Notes on Theory of Computation[1]

Hao Su

May 31, 2025

[1]Actually notes on MIT OCW 18.404J, but indexed according to Sipser's ***Introduction to the Theory of Computation*** (3rd ed).

Contents

Chapter 1

Regular Languages

Defn 1.1 (1.15) A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called ***alphabet***,
3. $\delta : Q \times \Sigma \to Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***. ◇

Defn 1.2 A ***string*** is a finite sequence of symbols in Σ. A ***language*** is a set of strings (finite or infinite). Definition of FAs ***accepting*** strings and ***recognizing*** languages on page 40. ◇

Defn 1.3 (1.16) A language is called a ***regular language*** if some finite automaton recognizes it. ◇

Rmk 1.4 To construct an FA is to keep track of several different possibilities with one state for each. ◇

Defn 1.5 (1.23) The ***regular operations*** include ***union*** $(A \cup B)$, ***concatenation*** (AB) and ***star*** A^*, where A and B are languages. ◇

Defn 1.6 (1.52) The set of ***regular expressions*** is generated from B by regular operations (1.5), where $B = \{\emptyset\} \cup \{\{a\} | a \in \Sigma\}$. ◇

Thm 1.7 (1.25) The class of regular languages is closed under the union operation. ◇

Proof Idea. We construct an FA that keeps track of the states of *both* given FAs, and terminates when either does. Hence $Q = Q_1 \times Q_2, F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$ and so forth. ⊣

Thm 1.8 (1.26) The class of regular languages is closed under the concatenation operation. ◇

Defn 1.9 (1.37) A ***Nondeterministic*** FA is again a 5-tuple, differing from an DFA in that the transition function δ is of the form $Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$. ◇

Rmk 1.10 Ways to think about nondeterminism:

1. Computational: Fork new parallel thread and accept if any thread leads to an accept state.
2. Mathematical: Tree with branches. Accept if any branch leads to an accept state.
3. Magical: Guess at each nondeterministic step which way to go. Machine always makes the right guess that leads to accepting, if possible. ◇

Thm 1.11 (1.39) Every nondeterministic finite automaton has an equivalent deterministic finite automaton. ◇

Proof Idea. We construct a DFA M' that keeps track of the set of possible states in the given NFA M. Hence $Q' = \mathcal{P}(Q), q_0' = \{q_0\}, F' = \{R \in Q' | R \cap F \neq \emptyset\}$ and so forth. ⊣

Proof Idea for 1.8. We construct an NFA by putting in ε transitions going from F_1 to q_2. The other accomodations are immediate. ⊣

Thm 1.12 The class of regular languages is closed under the star operation. ◇

Proof Idea. We construct an NFA by putting in ε transitions going from F to q_0. The new start state q_0' is in F' and has an ε transition pointing toward q_0. We can not simply put q_0 in F' for that if $q_0 \notin F$, M' might accept some string that M does not. ⊣

Lem 1.13 (1.55) If a language is described by a regular expression, then it is regular. ◇

Proof Idea. We construct NFAs for regular expressions in B (1.6), and composite them the way described when proving that the class of regular languages is closed under regular operations. (This is indeed indution.) ⊣

Lem 1.14 (1.60) If a language is regular, then it is described by a regular expression. ◇

Proof Idea. We construct a special type of NFAs that can be easily converted to a form equivalent to a regular expression, and then describe a procedure that takes a DFA, converts it to an NFA of such type and returns a regular expression. The conversion procedure should maintain the language our automata recognize.

Defn 1.15 (1.64) A *Generalized* NFA is an NFA that allows regular expressions as transition labels. ◇

For convenience assume that (1) one accept state, separate from the start state and (2) one arrow from each state to each state, except only exiting the start state and only entering the accept state. This is the type of NFAs we ask for, and the conversion from DFAs to this type is trivial. After conversion we prove by induction that this NFA is *indeed* equivalent to a regular expression. ⊣

Thm 1.16 (1.70) **Pumping lemma:** For every regular language A, there is a number p (the "pumping length") such that if $s \in A$ and $|s| \geq p$ then $s = xyz$ where

1. $xy^iz \in A$ for all $i \geq 0$
2. $y \neq \varepsilon$
3. $|xy| \leq p$ ◇

Proof Idea. There are finite states, and a long enough input string is going to go through some state twice. This lemma essentially formalizes this. ⊣

Eg 1.17 We show a bunch of examples of proving (by contradiction) non-regularity according to 1.16, where $\Sigma = \{0, 1\}$, L is the language, $s \in L$ and p is the pumping length.

1. $L = 0^k1^k$. We choose $s = 0^p1^p$.
2. $L = \{ww | w \in \Sigma^*\}$. We choose $s = 0^p10^p1$.
3. $L = \{w | w$ has equal numbers of 0s and 1s$\}$. We can easily show that regular languages are closed under intersection, so $L \cap 0^*1^*$ is regular, which is item 1. Thus contradiction. ◇

Exercises and Problems

1.31 Show that if language A is regular, so is its *reverse* $A^{\mathcal{R}}$. ◁

Given a DFA M that recognizes A, we construct an equivalent NFA M' by adding a new state q and putting ε transitions going from F to q. So q is the unique accept state in M'. Swap the start state and q in M' and define a new δ accordingly and we have an NFA that recognizes $A^{\mathcal{R}}$.

1.32 Let

$$\Sigma_3 = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}.$$

Σ_3 contains all size 3 columns of 0s and 1s. A string of symbols in Σ_3 gives three rows of 0s and 1s. Consider each row to be a binary number and let

$$B = \{w \in \Sigma_3^* \mid \text{the bottom row of } w \text{ is the sum of the top two rows}\}.$$

Show that B is regular. ◁

By 1.31 consider $B^{\mathcal{R}}$. A DFA is straightforward that only has 3 states.

1.41 For languages A and B, let the ***perfect shuffle*** of A and B be the language

$$\{w | w = a_1 b_1 \cdots a_k b_k, \text{ where } a_1 \cdots a_k \in A \text{ and } b_1 \cdots b_k \in B, \text{ each } a_i, b_i \in \Sigma\}$$

Show that the class of regular languages is closed under perfect shuffle. ◁

Similar to the proof of 1.7, we construct an FA that keeps track of the states of both given FAs *and* parity, and terminates when both does *and* the parity is even. Hence $Q = Q_1 \times Q_2 \times \{0, 1\}$, $F = F_1 \times F_2 \times \{0\}$, δ behaves like δ_A on the set of even states and so forth.

1.51 Let x and y be strings and let L be any language. We say that x and y are ***distinguishable by*** L if some string z exists whereby exactly one of the strings xz and yz is a member of L; otherwise, for every string z, we have $xz \in L$ iff $yz \in L$ and we say that x and y are ***indistinguishable by*** L (written $x \equiv_L y$). Show that $\equiv_L$ is an equivalence relation. ◁

Omitted as trivial.

1.52 Myhill-Nerode theorem. Refer to 1.51. Let L be a language and let X be a set of strings. Say that X is ***pairwise distinguishable by*** L if every two distinct strings in X are distinguishable by L. Define the ***index of*** L to be the maximum number of elements in any set that is pairwise distinguishable by L. The index of L may be finite or infinite.

a. Show that if L is recognized by a DFA with k states, L has index at most k.

b. Show that if the index of L is a finite number k, it is recognized by a DFA with k states.

c. Conclude that L is regular iff it has finite index. Moreover, its index is the size of the smallest DFA recognizing it. ◁

Proof. **a.** If α_1 and α_2 is distinguishable by DFA M, the states M ends in upon inputs α_1 and α_2 have to differ. Thus k states indicates the index of L is not greater than k.

b. Let $X = \{s_1, \ldots, s_k\}$ be pairwise distinguishable by L. We construct DFA $M = (Q, \Sigma, \delta, q_0, F)$ with k states recognizing L. Let $Q = \{q_1, \ldots, q_k\}$, $F = \{q_i \mid s_i \in L\}$, the start state q_0 be the q_i such that $s_i \equiv_L \varepsilon$ and define $\delta(q_i, a)$ to be q_j where $s_j \equiv_L s_i a$. M is constructed so that, for any state q_i, $\{s' \mid \delta(q_0, s') = q_i\} = \{s' \mid s' \equiv_L s_i\}$. Hence M recognizes L. (I thought I had to construct according to k *strings* but I was actually given the quotient of *all strings* w.r.t. $\equiv_L$ and that actually defines the DFA *itself*.)

c. This conclusion is immediate by item a and b. ⊣

1.53 Let $\Sigma = \{0, 1, +, =\}$ and

$$ADD = \{x = y + z \mid x, y, z \text{ are binary integers, and } x \text{ is the sum of } y \text{ and } z\}.$$

Show that ADD is not regular. ◁

Suppose ADD is regular and has pumping length p for the sake of contradiction. For $y, z \in 1^p\{0,1\}^*$, of course $|x| > p$. According to 1.16 we have multiple sum of y and z, hence contradiction.

