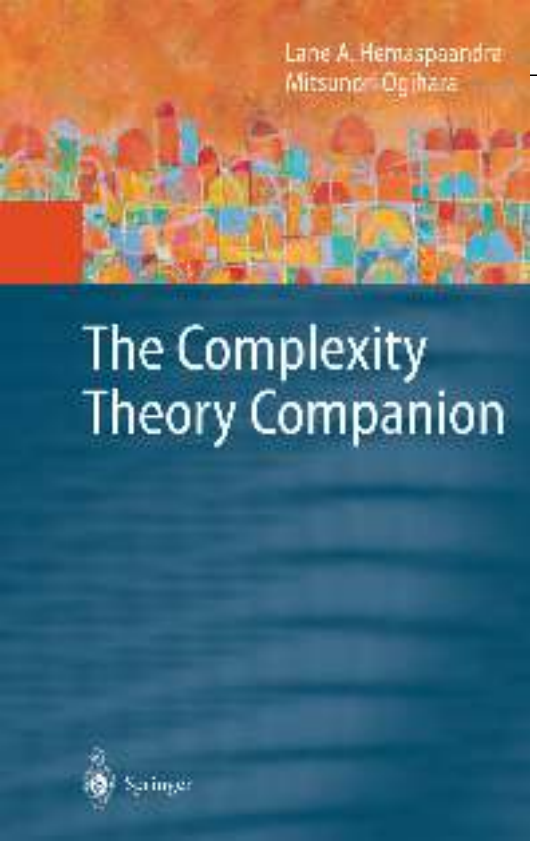
Lane A. Hemaspaandra
Mitsunori Ogihara
The Complexity
Theory Companion
Springer

Texts in Theoretical Computer Science
An EATCS Series

Springer-Verlag Berlin Heidelberg GmbH

Lane A. Hemaspaandra • Mitsunori Ogihara

The Complexity Theory Companion

With 43 Figures

Springer

Authors

Prof. Dr. Lane A. Hemaspaandra
Prof. Dr. Mitsunori Ogihara
Department of Computer Science
Rochester, NY 14627
USA
{lane,ogihara}@cs.rochester.edu

Series Editors

Prof. Dr. Wilfried Brauer
Institut für Informatik
Technische Universität München
Arcisstrasse 21, 80333 München, Germany
brauer@informatik.tu-muenchen.de

Prof. Dr. Grzegorz Rozenberg
Leiden Institute of Advanced Computer Science
University of Leiden
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
rozenber@liacs.nl

Prof. Dr. Arto Salomaa
Data City
Turku Centre for Computer Science
20 500 Turku, Finland
asalomaa@utu.fi

Library of Congress Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Hemaspaandra, Lane A.:
The complexity theory companion/Lane A. Hemaspaandra; Mitsunori Ogihara.
Berlin; Heidelberg; New York; Barcelona; Hong Kong; London; Milan;
Paris; Tokyo: Springer, 2002
(Texts in theoretical computer science)

ACM Computing Classification (1998): F.1, F.2.2, F.4

ISBN 978-3-642-08591-7 ISBN 978-3-662-04880-1 (eBook)
DOI 10.1007/978-3-662-04880-1

Originally published by Springer-Verlag Berlin Heidelberg New York in 2002.
Softcover reprint of the hardcover 1st edition 2002

Cover Design: KünkelLopka, Heidelberg
Typesetting: Camera-ready by the authors
Printed on acid-free paper SPIN 10725529 45/3142SR – 5 4 3 2 1 0

This book is dedicated to our families—the best of companions.

Preface

Invitation

Secret 1 *Algorithms are at the heart of complexity theory.*

That is the dark secret of complexity theory. It is recognized by complexity theorists, but would be literally incredible to most others. In this book, we hope to make this secret credible. In fact, the real secret is even more dramatic.

Secret 2 *Simple algorithms are at the heart of complexity theory.*

A corollary of Secret 2 is that every practitioner of computer science or student of computer science already possesses the ability required to understand, enjoy, and employ complexity theory.

