NOTES ON

Theory of Computation

Hao Su

Based on MIT OCW 18.404J and indexed according to *Introduction to the Theory of Computation* (3rd ed.), with some additional material.

July 18, 2026

Chapter 1

Regular Languages

Definition 1.1 (1.15). A *finite automaton* (DFA) 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*.

Definition 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.

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

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

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

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

Theorem 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. ⊣

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

Definition 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 $Q \times \Sigma_\varepsilon \to 2^Q$.

Remark 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.

Theorem 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 $Q' = 2^Q, q_0' = \{q_0\}, F' = \{R \in Q' | R \cap F \neq \emptyset\}$ 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. ⊣

Theorem 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. ⊣

Lemma 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.) ⊣

Lemma 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.

Definition 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. ⊣

Theorem 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. ⊣

Example 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

Exercise 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 $A^{\mathcal{R}}$.

Exercise 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

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

Show that B is regular.

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

Exercise 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.

Exercise 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.

Exercise 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 $X = \{s_1, \dots, s_k\}$ be pairwise distinguishable by L. We construct DFA $M = (Q, \Sigma, \delta, q_0, F)$ with k states recognizing L. Let $Q = \{q_1, \dots, q_k\}$, $F = \{q_i \mid s_i \in L\}$, 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. ⊣

Exercise 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.

Exercise 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

Exercise 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.

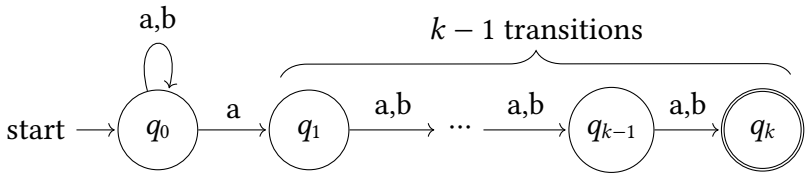

Figure 1.1: State diagram. A formal description is omitted.

Exercise 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 $b^k, b^{k-1}a, \ldots, a^k$ respectively. By 1.52 we are done. ⊣

Chapter 2

Context-Free Languages

Definition 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. $u \overset{*}{\Rightarrow} v$ (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.

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

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

Definition 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.

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*.

Definition 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.

Theorem 2.4 (2.9). *Any context-free language is generated by a context-free grammar in Chomsky normal form.*

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

Definition 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.

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.

Lemma 2.6 (2.21). *If A is a CFL then some PDA recognizes it.*

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. ⊣

Lemma 2.7 (2.27). *If a PDA recognizes A then it is a CFL.*

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 $p \in Q, (A_{p,p} \to \varepsilon) \in R$. ⊣

Theorem 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. *$uv^i x y^i z \in A$ 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". ⊣

Example 2.9. Usages of 2.8:

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

Obviously $0^k 1^k 2^l$ and $0^l 1^k 2^k$ 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

Exercise 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 $abbaa \notin A^*$. This exercise tell us to be careful with the original grammar.

Exercise 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 $S \to SS_1|\varepsilon$.

Exercise 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.

Exercise 18.

(a) Let C be a context-free language and R be a regular language. Prove that $C \cap R$ is context free.

(b) Let A be the language over $\{a, b, c\}$ whose strings contain equal numbers of a's, b's, and c's. Use part (a) to show that A is not context free.

(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 $a^k b^k c^k = A \cap a^* b^* c^*$, which is not by 2.8.

Exercise 22. Let $C = \{x\#y|x, y \in \{0, 1\}^* \text{ and } x \neq y\}$. Show that C is context-free.

to do

Exercise 24. Show that $E = \{a^i b^j | i \neq j \wedge 2i \neq j\}$ is a context-free language.

to do

Exercise 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.

Exercise 27. $G = (V, \Sigma, R, \langle \text{STMT} \rangle)$ is described as follows.

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

(a) Show that G is ambiguous.
(b) Give a new unambiguous grammar for the same language.

to do

Exercise 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

Definition 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.

Definition 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.

Definition 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.

Theorem 3.4 (3.13). *Every multitape Turing machine has an equivalent single-tape Turing machine.*

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. ⊣

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

Theorem 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. ⊣

Definition 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 $L(E) = \{w | E \text{ prints } w\}$.

Theorem 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. ⊣

Remark 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.

Remark 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

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

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

Remark 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. $\langle O_1, \dots, O_k \rangle$ 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

Exercise 12. A *Turing machine with left reset* is similar to an ordinary Turing machine, but the transition function has the form

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{R, \text{RESET}\}.$$

If $\delta(q, a) = (r, b, \text{RESET})$, when the machine is in state q reading an a, the machine's head jumps to the left-hand end of the tape after it writes b on the tape and enters state r. Note that these machines do not have the usual ability to move the head one symbol left. Show that Turing machines with left reset recognize the class of Turing-recognizable languages.

to do

Exercise 14. A *queue automaton* is like a push-down automaton except that the stack is replaced by a queue. 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

Exercise 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

Theorem 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. ⊣

Theorem 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. ⊣

Theorem 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. ⊣

Theorem 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. ⊣

Theorem 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.

Theorem 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. ⊣

Theorem 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

Theorem 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. ⊣

Corollary 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.

Remark 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".

Remark 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.

Theorem 4.12 (4.11). *$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM that accepts } w\}$ 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.

Theorem 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). ⊣

Corollary 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

Exercise 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!)

Exercise 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.

Exercise 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

Exercise 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

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

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. ⊣

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}.

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

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. ⊣

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

Definition 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.

Definition 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.

Theorem 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.

Corollary 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}).

Example 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.

Theorem 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. ⊣

Definition 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.

Definition 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.

Definition 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.

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

Proof Idea. For inputs of length n, an LBA can have only $|Q| \times n \times |\Gamma|^n$ different configurations. 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. ⊣

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

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. ⊣

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 ⊔'s) to make sure our $B_{M,w}$ is able to do that.

Theorem 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.

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:

$$\left[\frac{a q_{\text{accept}}}{q_{\text{accept}}}\right], \quad \left[\frac{q_{\text{accept}} a}{q_{\text{accept}}}\right], \quad \left[\frac{q_{\text{accept}} \# \#}{\#}\right]$$

⊣

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.

Theorem 5.15. *$ALL_{\text{CFG}} = \{\langle G\rangle | G$ 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

Exercise 1. Show that

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

is undecidable.

to do

Exercise 21. Question to fill in.

to do

Exercise 22. Show that A is T-recognizable iff $A \leq_{\text{m}} A_{\text{TM}}$.

($\Leftarrow$) 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 $A \leq_{\text{m}} A_{\text{TM}}$.

Exercise 23. Show that A is decidable iff $A \leq_{\text{m}} 0^*1^*$.

($\Leftarrow$) 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.

Exercise 25. Give an example of an undecidable language B, where $B \leq_{\text{m}} \overline{B}$. 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.

Exercise 26. Define a *two-headed finite automaton* (2DFA) to be a deterministic finite automaton that has two read-only, bidirectional heads that start at the left-hand end of the input tape and can be independently controlled to move in either direction. The tape of a 2DFA is finite and is just large enough to contain the input plus two additional blank tape cells, one on the left-hand end and one on the right-hand end, that serve as delimiters. A 2DFA accepts its input by entering a special accept state. For example, a 2DFA can recognize the language $\{a^n b^n c^n \mid n \geq 0\}$.

(a) Let

$$A_{\text{2DFA}} = \{\langle M, x\rangle \mid M \text{ is a 2DFA and } M \text{ accepts } x\}.$$

Show that A_{2DFA} is decidable.

(b) Let

$$E_{\text{2DFA}} = \{\langle M\rangle \mid M \text{ is a 2DFA and } L(M) = \emptyset\}.$$

Show that E_{2DFA} is not decidable.

to do

Chapter 6

Advanced Topics in Computability Theory

6.1 The Recursion Theorem

Lemma 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.*

Remark 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.

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.

Theorem 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 $w \in \Sigma^*$, $r(w) = t(\langle R \rangle, w)$.*

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. ⊣

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 on it and w, 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 with input being some TM M and w, 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 H says."

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

Theorem 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.

Theorem 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.

6.2 Decidability of Logical Theories

Theorem 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.

Example 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.

6.3 Turing Reducibility

Definition 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.

Definition 6.9 (6.20)**.** Say A is *Turing reducible* (or *decidable relative*, written $A \leq_{\text{T}} B$) to B, if there exists some M^B that decides A.

Example 6.10 (6.19). $E_{\text{TM}} \leq_{\text{T}} A_{\text{TM}}$, 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."

6.4 A Definition of Information

We define the quantity of information contained in an object to be the size of that object's smallest description. We should be able to recreate the object given this description. Note that this size is relative to *how* we describe things. That is, 00000 does not naturally contain less infomation than 10110. In this specific binary representation, of course it does.

Definition 6.11. Let x be a binary string. The *minimal description* of x, written $d(x)$, is the shortest string $\langle M, w \rangle$ where TM M on input w halts with x on its tape. If several such strings exist select the lexicographically first. The *Kolmogorov complexity* of x, written $\mathrm{K}(x)$, is $|d(x)|$.

Note that our definition of description is by *algorithms*. We would choose in practice to format $\langle M, w \rangle$ in the form of $\langle M \rangle w$, with a delimiter before w.

It is obvious that $\exists c \, \forall x \; \mathrm{K}(x) \leq |x| + c$: we can use a TM that halts as soon as it is started. This illustrates why we encoded 'description' using $\langle M, w \rangle$, instead of $\langle M \rangle$, for in that case we cannot 'decouple' strings into 'prime strings' and 'operations', and significantly larger descriptions are required to store the transition functions.

Theorem 6.12. $\exists c \, \forall x \; \mathrm{K}(xx) \leq \mathrm{K}(x) + c$.

Proof. Say that $d(x) = \langle N, w \rangle$. We have that $d(xx) \leq \langle M \rangle d(x)$, where

M = "On input $\langle N, w \rangle$:

1. Run N on w to obtain an output string s.
2. Print ss."

We are done, for that $|M|$ is a constant. ⊣

Theorem 6.13. $\exists c\ \forall x, y\ \mathrm{K}(xy) \leq 2\mathrm{K}(x) + \mathrm{K}(y) + c$.

