# 1. The Self-Reducibility Technique

A set $S$ is *sparse* if it contains at most polynomially many elements at each length, i.e.,

$$(\exists \text{ polynomial } p)(\forall\, n)[||\{x \mid x \in S \wedge |x| = n\}|| \leq p(n)]. \tag{1.1}$$

This chapter studies one of the oldest questions in computational complexity theory: Can sparse sets be NP-complete?

As we noted in the Preface, the proofs of most results in complexity theory rely on algorithms, and the proofs in this chapter certainly support that claim. In Sect. 1.1, we[1] will use a sequence of increasingly elaborate deterministic tree-pruning and interval-pruning procedures to show that sparse sets cannot be $\leq_m^p$-complete, or even $\leq_{btt}^p$-hard, for NP unless P = NP. (The appendices contain definitions of and introductions to the reduction types, such as $\leq_m^p$ and $\leq_{btt}^p$, and the complexity classes, such as P and NP, that are used in this book.)

Section 1.2 studies whether NP can have $\leq_T^p$-complete or $\leq_T^p$-hard sparse sets. $\mathrm{P}^{\mathrm{NP}[\mathcal{O}(\log n)]}$ denotes the class of languages that can be accepted by some deterministic polynomial-time Turing machine allowed at most $\mathcal{O}(\log n)$ queries to some NP oracle. In Sect. 1.2, we will—via binary search, self-reducibility algorithms, and nondeterministic algorithms—prove that sparse sets cannot be $\leq_T^p$-complete for NP unless the polynomial hierarchy collapses to $\mathrm{P}^{\mathrm{NP}[\mathcal{O}(\log n)]}$, and that sparse sets cannot be $\leq_T^p$-hard for NP unless the polynomial hierarchy collapses to $\mathrm{NP}^{\mathrm{NP}}$.

As is often the case in complexity-theoretic proofs, we will typically use in the construction of our algorithms the hypothesis of the theorem that the algorithm is establishing (e.g., we will build a P algorithm for SAT, and will use in the algorithm the—literally hypothetical—sparse $\leq_m^p$-complete set for NP). In fact, this "theorems via algorithms under hypotheses" approach is employed in each section of this chapter.

Furthermore, most of Sects. 1.1 and 1.2 are unified by the spirit of their algorithmic attack, which is to exploit the "(disjunctive) self-reducibility" of SAT—basically, the fact that a boolean formula is satisfiable if and only if either it is satisfiable with its first variable set to False or it is satisfiable with

[1] In this book, "we" usually refers to the authors and the readers as we travel together in our exploration of complexity theory.

its first variable set to True. A partial exception to the use of this attack in those sections is the left set technique, which we use in Sect. 1.1.2. This technique, while in some sense a veiled tree-pruning procedure inspired by a long line of self-reducibility-based tree-pruning procedures, adds a new twist to this type of argument, rather than being a direct invocation of SAT's self-reducibility.

Section 1.3 studies not whether there are sparse NP-complete sets, but rather whether NP − P contains *any* sparse sets at all. Like the previous sections, this section employs explicit algorithmic constructions that themselves use objects hypothesized to exist by the hypotheses of the theorems for which they are providing proofs. The actual result we arrive at is that NP − P contains sparse sets if and only if deterministic and nondeterministic exponential time differ.

Throughout this book, we will leave the type of quantified variables implicit when it is clear from context what that type is. For example, in equation 1.1, the "$(\forall n)$" is implicitly "$(\forall n \in \{0, 1, 2, \ldots\})$," and "$(\forall x)$" is typically a shorthand for "$(\forall x \in \Sigma^*)$." We will use a colon to denote a constraint on a variable, i.e., "$(\forall x : R(x))\,[S(x)]$" means "$(\forall x)\,[R(x) \implies S(x)]$," and "$(\exists x : R(x))\,[S(x)]$" means "$(\exists x)\,[R(x) \wedge S(x)]$." For any set $A$ and any natural number $n$, we will use $A^{\leq n}$ to denote the strings of $A$ that are of length at most $n$, and we will use $A^{=n}$ to denote the strings of $A$ that are of length exactly $n$. Given a Turing machine $M$, we will use $L(M)$ to denote the language accepted by the machine (with respect to whatever the acceptance mechanism of the machine is).

## 1.1 GEM: There Are No Sparse NP-Complete Sets Unless P=NP

### 1.1.1 Setting the Stage: The Pruning Technique

Before we turn to Mahaney's Theorem—NP has sparse complete sets only if P = NP—and its generalization to bounded-truth-table reductions, we first prove two weaker results that display the self-reducibility-based tree-pruning approach in a simpler setting. (Below, in a small abuse of notation we are taking "1" in certain places—such as in expressions like "$1^*$"—as a shorthand for the regular expression representing the set $\{1\}$.)

**Definition 1.1** *A set $T$ is a* tally set *exactly if $T \subseteq 1^*$.*

**Theorem 1.2** *If there is a tally set that is $\leq_m^p$-hard for* NP*, then* P = NP*.*

**Corollary 1.3** *If there is a tally set that is NP-complete, then* P = NP*.*

We note in passing that if P = NP, then the singleton set $\{1\}$ is trivially both NP-complete and coNP-complete. Thus, all the "if...then..." theorems of this section (Sect. 1.1) have true converses. We state them in "if...then..." form to stress the interesting direction.

**Proof of Theorem 1.2** Let $T$ be a tally set that is $\leq_m^p$-hard for NP. Since the NP-complete set

$$\mathrm{SAT} = \{f \mid f \text{ is a satisfiable boolean formula}\}$$

is in NP and $T$ is $\leq_m^p$-hard for NP, it follows that SAT$\leq_m^p T$. Let $g$ be a deterministic polynomial-time function many-one reducing SAT to $T$. Let $k$ be an integer such that $(\forall x)[|g(x)| \leq |x|^k + k]$; since $g$ is computable by some deterministic polynomial-time Turing machine, such a $k$ indeed must exist since that machine outputs at most one character per step.

We now give, under the hypothesis of the theorem, a deterministic polynomial-time algorithm for SAT, via a simple tree-pruning procedure. The input to our algorithm is a boolean formula $F$. Without loss of generality, let its variables be $v_1, \ldots, v_m$ and let $m \geq 1$. We will denote the result of assigning values to some of the variables of $F$ via expressions of the following form: $F[v_1 = \mathrm{True}, v_3 = \mathrm{False}]$, where True denotes the constant true and False denotes the constant false. For example, if $F = v_1 \vee v_2 \vee v_3$ then

$$F[v_1 = \mathrm{True}, v_3 = \mathrm{False}] = \mathrm{True} \vee v_2 \vee \mathrm{False},$$

and

$$(F[v_1 = \mathrm{True}])[v_3 = \mathrm{False}] = \mathrm{True} \vee v_2 \vee \mathrm{False}.$$

Our algorithm has stages numbered $0, 1, \ldots, m+1$. At the end of each stage (except the final one), we pass forward a collection of boolean formulas. Initially, we view ourselves as having just completed Stage 0, and we view ourselves as passing forward from Stage 0 a collection, $C$, containing the single formula $F$.

**Stage $i$, $1 \leq i \leq m$, assuming that the collection at the end of Stage $i-1$ is the following collection of formulas: $\{F_1, \ldots, F_\ell\}$.**

**Step 1** Let $\mathcal{C}$ be the collection

$$\{F_1[v_i = \mathrm{True}], F_2[v_i = \mathrm{True}], \ldots F_\ell[v_i = \mathrm{True}],$$
$$F_1[v_i = \mathrm{False}], F_2[v_i = \mathrm{False}], \ldots F_\ell[v_i = \mathrm{False}]\}.$$

**Step 2** Set $\mathcal{C}'$ to be $\emptyset$.

**Step 3** For each formula $f$ in $\mathcal{C}$ (in arbitrary order) do:

1. Compute $g(f)$.
2. If $g(f) \in 1^*$ and for no formula $h \in \mathcal{C}'$ does $g(f) = g(h)$, then add $f$ to $\mathcal{C}'$.

**End Stage $i$ [$\mathcal{C}'$ is the collection that gets passed on to Stage $i+1$]**

The action of our algorithm at Stage $m+1$ is simple: $F$ is satisfiable if and only if some member of the (variable-free) formula collection output by Stage $m$ evaluates to being true.

As to the correctness of our algorithm, note that after Stage 0 it certainly holds that

$$\begin{gathered}\text{the collection, } C\text{, contains some satisfiable formula}\\ \Longleftrightarrow \\ F \text{ is satisfiable,}\end{gathered} \tag{1.2}$$

since after Stage 0 formula $F$ is the only formula in the collection. Note also that, for each $i$, $1 \leq i \leq m$,

$$\begin{gathered}\text{the collection input to Stage } i \text{ contains some}\\ \text{satisfiable formula}\\ \Longleftrightarrow \\ \text{the collection output by Stage } i \text{ contains some}\\ \text{satisfiable formula.}\end{gathered} \tag{1.3}$$

Will now argue that this is so, via using a self-reducibility-based argument. In the present context, the relevant self-reducibility fact is that for any formula $F$ containing $v$ as one of its variables,

$$\begin{gathered}F \text{ is satisfiable } \Longleftrightarrow \\ ((F[v = \text{True}] \text{ is satisfiable}) \vee (F[v = \text{False}] \text{ is satisfiable})),\end{gathered}$$

since any satisfying assignment must assign some value to each variable. So Step 1 of Stage $i$ does no damage to our invariant, equation 1.3. What about Steps 2 and 3? (In terms of the connection to Step 1, it is important to keep in mind that if, for example, formula $F$ having variable $v$ is in our collection at the start of the stage and is satisfiable, then it must be the case that

$$(F[v = \text{True}] \text{ is satisfiable}) \vee (F[v = \text{False}] \text{ is satisfiable})\,,$$

so it must be the case that

$$g(F[v = \text{True}]) \in T \vee g(F[v = \text{False}]) \in T.$$

And of course, $T \subseteq 1^*$.) Steps 2 and 3 "prune" the formula set as follows. Each formula $f$ from Step 1 is kept unless either

a. $g(f) \notin 1^*$, or
b. $g(f) \in 1^*$ but some $h \in \mathcal{C}'$ has $g(f) = g(h)$.

Both these ways, (a) and (b), of dropping formulas are harmless. Recall that $\mathrm{SAT} \leq_m^p T$ via function $g$, and so if $f \in \mathrm{SAT}$ then $g(f) \in T$. However, regarding (a), $T \subseteq 1^*$ so if $g(f) \notin 1^*$ then $g(f) \notin T$, and so $f \notin \mathrm{SAT}$. Regarding (b), if $g(f) = g(h)$ and $h$ has already been added to the collection to be output by Stage (i), then there is no need to output $f$ as—since $\mathrm{SAT} \leq_m^p T$ via reduction $g$—we know that

$$f \in \mathrm{SAT} \iff g(f) \in T$$

and

$$h \in \mathrm{SAT} \iff g(h) \in T.$$

Thus, $f \in \mathrm{SAT} \iff h \in \mathrm{SAT}$, and so by discarding $f$ but leaving in $h$, we do no damage to our invariant, equation 1.3. So by equations 1.2 and 1.3 we see that $F$ is satisfiable if and only if some formula output by Stage $m$ is satisfiable. As the formulas output by Stage $m$ have no free variables, this just means that one of them must evaluate to being true, which is precisely what Stage $m + 1$ checks.

