Chapter 2

P and NP

2.1 Introduction

The $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ question is a fundamental open problem of theoretical computer science, and indeed of mathematics[1]. It asks whether all problems solvable in polytime on a nondeterministic TM can also be solved in polytime on a deterministic TM. Stephen Cook ([Coo71]) and Leonid Levin formulated the problem independently in 1971.

Standard complexity textbooks provide ample material on P and NP; we limit ourselves to a few observations. The class P consists of those languages that can be decided in polynomial time (*polytime*) in the length of the input, i.e., $\mathsf{P} = \cup_{k=1}^{\infty} \mathsf{TIME}(n^k)$, and the class NP consists of those languages that can be decided in polytime on a nondeterministic TM, i.e., $\mathsf{NP} = \cup_{k=1}^{\infty} \mathsf{NTIME}(n^k)$.

Sometimes it is convenient to use an alternative definition of NP, in terms of proof systems. We say that a language L has a *proof system* if there exists a polytime binary predicate $R(x, y)$, such that $x \in L \iff \exists y R(x, y)$ (here the "y" is called a *proof* or *certificate*). The prototypical example of such a

[1] In the year 2000, The Clay Mathematics Institute selected seven Prize Problems (`http://www.claymath.org/millennium/`) to mark the 100th anniversary of David Hilbert's lecture at the second International Congress of Mathematicians. One of these problems is the $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ question; see [Coo00] for Cook's recent manuscript prepared for the Clay Mathematics Institute. A nice introduction and history of the seven problems is given in [Dev05].

language is

$$\textsc{Taut} = \{\langle\phi\rangle |\ \phi \text{ is a tautology }\},$$

where ϕ is a Boolean formula constructed from $\{\wedge, \vee, \neg, \mathsf{T}, \mathsf{F}\}$ and Boolean variables, and $\langle\phi\rangle$ is some reasonable encoding of ϕ as a string.

In the context of TAUT, the certificate y is the encoding of a derivation of ϕ; this derivation could be, for example, a truth table. The predicate $R(\langle\phi\rangle, y)$ checks that y is indeed the encoding of a valid derivation of ϕ. What is important is that given a formula ϕ and an alleged derivation, it is possible to check in polytime that the derivation is indeed a correct derivation of ϕ.

A proof system is *polybounded* (polynomially bounded) if $|y|$ can be bounded by some polynomial in $|x|$. The prototypical example of a language with a polybounded proof system is

$$\textsc{Sat} = \{\langle\phi\rangle |\ \phi \text{ is satisfiable }\}.$$

Here the y could simply be the encoding of a truth assignment; a trivial encoding would work: if $y = a_1 a_2 \ldots a_n \in \{0,1\}^*$, then variable x_i would be assigned the truth value T if $a_i = 1$, and F if $a_i = 0$. Note that the certificate y is short in this case (in fact, of length bounded by $|\langle\phi\rangle|$).

Let co-NP be the class of languages whose complements are in NP, i.e., co-NP $= \{L : \overline{L} \in \mathsf{NP}\}$. Let UNSAT be the language of unsatisfiable Boolean formulas. Then UNSAT $= \overline{\textsc{Sat}}$ (see footnote[2]). By the same reasoning TAUT $\in$ co-NP.

NP $\stackrel{?}{=}$ co-NP is the questions of whether TAUT has polybounded proof system (we shall study this question in more detail in chapter 6).

Exercise 2.1.1 *Show that if $L \in \mathsf{NP}$ and $L' \in \mathsf{P}$, then $L \cap L' \in \mathsf{NP}$.*

Lemma 2.1.2 *A language L is in NP iff there exists a polytime binary predicate R, and a polynomial p, such that $x \in L \iff (\exists y \leq p(|x|))R(x,y)$. In short, a language is in NP iff it has a polybounded proof system[3].*

[2]This is not quite true, since $\overline{\textsc{Sat}}$, understood to be $\{0,1\}^* - \textsc{Sat}$, contains encodings of unsatisfiable formulas as well as "junk" strings, i.e., strings that do not encode a formula. Let WFF (well-formed formula) be the set of strings that encode formulas. When we talk of the complement of SAT we really mean $\overline{\textsc{Sat}} \cap$ WFF. Clearly, it can be checked in polytime whether $\langle\phi\rangle \in$ WFF. (See exercise 2.1.1.)

