Chapter 9

The Polynomial Hierarchy

9.1. Complementary Classes

Definition 9.1: $\mathcal{NP} = \bigcup_{c>0} \text{NTIME}(n^c)$.

Definition 9.2: $\mathcal{PSPACE} = \bigcup_{c>0} \text{DSPACE}(n^c) = \bigcup_{c>0} \text{NSPACE}(n^c) = \bigcup_{c>0} \text{ATIME}(n^c)$.

The last two equalities follow from Theorems 4.11 and 4.14.

We then have the following chain of containments:

$$\mathcal{L} \subseteq \mathcal{NL} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PSPACE}.$$

We know that $\mathcal{NL} \neq \mathcal{PSPACE}$ from the nondeterministic space hierarchy theorem, but we don't know if any of the other containments is proper. This section is devoted to the world between $\mathcal{NP}$ and $\mathcal{PSPACE}$.

Open Problem 9.3: Does $\mathcal{P} = \mathcal{PSPACE}$?

Definition 9.4: If $\mathcal{C}$ is a set of languages over some alphabet Σ , (i.e., $\mathcal{C} \subseteq 2^{\Sigma^*}$), then

$$co\mathcal{C} = \{\Sigma^* - L \mid L \in \mathcal{C}\}.$$

$\mathcal{C}$ is closed under complementation if and only if $\mathcal{C} = co\mathcal{C}$. We already have some results concerning complementary classes, for instance, $\mathcal{P} = co\mathcal{P}$, $\mathcal{NL} = co\mathcal{NL}$ (Theorem 4.23), and $\mathcal{PSPACE} = co\mathcal{PSPACE}$. However, the question for $\mathcal{NP}$ is unresolved:

Open Problem 9.5: Is $\mathcal{NP} = co\mathcal{NP}$?

Notice that this is related, but not equivalent, to Open Problem 4.25 on Page 34.

Proposition 9.6: If $\mathcal{NP} \neq co\mathcal{NP}$ then $\mathcal{P} \neq \mathcal{NP}$.

Proof: $\mathcal{P} = co\mathcal{P}$. □

Aside from the implications of the previous proposition, there are some natural languages in $co\mathcal{NP}$ that justify the study of this class. One such language is the set of propositional tautologies. The set of well formed formulas can be partitioned into those that are tautologies and those that are falsifiable; the latter can be checked in $\mathcal{NP}$ by guessing a truth assignment and checking that it makes the formula false. (The complement of the set of tautologies also includes all strings that are not well formed formulas, but these are easily recognized in polynomial time.) The question of whether $\mathcal{NP} = co\mathcal{NP}$ is the same as asking whether there is a proof system in which every tautology has a polynomial length proof. (See Cook and Reckhow [5] for more information.)

A useful equivalent characterization of $co\mathcal{NP}$ follows from DeMorgan's laws:

Proposition 9.7: $co\mathcal{NP}$ is the set of languages accepted by polynomial time bounded alternating Turing machines that have no existential states.

For instance, to show that the set of tautologies is in $co\mathcal{NP}$, an alternating Turing machine can universally choose all possible truth assignments and verify that each satisfies the input formula.

Another example of a natural problem in $co\mathcal{NP}$ is the set of prime numbers. In fact, this set is in $\mathcal{NP} \cap co\mathcal{NP}$. The fact that it is in $co\mathcal{NP}$ is easy: to determine that x is not prime, nondeterministically guess a proper factor and divide. The proof that the set of primes is in $\mathcal{NP}$ is due to Pratt [36] and is more difficult.

One might wonder if it is possible that $\mathcal{NP} \subseteq co\mathcal{NP}$ without the two classes being equal. The following general proposition shows that this cannot be the case.

Proposition 9.8: If $\mathcal{C} \subseteq co\mathcal{C}$ then $\mathcal{C} = co\mathcal{C}$.