Thus, the algorithm correctly checks whether $F$ is satisfiable. But is this a polynomial-time algorithm? $|F|$ will denote the length of $F$, i.e., the number of bits in the representation of $F$. Let $|F| = p$. Note that after any stage there are at most $p^k + k + 1$ formulas in the output collection, and each of these formulas is of size at most $p$. This size claim holds as each formula in an output collection is formed by one or more assignments of variables of $F$ to being True or False, and such assignments certainly will not cause an *increase* in length (in a standard, reasonable encoding). We will say that a string $s$ is a *tally string* exactly if $s \in 1^*$. The $p^k + k + 1$ figure above holds as (due to the final part of Step 3 of our algorithm) we output at most one formula for each tally string to which ($n^k + k$-time function) $g$ can map, and even if $g$ outputs a 1 on each step, $g$ can output in $p^k + k$ steps no tally string longer than $1^{p^k+k}$. So, taking into account the fact that the empty string is a (degenerate) tally string, we have our $p^k + k + 1$ figure. From this, from the specification of the stages, and from the fact that $g$ itself is a polynomial-time computable function, it follows clearly that the entire algorithm runs in time polynomial in $|F|$. ❑

In the proof of Theorem 1.2, we used self-reducibility to split into two each member of a set of formulas, and then we pruned the resulting set using the fact that formulas mapping to non-tally strings could be eliminated, and the fact that only one formula mapping to a given tally string need be kept. By repeating this process we walked down the self-reducibility tree of any given formula, yet we pruned that tree well enough to ensure that only a polynomial number of nodes had to be examined at each level of the tree. By the self-reducibility tree—more specifically this is a disjunctive self-reducibility tree—of a formula, we mean the tree that has the formula as its root, and in which each node corresponding to a formula with some variables unassigned has as its left and right children the same formula but with the lexicographically first unassigned variable set respectively to True and to False.

In the proof of Theorem 1.2, we were greatly helped by the fact that we were dealing with whether *tally sets* are hard for NP. Tally strings are easily identifiable as such, and that made our pruning scheme straightforward. We now turn to a slightly more difficult case.

**Theorem 1.4** *If there is a sparse set that is $\leq_m^p$-hard for* coNP*, then* $\mathrm{P} = \mathrm{NP}$.

**Corollary 1.5** *If there is a sparse coNP-complete set, then* $\mathrm{P} = \mathrm{NP}$.

The proof of Theorem 1.4 goes as follows. As in the proof of Theorem 1.2, we wish to use our hypothesis to construct a polynomial-time algorithm for SAT. Indeed, we wish to do so by expanding and pruning the self-reducibility tree as was done in the proof of Theorem 1.2. The key obstacle is that the pruning procedure from the proof of Theorem 1.2 no longer works, since unlike tally sets, sparse sets are not necessarily "P-capturable" (a set is P-capturable if it is a subset of some sparse P set). In the following proof, we replace the tree-pruning procedure of Theorem 1.2 with a tree-pruning procedure based on the following counting trick. We expand our tree, while pruning only duplicates; we argue that if the tree ever becomes larger than a certain polynomial size, then the very failure of our tree pruning proves that the formula is satisfiable.

**Proof of Theorem 1.4** Let $S$ be the (hypothetical) sparse set that is $\leq_m^p$-hard for coNP. For each $\ell$, let $p_\ell(n)$ denote the polynomial $n^\ell + \ell$. Let $d$ be such that $(\forall n)[||S^{\leq n}|| \leq p_d(n)]$.[2] Since SAT $\in$ NP, it follows that $\overline{\mathrm{SAT}} \leq_m^p S$. Let $g$ be a deterministic polynomial-time function many-one reducing $\overline{\mathrm{SAT}}$ to $S$. Let $k$ be an an integer such that $(\forall x)[|g(x)| \leq p_k(n)$; since $g$ is computed by a deterministic polynomial-time Turing machine, such a $k$ indeed exists.

We now give, under the hypothesis of this theorem, a deterministic polynomial-time algorithm for SAT, via a simple tree-pruning procedure. As in the proof of Theorem 1.2, let $F$ be an input formula, and let $m$ be the number of variables in $F$. Without loss of generality, let $m \geq 1$ and let the variables of $F$ be named $v_1, \ldots, v_m$. Each stage of our construction will pass forward a collection of formulas. View Stage 0 as passing on to the next stage the collection containing just the formula $F$. We now specify Stage $i$. Note that Steps 1 and 2 are the same as in the proof of Theorem 1.2, Step 3 is modified, and Step 4 is new.

**Stage $i$, $1 \leq i \leq m$, assuming the collection at the end of Stage $i-1$ is $\{F_1, \ldots, F_\ell\}$.**

**Step 1** Let $\mathcal{C}$ be the collection

$$\{F_1[v_i = \mathrm{True}], F_2[v_i = \mathrm{True}], \ldots F_\ell[v_i = \mathrm{True}],$$
$$F_1[v_i = \mathrm{False}], F_2[v_i = \mathrm{False}], \ldots F_\ell[v_i = \mathrm{False}]\}.$$

**Step 2** Set $\mathcal{C}'$ to be $\emptyset$.

**Step 3** For each formula $f$ in $\mathcal{C}$ (in arbitrary order) do:

1. Compute $g(f)$.
2. If for no formula $h \in \mathcal{C}'$ does $g(f) = g(h)$, then add $f$ to $\mathcal{C}'$.

[2] The $||S^{\leq n}||$, as opposed to the $||S^{=n}||$ that implicitly appears in the definition of "sparse set" (equation 1.1), is not a typographical error. Both yield valid and equivalent definitions of the class of sparse sets. The $||S^{=n}||$ approach is, as we will see in Chap. 3, a bit more fine-grained. However, the proof of the present theorem works most smoothly with the $||S^{\leq n}||$ definition.

**Step 4** If $\mathcal{C}'$ contains at least $p_d(p_k(|F|))+1$ elements, stop and immediately declare that $F \in \mathrm{SAT}$. (The reason for the $p_d(p_k(|F|))+1$ figure will be made clear below.)

**End Stage $i$ [$\mathcal{C}'$ is the collection that gets passed on to Stage $i+1$]**

The action of our algorithm at Stage $m+1$ is as follows: If some member of the (variable-free) formula collection output by Stage $m$ evaluates to being true we declare $F \in \mathrm{SAT}$, and otherwise we declare $F \notin \mathrm{SAT}$.

Why does this algorithm work? Let $n$ represent $|F|$. Since $p_d(p_k(n))+1$ is a polynomial in the input size, $F$, it is clear that the above algorithm runs in polynomial time. If the hypothesis of Step 4 is never met, then the algorithm is correct for reasons similar to those showing the correctness of the proof of Theorem 1.2.

If Step 4 is ever invoked, then at the stage at which it is invoked, we have $p_d(p_k(n))+1$ distinct strings being mapped to by the non-pruned nodes at the current level of our self-reducibility tree. (Recall that by the self-reducibility tree—more specifically this is a disjunctive self-reducibility tree—of a formula, we mean the tree that has the formula as its root, and in which each node corresponding to a formula with some variables unassigned has as its left and right children the same formula but with the lexicographically first unassigned variable set respectively to True and False.) Note that each of these mapped-to strings is of length at most $p_k(n)$ since that is the longest string that reduction $g$ can output on inputs of size at most $n$. However, there are only $p_d(p_k(n))$ strings in $S^{\leq p_k(n)}$. As usual, $\Sigma$ denotes our alphabet, and as usual we take $\Sigma = \{0,1\}$. So since the formulas in our collection map to $p_d(p_k(n))+1$ distinct strings in $(\Sigma^*)^{\leq p_k(n)}$, *at least one formula in our collection, call it $H$, maps under the action of $g$ to a string in $\overline{S}$.*[3] So $g(H) \notin S$. However, $\overline{\mathrm{SAT}}$ reduces to $S$ via $g$, so $H$ is satisfiable. Since $H$ was obtained by making substitutions to some variables of $F$, it follows that $F$ is satisfiable. Thus, if the hypothesis of Step 4 is ever met, it is indeed correct to halt immediately and declare that $F$ is satisfiable. ❑ Theorem 1.4

**Pause to Ponder 1.6** *In light of the comment in footnote 3, change the proof so that Step 4 does not terminate the algorithm, but rather the algorithm drives forward to explicitly find a satisfying assignment for $F$. (Hint: The crucial point is to, via pruning, keep the tree from getting too large. The following footnote contains a give-away hint.*[4]*)*

[3] Note that in this case we know that such an $H$ exists, but we have no idea which formula is such an $H$. See Pause to Ponder 1.6 for how to modify the proof to make it more constructive.

[4] Change Step 4 so that, as soon as $\mathcal{C}'$ contains $p_d(p_k(n))+1$ formulas, no more elements are added to $\mathcal{C}'$ at the current level.

### 1.1.2 The Left Set Technique

**1.1.2.1 Sparse Complete Sets for NP.** So far we have seen, as the proofs of Theorems 1.2 and 1.4, tree-pruning algorithms that show that "thin" sets cannot be hard for certain complexity classes. Inspired by these two results, Mahaney extended them by proving the following lovely, natural result.

**Theorem 1.7** *If* NP *has sparse complete sets then* P = NP.

**Pause to Ponder 1.8** *The reader will want to convince him- or herself of the fact that the approach of the proof of Theorem 1.4 utterly fails to establish Theorem 1.7. (See this footnote for why.*[5]*)*

We will not prove Theorem 1.7 now since we soon prove, as Theorem 1.10, a more general result showcasing the left set technique, and that result will immediately imply Theorem 1.7. Briefly put, the new technique needed to prove Theorems 1.7 and 1.10 is the notion of a "left set." Very informally, a left set fills in gaps so as to make binary search easier.

Theorem 1.7 establishes that if there is a sparse NP-complete set then P = NP. For NP, the existence of sparse NP-hard sets and the existence of sparse NP-complete sets stand or fall together. (One can alternatively conclude this from the fact that Theorem 1.10 establishes its result for NP-$\leq_{btt}^{p}$-hardness rather than merely for NP-$\leq_{btt}^{p}$-completeness.)

**Theorem 1.9** NP *has sparse* $\leq_m^p$*-hard sets if and only if* NP *has sparse* $\leq_m^p$*-complete sets.*

**Proof** The "if" direction is immediate. So, we need only prove that if NP has a $\leq_m^p$-hard sparse set then it has a $\leq_m^p$-complete sparse set. Let $S$ be any sparse set that is $\leq_m^p$-hard for NP. Since $S$ is $\leq_m^p$-hard, it holds that $\mathrm{SAT}\leq_m^p S$. Let $f$ be a polynomial-time computable function that many-one reduces SAT to $S$. Define

$$S' = \{0^k\#y \mid k \geq 0 \wedge (\exists x \in \mathrm{SAT})[k \geq |x| \wedge f(x) = y]\}.$$

The rough intuition here is that $S'$ is almost $f(\mathrm{SAT})$, except to make the proof work it via the $0^k$ also has a padding part. Note that if $0^k\#z \in S'$ then certainly $z \in S$. $S'$ is clearly in NP, since to test whether $0^k\#z$ is in $S'$ we nondeterministically guess a string $x$ of length at most $k$ and we nondeterministically guess a potential certificate of $x \in \mathrm{SAT}$ (i.e., we guess a complete assignment of the variables of the formula $x$), and (on each guessed path) we accept if the guessed string/certificate pair is such that $f(x) = z$

[5] The analogous proof would merely be able to claim that if the tree were getting "bushy," there would be at least one *unsatisfiable* formula among the collection. This says nothing regarding whether some other formula might be satisfiable. Thus, even if the set $\mathcal{C}'$ is getting very large, we have no obvious way to prune it.

and the certificate proves that $x \in \mathrm{SAT}$. Given that $S$ is sparse, it is not hard to see that $S'$ is also sparse. Finally, $S'$ is NP-hard because, in light of the fact that $\mathrm{SAT} \leq_m^p S$ via polynomial-time reduction $f$, it is not hard to see that $\mathrm{SAT} \leq_m^p S'$ via the reduction $f'(x) = 0^{|x|}\#f(x)$. ❑

We now turn to presenting the left set technique. We do so via proving that if some sparse set is NP-complete under bounded-truth-table reductions then P = NP.

**Theorem 1.10** *If there is a sparse set then* P = NP *that is* $\leq_{btt}^p$*-hard for* NP*, then* P = NP*.*

In the rest of the section, we prove Theorem 1.10.