We realize that these secrets fly in the face of conventional wisdom. Most people view complexity theory as an arcane realm populated by pointy-hatted (if not indeed pointy-bearded) sorcerers slinging snublions of recursion theory with wands of combinatorics, while chanting incantations involving complexity classes whose very names contain hundreds of characters and sear the tongues of mere mortals. This stereotype has sprung up in part due to the small amount of esoteric research that fits this bill, but the stereotype is more strongly attributable to the failure of complexity theorists to communicate in expository forums the central role that algorithms play in complexity theory.

Throughout this book—from the tree-pruning and interval pruning algorithms that shape the first chapter to the query-model procedures that dominate the last chapter—we will see that proofs in complexity theory usually employ algorithms as their central tools. In fact, to more clearly highlight the role of algorithmic techniques in complexity theory, *this book is organized by technique rather than by topic*. That is, in contrast to the organization of other books on complexity theory, each chapter of this book focuses on one technique—what it is, and what results and applications it has yielded.

The most thrilling times in complexity theory are when a new technique is introduced and sweeps like fire over the field. In addition to highlighting the centrality of algorithms in the proof arsenal of complexity theory, we feel that our technique-based approach more vividly conveys to the reader the flavor and excitement of such conflagrations. We invite the reader to come

with us as we present nine techniques, usually simple and algorithmic, that burned away some of the field's ignorance and helped form the landscape of modern complexity theory.

Usage

We intend this book as a companion for students and professionals who seek an accessible, algorithmically oriented, research-centered, up-to-date guide to some of the most interesting techniques of complexity theory. The authors and their colleague Joel Seiferas have test-driven the book's approach in two different courses at the University of Rochester. We have used this technique-based approach in Rochester's one-semester basic complexity theory course, which is taken by all first year computer science graduate students and also by those undergraduates specializing or specially interested in theoretical computer science, and in our second course on complexity theory, which is taken by all second-year graduate students as their theory "breadth" course.

We found in both these course settings that the technique-based approach allowed us to impart to students a significant amount of the feel and experience of complexity theory research and led to more student interest and involvement than occurred in earlier course incarnations using other texts. We expect that this will not only benefit the complexity theory students in the courses, but will also help all the course's students become prepared to do work that is theoretically aware, informed, and well-grounded.

At times, we stop the flow of a proof or discussion with a "Pause to Ponder." These are places at which we encourage the reader to pause for a moment and find his or her own solution to the issue raised. Even an unsuccessful attempt to craft a solution will usually make the proof/discussion that follows clearer and more valuable, as the reader will better understand the challenges posed by the hurdle that the proof/discussion overcomes.

With some exceptions due to result dependencies, the non-appendix chapters are generally ordered to put the easier chapters near the start of the book and the more demanding chapters near the end of the book.

Acknowledgments

We are extraordinarily indebted to the following people, who proofread one or more chapters, for their invaluable help, suggestions, corrections, and insights: Eric Allender, Russell Bent, Alina Beygelzimer, Matthew Brunell, Samuel Chen, Yin-He Chung, Louis Dorard, Gregory Goldstein, Fred Green, Ulrich Hertrampf, Chris Homan, Gabriel Istrate, Jason Ku, David Lagakos, Andrew Lenard, Tao Li, Ioan Macarie, Proshanto Mukherji, Kenneth Regan, William Scherer III, Alan Selman, D. Sivakumar, Howard Straubing, Robert Swier,