Proof. We do this exactly the way we did in 6.12, except we have to encode $d(x)$ and $d(y)$ in a single string. One approach is to write every bit of $d(x)$ twice, 0 as 00 and 1 as 11, and then use 01 as a delimiter. This approach only has complete our proof. ⊣

Comment. It is all about how do we separate $d(x)$ and $d(y)$. One may consider escape characters in programming languages, yet what if $d(x)$ is made up of the characters that need to be escaped? So, in the worst case, we will need $2\mathrm{K}(x)$. Another way is to prepend the length of $d(x)$ as a binary integer that has been doubled to differentiate it from $d(x)$, lowering the bound to $2\log_2(\mathrm{K}(x))+\mathrm{K}(x)+\mathrm{K}(y)+c$. We may take another log to obtain $2\log_2\log_2(\mathrm{K}(x))+2^{\lceil\log_2 \mathrm{K}(x)\rceil}+\mathrm{K}(y)$ (which is bad). C.f. 6.26.

Definition 6.14. We consider a general *description language* to be any computable function $p : \Sigma^* \to \Sigma^*$ and define the minimal description of x with respect to p, written $d_p(x)$, to be the lexicographically first description of x. Define $\mathrm{K}_p(x) = |d_p(x)|$.

Theorem 6.15. *For any description language p, a fixed constant c exists that depends only on p, where $\forall x\ \mathrm{K}(x) \leq \mathrm{K}_p(x) + c$.*

Proof. This is immediate. Consider a TM M that computes p, then we have

$$\forall x\ \mathrm{K}(x) \leq |\langle M\rangle d_p(x)| = \mathrm{K}_p(x) + |\langle M\rangle|.$$ ⊣

Definition 6.16. Say string x is *c-compressible* if $\mathrm{K}(x) \leq |x| - c$. If x is not c-compressible, say that x is *incompressible by c* (and incompressible if $c = 1$).

Theorem 6.17. *Incompressible strings of every length exist.*

Proof. Say that each string of length n can be compressed, that is, there are at most $\sum_{i\leq n-1} 2^i = 2^n - 1$ descriptions for 2^n strings, hence contradiction. ⊣

Corollary 6.18. *At least $2^n - 2^{n-c+1} + 1$ strings of length n are incompressible by c.*

Definition 6.19. A *property* of strings is a function that maps strings to {TRUE , FALSE}. Say that a property *holds for almost all strings* if the fraction of stings of length n on which it is FALSE approaches 0 as n grows large.

Theorem 6.20. *Let f be a computable property that holds for almost all strings. Then, for any $b > 0$, the property f is FALSE on only finitely many strings that are incompressible by b.*

Proof Idea. This theorem basically says that long enough strings are compressible if they do not have some *computable* property that holds for almost all strings. Say that a TM M_f computes this property. How can we compress such a string with this M_f (the only TM available to us)?

Obviously and only, by counting. We store the index k of this string among those tested FALSE of f and use $\langle M'_f \rangle k$ as a compression, where M'_f on input k outputs the k-th string f is FALSE on. Consider the case where f holds for half of strings, where k should save us one bit of description. Now we have that f holds for arbitrarily percentage strings, saving us arbitrarily many bits as long as strings are long enough, effortlessly covering $|\langle M'_f \rangle| + b$.

What's intuitive about this theorem is that strings with 'computable scarcity' are very special, and thus telling a lot about themselves. ⊣

This relation between property that holds for almost all strings and incompressibility readily leads us to discussions about probability and randomness (because several probabilisitc fact holds for 'almost all long enough sampling sequences'), magically corresponding to Shannon's definition of information.

Incompressible strings look like strings that are obtained from random coin tosses. For example, we can show that any incompressible string of length n has roughly an equal number of 0s and 1s, and that the length of its longest run of 0s is approximately $\log_2 n$.

We will fall short of an example of incompressible strings. Cf. 6.23, 6.24, and 6.25.

Theorem 6.21. *For some constant b, for every string x, $d(x)$ is incompressible by b.*

Proof. Consider $d(x) = \langle M \rangle w$. Say that TM M' prints $d(x)$ on input w'. Also consider TM R, which on input $\langle M' \rangle w'$ computes $\langle M \rangle w$ and then x. That is, $\langle R, \langle M', w' \rangle \rangle$ is another description of x. We have that

$$|\langle R \rangle| + |\langle M' \rangle w'| \geq \mathrm{K}(x) = |d(x)|,$$

which give that $d(x)$ is incompressible by $|\langle R \rangle| + 1$, where '1' ensures that the inequality holds. ⊣

Exercises and Problems

Exercise 1. Give an example in the spirit of the recursion theorem of a program in a real programming language that prints itself out.

```c
#include <stdio.h>
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, 10);%c        return 0;%c}%c";
int main()
{
        printf(program, 10, 34, program, 34, 10, 10, 10, 10, 10, 10);
        return 0;
}
```

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.

Exercise 23. * Show that the function $\mathrm{K}(x)$ is not a computable function.

Idea: If it is we can decide incompressible strings.

Assume to the contrary that it is. Then we can construct TM M which on input n computes $\mathrm{K}(x)$ of all strings x of length n and outputs the first incompressible string it finds (that is, $\mathrm{K}(x) \geq n$). For large n, that gives a short description of the incompressible string, and contradiction.

Exercise 24. Show that the set of incompressible strings is undecidable.

C.f. 6.23.

Exercise 25. Show that the set of incompressible strings contains no infinite subset that is Turing-recognizable.

Again c.f. 6.23. Suppose we have an enumerator for some infinite subset of the set of incompressible strings, then construct TM M that on input n output the first incompressible string of length larger than n. For large n that gives a short description and contradiction.

Comment. I tried using index as input (c.f. 6.20) but unsuccessfully for that is not necessarily a shorter description. The length of strings is *tighter*.

Exercise 26. * Show that for any c, some strings x and y exist, where

$$\mathrm{K}(xy) > \mathrm{K}(x) + \mathrm{K}(y) + c.$$

A trick that 'partially compresses' a string w of length n is described as follows: Let z be the first m bits of w. Let j be the value of $1z$ as a number in binary. Let x be the first k bits of w where $k = m + j$. Now, given trailing j bits of x, we are also given its length, namely j itself, by which we obtain z and after that x. Of course $|x| < |w|$, which requires that

$$2^{m+1} - 1 + m < n.$$

Note that x is in this manner compressed by $\Theta(\log |x|)$, thus asymptotically shorter than $|x| + c$ for any c. Now let the rest of w be y and pick w an incompressible string of length n. We have that $\mathrm{K}(xy) \geq n$ yet

$$\mathrm{K}(x) + \mathrm{K}(y) \leq n - \Theta(\log n) + c_1 + c_2 < n.$$

We may choose $m = \lfloor \frac{\log n}{2} \rfloor$.

Chapter 7

Time Complexity

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

Definition 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.

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

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

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

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. ⊣

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

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). ⊣

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.)

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

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

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. ⊣

Definition 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.

Definition 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,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

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

Theorem 7.10 (7.14). *$PATH \in \mathrm{P}$.*

Proof Idea. We use BFS. Cf. 4.3. Specifically, we mark nodes until all nodes are marked. This takes at most $O(n^4)$ stepes. ⊣

Definition 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.

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

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

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

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

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

Theorem 7.14. *HAMPATH* $\in$ NP.

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. ⊣

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 $\overline{\mathit{HAMPATH}} \in$ NP. If P = NP, then we can easily (i.e., polynomially) decide both *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 *COMPOSITES* be the set of all nonprime integers.

Theorem 7.15. *COMPOSITES* $\in$ NP.

Proof Idea. We nondeterministically guess some factor for each number and then verify. ⊣

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 *COMPOSITES* $\in$ P and *PRIMES* $\in$ 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.

Definition 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).

Theorem 7.17. *NP is the class of languages that are polynomially verifiable.*

Proof Idea. ($\Rightarrow$) The verifier can simulate the NTM by guessing the certificate and verifying it. ($\Leftarrow$) Given the NTM and the certificate, we construct a verifier that checkes if the certificate indicates an accepting branch of the NTM. $\dashv$

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.

Theorem 7.18. $A_{\text{CFG}} \in$ 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. $\dashv$

Theorem 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.

Definition 7.20 (7.29). A is *polynomial time reducible* to B ($A \leq_{\text{P}} B$) iff $A \leq_{\text{m}} B$ by a reduction function that is computable in polynomial time.

Theorem 7.21. *If* $A \leq_{\text{P}} B$ *and* $B \in \text{P}$ *then* $A \in \text{P}$.

Theorem 7.22 (7.27). $SAT \in \text{P}$ *iff* $\text{P} = \text{NP}$.

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.

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

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

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

Theorem 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.

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

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

Theorem 7.27. ***COOK-LEVIN 1971*** *SAT is NP-complete.*

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 $\langle \varphi_{N,w} \rangle \in SAT$.

$$
\begin{aligned}
\varphi_{N,w} &= \varphi_{cell} \wedge \varphi_{start} \wedge \varphi_{accept} \wedge \varphi_{move}, \text{ where} \\
\varphi_{\text{cell}} &= \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \ne t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right], \\
\varphi_{\text{start}} &= x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \cdots \wedge x_{1,n+2,w_n} \wedge \\
&\quad x_{1,n+3,\sqcup} \wedge \cdots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}, \\
\varphi_{\text{accept}} &= \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{\text{accept}}}, \text{ and} \\
\varphi_{\text{move}} &= \bigwedge_{1 \le i < n^k, 1 < j < n^k} \left[\bigvee_{legal(a_1,\ldots,a_6)} \begin{pmatrix} x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge \\ x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \end{pmatrix} \right].
\end{aligned}
$$

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. ⊣

An alternative proof is given at 9.19.

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 ≠ NP. (You cannot pick the wrong problem.)

Theorem 7.28. ***Ladner 1975*** *If* P ≠ NP*, then there exist languages in NP that are neither in P nor NP-complete.*

Theorem 7.29 (7.42)**.** *3SAT is NP-complete.*

Proof Sketch. We show that $SAT \le_{\text{P}} 3SAT$, 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.

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

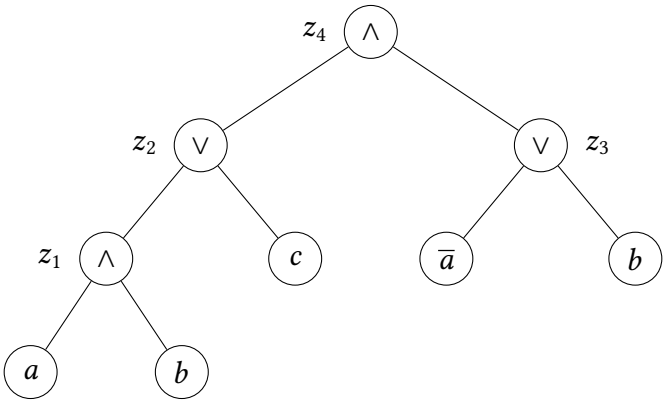

Figure 7.1: $\varphi = ((a \wedge b) \vee c) \wedge (\overline{a} \vee b)$

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. $\dashv$

Theorem 7.30 (7.46). *HAMPATH is NP-complete.*

Proof Idea. We simulate variables and clauses with "gadgets" made of subgraphs to reduce 3*SAT* 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. ⊣

Definition 7.31. A language B is *NP-hard* iff $\forall A \in \mathrm{NP}, A \leq_{\mathrm{P}} B$.