1.59 Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let h be a state of M called its "home". A ***synchronizing sequence*** for M and h is a string $s \in \Sigma^*$ where $\delta'(q, s) = h$ for every $q \in Q$, where δ' is intuitively extended onto strings. Say that M is ***synchronizable*** if it has a synchronizing sequence for some state h. Prove that if M is a k-state synchronizable DFA, then it has a synchronizing sequence of length at most k^3. Can you improve upon this bound? ◁

to do

1.60 Let $\Sigma = \{a, b\}$. $C_k = \Sigma^* a \Sigma^{k-1}$. Describe an NFA with $k + 1$ states that recognizes C_k in terms of both a state diagram and a formal description. ◁

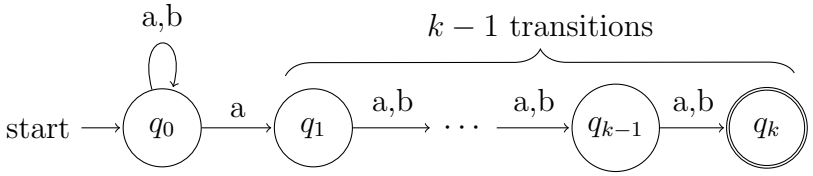

Figure 1.1: State diagram. A formal description is omitted.

1.61 Consider the languages C_k defined in Problem 1.60. Prove that for each k, no DFA can recognize C_k with fewer than 2^k states. ◁

Proof. There are 2^k distinct Myhill-Nerode equivalence classes of strings w.r.t. $\equiv_{C_k}$. They are exactly the equivalence classes of strings with suffix $b^k, b^{k-1}a, b^{k-2}ab, \ldots, a^k$ respectively. By 1.52 we are done. ⊣

Chapter 2

Context-Free Languages

Defn 2.1 (2.2) A ***context-free grammar*** G is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***,
2. Σ a finite set such that $\Sigma \cap V = \emptyset$ called ***terminals***,
3. R a finite set of ***rules*** where each rule is a funtion of the form $V \to (V \cup \Sigma)^*$ and
4. $S \in V$ the start variable.

For $u, v \in (V \cup \Sigma)^*$ write

1. $u \Rightarrow v$ (say that u ***yields*** v) if can go from u to v with one substitution step.
2. $u \stackrel{*}{\Rightarrow} v$ (say that u ***derives*** v) if can go from u to v with some number of substitions steps.
3. $u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k = v$ is called a ***derivation*** of v from u. If $u = S$ then "from u" can be omitted.

$L(G) = \{w | (w \in \Sigma^*) \wedge (S \stackrel{*}{\Rightarrow} v)\}$ is called a ***context-free language***. $\diamond$

Context-free means that we can apply substitutions regardless of the contexts.

Defn 2.2 (2.7) A derivation of a string w in a context-free grammar G is a ***leftmost derivation*** if at every step the leftmost remaining variable is the one replaced. w is derived ***ambiguously*** in G if it has two or more different leftmost derivations. G is ***ambiguous*** if it generates some string ambiguously. $\diamond$

Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language. Some context-free languages, however, can

be generated only by ambiguous grammars. Such languages are called ***inherently ambiguous***.

Defn 2.3 (2.8) A CFG is in ***Chomsky normal form*** if every rule is of the form $A \to BC$ or $A \to a$, where a is any terminal and A, B and C are any variables (except that B and C may not be the start variable). In addition permit $S \to \varepsilon$, where S is the start variable. $\diamond$

Thm 2.4 (2.9) Any context-free language is generated by a context-free grammar in Chomsky normal form. $\diamond$

Proof Idea. We basically convert an arbitrary CFG into Chomsky normal form. This is nontrivial and demands carefulness. For the proof see page 109. $\dashv$

Defn 2.5 (2.13) A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Γ is the stack alphabet and $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ the transition function. $\diamond$

Our PDA models is nondeterministic. The nondeterministic forks replicate the stack. Also, we can check if the stack is effectively empty by writing at the beginning a custom sign (say \$) at the bottom of it.

Lem 2.6 (2.21) If A is a CFL then some PDA recognizes it. $\diamond$

Proof Idea. PDA begins with starting variable and guesses substitutions. It keeps intermediate generated strings on stack. When done, compare with input. We can only substitute variables when on the top of stack. If a terminal is on the top of stack, pop it and compare with input. $\dashv$

Lem 2.7 (2.27) If a PDA recognizes A then it is a CFL. $\diamond$

Proof. Say that $P = \{Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\}\})$ and construct G. The variables of G are $\{A_{p,q} | p, q \in Q\}$. The start variable is $A_{q_0, q_{accept}}$. The set of rules R of G is described as follows.

1. For each $p, q, r, s \in Q, u \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $(r, u) \in \delta(p, a, \varepsilon) \wedge (q, \varepsilon) \in \delta(s, b, u), (A_{p,q} \to aA_{r,s}b) \in R$.
2. For each $p, q, r \in Q, (A_{p,q} \to A_{p,r}A_{r,q}) \in R$.
3. For each $p \in Q, (A_{p,p} \to \varepsilon) \in R$. $\dashv$

Thm 2.8 Pumping Lemma for CFLs: For every CFL A, there is a p such that if $s \in A$ and $|s| \geq p$ then $s = uvxyz$ where

1. $uv^ixy^iz \in A$ for all $i \geq 0$

2. $vy \neq \varepsilon$
3. $|vxy| \leq p$ ◇

Proof Idea. Let b be the length of the longest right hand side of a rule (the max branching of the parse tree). Let h be the height of the parse tree for s. A tree of height h and max branching b has at most b^h leaves, so $|s| \leq b^h$. Let $p = b^{|V|}+1$ where $|V|$ is the number of variables in the grammar. So if $|s| \geq p > b^{|V|}$ then $|s| > b^{|V|}$ and so $h > |V|$. Thus at least $|V| + 1$ variables occur in the longest path. So some variable R must repeat on a path. For item 1 we can cut and paste R arbitrarily. For item 2 we add a requirement that the parse tree has to be smallest, meaning that R to R without generating new symbols is not allowed during the construction of the parse tree. For item 3 choose the lowest repitition of R, for that if $|vxy| > p$, a lower reoccurrence would happen, contradicting "lowest". ⊣

Eg 2.9 Usages of 2.8:

1. $0^k1^k2^k$ is not a CFL. Suppose it is with pumping length p then we can pump $0^p1^p2^p$, which we cannot.
2. ww is not a CFL. Suppose it is with pumping length p then we can pump $0^p1^p0^p1^p$, which we cannot. (Use uv^0xy^0z, or any other pumping choices, as long as you take care of what is pumped.) ◇

Obviously $0^k1^k2^l$ and $0^l1^k2^k$ both are CFLs, thus we have a counterexample showing that the class of CFLs is not closed under intersection. By 2.16 it is closed under union. So it is not closed under complementation by De Morgan's law ($\overline{\overline{A} \cup \overline{B}} = A \cap B$). Why is it closed under union but not closed under intersection? Intuitively, given two CFGs, you can guess which one to use to get the union, but there is no obvious way to use them *both*. Also, given two PDAs, you can guess which of the stacks to keep, but you cannot easily simulate *both* with only one.

Exercises and Problems

2.15 Give a counterexample to show that the following construction fails to prove that the class of context-free languages is closed under star. Let A be a CFL that is generated by the CFG $G = (V, \Sigma, R, S)$. Add the new rule $S \to SS$ and call the resulting grammar G'. This grammar is supposed to generate A^*. ◁

Let $G = (\{S\}, \{a\}, \{S \to a\}, S)$, then $L(G')$ does not contain $\varepsilon \in A^*$. Another counterexample would be $G = (\{S\}, \{a, b\}, \{S \to aSa|b\}, S)$. Then $L(G')$ contains $abba \notin A^*$. This exercise tell us to be careful with the original grammar.

2.16 Show that the class of context-free languages is closed under the regular operations (1.5). ◁

Say that we are given CFGs G_1 and G_2. To get $G_\cup$ we add rule $S \to S_1|S_2$. To get $G_\circ$ we add rule $S \to S_1S_2$. To get G_* (of G_1) we add rule $S \to SS_1|\varepsilon$.