**1.1.2.2 The Left Set and $w_{\max}$.** Let $L$ be an arbitrary member of NP. There exist a polynomial $p$ and a language in $A \in \mathrm{P}$ such that, for every $x \in \Sigma^*$,

$$x \in L \iff (\exists w \in \Sigma^{p(|x|)})[\langle x, w\rangle \in A].$$

For each $x \in \Sigma^*$ and $w \in \Sigma^*$, call $w$ a *witness for $x \in L$ with respect to $A$ and $p$* if $|w| = p(|x|)$ and $\langle x, w\rangle \in A$. Define the *left set with respect to $A$ and $p$*, denoted by *Left*$[A, p]$, to be

$$\{\langle x, y\rangle \mid x \in \Sigma^* \wedge y \in \Sigma^{p(|x|)} \wedge (\exists w \in \Sigma^{p(|x|)})[w \geq y \wedge \langle x, w\rangle \in A]\},$$

i.e., *Left*$[A, p]$ is the set of all $\langle x, y\rangle$ such that $y$ belongs to $\Sigma^{p(|x|)}$ and is "to the left" of some witness for $x \in L$ with respect to $A$ and $p$. For each $x \in \Sigma^*$, define

$$w_{\max}(x) = \max\{y \in \Sigma^{p(|x|)} \mid \langle x, y\rangle \in A\};$$

if $\{y \in \Sigma^{p(|x|)} \mid \langle x, y\rangle \in A\}$ is empty, then $w_{\max}(x)$ is undefined. In other words, $w_{\max}(x)$ is the lexicographic maximum of the witnesses for $x \in L$ with respect to $A$ and $p$. Clearly, for every $x \in \Sigma^*$,

$$x \in L \iff w_{\max}(x) \text{ is defined},$$

and

$$x \in L \iff (\exists y \in \Sigma^{p(|x|)})[\langle x, y\rangle \in \mathit{Left}[A, p]].$$

Furthermore, for every $x \in \Sigma^*$, the set

$$\{y \in \Sigma^{p(|x|)} \mid \langle x, y\rangle \in \mathit{Left}[A, p]\}$$

equals $\{y \in \Sigma^{p(|x|)} \mid 0^{p(|x|)} \leq y \leq w_{\max}(x)\}$ if $x \in L$ and equals $\emptyset$ otherwise (see Fig. 1.1). More precisely,

$$(\forall x \in \Sigma^*)(\forall y \in \Sigma^{p(|x|)})[\langle x, y\rangle \in \mathit{Left}[A, p] \iff y \in w_{\max}(x)].$$

Also,

$$\begin{aligned}(\forall x \in \Sigma^*)(\forall y, y' \in \Sigma^{p(|x|)})[((\langle x, y\rangle \in \mathit{Left}[A, p]) \wedge (y' < y))\\ \implies \langle x, y'\rangle \in \mathit{Left}[A, p]].\end{aligned} \tag{1.4}$$

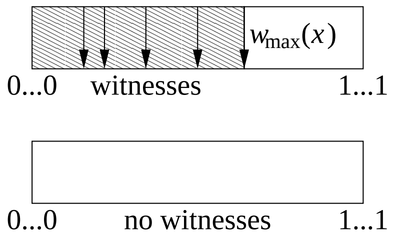

**Fig. 1.1** The left set *Left*$[A,p]$. *Top*: The case when $x \in L$. The arrows above are witnesses for $x \in L$ with respect to $A$ and $p$, $w_{\max}(x)$ is the rightmost arrow, and the shaded area is $\{y \in \Sigma^{p(|x|)} \mid \langle x,y\rangle \in \mathit{Left}[A,p]\}$. *Bottom*: The case when $x \notin L$. $w_{\max}(x)$ is undefined and $\{y \in \Sigma^{p(|x|)} \mid \langle x,y\rangle \in \mathit{Left}[A,p]\} = \emptyset$.

Note that *Left*$[A,p]$ is in NP via the nondeterministic Turing machine that, on input $\langle x,y\rangle$, guesses $w \in \Sigma^{p(|x|)}$, and accepts if

$$(y \in \Sigma^{p(|x|)}) \wedge (y \le w) \wedge (\langle x,y\rangle \in A)$$

and rejects otherwise.

Below, we introduce a useful characterization of $\le^p_{btt}$ reductions. Let $k \ge 1$. A *k-truth-table condition* is a $(k+1)$-tuple $C$ such that the first component of $C$ is a boolean function of arity $k$ and each of the other components of $C$ is a member of $\Sigma^*$. For a $k$-truth-table condition $C = (\alpha, v_1, \ldots, v_k)$, we call $\alpha$ the *truth-table of* $C$, call $\{w \mid (\exists i : 1 \le i \le k)[w = v_i]\}$ the *queries of* $C$, and, for each $i$, $1 \le i \le k$, call $v_i$ the *ith query of* $C$. For a language $D$, we say that the $k$-truth-table condition $(\alpha, v_1, \ldots, v_k)$ is satisfied by $D$ if $\alpha(\chi_D(v_1), \ldots, \chi_D(v_k)) = 1$.

**Proposition 1.11** *Suppose that a language $C$ is $\le^p_{btt}$-reducible to a language $D$. Then there exist an integer $k \ge 1$ and a polynomial-time computable function $f$ from $\Sigma^*$ to the set of all k-truth-table conditions, such that for all $u \in \Sigma^*$,*

$$u \in C \iff f(u) \text{ is satisfied by } D.$$

**Proof of Proposition 1.11** Suppose that, for some $k \ge 1$, a language $C$ is $\le^p_{k\text{-}tt}$-reducible to a language $D$ via $(f_0, B_0)$ such that $f_0 \in \mathrm{FP}$ and $B_0 \in \mathrm{P}$. For all $u \in \Sigma^*$, there exist some $l$, $1 \le l \le k$, and $v_1, \ldots, v_l \in \Sigma^*$, such that

- $f_0(u) = v_1\# \cdots \#v_l\#$, and
- $u \in \mathit{Left}[A,p] \iff u\#\chi_D(v_1) \cdots \chi_D(v_l) \in B_0$,

where $\# \notin \Sigma$. Let $f_1$ be the function from $\Sigma^*$ to $(\Sigma^*\#)^k$ defined for all $u \in \Sigma^*$ by

$$f_1(u) = v_1\# \cdots \#v_l\#^{k+1-l},$$

where $f_0(u) = v_1\# \cdots \# v_l\#$. Define $B_1$ to be the set of all $u\#b_1 \cdots b_k$, $u \in \Sigma^*$, $b_1, \ldots, b_k \in \Sigma$, such that

$$u\#b_1 \cdots b_l \in B_0,$$

where $l$ is the integer such that $f_0(u) \in (\Sigma^*\#)^l$. Since $B_0 \in \mathrm{P}$ and $f_0 \in \mathrm{FP}$, it holds that $B_1 \in \mathrm{P}$ and $f_1 \in \mathrm{FP}$.

Let $\beta = \chi_D(\epsilon)$. Let $u \in \Sigma^*$ and let $f_0(u) = v_1\# \cdots \# v_l\#$. Then

$$u\#\chi_D(v_1) \cdots \chi_D(v_l) \in B_0 \iff u\#\chi_D(v_1) \cdots \chi_D(v_l)\beta^{k-l} \in B_1.$$

Since $f_1(u) = v_1\# \cdots \# v_l\#^{k+1-l}$, we have that the pair $(f_1, B_1)$ witnesses that $C \leq^p_{k\text{-}tt} D$.

Define function $f$ from $\Sigma^*$ to the set of all $k$-truth-table conditions as follows: For each $u \in \Sigma^*$,

$$f(u) = (\alpha, v_1, \ldots, v_k),$$

where $f_1(u) = v_1\# \cdots \# v_k\#$ and $\alpha$ is the boolean function defined for all $b_1, \ldots, b_k \in \{0,1\}^k$ by

$$\alpha(b_1, \ldots, b_k) = \chi_{B_1}(u\#b_1 \cdots b_k).$$

Since $f_1 \in \mathrm{FP}$ and $k$ is a constant, $f \in \mathrm{FP}$. For every $u \in \Sigma^*$,

$$u \in C \iff u\#\chi_D(v_1) \cdots \chi_D(v_k) \in B_1$$

and

$$u\#\chi_D(v_1) \cdots \chi_D(v_k) \in B_1 \iff \alpha(\chi_D(v_1), \ldots, \chi_D(v_k)) = 1,$$

where $f_1(u) = v_1\# \cdots \# v_k\#$. So, for all $u \in \Sigma^*$,

$$u \in C \iff f(u) \text{ is satisfied by } D.$$

Thus, the statement of the proposition holds. ❑ Proposition 1.11

Suppose that NP has a sparse $\leq^p_{btt}$-hard set, $S$. Since $L$ was an arbitrary member of NP, it suffices to prove that $L \in \mathrm{P}$. Since $\mathit{Left}[A,p] \in \mathrm{NP}$, $\mathit{Left}[A,p] \leq^p_{btt} S$. So, by Proposition 1.11, there exist some $k \geq 1$ and $f \in \mathrm{FP}$ such that, for all $u \in \Sigma^*$, $f(u)$ is a $k$-truth-table condition, and

$$u \in \mathit{Left}[A,p] \iff f(u) \text{ is satisfied by } S.$$

In preparation for the remainder of the proof, we define some polynomials. Let $p_1$ be a strictly increasing polynomial such that for all $x \in \Sigma^*$ and $y \in \Sigma^{p(|x|)}$, $|\langle x, y\rangle| \leq p_1(|x|)$. Let $p_2$ be a strictly increasing polynomial such that for all $u \in \Sigma^*$, every query of $f(u)$ has length at most $p_2(|u|)$. Let $p_3$ be a strictly increasing polynomial such that for all integers $n \geq 0$ $||S^{\leq n}|| \leq p_3(n)$. Define $q(n) = p_3(p_2(p_1(n)))$. Then, for all $x \in \Sigma^*$,

$$||\{w \in S \mid (\exists y \in \Sigma^{p(|x|)})[w \text{ is a query of } f(\langle x, y\rangle)]\}|| \leq q(|x|).$$

Define $r(n) = k!2^k(2q(n)+1)^k$. Also, for each $d$, $0 \leq d \leq k$, define $r_d(n) = (k-d)!2^{k-d}(2q(n)+1)^{k-d}$. Note that $r_0 = r$ and $r_k$ is the constant 1 polynomial.

**1.1.2.3 A Polynomial-Time Search Procedure for $w_{\max}$.** To prove that $L \in \mathrm{P}$, we will develop a polynomial-time procedure that, on input $x \in \Sigma^*$, generates a list of strings in $\Sigma^{p(|x|)}$ such that, if $w_{\max}(x)$ is defined then the list is guaranteed to contain $w_{\max}(x)$. The existence of such a procedure implies $L \in \mathrm{P}$ as follows: Let $M$ be a Turing machine that, on input $x \in \Sigma^*$, runs the enumeration procedure to obtain a list of candidates for $w_{\max}(x)$, and accept if the list contains a string $y$ such that $\langle x, y\rangle \in A$ and reject otherwise. Since the enumeration procedure runs in polynomial time and $A \in \mathrm{P}$, $M$ can be made polynomial-time bounded. Since the output list of the enumeration procedure is guaranteed to contain $w_{\max}(x)$ if it is defined, $M$ accepts if $x \in L$. If $x \notin L$, there is no $y \in \Sigma^{p(|x|)}$ such that $\langle x, y\rangle \in A$, so $M$ rejects $x$. Thus, $M$ correctly decides $L$. Hence, $L \in \mathrm{P}$.

In the rest of the proof we will focus on developing such an enumeration procedure. To describe the procedure we need to define some notions.

Let $n \geq 1$ be an integer. Let $I$ be a subset of $\Sigma^n$. We say that $I$ is an *interval over* $\Sigma^n$ if there exist $y, z \in \Sigma^n$ such that

$$y \leq z \text{ and } I = \{u \in \Sigma^n \mid y \leq u \leq z\}.$$

We call $y$ and $z$ respectively the *left end* and the *right end* of $I$, and write $[y, z]$ to denote $I$. Let $I = [u, v]$ and $J = [y, z]$ be two intervals over $\Sigma^n$. We say that $I$ and $J$ are *disjoint* if they are disjoint as sets, i.e., either $v < y$ or $z < u$. If $I$ and $J$ are disjoint and $v < y$, we say that $I$ is *lexicographically smaller than* $J$, and write $I < J$.

