Learning from Different Teachers and Imperfect Queries

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Mārtiņš Kriķis

Dissertation Director: Dr. Dana Angluin

May 1998

Copyright © 1998 by Mārtiņš Kriķis

All Rights Reserved

To my wife Līga

Acknowledgements

First and foremost I am very grateful to my advisor Dana Angluin for all the many ways in which she has helped me finish this thesis. She has always been a great source of inspiration, always known how to pull something useful out of the most worthless looking result, and has always had numerous ideas for moving on with our research. Her incredible competence in the field and permanent readyness to answer all my questions have been undeserved luxuries to me. There have been many times when her encouragement and admirable ability to raise the spirits of others have put me back on the right track. She has viewed all my sudden shifts of interest as necessary digressions, and her support has never depended on my progress. All the work presented in this dissertation is the result of joint research with her. There are simply not enough words to express my gratitude to the best advisor that a graduate student could have.

Next, I wish to thank James Aspnes, Sally Goldman and Jeffery Westbrook for serving on my thesis committee and providing me with valuable and detailed comments that have helped improve this dissertation tremendously. James Aspnes and Jeffery Westbrook have also been my favorite consultants in many areas of Computer Science, no matter how unrelated to my research. I would also like to thank Robert Sloan and György Turán for their help with Part 2 of this dissertation and

for changing some of my writing habits.

Many people at Yale have expressed interest in my work, attended my talks, suggested directions for improvement and read drafts of papers. I cannot thank them all for there are too many to list, but am especially grateful to Michael Fischer, Stanley Eisenstat, Richard Beigel, Laszlo Lovasz and Lenore Zuck. Michael Fischer has also helped me with some of the mysteries of LaTeX.

For some time during my studies I was supported by the National Science Foundation grant CCR-9213881, for which I am very grateful. I am also indebted to my undergraduate advisor Rūsiņš Freivalds for introducing me to the exciting area of Inductive Inference and Computational Learning Theory in general; without his help and encouragement, my education at Yale would have never come about.

I have made many good friends during my years in New Haven and they have helped me stay sane and finish this work, perhaps unknowingly. Special thanks go to Imants and Vera Platais, John and Marti Peterson, and to Bommadevara Nagendrasrinivas. I would also like to thank my new friends and colleagues at Kenan Systems Corporation for their support and interest in my progress with this dissertation.

I wish to thank my parents, my brother and my grandmother for their love and encouragement, and for never losing hope in me. I have not been fair to them by going so far from home and visiting so seldomly. Finally, I would like to thank my wife Līga, who has patiently endured my long hours in front of the computer, and put up with everything in the best imaginable way. Her love and support have provided me with the environment so important for this endeavor. I dedicate this thesis to her as a small token of my love and gratitude.

Contents

Overview **1**

I Learning from Teachers that are Different **8**

1 Introduction **9**

2 Definitions and Notation **16**

2.1 Programming Systems and Complexity Measures 16

2.2 Black Boxes 19

2.3 b-relatedness 23

2.4 Learners and Teachers 28

2.5 Reliability and Proofness Properties 32

3 Teaching the Fast Learners **39**

4 Is the Teacher Important? **48**

5 Classifying the Successful Learners **54**

6 Is Everything Easy? **64**

7 Building More Powerful Teachers and Learners **82**

8 Conclusion **100**

Appendix to Part 1 **106**

Proof of Lemma 2 106

Proof of Weakened Corollary 10 110

Constructive Proof of Corollary 10 115

II Learning with Malicious Errors in Queries **122**

9 Introduction **123**

9.1 Query Models 123

9.2 Previous Work 127

10 Preliminaries **132**

10.1 Concepts and Concept Classes 132

10.2 Queries 133

10.3 Monotone DNF Formulas 137

11 Malicious Membership Queries **140**

11.1 The Algorithm 140

11.2 Analysis of LearnMonDNF 143

12 Finite Exceptions **151**

12.1 Exceptions 151

12.2 Examples and Lemmas 153

12.3 The Learning Algorithm 157

12.4 Analysis of the Algorithm 161

13 Exceptions and Errors **165**

14 Discussion and Open Problems **172**

III Learning with Random Errors in Queries **175**

15 Introduction and Definitions **176**

16 The Algorithm and the Game **182**

17 Probabilities Associated with the Game Events **189**

18 Bounds on Probabilities **201**

18.1 Upper Bound 201
18.2 Lower Bound for Success 202
18.3 The New Game 207
18.4 Missing Thin Bottles 211
18.5 The New Game is Not Easier 216
18.6 Getting Through the Top Levels 220
18.7 Succeeding on Lower Levels 225
18.8 Lower Bound for Complete Success 226

19 Comparison to a Simpler Algorithm and Conclusions 228

Appendix 232

Bibliography 238

List of Figures

2.1 Algorithm for the universal function 21

3.1 Learner's algorithm 43

5.1 Algorithm for computing the bound $b(x, s)$ 58

6.1 Subroutine TMDefine 66

6.2 Subroutine QuestionDefine 66

6.3 Algorithm DefineBB 67

6.4 Algorithm for the composition function 78

7.1 Algorithm UnionLearner 84

7.2 Subroutine HypoDecide 85

7.3 Algorithm UnionTeacher 86

7.4 Subroutine HypoDecidePrime 97

8.1 Algorithm UBB 107

8.2 Algorithm for constructing the bad Black Box 113

8.3 Algorithm that gives the values of the bad Black Box 116

10.1 Algorithm for learning monotone DNF from EQ's and standard MQ's 139

10.2 Subroutine Reduce 139

11.1 Subroutine CheckedMQ 141

11.2 Subroutine Reduce 141

11.3 Algorithm for learning monotone DNF from EQ's and MMQ's . . . 142

12.1 A decision tree corresponding to the formula f_- 156

12.2 Subroutine GetExceptions 158

12.3 Subroutine TheFunction 159

12.4 Subroutine Reduce 159

12.5 Algorithm for learning monotone DNF with finite exceptions 160

13.1 Subroutine NewMQ 167

13.2 The block of code replacing "$x = \mathrm{EQ}(h)$" or "**Output** h" 168

16.1 Algorithm RandomReduce 183

16.2 States, transitions and their probabilities in the game 188

17.1 The states of level f and transitions between them 191

18.1 States, transitions and their probabilities in the new game 210

List of Tables

17.1 Probability of Complete Success in the Game for $p = 0.001$ 199

17.2 Probability of Complete Success in the Game for $p = 0.0001$ 199

17.3 Probability of Complete Success in the Game for $p = 0.00001$ 200

Overview

This thesis belongs to the field of Computational Learning Theory, a part of Machine Learning (and Artificial Intelligence) research that specializes in using mathematical models to reason about such phenomena as learning, inference, induction, adaptation, prediction, self-discovery and others. The ultimate goal of Machine Learning is to build "intelligent" computer systems that "learn" from experience. These are systems that can re-program themselves, are capable of continuously acquiring knowledge, can change their behavior in order to better comply with a changing environment, can analyze data patterns, predict future events, or possess some other skills that are widespread among living beings but not machines. Such learning systems are urgently needed for a variety of practical applications. They would be invaluable in mobile robots, diagnostic equipment, text translation, language interpretation or handwriting recognition systems, and would be of immense help in automatic software development, various computer aided design tools and many other much more widespread tools, for example, text editors. When artificial learning systems become a reality they are going to find uses that are hard to imagine today. In order to build these complex devices we need to have clear mathematical models that specify how each system interacts with the world, what kind of knowledge it accumulates, what constitutes adequate change in the systems' behavior and many

other issues. It is very important to have at least a crude estimate as to how successful a system can be before actually wasting time and resources on an idea that may have a poor foundation.

Learning is a rather complicated process and up to this day no clear definition of it exists. Consequently, it is very hard to model it formally and a great number of assumptions and simplifications are to be expected. Computational Learning Theory research focuses on inventing and analyzing all kinds of models that demonstrate how learning can be done. None of these models give details for building a thinking machine, still a very distant goal. Instead they typically present some very concrete learning problem and either give an algorithm that solves it or prove that it is not solvable. There is no clear winner among the various models in existence, since many are far too different to compare in any way and there is no measure of goodness for these models. Each model seems to have its advantages and disadvantages and, not surprisingly, new models get introduced all the time. Usually it happens with the hope that they will better address a weakness of some other model or will overcome a difficulty observed when applying a theoretically plausible algorithm to a real-life problem. Other reasons for inventing a new model may range from simply exhibiting other methods of algorithmic learning to imitating various interesting features observed in human (or animal) learning and development. Every little peculiarity has the potential to influence how intelligent machines are eventually built.

This dissertation focuses on what has been the main emphasis of the field, inductive learning from examples. Such learning problems are in general modeled as consisting of a learning algorithm, an environment that it interacts with, and of success criteria for the algorithm. Typically, a learning algorithm is required to determine some general rule, given some kind of examples of this rule. This framework

has a wealth of formal models that differ from each other in many ways. Quite obviously there are many possible types of rules to infer: geometric objects, recursive functions, boolean formulas, to mention just a few. The environment can be modeled in many different ways, for example, it could be an adversarial process, a random process, or a process specifically designed to make the task of the learning algorithm simpler. There are many more variations in the success criteria or the environment for the learner that make each model unique, for example:

- There may or may not be a limit on the time the algorithm may spend for inferring the rule;
- The algorithm may be considered as reached its objective if it keeps outputting rules that are logically equivalent to the original, or it may be required to start outputting the same rule, or it may be required to stop in order to signify that the final answer has been produced;
- The output of the algorithm may or may not have to be in the same representation as the original rule was envisioned in;
- The algorithm may be required to output a rule logically equivalent to the original one, or it could be allowed to output any rule that is similar "enough" to it;
- The algorithm may or may not influence what examples of the rule it is given;
- The examples the algorithm sees may or may not be absolutely correct;
- The algorithm may or may not have prior knowledge of how the examples are distributed or what broader class of rules the unknown rule belongs to;

- The algorithm may or may not have access to other sources of input, such as, for example, a procedure for testing intermediate hypothesis.

Models where the environment of the learning algorithm is modeled by a random or adversarial process are customarily called learning models. They tend to focus mostly on studying the learning algorithm itself. Such models are good for studying the average or worst-case behavior of the algorithm. There are other models where the environment is specifically designed to help the algorithm. These are usually called teaching models, especially if there is another nontrivial algorithm generating examples for the learning algorithm or interacting with it in some way. In this case both the learning and the teaching algorithms are studied as well as the interaction between them. The advantage of teaching models is that they allow harder target rules to be learned or let the learning algorithm be more efficient in its use of resources, such as time or memory space. Some models cannot be easily classified one way or another, and both names seem to apply to them. This thesis considers several well known learning and teaching models. These models are extended to form new ones that either better represent certain trends observable in human learning or that overcome certain problems associated with applying in practice learning algorithms developed for other models.

Among the most common problem with implementations of learning algorithms is the possibility of errors that are not accounted for in the model. Algorithms that are designed for error-free models often cannot tolerate a single error. That is, instead of "almost learning" the target rule or taking longer to do that, they may in fact output a wrong rule, decide that there is no rule explaining all the examples, run forever, crash, or exhibit some other kind of unwanted behavior. A related problem with many algorithms is their high degree of specialization. That is, they are capable

of learning rules only from certain, often very narrow, classes of possible target rules. If the rule to be learned is not exactly in the class that the algorithm is tuned to work with, it need not do anything reasonable. Both these problems are considered in this work and in some sense compared.

A different kind of weakness that often troubles teaching models is the "coding issue". It arises when the environment or the teaching algorithm is too helpful, making learning trivial. For example, suppose that we have an algorithm that learns some rule from the examples of that rule, but only because there is an example that encodes the rule within itself. Or perhaps several examples together provide encoding of a formal representation of the rule. Can we call this a learning algorithm and present the model as a reasonable model exhibiting learning? The general feeling among the researchers is that the situation above should be called "coding" (or cheating) but not learning. Coding is the way we program our computers to do something today. It is a good approach for many tasks that computer systems do, but seems infeasible for certain things that we would like them to do, for example, replacing human experts in medical diagnosis. There are many other problems that we cannot imagine to have solutions that can be simply programmed. This in fact is one of the reasons that researchers have started to analyze the phenomenon of learning.

If we try to draw comparisons to human learning, it is also quite obvious that coding has no place there. Why are textbooks so full of examples and exercises as opposed to carefully thought out ready recipes for every skill that we may want to acquire? Of course, many of our skills are such that no one can imagine them described in any way on paper, but there are other skills that do have concise algorithms behind them. Take, for example, elementary arithmetic. We could come up with a set

of algorithms that specify how to perform addition, subtraction, multiplication, and division on decimal numbers. But we would not be comfortable learning these operations by memorizing the algorithms. We don't learn well by being "programmed" to do something and much rather prefer to "program" ourselves.

Now that we've concluded that coding is not learning, how does this affect a teaching model in which coding is possible? Usually, it means that extra steps need to be taken to eliminate coding as a way to find the target rule, or else the model will not be considered interesting. Many originally simple models have been augmented with various features that eliminate the possibility of coding. This may lessen the overall appeal of a model, especially if the anti-coding measures seem artificial and don't resemble any difficulties encountered in real-life learning situations. This work gives some reasons why coding is not a natural way of human learning and introduces a new model where there are no precautions against coding—it simply does not help. The new model reflects well another interesting characteristic of humans—the uncertainty about one's own capabilities. It is very hard to precisely define what constitutes learning and there is no artificial learning to experiment with, therefore it is very important and natural to build models that attempt to express what we can observe.

This work is organized into three nearly independent parts. Each part contains enough motivation, introduction and discussion of related previous work to be read as a separate unit. The first part takes a recursion-theoretic approach to learning and is concerned with issues such as coding and lack of knowledge about intrinsic capabilities. The models introduced in this part also exhibit what impact a teacher can have on learning. Parts 2 and 3 of this thesis belong to a different area of Computational Learning Theory that studies "concept" learning and puts a lot of

emphasis on efficiency. Every learning algorithm is required to learn the target rule in time that is polynomial in various parameters that depend on the concrete problem. Besides time, it is similarly constrained in the amount of memory it may use and the number of examples that it may consume. The focus in the second part of this dissertation is mostly on the effect of errors in the examples to the target rule, but the issue of designing algorithms that learn "broader than customary" classes of target rules is also considered. A relation between overcoming errors in examples and learning broader classes of rules is given. The third part is somewhat related to the second, as it explores a particular learning algorithm from Part 2 in a slightly relaxed model. The errors in examples as considered in the second part are "malicious", i.e., spread across the examples in the worst possible way. In Part 3 they are "random" and thus possibly easier to cope with. The goal of this part, however, is to determine whether the concrete algorithm developed for malicious errors can be adapted to work well with random errors and what would be its advantages over a trivial algorithm that works only for the related error-free model.

Many of the results proved in this dissertation are based on algorithms given in the accompanying figures. The algorithms are written in a syntax that resembles the "C" programming language. It has convenient flow control statements and it allows one to code very compactly. It may, however, look very obscure to somebody who is not familiar with its syntax. Therefore, a reference to a "classic" C book and explanations of most of the constructs used in this thesis are given in the Appendix, page 232.

Part I

Learning from Teachers that are Different

Chapter 1

Introduction

One of the goals of Computational Learning Theory is to find new *learning* and *teaching* models that better reflect different issues of human learning or help in building "intelligent" computer systems. Despite the large number and variety of existing models, the quest for better ones is far from over.

Learning Theory originally started with learning models (no teaching) but it was soon discovered that in order to make broader classes of concepts learnable, we need to help the learner somehow [38]. From then on both learning and teaching models have been considered. Teaching models are those where the environment is specifically designed to help the learning algorithm. Often it involves another algorithm, called the teaching algorithm (or simply, teacher) that interacts with the learning algorithm (or learner).

As soon as too much freedom is given to the teacher, a well known problem arises—the possibility of "outright coding". By this we mean a protocol where the teacher transmits (using an encoding via examples or other information) a represen-

tation of the target concept to the learner. This, most authors agree, does not seem to involve learning in an interesting sense, and is usually prevented by deliberately chosen features of the model. Different models address the issue of outright coding in different ways. Some require that the teacher be capable of teaching every *consistent* learner (one that only outputs hypotheses consistent with the examples seen) [22, 35]. Others require the learner to learn from every teacher supplying correct examples [3]. Still others introduce adversaries that add other examples or misorder the existing ones, before passing them from the teacher to the learner [21, 25]. Others require that the learner is never fooled into converging to a wrong concept [26]. There are other ways in which different authors deal with coding, but everybody tries to prevent it.

In short, it is a common belief that passing a description of the target concept from the teacher to the learner does not reflect learning. It may be a good way to program a computer, but there is more going on in the process that we call "learning". Unfortunately, many models need to include artificial features to cope with the possibility of coding. Therefore, we introduce yet another model of teaching. In this model we do not take any precautions to avoid coding. Both the teacher and the learner are welcome to "cheat" all they want. The good thing is that it does not help to do so. More precisely speaking, the negative results that we have for this model discourage the idea that there could be coding going on. At the same time, the positive ones are achieved in a straightforward way with no intention of outright coding.

One of the main goals of our model is to try to reflect in a more direct way the reasons why outright coding is not a common mode of human learning. Our analysis of this issue involves two related ideas:

1. The "hardware and software environment" differs substantially from one person to the next, and

2. In many human endeavors, simulation does not give a feasible solution.

In support of the first point, the interconnections of neurons in human neural circuitry appears to involve a degree of randomness, as well as influence by external stimuli, that suggests that a neuron-by-neuron isomorphism of two human nervous systems is extremely unlikely. The fantasy of somehow transferring the patterns of neural activation from one person to another thus allowing the latter to experience exactly what the former is experiencing is, to put it mildly, improbable. Taking the neural level as "hardware" and the pattern of activations as "software," our hardware and software are just different—one person cannot meaningfully run somebody else's program.

Of course, there are other, more abstract, levels of human cognitive functioning where we could make analogies to hardware and program, but it does not seem plausible that there is any level for which the analogy of transferring a program between two identical computers is very accurate. Interestingly, these observations also apply to the problem of transferring a program between two real computers, where differences of processors, buses, networks, peripherals, communications protocols, programming languages, operating systems, and other installed software rather complicate the process of porting a program.

This brings us to the second point, whether simulation can help. Simulation has been a powerful tool in theoretical computer science, in a huge variety of settings. Consequently, a theoretical computer scientist's almost immediate reaction to the situation of two different computational systems is to ask if they can efficiently

simulate each other. If so, then for most theoretical purposes they are equivalent. So, even if we assume that the teacher and the learner are modeled by non-identical computers, the learner can use the teacher's program by simulating it, and a version of outright coding is still possible.

Suppose we map the idea of simulation to the situation of someone trying to teach another person to juggle; what problems arise? The behavior, juggling, is an extended process that interacts physically with the world of muscles, juggling bags, air, light, and so on, in a time-critical fashion. What must the learner do to simulate the teacher's juggling program? The perceptual signals of vision, touch, and pressure from the learner must be translated into equivalent ones for the teacher's program. Motor signals generated by the teacher's program for the teacher's muscles must be translated to equivalent motor signals for the learner's muscles. And all this must be done before the juggling bags hit the floor. It seems extremely unlikely that people have the capacity to simulate one another in anything like this sense.

In addition to the obvious constraints on memory and speed, one major obstacle to simulation in this sense for humans is the fact that many of a person's capabilities are only partly and poorly known to that person. This is most evident in embodied capabilities, for example, the sequence of motor signals and responses that allow a person's hands to type a word, but it probably holds equally of more strictly cognitive capabilities, like the ability to picture a friend's face. Many of our capabilities are, in effect, "black boxes" in the sense that we learn to produce appropriate control signals to achieve certain effects, but the actual details of how the signals lead to the effects are opaque to us.

We have attempted to reflect some of these considerations in the model we present. We are aware that it is only a first approach to the issues, with many

drawbacks of its own, but we hope that it will inspire others to think more about models of teaching.

Since we are considering issues that lie at the foundation of theoretical computer science, we start from the theory of computable functions, complexity measures and identification in the limit. Instead of the usual Turing machine model of the learner, we assume that the learner is a Turing machine with "black box" access to a programming system with a complexity measure. Thus, the learner is an oracle Turing machine, where the oracle is answering questions about an unknown programming system. (Programming systems and complexity measures axiomatically generalize the notion of enumerations of programs and their corresponding step-counting functions.) Approximately speaking, the Turing machine component represents the purely cognitive operations of the learner, and the programming system in the black box represents the repertoire of possible actions of the learner, or, more succinctly, "thinking" and "doing". In this setting, the concept to be learned is a partial recursive function, and the goal of the learner is to find a correct program *in the programming system of the black box* for the target function. Again speaking approximately, the learner must learn to "do" the exemplified thing, and not just "imagine" doing it. That is, having a correct program for the target function in the standard Turing machine system is not enough; the learner must find a correct program in the black box system.

If the black box system is known to be the same as the standard Turing machine system, or if the learner knows a program in the black box system to simulate programs in the standard Turing machine system, then this distinction collapses. Therefore we assume that the programming system in the black box is more or less arbitrary and unknown to the learner, and require that a learning algorithm (for the

Turing machine, or "cognitive," component of the learner) work correctly for a whole class of possible programming systems in the black box. We can characterize this aspect of the model as treating the action-space of the learner as part of the initially unknown environment. A new-born animal may be in somewhat analogous situation, having to discover how to focus its eyes or move its limbs partly by experiment.

Modeling lack of self-knowledge in this way has some fairly strong consequences. For example, suppose that the black box may contain an arbitrary acceptable programming system (i.e., some "fairly natural" programming system; see the formal definitions in Section 2.1), and suppose that the task of the learner is to find in the limit a program in the black box system for the constant all-zero function, intuitively a very simple function. Theorems 2 and 4 below show this to be impossible. The explanation for this is that there are acceptable programming systems in which the constant all-zero function is not at all simple to compute.

We also model the teacher, who attempts to help the learner learn an arbitrary partial recursive function in the limit. We could model the teacher by a standard Turing machine, with no lack of self-knowledge, but we choose to model the teacher similarly by a pair consisting of a Turing machine and a black box containing a programming system. This assumption is made partly for uniformity, so that teacher and learner are agents of the same kind. (For example, extensions of the theory might naturally permit the learner to go on to become a teacher.) This assumption also strengthens our positive results, since if a teacher with a black box is able to teach effectively, then so also can a simple Turing machine teacher. By the same argument, the assumption would seem to weaken our negative results, since perhaps any difficulty comes from the teacher's lack of self-knowledge. However, as we note, all of our negative results actually hold also in the case in which the teacher is a

simple Turing machine.

Non-standard programming systems have seldom been brought up since the famous Rogers' Isomorphism Theorem [30]. This is no surprise, since all the non-awkward ones (i.e., all acceptable programming systems) are closely related to each other (both literally and nonliterally speaking, see Machtey and Young [30] and Chapter 2). Despite this, the main idea of our model is using different programming systems for the teacher and the learner. The objectives of this dissertation are to discover what relations between the teacher's and learner's programming systems allow certain classes of functions to be learned. In addition to that, we would like to know when these classes can be learned in a more robust way. For example, we prefer learners that can fail gracefully in cases such as:

1. The target function is not from the designated class;
2. The learner's black box is not from the designated class;
3. The teacher maliciously tries to mislead the learner.