2.17 Refer to 2.16 to give another proof that every regular language is context free by showing how to convert a regular expression directly to an equivalent context-free grammar. ◁

Define a CFG G according to a regular language R. Consider the subset R_G of a regular language that can be generated by G. Obviously R_G is closed under regular operations and contains the symbols that occur in R. By induction theorem we have $R_G = R$, so R is a CFL.

2.18 (a) Let C be a context-free language and R be a regular language. Prove that the language $C \cap R$ is context free. (b) Use part (a) to show that the language $A = \{w|w \in \{a, b, c\}^*$ and contains equal numbers of a's, b's, and c's$\}$ is not a CFL. ◁

(a) We can construct a PDA whose set of states is $Q = Q_C \times Q_R$, the transition function δ defined by $\delta((q_c, q_r), a, x) = \{((q'_c, q'_r), \gamma)|(q'_c, \gamma) \in \delta_C(q_c, a, x), q'_r = \delta_R(q_r, a)\}$ and accept states $F = F_C \times F_R$. (b) If it is, so would be $a^kb^kc^k = A \cap a^*b^*c^*$, which is not by 2.8.

2.22 Let $C = \{x\#y|x, y \in \{0, 1\}^*$ and $x \neq y\}$. Show that C is context-free. ◁

to do

2.24 Show that $E = \{a^ib^j|i \neq j \wedge 2i \neq j\}$ is a context-free language. ◁

to do

2.26 Show that, if G is a CFG in Chomsky normal form, then for any string $w \in L(G)$ of length $n \geq 1$, exactly $2n - 1$ steps are required for any derivation of w. ◁

Each application of a rule of the form $A \to BC$ increases the length of the string by 1 (because B and C are never to derive ε), so we have $n - 1$ steps here. We also need n applications of terminal rules $A \to a$ to convert the variables into terminals. So $2n - 1$ steps are needed as a minimum. Note also that any one more step will lead to a string different from w.

2.27 $G = (V, \Sigma, R, \langle \text{STMT} \rangle)$ is described as follows.

$$\begin{aligned}
\langle \text{STMT} \rangle &\to \langle \text{ASSIGN} \rangle | \langle \text{IF-THEN} \rangle | \langle \text{IF-THEN-ELSE} \rangle \\
\langle \text{IF-THEN} \rangle &\to \texttt{if condition then } \langle \text{STMT} \rangle \\
\langle \text{IF-THEN-ELSE} \rangle &\to \texttt{if condition then } \langle \text{STMT} \rangle \texttt{ else } \langle \text{STMT} \rangle \\
\langle \text{ASSIGN} \rangle &\to \texttt{a:=1}, \\
\Sigma &= \{\texttt{if,condition,then,else,a:=1}\} \\
V &= \{\langle \text{STMT} \rangle, \langle \text{IF-THEN} \rangle, \langle \text{IF-THEN-ELSE} \rangle, \langle \text{ASSIGN} \rangle\}
\end{aligned}$$

(a) Show that G is ambiguous.

(b) Give a new unambiguous grammar for the same language. ◁

to do

2.32 Let $\Sigma = \{1, 2, 3, 4\}$ and $C = \{w \in \Sigma^* \mid \text{in } w, \text{ the number of 1s equals the number of 2s, and the number of 3s equals the number of 4s}\}$. Show that C is not context-free. ◁

Suppose C is with pumping length p. But we can not pump $1^p 3^p 2^p 4^p$.

Chapter 3

The Church-Turing Thesis

Defn 3.1 (3.3) A ***Turing Machine*** (TM) is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where

1. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
2. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \in \Gamma$,
3. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the transition function,
4. $q_{accept} \neq q_{reject}$.

On input w a TM M may halt (enter q_{accept} or q_{reject}) or run forever ("loop"). M ***accepts*** w by entering q_{accept} and ***rejects*** w by entering q_{reject} *or* looping. $\diamond$

Defn 3.2 (3.5) The collection of strings M accepts is the ***language*** M, or the ***language*** recognized by M (written $L(M)$). Call a language ***Turing-recognizable*** (or ***recursively enumerable***) if some TM recognizes it. $\diamond$

Defn 3.3 (3.6) TM M is a ***decider*** if M halts on all inputs. Say that M decides A if $A = L(M)$ and M is a decider. Call a language ***Turing-decidable*** or simply ***decidable*** (or ***recursive***) if some TM decides it. $\diamond$

Thm 3.4 (3.13) Every multitape Turing machine has an equivalent single-tape Turing machine. $\diamond$

Proof Idea. Suppose we have a multi-tape TM M and a single-tape TM S. S can simulate M by storing the contents of multiple tapes on a single tape in "blocks" and record head positions with dotted symbols. To simulate each of M's steps, S can (a) scan entire tape to find dotted symbols, (b) scan again to update according to M's δ and (c) shift to add room as needed. Finally, S accepts/rejects if M does. $\dashv$

Defn 3.5 A ***Nondeterministic*** TM (NTM) is similar to a TM except for its transition function $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\})$. ◇

Thm 3.6 (3.16) Every nondeterministic Turing machine has an equivalent deterministic Turing machine. ◇

Proof Idea. Suppose we have an NTM N and a TM M. M can simulate N by storing each thread's tape in a separate "block" on its tape. M also need to store the head location, and the state for each thread, in the block. If a thread forks, then M copies the block. If a thread accepts then M accepts. ⊣

Defn 3.7 An ***enumerator*** is a TM with a printer. It starts on a blank tape and can print strings $w_1, w_2, w_3, \ldots$, possibly forever. For enumerator E say $L(E) = \{w | E \text{ prints } w\}$. ◇

Thm 3.8 (3.21) A is T-recognizable iff $A = L(E)$ for some enumerator E. ◇

Proof Idea. ($\Leftarrow$) Simulate E with TM M. For input w, whenever E prints x, test if $x = w$. Accept or continue accordingly.
($\Rightarrow$) Simulate TM M with E on *each* possible input w_i. Let E print accordingly whenever M accepts. We can do this in a "time-sharing" scheme, for example let M go through 1 transitions on input w_1, then 2 transitions on input w_1 and w_2 respectively, then 3 transitions on input w_1, w_2 and w_3 respectively and so forth. When switching the input that M is currently working on, keep track of the state and the place of the head on the tape and the input together on a block of the tape. ⊣

Note 3.9 The definition of ***algorithms*** came in the 1936 papers of Alonzo Church and Alan Turing. Church used a notational system called the λ-calculus to define algorithms. Turing did it with his "machines". These two definitions were shown to be equivalent. This equivalence relation (in some sense) between the informal notion of algorithm and the precise definition (TM) has come to be called the ***Church–Turing thesis***. Note that it is about the equivalence between the intuitive and the formal, thus *not* provable. Instead it's a philosophical postulate. ◇

Note 3.10 David Hilbert's tenth problem was to devise an algorithm that tests whether a polynomial has an integral root (*Diophantine eqations*), or equivalently, to decide

$$D = \{\langle p \rangle | \text{ polynomial } p(x_1, x_2, \ldots, x_k) = 0 \text{ has an integer root}\}.$$

D is easily recognizable but was proved to be not decidable in 1970. ◇

Note 3.11 If O is some object, write $\langle O \rangle$ to be an encoding of that object into a string. (For example $\langle M \rangle$, where M is a TM. $\langle O_1, \ldots, O_k \rangle$ is similarly defined.) Furthermore we will use high-level English descriptions of algorithms when we describe TMs, knowing that we could (in principle) convert those descriptions into states, transition function, etc. ◇

Exercises and Problems

3.12 Question to fill in. ◁

to do

3.14 A ***queue automaton*** is like a push-down automaton except that the stack is replaced by a queue. A queue is a tape allowing symbols to be written only on the left-hand end and read only at the right-hand end. Each write operation (we'll call it a push) adds a symbol to the left-hand end of the queue and each read operation (we'll call it a pull) reads and removes a symbol at the right-hand end. As with a PDA, the input is placed on a separate read-only input tape, and the head on the input tape can move only from left to right. The input tape contains a cell with a blank symbol following the input, so that the end of the input can be detected. A queue automaton accepts its input by entering a special accept state at any time. Show that a language can be recognized by a deterministic queue automaton iff the language is Turing-recognizable. ◁

to do

3.18 Show that a language is decidable iff some enumerator enumerates the language in the standard string order. ◁

to do

Chapter 4

Decidability

4.1 Decidable Languages

Thm 4.1 (4.1) $A_{\text{DFA}} = \{\langle B, w\rangle | B \text{ is a DFA that accepts } w\}$ is decidable. ◇