Proof: For any $L \in co\mathcal{C}$, $\Sigma^* - L \in \mathcal{C} \subseteq co\mathcal{C}$. Hence, $L \in \mathcal{C}$. □

9.2. Turing Reducibility

Informally, a language A might be considered to be reducible to a language B whenever there is an efficient algorithm that accepts A, given an efficient subroutine that accepts B. The definition of many-one reducibility given in Definition 5.2 is a very restricted version of this general notion. The more general notion of reducibility is formalized in the definition below of an "oracle Turing machine", where the "oracle" plays the role of the subroutine.

Definition 9.9: An *oracle Turing machine* M is defined as follows: M is a Turing machine with a distinguished query tape, and three distinguished states, $q_?$, q_Y, and q_N. The computation of M depends on some "oracle" language B. Whenever M enters state $q_?$, *in the next step* M will be in state q_Y (q_N) if the nonblank portion of the query tape contains a string that is (respectively, is not) in B. Let M^B denote the machine M with oracle B, and $L(M^B)$ denote the language accepted by this machine.

Definition 9.10: $A \leq_T^{\mathcal{P}} B$ ("A is polynomial time Turing reducible to B") if and only if there is a *deterministic* oracle Turing machine M such that M^B runs in polynomial time and $A = L(M^B)$.

This is the type of reduction Cook [4] used in his original work on complete problems for $\mathcal{NP}$. In his subsequent work, Karp [22] popularized the more restrictive many-one reducibility, which was found to suffice for all the natural reductions among $\mathcal{NP}$-complete problems. For these reasons, polynomial time Turing reducibility is often called "Cook reducibility", and polynomial time many-one reducibility is often called "Karp reducibility". (The latter, denoted $\leq_m^{\mathcal{P}}$, is defined exactly as $\leq_m^{\mathcal{L}}$, except that the reduction is allowed polynomial time.)

A natural question is whether polynomial time Turing reducibility is more powerful than polynomial time many-one reducibility. To shed some light on this question, consider the following example:

Example 9.11: Let SAT ($UNSAT$) be the set of satisfiable (respectively, unsatisfiable) propositional formulas. Then $UNSAT \leq_T^{\mathcal{P}} SAT$ by copying the input formula to the query tape, entering $q_?$, and making q_N the only final state. In contrast, if $UNSAT \leq_m^{\mathcal{P}} SAT$, then $\mathcal{NP} = co\mathcal{NP}$, because $UNSAT$ is $\leq_m^{\mathcal{P}}$–complete for $co\mathcal{NP}$, $\leq_m^{\mathcal{P}}$ is transitive, and $\mathcal{NP}$ is closed under $\leq_m^{\mathcal{P}}$ (all left as exercises).

Open Problem 9.12: Is $\mathcal{NP}$ closed under $\leq_T^{\mathcal{P}}$? If $\mathcal{NP} \neq co\mathcal{NP}$, then Example 9.11 shows that the answer is "no".

Definition 9.13: For any language B, let $\mathcal{P}^B = \{L(M^B) \mid M \text{ is a deterministic oracle Turing machine such that } M^B \text{ runs in polynomial time}\} = \{A \mid A \leq_T^{\mathcal{P}} B\}$.

For any language B, let $\mathcal{NP}^B = \{L(M^B) \mid M \text{ is a nondeterministic oracle Turing machine such that } M^B \text{ runs in polynomial time}\}$.

Definition 9.14: For any set $\mathcal{C}$ of languages, let

$$\mathcal{P}^{\mathcal{C}} = \bigcup_{B \in \mathcal{C}} \mathcal{P}^B, \text{ and}$$

$$\mathcal{NP}^{\mathcal{C}} = \bigcup_{B \in \mathcal{C}} \mathcal{NP}^B.$$

The following examples will help make these ideas more concrete.

Example 9.15: Does $\mathcal{P}^{\mathcal{NP}} = \mathcal{NP}$? $\mathcal{NP} \subseteq \mathcal{P}^{\mathcal{NP}}$ because $B \in P^B$ for any language B: simply copy the input onto the query tape and call the oracle. By negating the oracle's answer, it is also true that $co\mathcal{NP} \subseteq \mathcal{P}^{\mathcal{NP}}$. From this, the answer to the question must be "no", unless $\mathcal{NP}$=$co\mathcal{NP}$. In fact, this question is just a restatement of Open Problem 9.12.