[3]$(\exists y \leq p(|x|))R(x,y)$ denotes $\exists y(|y| \leq p(|x|) \wedge R(x,y))$, and $(\forall y \leq p(|x|))R(x,y)$ denotes $\forall y(|y| \leq p(|x|) \supset R(x,y))$.

PROOF: [$\Longrightarrow$] If $L \in \mathsf{NP}$, then there exists a nondeterministic polytime TM M deciding L. Let $R(x, y)$ be the predicate that checks whether y is (an encoding of) an accepting computation of M on x. [$\Longleftarrow$] Let M be a nondeterministic polytime TM which on input x "guesses" a y, and checks $R(x, y)$. When we say "guesses," we mean that the machine examines all the y's, each y on a different branch of the computation. □

Lemma 2.1.3 *Suppose that the language L has a polybounded proof system V, and suppose that $\mathsf{P} = \mathsf{NP}$. Then there exists a polytime function f, such that for every $x \in L$, $V(x, f(x))$ holds, and for $x \notin L$, $f(x) =$ "no". In other words, if $x \in L$, then $f(x)$ outputs in polytime (in $|x|$) a proof of membership, and if $x \notin L$, then f says so.*

PROOF: Let "$\cdot$" denote the *concatenation* of strings, that is, given two strings $x, y \in \Sigma^*$, $x = x_1 x_2 \dots x_n$ and $y = y_1 y_2 \dots y_m$, $x \cdot y = x_1 x_2 \dots x_n y_1 y_2 \dots y_m$. In practice we often omit the dot and write xy instead of $x \cdot y$.

Let Q be a binary predicate such that

$$Q(x, y) \stackrel{\text{dfn}}{\Longleftrightarrow} (\exists z \leq (p(|x|) - |y|))V(x, y \cdot z). \tag{2.1}$$

Note that the language $\{\langle x, y\rangle | Q(x, y)\}$ is in NP, and so by assumption it is in P. The algorithm computing $f(x)$ (recall that ε denotes the empty string):

On input x:
1. if $\neg Q(x, \varepsilon)$ then return "no"
2. while $\neg V(x, p)$
3. if $Q(x, p \cdot 0)$ then $p := p \cdot 0$
4. else $p := p \cdot 1$

The algorithm tries out consecutive bits of the proof and uses the decider for Q to check if it is following the right path. As a single check can be done in polynomial time and the length of the proof is polynomial in the length of x, f can be computed in polytime. □

Exercise 2.1.4 *Is the "$-|y|$" in the right-hand side of the definition given by (2.1) really necessary?*

2.2 Reductions and completeness

In the context of NP, it is customary to use *polytime many-one reductions*, denoted $\leq_{\mathsf{P}}^{m}$. But standard NP-hardness reductions can be usually carried out in L (logspace), so instead we use *logspace many-one reductions*[4], denoted $\leq_{\mathsf{L}}^{m}$. When we write "$\leq$" we mean "$\leq_{\mathsf{L}}^{m}$". Formally, $L_1 \leq L_2$ iff there exists a logspace function f such that $x \in L_1 \iff f(x) \in L_2$.

Exercise 2.2.1 *Show that logspace reductions are transitive, i.e., if $A \leq B$ and $B \leq C$, then $A \leq C$. Note that this requires a precise definition of what it means for a function to be computed in logspace (see the first paragraph of section 1.2).*

A language L is *$\mathcal{C}$-hard*, for some complexity class $\mathcal{C}$, if for every language $L' \in \mathcal{C}$ it is the case that $L' \leq L$. A language is *$\mathcal{C}$-complete* if it is $\mathcal{C}$-hard, and also in $\mathcal{C}$.

The next theorem introduces the notion of circuits which we are going to cover in chapter 5. To refresh the definition of circuits see section 5.1. The language CircuitValue, defined as the set of pairs $\langle C, x\rangle$ such that x is an input that satisfies the circuit C, is complete for the class P.