Proof Idea. We use a TM to simulate B on w, this is clearly a decider. ⊣

Thm 4.2 (4.2) Also, A_{NFA} is decidable. ◇

Proof Idea. Convert the NFA to a DFA and use the TM from 4.1 to decide. Actually an NFA might loop for that it takes ε. It actually still will be decidable, but that is something we have to prove. ⊣

Thm 4.3 (4.3) $E_{\text{DFA}} = \{\langle A\rangle | A \text{ is a DFA and } L(A) = \emptyset\}$ is decidable. ◇

Proof Idea. We use BFS to see which of the states of A are reachable. ⊣

Thm 4.4 (4.5) $EQ_{\text{DFA}} = \{\langle A, B\rangle | A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$ is decidable. ◇

Proof Idea. Make DFA C that accepts w where A and B disagree (easily, $L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$) and test if $L(C) = \emptyset$. If we are to prove it by feeding strings into the simulated DFAs it would be way more complicated. A basic idea is to try to prove that if $L(A) \neq L(B)$, some strings whose length are at most some bound (say the sum or product of the numbers of states in A and B) are going to behave differently when fed to A and B. ⊣

Thm 4.5 (4.7) $A_{\text{CFG}} = \{\langle G, w\rangle | G \text{ is a CFG that generates } w\}$ is decidable. ◇

Proof Idea. By 2.4 and 2.26 this is trivial. ⊣

Comment. So A_{PDA} is also decidable. Note that this is not easy to prove on its own, since PDAs may not halt.

Thm 4.6 (4.8) $E_{\text{CFG}} = \{\langle G\rangle | G \text{ is a CFG and } L(G) = \emptyset\}$ is decidable. ◇

Proof Idea. Mark all the terminals in G. Then repeat the following until new variables are marked: mark all occurrences of variable A if $A \to B_1 B_2 \cdots B_k$ is a rule and all B_i's were already marked. ⊣

Thm 4.7 (4.9) Every context-free language is decidable. ◇

Proof Idea. It follows immediately from 4.5. Note that we know it is decidable, without knowing *how* to decide it. ⊣

4.2 Undecidability

Thm 4.8 $\mathbb{R}$ is uncountable. ◇

Proof. We prove by contradiction and via diagonization. Say that each real number is assigned an index $i \in \mathbb{N}$. Then $r \in \mathbb{R}$ in its decimal representation that differs from the nth number in the nth digit for any $n \in \mathbb{N}$ is not ith number for any i. Hence $r \notin \mathbb{R}$ and contradiction. ⊣

Cor 4.9 The set of all languages over Σ is not countable. ◇

Proof. Σ^* is countable. We can assign each language an infinite *binary* decimal, with each digit of it representing if a string is present in the language. ⊣

Observe that the set $\mathcal{M}$ of all turing machines is countable. So some languages are not recognizable, a fortiori decidable.

Note 4.10 David Hilbert's first problem asked if there is a set of intermediate size between $\mathbb{N}$ and $\mathbb{R}$. Gödel and Cohen showed that we cannot answer this question by using the standard axioms of mathematics. We are even not sure if "infinite sets" has mathematical "reality". ◇

Rmk 4.11 A_{TM} is recognizable. ◇

We can prove it by simulating. Note that we do not ask the simulator TM to reject if M rejects by looping (one should prove that this can be carried out effectively, but 5.1 shows that it cannot, for that we can not tell if a TM is looping on some input) and instead we say nothing about the looping condition, for that if M rejects by looping, our simulator will as well. This simulator is of great historical importance, as the *first* machine described (due to Alan Turing) to operate based on a stored program, and it later came to be known as the Von Neumann architecture.

Thm 4.12 (4.11) $A_{\text{TM}} = \{\langle M, w\rangle | M \text{ is a TM that accepts } w\}$ is undecidable. ◇

Proof. Assume some TM H decides A_{TM}. We use H to construct TM

$D =$ "On input $\langle M\rangle$
1. Simulate H on input $\langle M, \langle M\rangle\rangle$.
2. Accept if H rejects and reject if H accepts."

Then D accepts $\langle D\rangle$ iff D does not accept $\langle D\rangle$. Hence contradiction. ⊣

Comment. Surprisingly $\langle M, \langle M\rangle\rangle$ is of some real world applications, for example self-hosting languages whose compilers can be written in themselves. Check the following figure to see that this proof uses *diagonalization*. To employ it we essentially attempt to show contradiction by constructing some item (by countability) that differs from all items of its type by some *decidable* process.

TMs	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\cdots$	$\langle D\rangle$
M_1	acc	rej	acc	acc	$\cdots$	acc
M_2	rej	rej	rej	rej	$\cdots$	rej
M_3	acc	acc	acc	acc	$\cdots$	acc
M_4	rej	rej	acc	acc	$\cdots$	rej
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
D	rej	acc	rej	rej	$\cdots$	?

Figure 4.1: Undecidability hinges on diagonalization.

Thm 4.13 A language is decidable iff both it and its complement are Turing-recognizable. ◇

Proof. We construct a decider by simulating both TMs in parallel (alternately). ⊣

Cor 4.14 $\overline{A_{\text{TM}}}$ is unrecognizable. ◇

Comment. 4.13 can be stated more generally: Let D and L be two languages, and L a decidable one. Then $D \cap L$ is decidable iff both it and $\overline{D} \cap L$ is recognizable.

Exercises and Problems

4.14 Show that

$$\{\langle G\rangle | G \text{ is a CFG over } \{0,1\} \text{ and } 1^* \cap L(G) \neq \emptyset\}$$

is decidable. ◁

to do (hard!)

4.18 Let C be a language. Prove that C is T-recognizable iff a decidable language D exists such that $C = \{x | \exists y(\langle x, y\rangle \in D)\}$. ◁

Proof. ($\Leftarrow$) Consider a string x, we enumerate $y \in \Sigma^*$ and see if $\langle x, y_i\rangle \in D$ for at least one i. Thus C is recognizable. *($\Rightarrow$) Say that $L(M) = C$, where M is a TM. Let $D = \{\langle x, y\rangle | M \text{ accepts } x \text{ in } y \text{ steps}\}$, and we are done. ⊣

Comment. A bound that can approach infinity can be useful to turn something recognizable into something decidable.

4.24 A *useless state* in a pushdown automaton is never entered on any input string. Consider the problem of determining whether a pushdown automaton has any useless states. Formulate this problem as a language and show that it is decidable. ◁

to do

4.27 Show that

$$E = \{\langle M\rangle | M \text{ is a DFA that accepts some string with more 1s than 0s}\}$$

is decidable. ◁

to do

Chapter 5

Reducibility

Thm 5.1 (5.1) $H_{\text{TM}} = \{\langle M, w\rangle | M \text{ halts on input } w\}$ is undecidable. $\diamond$

Proof. Assume that R decides H_{TM}. Construct TM

$S =$ "On input $\langle M, w\rangle$

1. Simulate R on input $\langle M, w\rangle$. If M does not halt reject.
2. Simulate M on w.
3. Accept if M accepts and reject if M rejects."

Then S decides A_{TM}. Hence contradiction. $\dashv$

Comment. This proof uses the *reducibility* method. If we have two statements A and B, then A is reducible to B means that to show A, it *suffices* to show B. Thus here A_{TM} is reducible to H_{TM}.

Thm 5.2 (5.2) $E_{\text{TM}} = \{\langle M\rangle | M \text{ is a TM and } L(M) = \emptyset\}$ is undecidable. $\diamond$

Proof. Assume that R decides E_{TM} and let W decide $\{w\}$. Construct TM

$S =$ "On input $\langle M, w\rangle$

1. Construct TM M_w that accepts k iff both M and W accept k, where W is a decider for $\{w\}$.
2. Simulate R on input $\langle M_w\rangle$.
3. Accept if R rejects and reject if R accepts."

Then S decides A_{TM}. Hence contradiction. $\dashv$

Comment. We are reducing A_{TM} to E_{TM}. Note that M_w works like M except that it always rejecets $x \neq w$.

Defn 5.3 (5.17) 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. ◇

Defn 5.4 (5.20) A is ***mapping reducible*** to B, written $A \leq_m B$, if there is a computable function f where $w \in A \Leftrightarrow f(w) \in B$. f is then called a ***reduction*** from A to B. ◇

Thm 5.5 (5.22, 5.28) If $A \leq_m B$ and B is decidable (or recognizable), then A is decidable (or recognizable). ◇

Its proof is omitted as trivial. And the following is its contraposition.

Cor 5.6 (5.23, 5.29) If $A \leq_m B$ and A is undecidable (or unrecognizable), then B is undecidable (or unrecognizable). ◇

Mapping reducibility is useful to prove unrecognizability (often reducing $\overline{A_{\text{TM}}}$), while "reducibility to a decider" is useful to prove undecidability (often reducing A_{TM}).