Let $x \in \Sigma^*$ and let $\Lambda$ be a set of pairwise disjoint intervals over $\Sigma^n$. We say that $\Lambda$ is *nice* for $x$ if

$$x \in L \Longrightarrow (\exists I \in \Lambda)[w_{\max}(x) \in I].$$

Note that for all $x \in \Sigma^*$

- $\{\,[0^{p(|x|)}, 1^{p(|x|)}]\,\}$ is nice for $x$ regardless of whether $x \in L$, and
- if $x \notin L$, then every set of pairwise disjoint intervals over $\Sigma^{p(|x|)}$ is nice for $x$.

Let $\tau$ be an ordered (possibly empty) list such that, if $\tau$ is not empty then each entry of $\tau$ is of the form $(w, b)$ for some $w \in \Sigma^*$ and $b \in \{0, 1\}$. We call such a list a *hypothesis list*. We say that a hypothesis list $\tau$ is *correct* if every pair $(w, b)$ in the list satisfies $\chi_S(w) = b$. Let $x \in \Sigma^*$, let $\Lambda$ be a set of pairwise disjoint intervals over $\Sigma^{p(|x|)}$, let $\Gamma$ be a subset of $\Lambda$, and let $\tau$ be a hypothesis list. We say that $\Gamma$ is a *refinement of* $\Lambda$ *for* $x$ *under* $\tau$ if

$$((\Lambda \text{ is nice for } x) \wedge (\tau \text{ is correct})) \Longrightarrow \Gamma \text{ is nice for } x.$$

The following fact states some useful properties of refinements.

**Fact 1.12**

1. *If $\Lambda = \Gamma_1 \bigcup \cdots \bigcup \Gamma_m$ and $\Gamma_1', \ldots, \Gamma_k'$ are refinements of $\Gamma_1, \ldots, \Gamma_k$ for $x$ under $\tau$, respectively, then $\Gamma_1' \bigcup \ldots \bigcup \Gamma_k'$ is a refinement of $\Lambda$ for $x$ under $\tau$.*
2. *If $\Theta$ is a refinement of $\Gamma$ for $x$ under $\tau$ and $\Theta'$ is a refinement of $\Theta$ for $x$ under $\tau$, then $\Theta'$ is a refinement of $\Gamma$ for $x$ under $\tau$.*

To generate candidates for $w_{\max}(x)$, starting with the initial value

$$\Lambda = \{\, [0^{p(|x|)}, 1^{p(|x|)}] \,\},$$

we repeat the following two-phase process $p(|x|)$ times.

- **Splitting** Split each interval in $\Lambda$ into upper and lower halves.
- **Culling** If $||\Lambda|| \leq r(|x|)$, skip this phase. If $||\Lambda|| \geq r(|x|) + 1$, do the following: Set $\tau$ to the empty list. Call a subroutine **CULL** on input $(x, \Lambda, \tau)$ to obtain $\Upsilon \subsetneq \Lambda$ that has cardinality less than or equal to $r(|x|)$ and is nice for $x$. Replace $\Lambda$ with $\Upsilon$.

When the the two-phase process has been executed $p(|x|)$ times, each interval in $\Lambda$ has size exactly 1, i.e., is of the form $[u, u]$ for some $u \in \Sigma^{p(|x|)}$. The output of the enumeration procedure is the list of all strings $u \in \Sigma^{p(|x|)}$ such that $[u, u] \in \Lambda$.

Note that if $\Lambda$ is nice for $x$ at the beginning of the splitting phase then it is nice for $x$ at the end of the splitting phase. Since both $p$ and $r$ are polynomials, if **CULL** runs in polynomial time, the entire generation procedure runs in polynomial time. Since **CULL** is guaranteed to output a refinement, if $x \in L$ then there is always one interval in $\Lambda$ that contains $w_{\max}(x)$. So, if $x \in L$, $w_{\max}(x)$ is included in the list of candidates at the end. So, we have only to show that a polynomial-time procedure **CULL** exists that, on input $(x, \Lambda, \tau)$ with $||\Lambda|| \geq r(|x|) + 1$, finds $\Upsilon \subsetneq \Lambda$ having cardinality at most $r(|x|)$ such that $\Upsilon$ is a refinement of $\Lambda$ for $x$ under $\tau$.

For the sake of simplicity, in the following discussion, let $x \in \Sigma^*$ be fixed. Since only splitting and elimination are the operations executed to modify intervals, we can assume that the intervals in $\Lambda$ are pairwise disjoint during the entire enumeration procedure. So, for every pair of distinct intervals, $I$ and $J$, appearing in the input to **CULL**, we will assume that they are disjoint, and thus, either $I < J$ or $I > J$. We also induce a mapping from the set of all interval over $\Sigma^{p(|x|)}$ to the set of all $k$-truth-table conditions. Let $I = [u, v]$ be an interval over $\Sigma^{p(|x|)}$. The image of $I$ induced by $f$, denoted by $f[I]$, is $f(\langle x, u \rangle)$. Let $f[I] = (\alpha, w_1, \ldots, w_k)$. For each $i$, $1 \leq i \leq k$, $Q[I, i]$ to denote $w_i$.

**1.1.2.4 The Structure of the Culling Method.** Each input $(x, \Gamma, \tau)$ to **CULL** is supposed to satisfy the following conditions:

- $\tau$ is a hypothesis list and its length, $|\tau|$, is less than or equal to $k$.

- $\Gamma$ is a set of pairwise disjoint intervals over $\Sigma^{p(|x|)}$ having cardinality strictly greater than $r_{|\tau|}(|x|)$.
- If $|\tau| \geq 1$, then the following condition holds: Let $d = |\tau|$ and $\tau = [(w_1, b_1), \ldots, (w_d, b_d)]$ for some $w_1, \ldots, w_d \in \Sigma^*$ and $b_1, \ldots, b_d \in \{0,1\}$. Then, for all $I \in \Gamma$, there exist $d$ pairwise distinct elements of $\{1, \ldots, k\}$, $j_1, \ldots, j_d$, such that for all $r$, $1 \leq r \leq d$, $Q[I, j_r] = w_r$.

Given an input $(x, \Gamma, \tau)$ that meets this specification, **CULL** may call itself recursively in the case when $|\tau| < k$. The number of recursive call that **CULL** makes is less than or equal to $2(k - |\tau|)(2q(|x|) + 1)$ and the input to each recursive call is a triple of the form $(x, \Gamma', \tau')$ for some $\Gamma' \subseteq \Gamma$ and a hypothesis list $\tau'$ such that $|\tau'| = |\tau| + 1$. Thus, the computation of **CULL** on input $(x, \Gamma, \tau)$ can be viewed as a tree of depth bounded by $k - |\tau|$.

The hypothesis list $\tau$ is used to refine, for each interval $I \in \Gamma$, the $k$-truth-table condition $f[I]$ to a $(k - |\tau|)$-truth-table condition. We denote the refinement of $f[I]$ with respect to $\tau$ by $f_\tau[I]$. Suppose that $\tau = [(w_1, b_1), \ldots, (w_d, b_d)]$ for some $d \geq 1$, $w_1, \ldots, w_d \in \Sigma^*$, and $b_1, \ldots, b_d \in \{0,1\}$. Let $I$ be an arbitrary interval in $\Gamma$ and let $f[I] = (\alpha, v_1, \ldots, v_k)$. Then, $f_\tau[I] = (\beta, v_{j_1}, \ldots, v_{j_{k-d}})$, where $\beta$ and $v_{j_1}, \ldots, v_{j_{k-d}}$ are defined as follows:

- For $s = 1, \ldots, d$, in that order, let $\rho_s$ be the smallest of $r$, $1 \leq r \leq k$, such that $(Q[I, r] = w_s) \wedge (\forall t : 1 \leq t \leq s-1)[r \neq \rho_t]$.
- For every $i$, $1 \leq i \leq k - d$, let $j_i$ be the $i$th smallest element in $\{1, \ldots, k\} - \{\rho_1, \ldots, \rho_d\}$.
- $\beta$ is the boolean function of arity $(k - d)$ that is constructed from $\alpha$ by simultaneously fixing for all $s$, $1 \leq s \leq d$, the argument at position $\rho_s$ to $b_s$.

We will write $\beta_\tau[I]$ to denote the truth-table of $f_\tau[I]$ and $Q_\tau[I]$ to denote the queries of $f_\tau[I]$. Note that if the hypothesis list $\tau$ is correct, then for all $I \in \Gamma$, $f[I]$ is satisfied by $S$ if and only if $f_\tau[I]$ is satisfied by $S$.

Suppose that $|\tau| = k$. Then, for all $I \in \Gamma$, $f_\tau[I]$ is a boolean function of arity 0, i.e., a boolean constant. Since $r_k(|x|) = 1$, **CULL** cannot select more than one interval from $\Gamma$ to generate its output. **CULL** computes $S = \{I \in \Gamma \mid f_\tau[I] = (\text{True})\}$. If $S$ is nonempty, **CULL** selects the largest element in $S$ in lexicographic order; if $S$ is empty, **CULL** outputs the empty set. We claim that the output of **CULL** is a refinement of $\Gamma$ for $x$ under $\tau$. To see why, suppose that $\Gamma$ is nice for $x$ and that the hypothesis list $\tau$ is correct. Then, for every interval $I = [u, v] \in \Gamma$, $w_{\max}(x)$ is less than or equal to $u$ if $\beta_\tau[I] = \text{True}$ and is strictly greater than $u$ otherwise. So, for every interval $I \in \Gamma$, $w_{\max}(x) \notin I$ if either $\beta_\tau[I] = \text{False}$ or $I$ is not the largest element in $S$ in lexicographic order. Thus, it is safe to to select the largest element in $S$ in lexicographic order.

On the other hand, suppose that $|\tau| < k$. Then **CULL** executes two phases, Phases 1 and 2. In Phase 1, **CULL** eliminates intervals from $\Gamma$ so that

$f_\tau[I] \neq f_\tau[J]$ for every pair of distinct intervals $I, J \in \Gamma$. In Phase 2, **CULL** selects from $\Gamma$ disjoint subsets, $\Gamma_1, \ldots, \Gamma_l$, $1 \leq l \leq 2(k - |\tau|)(2q(n) + 1)$, where it is guaranteed that no intervals in the rest of $\Gamma$ contain $w_{\max}(x)$ if $\tau$ is correct. For each of the subsets, **CULL** makes exactly two recursive calls. The output of **CULL**$(x, \Gamma, \tau)$ is the union of the outputs of all the $2l$ recursive calls.

Below we describe the two phases of **CULL**. Let $(x, \Gamma, \tau)$ be an input to **CULL**. Let $d = |\tau|$. Suppose that $0 \leq d \leq k - 1$.

*Phase 1: Making $f_\tau$ unambiguous* **CULL** executes the following:

- While there exist two intervals $I, I' \in \Gamma$ such that $I < I' \in \Gamma$ and $f_\tau[I] = f_\tau[I']$, find the smallest such pair $(I, I')$ in lexicographic order and eliminate $I$ from $\Gamma$.

Let $\Gamma'$ be the set $\Gamma$ when **CULL** quits the loop. We claim that $\Gamma'$ is a refinement of $\Gamma$ for $x$ under $\tau$. To see why, assume that $\tau$ is correct. Suppose that $\Gamma$ contains two intervals $I = [u, v]$ and $I' = [u', v']$ such that $I < I'$ and $f_\tau[I] = f_\tau[I']$. Then $f_\tau[I]$ is satisfied by $S$ if and only if $f_\tau[I']$ is satisfied by $S$. Since $\tau$ is correct, $f_\tau[I]$ is satisfied by $S$ if and only if $f[I]$ is satisfied by $S$. Also, $f_\tau[I']$ is satisfied by $S$ if and only if $f[I']$ is satisfied by $S$. So, $f[I]$ is satisfied by $S$ if and only if $f[I']$ is satisfied by $S$. Then, by equation 1.4, we have

$$w_{\max}(x) \geq u \iff w_{\max}(x) \geq u'.$$

In particular, $w_{\max}(x) \geq u \implies w_{\max}(x) \geq u'$. This implies that either $w_{\max}(x) < u$ or $w_{\max}(x) \geq u'$. Since $u' > v \geq u$, it holds that either $w_{\max}(x) < u$ or $w_{\max}(x) > v$. Thus, $w_{\max}(x) \notin I$. So, $\Gamma - \{I\}$ is a refinement of $\Gamma$ for $x$ under $\tau$. By part 2 of Fact 1.12, each time an interval is eliminate by executing the above algorithm, the resulting set is a refinement of $\Gamma$ for $x$ under $\tau$. Thus, $\Gamma'$ is a refinement of $\Gamma$ for $x$ under $\tau$.