Theorem 2.2.2 CircuitValue *remains* P*-complete with the following two restrictions: (1) all the gates are OR's and AND's (i.e., C is monotone), and (2) all the gates are arranged in alternating layers of AND's and OR's.*

Proof: For part (1), on input w, $|w| = n$, we can build an $n^k \times n^k$ *computation tableau*, which represents the history of the computation of a machine on a given input. The first row of the tableau is just the initial configuration, and then each row follows from the previous by a transition. Since a computation need not take exactly n^k many steps, but only at most these many, we make the convention that once a halting configuration is reached, it is repeated to fill exactly n^k many rows of the tableau.

It should be clear that for a given n, we can construct a circuit which on input w ($|w| = n$) computes each entry of the tableau (representing states and tape alphabet symbols with Boolean variables set to 0 or 1; for example, x_{ijs} could say that (i,j)-th entry of the tableau is $s \in Q \cup \Gamma$),

[4]Logspace reductions make more sense in the context of P, since if we allow polytime reductions we can "hide" all the computation in the reduction, in effect making every P language (except $\emptyset$ and Σ^*) P-complete.

and in particular this circuit implements the machine's transition function to compute each row of the tableau correctly for the given input w, and at the end, the circuit outputs 1 if the last row of the tableau represents an accepting configuration, and 0 otherwise[5].

By de Morgan laws, all the negations in such a circuit can be pushed to the input level (replicating gates on the way down if necessary), and then change the input to consist of two copies of w, where the second copy of w is inverted, i.e., every 1 becomes a 0, and every 0 becomes a 1. Then connect x_i's to the original input w, and $\bar{x}_i$ to the inverted w.

For part (2) see lemma 5.1.5. □

Exercise 2.2.3 *Show that the "gate replication" in the above proof can be carried out with only a polynomial blow-up in size. In fact, what is the increase in size (when this is done with thrift)?*

Theorem 2.2.4 *If B is* NP*-complete, and $B \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$.*

Exercise 2.2.5 *Prove theorem 2.2.4.*

Theorem 2.2.6 *If B is* NP*-complete, and $B \leq C$, for $C \in \mathsf{NP}$, then C is* NP*-complete.*

Exercise 2.2.7 *Prove theorem 2.2.6.*

Theorem 2.2.8 (Cook-Levin) SAT *is* NP*-complete.*

PROOF: We follow [Sip06b, theorem 7.37]. If $A \in \mathsf{NP}$, then $A = L(N)$, where N is a nondeterministic TM that runs in time n^k. Each configuration of N can be described with a string of length n^k (as in n^k-many steps the machine cannot write on more than n^k-many squares of the tape). Any branch of the computation has length at most n^k. So any branch can be described with an $n^k \times n^k$ tableau (same idea as in the proof of theorem 2.2.2). To determine whether N accepts w, we must determine if an accepting tableau exists.

We construct a reduction $f(\langle w \rangle) = \langle \phi_w \rangle$ such that $w \in A \iff \phi_w$ is satisfiable. The variables are x_{ijs}, where x_{ijs} is true if position (i, j) in the tableau contains the symbol $s \in Q \cup \Gamma \cup \{\#\}$. Then, let ϕ_w be

$$\phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}},$$

[5]It is not surprising that a computation can be efficiently simulated with circuits; after all, computers are built from circuits.

where the first formula, ϕ_{cell}, ensures that exactly one symbol is assigned to each cell, the second formula, ϕ_{start}, ensures that the first row of the tableau is the initial configuration, i.e.,

$$(q_0, \triangleright, w_1 w_2 \dots w_n \sqcup \sqcup \dots \sqcup)$$

The third formula, ϕ_{accept}, ensures that the last row is an accepting row (once we get an accepting row, we repeat it until we reach the n^k-th row). For example, ϕ_{accept} could be given as follows:

$$\bigvee_{1 \le i,j \le n^k} x_{ijq_{\text{accept}}}.$$

Finally, ϕ_{move} ensures that each row follows from the previous by a legal transition of N (or, in the case that the previous row was accepting, the current row is its copy). This can be implemented by observing that each row is exactly like the previous one, except in the six squares surrounding the state.