Eg 5.7 In 5.2, we essentially constructed $f : \langle M, w\rangle \mapsto \langle M_w\rangle$, where $\langle M, w\rangle \in A_{\text{TM}} \Leftrightarrow \langle M_w\rangle \in \overline{E_{\text{TM}}}$. Thus $A_{\text{TM}} \leq_m \overline{E_{\text{TM}}}$ and $\overline{A_{\text{TM}}} \leq_m E_{\text{TM}}$. By 5.6 E_{TM} is unrecognizable. ◇

Thm 5.8 (5.30) Let

$$EQ_{\text{TM}} = \{\langle M_1, M_2\rangle | M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}.$$

Then both EQ_{TM} and $\overline{EQ_{\text{TM}}}$ is unrecognizable. ◇

Proof. Let $T_{M,w}$ be a TM that simulates M on w, T_{rej} a TM that always rejects and T_{acc} a TM that always accepts. Construct $f_1 : \langle M, w\rangle \mapsto \langle T_{M,w}, T_{rej}\rangle$. It follows that $\overline{A_{\text{TM}}} \leq_m EQ_{\text{TM}}$. Similarly, $f_2 : \langle M, w\rangle \mapsto \langle T_{M,w}, T_{acc}\rangle$ shows that $\overline{A_{\text{TM}}} \leq_m \overline{EQ_{\text{TM}}}$, and we are done. ⊣

Defn 5.9 A ***configuration*** of a TM is a snapshot of a TM at some time, written triple (q, p, t), with q, p and t representing the current state, the head position and the tape contents respectively. ◇

Encode configuration (q, p, t) as the string $t_1 q t_2$ where $t = t_1 t_2$ and the head position is on the first symbol of t_2.

Defn 5.10 An (accepting) ***computation history*** for TM M on input w is a sequence of configurations $C_1, C_2, \ldots C_{acc}$ that M enters until it accepts. ◇

Encode a computation history as the string $C_1\#C_2\#\cdots\#C_{acc}$, where each C_i is encoded as a string.

Defn 5.11 (5.6) A ***linearly bounded automaton*** (LBA) is a 1-tape TM that cannot move its head off the input portion of the tape. $\diamond$

Thm 5.12 (5.9) $A_{\text{LBA}} = \{\langle B, w\rangle |$ LBA B accepts $w\}$ is decidable. $\diamond$

Proof Idea. For inputs of length n, an LBA can have only $|Q| \times n \times |\Gamma|^n$ different configuration. Therefore, if an LBA runs for longer, it must repeat some configuration and will never halt. So for a decider for A_{LBA}, just simulate it for this many steps. $\dashv$

Thm 5.13 (5.10) $E_{\text{LBA}} = \{\langle M\rangle | M$ is an LBA where $L(M) = \emptyset\}$ is undecidable. $\diamond$

Proof. Assume TM R decides E_{LBA}. Construct TM

$S =$ "On input $\langle M, w\rangle$

1. Construct LBA $B_{M,w}$ that accepts its input x iff
   x is an accepting computation history for M on w.
2. Simulate R on input $\langle B_{M,w}\rangle$.
3. Accept if R rejects and reject if R accepts."

Then S decides A_{TM}. Hence contradiction. $\dashv$

Comment. It is not immediately staightforward that $B_{M,w}$ is *capable* of doing that. But note that it obviously takes *finite* memory to check if a string is an accepting computation history for M on w, for that after confirming that a configuration is valid, whatever is before that configuration *does not* matter. So we may even add however much need finite memory to the right of the tape as input (in the form of $\sqcup$'s) to make sure our $B_{M,w}$ is able to do that.

Thm 5.14 (5.15) An instance of the ***Post Correspondence Problem*** (PCP) consists of a finite collection of dominos

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\},$$

where each t_i, b_i are strings over some alphabet Σ. A ***match*** is a nonempty sequence of indices (repetitions permitted) $i_1, i_2, \ldots, i_\ell$ such that

$$t_{i_1} t_{i_2} \cdots t_{i_\ell} = b_{i_1} b_{i_2} \cdots b_{i_\ell}.$$

Then $PCP = \{\langle P\rangle | P$ is an instance of PCP with a match.$\}$ is undecidable. $\diamond$

Proof. Assume TM R decides PCP. Construct TM

$S =$ "On input $\langle M, w\rangle$

1. Construct PCP instance $P_{M,w}$ where a match corresponds to a computation history for M on w.
2. Simulate R on input $\langle P_{M,w}\rangle$.
3. Accept if R accepts and reject if R rejects."

Then S decides A_{TM}. Hence contradiction.

Now we construct $P_{M,w}$, where *the only* match is a computation history for M on w.

1. Assume for simplicity that the match starts with a starting domino:
$$\left[\frac{u_1}{v_1}\right] = \left[\frac{\#}{\#q_0 w_1 w_2 \cdots w_n \#}\right]$$
2. Transition dominos: For each transition $\delta(q, a) = (r, b, R)$ (handle left moves similarly), include:
$$\left[\frac{qa}{br}\right]$$
along with identity dominos and blank-handling dominos:
$$\left[\frac{a}{a}\right], \quad \left[\frac{\#}{\#}\right], \quad \left[\frac{\#}{\sqcup\#}\right]$$
3. Accepting dominos: To terminate a valid computation history:
$$\left[\frac{aq_{\text{accept}}}{q_{\text{accept}}}\right], \quad \left[\frac{q_{\text{accept}}a}{q_{\text{accept}}}\right], \quad \left[\frac{q_{\text{accept}}\#\#}{\#}\right]$$

⊣

Comment. Believe it or not, we are *forcing* the match to make a computation history. To resolve the assumption we can convert the strings a little bit to force starting using the first domino. See page 233 for this technical detail. Note that PCP is recognizable, for that the sequences of dominos is countable.

The proof showing the undecidablity of 3.10 reduced A_{TM} to D. Similar to 5.14, on input $\langle M, w\rangle$, we construct a polynomial with several variables, where to get an integer root of that polynomial, one of the variables has to be assigned some encoding of a computation history of the TM of M on w, and the other are to help to decode that encoding.

Thm 5.15 $ALL_{\text{CFG}} = \{\langle G\rangle | G \text{ is a CFG and } L(G) = \Sigma^*\}$ is undecidable. ◇

Proof. Assume TM R decides ALL_{CFG}. Construct TM

$S =$ "On input $\langle M, w\rangle$
1. Construct PDA $B_{M,w}$ that accepts its input x iff
   x is *not* an accepting computation history for M on w.
2. Simulate R on input $\langle B_{M,w}\rangle$.
3. Accept if R rejects and reject if R accepts."

Then S decides A_{TM}. Hence contradiction.

$B_{M,w}$ is constructed so that it *nondeterministically* pushes some C_i and pop to compare with C_{i+1} (with M in its mind) and accepts if some step is invalid, or the input starts wrongly or the ending configuration is not accepting. We reverse the even-indexed C_i to facilitate the comparing using stack. ⊣

Comment. Similarly ALL_{LBA} is undecidable. Note that PDAs are not capable of checking if strings are computation histories, which resonates with 4.6.

The computation history method is useful for showing the undecidability of problems involving testing for the existence of some object.

Exercises and Problems

5.1 Show that

$$EQ_{\text{CFG}} = \{\langle G, H\rangle | G, H \text{ are CFGs and } L(G) = L(H)\}$$

is undecidable. ◁

to do

5.21 Question to fill in. ◁

to do

5.22 Show that A is T-recognizable iff $A \leq_m A_{\text{TM}}$. ◁

($\Leftarrow$) This is trivial by 5.5.
($\Rightarrow$) Say that $L(R) = A$, then $f : w \mapsto \langle R, w\rangle$, where $w \in A$ shows that $A \leq_m A_{\text{TM}}$.

5.23 Show that A is decidable iff $A \leq_m 0^*1^*$. ◁

($\Leftarrow$) 0^*1^* is decidable, so this is trivial.
($\Rightarrow$) This is also trivial in a sense: Say that R decides A. We modify R so that it leaves 01 on the tape if accepts and otherwise 10.

5.25 Give an example of an undecidable language B, where $B \leq_m \overline{B}$. Is it possible that B is a recognizable language? ◁

to do

No. In that case both B and $\overline{B}$ would both be recognizable, thus both decidable.