Mayur Thakur, Jonathan Turner, Jaroslav Tykal, Leen Troncelliti, Dieter van Melkebeek, Heribert Vollmer, Julie Zhong, and Marius Zimand. We also thank the many other people who have helped us with advice, comments, corrections, literature pointers, most-recent-version information, and suggestions: Andris Ambainis, Vikraman Arvind, Richard Beigel, Nate Blaylock, Daniel Bovet, Jin-Yi Cai, Diane Cass, Stephen Fenner, Lance Fortnow, William Gasarch, Ulrich Gerber, Oded Goldreich, Juris Hartmanis, Edith Hemaspaandra, Paul Iterdi, Sven Kosub, Richard Lipton, Alexis Maciel, Wolfgang Merkle, Christos Papadimitriou, Thanos Papathanasiou, Eduardo Pinheiro, Robert Rettinger, Jörg Rothe, Alex Samorodnitsky, Marcus Schaefer, Michael Schear, Uwe Schöning, Joel Seiferas, Samik Sengupta, Carl Smith, Scott Stones, Madhu Sudan, Chunjiang Tang, Jun Tarui, Thomas Thierauf, Luca Trevisan, Chris Umans, Osamu Watanabe, Chee Yap, and Stathis Zachos. Any remaining errors are the responsibility of the authors.

We thank our thesis advisors, Juris Hartmanis and Kojiro Kobayashi; their insightful, dedicated, joyous approach to research has been a continuing inspiration to us.

We appreciate the grants—NSF-CCR-8957604, NSF-INT-9116781/JSPS ENGR-207, NSF-CCR-9322513, NSF-INT-9513368/DAAD-315-PRO-fo-ab, NSF-CCR-9701911, NSF CCR-9725021, NSF-INT-9726724, NSF-INT-9815095/DAAD-315-PPP-gü-ab, NSF-DUE-9980943, DARPA-F30602-98-2-0133, NIA-R01-AG18231—that have supported our research programs during the planning and writing of this book.

For generous advice, help, and support, we thank the Springer-Verlag series editors and staff, namely, Wilfried Brauer, Grzegorz Rozenberg, Arto Salomaa, Alfred Hofmann, Frank Holzwarth, Ingeborg Mayer, Sherryl Snudell, and Hans Wössner. Our own department's technical and secretarial staff—especially Jill Forster, Elaine Heberle, and Jan Roche—was invaluable in keeping our computers up and our manuscript copied and circulating, and we much appreciate their help.

We are grateful to those colleagues—Peter van Emde Boas, Harald Hempel, Jörg Rothe, Alan Selman, Seinosuke Toda, Leen Torenvliet, Heribert Vollmer, Klaus Wagner, Osamu Watanabe, and Gerd Wechsung—who have generously hosted our research visits during the planning and writing of this book, and to the many colleagues, as cited in the Bibliographic Notes sections, who have collaborated with us on the research described in some sections of this book.

Above all, we thank Edith, Ellen, Eric, and Erica for their advice, love, and support.

Lane A. Hemaspaandra

Mitsunori Ogihara

Rochester, NY
October 2001

Contents

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 $\leq^{\mathrm{p}}_{\mathrm{m}}$-complete, or even $\leq^{\mathrm{p}}_{\mathrm{btt}}$-hard, for NP unless P = NP. (The appendices contain definitions of and introductions to the reduction types, such as $\leq^{\mathrm{p}}_{\mathrm{m}}$ and $\leq^{\mathrm{p}}_{\mathrm{btt}}$, and the complexity classes, such as P and NP, that are used in this book.)

Section 1.3 studies whether NP can have $\leq^{\mathrm{p}}_{\mathrm{T}}$-complete or $\leq^{\mathrm{p}}_{\mathrm{T}}$-hard sparse sets. $\mathrm{P}^{\mathrm{NP}[O(\log n)]}$ denotes the class of languages that can be accepted by some deterministic polynomial-time Turing machine allowed at most $O(\log n)$ 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 $\leq^{\mathrm{p}}_{\mathrm{T}}$-complete for NP unless the polynomial hierarchy collapses to $\mathrm{P}^{\mathrm{NP}[O(\log n)]}$, and that sparse sets cannot be $\leq^{\mathrm{p}}_{\mathrm{T}}$-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 $\leq^{\mathrm{p}}_{\mathrm{m}}$-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.

L. A. Hemaspaandra et al., The Complexity Theory Companion

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 "$(\forall x)\,[R(x) \Rightarrow S(x)]$," 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 $\leq_{\mathrm{tt}}^{\mathrm{p}}$ 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 $\leq_{\mathrm{m}}^{\mathrm{p}}$ hard for NP, it follows that $\mathrm{SAT} \leq_{\mathrm{m}}^{\mathrm{p}} T$. 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

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

Our algorithm has stages numbered $0, 1, \ldots, v_m + 1$. 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 C be the collection

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

Step 2 Set C' to be $\emptyset$.