Example 9.16: Does $\mathcal{NP}^{\mathcal{P}} = \mathcal{NP}$? Yes, since we can nondeterministically simulate a specific nondeterministic Turing machine N with an oracle for a language $B \in \mathcal{P}$, pausing in the simulation to answer oracle queries by simulating the oracle and possibly complementing its answer. Since N^B can only make polynomially many queries to B, each on inputs of only polynomial length, and since $B \in \mathcal{P}$, they can all be answered in polynomial time.

9.3. The Polynomial Hierarchy

The polynomial hierarchy was defined by Meyer and Stockmeyer [30], with further properties explored by Stockmeyer [43] and Wrathall [48]. The individual classes of the polynomial hierarchy are the Δ, Σ, and Π sets recursively defined as follows:

Definition 9.17:

$$\Delta_0^P = \Sigma_0^P = \Pi_0^P = \mathcal{P}$$

and, for all nonnegative integers k,

$$\begin{aligned} \Delta_{k+1}^P &= \mathcal{P}^{\Sigma_k^P}, \\ \Sigma_{k+1}^P &= \mathcal{NP}^{\Sigma_k^P}, \\ \Pi_{k+1}^P &= co\Sigma_{k+1}^P, \text{ and} \\ \mathcal{PH} &= \bigcup_{k \geq 0} \Sigma_k^P. \end{aligned}$$

Example 9.18: As concrete examples of these definitions, consider the first few levels of the polynomial hierarchy:

$$\begin{aligned} &\Sigma_1^P = \mathcal{NP}^{\Sigma_0^P} = \mathcal{NP}^{\mathcal{P}} = \mathcal{NP}. \\ &\Pi_1^P = co\Sigma_1^P = co\mathcal{NP}. \\ &\Delta_1^P = \mathcal{P}^{\Sigma_0^P} = \mathcal{P}^{\mathcal{P}} = \mathcal{P}. \\ &\Delta_2^P = \mathcal{P}^{\Sigma_1^P} = \mathcal{P}^{\mathcal{NP}} = \{A \mid [A \leq_T^P \mathcal{P}]\}. \end{aligned}$$

Open Problem 9.19: Does $\mathcal{PH} = \mathcal{PSPACE}$? We will see shortly that $\mathcal{PH} \subseteq \mathcal{PSPACE}$.

Open Problem 9.20: Does $\Sigma_k^P = \Pi_k^P$ or $\Sigma_{k-1}^P = \Sigma_k^P$, for any $k \geq 1$? We will see shortly that a positive answer to either of these questions would have farther-reaching consequences for the polynomial hierarchy. A negative answer to either question would show that $\mathcal{P} \neq \mathcal{PSPACE}$, resolving a major open question.

Proposition 9.21: For all $k \geq 0$,

$$\Sigma_k^P \cup \Pi_k^P \subseteq \Delta_{k+1}^P \subseteq \Sigma_{k+1}^P \cap \Pi_{k+1}^P.$$

(See Figure 9.1.)

Proof:

1. $\Sigma_k^P \subseteq \Delta_{k+1}^P = \mathcal{P}^{\Sigma_k^P}$, because $B \in \mathcal{P}^B$ for any language B.
2. $\Delta_{k+1}^P = \mathcal{P}^{\Sigma_k^P} \subseteq \mathcal{NP}^{\Sigma_k^P} = \Sigma_{k+1}^P$, because $\mathcal{P}^B \subseteq \mathcal{NP}^B$ for any language B.
3. $\Pi_k^P \subseteq \Delta_{k+1}^P \subseteq \Pi_{k+1}^P$ because $\Delta_{k+1}^P = co\Delta_{k+1}^P$.

□

Open Problem 9.22: Are any of the containments in Proposition 9.21 proper?

