number of symbols on the right-hand side of any rule in R.

Pumping theorem: Let $G = (V, \Sigma, R, S)$ be a context-free grammar Then any string $w \in L(G)$ of length greater than $\phi(G)^{|V-\Sigma|}$ can be rewritten as $w = uvxyz$ in such a way that eigher v or y is nonempty and $uv^n xy^n z$ is in $L(G)$ for every $n \geq 0$.

3.6 Algorithms for Context-Free Grammars

Polynomial algorithms: from CFG to pushdown automaton; from pushdown automaton to CFG; decide whether string $x \in L(G)$, G is a CFG.

A context-free grammar $G = (V, \Sigma, R, S)$ is said to be in *Chomsky normal form* if $R \subseteq (V - \Sigma) \times V^2$. We can transform any CFG into Chomsky normal form in polynomial time. After that, we can use dynamic programming to complete the acceptor algorithm.

3.7 Determinism and Parsing

Deterministic pushdown automaton: for each configuration there is at most one configuration taht can succeed it. Detrministic CFG are thost that are accepted by deterministic pushdown automata.

Formally, we call a language $L \subseteq \Sigma^*$ *deterministic context-free* if $L\$ = L(M)$ for some deterministic pushdown automaton M. Here $\$$ is a new symbol appended to each input string to mark its end.

The class of deterministic context-free language is *closed under complement.* The class of deterministic comtext-free language is *properly contained* in the class of context-free languages.

Top-Down Parsing: the steps in the computation where a nonterminal is replaced on top of the stack correpsond to the constrction of a parse tree from the root towards the leaves.

Left factoring: $F \to a\beta$, $F \to a\gamma$ to $F \to aE$, $E \to \beta$, $E \to \gamma$.

Bottom-Up Parsing: carry out a leftmost derivation on the stack; attempt to read the input first and, on the basis of the input actually read, deduce what derivation it should attempt to carry out.

Bottom-up pushdown automata construction: $G = (V, \Sigma, R, S)$ is the grammar, $M = (K, \Gamma, \Delta, p, F)$ is the automata, where $K = \{p, q\}$, $\Gamma = V$,$F = \{q\}$, and Δ contains the following: (1) $((p, a, e), (p, a))$ for each $a \in \Sigma$, (2) $((p, e, \alpha^R), (p, A))$ for each rule $A \to \alpha$ in R, (3) $((p, e, S), (q, e))$.

4 Turing Machines

4.1 The Definition of a Turing Machine

Unlike finite automata or pushdown autoamta, turing machine can be regarded as truly general models of computers.

Turing machines seem to form a *stable* and *maximal* class of computational devices, in terms of the computations they can perform.

A Turing machine is a quintuple $(K, \Sigma, \delta, s, H)$, where K is a finite set of *states*; Σ is an alphabet, containing the *blank symbol* $\sqcup$ and the *left end symbol* $\triangleright$, but not containing the symbols $\leftarrow$ and $\rightarrow$; $s \in K$ is the *initial state*, $H \subseteq K$ is the set of *halting states*; δ, the *transition function*, is a function from $(K - H) \times \Sigma$ to $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$ such that (a) for all $q \in K - H$, if $\delta(q, \triangleright) = (p, b)$, then $b = \rightarrow$, (b) for all $q \in K - H$ and $a \in \Sigma$, if $\delta(q, a) = (p, b)$ then $b \neq \triangleright$.

A *configuration* of a Turing machine $M = (K, \Sigma, \delta, s, H)$ is a member of $K \times \triangleright \Sigma^* \times (\Sigma^*(\Sigma - \{\sqcup\}) \cup \{e\})$. All configurations are assumed to start with the left end symbol and never end with a blank, unless the blank is currently scanned.

The definition of $(q_1, w_1\underline{a_1}u_1) \vdash_M (q_2, w_2\underline{a_2}u_2)$.

$C_1 \vdash_M^* C_2$, configuration C_1 *yields* configuration C_2. A *computation* by M is a sequence of configurations $C_0, C_1, \ldots, C_n$, for some $n \geq 0$ such that $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \cdots \vdash_M C_n$. This computation is of *length* n or it has n *steps*, and we write $C_0 \vdash_M^n C_n$.

Use a *hierarchical* notation, more and more complex machines are built from simpler materials.

The *basic machines* include the *symbol-writing machines* and the *head-moving machines.* If $a \in \Sigma$, the *a-writing machine* will be denoted simply as a. The head-moving machines L and R. Turing machines will ge combined in a way suggestive of the structure of a finite automaton.