We would also like to discover classes of functions that cannot be learned under certain circumstances, or that cannot be learned due to the possible problems mentioned above. Other questions that we could ask are about the usefulness of the teacher and the reasonableness of the model itself. For example, does knowledge of the teacher's programming system make learning any easier for the learner? Some answers to these and other questions are given in this part of the thesis.

Chapter 2

Definitions and Notation

In this chapter we give definitions for the concepts used extensively throughout Part 1 of the thesis. Some more specific definitions appear in the chapters where they are relevant. We assume that the reader is familiar with basic recursion theory and do not give definitions of, say, Turing machines. A good book to consult is "An Introduction to the General Theory of Algorithms" by Machtey and Young [30], in fact, many of the definitions below are taken from it. A more recent text covering the same concepts is "A Recursive Introduction to the Theory of Computation" by Carl Smith [37].

2.1 Programming Systems and Complexity Measures

All our functions work from subsets of the set of natural numbers to the set of natural numbers (denoted by $\mathbb{N}$), unless otherwise specified. By natural numbers we mean

all non-negative integers, i.e., $0 \in \mathbb{N}$. We consider only functions of one argument because all other functions can be "coded" into them by composing with appropriate projection functions. In particular, for situations in which we need two-argument functions, we use the following kind of encoding.

We fix some total recursive bijection between pairs of natural numbers and natural numbers. By $\langle \cdot, \cdot \rangle$ we denote the function that maps pairs to numbers and by $\pi_1(\cdot)$ and $\pi_2(\cdot)$ we denote the two functions that map natural numbers to the first and second components of their corresponding pairs, respectively. That is, $\pi_1(\langle x, y \rangle) = x$ and $\pi_2(\langle x, y \rangle) = y$. Now, when we wish to define a function f of two arguments, we simply define $f(\langle x, y \rangle)$ to have the value we would like to assign to $f(x, y)$. Having said this, we will sometimes omit the angle braces around arguments when it will create no confusion. When we need to encode a three-argument function as a one-argument function, we define $f(\langle\langle x, y \rangle, z \rangle)$ to have the value $f(x, y, z)$.

We denote the class of all partial recursive functions by P. We start our definitions by recalling some facts about programming systems and complexity measures.

Definition 1 A *programming system* is a listing ϕ_0, ϕ_1, ... which includes *all* partial recursive functions (of one argument, from $\mathbb{N}$ to $\mathbb{N}$).

Definition 2 Function U_ϕ is called the *universal* function for the programming system ϕ_0, ϕ_1, ..., if $U_\phi(i, x) = \phi_i(x)$ for all i and x. Note that the universal function itself need not be partial recursive and thus may not belong to the listing ϕ_0, ϕ_1, ...

Definition 3 A programming system ϕ_0, ϕ_1, ... is *universal* if the universal function U_ϕ for it is partial recursive. This means that the listing ϕ_0, ϕ_1, ... includes U_ϕ. In

this case we denote the universal function by ϕ_{univ} rather than U_ϕ, and treat *univ* as an index of the universal function in the listing.

Definition 4 Function C_ϕ is called a *composition* function for the programming system $\phi_0, \phi_1, \ldots$, if it is total and if $\phi_{C_\phi(i,j)} \equiv \phi_i \circ \phi_j$ for all i and j. That is, for all i, j, and x,

$$\phi_{C_\phi(i,j)}(x) = \begin{cases} \phi_i(\phi_j(x)), & \text{if } \phi_j(x) \text{ and } \phi_i(\phi_j(x)) \text{ are defined;} \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Note that a composition function need not be recursive.

Definition 5 A programming system $\phi_0, \phi_1, \ldots$ is *acceptable* if it is universal and if there exists a (total) recursive composition function for it.

Programming systems are often referred to elsewhere in the literature as *indexings* or *Gödel numberings* of the partial recursive functions. Some of the most frequently used indexings are obtained by fixing a particular encoding of Turing machine programs and ordering all valid programs based on some fixed total order on their encodings. In this thesis we often refer to the "Turing machine with index i". By this we mean the following.

We fix an encoding of Turing machine programs by strings over $\{0,1\}^*$. (There are a number of ways to do this, we just assume that we have decided on one particular encoding E, which is the one we always use and refer to.) Then we fix the total order on the encodings of programs to be the lexicographical one. The *Turing machine with index i* is the one with program p, where p encodes to string e (i.e., $E(p) = e$),

and e is the string with number i in the lexicographical order of all strings that are valid E-encodings of Turing machine programs.

Definition 6 The *Turing Machine Programming System* is the listing TM_0, TM_1, TM_2 ..., where each TM_i is the function computed by the Turing machine with index i. It is a widely used acceptable programming system.

Definition 7 Let ϕ_0, ϕ_1, ... be any acceptable programming system. A listing Φ_0, Φ_1, ... of partial recursive functions is a *computational complexity measure* (for the given acceptable programming system) if it satisfies the following conditions:

1. For all i and x, $\Phi_i(x)$ is defined if and only if $\phi_i(x)$ is defined.

2. Inequality $\Phi_i(x) \leqslant s$ is a recursive predicate of i, x, and s.

Definition 8 The *Turing Machine Complexity Measure* is the listing $\overline{TM}_0$, $\overline{TM}_1, \ldots,$ where each $\overline{TM}_i(x)$ is defined to be the number of steps in which the Turing machine with index i stops on input x after writing output. $\overline{TM}_i(x)$ is undefined if the machine never stops on x or does not produce output before stopping.

2.2 Black Boxes

Now we introduce some concepts and notation more specific to this thesis.

Definition 9 A *Black Box* is any total recursive three-argument function from $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ to $\mathbb{N} \cup \{?\}$. (We could recode $\mathbb{N} \cup \{?\}$ as natural numbers in a straightforward way, but we prefer human intelligibility.)

We often consider restricted classes of Black Boxes. For example, the *primitive recursive Black Boxes* are the the ones defined by primitive recursive three-argument functions. When such classes of Black Boxes are recursively enumerable (e.g., the class of primitive recursive three-argument functions), we sometimes use notation BB^{bb} to refer to the Black Box with index bb in the recursive enumeration of class BB. This superscript notation should not be confused with the subscript notation introduced in the next definition.

Definition 10 If BB is a Black Box, and $i \in \mathbb{N}$ then by BB_i we denote the function defined as follows:

$$BB_i(x) \stackrel{\text{def}}{=} \begin{cases} BB(i,x,s_0), & \text{if } BB(i,x,s_0) \neq ? \text{ and } BB(i,x,s) = ? \text{ for all } s < s_0; \\ \text{undefined}, & \text{if } BB(i,x,s) = ? \text{ for all } s. \end{cases}$$

We say that BB *contains* the functions BB_i, i.e., we think of BB as of a listing of functions BB_0, BB_1, ...

Definition 11 A Black Box BB is *full* if it contains every partial recursive function. That is, BB is full if the listing BB_0, BB_1, ... is a programming system.

Note that since a Black Box itself is a total recursive function, it cannot contain any uncomputable functions.

Definition 12 A Black Box BB is *universal* if the listing BB_0, BB_1, ... is a universal programming system. That is, BB is universal if it contains a universal programming system, i.e., a programming system such that its universal function is itself partial recursive.

```
UNIV(n)
{
    i = π_1(n);
    x = π_2(n);

    For (s = 0; ; s++)                    /* Infinite Loop */
    {
        v = DEFINEBB(i, x, s);

        If (v ≠ ?)                        /* BB_i(x) defined */
            Return v;

    }

}
```

Figure 2.1 Algorithm for the universal function

Lemma 1 *Every full Black Box is universal.*

Proof: The universal function for a Black Box BB is a function U such that for all i and x, $U(\langle i, x\rangle) = BB_i(x)$. It can be computed by the algorithm given in Figure 2.1, which just scans the values of $BB(i, x, s)$ for $s = 0, 1, \ldots$.

It is obvious that this algorithm can be implemented on a Turing machine that has access to a program DEFINEBB that gives the values of BB. There must be such a program, since every Black Box is a total recursive function. Therefore, this algorithm defines a partial recursive function, and since BB is full this function is contained in BB. Hence, BB is a universal Black Box. ■

From Definition 12 it follows that every universal Black Box is full, hence "full Black Box" and "universal Black Box" are synonyms. We use the term "universal Black Box" rather than "full" throughout the rest of Part 1 of the thesis. We hope that this will remind the reader about the existence of the universal function in the Black Box and will provide greater compatibility with the terminology of Machtey

and Young [30].

Definition 13 A Black Box BB is *acceptable* if the listing BB_0, BB_1, ... is an acceptable programming system. That is, BB is acceptable if it contains an acceptable programming system, which is one that is universal and for which there is a total recursive composition function.

Lemma 2 *There is a universal Black Box UBB which is not acceptable.*

The proof of Lemma 2 is given in the Appendix to Part 1, page 106. It is not hard but long enough to interfere with the focus of this chapter, which is definitions. It is not necessary to understand the proof before reading further, one should just keep in mind that there exist universal Black Boxes that are not acceptable.

Definition 14 If BB is a Black Box, and $i \in \mathbb{N}$ then by $\overline{BB_i}$ we denote the function defined as follows:

$$\overline{BB_i}(x) \stackrel{\text{def}}{=} \begin{cases} s_0, & \text{if } BB(i,x,s_0) \neq \text{?} \text{ and } BB(i,x,s) = \text{?} \text{ for all } s < s_0; \\ \text{undefined}, & \text{if } BB(i,x,s) = \text{?} \text{ for all } s. \end{cases}$$

We think of each $\overline{BB_i}$ as the complexity function corresponding to BB_i and call it the *measure* (of BB_i). We say that measures $\overline{BB_0}$, $\overline{BB_1}$, ... are contained in BB.

Note that since every Black Box BB is recursive by definition, the listing of measures $\overline{BB_0}$, $\overline{BB_1}$, ... satisfies both conditions of Definition 7 and is therefore a valid computational complexity measure for the functions contained in BB (assuming that BB is acceptable).

There is one particular acceptable Black Box which is of special interest to us. In addition to being acceptable, it is also primitive recursive, showing that there are very natural primitive recursive and acceptable Black Boxes.

Definition 15 The *Turing Machine Black Box* is denoted by *TMBB* and defined as follows:

$$TMBB(i,x,s) \stackrel{\text{def}}{=} \begin{cases} y, & \text{if } \overline{TM}_i(x) \leqslant s \text{ and } TM_i(x) = y; \\ ?, & \text{otherwise.} \end{cases}$$

Note that the listing $TMBB_0$, $TMBB_1$, $TMBB_2$ ... is just the Turing Machine Programming System (also denoted by TM_0, TM_1, TM_2 ...), and that the listing $\overline{TMBB}_0$, $\overline{TMBB}_1$, $\overline{TMBB}_2$ ... is the Turing Machine Complexity Measure (also denoted by $\overline{TM}_0$, $\overline{TM}_1$, $\overline{TM}_2$ ...). Another interesting property of the Turing Machine Black Box is that $TMBB(i,x,s) = ?$ implies that $\overline{TMBB}_i(x) > s$, which is not necessarily true for arbitrary Black Boxes.

2.3 *b*-relatedness

In this section we explore some relationships between Black Boxes.

Definition 16 Let f be any partial function from $\mathbb{N}$ to $\mathbb{N}$. The *domain* of f is the set of all points in $\mathbb{N}$ where f is defined. We denote this set by $\mathrm{Dom}(f)$.

Definition 17 Function f *extends* function g if $f(x) = g(x)$ for all $x \in \mathrm{Dom}(g)$ (i.e., f agrees with g on all points x where g is defined). We denote this fact by $f \geqq g$ and sometimes say that f is an *extension* of g.

Note that extending is a transitive relation, i.e., if $f \geqq g$ and $g \geqq h$ then $f \geqq h$. Also note that if $f \geqq g$ and $g \geqq f$ then $f \equiv g$. Let us now introduce two symbols that are sometimes used to save space in formulas.

Definition 18 The phrase *for all but finitely many* is denoted by $\overset{\infty}{\forall}$. This is very similar to saying *for all* ($\forall$), except that a finite number of exceptions is allowed. Similarly, the phrase *exist infinitely many* is denoted by $\overset{\infty}{\exists}$. This is like *exists* ($\exists$), except that not just one but infinitely many objects exist that satisfy the required condition.

We now proceed to the most important definition of this section. We define a relation of being "not more than b slower" between Black Boxes.

Definition 19 A Black Box BB' is b-related to the Black Box BB if there exists a total recursive two-argument function $b(x, s)$ such that for all i there exists a j satisfying the following properties:

1. $BB'_j \geqq BB_i$;
2. $\overline{BB'_j}(x) \leqslant b(x, \overline{BB}_i(x))$, for all but finitely many $x \in \mathrm{Dom}(BB_i)$.

It is easy to see that b-relatedness is essentially analogous to the bounding relationship between two acceptable programming systems with complexity measures, as given by the Rogers' Isomorphism Theorem and the theorem about recursive relatedness of complexity measures. The two relationships differ in that we do not require the Black Boxes to be universal or acceptable, we do not assume a recursive translation function t between them, and we allow the function in one Black Box

to extend the corresponding function in the other. We now develop some further properties of b-relatedness and find some cases when it provably exists. Most of the time we will need plain b-relatedness, but on some occasions we will use the extra features proved below.

Rogers' Isomorphism Theorem together with the theorem about recursive relatedness of complexity measures [30] imply that every two acceptable Black Boxes are b-related for some total recursive function b. Indeed, if the translation function between the two systems is known (intuitively, a function that will find a j for every i such that they satisfy property 1 above), it is easy to construct the necessary total recursive function b. Furthermore, b can be made such that not only BB' is b-related to BB but also BB is b-related to BB' and, in addition, b is monotone in its second argument.

In our model the recursive translation between the two Black Boxes is usually not known. Therefore, the total recursive function $b(\cdot, \cdot)$ cannot be constructed in a straightforward way, but we can still prove that it exists.

If the two Black Boxes are not acceptable, then the situation becomes even less promising, since then it is not known whether the translation function exists. The following lemma proves that it does exist between a universal and an acceptable Black Box.

Lemma 3 *Let BB' be an acceptable Black Box. Let BB be a universal Black Box. Then there exists a total recursive function t such that $BB_i \equiv BB'_{t(i)}$, for all $i \in \mathbb{N}$.*

Proof: Theorem 3.1.5 from Machtey and Young [30] proves that a translation function exists from any universal programming system into any programming system

with a total recursive *s-1-1* function. By Theorem 3.1.2 from Machtey and Young [30] (the *s-m-n* Theorem), every acceptable programming system has a total recursive *s-m-n* function, so our lemma follows immediately from these two theorems. ■

Knowing that some translation function t exists from a universal into an acceptable Black Box, we can now prove that they are b-related for some total recursive two-argument function b. The construction is a little simpler than the one used in the theorem about recursive relatedness of complexity measures because we require less of b.

Lemma 4 *Let BB' be an acceptable Black Box. Let BB be a universal Black Box. Then there exists a total recursive two-argument function $b(x, s)$ such that BB' is b-related to BB.*

Proof: By Lemma 3 there is a total recursive function t such that $BB_i \equiv BB'_{t(i)}$, for all $i \in \mathbb{N}$. We use a typical "maximizing" construction to prove the lemma. First we define an auxiliary three-argument function $b'(i, x, s)$ as follows:

$$b'(i, x, s) \stackrel{\text{def}}{=} \begin{cases} \overline{BB'}_{t(i)}(x), & \text{if } \overline{BB}_i(x) = s; \\ 0, & \text{otherwise.} \end{cases}$$

Now we define $b(x, s) \stackrel{\text{def}}{=} \max\{\, b'(i, x, s) : i \leqslant x \,\}$.

It is easy to see that when we fix any i, there exists $j = t(i)$, such that:

1. $BB_i \equiv BB'_j$ and thus $BB'_j \geqq BB_i$;
2. For all $x \geqslant i$, if $x \in \mathrm{Dom}(BB_i)$ then $\overline{BB}_i(x) = s$, for some $s \in \mathbb{N}$ and therefore, $\overline{BB'}_j(x) = b'(i, x, s) \leqslant b(x, s) = b(x, \overline{BB}_i(x))$;

Hence, Black Box BB' is b-related to Black Box BB. ■

We now continue by proving that b-relatedness can be made monotone in the second argument.

Lemma 5 *Let BB and BB' be two Black Boxes. If BB' is b-related to BB for some total recursive two-argument function $b(x,s)$, then BB' is also b'-related to BB by a total recursive two-argument function $b'(x,s)$ which is monotonically nondecreasing in its second argument.*

Proof: Define $b'(x,s)$ to be $\max\{\, b(x,z) : z \leqslant s \,\}$. Since b is total recursive, so is b', and from the definition it follows that $(s' \geqslant s) \Rightarrow (b'(x,s') \geqslant b'(x,s))$. Obviously, $b(x,s) \leqslant b'(x,s)$ for all x and s, so BB' is b'-related to BB. ■

Using this simple construction, we can now prove the transitivity of b-relatedness.

Lemma 6 *Let BB, BB' and BB'' be three Black Boxes. If BB' is b_1-related to BB for some total recursive two-argument function $b_1(x,s)$ and BB'' is b_2-related to BB' for some total recursive two-argument function $b_2(x,s)$, then there exists a total recursive two-argument function $b(x,s)$, such that BB'' is b-related to BB.*

Proof: By Lemma 5 we know that BB'' is b'_2-related to BB' for some total recursive two-argument function $b'_2(x,s)$ which is nondecreasing in its second argument. Thus, we have that for all i there exists a j and a k such that:

1. $BB''_k \geqq BB'_j \geqq BB_i$;
2. $\overline{BB''_k}(x) \leqslant b'_2(x, \overline{BB'_j}(x))$, for all but finitely many $x \in \mathrm{Dom}(BB'_j)$;

3. $\overline{BB'_j}(x) \leqslant b_1(x, \overline{BB_i}(x))$, for all but finitely many $x \in \mathrm{Dom}(BB_i)$.

Since $b'_2(x,s)$ is nondecreasing in its second argument, $\overline{BB''_k}(x) \leqslant b'_2(x, b_1(x, \overline{BB_i}(x)))$, for all but finitely many $x \in \mathrm{Dom}(BB'_j)$. We can now define $b(x,s) \stackrel{\text{def}}{=} b'_2(x, b_1(x,s))$, which is obviously a total recursive two-argument function. Since $BB'_j \geqq BB_i$, we have that $\mathrm{Dom}(BB_i) \subseteq \mathrm{Dom}(BB'_j)$, and therefore $\overline{BB''_k}(x) \leqslant b(x, \overline{BB_i}(x))$, for all but finitely many $x \in \mathrm{Dom}(BB_i)$. This concludes the proof of the lemma. ■

2.4 Learners and Teachers

Here we describe in detail our model of learning (and teaching). We have two agents, the *learner* and the *teacher*, usually denoted by L and T, respectively. Each of the agents is an oracle Turing machine, equipped with the usual *Work Tape* and a few special tapes. Both the teacher and the learner share a common *Input Tape*. We need another definition to describe the contents of this tape.

Definition 20 An *arbitrary enumeration* of a function f (also called simply an *enumeration*) is an infinite listing of *elements* a_0, a_1, ..., that satisfies the following two conditions:

1. Each a_i is either the symbol '$*$' or an ordered pair $(x, f(x))$, for some point $x \in \mathrm{Dom}(f)$;
2. Each point x from the domain of f appears as the first component of some pair in the listing.

Note that from this definition it follows that if (x,y) and (x,z) are two pairs in the listing, then $y = z$.

The Input Tape (of the teacher and the learner) contains an arbitrary enumeration of the target function, i.e., it contains distinct elements each of which is the symbol '$*$' or an ordered pair. Both agents can only read this tape and they can do it independently of each other. That is, they each have a tape-head on this tape, and the tape is read-only.

The teacher and the learner also share a common *Message Tape*. This tape is for their communication and they both can read and write it. It is assumed that this communication happens using some known alphabet and encodings for whatever messages they would like to exchange. More specifically, we assume that the learner has exclusive access to the Message Tape until it enters a special *send* state, at which time its computation is suspended. At this point, the teacher's (initially or previously) suspended computation is resumed and it has exclusive access to the Message Tape until the teacher enters its *send* state. Then the teacher's computation is suspended again and the learner's is resumed, and so it goes on forever.

Both the teacher and the learner have their private *Work Tape* and *Box Tape*, where they perform computations and communicate with their oracles, respectively. The teacher's oracle holds the teacher's Black Box, which we denote by TBB most of the time. The learner's oracle holds the learner's Black Box, which we usually denote by BB. The teacher and the learner can encode the question "Does program i on input x stop in s or less steps, and if so, what is the output?" on their Box Tapes and their oracles encode the appropriate answer, which is $TBB(i, x, s)$ for the teacher and $BB(i, x, s)$ for the learner. We don't describe the details of this mechanism, just say that the agents "find out" the respective values of their Black Boxes. We also say that the teacher T is *equipped* with a Black Box TBB and denote this by $T(TBB)$. Similarly, for the learner L with a Black Box BB we write $L(BB)$ and say that L is

equipped with a Black Box *BB*.

The teacher has one more tape, which is the *Answer Tape*. It is read only and contains an index i such that TBB_i extends the target function. The learner has no access to this tape, but the teacher is allowed to pass this index to the learner with the help of the Message Tape.

The learner also has one more tape, which is the *Output Tape*. From time to time it writes a number on this tape and puts a special *marker* at the end of the number to indicate that a new hypothesis has been output. For simplicity we assume that it moves to the right after each index (with marker) written and does not destroy the previous ones. The teacher cannot access this tape.

Definition 21 The learner *converges* to hypothesis h if after a finite number of steps it outputs (writes) the number h to the Output Tape and never again outputs a different number.

Note that the learner may converge to h in two ways: either by outputting h after some finite number of steps and never outputting any hypothesis again, or by beginning to systematically output h from some point on. That is, eventually h must become the last number written or the only number that will ever subsequently be written.

Definition 22 Given a learner L, a teacher T, a Black Box *BB* for the learner, a Black Box *TBB* for the teacher, a partial recursive function f, and an arbitrary enumeration of f on the Input Tape, and an arbitrary index i on the Answer Tape such that TBB_i extends f, we say that the learner $L(BB)$ *converges correctly* if

$L(BB)$ converges to h and BB_h extends the function f. We say that $L(BB)$ *converges incorrectly* if $L(BB)$ converges to h and BB_h does not extend the function f.

If there is no index h such that $L(BB)$ converges to h, then it must be that $L(BB)$ either does not output any hypothesis, or that it outputs infinitely many hypotheses on the Output Tape, and for each hypothesis output, there is a later time at which a different hypothesis is output. In the latter situation, we say that $L(BB)$ *changes its mind infinitely often.*

Definition 23 The learner L, equipped with a Black Box BB, *learns* the target function f from the teacher T equipped with a Black Box TBB, if TBB contains a function extending f, and for every enumeration of f given on the Input Tape and every index i such that TBB_i extends f on the Answer Tape, $L(BB)$ communicating with $T(TBB)$ converges correctly.

We extend this definition to the case of learning a class C of partial recursive functions in the following way.

Definition 24 The learner L, equipped with a Black Box BB, learns the class of partial recursive functions C from the teacher T, equipped with a Black Box TBB, if for each function $f \in C$, TBB contains a function extending f, and for every $f \in C$, every enumeration of f on the Input Tape, and every index i on the Answer Tape such that TBB_i extends f, $L(BB)$ communicating with $T(TBB)$ converges correctly.