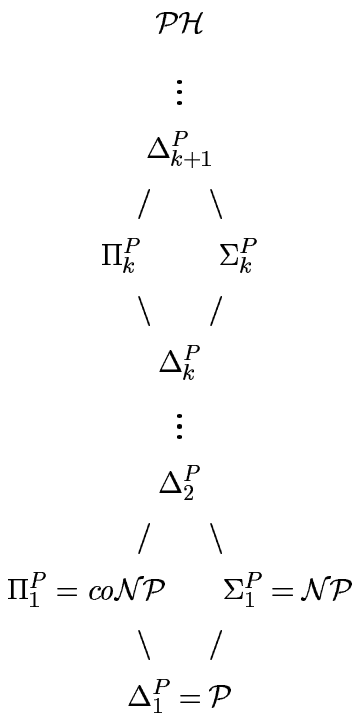

Figure 9.1: The Internal Structure of the Polynomial Hierarchy

9.4. A Sample Problem in $\mathcal{PH}$

In this section the polynomial hierarchy is illustrated further by discussing a sample problem in $\mathcal{PH}$ that does not appear to be in $\mathcal{NP} \cup co\mathcal{NP}$.

Example 9.23 (Papadimitriou [33]): Let *UNIQUE-SAT* be the set of propositional formulas with exactly one satisfying assignment. Although *SAT* is in $\mathcal{NP}$, it does not seem that *UNIQUE-SAT* is in $\mathcal{NP}$. While a nondeterministic Turing machine can find a satisfying assignment for F, it is not clear how it can check that the assignment it found is the only satisfying assignment for F, since this seems to be a universal statement (i.e., *all* other assignments do not satisfy F).

Although not a nondeterministic algorithm, this does give rise to an alternating Turing machine that accepts *UNIQUE-SAT* in polynomial time with a single alternation from existential to universal states. Letting A and B be truth assignments to the variables of F,

$$\textit{UNIQUE-SAT} = \{F \mid (\exists A)(\forall B)((A \text{ satisfies } F) \wedge ((B \neq A) \Rightarrow B \text{ does not satisfy } F))\}.$$

The alternating Turing machine existentially chooses a satisfying truth assignment A and universally verifies that no other truth assignment B satisfies F. By Theorem 9.26 below, this implies that *UNIQUE-SAT* is in Σ_2^P.

In fact, *UNIQUE-SAT* is in $\Delta_2^P = \mathcal{P}^{\mathcal{NP}}$: there is a polynomial time deterministic Turing machine M with an oracle for *SAT* that accepts *UNIQUE-SAT*. On input F with variables

$x_1, x_2, \ldots, x_k$, M executes the algorithm in Figure 9.2. Note the many invocations of the oracle for *SAT*.

if $F \notin SAT$ **then reject**;
comment: Construct a satisfying assignment $A_1 \wedge A_2 \wedge \cdots \wedge A_k$ by "self-reducibility";
for i **from** 1 **to** k **do**
  if $(F \wedge A_1 \wedge A_2 \wedge \cdots \wedge A_{i-1} \wedge x_i) \in SAT$
    then $A_i \leftarrow x_i$
    else $A_i \leftarrow \neg x_i$;
if $(F \wedge \neg(A_1 \wedge A_2 \wedge \cdots \wedge A_k)) \in SAT$
  then reject **comment:** there is at least one other satisfying assignment;
  else accept.

Figure 9.2: Algorithm Demonstrating That *UNIQUE-SAT* $\in \Delta_2^P$

9.5. Characterizing Σ_k^P and Π_k^P by Fixed Alternations

We next consider alternating Turing machines with a fixed number of alternations between existential and universal states. These machines provide an alternative characterization of the complexity classes that comprise the polynomial hierarchy.

Definition 9.24: A Σ_{k+1}^P machine is a polynomial time alternating Turing machine that starts in an existential state and, on any path of the computation tree, alternates between existential and universal states at most k times.

Definition 9.25: A Π_{k+1}^P machine is a polynomial time alternating Turing machine that starts in a universal state and, on any path of the computation tree, alternates between universal and existential states at most k times.

Theorem 9.26 (Stockmeyer and Meyer [44, 43]): For any positive integer k,

$$\Sigma_k^P = \{L \mid L \text{ is accepted by some } \Sigma_k^P \text{ machine}\}, \text{ and}$$
$$\Pi_k^P = \{L \mid L \text{ is accepted by some } \Pi_k^P \text{ machine}\}.$$

Proof: The proof is by induction on k.