*Phase 2: Refining $\Gamma'$* **CULL** splits $\Gamma'$ into two groups $\Delta_0$ and $\Delta_1$, where for each $b \in \{0, 1\}$,

$$\Delta_b = \{I \in \Gamma' \mid \beta_\tau[I](0, \ldots, 0) = b\}.$$

**CULL** refines $\Delta_0$ and $\Delta_1$ separately.

*Refining $\Delta_0$*: Suppose $\Delta_0$ is nonempty. **CULL** computes an integer $m \geq 1$ and a sequence of intervals $I_1, I_2, \ldots, I_m \in \Delta_0$, $I_1 < I_2 < \cdots < I_m$, as follows:

- $I_1$ is the lexicographic minimum of the intervals in $\Delta_0$.
- For each $t \geq 1$ such that $I_t$ is defined, let $S_{t+1} = \{I \in \Delta_0 \mid (\forall j : 1 \leq j \leq t)$ $[Q_\tau[I] \bigcap Q_\tau[I_j] = \emptyset]\}$. If $S_{t+1} = \emptyset$, then $I_{t+1}$ is undefined. If $S_{t+1} \neq \emptyset$, then $I_{t+1}$ is the lexicographic minimum of the intervals in $S_{t+1}$.
- $m$ is the largest $t$ such that $I_t$ is defined.

Define

$$\Delta'_0 = \begin{cases} \Delta_0 & \text{if } m \leq q(|x|), \\ \{J \in \Delta_0 \mid J < I_{q(|x|)+1}\} & \text{otherwise.} \end{cases}$$

We claim that $\Delta'_0$ is a refinement of $\Delta_0$ for $x$ under $\tau$. If $m \leq q(|x|)$, then $\Delta'_0 = \Delta_0$, so $\Delta'_0$ clearly is a refinement of $\Delta_0$ for $x$ under $\tau$. So, suppose that $m \geq q(n) + 1$, and thus, that $\Delta'_0 \subsetneq \Delta_0$. Suppose that $\tau$ is correct. For each $j$, $1 \leq j \leq m$, let $I_j = [u_j, v_j]$. Assume $u_{q(|x|)+1} \leq w_{\max}(x)$. Then for all $j$, $1 \leq j \leq q(|x|) + 1$, $u_j \leq w_{\max}(x)$. Since $\tau$ is correct, for all $j$, $1 \leq j \leq q(|x|) + 1$, $f_\tau[I_j]$ is satisfied by $S$. For each $j$, $1 \leq j \leq q(|x|) + 1$, $\beta_\tau[I_j](0, \ldots, 0) = 0$, and thus, $Q_\tau[I_j] \bigcap S \neq \emptyset$. The intervals $I_1, \ldots, I_m$ are chosen so that $Q_\tau[I_1], \ldots, Q_\tau[I_m]$ are pairwise disjoint. Thus,

$$||\, S \bigcap \bigcup_{1 \leq j \leq q(|x|)+1} Q_\tau[I_j] \,|| \geq q(|x|) + 1.$$

This is a contradiction, because the number of strings in $S$ that may appear as a query string in $f(\langle x, y\rangle)$ for some $y \in \Sigma^{p(|x|)}$ is at most $q(|x|)$. Thus, $w_{\max}(x) < u_{q(|x|)+1}$. So, all intervals $I \in \Delta_0$ whose left end is greater than or equal to $u_{q(|x|)+1}$ can be safely eliminated from $\Delta_0$. Hence, $\Delta'_0$ is a refinement of $\Delta_0$ for $x$ under $\tau$.

Let $m_0 = \min\{m, q(|x|)\}$ and

$$R = \bigcup_{1 \leq j \leq m_0} Q_\tau[I_j].$$

Let $h = ||R||$. Then $h \leq (k-d)m_0$. For every $I \in \Delta'_0$, there exists some $y \in R$ such that $y \in Q_\tau[I]$. Let $y_1, \ldots, y_h$ be the enumeration of the strings in $R$ in lexicographic order. For each $j$, $1 \leq j \leq h$, let

$$\Theta_j = \{I \mid (I \in \Delta_0) \wedge (\forall s : 1 \leq s \leq j-1)[I \notin \Theta_s] \wedge (y_j \in Q_\tau[I])\}.$$

By the choice of the intervals $I_1, \ldots, I_m$, $\Theta_1, \cdots, \Theta_h$ are all nonempty and $\Delta'_0 = \Theta_1 \bigcup \cdots \bigcup \Theta_h$. For each $j$, $1 \leq j \leq h$, and each $b \in \{0, 1\}$, let $\Theta'_{j,b}$ be the set of intervals that **CULL** on input $(x, \Theta_j, \tau)$ outputs as a refinement of $\Theta_j$ for $x$ under $\tau'$, where $\tau'$ is $\tau$ with the pair $(y_j, b)$ appended at the end. Let $\Upsilon_0 = \bigcup_{1 \leq j \leq h} \bigcup_{b \in \{0,1\}} \Theta'_{j,b}$. By part 1 of Fact 1.12, if **CULL** correctly computes a refinement for all the recursive calls, then $\Upsilon_0$ is a refinement of $\Delta_0$ for $x$ under $\tau$.

*Dividing* $\Delta_1$: Suppose that $\Delta_1 \neq \emptyset$. **CULL** computes an integer $m \geq 1$ and a sequence of intervals $I_1, I_2, \ldots, I_m \in \Delta_0$, $I_1 > I_2 > \cdots > I_m$, as follows:

- $I_1$ is the lexicographic maximum of the intervals in $\Delta_1$.
- For each $t \geq 1$ such that $I_t$ is defined, let $S_{t+1} = \{J \in \Delta_1 \mid (\forall j : 1 \leq j \leq t)$ $[Q_\tau[I] \bigcap Q_\tau[I_j] = \emptyset]\}$. If $S_{t+1} = \emptyset$, then $I_{t+1}$ is undefined. If $S_{t+1} \neq \emptyset$, then $I_{t+1}$ is the lexicographic maximum of the intervals in $\Delta_1$.
- $m$ is the largest $t$ such that $I_t$ is defined.

Define

$$\Delta_1' = \begin{cases} \Delta_1 & \text{if } m \leq q(|x|) + 1, \\ \{J \in \Delta_1' \mid J \geq I_{q(|x|)+1}\} & \text{otherwise.} \end{cases}$$

We claim that $\Delta_1'$ is a refinement of $\Delta_1$ for $x$ under $\tau$. If $m \leq q(|x|) + 1$, then $\Delta_1' = \Delta_1$, so $\Delta_1'$ clearly is a refinement of $\Delta_1$ for $x$ under $\tau$. So, suppose that $m \geq q(n) + 2$, and thus, that $\Delta_1' \subsetneq \Delta_1$. Suppose that $\tau$ is correct. For each $j$, $1 \leq j \leq m$, let $I_j = [u_j, v_j]$. Assume $u_{q(|x|)+1} > w_{\max}(x)$. Then for all $j$, $1 \leq j \leq q(|x|) + 1$, $u_j > w_{\max}(x)$. Since $\tau$ is correct, for all $j$, $1 \leq j \leq q(|x|)+1$, $f_\tau[I_j]$ is not satisfied by $S$. For every $j$, $1 \leq j \leq q(|x|)+1$, $\beta_\tau[I_j](0, \ldots, 0) = 1$. So, for every $j$, $1 \leq j \leq q(|x|) + 1$, $Q_\tau[I_j] \bigcap S \neq \emptyset$. The intervals $I_1, \ldots, I_m$ are chosen so that $Q_\tau[I_1], \ldots, Q_\tau[I_m]$ are pairwise disjoint. Thus,

$$|| S \cap \bigcup_{1 \leq j \leq q(|x|)+1} Q_\tau[I_j] \, || \geq q(|x|) + 1.$$

This is a contradiction. So, $w_{\max}(x) \geq u_{q(|x|)+1}$. This implies that if $\tau$ is correct, then all intervals $I \in \Delta_1$ whose right end is strictly less than $u_{q(|x|)+1}$ can be eliminated from $\Delta_1$. Hence, $\Delta_1'$ is a refinement of $\Delta_1$ for $x$ under $\tau$. The rest of the procedure is essentially the same as that of Case 1. The only difference is that the number of selected intervals is at most $q(|x|) + 1$, and thus, the total number of refinements that are combined to form a refinement of $\Delta_1'$ is at most $2(k-d)(q(|x|)+1)$. Let $\Upsilon_1$ denote the union of the refinements obtained by the recursive calls.

The output $\Upsilon$ of **CULL** is $\Upsilon_0 \bigcup \Upsilon_1$. Suppose that for each $b \in \{0, 1\}$, $\Upsilon_b$ is a refinement of $\Delta_b$ for $x$ under $\tau$. Since $\Delta = \Delta_0 \bigcup \Delta_1$, by part 1 of Fact 1.12, $\Upsilon$ is a refinement of $\Delta$ for $x$ under $\tau$. The total number of recursive calls that **CULL** makes is $2(k - |\tau|)(2q(|x|) + 1)$.

**1.1.2.5 Correctness and Analysis of the Culling Method.** Since the depth of recursion is bounded by $k$, the entire culling procedure runs in time polynomial in $|x|$. The correctness of **CULL** can be proven by induction on the length of the hypothesis list, going down from $k$ to 0. For the base case, let the length be $k$. We already showed that if the length of hypothesis list is $k$ then **CULL** works correctly. For the induction step, let $0 \leq d \leq k - 1$ and suppose that **CULL** works correctly in the case when the length of the hypothesis list is greater than or equal to $d + 1$. Suppose that $(x, \Gamma, \tau)$ is given to **CULL** such that $|\tau| = d$. In each of the recursive calls that **CULL** makes on input $(x, \Gamma, \tau)$, the length of the hypothesis list is $d + 1$, so by the induction hypothesis, the output of each of the recursive calls is correct. This implies that the output of **CULL** on $(x, \Gamma, \tau)$ is a refinement of $\Gamma$.

We also claim that, for every $d$, $0 \leq d \leq k$, the number of intervals in the output of **CULL** in the case when the hypothesis list has length $d$ is at most $r_d(|x|)$. Again, this is proven by induction on $d$, going down from $k$ to 0. The claim certainly holds for the base case, i.e., when $d = k$, since the output of

**CULL** contains at most one interval when the hypothesis list has length $k$ and $r_k$ is the constant 1 function. For the induction step, let $d = d_0$ for some $d_0$, $0 \leq d_0 \leq k-1$, and suppose that the claims holds for all values of $|\tau|$ between $d_0 + 1$ and $k$. Let $(x, \Gamma, \tau)$ be an input to **CULL** such that $|\tau| = d$. The number of recursive calls that **CULL** on input $(x, \Gamma, \tau)$ is at most

$$2((k-d)q(|x|) + 2(k-d)(q(|x|)+1)) = 2(k-d)(2q(|x|)+1).$$

In each of the recursive calls that are made, the length of the hypothesis list is $d+1$, so by the induction hypothesis, the output of each recursive call has at most $r_{d+1}(|x|)$ elements. So, the number of intervals in the output of **CULL** on input $(x, \Gamma, \tau)$ is at most

$$2(k-d)(2q(|x|)+1)r_{d+1}(2q(|x|)+1).$$

This is $r_d(|x|)$. Thus, the claim holds for $d$.