$x_{i(j-1)s_1}$	x_{ijs_2}	$x_{i(j+1)s_3}$
$x_{(i+1)(j-1)s_4}$	$x_{(i+1)js_5}$	$x_{(i+1)(j+1)s_6}$

Let $W_{ij}(s_1, \dots, s_6)$ be the conjunction of the entries of the above table. So, ϕ_{move} is

$$\bigwedge_{i,j} \bigvee_{\substack{s_1,\dots,s_6 \\ \text{legal window}}} W_{i,j}(s_1, \dots, s_6). \tag{2.2}$$

Claim 2.2.9 *ϕ_w can be constructed in logspace in $|w|$.*

Proving this claim in detail is the hard part of the proof. But, to be convinced of it, note that in logspace we can maintain constantly many pointers and counters of logarithmic length (and hence of sufficiently many bits to be able to index the tableau and ϕ_w). This is sufficient to construct ϕ_w because its structure is straightforward[6]. □

Corollary 2.2.10 3SAT *is* NP*-complete.*

PROOF: We start by showing that the formula ϕ_w (given in the proof of theorem 2.2.8) can be given as a CNF formula (conjunctive normal form formula) without a significant increase in size.

[6]Carrying the proof of this claim in detail once is what makes a complexity *Pollywog* into a complexity *Shellback*.

Exercise 2.2.11 *Show that ϕ_w can be efficiently transformed (in logspace in $|w|$) into a CNF formula.*

Then, we show how to translate a general CNF formula into a 3CNF formula. For example, if a clause has 1 or 2 literals, then we pad it with dummy variables.

Exercise 2.2.12 *Show how to pad clauses with 1 or 2 literals, while preserving satisfiability.*

Finally, for $n > 3$ literals, send $(a_1 \vee a_2 \vee \ldots \vee a_n)$ to

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z}_1 \vee a_3 \vee z_2) \wedge (\overline{z}_2 \vee a_4 \vee z_3) \wedge \ldots \wedge (\overline{z}_{n-3} \vee a_{n-1} \vee a_n),$$

where the z_i's are new variables. The claim is that the new formula ϕ'_w is satisfiable iff ϕ_w is satisfiable (note that the two formulas are not logically equivalent—ϕ'_w has more variables). □

Using reductions from SAT and 3SAT we can now start showing that very many of NP problems are in fact NP-complete; see the Appendix, section 8.1, for many examples. A comprehensive list of NP-complete problems and reductions can be found in the classic book [GJ79]. Some unexpected problems have also been shown to be NP-complete, for example MINESWEEPER[7].

Exercise 2.2.13 *Suppose that a Boolean expression on n variables has fewer than n^k clauses, each with at least $k \log(n)$ distinct variables. Show that it must have a satisfying truth assignment, and give a polytime algorithm for finding such an assignment.*

Let M be a nondeterministic TM, and let $\#\text{accept}_M(x)$ be the number of accepting paths of M on input x. We define the functional class #P to be the class of functions f for which there is a polytime nondeterministic TM M such that $f(x) = \#\text{accept}_M(x)$. The majority of the standard (logspace) many-one reductions for classical NP-complete problems are *parsimonious* (they preserve the number of solutions). Hence they can be used to show that the counting versions of these problems are complete for #P.

[7] http://www.claymath.org/Popular_Lectures/Minesweeper/

2.3 Self-reducibility of Sat

SAT is *self-reducible* in the following sense: if α is a Boolean formula and v is a variable, then

$$\alpha \in \text{SAT} \iff (\alpha[\mathsf{T}/v] \in \text{SAT} \vee \alpha[\mathsf{F}/v] \in \text{SAT})$$

where $\alpha[\mathsf{T}/v]$ is α with the variable v replaced troughout by T, and then the formula is simplified.

Theorem 2.3.1 *Consider languages over $\Sigma = \{0,1\}$. Suppose that we have a set $T \subseteq \{1\}^*$ (i.e., T is a set consisting of strings of 1s); such a set is called a* tally *set. If T is* NP*-hard, then* P = NP*.*

PROOF: Assume that T is NP-hard, and so SAT $\leq T$, and let g be the function implementing a (logspace) reduction from SAT to T.