Exercises and Problems

Exercise 13. Let

$$MODEXP = \left\{ \langle a, b, c, p \rangle \;\middle|\; a, b, c, p \in \mathbb{Z}_+ \wedge a^b \equiv c \pmod{p} \right\}.$$

Show that $MODEXP \in \mathrm{P}$.

Consider cases where $b = 2^j$. Compute in order $a \bmod p$, $a^2 \bmod p, \ldots, a^{2^j} \bmod p$, where each term is the square of its predecessor. Note that there are $O(n)$ steps, each taking polynomial time, thus the algorithm takes polynomial time. Other cases need trivial generalization.

This is a standard skill that handles exponentiation.

Exercise 15. Show that P is closed under the star operation.

to do

Exercise 18. Show that if P = NP, then every language $A \in \mathrm{P}$, except $A = \emptyset$ and $A = \Sigma^*$, is NP-complete.

Consider $B \in \mathrm{P}$ that is neither $\emptyset$ nor Σ^*. So there exists $a \notin B$ and $b \in B$. Consider also an arbitrary language $A \in \mathrm{P}$. 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.

Exercise 26. Question to fill in.

to do

Exercise 27. Question to fill in.

to do

Exercise 28. Question to fill in.

to do

Exercise 34. Show that D in 3.10 is NP-hard.

to do

Exercise 36. Question to fill in.

to do

Exercise 37. Question to fill in.

to do

Exercise 38. Show that if P = NP, a polynomial time algorithm exists that produces a satisfying assignment when given a satisfiable Boolean formula.

to do

Exercise 39. Question to fill in.

to do

Exercise 46. * A Boolean formula is *minimal* if no shorter Boolean formula is equivalent to it. Let *MIN-FORMULA* be the collection of minimal Boolean formulas. Show that if P = NP, then *MIN-FORMULA* $\in$ P.

First notice that the language of all pairs of Boolean formulas (φ_1, φ_2) such that ϕ_1 and ϕ_2 are not equivalent is an NP language. We can guess an assignment and check that the formulas differ on this assignment in polynomial time. Assuming P = NP, this language and its complement, the equivalence problem for formulas, are in P.

A formula is not minimal if there exists a smaller formula that is equivalent to it. Thus, given the above, *MIN-FORMULA* ∈ NP because we can guess a smaller formula and check equivalence. Again, assuming P = NP, we have $\overline{\textit{MIN-FORMULA}} \in$ P, and hence *MIN-FORMULA* ∈ P.

Exercise 51. Question to fill in.

to do

Chapter 8

Space Complexity

Definition 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.

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

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

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

Theorem 8.4. *For $t(n) \geq n$,*

$$\text{TIME}(t(n)) \subseteq \text{SPACE}(t(n))$$
$$\text{SPACE}(t(n)) \subseteq \text{TIME}(2^{O(t(n))}) = \bigcup_c \text{TIME}(c^{t(n)})$$

Proof Sketch. A TM that runs in $t(n)$ steps cannot use more than $t(n)$ tape cells. A TM that uses $t(n)$ tape cells cannot use more than $c^{t(n)}$ time without repeating a configuration and looping (for some c, for example $c^{t(n)} \leq |\Gamma|^{t(n)} \cdot t(n) \cdot |Q|$). ⊣

Comment. $t(n) \geq n$ would be a common assumption, for that we would like to consider bounds that are at least big enough to either read the input (time) or hold the input (space).

Corollary 8.5. $\text{P} \subseteq \text{PSPACE}$.

Theorem 8.6. NP $\subseteq$ PSPACE.

Proof Sketch. Obviously $SAT \in$ PSPACE. Also, if $A \leq_P B$ and $B \in$ PSPACE, then $A \in$ PSPACE, for that it takes polynomial time, thus space by 8.5, to convert A to B. Alternatively, we may devise a scheme to try all branches of a NTM in space polynomial to what is required to execute one. $\dashv$

Definition 8.7. coNP $= \{\overline{A} | A \in \text{NP}\}$.

Corollary 8.8. coNP $\subseteq$ PSPACE.

Proof Sketch. PSPACE = coPSPACE. $\dashv$

The complement of any NP-complete language is coNP-complete. (The reduction funcitons are shared.) Below is a figure on a candidate landscape of complexity classes. Interestingly, we do not even know if P = PSPACE.

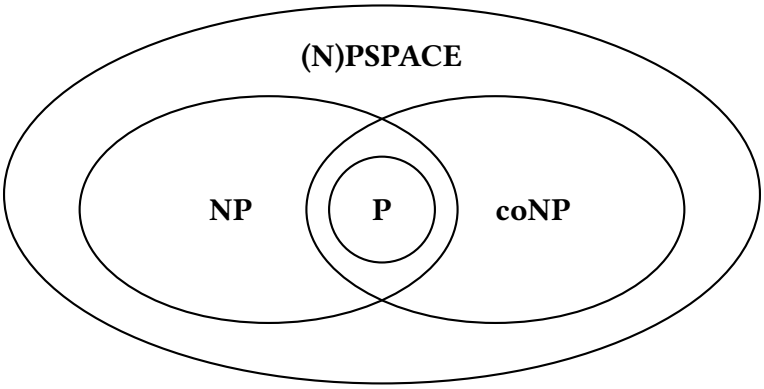

Figure 8.1: Candidate landscape of complexity classes.

Definition 8.9. A *quantified Boolean formula* (QBF) is a Boolean formula with quantifiers that cover all variables. It is very similar to the notion of sentence in logic. A QBF is `True` or `False`. $TQBF = \{\langle\varphi\rangle | \varphi$ is a `True` QBF$\}$.

Note that adding existential quantifiers ahead makes *SAT* a special case of *TQBF*.

Theorem 8.10. *TQBF* $\in$ PSPACE.

Proof Sketch. We remove quantifiers and evaluate the remaining formulas recursively. Each recursive level uses constant space to record the value we plug in. The recursion depth is the number of quantifiers, at most $n = |\langle\varphi\rangle|$. So $TQBF \in \text{SPACE}(n)$. ⊣

Definition 8.11. A *ladder* is a sequence of strings of a common length where consecutive strings differ in a single symbol. Let

$$LADDER_{\text{DFA}} = \{\langle B, u, v\rangle | B \text{ is a DFA and } L(B) \text{ contains a ladder } y_1, y_2, \dots, y_k \text{ where } y_1 = u \text{ and } y_k = v\}.$$

Theorem 8.12. $LADDER_{\text{DFA}} \in \text{NPSPACE}$.

Proof Sketch. We nondeterministically try out all possiblities of variants of u at each step and reject when a variant is not in $L(B)$. We will get this done within $|\Sigma|^{|u|}$ steps and $O(n)$ space. ⊣

Theorem 8.13. $LADDER_{\text{DFA}} \in \text{SPACE}(n^2)$.

Proof Sketch. We can do this similarly to 7.19, which uses DP. Note that we have arbitrarily long time to consume. Write $u \xrightarrow{b} v$ if there exists a ladder from u to v whose length is bounded by b. Let $B\text{-}L = \{\langle B, u, v, b\rangle | B \text{ is a DFA and } u \xrightarrow{b} v \text{ by a ladder in } L(B)\}$. To decide B-L, for $b = 1$, accept if $u, v \in L(B)$ differs at at most one place (they could be the same, to allow for a ladder length *bounded by*, not *equals*, b), else rejcet. For $b > 1$, recursively test $u \xrightarrow{b/2} w$ and $w \xrightarrow{b/2} v$. (If one ladder bounded by b exists, it will be discovered this way.) Now, apply this procedure with $b = |\Sigma|^{|u|}$. The recursion depth is $\log |\Sigma|^{|u|} = O(n)$ and each recursive level uses space $O(n)$ to record w, so the total space used is $O(n^2)$. ⊣

Theorem 8.14. ***SAVITCH*** *For $f(n) \geq n$, $\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$. (Thus* NPSPACE = PSPACE*.)*

Proof Sketch. We really simply replace ladders in 8.13 with tableaus, described in 7.27. So now u and v are configurations, b steps of computation, and $|u| = f(n)$. We then apply this procedure with $b_0 = |Q| \times f(n) \times |\Gamma|^{f(n)}$, u the starting configuration and v the accepting configuration. (We can force a single accepting configuration, by for example telling the machine to erase the tape on acceptence and having a single accepting configuration: an empty tape.) Again, each recursion level stores 1 config using space $O(f(n))$. The number of levels would be $\log b_0 = O(f(n))$. Thus

the total space used is $O(f^2(n))$. To know $f(n)$ on input w given N, we simply try $f(n) = 1, 2, \ldots$ using the procedure described until M accepts. ⊣

Theorem 8.15. *B is* PSPACE-hard *iff for all $A \in$ PSPACE, $A \leq_{\mathrm{P}} B$, and is* PSPACE-complete *if additionally we have $B \in$ PSPACE.*

We are not using $\leq_{\mathrm{PSPACE}}$, for that it will make *every* language in PSPACE PSPACE-complete, thus uninsteresting. In general, we would not like to make reductions too powerful.

Theorem 8.16. *TQBF is PSPACE-complete.*

Proof (Attempts). Let $A \in$ PSPACE be decided by TM M in space n^k. Construct $f : w \mapsto \langle \varphi_{M,w} \rangle$ such that $w \in A$ iff $\varphi_{M,w}$ is `True`. We essentially try to make $\varphi_{M,w}$ say that M accepts w, like what we did in 7.27. Yet we cannot use directly the formulas in that proof, for that the formulas will take exponential time to construct (due to our tableau size in this case being $n^k \times |\Sigma|^{n^k}$).

Instead for configurations c_i and c_j, we construct $\varphi_{c_i,c_j,b}$ which says $c_i \xrightarrow{b} c_j$ recursively:

$$\varphi_{c_i,c_j,b} = \exists c_{\mathrm{mid}} \left[\varphi_{c_i,c_{\mathrm{mid}},b/2} \wedge \varphi_{c_{\mathrm{mid}},c_j,b/2} \right].$$

Here $\exists c_{\mathrm{mid}}$ indeed says that there exists such a middle configuration in a fashion same as 7.27. At the base cases where $b = 1$, we can use formulas similar to those in 7.27 to say that a step is legal. Yet this construction fails again. We are effectively doing no more than 7.27, just fancier, and note that each recursive layer doubles the number of QBFs, so the final outcome is *still* a formula of exponential size (and time)!