In this thesis we primarily consider two cases: the learnability of a single function, or the learnability of P, the class of all partial recursive functions. In the second case, we generally assume that the teacher's Black Box is at least universal.

We also consider *independent learners*, that is, learners that have no send state. In this case, we can delete the teacher and the Answer Tape, and regard the Message Tape as just another work tape, since there can be no interaction with the teacher. Independent learners are analogous to inductive inference machines, except that inductive inference machines have a fixed, known programming system, while independent learners must work with an unknown Black Box.

2.5 Reliability and Proofness Properties

Motivated by the work of Minicozzi [31] and Blum and Blum [15] on reliable (or strong) identification, we are interested in designing learning protocols that "fail gracefully" in certain situations other than those for which the protocol is specifically intended. In particular, we would like the learner to avoid converging incorrectly. That is, if it does not converge correctly, then it should either not output any hypothesis or change its mind infinitely often. The possible "unanticipated situations" that a teacher and a learner might have to cope with are:

1. A function from outside the intended target class,
2. A Black Box from outside the intended class for the learner, or
3. A Black Box from outside the intended class for the teacher.

The first situation is considered both by Blum and Blum [15] and by Minicozzi [31], and the inductive learners that can overcome this difficulty are called *reliable* (or *strong*). We have extended the notion of reliability to cover all three situations mentioned above.

Definition 25 Let BBC and $TBBC$ be two classes of Black Boxes. Let C be a class of partial recursive functions. A learner L and a teacher T are called *$(C, BBC, TBBC)$-reliable* if for any learner's Black Box $BB \in BBC$, any teacher's Black Box $TBB \in TBBC$, any target function $f \in C$, any enumeration of f on the Input Tape and all indices i such that $TBB_i \geqq f$ on the Answer Tape, $L(BB)$ communicating with $T(TBB)$ does not converge incorrectly.

In this dissertation we frequently focus on specialized cases of $(C, BBC, TBBC)$-reliability, where only some parameters are considered "truly variable", while the rest are varying within their "acceptable", "designated" classes. For example, we may design the teacher and the learner to correctly learn some class of functions provided that their Black Boxes come from two fixed classes; then we may investigate what happens if, say, learner's Black Box does not conform to this requirement, while everything else does. We introduce a set of definitions that describe such specializations.

Definition 26 Let BBC, BBC', $TBBC$ and $TBBC'$ be classes of Black Boxes. Let C and C' be two classes of partial recursive functions. Let T and L be a teacher and a learner, respectively. Let the class of all Black Boxes be denoted by $ABBC$.

1. We say that L and T are *target-proof on C'* for BBC and $TBBC$ if L and T are $(C', BBC, TBBC)$-reliable.

2. We say that L and T are *learner-box-proof on BBC'* for C and $TBBC$ if L and T are $(C, BBC', TBBC)$-reliable.

3. We say that L and T are *teacher-box-proof on $TBBC'$* for C and BBC if L and T are $(C, BBC, TBBC')$-reliable.

4. We say that L and T are *target-proof* for BBC and $TBBC$ if L and T are $(P, BBC, TBBC)$-reliable.

5. We say that L and T are *learner-box-proof* for C and $TBBC$ if L and T are $(C, ABBC, TBBC)$-reliable.

6. We say that L and T are *teacher-box-proof* for C and BBC if L and T are $(C, BBC, ABBC)$-reliable.

7. We say that L and T are *learner-box-and-teacher-box-proof* for C if L and T are $(C, ABBC, ABBC)$-reliable.

8. We say that L and T are *target-learner-box-and-teacher-box-proof* if L and T are $(P, ABBC, ABBC)$-reliable.

The definitions above are meant to highlight the parameters of the generalized reliability that do not comply with the requirements of some learning protocol. Although no learning protocol is explicitly mentioned in these definitions, they become more useful and convenient when used in the context of some teacher and learner learning a class of concepts. Unless L and T are required to be able to learn some class of concepts, it is extremely easy to make them target-learner-box-and-teacher-box-proof. It suffices for this that the learner either never outputs a hypothesis or that it alternates between two different ones.

Note that (X', Y', Z')-reliability implies (X, Y, Z)-reliability for all $X \subseteq X'$, $Y \subseteq Y'$ and $Z \subseteq Z'$. It is an open question, however, whether (X', Y', Z)-reliability together with (X', Y, Z')-reliability imply (X', Y', Z')-reliability, for any $X \subseteq X'$, $Y \subseteq Y'$ and $Z \subseteq Z'$. Similar questions applied to the other pairs of parameters to reliability are also open.

Target-proofness is meant to be analogous to reliable (or strong) learning, explored by Minicozzi [31] and Blum and Blum [15]. Learner-box-proofness and teacher-box-proofness are introduced specifically for our model. Target-proofness is considered very briefly in this thesis since (nearly) all our learning results hold for P, the class of all partial recursive functions. Learner-box-proofness, however, is very important for some of our results. So is teacher-box-proofness, although for different reasons. It is the "dual" property of learner-box-proofness and is related to two properties of the learner and the teacher which we now describe.

If both the learner and the teacher are the (correctly functioning) agents designed for a specific learning problem, then there can be only the three above mentioned unanticipated situations that may prohibit them from performing the learning task successfully. From the learner's point of view, however, there is another, potentially more serious source of trouble—the teacher. We would like to build learners that do not converge incorrectly even when coupled with adversarial teachers. When considering teachers other than the intended ones, we permit any infinite sequence of messages for the Message Tape, say $m_1, m_2, \ldots$, which is used in place of the teacher as follows. When the learner enters its send state for the i-th time, the message m_i is placed on the Message Tape and the learner's computation is resumed.

Definition 27 Let C be a class of functions and BBC be a class of Black Boxes. We say that a learner L is *non-gullible* for C and BBC, if for all $f \in C$, any enumeration of f on the Input Tape, all Black Boxes $BB \in BBC$ and any infinite sequence of messages $m_1, m_2, \ldots$ used as responses on the Message Tape, $L(BB)$ does not converge incorrectly.

This definition models a large class of possible behaviors for teachers. It does not model the situation of the teacher causing the learner's computation to remain indefinitely suspended, however. Unfortunately, such an outcome cannot be ruled out easily, since many good teachers can fail to respond as a result of either of the three problems mentioned before:

1. The target function may not be from the designated class, causing the teacher to attempt some infinite computation, for example;

2. The learner's Black Box may not be from the intended class, causing the learner to ask an unexpected query to the teacher, which results in some infinite computation, for example;

3. The teacher's Black Box may not be from the intended class, causing an infinite search for a value, for example.

Therefore, we introduce a special responsiveness property for teachers that requires them to give a response despite any or all of the difficulties given above, and even in cases when the learner is adversarial. For this we need to consider a wide range of learner behaviors, which, as above, can be best accomplished by using an infinite sequence of messages m_1, m_2, ... for the Message Tape. Initially m_1 is placed on the Message Tape and the teacher's computation is started, and when the teacher enters its send state for the i-th time, the message m_{i+1} is placed on the Message Tape and the teacher's computation is resumed.

Definition 28 Let C be a class of functions and $TBBC$ be a class of Black Boxes. We say that a teacher T is *responsive* for C and $TBBC$, if for all Black Boxes $TBB \in TBBC$, all $f \in C$, any enumeration of f on the Input Tape, any index i

such that $TBB_i \geqq f$ on the Answer Tape, and any infinite sequence of messages m_1, m_2, ... used as responses on the Message Tape, $T(TBB)$ enters its send state infinitely many times.

The following lemma follows directly from the definitions of $(C, BBC, TBBC)$-reliability, non-gullible learners and responsive teachers.

Lemma 7 *Let C be a class of partial recursive functions. Let BBC and $TBBC$ be two classes of Black Boxes. Let L be a non-gullible learner for C and BBC. Let T be a responsive teacher for C and $TBBC$. Then L and T are $(C, BBC, TBBC)$-reliable.*

Proof: For any message that the learner may write on the Message tape, the teacher is required to provide a response. For whatever responses the teacher may provide, they can be modeled by some sequence of messages m_1, m_2, ... Therefore, the learner may not converge incorrectly. ■

Corollary 1 *Let L be a non-gullible learner for P and the class of all Black Boxes $ABBC$. Let T be a responsive teacher for P and $ABBC$. Then L and T are target-learner-box-and-teacher-box-proof.*

Proof: Follows immediately by replacing C with P, BBC with $ABBC$ and $TBBC$ with $ABBC$ in Lemma 7. ■

To put the results of Minicozzi [31], Blum and Blum [15] and our results in perspective, we now mention a few important results from their work and compare them with related results found in this dissertation. Blum and Blum introduce the notion of a function being *h-honest* and prove that for machines that are reliable on

the set of all partial recursive functions P, all the functions that a machine M can identify are h-honest for some total recursive function h (which can be uniformly constructed using M), and that all the functions that are h-honest can be identified by some inductive inference machine M (which can be uniformly constructed from h). Theorems 1 and 3 below give a somewhat similar result for learner-box-proof learning. Minicozzi proves the Union Theorem, which states that given two inductive inference machines that are reliable on some set of partial recursive functions S, one can construct another machine, reliable on S which is as powerful (on S) as both the two given machines. It is easy to see that if the set S in Minicozzi's Union Theorem is replaced by the set of all partial recursive functions P (thus weakening the theorem, of course), then it follows from the above-mentioned result by Blum and Blum. Similarly, Corollary 6, a result about unions of primitive recursive learner's Black Boxes, follows from Theorems 1 and 3. Minicozzi's Union Theorem in its stronger form is related to three theorems of Chapter 7, especially to Theorem 7. All these theorems have generalizations which are related to the generalization of Minicozzi's Union Theorem.

Chapter 3

Teaching the Fast Learners

Since we began with the issue of outright coding, it is instructive to examine a protocol for outright coding in this new setting. Consider the teacher T_0 that copies the contents of the Answer Tape to the Message Tape each time control is passed to T_0. Consider the learner L_0 that initially passes control to the teacher, and, when control returns to L_0, copies the contents of the Message Tape to the Output Tape and halts. Clearly, for every universal Black Box BB, $L_0(BB)$ learns every partial recursive function from $T_0(BB)$. That is, in case the Black Boxes of learner and teacher are the same, this setting permits a straightforward version of outright coding. Note that the agents are far from being learner-box-proof or teacher-box-proof and the learner is definitely not non-gullible; a different teacher or a slight difference between the teacher's and learner's Black Boxes suffice to make L_0 converge incorrectly.

To create an example of a teacher and a learner that are learner-box-proof and teacher-box-proof, and where the learner is non-gullible, we appeal to the well-known idea of using a computational complexity bound to guide learning. We describe a

teacher T_1 that supplies information about the computational complexity of the target function, obtained as follows.

Suppose TBB is the teacher's Black Box. The teacher has an index i on the Answer Tape such that TBB_i extends the target function f. Thus, for any x in the domain of f, the teacher can make oracle calls to find out $TBB(i, x, s)$ for $s = 0$, 1, 2, ... until it determines the minimum s for which $TBB(i, x, s) \neq ?$, i.e., until it determines the value of the measure $\overline{TBB}_i(x)$. For any entry $(x, f(x))$ on the Input Tape, the teacher T_1 supplies (on request) the value of $\overline{TBB}_i(x)$ on the Message Tape, where i is the index on the Answer Tape, and TBB is the teacher's Black Box.

We describe a learner L_1 that makes use of this complexity information. The learner L_1 initially outputs index 0 on the Output Tape, and sets its current hypothesis j to 0. For each new entry $(x, f(x))$ on the Input Tape, L_1 makes a request to the teacher, and then uses the number z returned on the Message Tape as a running-time bound, checking whether $BB_j(x) = f(x)$ or $\overline{BB}_j(x) > z$, where BB is the learner's Black Box. If $BB_j(x) = f(x)$, L_1 retains the hypothesis j and searches for the next entry $(x, f(x))$ to check on the Input Tape. Otherwise, L_1 writes $j+1$ on the Output Tape, sets j to $j + 1$, and restarts the process of checking values $(x, f(x))$ on the Input Tape from the beginning.

Then for every universal Black Box BB, $L_1(BB)$ learns all the partial recursive functions from $T_1(BB)$ and T_1 and L_1 are learner-box-proof and teacher-box-proof. (Recall that by the definition of teacher-box-proofness we still require that $TBB_i \geqq f$, even though TBB can be any Black Box. Therefore, it is safe for the teacher to compute $\overline{TBB}_i(x)$ for all $x \in \text{Dom}(f)$ even though TBB may be a Black Box that was not intended for T_1.) Furthermore, L_1 is non-gullible and T_1 with L_1 together are actually $(P, ABBC, ABBC)$-reliable, where $ABBC$ is the class of all Black Boxes.

In other words, they are target-learner-box-and-teacher-box-proof. This protocol no longer involves outright coding; T_1 does not send i to the learner, but rather uses i and access to its Black Box to provide information to guide the learner's own search. In fact, the Black Boxes of the teacher and learner need not be identical; as long as the learner's Black Box has a program that computes an extension of f and runs at least as fast on every input as program i in the teacher's Black Box, the learner will converge correctly.

Unless we specify in greater detail how L_1 and T_1 communicate, we cannot assert that T_1 is responsive. For example, the simplest way for a learner to specify that it needs to know complexity information for the pair $(x, f(x))$ would be to write x on the Message Tape before entering its send state. The teacher could then take the value x found on the Message Tape and determine the measure $\overline{TBB}_i(x)$ as described above. Unfortunately, if the learner for some reason writes $x \notin \mathrm{Dom}(f)$ on the Message Tape, there is no guarantee that $x \in \mathrm{Dom}(TBB_i)$ and the teacher may become stuck trying to find the measure $\overline{TBB}_i(x)$. If, however, we require the learner to just specify which element of the Input Tape it is seeking information about, this problem disappears, since the teacher can find the respective element on the Input Tape and then safely determine $\overline{TBB}_i(x)$, if this element is a pair. Or, if the element is a '$*$' then the learner has made a mistake in its request and the teacher can return some special value indicating this. When the latter specification is used, we can not only say that T_1 and L_1 are target-learner-box-and-teacher-box-proof and that the learner is non-gullible but also that the teacher is responsive.

By extending the above ideas somewhat more, we prove the following positive result.

Theorem 1 *There exists a learner* L^* *such that for every total recursive two-argument function* $b(x, s)$ *there is a teacher* T_b^* *such that for every universal Black Box* TBB *and for every Black Box* BB *that is b-related to* TBB*,* $L^*(BB)$ *learns* P *from* $T_b^*(TBB)$*. Furthermore,* L^* *is non-gullible,* T_b^* *is responsive and both agents together are target-learner-box-and-teacher-box-proof.*

Proof: Every time the teacher's computation is resumed, it reads the number k written by the learner from the Message Tape. It then finds the k-th element on the Input Tape. If this element is a '*', it clears the Message Tape, writes 0 on it and enters its send state. Otherwise, the k-th element is a pair $(x, f(x))$ and the teacher T_b^* computes $s = \overline{TBB}_i(x)$ as described above for teacher T_1, clears the Message Tape and writes z on it, where $z = b(x, s)$. The learner L^* is similar to L_1, with the addition of dovetailing to re-try previously discarded hypotheses, which allows for finitely many exceptions to the bounds supplied by the teacher. Intuitively, the teacher supplies information of the form: "it shouldn't take you longer than z steps to compute the result y from x," and the learner uses that information to prune fruitless searches for a satisfactory program in its Black Box. The algorithm for L^* is given in Figure 3.1.

In the Main Loop the learner picks a new hypothesis j, which selects a "hypothesis-function" BB_j from its Black Box. It also picks a new "testing parameter" m and enters the Testing Loop. There it tests whether BB_j agrees with the target function given on the Input Tape and whether its complexity is within the bound supplied by the teacher on the Message Tape. When a function fails this test, the learner breaks out of the Testing Loop and reiterates the Main Loop, i.e., picks another index, and repeats everything. This computation is "dovetailed", meaning that if no function meets the test requirements then the learner will return to each formerly abandoned

```
L*()
{
    For (n = 0; ; n++)                                          /* The Main Loop */
    {
        j = π1(n);
        m = π2(n);
        Output j + 1;
        Output j;

        For (k = 0; ; k++)                                      /* The Testing Loop */
        {
            Read k-th element e from the Input Tape;

            If (e is a pair (x, y))
            {
                Clear the Message Tape and write k on it;
                Enter the send state;                           /* Get Suspended */
                Read z from the Message Tape;   /* On Resuming Computation */

                For (s = 0; s ⩽ max(m, z); s++)                 /* The Bounded Loop */
                {
                    v = BB(j, x, s);

                    If (v ≠ ?)
                        Break;                                  /* Out of the Bounded Loop */

                }

                If ((v ≠ y) or (s > max(m, z)))
                    Break;                                      /* Out of the Testing Loop */

            }

        }

    }

}
```

Figure 3.1 Learner's algorithm

index infinitely often. Every time the learner returns to the same index j, the testing parameter m is different, and thus the test will eventually be made with arbitrarily large values of m.

The test itself is inside the Testing Loop and consists of comparing each pair (x, y) found on the Input Tape against the value of the current hypothesis-function on x. However, the hypothesis-function need not be defined on x at all, so the learner uses a bound on the complexity of this function. This bound is chosen to be the maximum of the teacher-supplied bound from the Message Tape and the parameter m, which is different for each return to the same hypothesis. The actual search for the value $BB_j(x)$ is in the Bounded Loop, which is controlled by the bound. As soon as the hypothesis-function fails to comply with the test being performed, the learner abandons this hypothesis, outputs a new one and starts testing it.

Now we prove that this learner is capable of learning every partial recursive target function f if its Black Box BB is b-related to the teacher's Black Box TBB.

Claim 1 *There exists a natural number n such that:*

1. $BB_{\pi_1(n)} \geqq f$;
2. $\overline{BB}_{\pi_1(n)}(x) \leqslant \max(\pi_2(n), b(x, \overline{TBB}_i(x)))$, *for all points* $x \in \mathrm{Dom}(f)$.

Proof: Recall that the teacher is given an index i such that $TBB_i \geqq f$. Since BB is b-related to TBB, there exists a j such that

1. $BB_j \geqq TBB_i \geqq f$;
2. $\overline{BB}_j(x) \leqslant b(x, \overline{TBB}_i(x))$, for all but finitely many $x \in \mathrm{Dom}(TBB_i)$.

Since $TBB_i \geqq f$ we have that $\mathrm{Dom}(f) \subseteq \mathrm{Dom}(TBB_i)$ and from item 2 above we now have that $\overline{BB}_j(x) \leqslant b(x, \overline{TBB}_i(x))$, for all but finitely many $x \in \mathrm{Dom}(f)$. Let X be the set of points in $\mathrm{Dom}(f)$ where $\overline{BB}_j(x) > b(x, \overline{TBB}_i(x))$. Let $m \stackrel{\mathrm{def}}{=} \max\{\, \overline{BB}_j(x) : x \in X \,\}$. Then it must be that $\overline{BB}_j(x) \leqslant \max(m, b(x, \overline{TBB}_i(x)))$, for all $x \in \mathrm{Dom}(f)$. Now, if we take $n = \langle j, m \rangle$, the claim follows. ■

Let n_0 be the least n that satisfies Claim 1. Once variable n in the Main Loop of the algorithm reaches n_0, the learner will output a correct hypothesis j and will never change it. For all those n that are less than n_0, however, the hypothesis will eventually be changed, because of our choice of n_0 and because the learner always outputs a hypothesis $j+1$ just before outputting j—this introduces at least one hypothesis change between these hypotheses.

What remains to be proved is that the agents are target-learner-box-and-teacher-box-proof and that L^* is non-gullible and T_b^* is responsive. Notice that the learner can converge to an index only if the algorithm stays in the Testing Loop forever, either suspended or reiterating the loop. It cannot be suspended forever, since the teacher T_b^* is clearly responsive. Thus, the learner can converge to an index only by reiterating the Testing Loop forever. This can happen only if the hypothesis-function BB_j agrees with the target function f on all points x that appear (as the first components) in the pairs in its enumeration. But all points $x \in \mathrm{Dom}(f)$ will appear eventually, and thus j must be an index for an extension of the target function f. In other words, L^* cannot converge incorrectly. Even when the target function is not partial recursive, which implies that learning in this model is impossible, the learner still cannot converge incorrectly as long as the teacher responds to every query, but must change its mind infinitely often. ■

In this chapter we showed how to construct a special teacher T_b^*, which depends on a total recursive two-argument function b, and a special learner L^*, such that they are target-learner-box-and-teacher-box-proof, the teacher is responsive and the learner is non-gullible and they learn the class of all partial recursive functions P if the learner's Black Box is b-related to the teacher's. From now on we refer to the teacher given in the proof of Theorem 1 as the *Standard Teacher (with bound b)* and we always denote it (and no other teacher) by T_b^*. Likewise, we call the learner given in this proof the *Standard Learner*, and we denote it (and no other learner) by L^*.

Theorem 1 has a simple non-constructive corollary.

Corollary 2 *Let BB be an acceptable Black Box and TBB be a universal Black Box. Then there exists a total recursive two-argument function $b(\cdot,\cdot)$, such that $L^*(BB)$ (i.e., the Standard Learner, equipped with the Black Box BB) learns P from $T_b^*(TBB)$ (i.e., the Standard Teacher with bound b, equipped with the Black Box TBB).*

Proof: By Lemma 4 from Chapter 2, every acceptable Black Box is b-related to every universal Black Box, for some total recursive function $b(\cdot,\cdot)$. Thus, although we do not know how to construct the function b or the Standard Teacher T_b^* that depends on it, we still know that it exists, which suffices to prove the corollary. ■

Theorem 1 seems in some respects fairly modest; L^* is able to learn all the partial recursive functions from T_b^*, but only when its Black Box is "not too much slower" (as measured by b) than the teacher's. However, it may be that a much stronger positive result is provable in this model, possibly by using a more elaborate version of outright coding. We address the following question:

Is there a learner L, a teacher T, and a Black Box TBB such that $L(BB)$

learns all the partial recursive functions from $T(TBB)$ for all acceptable Black Boxes BB?

An affirmative answer to this question would cast serious doubt on the model we have defined. In the next chapter we present a basic theorem, one corollary of which is a negative answer to this question.

Chapter 4

Is the Teacher Important?

In the previous chapter we proved a relatively simple theorem exhibiting successful learning. Here we present a basic negative result, which seems to point to the necessity of the teacher or at least the necessity of some knowledge about the learner's Black Box. It is implied by Theorem 4 from Chapter 6, but we present it here for its simple and interesting proof.

Definition 29 Let Z be the constant all-zero function, defined by $Z(x) \stackrel{\text{def}}{=} 0$, for all $x \in \mathbb{N}$.

The following theorem indicates how difficult it is to deal with all acceptable Black Boxes. In particular, there are acceptable programming systems in which an intuitively simple function like the constant all-zero function is not at all "simple" to compute. In particular, without additional information, no independent learner is capable of finding in the limit a program that computes the constant all-zero function Z for every acceptable Black Box.

Theorem 2 *There is no independent learner L such that $L(BB)$ learns the constant all-zero function Z for all acceptable Black Boxes BB.*

Proof: Assume to the contrary that such an independent learner L exists. For every acceptable Black Box BB given, and for every enumeration of the all-zero function Z on the Input Tape, $L(BB)$ correctly converges to an index for Z (that is, an index j such that $BB_j(x) = 0$, for all $x \in \mathbb{N}$). We show how to use L to construct an inductive inference machine M that identifies in the limit every total recursive function, which is known to be impossible [15].