We give a polytime algorithm for SAT using this g. The algorithm works in stages: at stage 0, let $C_0 = \{\alpha\}$, where α is the input formula. At stage $(i+1)$, $C_i = \{\alpha_1, \ldots, \alpha_n\}$ (where C_i is the result of the previous stage, i.e., stage i), and we create

$$C' = \{\alpha_1[\mathsf{T}/v_{i+1}], \alpha_1[\mathsf{F}/v_{i+1}], \ldots, \alpha_n[\mathsf{T}/v_{i+1}], \alpha_n[\mathsf{F}/v_{i+1}]\}.$$

Thus, C' contains all the formulas of C_i with v_{i+1} set to T and to F (so $|C'| = 2 \cdot |C_i|$), and simplified.

We now prune C' as follows: we compute $g(\beta)$ for every $\beta \in C'$. Whenever we get two formulas that map to the same string in $\{1\}^*$, we keep only one of them. (If $g(\beta)$ maps to some string not in $\{1\}^*$, we simply prune β.) We let C_{i+1} be the result of this pruning. At the end, when no variables are left, $C_k \subseteq \{\mathsf{T}, \mathsf{F}\}$, and we answer "yes" (i.e., "yes, α is satisfiable") iff $\mathsf{T} \in C_k$.

This algorithm is polytime, because the pruning ensures that the C_i's are always polysize (there are polynomially many unary strings of polynomial size). It is correct since g is assumed to be a correct reduction. □

Exercise 2.3.2 *Say that a language is* sparse *if there is a polynomial $p(n)$ such that the number of strings of a given length n in the language is at most $p(n)$. Formally, L is sparse if there exists a polynomial $p(n)$ such that $|\{x \in L : |x| = n\}| \leq p(n)$. Show that if there is a sparse language L which is hard for* co-NP *with respect to logspace many-one reductions (i.e., $\leq^m_{\mathsf{L}}$), then in fact* P = NP*.*

Theorem 2.3.3 (Mahaney) *If there are sparse languages which are hard for* NP*, then* P = NP

See [HO02, §1.1.2] for a proof of Mahaney's theorem. The *Berman-Hartmanis Isomorphism Conjecture* (IC) states that all NP-complete languages (here it is the $\leq^m_{\mathsf{P}}$ notion of completeness) are polytime isomorphic (inter-reducible with polytime many-one reductions that are bijections, with polytime inverses). In other words, there is only one NP-complete set in many guises ([HO02, pp. 26 & 282]). A line of attack on the IC was to show the existence of a sparse NP-complete language, since no dense NP-complete language (such as SAT) can be polytime isomorphic to a sparse language, thereby showing IC to be false. Mahaney's theorem shows the futility of this line of attack: if sparse NP-complete languages exist, then P = NP, in which case IC fails trivially anyways.

Exercise 2.3.4 *Observe that if the IC is true, then* P ≠ NP*.*

The next theorem also uses the idea of self-reducibility of SAT, and it introduces the notion of the Polytime Hierarchy (PH) which we are going to cover in more detail in section 4.3. Define Σ^p_i to be the class of languages L for which there is a polytime relation R such that

$$x \in L \iff \exists y_1 \forall y_2 \ldots Q y_i R(x, y_1, y_2, \ldots, y_i)$$

where $R(x, y_1, y_2, \ldots, y_i)$ is decidable in polytime in $|x|$. Let Π^p_i be defined analogously, but starting with a $\forall$ quantifier. Then, $\mathsf{PH} = \cup_i \Sigma^p_i = \cup_i \Pi^p_i$.