This time we use $\forall$, to avoid the exponentially growing size:

$$\varphi_{c_i,c_j,b} = \exists c_{\mathrm{mid}} \forall (c_g, c_h) \left[\left((c_g, c_h) \in \{(c_i, c_{\mathrm{mid}}), (c_{\mathrm{mid}}, c_j)\} \right) \to \varphi_{c_g,c_h,b/2} \right].$$

So same thing is said, except that this time each recursive level adds $O(n^k)$ to the QBF, not doubling its size. The number of levels is $\log |\Gamma|^{n^k} = O(n^k)$, thus the total size is $O(n^{2k})$, and we are done. Note that we did not use the determinism of M, and this actually gives a second (yet very same) proof of 8.14. ⊣

Definition 8.17. The *generalized geography game* is played on a directed graph. Players take turns picking nodes that form a simple path. The first player unable to move loses.

$$GG = \{\langle G, a \rangle | \text{ Player I has a forced win in generalized geography game on graph } G \text{ starting at node } a\}.$$

Theorem 8.18 (8.14). *GG is PSPACE-complete.*

We show this by showing that $TQBF \leq_P GG$, which is briefly described in 8.20.

Remark 8.19. We define the *Formula Game*. Given the QBF

$$\varphi = \exists x_1 \forall x_2 \cdots (\forall / \exists x_k) \psi,$$

consider two players $\exists$ and $\forall$, assigning values to their corresponding variables in the order of the quantifiers in φ. Player $\exists$ wins if the assignment satisfies ψ; otherwise, $\forall$ wins. Thus, $\exists$ has a forced win in the formula game on φ iff φ is `True`. Equivalently,

$$\{\langle \varphi \rangle \mid \exists \text{ has a forced win on } \varphi\} = TQBF.$$

Remark 8.20. It turns out we can carefully devise a graph such that the geography game on it essentially mimics the formula game. The complete proof can be found on page 345 of the book.

Definition 8.21. To define sublinear space computation, do not count input as part of space used. Use 2-tape TM model with read-only input tape.

$$\mathrm{L} = \mathrm{SPACE}(\log n)$$
$$\mathrm{NL} = \mathrm{NSPACE}(\log n)$$

Log space can represent a constant number of pointers into the input, for that each would take $\log n$ space. Perceive this idea like the internet: We visit the internet through links (pointers), and we never download it as whole.

Example 8.22. $\{ww^R\} \in \mathrm{L}$, for that we can test the first character is the same as the last, and so forth. $PATH \in \mathrm{NL}$, for that we can nondeterministically try out all paths in a graph, keeping in mind only the current node and rejecting a brach if the length of the path is larger than the number of nodes in the graph.

It is unsolved whether L equals NL.

Theorem 8.23. $\mathrm{L} \subseteq \mathrm{P}$.

Well, this simply says that 8.4 holds even when $t(n) < n$, and the proof is the same. We only need

to define a configuration for M on w seperately (which contains the working tape and the tape head on *both* tapes). Also, by 8.14 we have

Theorem 8.24. NL $\subseteq$ SPACE($\log^2 n$).

Note that $\log n$ is the lower threshold for $f(n)$ or $t(n)$, because with less space than that we cannot access the whole input by any means. In 8.14, particularly, we needed the space to store *one* configuration (containing *one* tape head on input), and that will already take $\log n$ space.

Theorem 8.25. NL $\subseteq$ P.

Proof Sketch. Say that NTM M decides A in log space.

(NL reduction to *PATH* in polynomial time.) Consider a graph $G_{M,w}$ for M on w that has nodes being all configurations for M on w and edges connecting c_i and c_j iff a legal step from c_i to c_j exists. Thus M accepts w iff the $G_{M,w}$ has a path from c_{start} to c_{accept}. (If we modify M to erase work tape and move heads to left end upon accepting so that there is only one accepting configuration.)

(*PATH* $\in$ P.) Now on input w, we construct $G_{M,w}$, which has a polynomial size and thus takes polynomial time to construct, and search for paths from c_{start} to c_{accept}, which also takes polynomial time. $\dashv$

Definition 8.26. B is NL-complete iff $B \in$ NL and $\forall A \in$ NL, $A \leq_{\text{L}} B$.

Definition 8.27. A *log-space transducer* is a TM with three tapes: a read-only input tape of size n, a work tape of size $O(\log n)$, and a write-only output tape. $\leq_{\text{L}}$ is defined using this notion.

Theorem 8.28. *If $A \leq_{\text{L}} B$ and $B \in$ L, then $A \in$ L.*

Proof Sketch. This is surprisingly not obvious for that $f(w)$ might cost us more than log space. A workaround is that we compute individual symbols of $f(w)$ that the decider for B requires at each computing step. So we may re-compute $f(w)$. $\dashv$

Remark 8.29. All typical NP-complete reductions can be done in log space.

Theorem 8.30. *PATH is NL-complete.*

Proof Sketch. So we really want to make the reduction in 8.25 a log-space one. To achieve this, we go over all possible *pairs* of configurations in some order and see if any of them makes a legal step in M. If one do print it as an edge. ⊣

Theorem 8.31. *$\overline{2SAT}$ is NL-complete.*

Proof Sketch. We show that $PATH \leq_{\mathrm{L}} \overline{2SAT}$. For each node u in G put a variable x_u in φ. For each edge (u, v) in G put a clause $(x_u \to x_v)$ in φ. In addition put the clauses $(x_s \vee x_s)$ and $(x_s \to \overline{x_t})$ in φ.Thus if φ is unsatisfiable, we conclude that path from s to t exists. Similarly, we have that $\overline{2SAT} \in$ NL, by nondeterministically guessing a starting variable x_u and again guessing a path, where each step is one of the clauses transformed as $(x_i \to x_j)$ to derive a contradiction.

In both parts, if contradictions do not exist, we can assign truth values to variables accordingly to satisfy φ. (For the second part, assign each variable a node to see that it is exactly the same as the first part.) That said, we state informally that graphs and $2SAT$ formulas are *isomorphic*. ⊣

Remark 8.32. NL is the class of languages that are log space verifiable. Cf. 7.17.

Theorem 8.33. ***Immerman-Szelepcsényi*** NL = coNL.

Proof Sketch. It suffices to show that $\overline{PATH} \in$ NL, which in turn reduces to computing $path$, defined as follows.

Definition 8.34. NTM M computes function f if for all w, if all branches of M on w halt with $f(w)$ on the tape or reject and some branch does not reject.

Let $path(\langle G, s, t\rangle) = \begin{cases} \text{YES}, & \text{if } G \text{ has path from } s \text{ to } t, \\ \text{NO}, & \text{if not.} \end{cases}$.
Let $R(G, s) = \{u | path(\langle G, s, u\rangle) = \text{YES}\}$, $c(\langle G, s\rangle) = |R(G, s)|$, and m be the number of nodes in G.

Lemma 8.35. *If some log-space NTM computes $path$, then another computes c.*

Proof Sketch. We simply go over all nodes and count. Computing $path$ can turn out unsuccessful on some branchs, yet *some* branch will successfully compute all paths and produce the right answer. Conversely, ⊣

Lemma 8.36. *If some log-space NTM computes c, then another computes $path$.*

Proof. We effectively guess all c nodes and see if t is among them.

}} On input $\langle G, s, t\rangle$

1. Compute c and set $k \leftarrow 0$.
2. For each node u, nondeterministically go to (a) or (b):
   (a) Nondeternimistically pick a path from s to u with length no more than m.
   If fail, *reject*.
   Else If $u = t$, then *output* YES.
   Else set $k \leftarrow k + 1$.
   (b) Skip u and continue.
3. If $k \neq c$, *reject*. Else *output* NO." ⊣

Comment. The description can appear a bit techical to correctly capture the notion of nondeterminism.

Switching gears!

Let $path_d(\langle G, s, t\rangle) = \begin{cases} \text{YES}, & \text{if } G \text{ has a path of length} \leq d \text{ from } s \text{ to } t, \\ \text{NO}, & \text{if not.} \end{cases}$

Let $R_d(G, s) = \{u | path_d(\langle G, s, u\rangle) = \text{YES}\}$ and $c_d(\langle G, s\rangle) = |R_d(G, s)|$.

Now, with almost the same proof, we have:

Corollary 8.37. *If some log-space NTM computes $path_d$, then another computes c_d.*

Altering the proof of 8.36 a little bit, changing "If $u = t$" to "If u has an edge to t", we obtain:

Corollary 8.38. *If some log-space NTM computes c_d, then another computes $path_{d+1}$.*

By the two corollaries above,

Corollary 8.39. *We can use a log-space NTM that computes $path_d$ to compute $path_{d+1}$.*

Then we compute $path_1$ all the way to $path_m$, and that is all we need. ⊣

$\overline{2SAT}$ is NL-complete, so it is coNL-complete, thus $2SAT$ is NL-complete.

Exercises and Problems

Exercise 8. 9.8.

Exercise 9. 8.13.

Exercise 10. Question to fill in.

to do

Exercise 13. Question to fill in.

to do

Exercise 14. Question to fill in.

to do

Exercise 16. Consider *MIN-FORMULA* in 7.46.

(a) Show that *MIN-FORMULA* ∈ PSPACE.

(b) Explain why this argument fails to show that *MIN-FORMULA* ∈ coNP:
If $\varphi \notin$ *MIN-FORMULA*, then φ has a smaller equivalent formula. An NTM can verify that $\varphi \in$ *MIN-FORMULA* by guessing that formula.

(a) NPSPACE = PSPACE, and both include NP, hence by 7.46 this is immediate.
(b) This argument ommitted the procedure to test equivalence. Inequivalence can be tested nondeterministically in polynomial time but we don't have knowledge for its counterpart.

Exercise 22. Question to fill in.

to do

Exercise 27. Question to fill in.

to do

Chapter 9

Intractability

9.1 Hierarchy Theorems

Definition 9.1. A function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is at least $O(\log n)$, is called *space constructible*, if the function is computable in space $O(f(n))$.

For example, $\log\log\log n$ is not one. In fact it is *equivalent* as constant space, indicating regular languages.