Since $r_0 = r$, the number of intervals in the output of the culling phase is at most $r(|x|)$ as desired. Hence, $L \in \mathrm{P}$. ❑

## 1.2 The Turing Case

In the previous section, we saw that if any sparse set is NP-hard or NP-complete with respect to many-one reductions or even bounded-truth-table reductions, then P = NP. In this section, we ask whether any sparse set can be NP-hard or NP-complete with respect to Turing reductions. Since Turing reductions are more flexible than many-one reductions, this is a potentially weaker assumption than many-one completeness or many-one hardness. In fact, it remains an open question whether these hypotheses imply that P = NP, though there is some relativized evidence suggesting that such a strong conclusion is unlikely. However, one can achieve a weaker collapse of the polynomial hierarchy, and we do so in this section.

Unlike the previous section, the results and proofs for the $\leq_T^p$-completeness and $\leq_T^p$-hardness cases are quite different. We start with the $\leq_T^p$-complete case, which uses what is informally known as a "census" approach. The hallmark of algorithms based on the census approach is that they first obtain for a set (usually a sparse set) the exact number of elements that the set contains up to some given length, and then they exploit that information.

The $\Theta_2^p$ level of the polynomial hierarchy (see Sect. A.4) captures the power of parallel access to NP. In particular, is known to equal the downward closure under truth-table reductions (see Sect. B.1) of NP; this closure is denoted (see the notational shorthands of Sect. B.2) $\mathrm{R}_{tt}^p(\mathrm{NP})$. Thus, Theorem 1.14 proves that if NP has Turing-complete sparse sets, then the entire polynomial hierarchy can be accepted via parallel access to NP. However, the following equality is known to hold.

**Proposition 1.13** $\mathrm{R}^p_{tt}(\mathrm{NP}) = \mathrm{P}^{\mathrm{NP}[\mathcal{O}(\log n)]}$.

We will use the latter form of $\Theta^p_2$ in the proof below.

**Theorem 1.14** *If* NP *has sparse, for* NP $\leq^p_T$*-complete sets then* $\mathrm{PH} = \Theta^p_2$.

**Proof** Let $S$ be a sparse set that is $\leq^p_T$-complete for NP. For each $\ell$, let $p_\ell(n)$ denote $n^\ell + \ell$. Let $j$ be such that $(\forall n)[||S^{\leq n}|| \leq p_j(n)]$. Let $M$ be a deterministic polynomial-time Turing machine such that $\mathrm{SAT} = L(M^S)$; such a machine must exist, as $S$ is Turing-hard for NP. Let $k$ be such that $p_k(n)$ bounds the runtime of $M$ regardless of the oracle $M$ has; without loss of generality, let $M$ be chosen so that such a $k$ exists.

**Pause to Ponder 1.15** *Show why this "without loss of generality claim" holds.*

(Answer sketch for Pause to Ponder 1.15: Given a machine $M$, let the machines $M_1, M_2, \ldots$, be as follows. $M^A_i(x)$ will simulate the action of exactly $p_i(|x|)$ steps of the action of $M^A(x)$, and then will halt in an accepting state if $M^A(x)$ halted and accepted within $p_i(|x|)$ steps, and otherwise will reject. Note that since the overhead involved in simulating one step of machine is at most polynomial, for each $i$, there will exist an $\widehat{i}$ such that for every $A$ it holds that $M^A_i$ runs in time at most $p_{\widehat{i}}(n)$. Furthermore, in each relativized world $A$ in which $M^A$ runs in time at most $p_i$, it will hold that $L(M^A) = L(M^A_i)$. Relatedly, in our proof, given the machine $M$ such that $\mathrm{SAT} = L(M^S)$, we will in light of whatever polynomial-time bound $M^S$ obeys similarly replace $M$ with an appropriate $M_j$ from the list of machines just described.)

Let $L$ be an arbitrary set in $\Sigma^p_2$. Note that, since SAT is NP-complete, it is clear that $\Sigma^p_2 = \mathrm{NP}^{\mathrm{SAT}}$. So, in particular, there is some nondeterministic polynomial-time Turing machine $N$ such that $L = L(N^{\mathrm{SAT}})$. Let $\ell$ be such that $p_\ell(n)$ bounds the nondeterministic runtime of $N$ for all oracles; without loss of generality, let $N$ be chosen such that such an integer exists (see Pause to Ponder 1.15). Note that $L = L(N^{L(M^S)})$.

Define:

$$V = \{0\#1^n\#1^q \mid ||S^{\leq n}|| \geq q\} \bigcup$$
$$\{1\#x\#1^n\#1^q \mid (\exists Z \subseteq S^{\leq n})[||Z|| = q \wedge x \in L(N^{L(M^Z)})]\}.$$

Note that, in light of the fact that $S$ is an NP set, $V \in \mathrm{NP}$.

We now give a $\Theta^p_2$ algorithm that accepts $L$. In particular, we give an algorithm that makes $\mathcal{O}(\log n)$ calls to the NP oracle $V$. Suppose the input to our algorithm is the string $y$.

**Step 1** In $\mathcal{O}(\log |y|)$ sequential queries to $V$ determine $||S^{\leq p_k(p_\ell(|y|))}||$. Our queries will be of the form "$0\#1^{p_k(p_\ell(|y|))}\#1^z$," where we will vary $z$ in a binary search fashion until we home in on the exact value of $||S^{\leq p_k(p_\ell(|y|))}||$. Since $||S^{\leq p_k(p_\ell(|y|))}||$ is bounded by a polynomial in $y$, namely, by $p_j(p_k(p_\ell(|y|)))$, it holds that $\mathcal{O}(\log |y|)$ queries indeed suffice for binary search to pinpoint the census value. Let the census value obtained be denoted $r$.

**Step 2** Ask to $V$ the query $1\#y\#1^{p_\ell(p_k(|y|))}\#1^r$, and accept if and only if the answer is that $1\#y\#1^{p_\ell(p_k(|y|))}\#1^r \in V$.

That completes the statement of the algorithm. Note that the algorithm clearly is a $\Theta_2^p$ algorithm. Furthermore, note that the algorithm indeed accepts $L$. This is simply because, given that Step 1 obtains the true census $r$, the Step 2 query to $V$ can accept only if the actual strings in $S^{\leq p_k(p_\ell(|y|))}$ are guessed (because there are only $r$ strings at those lengths, so if $r$ distinct strings in $S$ have been guessed, then we have guessed all of $S^{\leq p_k(p_\ell(|y|))}$) and, when used by $M$ to generate a prefix of SAT (and note that this prefix is correct on all queries to SAT of length at most $p_\ell(|y|)$, since such queries generate queries to $S$ of length at most $p_k(p_\ell(|y|))$), causes $N$ to accept.

So, since $L$ was an arbitrary set from $\Sigma_2^p$, we have $\Sigma_2^p = \Theta_2^p$. Since $\Theta_2^p$ is closed under complementation, this implies $\Sigma_2^p = \Pi_2^p$, which itself implies $\mathrm{PH} = \Sigma_2^p$. So $\mathrm{PH} = \Sigma_2^p = \Theta_2^p$, completing our proof. ❑

The proof for the case of $\leq_T^p$-hardness is more difficult than the case of $\leq_T^p$-completeness, since the census proof used above crucially uses the fact that the sparse set is in NP. The proof below rolls out a different trick. It extensively uses nondeterminism to guess a set of strings that, while perhaps not the exact elements of a prefix of the sparse NP-$\leq_T^p$-hard set, function just as usefully as such a prefix. The following result is often referred to as the Karp–Lipton Theorem.

**Theorem 1.16 (The Karp–Lipton Theorem)** *If* NP *has sparse* $\leq_T^p$-*hard sets then* $\mathrm{PH} = \mathrm{NP}^{\mathrm{NP}}$.

**Proof** Let $S$ be a sparse set that is $\leq_T^p$-hard for NP. For each $\ell$, let $p_\ell(n)$ denote $n^\ell + \ell$. Let $j$ be such that $(\forall n)[||S^{\leq n}|| \leq p_j(n)]$. Let $M$ be a deterministic polynomial-time Turing machine such that $\mathrm{SAT} = L(M^S)$; such a machine must exist, as $S$ is Turing-hard for NP. Let $k$ be such that $p_k(n)$ bounds the runtime of $M$ for all oracles; without loss of generality, let $M$ be such that such an integer exists (see Pause to Ponder 1.15).

Let $L$ be an arbitrary set in $\Sigma_3^p$. We will give a $\Sigma_2^p$ algorithm for $L$. This establishes that $\Sigma_2^p = \Sigma_3^p$, which itself implies that $\mathrm{PH} = \Sigma_2^p$, thus proving the theorem.

Note that, since SAT is NP-complete, it is clear that $\Sigma_3^p = \mathrm{NP}^{\mathrm{NP}^{\mathrm{SAT}}}$. So, in particular, there are two nondeterministic polynomial-time Turing machines $N_1$ and $N_2$ such that $L(N_1^{L(N_2^{\mathrm{SAT}})}) = L$. Let $\ell$ be such that $p_\ell(n)$ bounds the nondeterministic runtime of $N_1$ for all oracles, and such that $p_\ell(n)$ also bounds the nondeterministic runtime of $N_2$ for all oracles; without loss of generality, let $N_1$ and $N_2$ be such that such an integer exists (see Pause to Ponder 1.15).

Define

$V_0 = \{0\#1^n\#S' \mid (\exists z \in (\Sigma^*)^{\leq n})[$(a) $z$ is not a well-formed formula and $M^{S'}(z)$ accepts; or (b) $z$ is a well-formed formula free variables and either (b1) $M^{S'}(z)$ accepts and $z \notin$ SAT or (b2) $M^{S'}(z)$ rejects and $z \in$ SAT; or (c) $z$ is a well-formed formula variables $z_1, z_2, \ldots$ and it is *not* the case that: $M^{S'}(z)$ accepts if and only if

$$(M^{S'}(z[z_1 = \text{True}]) \text{ accepts } \vee M^{S'}(z[z_1 = \text{False}]) \text{ accepts})]\,\},$$

where, as defined earlier in this chapter, $z[...]$ denotes $z$ with the indicated variables assigned as noted.

$V_1 = \{1\#S'\#z \mid z \in L(N_2^{L(M^{S'})})\}$.

$V = V_0 \bigcup V_1$.

Note that $V \in$ NP. Informally, $V$ functions as follows. The $0\#1^n\#S'$ strings in $V$ determine whether given sets of strings "work" as sparse oracles that (on all instances of length at most $n$) allow $M$ to correctly accept SAT. Or, more exactly, it checks if a given set *fails* to simulate SAT correctly. Of course, the fact that $S$ is a sparse Turing-hard set for NP ensures that there are some such sets $S'$ that *do* simulate SAT correctly in this sense; however, it is possible that sets $S'$ other than prefixes of $S$ may also happen to simulate SAT correctly in this sense. The $1\#\cdots$ part of $V$ takes a set of strings that happens to simulate SAT as just described, and uses them, in concert with $M$, to simulate SAT.

We now give a $\text{NP}^{\text{NP}}$ algorithm that accepts $L$. In particular, we give an $\text{NP}^V$ algorithm. Suppose the input to our algorithm is the string $y$. Note that the longest possible query to SAT that $N_2$ will make on queries $N_1$ asks to its oracle during the run of $N_1^{L(N_2^{\text{SAT}})}(y)$ is $p_\ell(p_\ell(|y|))$. Note also that $M$, on inputs of length $p_\ell(p_\ell(|y|))$, asks its oracle only questions of length at most $p_k(p_\ell(p_\ell(|y|)))$. And finally, note that there is some sparse oracle$U$ such that $L(M^{(U^{\leq p_k(p_\ell(p_\ell(|y|)))})}) = \text{SAT}^{\leq p_\ell(p_\ell(|y|))}$; for example, the set $S$ is such an oracle.

**Step 1** Nondeterministically guess a set $S' \subseteq (\Sigma^*)^{\leq p_k(p_\ell(p_\ell(|y|)))}$ satisfying $||S'|| \leq p_j(p_k(p_\ell(p_\ell(|y|))))$. If $0\#1^{p_k(p_\ell(p_\ell(|y|)))}\#S' \in V$ then reject. Otherwise, go to Step 2.