Theorem 2.3.5 (Karp-Lipton) *If all languages in* NP *have polysize circuits, that is,* NP ⊆ P/poly*, then* PH *collapses to its second level, that is,* PH = Σ^p_2*.*

PROOF: It is enough to show that if NP ⊆ P/poly, then $\Pi^p_2 \subseteq \Sigma^p_2$ (see footnote[8]). To show that NP ⊆ P/poly ⇒ $\Pi^p_2 \subseteq \Sigma^p_2$ argue as follows: assume L is in Π^p_2, so $L = \{x | \forall y \exists z R(x, y, z)\}$ (where $|y|, |z|$ are implicitly bounded by a polynomial in $|x|$). Consider the language $L' = \{\langle x, y\rangle | \exists z R(x, y, z)\}$, which is in NP by lemma 2.1.2. Thus, there exists a polytime function f such that $L = \{x | \forall y\ f(\langle x, y\rangle) \in \text{SAT}\}$.

Note that $x \in L \iff \exists T \forall y[\ T(y)$ satisfies $f(x, y)\ ]$, where $T(y)$ is a truth assignment that satisfies $f(x, y)$ for the given y (if one exists), is a

[8] It follows more or less from definition that if $\Pi^p_2 \subseteq \Sigma^p_2$ then in fact the entire hierarchy collapses to Σ^p_2, i.e., PH = Σ^p_2, but see section 4.3 for the necessary background.

way of stating what we want with the right alternation of quantifiers. The problem is that the naive way of representing T would be as a string of truth assignments for each y, and there are exponentially many y's (in $|x|$), so this does not work.

By $\mathsf{NP} \subseteq \mathsf{P/poly}$ and the self-reducibility of SAT we know that for all n, there exists a C_n such that for all ϕ, $|\phi| = n$, C_n outputs a satisfying assignment to ϕ (if one exists).

Now note that $|y| \leq p(|x|)$, for some polynomial p, so for any given x_0, $|f(\langle x_0, y\rangle)| \leq q = q(|x_0|)$, where q is a polynomial that depends on p and the polynomial bounding f. So, $x \in L$ iff $\exists C = \langle C_0 C_1 \dots C_q\rangle \forall y$ $[C_{|f(\langle x,y\rangle)|}(f(\langle x, y\rangle))$ satisfies $f(\langle x, y\rangle)]$, where $|C|$ and $|y|$ can be bounded by a polynomial in $|x|$, and the predicate decided in time polynomial in $|x|$. Hence L is in Σ_2^p. □

2.4 Padding argument

Define the *padding function* as $\mathrm{pad} : \Sigma^* \times \mathbb{N} \longrightarrow (\Sigma \cup \{\#\})^*$ where $\mathrm{pad}(s, l) = s\#^j$ for $j = \max(0, l - |s|)$. For any language L and function $f : \mathbb{N} \longrightarrow \mathbb{N}$ let

$$\mathrm{pad}(L, f(n)) = \{\mathrm{pad}(s, f(|s|))| \text{ where } s \in L \}.$$

Note, for example, that if $L \in \mathsf{TIME}(n^6)$, then $\mathrm{pad}(L, n^2) \in \mathsf{TIME}(n^3)$. To see this, use essentially the same machine that decides L in time n^6 to decide $\mathrm{pad}(L, n^2)$ in time n^3 by making it "ignore" the trailing junk[9] (note that $|\mathrm{pad}(x, |x|^2)| = |x|^2$, and x is decided in time $|x|^6 = (|x|^2)^3$).

This innocuous trick to seemingly reduce the computational time of TMs has some interesting applications.

Theorem 2.4.1 *If* $\mathsf{NEXPTIME} \neq \mathsf{EXPTIME}$*, then* $\mathsf{P} \neq \mathsf{NP}$*.*

PROOF: Show the contrapositive: assume $\mathsf{P} = \mathsf{NP}$ and L is in $\mathsf{NEXPTIME}$. Then $\mathrm{pad}(L, 2^{n^k})$ is in NP, and so it is in P, and hence L is $\mathsf{EXPTIME}$. □

Theorem 2.4.2 (Ladner) *If* $\mathsf{P} \neq \mathsf{NP}$*, then there is a language in* $\mathsf{NP} - \mathsf{P}$ *which is* not NP*-complete*[10]*.*

[9] Not quite ignore, as it has to check that the input string is of the form $w = \mathrm{pad}(x, |x|^2)$, and reject if not.