Basis ($k = 1$):

$$\Sigma_1^P = \mathcal{NP} = \{L \mid L \text{ is accepted by some } \Sigma_1^P \text{ machine}\}, \text{ and}$$
$$\Pi_1^P = co\mathcal{NP} = \{L \mid L \text{ is accepted by some } \Pi_1^P \text{ machine}\}.$$

Induction ($k \geq 1$):

1. $\Sigma_{k+1}^P \subseteq \{L \mid L \text{ is accepted by some } \Sigma_{k+1}^P \text{ machine}\}$:

By definition, $\Sigma_{k+1}^P = \mathcal{NP}^{\Sigma_k^P}$. Let $L \in \Sigma_{k+1}^P$ be accepted by a nondeterministic oracle Turing machine M that runs in some polynomial $p_1(n)$ time with an oracle for $B \in \Sigma_k^P$. We want to

simulate M by a Σ_{k+1}^P machine A. The problem is that M may query its oracle a polynomial number of times, and simulating each of these queries requires some alternations.

We will show how to simulate M so that all oracle calls along any computation path are postponed until the end of that computation path. A simulates M, except that when M calls the oracle with query y_i, A guesses $b_i \in \{Y, N\}$ and records the pair (y_i, b_i) on a separate worktape. A then enters state q_{b_i} and continues the simulation of M. If M rejects, A rejects. If M accepts, then A must verify all guesses b_i before accepting.

The series of guesses left on the worktape are pairs of the form (y_i, b_i) and correspond to verifying that $y_i \in B$ if $b_i = Y$, and $y_i \in \Sigma^* - B$ if $b_i = N$. By the induction hypothesis, B is accepted by some Σ_k^P machine M_B and $\Sigma^* - B$ is accepted by some Π_k^P machine $M_{\Sigma^* - B}$, each running in some polynomial $p_2(n)$ time. It would be simple if A could now universally verify these guesses, but those with $b_i = Y$ would introduce one too many alternations. Instead, A first makes one pass over the worktape containing the pairs (y_i, b_i) and does the following:

- If $b_i = N$, A replaces the pair (y_i, b_i) by the initial configuration of $M_{\Sigma^* - B}$ on input y_i.
- If $b_i = Y$, A simulates M_B on input y_i until M_B reaches its first universal configuration Q, and replaces the pair (y_i, b_i) by Q.

Until this point, A has used no universal configurations. A now universally verifies that each configuration remaining on the tape is accepting by simulating M_B or $M_{\Sigma^* - B}$, as appropriate. This takes at most another $k - 1$ alternations, since these are configurations accepted by Π_k^P machines.

The running time of this simulation is $O((p_1(n))^2 + p_2(p_1(n)))$, which is also polynomial, as follows. M may query its oracle at most $p_1(n)$ times on queries y_i each of length at most $p_1(n)$. For each, it may take time $O(p_1(n))$ to record y_i, and time $O(p_2(p_1(n)))$ to simulate M_B or $M_{\Sigma^* - B}$ on input y_i.

2. $\Sigma_{k+1}^P \supseteq \{L \mid L \text{ is accepted by some } \Sigma_{k+1}^P \text{ machine}\}$:

Let L be accepted by some Σ_{k+1}^P machine A. Let $B = \{Q \mid Q$ is a universal configuration of A that leads to acceptance in at most $k - 1$ alternations$\}$. B is accepted by a Π_k^P machine that simulates A starting at Q so, by the induction hypothesis, $B \in \Pi_k^P$.

We construct a nondeterministic oracle Turing machine M such that $L = L(M^B)$. M simulates A until A enters its first universal configuration Q, and then queries its oracle to determine if $Q \in B$, accepting if and only if this is so. The running time of M is less than the running time of A, so it is certainly polynomial.