**Step 2** Simulate the action of $N_1(y)$ except that, each time $N_1(y)$ makes a query $z$ to its $L(N_2^{\text{SAT}})$ oracle, ask instead the query $1\#S'\#z$ to $V$.

That completes the description of the algorithm. Note that the algorithm we have given is clearly a $\Sigma_2^p$ algorithm. Furthermore, note that the algorithm indeed accepts $L$. This is simply because Step 1 obtains a valid set of strings $S'$ that either are $S^{\leq p_k(p_\ell(p_\ell(|y|)))}$, or that, in the action of machine $M$, function just as well as $S^{\leq p_k(p_\ell(p_\ell(|y|)))}$ in simulating SAT. That is, we obtain a set of strings $S'$ such that

$$\mathrm{SAT}^{\leq p_k(p_\ell(p_\ell(|y|)))} = \left(L(M^{S'})\right)^{\leq p_k(p_\ell(p_\ell(|y|)))}.$$

This correct prefix of SAT is just long enough that it ensures that Step 2 of the algorithm will correctly simulate $N_2^{\mathrm{SAT}}$. ❑

This result has been extended in various ways. One very useful strengthening that we will refer to later is that one can replace the base-level NP machine with an expected-polynomial-time probabilistic machine. (The parenthetical equivalence comment in the theorem is based on the well-known fact, which is an easy exercise that we commend to the reader, that $\mathrm{R}_T^p(\{S \mid S \text{ is sparse}\}) = \mathrm{P/poly}$.)

**Theorem 1.17** *If* NP *has sparse* $\leq_T^p$*-hard sets (equivalently, if* NP $\subseteq$ P/poly*), then* PH = ZPP$^{\mathrm{NP}}$*.*

## 1.3 The Case of Merely Putting Sparse Sets in NP − P: The Hartmanis–Immerman–Sewelson Encoding

In the previous sections we studied whether classes such as NP had complete or hard sparse sets with respect to various reductions. We know from Theorem 1.7, for example, that there is no NP-complete sparse set unless P = NP.

In this section, we ask whether there is any sparse set in NP − P. Note in particular that we are not here asking whether there is any sparse set in NP − P that is NP-complete; by Theorem 1.7 the answer to that question is clearly "no." We here are instead merely asking whether any sparse set in NP can be so complex as to lack deterministic polynomial-time algorithms.

Before approaching this question, let us motivate it from a quite different direction. One central goal of computational complexity theory is to understand the relative power of different complexity classes. For example, is deterministic polynomial-time computation a strictly weaker notion than nondeterministic polynomial-time computation, that is P ≠ NP? The ideal results along such lines are results collapsing complexity classes or separating complexity classes.

In fact, complexity theorists have achieved a number of just such results—outright, nontrivial complexity class collapses and separations. For example, the strong exponential hierarchy—an exponential-time analog of the polynomial hierarchy—is known to collapse, and for very small space bounds a space analog of the polynomial hierarchy is known to truly separate. The famous time and space hierarchy theorems also provide unconditional separation results. Unfortunately, not one such result is known to be useful in the realm between P and PSPACE. It remains plausible that P = PSPACE and it remains plausible that P ≠ PSPACE.

Given this disturbingly critical gap in our knowledge of the power of complexity classes between P and PSPACE—exactly the computational realm in which most interesting real-world problems fall—what can be done? One approach is, instead of directly trying to separate or collapse the classes, to link the many open questions that exist within this range. The philosophy behind this is very similar to the philosophy behind NP-completeness theory. There, we still do not know whether NP-complete problems have polynomial-time algorithms. However, we do know that, since all NP-complete problems are $\leq_m^p$-interreducible, they stand or fall together; either all have polynomial-time algorithms or none do.

In the context of complexity classes between P and PSPACE, the goal along these lines would be to link together as many open questions as possible, ideally with "if and only if" links. It turns out that it often is easy to "upward translate" collapses, that is, to show that if small classes collapse then (seemingly) larger classes collapse. The truly difficult challenge is to "downward translate" equalities: to show that if larger classes collapse then (seemingly) smaller classes collapse.

In this section we study a famous downward translation that partially links the P = NP question to the collapse of exponential-time classes. In particular, we will ask whether the collapse of deterministic and nondeterministic exponential time implies any collapse in the classes between P and PSPACE. The really blockbuster result to seek would be a theorem establishing that $\mathrm{E} = \mathrm{NE} \implies \mathrm{P} = \mathrm{NP}$. However, it is an open question whether this can be established. What is known, and what we will here prove, is the following theorem, which says that the collapse of NE to E is equivalent to putting into P all sparse sets in NP.

**Theorem 1.18** *The following are equivalent:*

1. E = NE.
2. NP − P *contains no sparse sets.*
3. NP − P *contains no tally sets.*

**Proof** Part 2 clearly implies part 3, as every tally set is sparse. The theorem follows immediately from this fact, and from Lemmas 1.19 and Lemma 1.21. ❑

The following easy lemma shows that if no tally sets exist in NP − P, then NE collapses to E.

**Lemma 1.19** *If* NP − P *contains no tally sets then* E = NE.

**Proof** Let $L$ be some set in NE, and assume that NP − P contains no tally sets. Let $N$ be a nondeterministic exponential-time machine such that $L(N) = L$. Define $L' = \{1^k \mid (\exists x \in L)[k = (1x)_{\mathrm{bin}}]\}$, where for any string (over {0,1}) $z$ the expression $(z)_{\mathrm{bin}}$ denotes the integer the string represents when viewed as a binary integer, e.g., $(1000)_{\mathrm{bin}} = 8$.

Note that $L' \in \mathrm{NP}$, since the following algorithm accepts $L'$. On input $y$, reject if $y$ is not of the form $1^k$ for some $k > 0$. Otherwise $y = 1^k$ for some

$k > 0$. Write $k$ in binary, and let $s$ be the binary of representation of $k$ to the right of, and not including, its leftmost one, viewed as a binary string. Call this string $w$. (If $k = 1$, then $w = \epsilon$.) Simulate $N(w)$ (thus accepting if and only if $N(w)$ accepts). Though $N$ is an exponential-time machine, the length of $w$ is logarithmic in the length of $y$, and thus the overall nondeterministic runtime of this algorithm is, for some constant $c$, at most $\mathcal{O}(2^{c \log n})$.

Thus, $L' \in$ NP. However, by hypothesis this implies that $L'$ is in P. So, let $M$ be a deterministic polynomial-time machine such that $L(M) = L'$. We now describe a deterministic exponential-time algorithm for $L$. On input $a$, compute the string $b = 1^{(1a)_{\mathrm{bin}}}$, and then simulate $M(b)$, accepting if and only if $M(b)$ accepts. Since $M$ is a polynomial-time machine and $|b| \leq 2^{|a|}$, the number of steps that $M(b)$ runs is $(2^n)^c = 2^{cn}$. As the overhead of doing the simulation and the cost of obtaining $b$ from $a$ are also at most exponential in the input's length, clearly our algorithm for $L$ is a deterministic exponential-time algorithm. Thus, $L \in$ E, which completes our proof. ❑

Finally, we must prove that if E = NE then all sparse NP sets in fact are in P.

**Pause to Ponder 1.20** *As an easy warmup exercise, try to prove the simpler claim: If* E = NE *then all tally sets in* NP *are in* P*.*

A sketch of the solution to Pause to Ponder 1.20 is as follows. If $L$ is a tally set in NP, then let $L' = \{x \mid (x$ is 0 or $x$ is a binary string of nonzero length with no leading zeros) and $1^{(x)_{\mathrm{bin}}} \in L\}$. It is not hard to see that $L' \in$ NE. Thus by assumption $L' \in$ E, and thus there is a natural P algorithm for $L$, namely, the algorithm that on input $a$ rejects if $a \notin 1^*$ and that if $a = 1^k$ writes $k$ as 0 if $k = 0$ and otherwise as $k$ in binary with no leading zeros, and then simulates the E algorithm for $L'$ on this string. This concludes the proof sketch for Pause to Ponder 1.20.

However, recall that we must prove the stronger result that if E = NE then all sparse NP sets are in P. Historically, the result in Pause to Ponder 1.20 was established many years before this stronger result. If one looks carefully at the proof just sketched for Pause to Ponder 1.20, it is clear that the proof, even though it works well for the stated case (tally sets), breaks down catastrophically for sparse sets. The reason it fails to apply to sparse sets is that the proof is crucially using the fact that the length of a string in a tally set fully determines the string. In a sparse set there may be a polynomial number of strings at a given length. Thus the very, very simple encoding used in the proof sketch of Pause to Ponder 1.20, namely, representing tally strings by their length, is not powerful enough to distinguish same-length strings in a sparse set.

To do so, we will define a special "Hartmanis–Immerman–Sewelson encoding set" that crushes the information of any sparse NP set into extremely bite-sized morsels from which the membership of the set can be easily reconstructed. In fact, the encoding manages to put all useful information about a sparse NP set's length $n$ strings into length-$\mathcal{O}(\log n)$ instances of the encoding set—and yet maintain the easy decodability that is required to establish the following lemma.

**Lemma 1.21** *If* E = NE *then* NP − P *contains no sparse sets.*

**Proof** Let $L$ be some sparse NP set, and assume that E = NE. Given that $L$ is sparse, there is some polynomial, call it $q$, such that $(\forall n)[||L^{=n}|| \leq q(n)]$. Define the following encoding set:

$$\begin{aligned} L' = {} & \{0\#n\#k \mid ||L^{=n}|| \geq k\} \bigcup \\ & \{1\#n\#c\#i\#j \mid (\exists z_1, z_2, \ldots, z_c \in L^{=n})[z_1 <_{\text{lex}} z_2 <_{\text{lex}} \\ & \cdots <_{\text{lex}} z_c \wedge \text{ the } j\text{th bit of } z_i \text{ is } 1]\}. \end{aligned}$$

Since $L \in$ NP, it is clear that $L' \in$ NE. So by our assumption, $L' \in$ E.

We will now use the fact that $L' \in$ E to give a P algorithm for $L$. Our P algorithm for $L$ works as follows. On input $x$, let $n = |x|$. Query $L'$ to determine which of the following list of polynomially many strings belong to $L'$: $0\#n\#0, 0\#n\#1, 0\#n\#2, \ldots, 0\#n\#q(n)$, where here and later in the proof the actual calls to $L'$ will for the numerical arguments (the $n$'s, $c$, $i$, $j$, and $k$ of the definition of $L'$) be coded as (and within $L'$ will be decoded back from) binary strings. Given these answers, set

$$c = \max\{k \mid 0 \leq k \leq q(n) \wedge 0\#n\#k \in L'\}.$$

Note that $c = ||L^{=n}||$. Now ask the following questions to $L'$:

$$\begin{gathered} 1\#n\#c\#1\#1, 1\#n\#c\#1\#2, \ldots, 1\#n\#c\#1\#n, \\ 1\#n\#c\#2\#1, 1\#n\#c\#2\#2, \ldots, 1\#n\#c\#2\#n, \\ \cdots, \\ 1\#n\#c\#c\#1, 1\#n\#c\#c\#2, \ldots, 1\#n\#c\#c\#n. \end{gathered}$$

The answers to this list of polynomially many questions to $L'$ give, bit by bit, the entire set of length $n$ strings in $L$. If our input, $x$, belongs to this set then accept, and otherwise reject. Though $L' \in$ E, each of the polynomially many queries asked to $L'$ (during the execution of the algorithm just described) is of length $\mathcal{O}(\log n)$. Thus, it is clear that the algorithm is indeed a polynomial-time algorithm. ❑

Theorem 1.18 was but the start of a long line of research into downward translations. Though the full line of research is beyond the scope of this book, and is still a subject of active research and advances, it is now known that the query hierarchy to NP itself shows certain downward translations of equality. In particular, the following result says that if one and two questions to $\Sigma_k^p$ yield the same power, then the polynomial hierarchy collapses not just to $\mathrm{P}^{\Sigma_k^p[1]}$ but in fact even to $\Sigma_k^p$ itself.