Theorem 9.2. *For any space constructible function $f : \mathbb{N} \to \mathbb{N}$, a language A exists that is decidable in $O(f(n))$ space but not in $o(f(n))$ space. In other words,* $\text{SPACE}(o(f(n))) \subsetneq \text{SPACE}(O(f(n)))$.

Proof Sketch. We exhibit $A \in \text{SPACE}(f(n))$ but $A \notin \text{SPACE}(o(f(n)))$, where $A = L(D)$ and D is a TM we are to construct that runs in $f(n)$ space and ensures that $L(D) \neq L(M)$ for every M that runs in $o(f(n))$ space.

$D =$ }} On input w

1. Mark off $f(|w|)$ tape cells. If ever try to use more tape, *reject*.
2. If $w \neq \langle M \rangle 10^*$ for some TM M, *reject*.
3. Simulate M on w for $|\Gamma_M|^{f(n)} \cdot f(n) \cdot |Q_M|$ steps.
   Accept if M rejects.
   Reject if M accepts or hasn't halted."

Subtleties:

1. We actually do not care the case where D or M loops, for that we only consider deciders when talking about space complexity.
2. D can simulate M with a constant factor space overhead, even at worst cases where M has a larger alphabet than D.

3. In case where M runs in $o(f(n))$ but with a huge constant, we simulate on infinitely many ws, adding 0s to the input and removing them during computation. Note that this affects the strings that is going to be accepted by D, but not the determinism of D. ⊣

Comment. This construction, namely diagonization, is based on a key observation: a TM can simulate another given asymtotically larger running space. To utilize this we need a one-to-one mapping between TMs and strings to make sure that D behaves differently on at least one string from every TM with a smaller running space. Suppose there is an oracle that one-to-one maps TMs to an arbitrary set of strings and the other way, we might as well query it to construct D so that it behaves differently from M on the string mapped from M. What's so special about $\langle M \rangle$ is that it *is* one-to-one mapped from M, easy to decode, and does not demand an oracle or some infinite storage, and that is why we choose to define D's behavior on $\langle M \rangle$ instead of some random string.

By 8.14 we have:

Corollary 9.3. NL $\subsetneq$ PSPACE.

Corollary 9.4. *TQBF* $\notin$ NL.

Proof. Suppose it is. Note that the reduction in 8.16 is a log space one. That is, *every* PSPACE language is in NL, contradicting 9.3. ⊣

Also, L = P and P = PSPACE cannot be both true.

Theorem 9.5. *For any time constructible (similarly defined as 9.1) function $f : \mathbb{N} \to \mathbb{N}$, there is a language A where A is decidable in $O(f(n))$ time not decidable in $o\left(\frac{f(n)}{\log f(n)}\right)$ time.*

Proof Sketch. This theorem is proved almost exactly the same as 9.2, except that we do not limit the space used and simulate M on w for $\frac{f(n)}{\log f(n)}$ steps, so that if M can be simulated in $\frac{f(n)}{\log f(n)}$ steps, D behaves differently from M on $\langle M \rangle$. But we lose a $\log f(n)$ factor, for that to count the steps used we need to move with the tape head a counter of size $\log f(n)$. (We don't want to keep it at the beginning of the tape for that will cost more time to update.) ⊣

We may need some method other than diagonization to prove L $\neq$ P and P $\neq$ PSPACE.

Definition 9.6.

$$\text{EXPTIME} = \bigcup_k \text{TIME}\left(2^{n^k}\right),$$
$$\text{EXPSPACE} = \bigcup_k \text{SPACE}\left(2^{n^k}\right).$$

We can simulate an amount of space using exponential time, so now the landscape is

$$\text{L} \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{EXPSPACE}.$$

EXPTIME-complete, EXPTIME-hard, EXPSPACE-complete, and EXPSPACE-hard are defined the way you would suppose they are. Reductions are polynomial.

Call a language *intractable* if it is not in P. By 9.2 and 9.5, we have that EXPTIME-complete and EXPSPACE-complete languages are not in P, thus intractable.

Definition 9.7.

$$EQ_{\text{REX}} = \{\langle R_1, R_2\rangle \mid L(R_1) = L(R_2)\}.$$

Theorem 9.8. $EQ_{\text{REX}} \in \text{PSPACE}$.

Proof. We want to capture the finiteness of regular expressions, exhausting all possible equivalence classes of strings, and converting them to NFA helps. We cannot convert them to DFAs, for that conversion can take exponential space.

Convert R_1 and R_2 to NFAs M_1 and M_2 respectively. And we nondeterministically guess for $2^{|Q_1|+|Q_2|}$ symbols making up a string that is accepted by R_1 and rejected by R_2. If there exists one, there exists one with length smaller than $2^{|Q_1|+|Q_2|}$, for that if a longer one exists, the set of states in both NFAs would have repeated, indicating that the section of the string between repetitions can be removed to obtain a shorter satisfying string. ⊣

Regular expressions with exponentiation are regular expressions (1.6) that are allowed *exponentiation* (e.g., $R^2 = RR$) operation at generation.

Definition 9.9. Let

$$EQ_{\text{REX}\uparrow} = \{\langle R_1, R_2\rangle | R_1 \text{ and } R_2 \text{ are equivalent regular expressions with exponentiation}\}.$$

Theorem 9.10. *$EQ_{\text{REX}\uparrow}$ is EXPSPACE-complete.*

Proof Sketch. Given regular expressions with exponentiation R_1 and R_2. Expand them to regular expressions and by 9.8 we have $EQ_{\text{REX}} \in \text{PSPACE}$.

Consider $A \in \text{EXPSPACE}$ decided by TM M in space $2^{(n^k)}$. We are to construct a polynomial time reduction f mapping A to $EQ_{\text{REX}\uparrow}$, that is, $f(w) = (R_1, R_2)$ and $w \in A$ iff $L(R_1) = L(R_2)$. Now construct R_1 so that $L(R_1)$ is the set of all strings excepts *rejecting* computation histories for M on w and $R_2 = \Delta^*$, where $\Delta = \Gamma \cup Q \cup \{\#\}$, being the alphabet for computation histories.

A string can fail to be a rejecting computation if it starts wrongly, moves wrongly, or rejects wrongly, that is, $R_1 = R_{\text{bad-start}} \cup R_{\text{bad-move}} \cup R_{\text{bad-reject}}$.

First, pad all configurations with blanks to have length $2^{(n^k)}$ as an encoding requirement to facilitate the construction of $R_{\text{bad-move}}$.

The size of the rejecting computation history for M on w is roughly $2^{2^{(n^k)}}$, and we can only construct R_1 with length being polynomial of $2^{(n^k)}$, because f is a polynomial time reduction. Therefore we need to capture 'not being a rejecting computation history' within polynomial length, by utilizing exponentiation in regular expressions.

$R_{\text{bad-start}}$ generates all strings that do not start with $C_{\text{start}} = q_0 w_1 w_2 \cdots w_n \sqcup \cdots \sqcup$. Let $\Delta_\varepsilon = \Delta \cup \{\varepsilon\}$ and $\Delta_{-b} = \Delta \setminus \{b\}$. Then consider

$$R_{\text{bad-start}} = S_0 \cup S_1 \cup \cdots \cup S_n \cup S_{\text{blanks}} \cup S_\#.$$

where $S_0 = \Delta - q_0\Delta^*, S_1 = \Delta\Delta_{-w_1}\Delta^*, \ldots, S_n = \Delta^n\Delta_{-w_n}\Delta^*$, and

$$S_{\text{blanks}} = \Delta^{n+1}\Delta_\varepsilon^{2^{(n^k)}-(n+2)}\Delta_{-\sqcup}\Delta^*,$$

capturing all strings with correct prefix $q_0 w_1 w_2 \cdots w_n$ yet violating the following blanks. And $S_\# = \Delta^{2^n}\Delta_{-\#}\Delta^*$.

S_{blanks}, though a little bit wierd, is a valid regular expression. In fact it is trivial that $R_{\text{bad-start}}$ (and the other two R's) can be validly expressed, by that FAs are equivalent to regular expressions and those three notions (including S_{blanks}, of course), containing *finite* equivalence classes (c.f. 1.52), can surely be described by FAs. We are only using *exponentiation* (in $R_{\text{bad-start}}$ and $R_{\text{bad-move}}$ in particular) to make this expression *polynomial* in length.

$$R_{\text{bad-move}} = \bigcup_{\text{illegal(a,b,c,d,e,f)}} \left[\Delta^* \text{abc} \Delta^{2^{(n^k)-2}} \text{def} \Sigma^*\right],$$

where illegal is defined according to how legal is defined in 7.27. This is a delicate expression that makes good use of exponentiation and our paddings.

At last we have $R_{\text{bad-reject}} = \Delta^*_{-q_{\text{reject}}}$, a straightforward one. ⊣

So $EQ_{\text{REX}\uparrow}$ is intractable.

9.2 Relativization

Definition 9.11. We further specify the behavior of oracles (c.f. 6.8): Whenever M^A writes a string on a special oracle tape, it is informed whether that string is a member of A in a single computation step. And Let P^A be the class of languages decidable with a polynomial time oracle Turing machine that uses oracle A. Define NP^A similarly.

Thus $\mathrm{NP} \subseteq \mathrm{P}^{SAT}$. Also $\mathrm{coNP} \subseteq \mathrm{P}^{SAT}$ because P^{SAT} is closed under complementation.

Example 9.12. $\overline{\textit{MIN-FORMULA}} \in \mathrm{NP}^{SAT}$ (7.46). On input $\langle\varphi\rangle$, we can guess a shorter formula φ' and query the oracle if $\varphi \neq \varphi'$ (namely $\varphi \leftrightarrow \overline{\varphi'}$) is satisfiale. (Or, we can use the fact that $\mathrm{coNP} \in \mathrm{NP}^{SAT}$ and apply the oracle to test equivalence, which is a coNP problem.)

Theorem 9.13. *An oracle B exists whereby $\mathrm{P}^B = \mathrm{NP}^B$.*
An oracle A exists whereby $\mathrm{P}^A \neq \mathrm{NP}^A$.

Proof. Let $B = TQBF$ (or any PSPACE-complete language.) It is obvious that

$$\mathrm{NP}^{TQBF} \subseteq \mathrm{NPSPACE} = \mathrm{PSPACE} \subseteq \mathrm{P}^{TQBF}.$$

Now we construct A, so that a language L_A exists that is in P^A and not in NP^A. Note that the key restriction on P^A is that only polynomially many queries can be made, yet A as a language can bear exponential possibilities (given length). Thus we try to force every polynomial TM to decide wrongly whether a string is in L_A. The construction scheme is as follows.