5.26 Question to fill in. ◁

to do

Chapter 6

Advanced Topics in Computability Theory

Lem 6.1 (6.1) There is a computable function $q : w \mapsto \langle P_w \rangle$, where $w \in \Sigma^*$ and P_w is a TM that prints out w and then halts. $\diamond$

Rmk 6.2 We describe in spirit and not formally a TM *SELF*, that halts with $\langle SELF \rangle$ on the tape. *SELF* will be the combination of two TMs A and B. $A = P_{\langle B \rangle}$. And B on input $\langle M \rangle$ computes $q(\langle M \rangle)$, combines the result with $\langle M \rangle$ to make a complete TM, and finally prints the description of it and halt. $\diamond$

An English implementation would be:

```
Print out two copies of the following, the second one in quotes:
"Print out two copies of the following, the second one in quotes:"
```

The second line is A and the first B. A provides a description of B. B knows what itself (the instructions, not a string in quotes) is upon that description and computes A that will generate that description. Finally B prints them together. Consider that B would be fixed when deciding A, adding quotes to whatever it is.

Thm 6.3 Recursion Theorem Let TM T compute $t : \Sigma^* \times \Sigma^* \to \Sigma^*$. Then there is a TM R that computes $r : \Sigma^* \to \Sigma^*$, where for all $w \in \Sigma^*$, $r(w) = t(\langle R \rangle, w)$. $\diamond$

Proof Idea. Similar to 6.2, we desicribe without formality a TM R which is the combination of three TMs A, B and T, where T is given. $A = P_{\langle BT \rangle}$. B on input $\langle M \rangle$ computes $q(\langle M \rangle)$ and combines the result with $\langle B \rangle$ and $\langle T \rangle$ into a single TM. Finally T acts however it was intended to. $\dashv$

Take 6.3 in the following fashion: consider a TM T that computes $t(\langle M\rangle, w)$ (indicating, if $\langle M\rangle$ is the description of a TM, then compute within it and w and otherwise just print error) then there magically exists an R that computes $r(w) = t(\langle R\rangle, w)$. That means if we want to construct a TM that computes upon w and some TM M, M can be exactly the TM we are to construct!

Second Proof of 4.12. Assume that H decides A_{TM}. Construct TM

$B =$ "On input $\langle w\rangle$
1. Obtain via 6.3 $\langle B\rangle$.
2. Simulate H on $\langle B, w\rangle$.
3. Do the opposite of what Hsays."

Thus H fails to decide A_{TM}. Hence contradiction and we are done. ⊣

Thm 6.4 (6.7) $MIN_{\text{TM}} = \{\langle M\rangle | M$ is a minimal TM under some encoding$\}$ is not recognizable. ◇

Proof. Assume that E enumerates MIN_{TM}. Construct TM

$C =$ "On input $\langle w\rangle$
1. Obtain via 6.3 $\langle C\rangle$.
2. Run E until some TM D appears that is longer than C.
3. Simulate D on input w."

Hence contradiction and we are done. ⊣

Comment. MIN_{TM} is notable in that any infinite subset of it is not recognizable.

Thm 6.5 Let $t : \Sigma^* \to \Sigma^*$ be a computable function. Then there is a TM F such that $t(\langle F\rangle)$ describes a TM equivalent to F. ◇

Proof. Construct

$F =$ "On input $\langle w\rangle$
1. Obtain via 6.3 $\langle F\rangle$.
2. Compute $t(\langle F\rangle)$ to obtain the description of a TM G.
3. Simulate G on input w."

Thus G will be equivalent to F. ⊣

Comment. F is a fixed point w.r.t. t.

Thm 6.6 Gödel's First Incompleteness Theorem In any reasonable formal system, some true statements are not provable. ◇

Proof Idea. Suppose otherwise. If we can always prove (that is, to decidably check the truthfulness of something) $\langle M, w\rangle \in \overline{A_{TM}}$ when it is true, then $\overline{A_{\text{TM}}}$ is recognizable, and hence contradiction. ⊣

Comment. To prove is essentially the same as to compute. There are things we can not recognize, then sure enough facts exists that we can not verify.

Consider the statement "This statement is unprovable". It has to be true, thus unprovable.

Eg 6.7 Let ϕ be the statement $\langle R, 0\rangle \in \overline{A_{\text{TM}}}$ where R is the following TM:

$R =$ "On any input
1. Obtain via 6.3 $\langle R\rangle$ and use it to obtain ϕ.
2. Enumerate all proofs and accept if one is found for ϕ.

A proof or the falsity of ϕ each leads to a contradiction. Thus we have an example statement that is true but without proof. ◇

Defn 6.8 (6.18) An ***oracle*** for a language B is an external device that is capable of reporting whether any string w is a member of B. An oracle TM is a modified TM that has the additional capability of querying an oracle. We write M^B to describe an oracle TM that has an oracle for language B. ◇

Defn 6.9 (6.20) Say A is ***Turing reducible*** (or ***decidable relative***, written $A \leq_T B$) to B, if there exists some M^B that decides A. ◇

Eg 6.10 (6.19) $E_{\text{TM}} \leq_T A_{\text{TM}}$, for that we can construct TM

$T^{A_{\text{TM}}} =$ "On input $\langle M\rangle$
1. Construct TM N that simulates M on all strings in parallel and accepts if M accepts any string.
2. Query the oracle to determine whether $\langle N, 0\rangle \in A_{\text{TM}}$.
3. If the oracle answers no, accept; if yes, reject." ◇

Exercises and Problems

6.1 Give an example in the spirit of the recursion theorem of a program in a real programming language that prints itself out. ◁

```
1 #include <stdio.h>
2 char *program = "#include <stdio.h>%cchar *program = %c%s%c;%
    cint main()%c{%c        printf(program, 10, 34, program,
    34, 10, 10, 10, 10, 10);%c        return 0;%c}%c";
3 int main()
4 {
5         printf(program, 10, 34, program, 34, 10, 10, 10, 10,
             10, 10);
6         return 0;
7 }
```

Figure 6.1: A piece of C that prints itself.

10 is the ASCII code for `NEWLINE` and 34 for `"`. To relate to 6.2, consider that A is the string `*program`. B, the "printer", upon seeing a string, combines it with what ever will generate it. So the first `program` in line 5 is what B sees, and the second is what generates the string. The combination is done in a nested fashion.

6.11 Question to fill in. ◁

to do

Chapter 7

Time Complexity

Computibility theory asks *"Is A decidable?"*, and complexity theory asks *"Is A decidable with restricted resources?"*.

Defn 7.1 (7.1) Let M be a deterministic TM 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 (worst case) of steps that M uses on any input of length n. $\diamond$

Defn 7.2 (7.2) $f(n)$ is $O(g(n))$ if $f(n) \leq cg(n)$ for some fixed c independent of n. $\diamond$

Defn 7.3 (7.5) $f(n)$ is $o(g(n))$ if $f(n) \leq \varepsilon g(n)$ for all $\varepsilon > 0$ and large n. $\diamond$

Eg 7.4 A 1-tape TM M can decide $a^k b^k$ using $O(n^2)$ steps. $\diamond$

Proof Idea. Check if $w \in a^*b^*$ ($O(n)$). Repeatedly scan the tape, crossing off one a and one b ($O(n^2)$), and reject or accept accordingly. $\dashv$

Eg 7.5 A 1-tape TM M can decide $a^k b^k$ using $O(n \log n)$ steps. $\diamond$

Proof Idea. Repeatedly scan the tape, crossing off every other a and b, reject if even/odd parities disagree ($O(\log n)$ iterations $\times O(n)$ steps). $\dashv$

Comment. We state without proof that a 1-tape TM cannot decide $a^k b^k$ using $o(n \log n)$ steps. And any language decidable (by a 1-tape TM) in $o(n \log n)$ steps is regular. And all regular languages can be decided (by a 1-tape TM) in $O(n)$ steps (this one is easy to prove, for that DFAs decide in n steps.)

Defn 7.6 (7.7) Let $t : \mathbb{N} \to \mathbb{N}$ be a function. Define the ***time complexity class*** TIME($t(n)$) to be the collection of all languages decidable by an $O(t(n))$ 1-tape TM. $\diamond$

Thm 7.7 (7.8) If a multi-tape TM decides B in time $t(n)$, then $B \in \text{TIME}(t^2(n))$. ◇

Proof Idea. To simulate 1 step of the multi-tape TM M's computation, the 1-tape TM S uses $O(t(n))$ (the available lengths of tapes of M combined) steps. Cf. 3.4. ⊣

Defn* 7.8 Two models of computation are ***polynomially equivalent*** iff each can simulate the other with a polynomial overhead: $t(n)$ time on one model $\to t^k(n)$ time on the simulator model, for some k. ◇

All reasonable deterministic computational models are polynomially equivalent. For example 1-tape TMs, multi-tape TMs, multi-demensional TMs, random access machines (RAMs), cellular automata, etc. "Reasonable" is not to be precisely defined here. We may consider that one step should fail to accomplish "exponential amount of work".