M gets the values of a total recursive target function f on its input tape. That is, it gets pairs $(x, f(x))$ (intermixed with $*$'s, possibly). Every $x \in \mathbb{N}$ appears at least once on the tape as the first component of a pair. Let ϕ_0, ϕ_1, ϕ_2, ... denote a standard Turing machine programming system (i.e., $\phi_i \equiv TM_i$, for all $i \in \mathbb{N}$), with a step-counting complexity measure Φ_0, Φ_1, Φ_2, ... (i.e., $\Phi_i \equiv \overline{TM_i}$, for all $i \in \mathbb{N}$). The goal of the inductive inference machine M is to converge correctly to an index i for f in the Turing Machine Programming System, that is, an index i such that $\phi_i \equiv f$.

The inductive inference machine M "builds" another acceptable programming system ψ (i.e., a listing of partial recursive functions ψ_0, ψ_1, ψ_2, ...) using ϕ (i.e., the listing ϕ_0, ϕ_1, ϕ_2, ...) and the values of f on the input tape. In the new system ψ, the constant all-zero function Z has the same indices as the total recursive function f has in ϕ. That is, $\phi_i \equiv f$ if and only if $\psi_i \equiv Z$. M simulates the independent learner L with some enumeration of the constant all-zero function on its Input Tape and a Black Box BB containing the programming system ψ with complexity measure Φ. According to our assumption, $L(BB)$ converges to an index for Z in the programming

system ψ, which is an index for f in the Turing Machine Programming System ϕ. We now describe the construction of ψ.

Definition 30 Let $\ominus$ be a binary operator on natural numbers defined by $x \ominus y \stackrel{\text{def}}{=} |x - y|$.

The new programming system ψ is defined by $\psi_i(x) \stackrel{\text{def}}{=} \phi_i(x) \ominus f(x)$. If $\phi_i(x)$ is undefined, so is $\psi_i(x)$. The complexity measure for ψ is defined to be the Turing Machine Complexity Measure Φ (i.e., the listing Φ_0, Φ_1, Φ_2, ...). It is clear that i is an index of f in ϕ if and only if i is an index of Z in ψ.

In order to simulate L, the inductive inference machine M must supply the answers to L's queries to the Black Box BB containing ψ. In order to compute the value of $BB(i, x, s)$, M reads its own input tape with the enumeration of f until it finds the pair $(x, f(x))$. We are only concerned with learning all the total recursive functions, so this value will be found. M also simulates the Turing machine with index i on input x for s steps. If the machine stops, it answers the query with the value $\phi_i(x) \ominus f(x)$. Otherwise, the answer is a ?, meaning that $\overline{BB}_i(x) > s$. (Recall that in general $BB(i, x, s) = ?$ does not imply that $\overline{BB}_i(x) > s$, but it does in this case, due to the Turing machine simulation that takes place.)

When M simulates L as described above, it appears to L that it is equipped with a Black Box BB, containing the programming system ψ with the complexity measure Φ and that its Input Tape contains an enumeration of the constant all-zero function Z. When L outputs a hypothesis, M outputs the same hypothesis and after L has converged to an index for Z, M will have converged to an index for f in the Turing Machine Programming System ϕ.

One thing remains to be proved, namely, that ψ is an acceptable programming system. We show this with three claims.

Claim 2 *Listing ψ_0, ψ_1, ... is a programming system, that is, it contains all the partial recursive functions.*

Proof: Suppose, by way of contradiction, that g is a partial recursive function missing in the listing ψ_0, ψ_1, ... Then there is a partial recursive function $h(x) \stackrel{\text{def}}{=} f(x) + g(x)$, which is missing in the programming system ϕ. This is a contradiction, because a programming system by definition contains all the partial recursive functions. ■

Claim 3 *The programming system ψ is universal, that is, the universal function U_ψ for this system is itself partial recursive and thus belongs to ψ.*

Proof: We need to prove that there exists an index $univ_\psi$, such that $\psi_{univ_\psi}(\langle i, x\rangle) = \psi_i(x)$. We can take $univ_\psi$ to be an index for the Turing machine that takes input $\langle i, x\rangle$ and outputs $(\phi_i(x) \ominus f(x)) + f(\langle i, x\rangle)$, assuming that it has access to a program computing the values of the total recursive function f. Then we will have that $\phi_{univ_\psi}(\langle i, x\rangle) \ominus f(\langle i, x\rangle) = \phi_i(x) \ominus f(x)$, from which it follows that $\psi_{univ_\psi}(\langle i, x\rangle) = \psi_i(x)$. Thus, we have shown that knowing the Turing machine program that computes f, it is straightforward to find an index $univ_\psi$ such that $\psi_{univ_\psi} \equiv U_\psi$. ■

Claim 4 *The programming system ψ is acceptable, that is, there exists a total recursive composition function C_ψ for it.*

Proof: We need to show that there exists a total recursive two-argument function $C_\psi(\cdot, \cdot)$ such that $\psi_{C_\psi(i,j)}(x) = \psi_i(\psi_j(x))$, for all indices i and j. Using the definition

of the programming system ψ, we know that $\psi_{C_\psi(i,j)}(x) = \phi_{C_\psi(i,j)}(x) \ominus f(x)$. We also know that $\psi_i(\psi_j(x)) = \phi_i(\psi_j(x)) \ominus f(\psi_j(x))$, which can be expanded farther into

$$\psi_i\big(\psi_j(x)\big) = \phi_i\big(\phi_j(x) \ominus f(x)\big) \ominus f\big(\phi_j(x) \ominus f(x)\big).$$

Therefore, we can take $C_\psi(i,j)$ to be an index for the Turing machine that takes input x and outputs $\phi_i(\phi_j(x) \ominus f(x)) \ominus f(\phi_j(x) \ominus f(x)) + f(x)$, assuming again that it has access to a program computing the values of the total recursive function f. Then we will have that $\phi_{C_\psi(i,j)}(x) \ominus f(x) = \phi_i(\phi_j(x) \ominus f(x)) \ominus f(\phi_j(x) \ominus f(x))$, from which it follows that $\psi_{C_\psi(i,j)}(x) = \psi_i(\psi_j(x))$. Thus, knowing the Turing machine program that computes f, we can construct a total recursive two-argument function $C_\psi(i,j)$ which is a composition function for the programming system ψ. ■

Claims 2, 3 and 4 show that ψ is an acceptable programming system, and therefore the simulated Black Box BB is an acceptable Black Box. Theorem 2 is proved. ■

Corollary 3 *There is no learner L, no teacher T and no Black Box TBB for the teacher, such that the learner $L(BB)$ learns the constant all-zero function Z from the teacher $T(TBB)$ for all acceptable Black Boxes BB.*

Proof: If for some teacher T and some teacher's Black Box TBB there exists such a learner L, then there exists an independent learner $L'(BB)$ which simulates both $T(TBB)$ and $L(BB)$ and thus learns Z for all Black Boxes BB. ■

Corollary 4 *There is no learner L, no teacher T and no Black Box TBB for the teacher, such that the learner $L(BB)$ learns all the partial recursive functions from the teacher $T(TBB)$ for all acceptable Black Boxes BB.*

Proof: Since there is no learner, teacher and Black Box such that the learner can learn the constant all-zero function Z from the teacher, there cannot be one that can learn all partial recursive functions. ■

This answers the question raised above of whether there might exist a teacher and a learner such that the learner can learn all the partial recursive functions using any acceptable Black Box, and shows that the issue of outright coding does not trivialize the model. It is important to note that these negative results do not depend on the teacher having only black box access to its programming system; the teacher and its Black Box can be chosen arbitrarily (e.g., the programming system can be the standard Turing Machine Programming System), and still there is no learner that can cope with all acceptable Black Boxes.

Chapter 5

Classifying the Successful Learners

Our definition permits a Black Box to be any total recursive function of three arguments; this parallels the generality of Blum's definition of an abstract complexity measure for a programming system, and allows any programming system with a complexity measure to be represented as a Black Box. However, one cost of this generality is that the class of all Black Boxes is not recursively enumerable. Because our motivation is ultimately to gain insight into practical situations involving teaching and learning, we now move away from this generality and restrict our attention to the class of Black Boxes that are not only recursive, but also primitive recursive. In the general case, we are interested in recursively enumerable classes of Black Boxes that have certain additional properties; however, for concreteness we consider the specific class of primitive recursive Black Boxes. Black Boxes derived from many natural programming systems (for example, Turing machines measured by step-counting functions) are very easy to compute; primitive recursive is more than sufficient. In a practical setting, a particular kind of robot might have an action space drawn from a rather limited set of possibilities.

One interesting consequence of restricting the class of Black Boxes for the learner to be primitive recursive is that the learner is able to find, in the limit, a primitive recursive program that is equivalent to its Black Box. This follows from the well-known result that any recursively enumerable class of total recursive functions can be identified in the limit by an inductive inference machine. In particular, the learner enumerates the primitive recursive Black Boxes and compares their values on all triples of inputs with the values returned by its calls to its own Black Box, rejecting any Black Box that does not agree with its own. This ability to gain a certain kind of "self-knowledge" in the limit suggests that the restriction to primitive recursive Black Boxes might make the learner's job considerably easier in general. However, in Chapter 6 we see that this optimism is not borne out.

The restriction to primitive recursive Black Boxes does allow us to prove an approximate converse of Theorem 1, showing that in this case, for each teacher–learner pair such that they are learner-box-proof on the class of all primitive recursive Black Boxes (for P and $\{TBB\}$), there is a partial recursive function b such that the standard teacher T_b^* (equipped with TBB) and the standard learner L^* are as powerful as the given pair, in terms of the class of primitive recursive learner's Black Boxes for which they can learn P.

We define a notation to represent the class of primitive recursive Black Boxes on which a given teacher and a learner "succeed" in this sense. We fix a teacher T, a learner L and a teacher's Black Box TBB and we focus on class G of primitive recursive Black Boxes for L which contains all Black Boxes BB such that the learner $L(BB)$ learns P from the teacher $T(TBB)$. We show that if T and L are learner-box-proof on the class of all primitive recursive Black Boxes for P and $\{TBB\}$ then there exists a total recursive function b such that every Black Box in class G is b-related to

the teacher's Black Box TBB. Intuitively, this result says that if the teacher and the learner can learn all the partial recursive functions and be learner-box-proof, then the learner's Black Box must not be too much slower (as measured by b) than the teacher's. Together with Theorem 1 this result nicely characterizes the relationship between the programming system complexities of a successful teacher–learner pair. Now we present the results and their consequences in a more formal way.

Definition 31 Let T be a teacher, L be a learner, and TBB be a universal Black Box for the teacher. The set of *good* Black Boxes with respect to T, TBB and L is denoted by $G(T, TBB, L)$ and defined to be the set of all those primitive recursive Black Boxes BB for which $L(BB)$ learns P from $T(TBB)$.

As mentioned above, for the sake of the following theorem it is not at all important that $G(T, TBB, L)$ is defined as a class of primitive recursive Black Boxes. It could equally well be any recursively enumerable set of Black Boxes and the theorem would still hold provided that T and L are learner-box-proof on this set. We chose to avoid overgeneralization of the result in order to provide greater compatibility with the rest of the Part 1 of the thesis, where being primitive recursive is an important property of a Black Box, as it can be easily verified given a primitive recursive program that defines it.

Theorem 3 *Let PBBC be the class of all primitive recursive Black Boxes. Let T be a teacher, L be a learner and TBB be a universal Black Box for the teacher such that T and L are learner-box-proof on PBBC for P and* $\{TBB\}$*. Then there exists a total recursive function* $b(x, s)$*, such that every Black Box BB in* $G(T, TBB, L)$ *is b-related to TBB.*

The theorem says that there must be a way to construct a bound b from all the good Black Boxes so that they all are b-related to the teacher's Black Box. By Lemma 4 of Chapter 2, we know that every acceptable Black Box is b-related to every universal Black Box for some total recursive two-argument function b. However, when the translation between the programming systems in these Black Boxes is not known, we do not know how to construct this function b. Here we use the fact that the learner learns all the partial recursive functions from the teacher and that the agents are learner-box-proof on the class of all primitive recursive Black Boxes, and we construct one b which works with all the Black Boxes that are good. The algorithm to compute the bound is given in Figure 5.1.

The algorithm is based on simulations of the teacher and the learner on target functions taken from the teacher's Black Box and using different primitive recursive Black Boxes for the learner. These simulations are done in the subroutine RunThem. When called with parameters T, TBB, $enum(TBB_i)$, L, BB^{bb} and *steps*, this subroutine performs a simulation of the teacher T and the learner L for *steps* steps. RunThem returns the last hypothesis output by the learner, or ?, if the learner does not output one in *steps* steps. The teacher's Black Box used in the simulation is TBB and the learner's Black Box is BB^{bb}. That is, it is the Black Box defined by the primitive recursive function with index bb in some fixed recursive enumeration of all primitive recursive functions. Both T and L share access to the common Input Tape, containing an enumeration of TBB_i. It is not crucial what particular enumeration of TBB_i is used, as long as it is the same whenever we need an enumeration of TBB_i, even for different values of x and s, that is, even in different runs of the algorithm $\mathrm{B}(x, s)$. Therefore, for simplicity, we decide on one particular enumeration, and we use this enumeration for all functions TBB_i in the teacher's Black Box TBB and at

```
B(x, s)
{
    b = 0;

    For (i = 0; i < x; i++)                                   /* The Outer Loop */
        If (TBB‾_i(x) == s)
            For (bb = 0; bb < x; bb++)                        /* The Middle Loop */
            {
                h = RunThem(T, TBB, enum(TBB_i), L, BB^bb, x);

                If (h == ?)
                    Continue;                       /* Reiterate the Middle Loop */

                For (z = 0; BB^bb(h, x, z) == ?; z++)          /* The Inner Loop */
                {
                    h' = RunThem(T, TBB, enum(TBB_i), L, BB^bb, x + z);

                    If (h' ≠ h)
                        Break;                          /* Out of the Inner Loop */

                }

                b = max(b, z);
            }

    Return b;
}
```

Figure 5.1 Algorithm for computing the bound $b(x, s)$

all times. This is the enumeration that we denote by $enum(\cdot)$. For example, it could be the enumeration defined as follows (for every function TBB_i contained in TBB).

Example 1 Enumeration $enum^{TBB}(TBB_i)$ is a listing $e_0, e_1, \ldots$ of elements, where each e_n is defined by

$$e_n \stackrel{\text{def}}{=} \begin{cases} \big(\pi_1(n), TBB_i(\pi_1(n))\big), & \text{if } \overline{TBB}_i(\pi_1(n)) = \pi_2(n); \\ *, & \text{otherwise.} \end{cases}$$

In particular, $enum^{TMBB}(TMBB_i)$ gives what is called elsewhere in the literature a *primitive recursive enumeration* of some partial recursive function TM_i.

Intuitively, the algorithm for computing the bound $b(x, s)$ does the following. Inside the Outer Loop, an "implicit bound" $b'(i, x, s)$ is computed as if the target function TBB_i were fixed. Then with the help of the Outer Loop, it maximizes over the first x i's. The best way to compute the implicit bound would be to take all the good Black Boxes, run the learner with each one of them, wait for the last (and therefore correct) hypotheses, and then maximize over the number of steps the corresponding hypothesis-functions take on input x. Unfortunately, there is no way to take exactly all the good Black Boxes, so we have to consider them all. Maximizing over an infinite set also poses a problem, so in reality the algorithm only considers the first x Black Boxes for L, which it does inside the Middle Loop. Since it cannot tell which hypothesis is right and which is not, and whether there will be a right one at all, the algorithm only simulates T and L for x steps and uses whatever hypothesis was produced by L, if any. In the Inner Loop it waits for the value of the hypothesis-functions on x and at the same time continues simulating T and L. If a different hypothesis is produced, the algorithm abandons the current one. Here we use the assumption that the agents are learner-box-proof, which guarantees that for no Black Box that we try will L converge incorrectly. In other words, every hypothesis will get changed eventually, unless it is a correct one.

Now we present the proof in a more formal way.

Proof: We begin the proof by showing that the function $b(x, s)$, as computed by the algorithm B(x, s), is a total recursive function.

Claim 5 $b(x, s)$ *is total recursive.*

Proof: Obviously, this algorithm can be implemented on a Turing machine, having access to the programs of T and L and to a program returning the values $TBB(i, x, s)$. Thus, the bound is partial recursive.

It is not as obvious that it is total. Both the Outer and the Middle Loops are clearly bounded, but the situation with the Inner Loop is unclear and, in fact, suspicious. All the non-loop statements, however, are either simple and doable in constant time, or they involve bounded loops (the first **If** statement or simulations in RunThem). Therefore, we only need to show that the Inner Loop terminates. This loop continues as long as $BB^{bb}(h, x, z)$ is equal to ?, and z gets incremented with every iteration. First, let us observe that we only get to this loop if the target function TBB_i is defined on x (and, in fact, has complexity s there). Thus, if $BB^{bb}(h, x, z)$ gives ? forever, then h is not the right hypothesis for the target function. Regardless of whether or not the Black Box BB^{bb} is good (with respect to T, TBB and L), we know that L will eventually change h to a different hypothesis because the agents are learner-box-proof on the class of all primitive recursive Black Boxes. But a change of hypothesis causes the algorithm to leave the Inner Loop. Therefore the algorithm necessarily terminates and thus, $b(x, s)$ is total recursive. ■

Now we can prove that $b(x, s)$ satisfies the other requirements of the theorem, namely that every Black Box $BB^{bb} \in G(T, TBB, L)$ is b-related to the teacher's Black Box TBB. Note that bb is an index in some fixed recursive enumeration of all primitive recursive functions for the function that defines BB^{bb}.

Fix an arbitrary bb such that $BB^{bb} \in G(T, TBB, L)$. Fix an arbitrary i and place it on the Answer Tape. Simulate the teacher T with its universal Black Box TBB and the learner L with BB^{bb} on a common Input Tape containing $enum(TBB_i)$. Since

BB^{bb} is good, L will eventually converge to a hypothesis j such that $BB_j^{bb} \geqq TBB_i$. Suppose that this happens in *steps* steps.

Clearly, for all $x > \max(i, bb, steps)$, if $\overline{TBB}_i(x) = s$ then the algorithm simulates the learner $L(BB^{bb})$ with the teacher $T(TBB)$ on the enumeration of the target function TBB_i for enough steps for L to produce the final (and correct) hypothesis j. This implies that property 1 of b-relatedness is satisfied. Having a correct and final hypothesis j, however, causes the algorithm to reiterate the Inner Loop until the complexity of BB_j^{bb} on input x is determined. Therefore, the algorithm sets the bound $b(x, s)$ to at least the value of $\overline{BB_j^{bb}}(x)$, which implies that property 2 of b-relatedness is satisfied as well. This concludes the proof of the theorem. ∎

As mentioned above, without knowing a translation function between a universal and an acceptable programming system, we do not know how to construct the total recursive function b that b-relates them. It is known, however, that such a function exists. The achievement of Theorem 3 is the actual construction of the total recursive bound for all good (with respect to T, TBB and L) primitive recursive Black Boxes (even those that are not acceptable).

We now present some important corollaries. Theorem 1 from Chapter 3 showed that for each total recursive two-argument function $b(x, s)$, we can construct the Standard Teacher (with bound b) and the Standard Learner, such that the learner learns all the partial recursive functions for all Black Boxes that are b-related to the teacher's Black Box, and that the agents are target-learner-box-and-teacher-box-proof. If we combine Theorem 1 with Theorem 3, we have that teacher–learner pairs in the form T_b^* and L^* are as powerful in this setting as any pair T and L that are learner-box-proof on the class of all primitive recursive Black Boxes (for P and some

class $\{TBB\}$).

Corollary 5 *Let PBBC be the class of all primitive recursive Black Boxes. Let L be a learner, T be a teacher and TBB be a universal Black Box for the teacher such that T and L are learner-box-proof on PBBC for P and $\{TBB\}$. Then there exists a total recursive two-argument function $b(x, s)$ such that $G(T, TBB, L) \subseteq G(T^*_b, TBB, L^*)$.*

Proof: Theorem 3 allows us to construct a bound $b(x, s)$, which is exactly what is needed to apply Theorem 1. The result follows immediately. ■

We can derive a type of closure result using Corollary 5. (A variety of interesting closure results are given by Minicozzi for the case of target-proof (i.e., reliable) learning [31].)

Corollary 6 *Let PBBC be the class of all primitive recursive Black Boxes. Let L_1 and L_2 be learners and let T_1 and T_2 be teachers. Let TBB be an arbitrary universal Black Box for both T_1 and T_2 such that both T_1 with L_1 and T_2 with L_2 are learner-box-proof on PBBC for P and $\{TBB\}$. Then there exists a total recursive two-argument function $b(x, s)$ such that $G(T_1, TBB, L_1) \cup G(T_2, TBB, L_2) \subseteq G(T^*_b, TBB, L^*)$.*

Proof: Corollary 5 implies that there is a total recursive two-argument function $b_1(x, s)$, such that $G(T_1, TBB, L_1) \subseteq G(T^*_{b_1}, TBB, L^*)$, and a total recursive two-argument function $b_2(x, s)$, such that $G(T_2, TBB, L_2) \subseteq G(T^*_{b_2}, TBB, L^*)$. Let us define b to be the greater of b_1 and b_2, i.e., $b(x, s) \stackrel{\text{def}}{=} \max(b_1(x, s), b_2(x, s))$. Then we have that all Black Boxes BB that are b_1-related or b_2-related to TBB, are also b-related to it. The result follows. ■

Corollary 6 is analogous to a weakened version of Minicozzi's [31] Union Theorem. We now explain why.

In full generality this theorem asserts that given two inductive inference machines that are reliable (or strong) on some class of partial recursive functions S, one can construct a new machine that is also reliable on S and as powerful (on S) as both of its predecessors combined. If, however, S is replaced with the set of all partial recursive functions P (weakening the theorem), then this theorem follows from the theorem about "A Priori Inference" by Blum and Blum [15]. In our work Theorems 1 and 3 together form a characterization result somewhat similar to that by Blum and Blum. Corollary 6 follows from these theorems, just as the weakened Union Theorem follows from the result of Blum and Blum. Theorem 3 has the obvious disadvantage of working with a restricted class of Black Boxes, namely, the primitive recursive ones. Similarly, the result by Blum and Blum requires reliability on P; reliability on any smaller set does not suffice. These peculiarities suggest there may be other closure results that are stronger than Corollary 6 (i.e., are not restricted to primitive recursive Black Boxes) and that can be proved directly, not via Theorems 1 and 3. We focus on such closure results in the Chapter 7, after we have developed some important negative results.

Chapter 6

Is Everything Easy?

Recall that the restriction to primitive recursive Black Boxes allows the learner to find in the limit a primitive recursive program for its own Black Box. This, and the fact that the counterexample Black Box constructed in the proof of Theorem 2 is not necessarily primitive recursive, make it imperative to address the question:

> *Does there exist a learner L, a teacher T, and a Black Box TBB for the teacher such that $L(BB)$ learns all the partial recursive functions from $T(TBB)$ for all primitive recursive acceptable Black Boxes BB?*

Corollary 4 answered this question in the negative without the restriction to primitive recursive Black Boxes, since it was proved using Theorem 2. However, even with the restriction to primitive recursive Black Boxes, the answer is still "no." Despite knowing (in the limit) how to build the Black Box on its own, the learner still cannot do much without the help of a teacher (or without some more knowledge about the Black Box). In particular, if asked to find the index for the constant all-zero function Z, it cannot do it. The following theorem strengthens Theorem 2 for

the case of primitive recursive Black Boxes.