First, for any oracle A, let

$$L_A = \{w \mid \exists x \in A, |x| = |w|\},$$

making it possible to force queries on the membership of *all* strings of length n to decide whether 1^n is in L_A. Also, obviously $L_A \in \mathrm{NP}^A$.

Now construct A. Let $M_1, M_2, \ldots$ be a list of all polynomial time oracle TMs. Assume for simplicity that M_i runs in time n^i. We begin with no knowledge of A and gradually declare that some string is or is not in A. For each i, pick a n that is (1) larger than the length any string whose membership has been declared and (2) large enough that $2^n > n^i$.

Run M_i on 1^n and respond to its queries as follows until it halts. If M_i queries a string whose membership has been declared, repond consistently. Otherwise respond NO.

Then *force* its decision to be wrong. If M_i rejects, find a string of length n that M_i hasn't queried and declare it to be in A. Otherwise declare that all strings of length n is not in A. After that declare the membership of any string of length at most n that is yet to be declared arbitrarily and proceed to $i + 1$. ⊣

The existence of A in the above theorem, though demonding more writting, feels trivial: the existence of B tells us that P and NP are 'not that different'.

Now consider $EQ_{\text{REX}\uparrow} \notin \text{PSPACE}$ (9.10) which essentially depends on diagonlization, where the TM D simulates some TM M (9.2). This is not a scheme that is going to apply in the proof of $SAT \in \text{P}$ (if there is to be one), because diagonlization *does not forbid* the use of oracles. That is, if we can prove $\text{P} \neq \text{NP}$ by simulation, we can as well prove $\text{P}^A \neq \text{NP}^A$ for every oracle A, which is not the case.

That asks for some kind of *analysis* aside of naive *simulation*.

9.3 Circuit Complexity

Computers are built from *digital circuits*. We can simulate models like TMs with the theoretical counterpart to digital circuits called *Boolean circuits*. Researchers believe that circuits simulating TMs provide a model for attacking the P versus NP and related questions.

Definition 9.14. A *Boolean circuit* is a collection of *gates* and *inputs* connected by *wires*. Cycles are not permitted. Gates take three forms: AND gates, OR gates, and NOT gates. The wires carry the Boolean values 0 and 1. The inputs are labeled $x_1, \dots, x_n$. One of the gates is designated the *output gate*. To a Boolean circuit C with n input variables, we associate a function $f_C : \{0,1\}^n \to \{0,1\}$ (the range can be $\{0,1\}^k$ in cases where there are k outputs) and say that C computes f_C.

Definition 9.15. A *circuit family* C is an infinite list $(C_0, C_1, C_2, \dots)$, where C_n is a circuit with n input variables. We say that C decides a language A over $\{0,1\}$ if, for every string w,

$$w \in A \quad \text{iff} \quad C_n(w) = 1,$$

where n is the length of w.

Definition 9.16. The *size* of a circuit is the number of gates that it contains. Two circuits are equivalent if they compute the same function. A circuit is *size minimal* if no smaller circuit is equivalent to it. A circuit family is minimal if every C_i on the list is a minimal circuit. Complexity, or *size complexity*, of a circuit family $(C_0, C_1, C_2, \dots)$ is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the size of C_n. The *circuit complexity* of a language is the size complexity of a minimal circuit family for that language.
The *depth* of a circuit is the length (number of wires) of the longest path from an input variable to the output gate. The above notions are defined with circuit depth as with did with circuit size.

Theorem 9.17. *Let $t : \mathbb{N} \to \mathbb{N}$ be a function, where $t(n) \geq n$. If $A \in \mathrm{TIME}(t(n))$, then A has circuit complexity $O(t^2(n))$.*

Proof Idea. Let M be a TM that decides A in time $t(n)$. For each n, construct a circuit C_n that simulates M on inputs of length n. The gates of C_n are organized in rows, one for each of the $t(n)$ steps in M's computation on an input of length n. Each row of gates represents the configuration of M at that corresponding step. So C_n basically forms a tableau (7.27). Each cell in the tableau holds a number of Boolean variables with exactly one of them being true (just like $x_{i,j,s}$ in 7.27). Each row is wired into the previous row so that it can calculate its configuration from the previous row's configuration.

For example, say that $cell[i-1, j-1]$, $cell[i-1, j]$, and $cell[i-1, j+1]$ containing a, b, c leads to $cell[i, j]$ containing s according to δ, then connect the three variables at the $i-1$ level to an AND gate whose output is connected to the corresponding variable in $cell[i, j]$.

Modify M so that the input is encoded into $\{0, 1\}$. To designate a gate as the output gate, we require that M moves its head onto the leftmost tape cell and writes the $\sqcup$ symbol on that cell before entering the accept state. ⊣

Say that a Boolean circuit is *satisfiable* if some setting of the inputs causes the circuit to output 1. Let

$$\textit{CIRCUIT-SAT} = \{\langle C \rangle | C \text{ is a satisfiable Boolean circuit}\}.$$

Theorem 9.18. *CIRCUIT-SAT is NP-complete.*

Proof. *CIRCUIT-SAT* is easily in NP. For any language A in NP, we give a polynomial time reduction $f : w \mapsto \langle C \rangle$ satisfying that $w \in A$ iff Boolean circuit C is satisfiable.

A is in NP, so it has a polynomial time verifier V (7.16) whose input has the form $\langle x, c\rangle$. Obtain the circuit simulating V using the method in 9.17. Fill the inputs to the circuit that correspond to x with the symbols of w. This is the circuit C.

If C is satisfiable, a certificates exists and $w \in A$. Conversely, if w is in A, a certificate exists and C is satisfiable. That the reduction takes polynomial time is trivial. ⊣

Below is an alternative proof of the Cook-Levin theorem (7.27).

Theorem 9.19. *3SAT is NP-complete.*

Second Proof of 7.29. We reduce *CIRCUIT-SAT* to *3SAT*, simulating the circuits with formulas. So very similar to, again, 7.27. Much easier this time, for that gates are themselves Boolean functions.

Given a circuit C, the $\phi \in$ *3SAT* is built with variables corresponding to wires in C. Its clauses include expressions that make sure that each gate's input and output wires are in line with the Boolean function described by the gate and one specific clause $(w_m \vee w_m \vee w_m)$, where w_m is the output wire, ensuring that when ϕ is satisfied, the corresponding inputs make C output 1, thus satisfiable. ⊣

Exercises and Problems

Exercise 17. Prove that P contains a language not recognizable by a 2DFA (5.26).

to do

Exercise 19. Define the *unique-sat* problem to be

$$USAT = \{\langle \phi \rangle \mid \phi \text{ has exactly one satisfying assignment}\}.$$

Show that $USAT \in \mathrm{P}^{SAT}$.

On input $\langle \phi \rangle$ containing variables $x_1, \dots, x_n$, query first if ϕ is satisfiable. If it is, let $\phi = \phi_1$ and do the following for $i = 1, \dots, n$: fix x_i in ϕ_i as `True` and query again. If it is still satisfiable, fix x_i as `True` for later i's. If not fix x_i as `False`.

Thus we obtain a satisfying truth assignment of ϕ. Then we construct a CNF ϕ' satisfied only by this assignment and query if $\overline{\phi'} \wedge \phi$ is satisfiable. If it is, reject. Otherwise accept.

Chapter 10

Advanced Topics in Complexity Theory

10.1 Approximation Algorithms

A problem that has a yes/no answer is roughly called a *decision problem*, and in certain problems called *optimization problems*, we seek the best solution among a collection of possible solutions. In practice, we may not need the *optimal* solution to an optimization problem. *Approximation algorithms* are designed to find approximately optimal solutions.

An approximation algorithm for a minimization problem is *k-optimal* if it always finds a solution that is not more than k times optimal. An approximation algorithm for a maximization problem is *k-optimal* if it always finds a solution that is at least $1/k$ times the size of the optimal.

10.2 Probabilistic Algorithms

10.2.1 The Class BPP

Definition 10.1. A *probabilistic Turing machine* M is a variant of NTM where each nondeterministic step is called a *coin-flip step* and has two legal next moves. Assign a probability to each branch b of M's computation on input w being $\Pr(b) = 2^{-k}$, where k is the number of coin-flip steps that occur on branch b. Define the probability that M accepts w to be

$$\Pr(M \text{ accepts } w) = \sum_{b \text{ accepts}} \Pr(b),$$

and let $\Pr(M \text{ rejects } w) = 1 - \Pr(M \text{ accepts } w)$.

Definition 10.2. For $0 \leq \epsilon < 1/2$, say that M *decides* A *with error probability* ϵ if

1. $w \in A$ implies $\Pr(M \text{ accepts } w) \geq 1 - \epsilon$, and
2. $w \notin A$ implies $\Pr(M \text{ rejects } w) \geq 1 - \epsilon$.

Note that $\epsilon \geq 1/2$ is uninteresting. Also, unlike the case in NP, M may accept even if $w \notin A$. That is, it is possible that $A \neq L(M)$. We only obtain an approximation.

Definition 10.3.

$$\text{BPP} = \left\{ A \,\middle|\, \begin{array}{c} \text{some polynomial-time PTM decides } A \\ \text{with error probability } 1/3 \end{array} \right\}.$$

$1/3$ is arbitrarily chosen in $[0, 1/2)$, due to the following lemma.

Lemma 10.4. ***Amplification*** *Fix $\epsilon \in [0,\ 1/2)$, then for any polynomial $p(n)$, a polynomial time PTM M_1 with error probability ϵ has an equivalent polynomial time PTM M_2 with error probability $2^{-p(n)}$.*

Proof. The idea is to run the PTM polynomial times and take the majority vote of the outcomes. Say that M_2 runs M_1 for $2k+1$ times, then

$$\begin{aligned} &\Pr(M_2 \text{ outputs incorrectly}) \\ &= \sum_{i>k} \binom{2k+1}{i} \epsilon^i (1-\epsilon)^{2k+1-i} \leq \sum_{i>k} \binom{2k+1}{i} \epsilon^{k+1} (1-\epsilon)^k \\ &\leq \left[\sum_i \binom{2k+1}{i} \right] \epsilon^{k+1} (1-\epsilon)^k \leq 2^{2k+1} \epsilon \left(\epsilon(1-\epsilon)\right)^k \\ &\leq 2\epsilon (4\epsilon(1-\epsilon))^k, \end{aligned}$$

where $4\epsilon(1-\epsilon) < 1$, completing our proof. ⊣

10.2.2 Primality

We describe a probabilistic polynomial time algorithm for primality testing. Testing input of length n by trying all its factors will take exponential time. First let $\mathbb{Z}_p = \{0, \ldots, p-1\}$ and $\mathbb{Z}_p^+ = \{1, \ldots, p-1\}$.