[10] We suspect that GRAPHISOMORPHISM $= \{\langle G, G'\rangle : G \cong G'\}$ is an example of such a language. It is easy to see that GRAPHISOMORPHISM is in NP: the certificate is π, a relabelling of the vertices of G such that $\pi(G) = G'$.

PROOF: Consider PADSAT $= \{\text{pad}(\phi, |\phi|^{p(|\phi|)}) |\ \phi \in \text{SAT}\ \}$, where we define the padding function $p(n)$ as follows. First, fix some encoding of TMs, so we have a list of all the TMs under the sun. This can be accomplished with a Universal TM U. Now, let $M_1, M_2, M_3, \ldots$ be this list but where each TM occurs infinitely often.

Let $p(n)$ be the smallest $i < \log\log n$ such that for every $x \in \{0,1\}^{\leq \log(n)}$ the following holds: M_i halts on x within $(|x|^i + i)$ many steps and M_i accepts iff $x \in$ PADSAT. If there is no such i, we let $p(n) = \log\log n$.

The idea for this "strange" definition of $p(n)$ is to have a function that grows fast enough so that PADSAT is not NP-complete, but slowly enough to ensure that is it is not in P. In what follows we show that this is indeed the case.

Note that the padding function p is well defined. This is an issue, because p is defined in terms of itself, i.e., recursively, but note that to compute $p(n)$ we only need to consider the values of $p(k)$ for $k \leq \log n$.

Claim 2.4.3 PADSAT *is not in* P.

PROOF: Suppose that it is, and it is decided by some M running in time $(n^k + k)$. There is an $i > k$ such that $M = M_i$. Therefore, by the definition of p, for all $n > 2^{2^i}$, $p(n) \leq i$. But this means that for $n > 2^{2^i}$, PADSAT is just SAT padded with # to be of length n^i. So if PADSAT were in P, so would SAT: given ϕ, $|\phi| = n > 2^{2^i}$, check if $\phi\#^{n^i - n} \in$ PADSAT. This contradicts the assumption that P $\neq$ NP. □

Claim 2.4.4 $\lim_{n\to\infty} p(n) = \infty$.

PROOF: Since PADSAT $\notin$ P (by claim 2.4.3), for each i we know that there exists an x such that given time $(|x|^i + i)$, M_i gives the incorrect answer to the question $x \stackrel{?}{\in}$ PADSAT. Then, we know (from the definition of p) that for every $n > 2^{|x|}$, $p(n) \neq i$. So for every i there are only finitely many n's such that $p(n) = i$. □

Claim 2.4.5 PADSAT *is not* NP*-complete.*

PROOF: As $p(n)$ tends to infinity with n (by claim 2.4.4), the padding is of super-polynomial size. Suppose PADSAT is NP-complete, so SAT $\leq$ PADSAT. This reduction takes ψ (with $|\psi| = n$), to $\text{pad}(\phi, |\phi|^{p(|\phi|)})$, such that $|\text{pad}(\phi, |\phi|^{p(|\phi|)})|$ is $O(n^k)$ for some fixed k. Therefore, $|\phi|^{p(|\phi|)}$ is $O(n^k)$.

But this means that $|\phi|$ must be $o(n)$ (i.e., we can find a fraction $\frac{p}{q}$ such that $p, q \in \mathbb{N}$, and $p < q$, and $|\phi| < n^{\frac{p}{q}}$). Using this fact, we can now design a polytime algorithm for SAT which applies this procedure repeatedly (given a constant a, we want an i such that $n^{\left(\frac{p}{q}\right)^i} < a$; this i is $O(\log\log(n))$), each time obtaining a smaller ϕ, until it is of constant size, and can be solved by brute force. It follows that $\mathsf{P} = \mathsf{NP}$; contradiction. □ □

2.5 Answers to selected exercises

Exercise 2.1.4. No.

Exercise 2.2.1. The difficulty is that while $f(x)$ can be computed in logspace, $|f(x)|$ may still be of polynomial length in $|x|$, and it is perfectly legal to have a long $y = f(x)$ on the output tape. So, when we are computing the composition of two logspace functions f, g, once we have computed $f(x)$, we cannot put it on the worktape, and comput $g(f(x)$ directly. To fix this, we recompute the i-th bit of $f(x)$ each time it is requires in the computation of $g(f(x))$.