Step 3 For each formula f in C (in arbitrary order) do:

1. Compute $g(f)$.
2. If $g(f) \in T$ and for no formula $h \in C'$ does $g(f) = g(h)$, then add f to C'.

End Stage i [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

$$\begin{array}{c}\text{The collection, } C, \text{ contains some satisfiable formula} \\ \Longleftrightarrow \\ F \text{ is satisfiable,}\end{array} \tag{1.2}$$

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

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

We 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,

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

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 $h \in C'$ has $g(f) = g(h)$.

Both these ways, (a) and (b), of dropping formulas are harmless. Recall that $\text{SAT} \leq_m^p T$ via function g, and so if $f \in \text{SAT}$ then $g(f) \in T$. However, regarding (a), $T \subseteq 1^*$ so if $g(f) \notin 1^*$ then $g(f) \notin T$, and so $f \notin \text{SAT}$. 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 $\text{SAT} \leq_m^p T$ via reduction g—we know that

$$f \in \text{SAT} \Longrightarrow g(f) \in T$$

and

$$h \in \text{SAT} \Longleftrightarrow g(h) \in T.$$

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 specifications 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 $\leq^p_m$-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 k, let $p_k(n)$ denote the polynomial $n^k + k$. Let d be such that $(\forall n)[|S^{\leq n}| \leq p_d(n)]$.[2] Since SAT $\subset$ NP, it follows that $\overline{\text{SAT}} \leq_m^p S$. Let g be a deterministic polynomial-time function many-one reducing $\overline{SAT}$ to S. Let k be 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 consisting 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 C be the collection

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

Step 2 Set C' to be $\emptyset$.

Step 3 For each formula f in C (in arbitrary order) do:

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

[2] The $|S^{\leq n}|$, as opposed to the $\|S^{\leq n}\|$ 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 $\|S^{\leq n}\|$ 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 C' contains at least $p_g(p_g(|F|))+1$ elements, stop and immediately declare that $F \in SAT$. (The reason for the $p_g(p_g(|F|))+1$ figure will be made clear below.)

End Stage i [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 $F \in SAT$, and otherwise we declare $F \notin SAT$.

Why does this algorithm work? Let n represent $|F|$. Since $p_g(p_g(n))+1$ 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 $p_g(p_g(n))+1$ distinct strings being mapped to by the non-primed 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 $p_g(n)$ since that is the longest string that reduction g can output on input s of size at most n. However, there are only $p_g(p_g(n))$ strings in $S^{\leq p_g(n)}$. As usual, Σ denotes our alphabet, and as usual we take $\Sigma = \{0,1\}$. So since the formulas in our collection map to $p_g(p_g(n))+1$ distinct strings in $(\Sigma^*)^{\leq p_g(n)}$, *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, $\overline{SAT}$ reduces to S via g, as 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 1.6

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 C' contains $p_g(p_g(n))+1$ formulas, no more elements are added to 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, two 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.*[7]*)*

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-$\leq^{\mathrm{p}}_{\mathrm{tt}}$-hardness rather than merely for NP-$\leq^{\mathrm{p}}_{\mathrm{tt}}$-completeness.)

Theorem 1.9 *NP has sparse $\leq^{\mathrm{p}}_{\mathrm{m}}$-hard sets if and only if NP has sparse $\leq^{\mathrm{p}}_{\mathrm{m}}$-complete sets.*

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

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

The rough intuition here is that S' is almost $f(\mathrm{SAT})$, except to make the proof work in 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 verification 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 $f(x) = z$

[7] The analogous proof would merely be able to claim that if the tree were getting "bushy," there would be at least one *satisfiable* formula among the collection. This says nothing regarding whether some other formula might be satisfiable. Thus, even if the set 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 $\mathrm{P} = \mathrm{NP}$.