Theorem 10.5. ***Fermat's Little Theorem***
If p is prime and $a \in \mathbb{Z}_p^+$, then $a^{p-1} \equiv 1 (\text{mod } p)$.

Proof. We have that

$$a^p = (1 + 1 + \cdots + 1)^p = \sum_{\substack{k_1,k_2,\ldots,k_a \\ k_1+k_2+\cdots+k_a=p}} \binom{p}{k_1, k_2, \ldots, k_a}$$

If p is prime, and if k_j is not equal to p for any j, we have

$$\binom{p}{k_1, k_2, \ldots, k_a} = \frac{p!}{\prod_{i=1}^{a} k_i!} \equiv 0 \pmod{p},$$

and if $k_j = p$ for some j, then

$$\binom{p}{k_1, k_2, \ldots, k_a} = 1.$$

There are a elements such that $k_j = p$ for some j and the theorem follows. ⊣

Definition 10.6. Say that p passes the *Fermat test* at a if $a^{p-1} \equiv 1 \pmod{p}$. Call a number *pseudoprime* if it passes Fermat tests for all smaller a's *relatively prime* to it.

Pseudoprimes are introduced to make good use of 10.8 in the designing of an efficient probabilistic primality testing algorithm.

Definition 10.7. With the exception of the infrequent *Carmichael numbers* n, where

$$\forall\, a\ (1 \le a < n,\ \gcd(a, n) = 1)\ :\ a^{n-1} \equiv 1 \pmod{n},$$

the pseudoprime numbers are identical to the prime numbers.

Theorem 10.8. *For any integer $p > 1$, if p is not pseudoprime, then p fails the Fermat test for at least half of all numbers in $\mathbb{Z}_p^+$.*

Proof. Call a a *witness* if it fails the Fermat test for p; that is, if $a^{p-1} \not\equiv 1 \pmod{p}$. Let $\mathbb{Z}_p^*$ be the set of all numbers in $\{1, \ldots, p-1\}$ that are relatively prime to p. If p is not pseudoprime, it has a witness a in $\mathbb{Z}_p^*$.

Use a to obtain many more witnesses. Find a unique witness in $\mathbb{Z}_p^*$ for each non-witness. If $d \in \mathbb{Z}_p^*$ is a non-witness, then $d^{p-1} \equiv 1 \pmod{p}$. Hence

$$(da \bmod p)^{p-1} \not\equiv 1 \pmod{p},$$

so $da \bmod p$ is a witness. If d_1 and d_2 are distinct non-witnesses in $\mathbb{Z}_p^*$, then $d_1 a \bmod p \neq d_2 a \bmod p$; otherwise

$$(d_1 - d_2)a \equiv 0 \pmod{p},$$

implying $(d_1 - d_2)a = cp$ for some integer c. Because $d_1, d_2 \in \mathbb{Z}_p^*$ we have $0 < d_1 - d_2 < p$, so

$$a = \frac{cp}{d_1 - d_2}$$

shares a non-trivial factor with p, contradicting $\gcd(a, p) = 1$. Thus the number of witnesses in $\mathbb{Z}_p^*$ is at least as large as the number of non-witnesses in $\mathbb{Z}_p^*$, and consequently at least half of the elements of $\mathbb{Z}_p^*$ are witnesses.

Next, show that every $b \in \mathbb{Z}_p^+$ that is *not* relatively prime to p is a witness. If b and p share a factor, then b^e and p share that factor for any $e > 0$; hence

$$b^{p-1} \not\equiv 1 \pmod{p}.$$

Therefore, at least half of the members of $\mathbb{Z}_p^+$ are witnesses. ⊣

Remark 10.9. We can select $a_1, \ldots, a_k$ randomly in $\mathbb{Z}_p^+$ and compute $a_i^{p-1} \pmod{p}$ for each i to test for *pesudoprimes*. This algorithm operates in polynomial time because of 7.13. Yet this procedure can not tell Carmichael numbers from primes, which is fixed by the procedure described as follows.

Algorithm 10.10. **Miller–Rabin** *PRIME* = "On input p:

1. If p is even, *accept* if $p = 2$; otherwise, *reject*.
2. Select $a_1, \ldots, a_k$ randomly in $\mathbb{Z}_p^+$.
3. For each i from 1 to k:
   - 3.1. Compute $a_i^{p-1} \bmod p$ and *reject* if different from 1.
   - 3.2. Let $p - 1 = s \cdot 2^l$ where s is odd.
   - 3.3. Compute the sequence $a_i^{s \cdot 2^0}$, $a_i^{s \cdot 2^1}$, $a_i^{s \cdot 2^2}$, …, $a_i^{s \cdot 2^l} \bmod p$.
   - 3.4. If some element of this sequence is not 1, find the last element that is not 1 and *reject* if that element is not -1.
4. All tests have passed at this point, so *accept*."

Theorem 10.11. ***Chinese Remainder Theorem*** *A one-to-one correspondence exists between $\mathbb{Z}_{pq}$ and $\mathbb{Z}_p \times \mathbb{Z}_q$ if p and q are relatively prime. Each number $r \in \mathbb{Z}_{pq}$ corresponds to a pair*

(a, b), *where* $a \in \mathbb{Z}_p$ *and* $b \in \mathbb{Z}_q$, *such that*

$$r \equiv a \pmod{p}, \text{ and}$$
$$r \equiv b \pmod{q}.$$

Theorem 10.12. *Correctness of 10.10.*

(a) If p is prime, $\Pr(\textit{PRIME accepts } p) = 1$.
(b) If p is composite, $\Pr(\textit{PRIME accepts } p) \leq 2^{-k}$.

Proof. We only consider cases where p is odd.

(a) If p is rejected at 3.1, then by 10.5 p is composite. If p is rejected at 3.4, then some b exists in $\mathbb{Z}_p^+$, where $b \not\equiv \pm 1 \pmod{p}$ and $b^2 \equiv 1 \pmod{p}$. Therefore

$$p \nmid b + 1,$$
$$p \nmid b - 1,$$
$$p \mid (b + 1)(b - 1).$$

If p is prime then a contradiction will arise.

(b) Say that a_i is a witness if *PRIME* rejects using a_i. We show that if p is an odd composite number and a is selected randomly in $\mathbb{Z}_p^+$,

$$\Pr(a \text{ is a witness}) \geq 1/2$$

by demonstrating that at least as many witnesses as nonwitnesses exist in $\mathbb{Z}_p^+$. We do so by finding a unique witness for each nonwitness.

In every nonwitness, the sequence computed in 3.3 is either all 1s or contains -1 at some position, followed by 1s. Among all nonwitnesses of the second kind, find a nonwitness for which the -1 appears in the largest position in the sequence. Let h be that nonwitness and let j be the position of -1 in its sequence, where the sequence positions are numbered starting at 0. Hence

$$h^{s \cdot 2^j} \equiv -1 \pmod{p}.$$

Because p is composite, either it is the power of a prime or we can write p as the product of q and r—two numbers that are relatively prime. Consider the latter case first. By 10.11 some number t exists in $\mathbb{Z}_p$ whereby

$$t \equiv h \pmod{q} \quad \text{and} \quad t \equiv 1 \pmod{r}.$$

Therefore,

$$t^{s\cdot 2^j} \equiv -1 \pmod{q} \quad \text{and} \quad t^{s\cdot 2^j} \equiv 1 \pmod{r}.$$

Hence t is a witness because $t^{s\cdot 2^j} \not\equiv \pm 1 \pmod{p}$ but $t^{s\cdot 2^{j+1}} \equiv 1 \pmod{p}$.

Now that we have one witness, we can get many more. We prove that $dt \bmod p$ is a unique witness for each nonwitness d by making two observations.

First,

$$d^{s\cdot 2^j} \equiv \pm 1 \pmod{p} \quad \text{and} \quad d^{s\cdot 2^{j+1}} \equiv 1 \pmod{p}$$

owing to the way j was chosen. Therefore, $dt \bmod p$ is a witness because

$$(dt)^{s\cdot 2^j} \not\equiv \pm 1 \pmod{p} \quad \text{and} \quad (dt)^{s\cdot 2^{j+1}} \equiv 1 \pmod{p}.$$

Second, if d_1 and d_2 are distinct nonwitnesses, $d_1 t \bmod p \neq d_2 t \bmod p$. The reason is that $t^{s\cdot 2^{j+1}} \bmod p = 1$. Hence

$$t \cdot t^{s\cdot 2^{j+1}-1} \bmod p = 1.$$

Therefore, if $t d_1 \bmod p = t d_2 \bmod p$, then

$$d_1 = t \cdot t^{s\cdot 2^{j+1}-1} d_1 \bmod p = t \cdot t^{s\cdot 2^{j+1}-1} d_2 \bmod p = d_2.$$

Thus, the number of witnesses must be as large as the number of nonwitnesses, and we have completed the analysis for the case where p is not a prime power.

For the prime power case, we have $p = q^e$ where q is prime and $e > 1$. Let $t = 1 + q^{e-1}$. Expanding t^p using the binomial theorem, we obtain

$$t^p = (1 + q^{e-1})^p = 1 + p \cdot q^{e-1} + \text{multiples of higher powers of } q^{e-1},$$

which is equivalent to 1 mod p. Hence t is a witness at stage 3.1 because if $t^{p-1} \equiv 1 \pmod{p}$, then $t^p \equiv t \not\equiv 1 \pmod{p}$. Note that $\gcd(t, p) = 1$, so by 10.8 we are done already. Alternatively, as in the previous case, we use this one witness to get many others. If d is a nonwitness, we have $d^{p-1} \equiv 1 \pmod{p}$, but then $dt \bmod p$ is a witness. Moreover, if d_1 and d_2 are distinct nonwitnesses, then $d_1 t \bmod p \neq d_2 t \bmod p$. Otherwise,

$$d_1 = d_1 \cdot t \cdot t^{p-1} \bmod p = d_2 \cdot t \cdot t^{p-1} \bmod p = d_2.$$

Thus, the number of witnesses must be as large as the number of nonwitnesses and the proof is complete. ⊣

Definition 10.13. **RP** (Randomized Polynomial time) is the class of languages that are decided by probabilistic polynomial time TMs where inputs in the language are accepted with a probability of at least 1/2 , and inputs not in the language are rejected with a probability of 1.

Thus by 10.10 and 10.12, *COMPOSITES* ∈ RP.