Theorem 4 *There is no independent learner L such that the learner $L(BB)$ learns the constant all-zero function Z for all primitive recursive acceptable Black Boxes BB.*

Proof: Suppose such a learner L exists. We describe the algorithm DefineBB which, given a program for any independent learner L, defines a Black Box $BB^{(L)}$ such that $L(BB^{(L)})$ cannot find an index for Z, even though $BB^{(L)}$ is primitive recursive and acceptable.

We assume that the algorithm knows the program for L, which is a Turing machine. The algorithm simulates L step by step and observes its behavior, namely, whether it has output a new hypothesis and whether it needs a response from its oracle. DefineBB provides L with such responses. It is given in Figure 6.3 and it uses two subroutines given in Figures 6.1 and 6.2. The subroutine TMDefine, given in Figure 6.1 uses another subroutine TMBB which returns the values of the Turing Machine Black Box *TMBB*. Given arguments i, x and s, this subroutine simulates the Turing machine with index i on input x for at most s steps and returns ? if the machine does not stop or does not produce output before stopping, or a value y if the machine stops with output y.

DefineBB, started with arguments i, x and s, defines the value $BB^{(L)}(i, x, s)$ of the Black Box in the following way. It starts with an infinite three-dimensional array *Table*, each entry of which is initialized to a special value "undefined", not equal to any natural number or ?. We have put the line

$$\mathit{Table}[\infty][\infty][\infty] = \{ \text{ undefined, undefined, } \ldots \};$$

```
TMDefine(beginning_index, size, Table)
{
    For (i = 0; i ⩽ size; i++)
        For (x = 0; x ⩽ size; x++)
            For (s = 0; s ⩽ size; s++)
                If (Table[beginning_index + 2 · i][x][s] == undefined)
                    Table[beginning_index + 2 · i][x][s] = TMBB(i, x, s);
}
```

Figure 6.1 Subroutine TMDefine

```
QuestionDefine(size, Table)
{
    For (i = 0; i ⩽ size; i++)
        For (x = 0; x ⩽ size; x++)
            For (s = 0; s ⩽ size; s++)
                If (Table[i][x][s] == undefined)
                    Table[i][x][s] = ?;
}
```

Figure 6.2 Subroutine QuestionDefine

in the algorithm to indicate that the table is initialized to these values.

Definition 32 Let BB be a Black Box. A *block* of size n is the set of values $BB(i, x, s)$, such that $0 \leqslant i, x, s \leqslant n$.

The algorithm initializes some local variables and after that computes the value $n = \max(i, x, s)$, which is the size of the block to be defined. In order to compute the value $BB^{(L)}(i, x, s)$, the algorithm actually computes a block of size n of values. These values are stored in the table *Table* for easy reference if, for example, L asks a query about one of them. The computation of these values depends on the simulation of L, but basically we can think of it as of four "processes" running simultaneously,

```
DefineBB(i, x, s)
{
    Table[∞][∞][∞] = { undefined, undefined, ... };
    previous_hypothesis = undefined;
    TM_zone = 1;
    max_spoiled_i = 1;
    n = max(i, x, s);
    Initialize the simulation of L;

    For (size = 0; size ⩽ n; size++)
    {
        Perform step size of the simulation of L;

        If (L asks for the value of BB^(L)(j, y, z))
        {

            If (Table[j][y][z] == undefined)
                Table[j][y][z] = ?;                              /* "Direct Process" */

            Give Table[j][y][z] as answer to L;
        }

        If (L outputs a hypothesis h)
            If (h ≠ previous_hypothesis)
            {
                previous_hypothesis = h;

                If (h is odd and h ⩾ TM_zone)
                    TM_zone = max_spoiled_i = max(max_spoiled_i, h) + 2;

                TMDefine(0, 2 · size, Table);                     /* "Even Process" */
            }

        TMDefine(TM_zone, 2 · size, Table);                       /* "Odd Process" */
        max_spoiled_i = TM_zone + 4 · size;
        QuestionDefine(size, Table);                              /* "Slow Process" */
    }

    Return Table[i][x][s];
}
```

Figure 6.3 Algorithm DefineBB

each attempting to define certain values of the Black Box that are not defined yet.

Since we are interested in what the Black Box $BB^{(L)}$ would look like if all its values had been defined, it is convenient to imagine that $n = \infty$ and that the algorithm DefineBB runs forever, defining the values of $BB^{(L)}$. The **For** loop in it insures that it first defines a block of size 0, then a block of size 1, 2, and so on. But first let us discuss separately each of the four processes that each define certain values of the Black Box.

The "Slow Process" is hidden in the subroutine QuestionDefine, given in Figure 6.2. It is the only process that defines a true block of values and it defines all these values to be ? (i.e., all those values which have not been defined yet). It is the last thing called in every iteration of the **For** loop in algorithm DefineBB and it insures that after the iteration numbered *size* at least a block of size *size* has been defined. Unless other processes interfere with the Slow Process, it will define all values of $BB^{(L)}$ as ?, causing all functions contained in the Black Box to be undefined on absolutely all inputs. The other processes are faster than this, however, and thus manage to define some points of the Black Box in a more interesting way, which we discuss next.

There are two "Fast Processes", each hidden in a call to the subroutine TMDefine, given in Figure 6.1. Their goal is to construct all the partial recursive functions somewhere in the Black Box. For this they make use of the subroutine TMBB, which returns the values of the Turing Machine Black Box *TMBB*. The Fast Processes write these values in the corresponding places in the table *Table*, unless they are already defined. The only real difference from the values of *TMBB* is in the function index, i.e., the first variable. These processes only define the values for every other function, and one of them does not define functions with indices smaller than the value

of variable *TM_zone*. We now elaborate on these peculiarities.

One of the Fast Processes is called the "Even Process". It only defines the values of the functions with even indices in the Black Box. It is invoked every time L changes its hypothesis h. Its goal is to insure that if L changes its mind infinitely often, the Black Box $BB^{(L)}$ contains all the partial recursive functions. If L does not output any hypothesis or converges, this process will not be invoked after some time and thus stay suspended forever. If this happens, all but a finite number of functions with even indices will be undefined everywhere (due to the Slow Process, which will get to them eventually). The other functions with even indices will be undefined on all but a finite number of inputs at most (again, due to the Slow Process, which will get to the rest of the inputs eventually).

The other Fast Process is called the "Odd Process". It defines only the values of functions with odd indices in the Black Box and it does so only for indices that are big enough. The algorithm maintains variables *TM_zone* and *max_spoiled_i*, which hold the lowest and highest indices, respectively, of the functions that were defined on more inputs by the last invocation of the Odd Process. Every time L outputs an odd index greater or equal to *TM_zone*, the algorithm DefineBB changes the values of *TM_zone* and *max_spoiled_i*. The next invocation of the Odd Process then tries to define the values of functions with odd indices sufficiently large that they have not been touched by the Odd Process yet and L has not yet made a hypothesis equal to any of them. The goal of the Odd Process is to insure that if L does not output any hypothesis or if it converges to some index i^*, the Black Box $BB^{(L)}$ contains all the partial recursive functions, but all the indices of the constant all-zero function exceed i^* and therefore the hypothesis cannot be correct. In this case the functions with smaller odd indices (i.e., smaller than *TM_zone*) will be undefined on all but

a finite number of inputs (due to the Slow Process, which will get to these inputs eventually). If, on the other hand, L changes its mind infinitely often, then the Odd Process will get invoked with bigger and bigger values of *TM_zone* and will never have a chance to define any function for all inputs. All the functions with odd indices will be undefined on all but a finite number of inputs (due to the Slow Process, again).

Yet another way to define a point in the Black Box is when L directly asks for a value of a point which has not been defined by any of the three processes discussed above. We refer to such a situation as the "Direct Process". If L asks for a value of the Black Box $BB^{(L)}$ which is not defined yet, the algorithm has to provide an answer and so it does. The answer given is always ?, and, of course, it is marked in the table *Table* as such. Thus, the Direct Process fills the table with ?'s, just like the Slow Process. The difference between them is that the Slow Process does it systematically, by filling bigger and bigger blocks with ?'s, while the Direct Process does it in some arbitrary hard-to-predict way. There is a good thing about the ? values in the Black Box, though. If we fix an index for a function i and fix an input x, then unless $BB^{(L)}(i, x, s) = ?$ for all $s \in \mathbb{N}$, there is always a chance to define the function $BB_i^{(L)}$ on input x in any way we like.

Thus we have come to a very important factor in the coexistence of the four processes capable of defining the values of the Black Box, which is their relative speed. We need the Fast Processes to always "be ahead" of the Slow and Direct Processes, each of which try to make all functions undefined everywhere. This is why the Odd Process defines blocks twice as big as the Slow Process and so does the Even Process, except that it sometimes "misses its turn" (since L does not necessarily change its hypothesis in every step) and then catches up rapidly.

We now prove that for every independent learner L, the Black Box $BB^{(L)}$ given

by the algorithm DefineBB is a primitive recursive and acceptable Black Box, and that $L(BB^{(L)})$ does not converge to a correct index for Z. We start with the latter assertion.

Claim 6 *$L(BB^{(L)})$ never converges to a correct index for the all-zero function Z.*

Proof: Suppose L does converge to some index i^*. Furthermore, let s^* be the least s such that i^* was output in step s of L and was never changed afterwards. Here we start counting the steps of L with 0, which means that the value of variable *size* was s^* at the moment this hypothesis i^* was output. In this case, the last invocation of the Even Process is by a call to TMDefine with parameters 0, $2s^*$ and *Table*. It possibly defines the values of functions with even indices in the Black Box where the indices do not exceed $4s^*$ and the inputs do not exceed $2s^*$. The Slow Process eventually defines the rest and therefore all but these first $2s^* + 1$ functions with even indices are undefined on all inputs. The first $2s^* + 1$ functions with even indices are undefined on all but finitely many inputs. Obviously, none of the functions with even indices can be the constant all-zero function Z. Therefore, if i^* is even, it is clearly not an index for the all-zero function. If, however, it is odd, we have to do some further analysis.

When an odd index i^* is output in step s^*, it is different from the previous hypothesis of L by our choice of s^*. Therefore, the algorithm compares it to the current value of *TM_zone*. If $i^* \geqslant$ *TM_zone*, L changes the *TM_zone* to be at least $i^* + 2$. From then on, the Odd Process defines the functions with odd indices that are greater than or equal to *TM_zone*, which is strictly greater than i^*. (Obviously, the same happens in the case when $i^* <$ *TM_zone*, too.) Even though the Odd Process creates many all-zero functions, i^* is clearly too small to be an an index for any of

them. The functions with indices that are odd and less than or equal to i^* become undefined (due to the Slow Process) on all but a finite number of inputs. That is, they may be defined on the first $2(s^* - 1) + 1 = 2s^* - 1$ inputs by the Odd Process but are undefined on the rest of the inputs. Therefore, i^* cannot be equal to an index for the all-zero function Z. ■

We now show that $BB^{(L)}$ is primitive recursive and acceptable. The first claim is easy—it suffices to note that the algorithm DEFINEBB and its subroutines consist entirely of **For** loops with fixed bounds and of bounded simulations of Turing machines. Therefore it describes a primitive recursive function.

It only remains to prove that the Black Box $BB^{(L)}$ is acceptable; that is, it contains all the partial recursive functions (i.e., is full and, therefore, by Lemma 1 universal) and there exists a total recursive composition function for the programming system in $BB^{(L)}$.

The following claim greatly simplifies the proof that $BB^{(L)}$ is a universal Black Box.

Claim 7 *Let Tablerow be an infinite one-dimensional array (indexed from* 0*). Let* D*,* F *and* S *be processes capable of defining values of this array. Let there be an infinite loop* **For**($size = 0$; ;$size$++) *invoking the processes in the following manner.*

Each iteration starts with process D *which defines one value* $Tablerow[s]$ *for some* s *(unless already defined). After that some arbitrary event determines whether process* F *gets invoked or does not. It is guaranteed, however, that* F *will be invoked infinitely many times. If invoked,* F *defines the first* $2 \cdot size + 1$ *values of Tablerow excluding the ones that are defined already (i.e., it defines the undefined values*

Tablerow$[i]$ *such that* $0 \leqslant s \leqslant 2 \cdot size$*). Each iteration of the loop concludes with an invocation of process* S *which defines the first* $size + 1$ *values of Tablerow excluding the ones that are defined already (i.e., it defines the undefined values Tablerow*$[i]$ *such that* $0 \leqslant s \leqslant size$*).*

Then for all natural numbers s_0*, there is at least one value Tablerow*$[s]$*, such that* $s \geqslant s_0$ *and it was defined by process* F*.*

Before proceeding to the proof of this claim, let us explain its relationship to the algorithm DefineBB and the proof of Theorem 4. Claim 7 focuses on a scenario where the Slow and Direct Processes have allied to overcome one of the Fast Processes. They have fixed some i and x and are trying to make sure that all values $Table[i][x][s]$ are defined by them as ?. If they succeed, the function $BB_i^{(L)}$ will be undefined on input x. By choosing the values i and x appropriately, they could undermine the goal of either of the Fast Processes to insure that every partial recursive function is contained in the Black Box $BB^{(L)}$.

Quite obviously, process D in this claim corresponds to the Direct Process of algorithm DefineBB. Similarly, process S corresponds to the Slow Process and process F can be either of the Fast Processes. If F is the Odd Process, then it is invoked in every iteration. If it is the Even Process, then we do not know whether it will or will not be invoked in a particular iteration. However, the assumption that the fast process will be invoked infinitely many times is valid, as will be shown in the proof of Claim 8.

Proof: Assume that the claim does not hold. Let us look at an iteration of the loop such that $size \geqslant s_0$ and process F does get invoked in it. Consider the situation just before the invocation of F. Denote the current value of $size$ by $size_0$. In this

and previous iterations of the loop, process D may have defined at most $size_0 + 1$ values of $Tablerow$. In addition, process S may have defined at most $size_0$ values with indices ranging from 0 to $size_0 - 1$. At this moment process F gets invoked and it attempts to define all the undefined values $Tablerow[s]$, where $0 \leqslant s \leqslant 2 \cdot size_0$. In particular, there are $size_0 + 1$ values such that $size_0 \leqslant s \leqslant 2 \cdot size_0$, which cannot have been defined by process S yet. We distinguish two cases.

1. If not all of the values $Tablerow[s]$ such that $size_0 \leqslant s \leqslant 2 \cdot size_0$ have been defined by process D, then process F actually defines at least one of them and we are done.

2. Otherwise, i.e., if all of the entries $Tablerow[s]$ such that $size_0 \leqslant s \leqslant 2 \cdot size_0$ have been defined by process D, we have to look at a later iteration of the **For** loop where process F is invoked again. Denote the current value of $size$ by $size_1$. Clearly, $size_1 > size_0 \geqslant s_0$. In this and previous iterations, process D may have defined at most $size_1 + 1$ values of $Tablerow$. Process S may have defined at most $size_1$ values of $Tablerow[i]$, with indices ranging from 0 to $size_1 - 1$. Process F attempts to define all the undefined values $Tablerow[s]$, where $0 \leqslant s \leqslant 2 \cdot size_1$. In particular, there are $size_1 + 1$ values such that $size_1 \leqslant s \leqslant 2 \cdot size_1$, which cannot have been defined by process S yet. In this iteration of the loop it is impossible that all of them have been defined by process D since it has defined at most $size_1 + 1$ values and one of those values definitely is $Tablerow[size_0]$. Therefore, process F will define at least one of the values such that $size_1 \leqslant s \leqslant 2 \cdot size_1$.

We have proved that sooner or later process F will define a value $Tablerow[s]$ such that $s \geqslant s_0$. Therefore the claim holds. ■

Now we prove that the Black Box BB is universal. By Lemma 1 it suffices to show the following.

Claim 8 *The Black Box $BB^{(L)}$ contains every partial recursive function.*

Proof: Let us pick an arbitrary index i. We will show that the partial recursive function TM_i, defined by the Turing machine with index i, is contained in $BB^{(L)}$. That is, we need to find an index i' such that for every x:

1. If the value $TM_i(x)$ is defined and equal to y then there exists s' such that $BB^{(L)}(i', x, s') = y$ and $BB^{(L)}(i', x, s) = ?$, for all $s < s'$;
2. If $TM_i(x)$ is undefined, then $BB^{(L)}(i', x, s) = ?$, for all s.

First, let us suppose that L changes its mind infinitely often. In this case, the Even Process gets invoked infinitely many times and keeps defining bigger and bigger blocks of the functions with even indices. The function TM_i has index $i' = 2i$ in the Black Box $BB^{(L)}$. Let us verify this. Pick an arbitrary x. If $TM_i(x)$ is undefined, then simulating the Turing machine i on data x for any number of steps will yield nothing and the call to the subroutine TMBB with arguments $2i$, x and s will return ? for any value of s. Therefore, for any s, the value of $Table[2i][x][s]$ will be ? if first defined by the Even Process and also ? if first defined by the Slow or the Direct Processes. (The Odd Process cannot define functions with even indices.)

If, however, $TM_i(x) = y$, then let s_0 be the least s such that at step s the Turing machine with index i outputs y when run on input x. Then $TMBB(i, x, s) = y$ for all $s \geqslant s_0$. The entry $Table[2i][x][s]$ will be ? for all $0 \leqslant s < s_0$, regardless of which process defined it. For all $s \geqslant s_0$, however, the value of $Table[2i][x][s]$ will

be y if defined by the Even Process, or ? if defined by the Slow or Direct Process. (The Odd Process cannot define functions with even indices). By Claim 7 we know that the Slow and Direct Processes cannot define all entries $Table[2i][x][s]$, where $s \geqslant s_0$, which means that there will be an $s' \geqslant s_0$ such that $Table[2i][x][s'] = y$ and $Table[2i][x][s] = ?$ for all $s \leqslant s'$. Therefore $BB^{(L)}_{2i}(x) = y = TM_i(x)$ and we are done with the case when L never converges.

Now, let us suppose that L converges to some index i^*. Let s^* be the least s, such that i^* is output in step s of L and is never changed afterwards. Recall that we start counting the steps of L with 0, which means that the value of variable *size* was s^* at the moment this hypothesis i^* was output. If the hypothesis is never changed after step s^*, then the variable *TM_zone* may be changed in this iteration of the **For** loop, but not in any later one. Let t denote the final value of *TM_zone*. Recall that *TM_zone* is odd at all times. Also note that every time when a new value of *TM_zone* is chosen, it is taken larger than any k such that the Odd Process has accessed $Table[k][x][s]$, for any x and s. Variable *max_spoiled_i* controls this choice.

The function TM_i has index $i' = t+2i$ in the Black Box $BB^{(L)}$. Let us verify this. Pick an arbitrary x. If $TM_i(x)$ is undefined, then simulating the Turing machine with index i on input x for any number of steps will yield nothing. Therefore, for every s the entry $Table[t + 2i][x][s]$ will be ? if first defined by the Odd Process and also ? if first defined by the Slow or Direct Processes. (The Even Process cannot define functions with odd indices.)

If, however, $TM_i(x) = y$, then let s_0 be the least s such that at step s the Turing machine with index i outputs y when run on input x. Then $TMBB(i, x, s) = y$ for all $s \geqslant s_0$ by the definition of the Turing Machine Black Box. The value of $Table[t + 2i][x][s]$ will be ?, for all $0 \leqslant s < s_0$, regardless of which process defined it.

For all $s \geqslant s_0$, however, the value of $\mathit{Table}[t+2i][x][s]$ will be y if defined by the Odd Process or ?, if defined by the Slow or Direct Processes. (The Even Process cannot define functions with odd indices.) By Claim 7 we know that the Slow and Direct Processes cannot define all entries $\mathit{Table}[t+2i][x][s]$, where $s \geqslant \max(s_0, s^*)$, which means that there will be an $s' \geqslant \max(s_0, s^*)$ such that $\mathit{Table}[t+2i][x][s'] = y$ and $\mathit{Table}[t+2i][x][s] = ?$ for all $s \leqslant s'$. Therefore $BB^{(L)}_{t+2i}(x) = y = TM_i(x)$ and we are done with the case when L converges to some index i^*.

The case when L never outputs any hypothesis is essentially analogous to the previous case. The function TM_i has index $i' = 1 + 2i$ in the Black Box $BB^{(L)}$ since the final and only value of the variable *TM_zone* is 1.

This completes the proof that the Black Box BB as defined by the algorithm DEFINEBB is universal. ■

And finally, it remains to prove that the programming system in $BB^{(L)}$ is an acceptable one, i.e., that there exists a total recursive composition function for it.

Claim 9 *There exists a total recursive function C such that $BB^{(L)}_{C(\langle i,j\rangle)} = BB^{(L)}_i \circ BB^{(L)}_j$, for every i and j.*

Proof: In other words, we have to prove that for every i and j, we can find an index $C(i,j)$ such that for every x, if $BB^{(L)}_j(x)$ is defined and equal to y and if also $BB^{(L)}_i(y)$ is defined and equal to z, then $BB^{(L)}_{C(i,j)}(x) = z$, while in all other cases $BB^{(L)}_{C(i,j)}(x)$ is not defined.

Let UNIV denote the universal function for the programming system in $BB^{(L)}$. (Recall that every universal programming system has a partial recursive universal

```
COMPOSE(x)
{
    y = UNIV(⟨j, x⟩);
    Return UNIV(⟨i, y⟩);
}
```

Figure 6.4 Algorithm for the composition function

function and therefore UNIV is partial recursive.) In Figure 6.4 we have illustrated an algorithm with built in (i.e., "hard-coded") values of i and j, which takes an argument x and finds the value of $BB_j(BB_i(x))$, unless it is undefined. This algorithm uses UNIV as a subroutine.

It is obvious that for any i and j this algorithm is easy to implement on a Turing machine. Thus, there is a total recursive function c such that $c(i,j)$ is an index for a Turing machine that implements the above algorithm. We need to find an index $C(i,j)$ such that $BB^{(L)}_{C(i,j)} = TM_{c(i,j)}$. For this we need to know whether $L(BB^{(L)})$ converges and, if so, what is the final value of the variable *TM_zone* in algorithm DEFINEBB. As described in the proof of Claim 8, the required index is $C(i,j) = 2 \cdot c(i,j)$ if $L(BB^{(L)})$ changes its mind infinitely often, and $C(i,j) = t + 2 \cdot c(i,j)$ if $L(BB^{(L)})$ does not output any hypothesis or converges and the final value of *TM_zone* in DEFINEBB is t. Note that we cannot tell which of the total recursive functions C is the right one, but for each possible behavior of the algorithm $L(BB^{(L)})$ there definitely is one. ■

This completes the proof of Theorem 4. ■

We now list the corollaries of Theorem 4. The first two strengthen Corollary 3 and Corollary 4, respectively.