3. $\Pi_{k+1}^P = \{L \mid L \text{ is accepted by some } \Pi_{k+1}^P \text{ machine}\}$:

$$\begin{aligned} \Pi_{k+1}^P &= co\Sigma_{k+1}^P \\ &= \{\Sigma^* - L \mid L \text{ is accepted by some } \Sigma_{k+1}^P \text{ machine}\} \\ &= \{L \mid L \text{ is accepted by some } \Pi_{k+1}^P \text{ machine}\}. \end{aligned}$$

The last equality follows by interchanging existential and universal states, and applying deMorgan's laws. □

One corollary of Theorem 9.26 provides a useful normal form for nondeterministic oracle Turing machines:

Corollary 9.27: $L \in \Sigma_{k+1}^P$ if and only if $L = L(M^B)$, where $B \in \Pi_k^P$ and M is a nondeterministic oracle Turing machine that runs in polynomial time, queries B once, and accepts if and only if the answer is yes.

$L \in \Pi_{k+1}^P$ if and only if $L = L(M^B)$, where $B \in \Sigma_k^P$ and M is a "co-nondeterministic" oracle Turing machine (i.e., no existential states) that runs in polynomial time, queries B once, and accepts if and only if the answer is yes.

Proof: This restricted form of oracle Turing machine is exactly what arises in part 2 of the proof of Theorem 9.26. □

Note the similarity between this normal form for nondeterministic oracle Turing machines and many-one reductions. The only difference is that in Corollary 9.27 the reduction is nondeterministic.

Corollary 9.28: $\mathcal{PH} \subseteq \mathcal{PSPACE}$.

Proof: $\mathcal{PH}$ is the subset of $\mathcal{PSPACE}$ obtained by fixing the number of alternations of a polynomial time alternating Turing machine. □

9.6. How to Collapse the Polynomial Hierarchy

Theorem 9.29 (Stockmeyer [43]): If $\Sigma_k^P = \Pi_k^P$ for any $k \geq 1$, then $\Sigma_j^P = \Pi_j^P = \Sigma_k^P$ for all $j \geq k$.

Proof: The proof is by induction on j.

BASIS ($j = k$): Vacuous.

INDUCTION ($j \geq k$): Assume by the induction hypothesis that $\Sigma_j^P = \Pi_j^P = \Sigma_k^P$.

1. $\Sigma_{j+1}^P \subseteq \Sigma_k^P$: Let $L \in \Sigma_{j+1}^P$. By Corollary 9.27, $L = L(M^B)$ where $B \in \Pi_j^P$, and M is a polynomial time nondeterministic oracle Turing machine that makes one query "$y \in B$?", and accepts if and only if the answer is yes. By the induction hypothesis, $B \in \Sigma_k^P$, so B is accepted by some Σ_k^P machine A by Theorem 9.26. Then L is also accepted by some Σ_k^P machine, as follows. Simulate M until it enters state $q_?$ with y on its query tape, and then simulate A on y. Since M is nondeterministic and A is a Σ_k^P machine, there are only $k-1$ alternations. Since M and A each run in polynomial time and y has polynomial length, the total time is polynomial. Therefore, by Theorem 9.26 again, $L \in \Sigma_k^P$.

2. $\Pi_{j+1}^P \subseteq \Sigma_k^P$: $\Pi_{j+1}^P = co\Sigma_{j+1}^P \subseteq co\Sigma_k^P = \Pi_k^P = \Sigma_k^P$. □

The following corollary shows that separating $\mathcal{PH}$ from $\mathcal{P}$ would be sufficient to prove $\mathcal{P} \neq \mathcal{NP}$.

Corollary 9.30: $\mathcal{P} \neq \mathcal{PH}$ if and only if $\mathcal{P} \neq \mathcal{NP}$.

Proof:

"if" clause: $\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PH}$, so $\mathcal{P} \neq \mathcal{NP}$ implies $\mathcal{P} \neq \mathcal{PH}$.