Theorem 1.10 *If there is a sparse set that* P = NP *that is* $\leq_{\mathrm{btt}}^p$*-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

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

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

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

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,

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

Also,

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

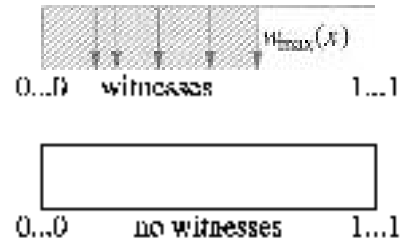

Fig. 1.1 The left set $Left(A,p)$. *Top:* The case when $x \in L$. The arrows above are witnesses for $x \in L$ with respect to A and p. $w_{max}(x)$ is the rightmost arrow, and the shaded area is $\{y \in \Sigma^{p(|x|)} \mid (x,y) \in Left(A,p)\}$. *Bottom:* The case when $x \notin L$. $w_{max}(x)$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

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

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

and rejects otherwise.

Below, we introduce a useful characterization of $\leq^p_{btt}$ reductions. Let $k \geq 1$. 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 Σ^*. For a k-truth-table condition $C = (\alpha, v_1, \ldots, v_k)$, we call α the *truth-table of* C, call $\{v_i \mid (\exists i : 1 \leq i \leq k)[w = v_i]\}$ the *queries of* C, and, for each i, $1 \leq i \leq k$, 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 $\leq^p_{btt}$-reducible to a language D. Then there exist an integer $k \geq 1$ and a polynomial-time computable function f from Σ^* to the set of all k-truth-table conditions, such that for all $w \in \Sigma^*$,*

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

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

- $f_0(w) = v_1\# \cdots \#v_l\#$, and
- $w \in Left(A,p) \iff w\#\chi_D(v_1)\cdots\chi_D(v_l) \in B_0$,

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

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

where $f_0(w) = w\#^1 \cdots \#^n$. Define B_1 to be the set of all $w\#b_1 \cdots b_k$, $w \in \Sigma^*$, $b_1, \ldots, b_k \in \Sigma$, such that

$$w\#^l b_1 \cdots b_k \in D_0,$$

where l is the integer such that $f_0(w) \in (D')^l$. 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 $\beta' = \chi_D(\varepsilon)$. Let $w \in \Sigma^*$ and let $f_0(w) = w_1\# \cdots \#w_n\#$. Then

$$w\#\chi_D(w_1) \cdots \chi_D(w_i) \in D_0 \iff w\#\chi_D(w_1) \cdots \chi_D(w_i)\beta'^{k-1} \in B_1.$$

Since $f_1(w) = w_1\# \cdots \#w_i\#^{k+1-i}$, we have that the pair (f_1, B_1) witnesses that $C \leq^p_{k\text{-tt}} D$.

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

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

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

$$\alpha(b_1, \ldots, b_k) = \chi_D(w\#b_1 \cdots b_k).$$

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

$$w \in C \iff w\#\chi_D(v_1) \cdots \chi_D(v_k) \in B$$

and

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

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

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

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

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

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

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 $y \in \Sigma^{p(|x|)}$, $|(x, y)| \leq p_1(|x|)$. Let p_2 be a strictly increasing polynomial such that for all $w \in \Sigma^*$, every query of $f(w)$ has length at most $p_2(|w|)$. 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 $x \in \Sigma^*$,

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

Define $r(n) = k!3^k(2q(n)+1)^k$. 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 $L \in P$, 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 $L \in P$ 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 accepts if the list contains a string y such that $(x, y) \in A$ and rejects otherwise. Since the enumeration procedure runs in polynomial time and $A \in P$, 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 $(x, y) \in A$, so M rejects x. Thus, M correctly decides L. Hence, $L \in P$.

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 Σ^n. We say that I is an *interval over* Σ^n if there exist $y, z \in \Sigma^n$ such that

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

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 Σ^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 A be a set of pairwise disjoint intervals over Σ^n. We say that A is *nice for* x if

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

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

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

The following fact states some useful properties of enumerators.