Corollary 7 *There is no learner L, no teacher T and no Black Box TBB for the teacher such that the learner L(BB) learns the constant all-zero function Z from the teacher T(TBB) for all primitive recursive acceptable Black Boxes BB.*

Proof: If for some teacher T and some teacher's Black Box TBB there exists such a learner L, then there exists an independent learner $L'(BB)$ which simulates both $T(TBB)$ and $L(BB)$ and thus learns Z for all primitive recursive acceptable Black Boxes BB. ■

Corollary 8 *There is no learner L, no teacher T and no Black Box TBB for the teacher such that the learner L(BB) learns P from the teacher T(TBB) for all primitive recursive acceptable Black Boxes BB.*

Proof: Since there is no learner, teacher and Black Box such that the learner learns the constant all-zero function Z from the teacher, there cannot be one that learns all partial recursive functions. ■

Corollary 9 *There is no total recursive function b and no universal Black Box TBB such that $L^*(BB)$ (i.e., the Standard Learner equipped with a Black Box BB) learns P from $T^*_b(TBB)$ (i.e., the Standard Teacher with bound b, equipped with a Black Box TBB) for all primitive recursive acceptable Black Boxes BB.*

Proof: This is a special case of Corollary 8. ■

Corollary 10 *For every universal Black Box TBB and for every total recursive two-argument function $b(x, s)$ there exists a primitive recursive acceptable Black Box BBB which is not b-related to TBB.*

Proof: Suppose to the contrary that there exists a universal Black Box *TBB* and a total recursive function $b(x, s)$ such that every primitive recursive acceptable Black Box is b-related to *TBB*. Then, applying Theorem 1, for every Black Box *BB* that is b-related to *TBB*, the Standard Learner $L^*(BB)$ learns all the partial recursive functions from the Standard Teacher (with bound b) $T_b^*(TBB)$. In particular, for this b, $L^*(BB)$ learns all the partial recursive functions from $T_b^*(TBB)$ for every primitive recursive acceptable Black Box *BB*, because of our assumption that they are all b-related to *TBB*. This contradicts Corollary 9. ■

Corollary 10 says that for every universal teacher's Black Box and every total recursive function b, there is some "bad" but primitive recursive and acceptable Black Box for the learner. We call it bad because it is not b-related to the teacher's Black Box. Furthermore, since this Black Box is both acceptable and primitive recursive, it must not be inherently awkward. In fact, it is quite "normal" according to the functions and measures that it contains. In particular, it can be translated to and from the most natural Black Boxes (including the Turing Machine Black Box), and the measures that it contains are recursively related to the most natural ones. In other words, it is b'-related to any other acceptable Black Box by some function b'. Its only drawback must be that the measures that it contains are not good enough for the particular teacher's Black Box *TBB* and the particular function b.

The proof of Corollary 10 only proves the existence of a bad Black Box for every total recursive two-argument function $b(x, s)$. It is possible, however, to give a constructive proof for this result. That is, given an index for a Turing machine computing b, we can construct an algorithm that defines the bad Black Box. The construction is quite simple if $TBB \equiv TMBB$ and such an approach easily proves the corollary, if restricted only to acceptable Black Boxes *TBB*. A direct and construc-

tive proof of the more general case is harder and the simpler proof is recommended as an introduction to it. Both these proofs are given in the Appendix to Part 1, pages 110 and 115, respectively.

Finally, we observe that for every universal Black Box TBB and every total recursive function b_0, there is some bad but primitive recursive and acceptable Black Box $BB^{(0)}$ for the learner, such that $L^*(BB^{(0)})$ cannot learn all the partial recursive functions from $T^*_{b_0}(TBB)$. However, because $BB^{(0)}$ is acceptable, it is b_1-related to TBB for some (larger) total recursive function b_1, which means that $L^*(BB^{(0)})$ can learn all the partial recursive functions from $T^*_{b_1}(TBB)$. However, for b_1 there is another bad but primitive recursive and acceptable Black Box $BB^{(1)}$ for the learner, for which $L^*(BB^{(1)})$ cannot learn all the partial recursive functions from $T^*_{b_1}(TBB)$. Continuing in this way, there exists an infinite sequence of "worse and worse" but still primitive recursive and acceptable Black Boxes for the learner.

Chapter 7

Building More Powerful Teachers and Learners

In this chapter we prove three closure results analogous to Minicozzi's [31] Union Theorem by giving one fairly general construction. Afterwards we show how to extend this construction to prove even stronger versions of these theorems.

Let us start by presenting one of the three union theorems, and, in particular, the one that generalizes Corollary 6.

Theorem 5 *Let TBBC, BBC_0 and BBC_1 be three classes of Black Boxes. Let C be a class of partial recursive functions. Let L_0 and L_1 be learners and let T_0 and T_1 be teachers such that:*

1. *$L_0(BB)$ learns C from $T_0(TBB)$ for all $BB \in BBC_0$ and $TBB \in TBBC$ and the agents are learner-box-proof for C and TBBC;*
2. *$L_1(BB)$ learns C from $T_1(TBB)$ for all $BB \in BBC_1$ and $TBB \in TBBC$ and*

the agents are learner-box-proof for C and $TBBC$.

Then there exists a teacher T and a learner L such that $L(BB)$ learns C from $T(TBB)$ for all $BB \in BBC_0 \cup BBC_1$ and $TBB \in TBBC$ and the agents are learner-box-proof for C and $TBBC$.

Proof: The main ideas of the proof are the following. The learner L simulates both learners L_0 and L_1 so that they each have exactly the same environment as when running on their own. That is, they each have their own copies of L's Input, Message, Work, Oracle and Output Tapes. The same holds for the teacher T simulating T_0 and T_1, i.e., they have exact copies of T's Input, Message, Work, Oracle and Answer Tapes. The learner L then monitors the Output Tapes of L_0 and L_1 and chooses its own hypothesis depending on what the simulated learners output. Furthermore, T and L must collaborate in such a way that the T_0 and L_0 pair is being simulated in parallel with the T_1 and L_1 pair. If this were not the case, problems could arise due to the fact that one or both pairs of agents can output nothing and abandon all communication among themselves, thus avoiding converging incorrectly, which is an acceptable behavior if the environment is not as intended for learning.

The algorithm for the learner L is given in Figure 7.1. It uses a subroutine HypoDecide given in Figure 7.2. The algorithm for the teacher T is given in Figure 7.3.

We first explain how T and L as given by the algorithms UnionTeacher and UnionLearner perform the simulations of T_0, T_1, L_0 and L_1. We will refer to T and L as "agents" and to T_0, T_1, L_0 and L_1 as "sub-agents" in this discussion. At any time one of the agents T and L is suspended and the other is not, as defined by the model. Initially the teacher is suspended. The agents can perform only a few

```
UnionLearner()
{
    suspended[2] = { 0, 0 };
    step[2] = { 0, 0 };
    Initialize the simulation of L_0;
    Initialize the simulation of L_1;

    For (which = 0; ; which = 1 − which)
    {

        If (suspended[which] == 0)
        {
            Perform step step[which] of the simulation of L_which;
            step[which] = step[which] + 1;

            If (L_which outputs a hypothesis h)
                HypoDecide(which, h);

            If (L_which enters the send state)
            {
                Let message be the contents of L_which's Message Tape;
                Replace ? and ? with which and message on the Message Tape;
                suspended[which] = 1;
            }

        }

        Enter the send state;                             /* Get Suspended */
        Read i and message from the Message Tape;   /* On Resuming Comp. */
        Replace i and message with ? and ? on the Message Tape;

        If (i ≠ ?)
        {
            Clear the Message Tape of L_i and write message on it;
            suspended[i] = 0;
        }

    }

}
```

Figure 7.1 Algorithm UnionLearner

```
HypoDecide(which, h)
{
    static previous_hypothesis[2] = { undefined, undefined };
    static last_by_whom = undefined;

    If (previous_hypothesis[which] == h)
        Return;

    previous_hypothesis[which] = h;

    If (previous_hypothesis[1 − which] == undefined)
    {
        last_by_whom = which;
        Output h;
        Return;
    }

    If (last_by_whom ≠ which)
        Return;

    last_by_whom = 1 − which;
    Output h;
    Output previous_hypothesis[last_by_whom];
}
```

Figure 7.2 Subroutine HypoDecide

actions every time that their computation is resumed (or started initially), before becoming suspended again. In particular, they can:

1. Copy the contents of the Message Tape to the Message Tape of one of the sub-agents;
2. Simulate one of the sub-agents;
3. Copy the contents of the Message Tape of the simulated sub-agent to the Message Tape.

It is not guaranteed, however, that any of these actions will be performed, as each action is governed by certain variables and conditions.

```
UnionTeacher()
{
    suspended[2] = { 1, 1 };
    step[2] = { 0, 0 };
    Initialize the simulation of T_0;
    Initialize the simulation of T_1;

    For (which = 0; ; which = 1 − which)
    {
        Read i and message from the Message Tape;
        Replace i and message with ? and ? on the Message Tape;

        If (i ≠ ?)
        {
            Clear the Message Tape of T_i and write message on it;
            suspended[i] = 0;
        }

        If (suspended[which] == 0)
        {
            Perform step step[which] of the simulation of T_which;
            step[which] = step[which] + 1;

            If (T_which enters the send state)
            {
                Let message be the contents of T_which's Message Tape;
                Replace ? and ? with which and message on the Message Tape;
                suspended[which] = 1;
            }

        }

        Enter the send state;                                /* Get Suspended */
        Continue;                 /* On Resuming Computation, Reiterate Loop */
    }

}
```

Figure 7.3 Algorithm UnionTeacher

As long as neither L_0 nor L_1 have entered their send states, the computation of T and L proceeds as follows. Each time when L's computation is resumed, it simulates one step of either L_0 or L_1—variable *which* controls which of the sub-agents is simulated. The next time when L's computation is resumed, *which* has changed its value, thus causing the other sub-agent to be simulated. Similarly, the teacher T has a variable *which* that lets it alternate between simulating the sub-agents T_0 and T_1. Initially, both these sub-agents are suspended and therefore do not get simulated. The messages exchanged between T and L on the message tape consist of two ?'s.

When either one of L's sub-agents enters its send state, the contents of its Message Tape are copied to the Message Tape of L, and are prepended by the index of the sub-agent, held in *which* at this moment. The sub-agent who entered the send state is marked as suspended. At this point, when T's computation is resumed, it reads the index i that comes first on the Message Tape, then copies the remaining contents of the Message Tape to the Message Tape of T_i and marks T_i as not suspended (let us denote the current value of i by i' at this moment, since we will need to refer to it later when i will have changed). From now on $T_{i'}$ will get simulated every other time that teacher's computation is resumed (i.e., whenever teacher's variable *which* is equal to i'), and as long as $T_{i'}$ has not entered its send state. $L_{i'}$, however, will not be simulated as it is now marked as suspended. Similarly, if the other learner's sub-agent enters its send state, then it is marked as suspended and the corresponding teacher's sub-agent is marked as unsuspended. In this case, both learner's sub-agents become suspended and both teacher's sub-agents unsuspended. When either of the teacher's sub-agents enters its send state (which is only possible if it is unsuspended and being simulated at the moment), it is marked as suspended and the corresponding learner's

sub-agent unmarked. Thus, the T_0 and L_0 pair and the T_1 and L_1 pair are simulated in parallel, and within each pair, one of the sub-agents is suspended and the other is not.

Let us now look at what happens when L is equipped with a Black Box $BB \in BBC_0 \cup BBC_1$ and T is equipped with a Black Box $TBB \in TBBC$, and the target function is $f \in C$. If $BB \in BBC_0$ then L_0 and T_0 correctly learn the target function. That is, L_0 eventually converges to an index h such that $BB_h \geqq f$. The other pair of sub-agents, L_1 and T_1, are not required to learn this target function in this case but they are required not to converge incorrectly because they are learner-box-proof for C and $TBBC$. Similarly, if $BB \in BBC_1$ then L_1 and T_1 correctly learn the target function, while L_0 and T_0 may or may not learn it, but at least they do not converge incorrectly. One more case that is of interest to us is when $BB \notin BBC_0 \cup BBC_1$. In this case neither of the sub-agent pairs has to learn the target function but both are required not to converge incorrectly.

Armed with this knowledge we can now see whether L converges to some hypothesis or does not, for each of the cases mentioned above. The decision of whether to output some hypothesis or not is done in the subroutine HypoDecide, which is called every time after one of the learner's sub-agents outputs a hypothesis. This subroutine keeps track of the previous hypothesis (if any) output by each of the sub-agents. For this purpose it has an array *previous_hypothesis*. It also keeps track of which sub-agent's hypothesis was last chosen to be output by L itself, if any. The respective sub-agent's index is kept in the variable *last_by_whom*. Both the array and the variable are `static`, meaning that they get initialized to the values given in the pseudo-code (i.e., "undefined"), but in every subsequent invocation of the subroutine they retain their values from the previous invocation. In this respect, they resemble

global variables.

The subroutine takes two arguments, *which* and h. These arguments denote the index and the hypothesis output by a sub-agent. First, HypoDecide compares h to the previous hypothesis output by L_{which} to see whether there has been a change in hypothesis for L_{which}. The first hypothesis output by L_{which} is treated as a changed hypothesis by this test. If there is no change then this hypothesis can be ignored, so the subroutine just returns. Otherwise, there has been a change in hypothesis and thus some further processing is required. First, the hypothesis h is saved in *previous_hypothesis*[*which*] for use in later calls to HypoDecide. Then the subroutine checks whether there is any hypothesis already output by the other sub-agent, $L_{1-which}$. If there is none, then the same hypothesis h is output by L, the index *which* of the sub-agent who output it originally is saved in *last_by_whom* and the subroutine returns. Otherwise, there is some hypothesis output by the other sub-agent and the learner L may decide to use that hypothesis.

This requires more explanation. As long as neither of the sub-agents L_0 and L_1 have output hypotheses, L does not output any hypothesis either. When one of the sub-agents outputs a hypothesis (as seen from the fact that it writes the special marker on its Output Tape and moves to the right), L outputs the same hypothesis. Then, in general, it holds on to this hypothesis as long as it is not changed by the sub-agent who output it. At this point, usually, it outputs the hypothesis output by the other sub-agent and, again, holds it until it is changed by this sub-agent. Since both sub-agents are required to either converge to the correct index or not to converge at all, this strategy is meant to insure that L changes its mind infinitely often if both of its sub-agents do so, that it does not output any hypothesis if neither of its sub-agents do, and that it converges to the correct index if at least one sub-agent

converges. The greater complexity of the subroutine HypoDecide comes merely from various exceptions to the overall principle. For example, special attention has to be paid to the case when one or both sub-agents have not output any hypothesis yet, when the change of hypothesis is done by the sub-agent whose hypothesis is not currently held on to by L, or when switching to the other sub-agent's hypothesis does not constitute a change in L's hypothesis.

Our analysis of HypoDecide has now reached the third **If** statement in the pseudo-code. This is the test where the subroutine checks whether the change of hypothesis done by L_{which} warrants switching to this newest hypothesis. If *last_by_whom* is not equal to *which*, i.e., if L has last output the hypothesis produced by the other sub-agent $L_{1-which}$ and is currently holding on to it, then there is no reason to switch. Intuitively, at this point it seems to L that $L_{1-which}$ may be converging while L_{which} is changing its mind, thus it keeps holding on to the hypothesis of $L_{1-which}$ which looks more credible. In this case there is nothing more to do, since the last hypothesis output by L_{which} was already saved before, and the subroutine just returns.

Finally, if the subroutine has not returned yet, we know the following. The sub-agent L_{which}, whose last hypothesis was so far the one output and held on to by L, has output a new, different hypothesis h, already saved in $previous_hypothesis[which]$. The other sub-agent $L_{1-which}$ has also produced some hypotheses before and the last one of those is saved in $previous_hypothesis[1 - which]$. Intuitively, at this moment it looks like L_{which} is changing its mind and $L_{1-which}$ is the one who may be on its way to converging. Thus, the subroutine marks that the last hypothesis output by L is now the one that was last output by $L_{1-which}$ (by saving $1 - which$ in *last_by_whom*) and really outputs the hypothesis from $previous_hypothesis[last_by_whom]$, which equals $previous_hypothesis[1 - which]$. Before doing so, however, it outputs the hypothesis

h, to force a hypothesis change. This avoids the following undesirable scenario.

Suppose L_{which} had output the hypothesis h' which was also output and held on to by L until the current invocation of HypoDecide where it was discovered that L_{which} has now output a new, different hypothesis h. Suppose also that the other learner $L_{1-which}$ has last output the hypothesis h'. If L just switched to the hypothesis output by $L_{1-which}$, it would output another h', i.e., no change in L's hypothesis would happen. This could continue indefinitely, e.g., L_{which} could now output h' and then $L_{1-which}$ could output h and L would switch to the other sub-agent's hypothesis, i.e., output yet another h'. In other words, unless we force a hypothesis change for L, the simple scheme described above may lead to a situation where both sub-agents change their minds infinitely often, but L nevertheless converges, possibly incorrectly. Proceeding as given by the pseudo-code, there is always a hypothesis change since it was discovered earlier that L_{which} is the learner whose last hypothesis was output and held on to by L and that L_{which} really performed a change of hypothesis by outputting h. Thus, L changes its hypothesis by outputting h and then possibly changes it again by outputting $previous_hypothesis[last_by_whom] = previous_hypothesis[1 - which]$.

To summarize, the subroutine HypoDecide achieves the following. If L ever converges, then at least one of the simulated learners must have converged, namely, the one whose index is saved in the variable *last_by_whom*, and which produced the last hypothesis output by L. If neither L_0 nor L_1 converges, then L cannot converge due to the forced hypothesis change that it always performs before changing to the hypothesis produced by the other sub-agent. Since both L_0 with T_0 and L_1 with T_1 are learner-box-proof for C and *TBBC*, neither L_0 nor L_1 can converge incorrectly. Thus, L cannot converge incorrectly either, making L and T learner-box-proof for C and *TBBC*, as required by the theorem. Furthermore, if L does not converge, then

neither L_0, nor L_1 can have converged, since if one of them or both converged, so would L. When $BB \in BBC_0 \cup BBC_1$, at least one of the sub-agents L_0 and L_1 must converge correctly and therefore so must L, meaning that L and T learn C for all learners Black Boxes $BB \in BBC_0 \cup BBC_1$ and teachers Black Boxes $TBB \in TBBC$.

■

Note that for the sake of this theorem we could allow that T and L be learner-box-proof on some other class of Black Boxes (not on all possible Black Boxes) and the theorem would still hold as long as this class is a superset of $BBC_0 \cup BBC_1$.

The construction presented in the proof of Theorem 5 can also be used to prove two other union theorems. We now present them both, and give brief explanations of why the same construction works.

Theorem 6 *Let $TBBC_0$, $TBBC_1$ and BBC be three classes of Black Boxes. Let C be a class of partial recursive functions. Let L_0 and L_1 be learners and let T_0 and T_1 be teachers such that:*

1. *$L_0(BB)$ learns C from $T_0(TBB)$ for all $TBB \in TBBC_0$ and $BB \in BBC$ and the agents are teacher-box-proof for C and BBC;*
2. *$L_1(BB)$ learns C from $T_1(TBB)$ for all $TBB \in TBBC_1$ and $BB \in BBC$ and the agents are teacher-box-proof for C and BBC.*

Then there exists a teacher T and a learner L such that $L(BB)$ learns C from $T(TBB)$ for all $TBB \in TBBC_0 \cup TBBC_1$ and $BB \in BBC$ and the agents are teacher-box-proof for C and BBC.

Proof: Since both L_0 with T_0 and L_1 with T_1 are teacher-box-proof for C and BBC, they cannot converge incorrectly if $f \in C$ and $BB \in BBC$. This means that L and T are also teacher-box-proof for C and BBC. Furthermore, if $f \in C$, $BB \in BBC$ and $TBB \in TBBC_0 \cup TBBC_1$, then at least one of the sub-agents L_0 and L_1 converges correctly. Therefore, T and L learn C for all $BB \in BBC$ and $TBB \in TBBC_0 \cup TBBC_1$. Again, for this theorem it was not important that T and L were teacher-box-proof on the class of all Black Boxes. ■

Theorem 7 *Let $TBBC$ and BBC be two classes of Black Boxes. Let C_0, C_1 and C' be classes of partial recursive functions such that $C_0 \cup C_1 \subseteq C'$. Let L_0 and L_1 be learners and let T_0 and T_1 be teachers such that:*

1. *$L_0(BB)$ learns C_0 from $T_0(TBB)$ for all $BB \in BBC$ and $TBB \in TBBC$ and the agents are target-proof on C' for BBC and $TBBC$;*
2. *$L_1(BB)$ learns C_1 from $T_1(TBB)$ for all $BB \in BBC$ and $TBB \in TBBC$ and the agents are target-proof on C' for BBC and $TBBC$;*

Then there exists a teacher T and a learner L such that $L(BB)$ learns $C_0 \cup C_1$ from $T(TBB)$ for all $BB \in BBC$ and $TBB \in TBBC$ and the agents are target-proof on C' for BBC and $TBBC$.

Proof: Since both L_0 with T_0 and L_1 with T_1 are target-proof on C' for BBC and $TBBC$, they cannot converge incorrectly if $f \in C'$, $BB \in BBC$ and $TBB \in TBBC$. This means that L and T are also target-proof on C' for BBC and $TBBC$. Furthermore, if $f \in C_0 \cup C_1$, $BB \in BBC$ and $TBB \in TBBC$, then at least one of the sub-agents L_0 and L_1 converges correctly. Therefore, T and L learn $C_0 \cup C_1$ for all $BB \in BBC$ and $TBB \in TBBC$. ■

Theorem 7 is our closest analogue to the Union Theorem by Minicozzi [31]. The statement of the theorem is essentially the same, although the terminology and the model differ somewhat. The proof of our union theorems is slightly more intricate, since we cannot use the same method of choosing the hypotheses of simulated learners. Minicozzi first transforms the simulated machines so that they give only monotonically increasing hypotheses and always chooses the smaller of the last hypotheses output. Because our learners work with arbitrary Black Boxes, they in general have no way of finding a bigger index for the same function in the Black Box. A padding function is not in general known, even if it exists. This difficulty was overcome using our method of holding the hypothesis that does not change.

Like Minicozzi's theorem, our three union theorems can be generalized for infinite but countable unions. We now give the statements of all the generalized theorems.

Theorem 5′ *Let TBBC be a fixed class of Black Boxes and C be a fixed class of partial recursive functions. Let* BBC_0, BBC_1, $\ldots$ *be a sequence of classes of Black Boxes. Let* $T_0, T_1, \ldots$ *be a recursively enumerable sequence of teachers and* $L_0, L_1, \ldots$ *be a recursively enumerable sequence of learners. Assume that for each* $i \in \mathbb{N}$,

- $L_i(BB)$ *learns C from* $T_i(TBB)$ *for all* $BB \in BBC_i$ *and* $TBB \in TBBC$, *and*
- L_i *and* T_i *are learner-box-proof for C and TBBC.*

Then there exists a teacher T and a learner L such that $L(BB)$ *learns C from* $T(TBB)$ *for all* $TBB \in TBBC$ *and* $BB \in \bigcup_{i\in\mathbb{N}} BBC_i$ *and the agents T and L are learner-box-proof for C and TBBC.*

Theorem 6′ *Let BBC be a fixed class of Black Boxes and C be a fixed class of partial recursive functions. Let* $TBBC_0$, $TBBC_1$, $\ldots$ *be a sequence of classes of*

Black Boxes. Let T_0, T_1, ... be a recursively enumerable sequence of teachers and L_0, L_1, ... be a recursively enumerable sequence of learners. Assume that for each $i \in \mathbb{N}$,

- *$L_i(BB)$ learns C from $T_i(TBB)$ for all $BB \in BBC$ and $TBB \in TBBC_i$, and*
- *L_i and T_i are teacher-box-proof for C and BBC.*

Then there exists a teacher T and a learner L such that $L(BB)$ learns C from $T(TBB)$ for all $BB \in BBC$ and $TBB \in \bigcup_{i \in \mathbb{N}} TBBC_i$ and the agents T and L are teacher-box-proof for C and BBC.