"only if" clause: If $\mathcal{P} = \mathcal{NP}$ then $\Sigma_1^P = \mathcal{NP} = \mathcal{P} = co\mathcal{P} = co\mathcal{NP} = \Pi_1^P$. By Theorem 9.29, then, $\mathcal{PH} = \Sigma_1^P = \mathcal{NP} = \mathcal{P}$. □

9.7. Relating Circuits to $\mathcal{NP}$

Having shown in Theorem 7.17 that $\mathcal{P} \subseteq \mathcal{PSIZE}$, and in Exercise 7.20 that $\mathcal{P} \neq \mathcal{PSIZE}$, a natural question to ask is whether $\mathcal{NP} \subseteq \mathcal{PSIZE}$. In this section we will see that if the answer is positive, then the polynomial hierarchy collapses to $\Sigma_2^P \cap \Pi_2^P$. Therefore, such a containment seems unlikely.

Definition 9.31: *SAT-CIRCUIT* $= \{C \mid C$ has m inputs and accepts the set of length m encodings of satisfiable formulas$\}$.

Definition 9.32: If F is a propositional formula and $A_1, A_2, \ldots, A_i$ are literals of distinct variables of F, then $F \mid (A_1, A_2, \ldots, A_i)$ denotes F with the substitutions $A_1 = A_2 = \cdots = A_i = 1$ and $\neg A_1 = \neg A_2 = \cdots = \neg A_i = 0$.

Any standard binary encoding of formulas will suffice for Definition 9.31, provided that the encoding of $F \mid (A_1, A_2, \ldots, A_i)$ is no longer than the encoding of F. For example, it suffices if the constants 0 and 1 have shorter encodings than any literal. The method must also be capable of padding out such shorter encodings to m bits.

Lemma 9.33: *SAT-CIRCUIT* $\in$ *co*$\mathcal{NP}$.

Proof: To avoid later confusion, note that three different levels of computing devices are involved in the proof: first there is the *co*$\mathcal{NP}$ machine whose input is the encoding of a circuit; this circuit itself has as input the encoding of a formula, which in turn has as input a truth assignment to its variables.

CONSTRUCTION: On input C, a circuit with m inputs, construct a *co*$\mathcal{NP}$ machine M that executes the algorithm in Figure 9.3.

CORRECTNESS: The only part of the correctness that is not immediate from the construction is the self-reducibility segment. If $A_1 = A_2 = \cdots = A_k = 1$ satisfies F, then C has correctly accepted F as satisfiable, and M should accept. Assume, then, that $A_1 = A_2 = \cdots = A_k = 1$ does not satisfy F, yet $C \in$ *SAT-CIRCUIT*. Let i be any value such that $F \mid (A_1, A_2, \ldots, A_{i-1})$ is satisfiable, but $F \mid (A_1, A_2, \ldots, A_{i-1}, A_i)$ is not; there must be such an i, since C (correctly) output 1 on input F, but $F \mid (A_1, A_2, \ldots, A_k)$ is false. If $A_i = x_i$, then C erroneously stated that $F \mid (A_1, A_2, \ldots, A_{i-1}, A_i)$ is satisfiable. If $A_i = \neg x_i$, then both $F \mid (A_1, A_2, \ldots, A_{i-1}, x_i)$ and $F \mid (A_1, A_2, \ldots, A_{i-1}, \neg x_i)$ are unsatisfiable, so that C erroneously stated that $F \mid (A_1, A_2, \ldots, A_{i-1})$ is satisfiable. In either case, $C \notin$ *SAT-CIRCUIT*, so M should reject.

ANALYSIS: There are five lines in Figure 9.3 marked with asterisks, in which M must solve a circuit value or formula value problem. By Theorem 7.11, each can be done deterministically in polynomial time. □

Theorem 9.34 (Karp and Lipton [23]): If $\mathcal{NP} \subseteq \mathcal{PSIZE}$ then $\mathcal{PH} = \Sigma_2^P \cap \Pi_2^P$.