4.2 Computing with Turing Machines

Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine, $H = \{y, n\}$ consists of two halting states, denoting *accepting configuration* and *rejecting configuration.* We say that M *decides* a language L if for any string w, $w \in L$ then M accepts w; and if $w \notin L$ then M rejects w. We call a language *recursive* if there is a Turing machine that decides it.

Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine. Suppose M halts on input w, and that $(s, \triangleright \underline{\sqcup} w) \vdash_M^* (s, \triangleright \underline{\sqcup} y)$ for some y. Then y is called the *output of M on input w*, and is denoted $M(w)$. Notic that $M(w)$ is defined *only if M halts on input w.*

Let f be any function from Σ_0^* to Σ_0^*. We say that M *computes* function f if, for all $w \in \Sigma_0^*$, $M(w) = f(w)$. A function is called *recursive*, if there is a Turing machine M that computes f.

Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine, and let $L \subseteq \Sigma_0^*$ be a language. We say that M *semidecides* L if for any string $w \in \Sigma_0^*$ the following is true: $w \in L$ if and only if M halts on input w. A language L is *recursively enumerable* if and only if there is a Turing machine M that semidecides L.

If a language is recursive, then it is recursively enumerable.

If L is a recursive language, then its complement $\overline{L}$ is also recursive.

4.3 Extensions of the Turing Machine

The additional features in Turing Machines do not add to the classes of computable functions or decidable languages, since each instance can be *simulated* by the staandard model.

k-tape Turing machine are capable of quite complex computatioinal tasks.

Let $M = (K, \Sigma, \delta, s, H)$ be a k-tape Turing machine for some $k \geq 1$. Then there is a standard Turing machine $M' = (K', \Sigma', \delta', s', H)$, where $\Sigma \subseteq \Sigma'$, and such that the following holds: For any input string $x \in \Sigma^*$, M on input x halts with output y on the first tape if and only if M' on input x halts at the same halting state, and with the same output y on its tape. Furthermore, if M halts on input x after t steps, then M' halts on input x after a number of steps which is $\mathcal{O}(t \cdot (|x| + t))$.

A two-way infinite tape can be easily simulated by a 2-tape machine.

A Turing machine with one tape and several heads or with two-dimensional tape can be simulated too.

Any function that is computed or language that is decided or semidecided by Turing machine with several tapes, heads, two-way infinite tapes, or multi-dimensional tapes, is also computed, decided, or semidecided, respectively, by a standard Turing machine.

4.4 Random Access Turing Machines

A *random access Turing machine* has a fixed number of *registers* and a one-way infinite tape; each register and each tape square is capable of containing an arbitrary natural number. The program of a random access Turing machine is a sequence of *instructions*.

A random access Turing machine is a pair $M = (k, \Pi)$, where $k > 0$ is the number of registers, and $\Pi = (\pi_1, \pi_2, \ldots, \pi_p)$, the program is a finite sequence of instructions. A configuration of a random access machine (k, Π) is a $k+2$-tuple $(\kappa, R_0, R_1, \ldots, R_{k-1}, T)$, where $\kappa \in \mathrm{N}$ is the program counter, $R_j \in \mathrm{N}$ is the current value of Register j. T is the tape contents.

Any language decided or semidecided by a random access Turing machine, and any function computable by a random access Turing machine, can be decided, semidecided, and computed, respectively, by a standard Turing machine.

4.5 Nondeterministic Turing Machines

A *nondeterministic Turing machine* is a quintuple $(K, \Sigma, \Delta, s, H)$, where K, Σ, s, and H are as for standard Turing machines, and Δ is a subset of $((K - H) \times \Sigma) \times (K \times (\Sigma \cup \{\leftarrow, \rightarrow\}))$, rather than a function.

If a nondeterministic Turing machine M semidecides or decides a language, or computes a function, then there is a standard Turing machine M' semideciding or deciding the same language, or computing the same function.

4.6 Grammars

A *grammar* (or *unrestricted grammar*, or a *rewriting system*) is a quadruple $G = (V, \Sigma, R, S)$, where V is an alphabet; $\Sigma \subseteq V$ is the set of terminal symbols, and $V - \Sigma$ is called the set of nonterminal symbols; $S \in V - \Sigma$ is the start symbol; and R, the set of rules, is a finite subset of $(V^*(V - \Sigma)V^*) \times V^*$.

A language is generated by a grammar if and only if it is recursively enumerable.

Let G be a grammar, and let $f : \Sigma^* \mapsto \Sigma^*$ be a function. We say that G *computes* f if, for all $w, v \in \Sigma^*$, the following is true: $SwS \Rightarrow_G^* v$ if and only if $v = f(w)$. The function is called *grammatically computable*, which is in turn *recursive*.