Theorem 7′ *Let BBC and $TBBC$ be two fixed classes of Black Boxes and C' be a fixed class of partial recursive functions. Let C_0, C_1, ... be a sequence of classes of partial recursive functions such that $\bigcup_{i \in \mathbb{N}} C_i \subseteq C'$. Let T_0, T_1, ... be a recursively enumerable sequence of teachers and L_0, L_1, ... be a recursively enumerable sequence of learners. Assume that for each $i \in \mathbb{N}$,*

- *$L_i(BB)$ learns C_i from $T_i(TBB)$ for all $BB \in BBC$ and $TBB \in TBBC$, and*
- *L_i and T_i are target-proof on C' for BBC and $TBBC$.*

Then there exists a teacher T and a learner L such that $L(BB)$ learns $\bigcup_{i \in \mathbb{N}} C_i$ from $T(TBB)$ for all $BB \in BBC$ and $TBB \in TBBC$ and the agents T and L are target-proof on C' for BBC and $TBBC$.

Proof of Theorems 5′, 6′ and 7′: The construction given in the proof of Theorem 5 has to be changed to allow for the simulation of a recursively enumerable set

of teachers and learners. Both T and L have to be able to simulate an infinite (but countable) number of subagents; thus infinite arrays (or some other data structures) have to be used for *suspended*, *step* and *previous_hypothesis*. It is also convenient to use another infinite array *initialized* which allows the main agent to determine whether a particular subagent has or has not been initialized yet. A subagent is initialized just before it is simulated for the first time. Other than this, the only fundamental change is in the mechanism of choosing the next sub-agent to simulate and the sub-agent whose last hypothesis should be output by L.

In the simple case of two sub-agents they could both be simulated in parallel by using a variable *which*, alternating between the values 0 and 1. Now a more complicated scheme involving dovetailing is needed, for example, the **For** statement in UnionTeacher and UnionLearner could be replaced by the following one:

For $(n = 0,\ \mathit{which} = \pi_1(n);\ ;\ n{+}{+},\ \mathit{which} = \pi_1(n))$.

This way L simulates each learner in the countable set for infinitely many steps and T simulates each teacher.

Choosing the next hypothesis to output is a little more complicated than choosing the next sub-agent to simulate. As before, if the sub-agent L_{which} outputs a hypothesis (as seen from it writing the special marker on its (simulated) Output Tape and moving to the right), a special subroutine HypoDecidePrime is called with parameters *which* and h, where h is the new hypothesis. This subroutine is given in Figure 7.4 and is similar in its logic to the subroutine HypoDecide used in the simpler case of two sub-agents.

As before, the last hypotheses of all those sub-agents that have output any are

```
HypoDecidePrime(which, h)
{
    static previous_hypothesis[∞] = { undefined, undefined, ... };
    static last_by_whom = undefined;
    static n = −1;

    If (previous_hypothesis[which] == h)
        Return;

    previous_hypothesis[which] = h;

    If ((last_by_whom ≠ undefined) and (last_by_whom ≠ which))
        Return;

    For (n++; ; n++)
    {
        next = π1(n);

        If (previous_hypothesis[next] ≠ undefined)
            Break;                          /* Out of this loop */

    }

    last_by_whom = next;
    Output h;
    Output previous_hypothesis[last_by_whom];
}
```

Figure 7.4 Subroutine HypoDecidePrime

kept in the array *previous_hypothesis* and the index of the sub-agent whose hypothesis was last output by L is stored in the variable *last_by_whom*. There is another `static` variable n which insures that every sub-agent is treated "fairly" by L when it is looking for the next hypothesis to output. The first **If** statement checks whether there has been a hypothesis change done by L_{which} or whether it just output its previous hypothesis again. In the latter case the subroutine just returns. In the former case it is known that L_{which} has changed the hypothesis (or output its first one) and its current hypothesis is stored in *previous_hypothesis*[*which*]. After this the subroutine checks whether there is a reason for L to output a hypothesis. In the case

when L has already output some hypothesis previously (as seen from $last_by_whom \neq$ undefined) and the last hypothesis output by L was not produced by L_{which} (as seen from $last_by_whom \neq which$), there is no need to output a new hypothesis and the subroutine returns.

At the **For** loop of the subroutine it is known that either h is the very first hypothesis produced by any sub-agent of L or that the last hypothesis output by L was produced by L_{which} and that L_{which} has now made a hypothesis change by outputting h. The loop finds the index *next* of some sub-agent that has also output at least one hypothesis (the last hypothesis of this sub-agent will be output by L in a moment). In the case when h is the first hypothesis output by any sub-agent, $next = which$. In this case L outputs two hypotheses h. Even when h is not the first hypothesis output by any sub-agent of L, it can still happen that $next = which = last_by_whom$ and that L outputs two hypotheses h. Note that outputting the first h causes a hypothesis change of L (although it does not cause a change in *last_by_whom*). This situation arises when the values that n goes through in the **For** loop are such that $\pi_1(n)$ is not equal to an index of any sub-agent that has already output a hypothesis, before becoming equal to *which* (again). It is also possible, of course, that $next \neq which$ after the loop and that L performs one hypothesis change by outputting h and then possibly another one by outputting $previous_hypothesis[last_by_whom] = previous_hypothesis[next]$. The index of the sub-agent whose last hypothesis was output by L is saved in *last_by_whom* in all cases.

Intuitively speaking, HypoDecidePrime provides another dovetailing which is independent from the one introduced in the revised version of UnionLearner, and which insures that the following two possible cases have a satisfactory conclusion:

1. If no simulated learners converge but at least one of them them outputs some hypothesis then L should change its mind infinitely often. In this case all simulated learners that have output hypotheses have to change their mind infinitely often. L visits all these sub-agents infinitely often for the purpose of outputting their last hypotheses as its own. This is so because π_1 returns the first components of all pairs of natural numbers according to some recursive enumeration of them. Due to the inevitable hypothesis change when outputting h (the first one if two in a row), the learner L does not converge but changes its mind infinitely often.

2. If at least one of the simulated learners converges then L should converge to the same hypothesis as this sub-agent. Due to the properties of π_1, one of the converging learners will be reached by the **For** loop (i.e., *next* will hold its index) and reached after it has output a hypothesis that will not change. Thus, L converges to the same hypothesis.

In the trivial case when none of the simulated sub-agents outputs any hypotheses, the subroutine HypoDecidePrime does not get called and L does not output any hypothesis. We know, of course, that none of the simulated learners can converge incorrectly (because of learner-box-proofness, teacher-box-proofness or target-proofness, depending on which theorem is being proved). From the discussion above it follows that L cannot converge incorrectly either. This makes L and T learner-box-proof, teacher-box-proof or target-proof on C', depending on which theorem is being proved (for the appropriate classes). It can also be seen that if at least one of the simulated learners converges then it converges correctly and so does L, although it could converge to some other sub-agent's (correct) hypothesis. This concludes the proof of all three generalized union theorems. ■

Chapter 8

Conclusion

In this chapter we review our new learning (and teaching) model and sketch a few of the many questions and directions that could be explored. We address questions such as: "Why step back from all the achievements provided by Rogers' Isomorphism Theorem?", "Why hide the Black Box from the learner (or teacher)?", "How strong is the model?", "Isn't it too strong?", "How feasible is coding?", "Can the model be made more realistic?", "Can it be extended/generalized?", "What else can we do next?", and others.

We start with some justification of the model. It is true that Rogers' Isomorphism Theorem and also the recursive relatedness theorem for complexity measures have had a great impact on the development of all of theoretical computer science. Basically, these theorems imply that all the normal programming systems are closely related and, therefore, what can be done in one, can be done in all others with no significant sacrifice in performance. Thus, when we need to prove something for Turing machines and it gets hard, we have the option of switching to RAMs, for example. Or, what happens more often, perhaps, in order to prove that some conventional

programming language cannot achieve some goal, we prove that a Turing machine cannot do it either.

Despite this, we feel that in learning theory there is a reason to look at different programming systems and not to think of them as related. It is desirable that learning theory reflects learning in real life where the capabilities of different learners do not seem to be the same. Or, if we feel that the goal of learning theory is to get us closer to building intelligent machines, where does Rogers' Isomorphism Theorem take us? To coding. That is how we program our computers and that is the way they respond to our wishes best. But is it learning? Well, most researchers would say "no". As mentioned above, nearly every model in Learning Theory includes some features that insure that learning does not become too trivial. Usually this concerns the possibility of coding.

In our model we have tried to make coding useless as opposed to taking steps to prevent it. That is, we have taken one big step which seems to destroy the common ground that permits coding, even if it is a step back from the isomorphism of all acceptable programming systems. Once the teacher and the learner have different programming systems, with no known translation, we can allow them to exchange information which would be regarded as highly confidential if the translation were known. We believe that this "freedom of speech" in our model is something that makes it worth exploring farther.

Another decision about the model, which may seem strange, is the separation of agents and their Black Boxes. This is done to reflect another phenomenon which we believe exists in reality—not knowing one's own capabilities before they have been tried out. Humans tend to feel very uncertain regarding their strengths with respect to new things—certainty or at least optimism usually comes only after something

similar has been successfully tried out. This applies to each new task and to physical and mental strengths. We are not quite sure about how “smart” we are (especially, while still in graduate school). We have much higher confidence with more familiar issues than with strange, unfamiliar ones.

We now discuss how strong the model is. As the reader must have noticed, Theorems 1 and 3 both refer to a situation where the class of all the partial recursive functions is learned. This is a very strong result, since in regular inductive inference even the class of total recursive functions cannot be learned [15]. This may suggest that the model is on the strong side. This also urges us to look for other relationships between the teacher's and the learner's Black Boxes, that would perhaps allow smaller classes of functions to be learned. However, it is also obvious that the “strength” of the model is due to an excellent job on the teacher's side. As soon as the teacher leaves, the learner (i.e., the independent learner) becomes quite helpless, being unable to learn the constant all-zero function, as proved by Theorem 4. Both this inability of the learner alone and the strength of it when the teacher knows the bound relating the complexities of their Black Boxes are somewhat extreme.

There is, however, a definite positive side to the learner's weakness in the absence of the teacher. Namely, it discourages the idea that there could be some coding going on between the teacher and the learner. When the learner tries to learn the all-zero function without a teacher, there is not much that it could be missing. It knows what the all-zero function looks like and it knows a number of indices for the all-zero function in any natural programming system that the teacher could have had in its Black Box. That is, it could simulate a number of teachers itself, which of course would not help. According to Theorem 1, what it really needs is the knowledge of a bound $b(\cdot, \cdot)$ such that its Black Box is b-related to some other reasonable Black Box

that it can simulate. Without the bound, even the knowledge of how to build its own Black Box (which is attainable in the limit for primitive recursive Black Boxes) does not help. The bound b, however, does not give the impression of being some encoding for the target function. It is the same total recursive function for whatever partial recursive function the target could be. Besides, the Standard Teacher with bound b and the Standard Learner are target-learner-box-and-teacher-box-proof and in addition the Standard Teacher is responsive and the Standard Learner is non-gullible. This is additional evidence that coding is unlikely here.

The above discussion exposes a question about the model. We could combine the separate agents of the teacher and the learner into one agent with a somewhat different task. This agent would be a Turing machine with two oracles, each giving values of one Black Box. It would be given an index for the target function in one of the Black Boxes and required to find a correct index in the other. Of course, if the agent possesses some valuable information about the relationship of the programming systems in the Black Boxes, like a translation function or the bound for b-relatedness, it could then find an index in the other Black Box. But this does not look like such a natural model any more. We feel that it is much nicer to separate the Black Boxes by making each belong to a different agent, thus providing some symmetry to the model. As was mentioned above, this could provide for easy extensions of the model in the future, for example, the learner could become a teacher. Furthermore, the intuition that one can teach "anything" to somebody who is not too much "slower" (or "dumber") is quite satisfying. The "opposite" result given by Theorem 3 then also has an interesting interpretation: if somebody can learn everything from you, then he/she cannot be much "slower" than you are. Thus, we feel that the model has just the right balance of tools given to each agent.

Now we discuss a wish-list for our new model. Most of our results concern learning the class of all partial recursive functions in the limit; smaller target classes and more restricted versions of relationships between Black Boxes are natural directions. Perhaps there can be more natural relationships than b-relatedness. For example, we could consider polynomial-time related Black Boxes. We could also consider nicer classes of Black Boxes, e.g., polynomial-time computable as opposed to primitive recursive.

Questions about the relationships between target-proofness, learner-box-proofness, teacher-box-proofness, and combinations of them (for example, learner-box-and-teacher-box-proofness) are another possible direction of future research. For example, there might be some class of Black Boxes for the teacher and some class of Black Boxes for the learner such that some class of target functions can be learned by two agents that are both learner-box-proof and teacher-box-proof but are not learner-box-and-teacher-box-proof. Union theorems open the door to many other questions regarding combinations of proofness properties. For example, can we do unions on both learners' and teachers' Black Boxes? A very interesting and probably difficult question is how to translate some of these insights into the domain of polynomial-time learning, instead of learning in the limit.

Rather than taking the model in a more applied direction, we could move in the opposite direction, and allow an action-space to contain uncomputable functions. For example, we might define a *Pitch-Black Box* to be any total three-argument function, and provide our agents with those. The contents of the Input Tape are then an enumeration of a function such that it has an extension contained in the teacher's Pitch-Black Box, and the other basic definitions remain essentially the same. In this setting it appears that the analog of Theorem 1 still holds, but, beyond that, things

rapidly become very murky. However, the gap between the (computable) cognitive space and the (potentially uncomputable) action space might provide interesting analogies to the opacity of some of our own "black box" capabilities.

Appendix to Part 1

Proof of Lemma 2

This section of the appendix contains the proof of Lemma 2 from Chapter 2.

Lemma 2 *There is a universal Black Box UBB which is not acceptable.*

One might think that an argument similar to the one presented in the proof of Claim 9 in Chapter 6 would allow to conclude that every universal Black Box is also acceptable. However, this is not so, since without knowing whether there is a translation from some well-known programming system to this Black Box, we cannot prove the existence of the necessary composition function. Now we present a construction of a universal Black Box which is not acceptable.

Proof: A simple algorithm that gives the values of the Black Box *UBB* is given in Figure 8.1. It uses the subroutine TMBB which returns the values of the Turing Machine Black Box *TMBB* and was first used in Theorem 4 of Chapter 6. That is, given arguments i, x and s, it simulates the Turing machine with index i on input x for at most s steps and returns ? if the machine does not stop or does not produce output before stopping, or a value y if the machine stops with output y.

```
UBB(j, x, s)
{
    i = ⌊log(j + 2)⌋ − 1;          /* 2^{i+1} − 2 ⩽ j < 2^{i+2} − 2 */
    base = 2^{i+1} − 2;
    offset = 0;

    For (k = 0; k ⩽ i; k++)
        If (TMBB(k, k, s) ≠ ?)
            offset = offset + 2^k;

    If (j == base + offset)
        Return TMBB(i, x, s);
    Else
        Return ?;

}
```

Figure 8.1 Algorithm UBB

We can describe the functions contained in the Black Box *UBB* as follows. First, separate the functions that it contains into clusters: the first two functions form the first cluster, the next four functions form the second cluster, the next eight functions form the third cluster, and so on. In general, if we number the clusters beginning with 0, then cluster i contains functions with indices from $2^{i+1} - 2$ to $2^{i+2} - 3$, inclusive. Furthermore, cluster i contains the partial recursive function given by the Turing machine with index i somewhere. The exact offset of the function in the cluster depends on which Turing machines with indices from 0 to i stop on their own indices and which do not. Now we elaborate on this property and explain the algorithm in detail.

The algorithm starts by finding i such that $2^{i+1} - 2 \leqslant j < 2^{i+2} - 2$. In other words, it finds the cluster to which function UBB_j belongs. The beginning of this cluster is held in variable *base*. Next it enters a **For** loop in which it attempts to calculate the offset within the cluster, at which the function TM_i should be. It is

obvious that for different s this calculated offset will be different but as $s \to \infty$, *offset* will converge to the number $\sum_{k=0}^{i} c_k 2^k$, where $c_k = 1$ if TM_k stops on input k and $c_k = 0$ otherwise. In other words, the bit values of the final value of *offset* tell which Turing machines with indices from 0 to i stop on their own indices as inputs. Of course, for small enough s the value of *offset* may not have converged yet and thus algorithm UBB may define some finite number of values for any function in cluster i. However, at most one of these functions (namely, the one that has offset equal to the limit value of variable *offset*, and which is given by TM_i) can be defined on infinitely many inputs. (Note that this function TM_i will in general have complexity in *UBB* much higher than that of $\overline{TM_i}$.)

Finally, the algorithm checks whether j matches its current estimate of where function TM_i should go in the Black Box *UBB*. If it does, it outputs whatever the Turing machine with index i would output on input x in s steps. Otherwise, it outputs ?.

From this description of the algorithm UBB it should be clear that every partial recursive function is contained in *UBB*. Namely, function TM_i can be found in cluster i with offset such that its bit values encode which Turing machines with indices from 0 to i stop on their own indices as inputs. It is also obvious that the algorithm computes a total recursive three-argument function. Therefore, *UBB* is a full Black Box. From Lemma 1 it follows that *UBB* is a universal Black Box. Now we need to prove that it is not an acceptable Black Box.

Suppose that it is. Then there exists a total recursive composition function for it. Also, there is a translation from the Turing Machine Black Box *TMBB* into our new Black Box *UBB*. There certainly is also a translation from *UBB* into *TMBB*, as there is between every universal and acceptable Black Box. Furthermore,

by Proposition 3.4.5 (Padding Lemma) from Machtey and Young [30], using these translations we can find a total recursive one-to-one *padding* function $p(x, y)$, such that $UBB_x \equiv UBB_{p(x,y)}$, for all natural x and y. It is important that the padding function is one-to-one, since it implies that for every x and every z, there exists a y such that $p(x, y) \geqslant z$.

Now we show that our assumption that *UBB* is acceptable can be used to solve the diagonalized halting problem. Namely, we can now determine whether an arbitrary Turing machine with index i stops on input i. For this we do the following. First take an index i_0 for the Turing machine that computes the all-zero function Z. Now, find its image j_0 in *UBB* under the translation from *TMBB* to *UBB*. That is, j_0 is such that $UBB_{j_0}(x) = 0$, for all $x \in \mathbb{N}$. If $j_0 < 2^{i+1} - 2$, we need to find a bigger index for the all-zero function. We find the smallest y such that $p(j_0, y) \geqslant 2^{i+1} - 2$, where $p(\cdot, \cdot)$ is the padding function for *UBB*. Now we have some index m such that $m \geqslant 2^{i+1} - 2$ and UBB_m computes the all-zero function Z. Since Z is defined on all inputs and in every cluster at most one function can be defined on infinitely many inputs, we must have found the only all-zero function in that cluster. What remains is to obtain the offset of UBB_m within its cluster and decode the information stored in the bit values.

The base of the cluster can be found by computing $i' = \lfloor \log(m + 2) \rfloor - 1$. The offset is then given by $m - i'$. We are interested in seeing whether bit i is set in $m - i'$, where bits are numbered from right to left, beginning with 0. This can be readily checked by taking the offset modulo 2^{i+1} and then comparing it to 2^i. That is, bit i is set and the Turing machine with index i stops on input i if and only if $(m - i') \bmod 2^{i+1} \geqslant 2^i$.

We have now given an algorithmic method to solve the diagonalized halting prob-

lem using the assumption that the Black Box *UBB* is acceptable. It is well known that this problem cannot be solved and therefore the assumption must be wrong. Thus, the Black Box *UBB*, as given by the algorithm UBB, is universal but not acceptable. ■

Proof of Weakened Corollary 10

In Chapter 6 we proved Theorem 4 which implied Corollary 10. This corollary says that there is no universal Black Box *TBB* and no total recursive two-argument function $b(\cdot,\cdot)$ such that all the primitive recursive acceptable Black Boxes *BB* are b-related to *TBB*. In other words, for every universal Black Box *TBB* and for every total recursive two-argument function b there is some "bad" primitive recursive acceptable Black Box *BBB*, such that *BBB* is not b-related to *TBB*.

The proof of this corollary, as given in Chapter 6, was not constructive, i.e., it was only proved that no bound can b-relate all primitive recursive acceptable Black Boxes to any universal Black Box. It is possible, however, to give a construction of a bad primitive recursive Black Box *BBB* for every given universal Black Box *TBB* and every total recursive two-argument function $b(\cdot,\cdot)$. Presenting this proof in detail is the goal of the next section, here we give a simpler, more elegant proof which is not completely constructive in its nature and is good only for all acceptable (and not all universal) Black Boxes *TBB*. Nevertheless, the construction of the bad primitive recursive Black Box in this proof is very similar to that in the stronger proof and therefore reading this section prior to the next one is highly recommended.

We now present the exact statement of the weakened Corollary 10 which we are

going to prove.

Corollary 10′ *For every acceptable Black Box TBB and every total recursive two-argument function $b(x, s)$ there exists a primitive recursive acceptable Black Box BBB which is not b-related to TBB.*

Proof: First we fix a very special Black Box for the teacher, namely, the Turing Machine Black Box *TMBB*. We prove a lemma which says that in this special case, for every total recursive two-argument function b there is a bad acceptable primitive recursive Black Box for the learner. Then we assume that for some other acceptable teacher's Black Box *TBB* the corollary is not true, and use this assumption to prove that it is not true for the Turing Machine Black Box *TMBB* as well, which contradicts the lemma. In this step of the proof we rely on the b-relatedness of all acceptable Black Boxes as well as the transitivity of b-relatedness.

Lemma 8 *For every total recursive two-argument function $b(x, s)$ there exists an acceptable primitive recursive Black Box BBB which is not b-related to the Turing Machine Black Box TMBB.*

Proof: First let us analyze what it means for a Black Box *BBB* to be not b-related to the Turing Machine Black Box *TMBB*. Then there must exist an i_0 such that for all j either:

1. $BBB_j \not\geq TM_{i_0}$, or
2. $\overset{\infty}{\exists} x \in \mathrm{Dom}(TM_{i_0})$ such that $\overline{BBB}_j(x) > b(x, \overline{TM}_{i_0}(x))$.

In other words, there exists an i_0 such that for all j:

$$\Big(BBB_j \geqq TM_{i_0}\Big) \Rightarrow \Big(\overset{\infty}{\exists} x \in \mathrm{Dom}(TM_{i_0}) \text{ such that } (\overline{BBB}_j(x) > b(x, \overline{TM}_{i_0}(x)))\Big).$$

We choose the index i_0 first. Let i_0 be an index (in *TMBB*) for the constant all-zero function Z. That is, $TM_{i_0} \equiv Z$, meaning that $TM_{i_0}(x) = 0$, for all $x \in \mathbb{N}$. Furthermore, let us specify that i_0 is not just any index for Z but the one corresponding to the Turing machine which simply ignores any input, outputs a 0 and then stops. Thus, we can also assume that this Turing machine takes exactly s_0 steps for any input. That is, $\overline{TM}_{i_0}(x) = s_0$, for all $x \in \mathbb{N}$. Intuitively, we would like to define $BBB(i, x, s)$ to be the same as $TMBB(i, x, s)$ except that on those values of x where $TM_{i_0}(x) = 0$ we would like to insure that $BBB(i, x, s) = ?$ for all $s \leqslant b(x, s_0)$ and $BBB(i, x, s_0 + 1) = 0$, so that $\overline{BBB}_i(x) > b(x, \overline{TM}_{i_0}(x))$. However, since the function b may not be primitive recursive, we use the step bound s in the computation of $b(x, s_0)$.