Exercise 2.2.11. The formula ϕ_w is almost in CNF form, we only have to iron out a few wrinkles. Note that ϕ_w is composed of four parts:

$$\phi_{\text{cell}}, \phi_{\text{start}}, \phi_{\text{move}}, \phi_{\text{accept}}.$$

We have to make sure that each part is either a disjunction, a conjunction, or a conjunction of disjunctions. The formula ϕ_{accept} is just a disjunction, so it is already in the right form. The formula ϕ_{start} can be given as a conjunction of the variables x_{1js} asserting that the j-th symbol is the j-th symbol of the initial configuration. The formula ϕ_{cell} asserts that each cell contains exactly once symbol, so it is of the form

$$\bigwedge_{i,j} \bigvee_{s} \left(x_{ijs} \wedge \bigwedge_{s' \neq s} \neg x_{ijs'} \right).$$

Using distributivity of $\wedge$ and $\vee$, we can push the $\bigvee_s$ inside, incurring a small increase in size, because the number of symbols s is a constant. This is a good place to make an observation: suppose we have a DNF formula of the form:

$$\overbrace{\underbrace{(\ldots \wedge \ldots)}_{n} \vee (\ldots \wedge \ldots) \vee \ldots \vee (\ldots \wedge \ldots)}^{m},$$

i.e., each clause has n literals, and there are m clauses. Then, using distributivity, this can be transformed into CNF so that each clause has m literals, and there are n^m many clauses. Of course, the same can be done transforming CNF into DNF (we shall use this fact in section 5.3.1).

Thus, we can put ϕ_{cell} into CNF effectively, since the number of symbols s is a constant, so from s clauses, by the above observations, we get s^s many clauses—still a constant.

Finally, we deal with ϕ_{move} (see (2.2)). Again, there are constantly many legal windows, so we can push the $\bigvee$ all the way inside incurring little increase in size.

Exercise 2.2.12. $(l_1 \vee l_2) \mapsto (l_1 \vee l_2 \vee x) \wedge (l_1 \vee l_2 \vee \bar{x})$. For the case (l_1) introduce two new variables x, y and four clauses, each with l_1 and an x-literal and a y-literal, with the four possible arrangements of negations for x and y.

Exercise 2.2.13. If a clause has at least $k \log(n)$ distinct variables, then the number of truth assignments which falsify it is at most than $2^{n-k\log(n)} = \frac{2^n}{n^k}$. Then the number of truth assignments which falsify at least one clause is less than $n^k \frac{2^n}{n^k} = 2^n$, and therefore there must be at least one truth assignment which satisfies the whole formula. Consider the original set of clauses ϕ and set the first variable to either 0 or 1, thereby getting two sets of clauses, ϕ_0 and ϕ_1. If we denote by $f(\psi)$ the number of truth assignments falsifying ψ, we can easily see that $f(\phi) = f(\phi_0) + f(\phi_1)$. But we know that $f(\phi) < 2^n$ and so $f(\phi_i) < 2^{n-1}$, for $i = 0$ or $i = 1$. It turns out, that if we select the i which satisfies the most clauses (and choose i arbitrarily if there is a tie) we can get a satisfying assignment (see [Pap94, theorem 13.2]). Repeating the above step n times we will get a total truth assignment satisfying the original formula.

Exercise 2.3.4. Suppose the IC is true, and so is $\mathsf{P} = \mathsf{NP}$. Then every language in P (other than $\emptyset$ and Σ^*) is NP-complete (with respect to polytime reductions), and there are finite languages in P.

2.6 Notes

Exercise 2.2.13 is [Pap94, exr. 11.5.23]. Possible references for theorem 2.3.1 are [Pap94, Theorem 14.3, pg. 337] and [HO02, Theorem 1.2, pg. 3]. Theorem 2.3.5 is based on [KL80]. For a proof of theorem 2.3.3, Mahaney's theorem, see [HO02, §1.1.2]; a new technique is needed for this proof, the so

called *left set* technique.