Proof: Suppose some family of circuits C_n of polynomial size $p_1(n)$ accepts *SAT*. We will show that $\Pi_2^P \subseteq \Sigma_2^P$, and the theorem then follows from Proposition 9.8 and Theorem 9.29.

```
universally choose F ∈ {0,1}^m ;
comment: verify that C behaves correctly on every input F ;
if F does not encode a propositional formula with some number k of variables
   then
      if C outputs 0 on input F                                          *
         then accept
         else reject
   else
      if C outputs 0 on input F                                          *
         then begin
            universally choose truth assignment A ∈ {0,1}^k ;
            if A falsifies F                                             *
               then accept
               else reject
         end
         else begin
            comment: use self-reducibility to verify that F is satisfiable ;
            for i from 1 to k do
               if C outputs 1 on input F | (A_1, A_2, ..., A_{i-1}, x_i)  *
                  then A_i ← x_i
                  else A_i ← ¬x_i ;
            if A_1 = A_2 = ··· = A_k = 1 satisfies F                     *
               then accept
               else reject
         end .
```

Figure 9.3: A $co\mathcal{NP}$ Algorithm That Accepts *SAT-CIRCUIT*

CONSTRUCTION: Let $L \in \Pi_2^P$. By Corollary 9.27, $L = L(M^{SAT})$, where M is a "co-nondeterministic" oracle Turing machine (i.e., no existential states) that runs in polynomial time $p_2(n)$ and makes one call on an oracle for *SAT*. Construct a Σ_2^P machine A that accepts L as described below, and then apply Theorem 9.26.

On input x, A does the following:

```
existentially choose the encoding of a circuit C ;
universally choose b ∈ {0,1};
   case b of
      0: if C ∈ SAT-CIRCUIT then accept else reject;
      1: simulate M on input x using C to answer the oracle query ;
         if C does not have enough inputs to accommodate the query then reject;
         if M accepts x then accept else reject
   end .
```

CORRECTNESS: If M^{SAT} accepts x, then some choice of C will be in *SAT-CIRCUIT*, and will have enough inputs to accommodate M's query. For this choice of C, A will accept x. Conversely, if A accepts x, then $C \in$ *SAT-CIRCUIT* and hence answers the oracle query correctly. Hence, M^{SAT} accepts x.

ANALYSIS: By Lemma 9.33, *SAT-CIRCUIT* $\in co\mathcal{NP}$, so there is only one alternation. Let $p_3(n)$ be the running time of the $co\mathcal{NP}$ machine of Lemma 9.33, and $p_4(n)$ be the time to solve the circuit value problem on inputs of length n. M can only create a query of length $p_2(n)$, so it suffices if C has $p_2(n)$ inputs and, by hypothesis, $p_1(p_2(n))$ size. Such a circuit has an encoding of length $p(n) = O(p_1(p_2(n)) \log p_1(p_2(n)))$. Therefore, A runs in time $O(p(n)+p_3(p(n))+p_2(n)+p_4(p(n)))$, a polynomial. □

9.8. Exercises

1. If $\mathcal{C}$ is a set of languages over some alphabet Σ and L is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{C}$, prove that $\Sigma^* - L$ is $\leq_m^{\mathcal{L}}$–complete for $co\mathcal{C}$.

2. Prove that $\leq_m^{\mathcal{P}}$ is transitive.

3. Prove that $\leq_T^{\mathcal{P}}$ is transitive.

4. Prove that $\mathcal{NP}$ is closed under $\leq_m^{\mathcal{P}}$.

5. Prove that the following statements are equivalent:

   (a) $\mathcal{NP} = co\mathcal{NP}$.

   (b) Some language that is $\leq_m^{\mathcal{P}}$–complete for $\mathcal{NP}$ is in $co\mathcal{NP}$.

   (c) Some language that is $\leq_T^{\mathcal{P}}$–complete for $\mathcal{NP}$ is in $co\mathcal{NP}$.

6. What is wrong with the following proof that $\mathcal{P} \neq \mathcal{NP}$?

   For any two languages A and B in $\mathcal{P}$, A is reducible to B by a polynomial time reduction, since the reduction can solve membership in A without even using B. But $SAT \not\leq_m^{\mathcal{P}} \emptyset$, where $\emptyset$ is the empty set, since, for any satisfiable formula F and any reduction f, it is not true that $F \in SAT$ if and only if $f(F) \in \emptyset$. Since $\emptyset$ is obviously in $\mathcal{P}$, $SAT \notin \mathcal{P}$.