10.2.3 Read-Once Branching Programs

Definition 10.14. A *branching program* (BP) is a directed acyclic graph that has

1. *Query nodes* labeled x_i and having two outgoing edges labeled 0 and 1.
2. Two *output nodes* labeled 0 and 1 and having no outgoing edges.
3. A designated *start node*.

BP B with query nodes $x_1, \ldots, x_m$ describes a Boolean function $f_B : \{0,1\}^m \to \{0,1\}$: follow the path designated by the query nodes' outgoing edges from the start node until reaching an output node. BPs are equivalent if they describe the same Boolean function. Let

$$EQ_{\text{BP}} = \{\langle B_1, B_2\rangle \mid B_1 \text{ and } B_2 \text{ are equivalent BPs.}\}$$

Theorem 10.15. *EQ_{BP} is coNP-complete.*

Proof. Note that $NSAT = \{\langle\phi\rangle \mid \phi \text{ is not a satisfiable Boolean formula}\}$ is coNP-complete. We then reduce $NSAT$ to EQ_{BP}. A BP that describes the Boolean function $(x_1, \ldots, x_m) \mapsto 0$ is trivial, so we only need a polynomial time function $f : \phi \mapsto B$, where B is a BP, described as follows.

Include in B a query node for each Boolean variable in ϕ. Let B start with arbitrary query node and define the edges of B according to the tree structure that describes ϕ, so that B computes the same Boolean function as ϕ does. For example, for $\phi = a \wedge b$, we may query b first. If $b = 0$ output 0, otherwise query a, and so on. ⊣

Definition 10.16. A BP is *read-once* if it never queries a variable more than once on any path from the start node to an output node. Let

$$EQ_{\text{ROBP}} = \{\langle B_1, B_2\rangle \mid B_1 \text{ and } B_2 \text{ are equivalent read-once BPs.}\}$$

Theorem 10.17. $EQ_{\text{ROBP}} \in \text{BPP}$.

Proof Attempt. We may attempt by randomly choosing k inputs and see if the outputs are same, yet k should be $O(2^m)$, where m is length of the input, to provide necessary confidence in our decision. ⊣

We instead describe an algorithm as follows.

Algorithm 10.18. Arithmetization Assign polynomials over $x_1, \ldots, x_m$ to the nodes and to the edges of a read-once branching program B as follows. The constant function 1 is assigned to the start node. If a node labeled x has been assigned polynomial p, assign the polynomial xp to its outgoing 1-edge, and assign the polynomial $(1-x)p$ to its outgoing 0-edge. If the edges incoming to some node have been assigned polynomials, assign the sum of those polynomials to that node. Assign the polynomial that has been assigned to the output node labeled 1 to B.
Let $\mathbb{F}$ be a finite field with at least $3m$ elements. On input $\langle B_1, B_2 \rangle$

1. Select elements a_1 through a_m at random from $\mathbb{F}$.
2. Evaluate the assigned polynomials p_1 and p_2 at a_1 through a_m.
3. If $p_1(a_1, \ldots, a_m) = p_2(a_1, \ldots, a_m)$, accept; otherwise, reject.

Lemma 10.19. *For every $d \geq 0$, a degree-d polynomial p on a single variable x either has at most d roots, or is everywhere equal to 0.*

Proof. Use induction on d. The base case where $d = 0$ is trivial.

Assume true for $d-1$. If p is a nonzero polynomial of degree d with a root at a, then $x - a$ divides p. That is, $p/(x-a)$ is a nonzero polynomial of degree $d-1$, having at most $d-1$ roots. Hence p has at most d roots. ⊣

Lemma 10.20. ***Schwarts-Zippel*** *Let $\mathbb{F}$ be a finite field with f elements and let p be a nonzero polynomial on the variables $x_1, \ldots, x_m$, where each variable has degree at most d. If $a_1, \ldots, a_m$ are randomly selected in $\mathbb{F}$, then*

$$\Pr(p(a_1, \ldots, a_m) = 0) \leq md/f.$$

Proof. We use induction on m. The base case where $m = 1$ follows from 10.19.

Assume true for $m-1$. Let x_1 be one of p's variables. For each $i \leq d$, let p_i be the polynomial comprising the terms of p containing x_1^i, but where x_1^i has been factored out. Then

$$p = p_0 + x_1 p_1 + x_1^2 p_2 + \cdots + x_1^d p_d.$$

If $p(a_1, \ldots, a_m) = 0$, one of two cases arises. Either all p_i evaluate to 0, or some p_i doesn't evaluate to 0 and a_1 is a root of the single variable polynomial obtained by evaluating p_0 through p_d on $a_2, \ldots, a_m$.

Case 1. For some j, p_j must be nonzero because p is nonzero. Then the probability that all p_i evaluate to 0 is at most the probability that p_j evaluates to 0, being $(m-1)d/f$ by IH. Case 2. On the assignment of $a_2, \ldots, a_m$, p reduces to a nonzero polynomial in x_1, and a_1 is a root of p with a probability of at most d/f.

Therefore, the probability that a_1 through a_m is a root of the polynomial is at most $(m-1)d/f + d/f = md/f$. ⊣

Theorem 10.21. *Correctness of 10.18.*

(a) If $B_1 \equiv B_2$, the algorithm accepts.
(b) If $B_1 \not\equiv B_2$, the algorithm accepts with a probability at most $1/3$.

Proof. Let's examine the relationship between a read-once BP B and its assigned polynomial p. Observe that for any Boolean assignment to B's variables, all polynomials assigned to its nodes evalute to either 0 or 1. The polynomials that evaluate to 1 are those on the computation path for that assignment. Hence B and p agree when the variables take on Boolean values. Because B is read-once, we may write p as $\sum \prod_{i=1}^{m} y_i$, where each y_i is x_i, $(1-x_i)$, or 1 (this is by induction), and where each product term corresponds to a path in B from the start node to the output node labeled 1. Observe from above that q is the sum of product terms corresponding to assignments on which B evaluates to 1. Hence a bijection exists that maps q's to (the truth tables of) B's. By 10.20 we are done. ⊣

Comments. The observation is not a trivial one, in paticular we have to split each y_i in p that equals 1 into x_i and $1-x_i$. That is, we force B to read each variable *exactly* once. Note that if $p = 1$, exactly one of the product terms should equal 1, which corresponds to a truth assignment. If the BPs are not read-once, the bijection breaks, and this method no longer suits.

10.4 Interactive Proof Systems

Definition 10.22. Call graphs G and H *isomorphic* if the nodes of G may be reordered so that it is identical to H. Let

$$ISO = \{\langle G, H\rangle \mid G \text{ and } H \text{ are isomorphic graphs}\}.$$

Although ISO is obviously in NP, extensive research has so far failed to demonstrate either a polynomial time algorithm for this problem or a proof that it is NP-complete. It is one of a relatively small number of naturally occurring languages in NP that haven't been placed in either category.

$\overline{ISO}$ is not known to be in NP because we do not know how to provide certificates that graphs are not isomorphic. However when two graphs are not isomorphic, a Prover can convince a Verifier of this fact.

Definition 10.23. The *Verifier* is a function

$$V : (w, r, m_1\# \cdots \# m_i) \mapsto x \in \{m_{i+1}, \mathit{accept}, \mathit{reject}\},$$

where w is the input string, r some random string V is allowed access to (equivalent to coin-flip steps), $m_1\# \cdots \# m_i$ the exchange of message up to the present point, and m_{i+1} the next message to be sent.
The *Prover* is a function $P : (w, m_1\# \cdots \# m_i) \mapsto m_{i+1}$ defined similarly.
Write $(V \leftrightarrow P)(w, r) = \mathit{accept}$ if a message sequence $m_1, \ldots, m_k$ exists for some k whereby

1. for $0 \le i < k$, where i is even, $V(w, r, m_1\# \cdots \# m_i) = m_{i+1}$;
2. for $0 < i < k$, where i is odd, $P(w, m_1\# \cdots \# m_i) = m_{i+1}$; and
3. the final message is *accept*.

To define IP, assume that the lengths of the Verifier's random input and each of the messages exchanged are $p(n)$, and the total number of messages is at most $p(n)$, where p is determined according to V. Define

$$\Pr(V \leftrightarrow P \text{ accepts } w) = \Pr((V \leftrightarrow P)(w, r) = \mathit{accept}),$$

where r is a randomly selected string of length $p(n)$.

Definition 10.24. Say that language A is in **IP** if some polynomial time computable function V exists such that for some ("honest") function P and for every ("crooked") function $\widetilde{P}$ and for every string w,

1. $w \in A$ implies $\Pr(V \leftrightarrow P \text{ accepts } w) \geq 2/3$, and
2. $w \notin A$ implies $\Pr\big(V \leftrightarrow \widetilde{P} \text{ accepts } w\big) \leq 1/3$.

The basic motivation of this model is to give a probabilistic version of NP, according to the "polynomial verifier" notion. Note also that 10.4 applies here as well.

Example 10.25. $\overline{ISO}$ is in IP. C.f. 10.22. If the Prover states that G_1 and G_2 are not isomorphic, the Verifier randomly selects either G_1 or G_2 and then randomly reorders its nodes to obtain a graph H, and sends H to the Prover. The Prover must respond by declaring whether G_1 or G_2 was the source of H. The Prover might guess if G_1 and G_2 *are* indeed isomorphic, but the probability of two consecutive successful guesses is $1/4 \leq 1/3$.

We show the following theorem in steps.

Theorem 10.26. IP = PSPACE.

Lemma 10.27. IP $\subseteq$ PSPACE.

Let $\#SAT = \{\langle \phi, k \rangle \mid \phi \text{ is a CNF with exactly } k \text{ satisfying assignments}\}$.

Lemma 10.28. $\#SAT \in$ IP.

This implies that coNP $\subseteq$ IP.

Lemma 10.29. PSPACE $\subseteq$ IP.

Exercises and Problems

Exercise 15. * 10.5

Exercise 16. * 10.8

Exercise 19. Show that if NP $\subseteq$ BPP, then NP = RP.

to do

Exercise 20. Define a *ZPP-machine* to be a probabilistic TM that is permitted three types of output on each of its branches: *accept*, *reject*, and ?. A ZPP-machine M decides a language A if M outputs the correct answer on every input string w (*accept* if $w \in A$ and *reject* if $w \notin A$) with probability at least $2/3$, and M never outputs the wrong answer. On every input, M may output ? with probability at most $1/3$. Furthermore, the average running time over all branches of M on w must be bounded by a polynomial in the length of w. Show that RP $\cap$ coRP = ZPP, where ZPP is the collection of languages that are recognized by ZPP-machines.

to do

Exercise 21. 10.15