Computability theory is desirable in a sense that it achieves model independence: it does not matter if you are using TMs or lambda calculus. Consider that obviously a multi-tape TM M can decide $a^k b^k$ using $O(n)$ steps (cf. 7.5) to realize that complexity theory *is* model dependent. It is model independent though, if we take polynomially equivalence in to consideration. Thus we will keep using 1-tape TMs for our analysis for the sake of simplicity.

Defn 7.9 (7.12) P is the class of languages that are decidable in polynomial time on a deterministic 1-tape TM. In other words,

$$\text{P} = \bigcup_k \text{TIME}(n^k). \qquad \diamond$$

Let $PATH = \{\langle G, s, t\rangle | G \text{ is a directed graph with a path from } s \text{ to } t\}$.

Thm 7.10 (7.14) $PATH \in \text{P}$. ◇

Proof Idea. We use BFS. Cf. 4.3. ⊣

That BFS is polynomial is nontrivial. (Or it is?)

Defn 7.11 (7.9) Let N be a nondeterministic TM that is a decisider. 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. ◇

Defn 7.12 (7.21) NTIME($t(n)$) is the collection of all languages decidable by an $O(t(n))$ nondeterministic TM. ◇

Defn 7.13 (7.22) NP is the class of languages that are decidable in polynomial time on a nondeterministic TM. In other words,

$$\text{NP} = \bigcup_k \text{NTIME}(n^k). \qquad \diamond$$

Like P, NP is independent of the choice of model. And it corresponds roughly to the set of easily verifiable problems. Cf. 7.17.

A ***Hamiltonian path*** in a directed graph G is a directed path that goes through each node exactly once. Let

$$\begin{aligned} \mathit{HAMPATH} = \{\langle G, s, t\rangle \mid & G \text{ is a directed graph} \\ & \text{with a Hamiltonian path from } s \text{ to } t\}. \end{aligned}$$

Thm 7.14 $\mathit{HAMPATH} \in \text{NP}$. $\diamond$

Proof Idea. We nondeterministically *guess* a path, i.e., a permutation of the vertices in G, and then verify that it is actually a path in G. $\dashv$

Comment. We may as well try all permutations of vertices on a deterministic model, but that will take more than polynomial (or exponential) time. We cannot tell if $\overline{\mathit{HAMPATH}} \in \text{NP}$. If P = NP, then we can easily (i.e., polynomially) decide both $\mathit{HAMPATH}$ and $\overline{\mathit{HAMPATH}}$, yet we are not sure of that. Also, inverting the ouput of the NTM used in the proof will not work, for that we will have to make sure *every* branch of it rejects, which is a *deterministic* process.

Let $\mathit{COMPOSITES}$ be the set of all nonprime integers.

Thm 7.15 $\mathit{COMPOSITES} \in \text{NP}$. $\diamond$

Proof Idea. We nondeterministically guess some factor for each number and then verify. $\dashv$

Comment. Bad encodings may change the game. For example writing number k in unary (as a string 1^k) will make it exponentially longer. We state without proof that $\mathit{COMPOSITES} \in \text{P}$ and $\mathit{PRIMES} \in \text{P}$ (these are equivalent.) And the algorithm does not work by factoring (factoring is not known to be solvable in polynomial time.) So it is noteworthy that some method other than searching for a certificate (see below) may turn out to be useful in polynomially deciding things. Cf. 7.19.

Defn 7.16 (7.18) A ***verifier*** for a language A is an algorithm V, where

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

A ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier. Here c is called a ***certificate***. ◇

For example, a certificate for a string in *HAMPATH* could be a Hamiltonian path, a certificate for a composite could be one of its factors. A certificate may not seem *trivial* though (e.g., the certificate for a prime). But as long as we are dealing with some member of NP, the verification should be easy (i.e., polynomial).

Thm 7.17 NP is the class of languages that are polynomially verifiable. ◇

Proof Idea. ($\Leftarrow$) The NTM can simulate the verifier by guessing the certificate.
($\Rightarrow$) Given the NTM and the certificate, the verifier checkes if the certificate indicates an accepting branch of the NTM. ⊣

Intuitively,

$$\begin{aligned} \text{P} &= \text{ the class of languages for which membership can be decided quickly.} \\ \text{NP} &= \text{ the class of languages for which membership can be } \textit{verified} \text{ quickly.} \end{aligned}$$

Also, the distinction between P and NP roughly relates to whether searching is needed in the decision.

Thm 7.18 $A_{\text{CFG}} \in \text{NP}$. ◇

Proof Idea. Cf. 4.5. On input $\langle G, w\rangle$, We first convert G into Chomsky normal form (should be polynomial). Then *nondeterministically* choose a derivation of length $2|w| - 1$. Accept if w is derived. ⊣

Thm 7.19 $A_{\text{CFG}} \in \text{P}$. ◇

Proof Attempt. We first devise a recursive algorithm C that does something slightly more general.

$C =$ "On input $\langle G, w, R\rangle$

1. For each way to divide $w = xy$ and each rule $R \to ST$,
   a. Use C to test $\langle G, x, S\rangle$ and $\langle G, y, T\rangle$,
   b. Accept if both accept,
2. Reject if no acceptance."

Then decide A_{CFG} by starting from G's start variable.

Comment. C is correct, but it takes non-polynomial time. Each recursion makes $O(n)$ calls and depth is roughly $\log n$. Observe that w of length n has $O(n^2)$ substrings,

therefore there are only $O(n^2)$ (actually possibly more, considering G as input, and some weirdness of 1-tape TM) possible sub-problems to solve. We may use recursion with memory, which is weirdly called ***dynamic programming*** (DP).

Proof Idea. Use C in the attempt, except that we add a step 0: If previously solved $\langle G, w, R\rangle$, answer the same (memoization). ⊣

Comment. C is top-down. We may also do this bottom-up. It is like filling out a table.

Let SAT denote all satisfiable boolean formulas.

Defn 7.20 (7.29) A is ***polynomial time reducible*** to B ($A \leq_p B$) iff $A \leq_m B$ by a reduction function that is computable in polynomial time. ◇

Thm 7.21 If $A \leq_p B$ and $B \in$ P then $A \in$ P. ◇

Thm 7.22 (7.27) $SAT \in$ P iff P = NP. ◇

This result is due to 7.26 and 7.27

We basically attempt to show that all members of NP is polynomial time reducible to SAT, which is a very similar result to 5.22.

Defn 7.23 A ***literal*** is a Boolean variable or its negation. A ***clause*** is the disjuction of several literals. A formula is in ***conjunctive normal form*** (CNF), iff it is the conjunction of several clauses. It is a ***3cnf-formula*** iff all its clauses have 3 literals. ◇

Let $3SAT = \{\langle\phi\rangle | \phi$ is a satisfiable 3cnf-formula$\}$.

Defn 7.24 A ***clique*** in an undirected graph is a fully connected subgraph. A ***k-clique*** is one containing k nodes. ◇

Let $CLIQUE = \{\langle G, k\rangle | G$ is an undirected graph with a k-clique$\}$. $CLIQUE \in$ NP, for that the clique is a certificate.

Thm 7.25 (7.32) $3SAT \leq_p CLIQUE$. ◇

Proof Idea. On input $\langle\phi\rangle$, where $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$, we construct a corresponding graph $G = (V, E)$. Whenever a literal l occurs in clause C_i, add a vertex $v_{i,l}$ to V. Connect two distinct vertices v_{i,l_1} and v_{j,l_2} iff $i \neq j$ and the literals they represent are consistent. This reduction is computable in polynomial time. And its correctness is due to the proof of the following claim: ϕ is satisfiable iff G has a k-clique. ⊣

Comment. The size of clauses (3 here) does not matter here.

Defn 7.26 (7.34) A language B is ***NP-complete*** iff (1) $B \in$ NP, and (2) $\forall A \in$ NP, $A \leq_p B$. $\diamond$

If B is NP-complete and $B \in$ P then P = NP.

Thm 7.27 COOK-LEVIN 1971 *SAT* is NP-complete. $\diamond$

Proof Sketch. We force the wff to simulate the machine, which is just like the construction in 5.14, so that a satisfying assignment to $\varphi_{M,w}$ is a computation history for M on w.