**Theorem 1.22** *Let* $k > 1$. $\Sigma_k^p = \Pi_k^p$ *if and only if* $\mathrm{P}^{\Sigma_k^p[1]} = \mathrm{P}^{\Sigma_k^p[2]}$.

## 1.4 OPEN ISSUE: Does the Disjunctive Case Hold?

Theorem 1.7 shows that NP lacks sparse $\leq_m^p$-complete sets unless P = NP. Does this result generalize to bounded-truth-table, conjunctive-truth-table, and disjunctive-truth-table reductions: $\leq_{btt}^p$, $\leq_{ctt}^p$, and $\leq_{dtt}^p$?

Theorem 1.10 already generalizes Theorem 1.7 to the case of $\leq_{btt}^p$-hardness. Using the left set technique it is also easy to generalize the result to the case of $\leq_{ctt}^p$-hardness: If NP has $\leq_{ctt}^p$-hard sparse sets then P = NP.

The case of $\leq_{dtt}^p$-hardness remains very much open.

**Open Question 1.23** *Can one prove: If* NP *has* $\leq_{dtt}^p$*-hard sparse sets, then* P = NP*?*

However, it is known that proving the statement would be quite strong. In particular, the following somewhat surprising relationship is known.

**Proposition 1.24** *Every set that* $\leq_{btt}^p$*-reduces to a sparse set in fact* $\leq_{dtt}^p$*-reduces to some sparse set.*

Thus, if one could prove the result of Open Question 1.23, that result would immediately imply Theorem 1.10.

## 1.5 Bibliographic Notes

Theorem 1.2 (which is often referred to as "Berman's Theorem") and Corollary 1.3 are due to Berman [Ber78], and the proof approach yields the analog of these results not just for the tally sets but also for the P-capturable [CGH$^+$89] sets, i.e., the sets having sparse P supersets. Theorem 1.4 and Corollary 1.5 are due to Fortune [For79]. Among the results that followed soon after the work of Fortune were advances by Ukkonen [Ukk83], Yap [Yap83], and Yesha [Yes83].

Theorems 1.7 (which is known as "Mahaney's Theorem") and Theorem 1.9 are due to Mahaney [Mah82]. The historical motivation for his work is sometimes forgotten, but is quite interesting. The famous Berman–Hartmanis Isomorphism Conjecture [BH77], which conjectures that all NP-complete sets are polynomial-time isomorphic, was relatively new at the time. Since no dense set (such as the NP-complete set SAT) can be polynomial-time isomorphic to any sparse set, the existence of a sparse NP-complete set would immediately show the conjecture to be false. Thus, Mahaney's work was a way of showing the pointlessness of that line of attack on the Berman–Hartmanis Isomorphism Conjecture (see [HM80]): if such a set exists, then P = NP, in which case the Berman–Hartmanis Isomorphism Conjecture fails trivially anyway.

Theorem 1.10 (which is often referred to as "the Ogiwara–Watanabe Theorem") is due to Ogiwara and Watanabe ([OW91], see also [HL94]), who introduced the left set technique in the study of sparse complete sets. Somewhat more general results than Theorem 1.10 are now known to hold, due to work of Homer and Longpré [HL94], Arvind et al. [AHH$^+$93], and Glaßer ([Gla00], see also [GH00]). Our presentation is based on the work of Homer and Longpré [HL94].

These results are part of a rich exploration of sparse completeness results, with respect to many complexity classes and many types of reductions, that started with Berman's work and that continues to this day. Numerous surveys of general or specialized work on sparse complete sets exist [HM80,Mah86, Mah89,You92,HOW92,vMO97,CO97,GH00].

Regarding the relativized evidence mentioned on page 18, Immerman and Mahaney [IM89] have shown that there are relativized worlds in which NP has sparse Turing-hard sets yet $\mathrm{P} \neq \mathrm{NP}$. Arvind et al. [AHH$^+$93] extended this to show that there are relativized worlds in which NP has sparse Turing-complete sets yet the boolean hierarchy [CGH$^+$88] does not collapse, and Kadin [Kad89] showed that there are relativized worlds in which NP has sparse Turing-complete sets yet some $\Theta_2^p$ languages cannot be accepted via P machines making $o(\log n)$ sequential queries to NP.

Proposition 1.13 is due to Hemachandra [Hem89]. Theorem 1.14 is due to Kadin [Kad89]. Theorem 1.16 is due to Karp and Lipton [KL80], and we prove it here using a nice, intuitive, alternate proof line first suggested by Hopcroft ([Hop81], see also [BBS86]). The fact that $\mathrm{R}_T^p(\{S \mid S \text{ is sparse}\}) =$ P/poly appears in a paper by Berman and Hartmanis [BH77], where it is attributed to Meyer. Theorem 1.17 is due to Köbler and Watanabe ([KW98], see also [KS97]) based on work of Bshouty et al. [BCKT94,BCG$^+$96].

Cai [Cai01] has proven that the "symmetric alternation" version of $\mathrm{NP}^{\mathrm{NP}}$, a class known as $\mathrm{S}_2^p$ [Can96,RS98], satisfies $\mathrm{S}_2^p \subseteq \mathrm{ZPP}^{\mathrm{NP}}$. In light of Sengupta's observation (see the discussion in [Cai01]) that a Hopcroft-approach proof of Theorem 1.16 in fact can be used to conclude that $\mathrm{S}_2^p = \mathrm{PH}$, Cai's result says that Sengupta's collapse to $\mathrm{S}_2^p$ is at least as strong as, and potentially is even stronger than, that of Theorem 1.16.

The collapse of the strong exponential-time hierarchy referred to near the start of Sect. 1.3 is due to Hemachandra [Hem89], and the separation of small-space alternation hierarchies referred to in Sect. 1.3 is due, independently (see [Wag93]), to Liśkiewicz and Reischuk [LR96,LR97], von Braunmühl, Gengler, and Rettinger [vBGR93,vBGR94], and Geffert [Gef94]. The study of time and space hierarchy theorems is a rich one, and dates back to the pathbreaking work of Hartmanis, Lewis, and Stearns [HS65,LSH65,SHL65].

Lemma 1.19 and the result stated in Pause to Ponder 1.20—and thus the equivalence of parts 1 and 3 of Theorem 1.18—are due to Book [Boo74b].

The Hartmanis–Immerman–Sewelson Encoding, and in particular Lemma 1.21 (and thus in effect the equivalence of parts 1 and 2 of Theorem 1.18), was first employed by Hartmanis [Har83]. The technique was further explored by Hartmanis, Immerman, and Sewelson ([HIS85], see also [All91,AW90]). Even the Hartmanis–Immerman–Sewelson Encoding has its limitations. Though it does prove that E = NE if and only if NP − P has sparse sets, it does not seem to suffice if we shift our attention from NP (and its exponential analog, NE) to UP, FewP, $\oplus$P, ZPP, RP, and BPP (and their respective exponential analogs). In fact, the Buhrman–Hemaspaandra–Longpré Encoding [BHL95], a different, later encoding encoding based on some elegant combinatorics [EFF82,EFF85,NW94], has been used by Rao, Rothe, and Watanabe [RRW94] to show that the $\oplus$P and "FewP" analogs of Theorem 1.18 do hold. That is, they for example prove that E equals, "$\oplus$E," the exponential-time analog of $\oplus$P, if and only if $\oplus$P − P contains sparse sets. In contrast with this, Hartmanis, Immerman, and Sewelson showed that there are oracles relative to which the coNP analog of Theorem 1.18 fails. Hemaspaandra and Jha [HJ95a] showed that there are oracles relative to which the the ZPP, R, and BPP analogs of Theorem 1.18 fail, and they also showed that even for the NP case the "immunity" analog of Theorem 1.18 fails. Allender and Wilson [All91,AW90] have shown that one claimed "supersparse" analog of Theorem 1.18 fails, but that in fact certain analogs can be obtained. For some classes, for example UP, it remains an open question whether an analog of Theorem 1.18 can be obtained.

The proof of Lemma 1.21 proves something a bit stronger than what the lemma itself asserts. In particular, the proof makes it clear that: If E = NE then every sparse NP set is P-printable (i.e., there is an algorithm that on input $1^n$ prints *all* length $n$ strings in the given sparse NP set). This stronger claim is due to Hartmanis and Yesha [HY84].

Regarding downward translations of equality relating exponential-time classes to smaller classes, we mention that a truly striking result of Babai, Fortnow, Nisan, and Wigderson [BFNW93] shows: If a certain exponential-time analog of the polynomial hierarchy collapses to E, then P = BPP. This is not quite a "downward" translation of equality, as it is not clear in general whether BPP $\subseteq$ E (though that does hold under the hypothesis of their theorem, due to the conclusion of their theorem), but this result nonetheless represents a remarkable connection between exponential-time classes and polynomial-time classes.

A $\Sigma_k^p = \Pi_k^p$ conclusion, and thus a downward translation of equality for classes in the NP query hierarchy, was reached by Hemaspaandra, Hemaspaandra, and Hempel [HHH99a] for the case $k > 2$. Buhrman and Fortnow [BF99] extended their result to the $k = 2$ case. These appear as Theorem 1.22. Downward translations of equality are known not just for the 1-vs-2 query case but also for the $j$-vs-$(j+1)$ query case ([HHH99a,HHH99b], see also [HHH98]), but they involve equality translations within the bounded-access-to-$\Sigma_k^p$ hierarchies, rather than equality translations to $\Sigma_k^p = \Pi_k^p$.

In contrast with the difficulty of proving downward translations of equality, upward translations of equality are so routine that they are considered by most people to be "normal behavior." For example, it is well-known for almost all pairs of levels of the polynomial hierarchy that if the levels are equal then the polynomial hierarchy collapses. This result dates back to the seminal work of Meyer and Stockmeyer, who defined the polynomial hierarchy [MS72,Sto76]. The fascinating exception is whether $\Theta_k^p = \Delta_k^p$ implies that the polynomial hierarchy collapses. Despite intense study, this issue remains open—see the discussion in [Hem94,HRZ95].

Nonetheless, it is far from clear that the view that upward translation of equality is a “normal” behavior of complexity classes is a itself a correct view. It does tend to hold within the polynomial hierarchy, which is where the intuition of most complexity theorists has been forged, but the polynomial hierarchy has many peculiar properties that even its close cousins lack (stemming from such features as the fact that the set of all polynomials happens to be closed under composition—in contrast to the set of logarithmic functions or the set of exponential functions), and thus is far from an ideal basis for predictions. In fact, Hartmanis, Immerman, and Sewelson [HIS85] and Impagliazzo and Tardos [IT89] have shown that there is an oracle relative to which upward translation of equality fails in an exponential-time analog of the polynomial hierarchy, and Hemaspaandra and Jha ([HJ95a], see also [BG98]) have shown the same for the limited-nondeterminism hierarchy of NP—the so-called $\beta$ hierarchy of Kintala and Fischer [KF80] and Díaz and Torán [DT90].

Proposition 1.24 is due to Allender et al. [AHOW92]. Though Open Question 1.23, with its P = NP conjectured conclusion from the assumption of there being sparse $\leq_{dtt}^{p}$-hard sets for NP, indeed remains open, some consequences—though not as strong as P = NP—are known to follow from the existence of sparse $\leq_{dtt}^{p}$-hard sets for NP. In particular, Cai, Naik, and Sivakumar have shown that if NP has sparse $\leq_{dtt}^{p}$-hard sets then RP = NP [CNS96]. It is also known that if NP has sparse $\leq_{dtt}^{p}$-hard sets, then there is a polynomial-time set $A$ such that every unsatisfiable boolean formula belongs to $\overline{A}$ and every boolean formula that has exactly one satisfying assignment belongs to $A$ (implicit in [CNS95], as noted by van Melkebeek [vM97] and Sivakumar [Siv00]). That is, $A$ correctly solves SAT on all inputs having at most one solution, but might not accept some satisfiable formulas having more than one satisfying assignment.