Take any language $A \in$ NP. Let N be a nondeterministic TM that decides A in n^k time for some constant k. A ***tableau*** for N on w is an $n^k \times n^k$ table whose rows are the configurations of a branch of the computation of N on input w. A tableau is ***accepting*** if any row of the tableau is an accepting configuration. Thus every accepting tableau for N on w corresponds to an accepting computation branch of N on w. Each of the n^{2k} entries of a tableau is called a ***cell***. The cell in row i and column j is called $cell[i,j]$ and contains a symbol from $C = \Gamma \cup \Sigma$. A 2×3 window in a tableau is ***legal*** if it does not violate the actions specified by N's transition function.

For each i and j between 1 and n^k and for each $s \in C$, we have a variable, $x_{i,j,s}$. If $x_{i,j,s}$ takes on value 1, it means that $cell[i,j]$ contains an s. Now we describe $\varphi_{N,w}$, which is satisfiable iff an accepting tableau exists that corresponds to an accepting computation branch of N on w, that is, we describe the reduction $f : w \mapsto \varphi_{N,w}$ such that $w \in L(N)$ iff $\langle \varphi_{N,w} \rangle \in SAT$.

$$\varphi_{N,w} = \varphi_{cell} \wedge \varphi_{start} \wedge \varphi_{accept} \wedge \varphi_{move}, \text{ where}$$

$$\varphi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right],$$

$$\varphi_{\text{start}} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \cdots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \cdots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#},$$

$$\varphi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}}, \text{ and}$$

$$\varphi_{\text{move}} = \bigwedge_{1 \leq i < n^k, 1 < j < n^k} \left[\bigvee_{legal(a_1,\ldots,a_6)} \begin{pmatrix} x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge \\ x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \end{pmatrix} \right].$$

φ_{cell} says that each cell contains exactly one symbol in C. φ_{start} says that the tableau starts with a valid starting configuration. φ_{accept} says that q_{accept} appears in one of

the cells, indicating a accepting tableau. And φ_{move} says that the computation is carried out legally. We assert, without proof, that this is a correct construction. $\dashv$

Importance of NP-completeness: (1) Showing B is NP-complete is evidence of computational intractability. (It is almost certainly not in P.) (2) Giving a good candidate for proving P $\neq$ NP. (You cannot pick the wrong problem.)

Thm 7.28 Ladner 1975 If P $\neq$ NP, then there exist languages in NP that are neither in P nor NP-complete. $\diamond$

Thm 7.29 (7.42) *3SAT* is NP-complete. $\diamond$

Proof Sketch. We show that $SAT \leq_p 3SAT$, by giving a polynomial reduction $f : \varphi \mapsto \varphi'$, where $\varphi \in SAT$ and $\varphi' \in 3SAT$, preserving satisfiability (an equivalent conversion would likely take exponential time.)

First conduct the trivial and equivalent convertion that makes sure negation only occurs on leaves of the construction tree of a formula. Then assign a new variable to each binary conjunction (or disjunction) that is satisfiable iff the conjunction (or disjunction) is satisfiable. For conjunction we have $a \wedge b$ is satisfiable iff c is satisfiable, where c is s.t.

$$((a \wedge b) \to c) \wedge ((a \wedge \neg b) \to \neg c) \wedge ((\neg a \wedge b) \to \neg c) \wedge ((\neg a \wedge \neg b) \to \neg c).$$

For disjunction, similarly, $a \vee b$ is satisfiable iff c is satisfiable, where c is s.t.

$$((a \wedge b) \to c) \wedge ((a \wedge \neg b) \to c) \wedge ((\neg a \wedge b) \to c) \wedge ((\neg a \wedge \neg b) \to \neg c).$$

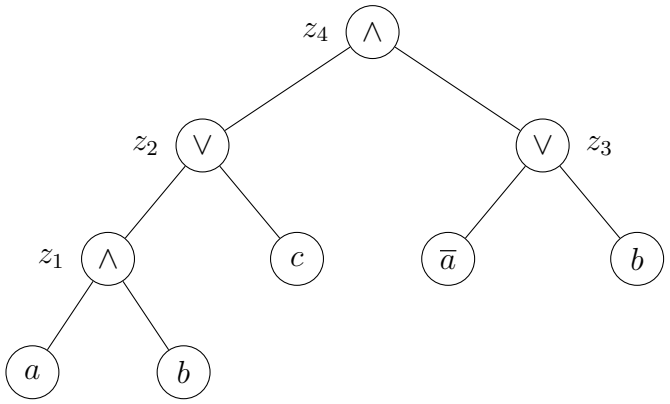

Figure 7.1: $\varphi = ((a \wedge b) \vee c) \wedge (\overline{a} \vee b)$

Thus, for example, $\varphi = ((a \wedge b) \vee c) \wedge (\overline{a} \vee b)$ is satisfiable iff

$$\begin{aligned}\varphi' = &((a \wedge b) \to z_1) \wedge ((\neg a \wedge b) \to \neg z_1) \wedge ((a \wedge \neg b) \to \neg z_1) \wedge ((\neg a \wedge \neg b) \to \neg z_1) \wedge \\ &((z_1 \wedge c) \to z_2) \wedge ((\neg z_1 \wedge c) \to z_2) \wedge ((z_1 \wedge \neg c) \to z_2) \wedge ((\neg z_1 \wedge \neg c) \to \neg z_2) \wedge \\ &((\neg a \wedge b) \to z_3) \wedge ((a \wedge b) \to z_3) \wedge ((\neg a \wedge \neg b) \to z_3) \wedge ((a \wedge \neg b) \to \neg z_3) \wedge \\ &((z_2 \wedge z_3) \to z_4) \wedge ((\neg z_2 \wedge z_3) \to \neg z_4) \wedge \\ &((z_2 \wedge \neg z_3) \to \neg z_4) \wedge ((\neg z_2 \wedge \neg z_3) \to \neg z_4) \wedge \\ &(z_4 \vee z_4 \vee z_4)\end{aligned}$$

is satisfiable. Also observe that $((a \wedge b) \to c)$ is logically equivalent to $(\neg a \vee \neg b \vee c)$. Thus we are done. The correctness of this construction, as usual, is asserted without a proof. It should make a decent mathematical logic exercise. ⊣

Thm 7.30 (7.46) *HAMPATH* is NP-complete. ◇

Proof Idea. We simulate variables and clauses with "gadgets" made of subgraphs to reduce *3SAT* to *HAMPATH*. See page 314 for how this is done. Basically, the direction (zig-zag or zag-zig) in which the path goes through variables gadgets is the truth assignment.And clause gadgets are add so that if any of them is passed through, one of the corresponding literals should be assigned True. ⊣

Defn 7.31 A language B is ***NP-hard*** iff $\forall A \in \text{NP}, A \leq_p B$. ◇

Exercises and Problems

7.13 Question to fill in. ◁

to do

7.15 Show that P is closed under the star operation. ◁

to do

7.18 Show that if P = NP, then every language $A \in \text{P}$, except $A = \emptyset$ and $A = \Sigma^*$, is NP-complete. ◁

Consider $B \in \text{P}$ that is neither $\emptyset$ nor Σ^*. So there exists $a \notin B$ and $b \in B$. Consider also an arbitrary language $A \in \text{P}$. Say that M decides A in polynomial time. Our f works as follows: on input $\langle w \rangle$, simulate M on w, if accepts output b, and if rejects output a.

7.26 Question to fill in. ◁

to do

7.27 Question to fill in. ◁

to do

7.28 Question to fill in. ◁

to do

7.34 Show that D in 3.10 is NP-hard. ◁

to do

7.36 Question to fill in. ◁

to do

7.37 Question to fill in. ◁

to do

7.38 Show that if P = NP, a polynomial time algorithm exists that produces a satisfying assignment when given a satisfiable Boolean formula. ◁

to do

7.39 Question to fill in. ◁

to do

7.51 Question to fill in. ◁

to do

Chapter 8

Space Complexity

Defn 8.1 (8.1) Let $f : \mathbb{N} \to \mathbb{N}$ where $f(n) \geq n$. Say a 1-tape TM M ***runs in space*** (or the ***space complexity*** of M is) $f(n)$ iff M always halts and uses at most $f(n)$ tape cells on all inputs of length n. Say an NTM N runs in space $f(n)$ iff all branches halt and each branch uses at most $f(n)$ tape cells on all inputs of length n. $\diamond$

Defn 8.2 (8.2) Let $f : \mathbb{N} \to \mathbb{R}^+$ be a function.

$$\text{SPACE}(f(n)) = \{L | L \text{ is decided by an } O(f(n)) \text{ space TM.}\}$$
$$\text{NSPACE}(f(n)) = \{L | L \text{ is decided by an } O(f(n)) \text{ space NTM.}\}$$ $\diamond$

Defn 8.3 (8.6) $\text{PSPACE} = \bigcup_k \text{SPACE}(n^k)$. $\diamond$