In Figure 8.2 we present algorithm BBB which gives the values of the bad Black Box *BBB*. We assume that the algorithm knows i_0, s_0, and an index β, such that the Turing machine with index β computes the values of $b(x, s)$. That is, $TM_\beta(\langle x, s\rangle) = b(x, s)$, which is a total recursive function.

The algorithm uses the subroutine TMBB, known from the proof of Theorem 4, which computes the values of the Turing machine Black Box *TMBB*. That is, given arguments i, x and s, it simulates the Turing machine with index i on input x for at most s steps and returns ? if the machine does not stop or does not produce output before stopping, or a value y if the machine stops with output y.

BBB begins by finding the value of $TMBB(i, x, s)$, which it puts in variable v.

```
BBB(i, x, s)
{
    v = TMBB(i, x, s);

    If (v ≠ 0)                          /* ‾TM‾_i(x) > s or TM_i(x) ≠ 0 */
        Return v;

    v = TMBB(β, ⟨x, s_0⟩, s);

    If (v == ? or v ⩾ s)          /* ‾TM‾_β(⟨x, s_0⟩) > s or b(x, s_0) ⩾ s */
        Return ?;

    Return 0;
}
```

Figure 8.2 Algorithm for constructing the bad Black Box

A value $v = ?$ means that the Turing machine with index i does not stop on input x in s or less steps. A value $v > 0$ means that it does stop but that its value is not 0. In both cases, checked by the test **If** $(v \neq 0)$, the algorithm returns the value v, which is the same value as *TMBB* would give. Only if $v = 0$, meaning that the Turing machine with index i does stop on input x in s or less steps and gives output 0, the algorithm performs additional computation, which we now describe.

The algorithm knows the complexity of TM_{i_0} on every input, which is s_0. It now tries to compute the bound $b(x, s_0)$. This is done by calling the subroutine TMBB with parameters β, $\langle x, s_0 \rangle$, and s. The value is once again put in the variable v. Since b is a total recursive function and $TM_\beta(\langle x, s_0 \rangle)$ computes $b(x, s_0)$, there will be non-? return values from this call of TMBB for sufficiently big values of s. For smaller values of s, however, v will be ?. If this is the case, the algorithm returns a ?. If $v \neq ?$, then it must be that the Turing machine with index β stops on input $\langle x, s_0 \rangle$, producing output v. Therefore, $b(x, s_0) = v$. The algorithm compares v to s and if $v \geqslant s$, it returns ? again. It returns 0 only if $s > v = b(x, s_0)$.

Since the algorithm has no loops and only bounded simulations of Turing machines (performed inside the subroutine TMBB), it is clear that it computes a primitive recursive three-argument function. Therefore, we can say that it defines a primitive recursive Black Box. It is also obvious that this Black Box BBB contains exactly the same functions as $TMBB$, and in the same order. This means that it is an acceptable primitive recursive Black Box.

The algorithm can only return 0 if $s > b(x, s_0)$, and it does output 0 for all x, if i is an index for the all-zero function and s is sufficiently big. This means that for all x and for all j such that $BBB_j \equiv Z$, we have that $\overline{BBB}_j(x) > b(x, s_0)$, which completes the proof of the lemma. ■

We have constructed a primitive recursive acceptable Black Box BBB which is not b-related to the Turing Machine Black Box $TMBB$. Now we prove Corollary 10′ for all acceptable Black Boxes. We assume that the corollary does not hold. Then there is some acceptable Black Box TBB and there is a total recursive two-argument function $b(x, s)$ such that all primitive recursive acceptable Black Boxes are b-related to TBB.

Since by Lemma 4 every two acceptable Black Boxes are b'-related for some total recursive two-argument function b', we have that TBB is b'-related to $TMBB$. Now, using Lemma 6, the transitivity of b-relatedness, we have that all the primitive recursive acceptable Black Boxes are b''-related to $TMBB$, for some total recursive two-argument function b''. This contradicts Lemma 8 and our assumption must be incorrect. ■

This proof is simpler than the one given in the next section. However, it is not as strong in many respects. Most importantly, it does not give a construction of the bad

Black Box for a given acceptable Black Box *TBB* and total recursive function $b(\cdot,\cdot)$. It only constructs one for the special Black Box *TMBB* and proves that there exists one for *TBB* as well. Secondly, the bad Black Box whose existence is proved (not the Black Box *BBB* that is actually constructed) has high complexity for the constant all-zero function on all but finitely many inputs x (since we used the fact that *TBB* is b'-related to *TMBB*), as opposed to absolutely all inputs which is achieved in the directly constructive proof of Corollary 10. And, finally, this proof only works for acceptable Black Boxes instead of all universal Black Boxes.

Constructive Proof of Corollary 10

In this section we give a direct, constructive proof of Corollary 10.

Corollary 10 *For every universal Black Box TBB and for every total recursive two-argument function $b(x,s)$ there exists a primitive recursive acceptable Black Box BBB which is not b-related to TBB.*

Proof (direct and constructive): As in the simpler proof above, we prove that there exists a Black Box *BBB* and an index i_0 such that for all j:

$$\Big(BBB_j \geqq TBB_{i_0}\Big) \Rightarrow \Big(\overset{\infty}{\exists} x \in \mathrm{Dom}(TBB_{i_0}) \text{ such that } \overline{BBB}_j(x) > b(x, \overline{TBB}_{i_0}(x))\Big).$$

We choose the index i_0 first. Let i_0 be an index (in the teacher's Black Box *TBB*) for the constant all-zero function Z. That is, $TBB_{i_0} \equiv Z$, meaning that $TBB_{i_0}(x) = 0$, for all $x \in \mathbb{N}$. The choice of this function is not very important—any primitive recursive function would do.

```
BBB(i, x, s)
{
    v = TMBB(i, x, s);

    If (v ≠ 0)                                    /* ‾TM_i(x) > s or TM_i(x) ≠ 0 */
        Return v;

    For (z = 0; z ⩽ s; z++)
    {
        v = BBSim(tbb, ⟨⟨i_0, x⟩, z⟩, s);            /* Try to compute TBB(i_0, x, z) */

        If (v == ??)                               /* ‾TM_tbb(⟨⟨i_0, x⟩, z⟩) > s */
            Return ?;

        If (v ≠ ?)                                 /* z == ‾TBB_{i_0}(x) */
            Break;                                 /* Out of this loop */

    }
    If (v == ?)                                    /* ‾TBB_{i_0}(x) > s */
        Return ?;

    v = TMBB(β, ⟨x, z⟩, s);

    If (v == ? or v ⩾ s)     /* ‾TM_β(⟨x, ‾TBB_{i_0}(x)⟩) > s or b(x, ‾TBB_{i_0}(x)) ⩾ s */
        Return ?;

    Return 0;
}
```

Figure 8.3 Algorithm that gives the values of the bad Black Box

Now we build the bad Black Box BBB for the learner such that for all j and all x, if $BBB_j(x) = 0$ then $\overline{BBB}_j(x) > b(x, \overline{TBB}_{i_0}(x))$. This Black Box is "very bad" in the sense that not only the all-zero function has high complexity for every occurrence of it in the Black Box, but, in fact, every function the value of which is 0 on some input x has high complexity on that input. Hence, every occurrence of the all-zero function Z in the Black Box BBB has high complexity on all inputs, not just on infinitely many.

The algorithm BBB for building the bad Black Box BBB (with respect to the

Black Box *TBB* and a total recursive two-argument function b) is given in Figure 8.3. We assume that it knows an index *tbb* such that the Turing machine with index *tbb* computes the values of $TBB(i, x, s)$. That is, $TM_{tbb}(\langle\langle i, x\rangle, s\rangle) = TBB(i, x, s)$. We also assume that the algorithm knows an index i_0, such that $TBB_{i_0} \equiv Z$ and that it knows an index β such that the Turing machine with index β computes the values of $b(x, s)$. That is, $TM_\beta(\langle x, s\rangle) = b(x, s)$, which is total recursive.

The algorithm uses the subroutine TMBB, from the proof of Theorem 4, which computes the values of the Turing Machine Black Box *TMBB*. That is, given arguments i, x and s, it simulates the Turing machine with index i on input x for at most s steps and returns ? if the machine does not stop or does not produce output before stopping, or a value y if the machine stops with output y. This subroutine is good for simulating Turing machines that do not output ? themselves. For those that do, for example, the ones that implement Black Boxes, our algorithm uses a more sophisticated subroutine BBSim. It takes three arguments i, x and s and works almost like TMBB, except that it returns the special value ?? (instead of ?) if the simulated Turing machine i does not stop or does not produce output before stopping, or a value y (where y can be ? as well) if the machine stops with output y.

BBB begins by finding the value of $TMBB(i, x, s)$, which it puts in variable v. A value $v =$? means that the Turing machine with index i does not stop on input x in s or less steps. A value $v > 0$ means that it does stop but that its value is not 0. In both cases, checked by the test **If** $(v \neq 0)$, the algorithm returns the value v, which is the same value as *TMBB* would give. Only if $v = 0$, meaning that the Turing machine with index i does stop on input x in s or less steps and gives output 0, the algorithm performs additional computation, which we now describe.

The algorithm then enters a **For** loop, which has a parameter z, initialized to

0 and incremented after each iteration. In the last iteration of the loop $z = s$, unless either the **Return** or the **Break** statement inside it gets executed sooner. In each iteration the algorithm gets the value of $\text{BBS}\textsc{im}(tbb, \langle\langle i_0, x\rangle, z\rangle, s)$, which it puts in variable v. Recall that tbb is the index for a Turing machine that gives the values of TBB. However, for small values of s this Turing machine may not stop, in which case v is assigned a ??. This is tested in the next **If** statement and the algorithm returns a ? if s is indeed not big enough for this Turing machine to stop. The second **If** statement inside the loop checks whether $v = ?$. If so then the loop is reiterated, since it means that the Turing machine with index tbb does stop on input $\langle\langle i_0, x\rangle, z\rangle$ in s steps and that its output is ?, which, of course, means that the value $TBB(i_0, x, z)$ is ?. The algorithm is interested in finding the least value of z such that $TBB(i_0, x, z) \neq ?$, which is the complexity of TBB_{i_0} on input x. Thus, if it ever happens that $v \neq ?$ in this second test of the loop, the algorithm has the complexity $\overline{TBB}_{i_0}(x)$ in variable z. Then it leaves the loop and, of course, gets past the immediately following test **If** ($v == ?$). If, however, the loop was terminated because z reached $s + 1$, then it must be that $v = ?$ and the immediately following test succeeds, causing the algorithm to return a ?. This occurs if the complexity of TBB_{i_0} on input x is greater than s and therefore could not be captured in variable z during this bounded loop.

The following claim proves that for sufficiently big s, the loop will terminate via the **Break** statement, meaning that $\overline{TBB}_{i_0}(x)$ will be found, for all x. If this were not the case, then the algorithm could never return 0 and the Black Box defined by it could not contain any functions with 0 values.

Claim 10 *For every x there is an s such that the* **For** *loop terminates with $z \leqslant s$ and $v \neq ?$, i.e., via the* **Break** *statement.*

Proof: Suppose there is an x for which it does not happen. Let us denote by $steps_z \stackrel{\text{def}}{=} \overline{TM}_{tbb}(\langle\langle i_0, x\rangle, z\rangle)$ the number of steps it takes for the Turing machine with index tbb to stop on input $\langle\langle i_0, x\rangle, z\rangle$. That is, BBSIM$(tbb, \langle\langle i_0, x\rangle, z\rangle, steps_z)$ cannot output ??. Let us define M as $M \stackrel{\text{def}}{=} \max\{\, steps_z : z \leqslant \overline{TBB}_{i_0}(x)\,\}$. Let us now look at what happens when $s \geqslant \max(\overline{TBB}_{i_0}(x), M)$. It is clear that for all $z \leqslant \overline{TBB}_{i_0}(x)$, the Turing machine with index tbb stops on input $\langle\langle i_0, x\rangle, z\rangle$ in s or less steps. Thus, the value of v as returned by the subroutine BBSIM will not be ??, meaning that the loop will not terminate via the **Return** statement for these z. It is also clear, that $TBB(i_0, x, z) = ?$, for all $z < \overline{TBB}_{i_0}(x)$. Therefore the subroutine BBSIM will definitely output ? for these z, which will cause the loop to be reiterated, until z reaches $\overline{TBB}_{i_0}(x)$. At this point z will be big enough for $TBB(i_0, x, z)$ to have a non-? value, and s will be big enough (recall that $s \geqslant \max(\overline{TBB}_{i_0}(x), M))$ for BBSIM to discover this. Therefore, at this point the BBSIM will return $v \neq ?$ and the loop will exit via the **Break** statement. ∎

Thus the **For** loop succeeds in capturing $\overline{TBB}_{i_0}(x)$ in variable z, provided s is sufficiently large, by Claim 10. If it succeeds, then the algorithm also gets past the **If** test immediately following the loop. After that the algorithm tries to compute the bound $b(x, z)$. This is done by calling the subroutine TMBB with parameters β, $\langle x, z\rangle$ and s. The value is once again put in the variable v. Since b is a total recursive function and $TM_\beta(\langle x, z\rangle)$ computes $b(x, z)$, there will be non-? return values from this call of TMBB for sufficiently big values of s. For smaller values, however, the return value will be ?. If this is the case, the algorithm returns a ?. If $v \neq ?$, then it must be that the Turing machine with index β has stopped on input $\langle x, z\rangle$, with output v. Therefore, $b(x, z) = v$. The algorithm compares v to s and if $v \geqslant s$, it returns a ? again. It returns 0 only if $s > v = b(x, z) = b(x, \overline{TBB}_{i_0}(x))$.

Since the loop in the algorithm is bounded and it performs only bounded simulations of Turing machines, it follows that it computes a primitive recursive three-argument function. Thus it defines a primitive recursive Black Box. From the discussion above it can also be seen that this Black Box *BBB* contains exactly the functions of *TMBB* and in the same order. Therefore it is an acceptable Black Box. The measures contained in this Black Box are different from the Turing Machine Complexity Measure, though. We now explain how.

The first call to the subroutine TMBB with parameters i, x and s implies that $BBB(i,x,s) = ?$ whenever $TMBB(i,x,s) = ?$. In other words, $\overline{BBB}_i(x) \geqslant \overline{TM}_i(x)$, meaning that the complexity of any function BBB_i on any input x in the Black Box *BBB* is at least as high as the Turing machine complexity for the machine with index i on input x. When $TM_i(x) \neq 0$, we actually have that $\overline{BBB}_i(x) = \overline{TM}_i(x)$. When $TM_i(x) = 0$, the complexity of BBB_i on input x may be higher. In this case it is at least as high as:

1. The value of $\overline{TM}_i(x)$, as explained above;
2. The value of $\max(M, \overline{TBB}_{i_0}(x))$, as discussed in the proof of Claim 10;
3. The value of $\overline{TM}_\beta(\langle x, \overline{TBB}_{i_0}(x)\rangle)$, since the Turing machine with index β must be simulated long enough on input $\langle x, z\rangle$ for it to produce output;
4. The value of $b(x, \overline{TBB}_{i_0}(x))+1$, since only for $s > v = b(x,z) = b(x, \overline{TBB}_{i_0}(x))$ can the algorithm return 0.

The last bound implies that for all x and for all j such that $BBB_j \equiv Z$ we have that $\overline{BBB}_j(x) > b(x, \overline{TBB}_{i_0}(x))$. Thus, *BBB* is a primitive recursive acceptable Black Box and it is not b-related to *TBB*. This concludes the proof of Corollary 10. ■

Note that this proof of Corollary 10 guarantees that every occurrence of the all-zero function in BBB has high complexity on every input x, although this was required only on infinitely many x. (The preceding proof of Corollary $10'$ works for all but finitely many x.)

Part II

Learning with Malicious Errors in Queries

Chapter 9

Introduction

9.1 Query Models

In this and the following part of the dissertation we focus on models of learning different from the ones explored in Part 1. Partial recursive functions are an extremely powerful target rule to learn but the models where learning them is possible do not require the algorithms to stop. The identification happens in the limit and it is not possible to argue about the complexity of these algorithms. Therefore, we take a different approach to learning and require that learning algorithms be efficient. This generally means that their running time, used memory space and the number of examples that they require are polynomial in various input parameters that depend on the concrete problem. In order to make this possible we have to replace the partial recursive functions with simpler target rules, namely, *concepts*. Informally, a concept is a binary classification of all the examples that a learner may see.

The models that we consider in this and the next part are all derived from that of

a "minimally adequate teacher" [3]. In this model the learner is assisted by a teacher that answers two types of queries: equivalence and membership. An equivalence query corresponds to the learner verifying its current hypothesis about the target function; the teacher either acknowledges that they match or provides a counterexample. A membership query is a request by a learner to obtain the classification of a particular example it has chosen.

There is an impressive and growing number of polynomial-time algorithms, many of them quite beautiful and ingenious, to learn various interesting classes of concepts using equivalence and membership queries. To apply such algorithms in practice, however, researchers need to overcome a number of problems.

One significant issue is the problem of errors or omissions in answers to queries. (An omission is an indecisive answer given by a teacher when asked to classify a particular example.) Previous learning algorithms in the equivalence and membership query model are guaranteed to perform well assuming that queries are answered correctly, but there is often no guarantee that the performance of the algorithm will "degrade gracefully" if that assumption is not exactly satisfied.

A related issue is the assumption that the target concept is drawn from a particular class of concepts, for example, monotone DNF formulas. Even if the target concept is "nearly" a monotone DNF formula, there is typically no guarantee that the learning algorithm will do anything reasonable. We address these two issues, and demonstrate a useful relationship between them.

The focus of this part of the thesis is the model of equivalence and *malicious membership queries*. In a malicious membership query, the answer given may be correct or it may be an error. The answers are *persistent*; that is, repeated queries about

the same strings (examples) are given the same answer. Intuitively, the teacher never admits its errors. The choice of which examples are classified incorrectly is assumed to be made by a malicious adversary, hence the name of the model. Persistent errors are the hardest to overcome; non-persistent errors may provide additional information if contradicting answers are given, namely, that there has been an error and that one of the answers is wrong.

We assume that equivalence queries remain correct, that is, the counterexamples returned are always correct. Very interesting models can be constructed by allowing incorrect counterexamples to be returned, but they are all beyond the scope of this dissertation. Equivalence queries can be thought of as comparing the hypothesis to some actual phenomenon existing in nature. They may seem too powerful at first but are no stronger than the ability to draw random samples according to an unknown distribution—the basis of the PAC model [39]. Algorithms that learn from equivalence and membership queries can be translated to learn in the PAC model with membership queries.

We introduce a new parameter L to quantify the "amount" of errors in membership queries—it is a bound on the table-size of the set of strings on which the adversarial teacher gives the wrong answer. (The table-size of a set of strings is the number of strings in the set plus the sum of their lengths. Note that strings may have different lengths, including 0. Table-size is 0 if and only if the set is empty.) A polynomial-time learning algorithm is permitted time polynomial in the usual parameters and L. We give a polynomial-time algorithm to learn monotone DNF formulas using equivalence and malicious membership queries.

When considering target concepts that are "nearly" in some fixed class of concepts, we represent them as concepts from the class with "few" exceptions. That

is, we consider variants of the concepts in a given class and use table-size of the set of exceptions as a measure of "how different" the target concept is from one in the specified class. We define what it means for a concept class to be polynomially closed under finite exceptions. Some concept classes, for example, DFA's and decision trees, are polynomially closed under finite exceptions, while others, like monotone DNF formulas, are not. For the latter, we define a natural operation of adding exception tables to the concept class to make it polynomially closed under exceptions. We give a polynomial-time learning algorithm for the resulting class of monotone DNF formulas with finite exceptions, using equivalence queries and standard membership queries.

We then give a general transformation that shows that any class of concepts that is polynomially closed under exceptions and polynomial-time learnable using equivalence queries and standard membership queries is also polynomial-time learnable using equivalence queries and malicious membership queries. Corollaries include polynomial-time learning algorithms using equivalence queries and malicious membership queries for the concept classes of DFA's, decision trees, and monotone DNF formulas with finite exceptions.

The notion of a finite variant of a concept, that is, a concept with a finite set of exceptions, is a unifying theme in this part. Our model of errors in membership queries can be viewed as asking equivalence queries to a teacher that knows the target concept and asking membership queries to a teacher that knows a finite variant of the target concept. When learning concepts with finite exceptions we can view the learning problem as one in which the same teacher answers equivalence and membership queries but it knows a finite variant of the monotone DNF formula. In both cases, the goal of the learner is to exactly identify the concept according to

which equivalence queries are answered.

9.2 Previous Work

The effect of errors in examples in the PAC model has been studied extensively, starting with the first error-tolerant algorithm for the model, given by Valiant [39]. In this case the goal is PAC-identification of the target concept, despite the corruption of the examples by one or another kind of error, for example, random or malicious classification errors, random or malicious attribute errors, or malicious errors (in which both attributes and classification may be arbitrarily changed).

There has been not as much work on errors or omissions in learning models in which membership queries are available, and the issues are not as well understood. One relevant distinction is whether the errors or omissions in answers to membership queries are persistent or not. In general, the case of persistent errors or omissions is more difficult, since non-persistent errors or omissions can yield extra information, and can always be made persistent simply by caching and using the first answer for each domain point queried.

Sakakibara defines one model of non-persistent errors, in which each answer to a query may be wrong with some probability, and repeated queries constitute independent events [34]. He gives a general technique of repeating each query sufficiently often to establish the correct answer with high probability. This yields a uniform transformation of existing query algorithms. The method also works for both of Bultman's models [18]. This could be a reasonable model of a situation in which the answers to queries were being transmitted through a medium subject to random

independent errors; then the technique of repeating the query is eminently sensible.

A related model is considered by Dean et al. for the case of a robot learning a finite-state map of its environment using faulty sensors and reliable effectors [19]. This model assumes that observation errors are independent as long as there is a nonempty action sequence separating the observations. This means that there is no simple way to "repeat the same query", since a nonempty action sequence may take the robot to another state, and no reset operation is available. A polynomial-time learning algorithm is given for the situation in which the environment has a known distinguishing sequence. It achieves exact identification with high probability.

The method of "repeating the query" is insufficient for the more difficult case of persistent errors or omissions in membership queries. In this case, we must exploit the error-correcting properties of groups of "related" queries. In an explicit and very interesting application of the ideas of error-correcting algorithms, Ron and Rubinfeld use the criterion of PAC-identification with respect to the uniform distribution, and give a polynomial-time randomized algorithm using membership queries to learn DFA's with high rates of random persistent errors in the answers to the membership queries [33].

Algorithms that use membership queries to estimate probabilities (in the spirit of the statistical queries defined by Kearns [27]) are generally not too sensitive to small rates of random persistent errors in the answers to queries. For example, Goldman, Kearns, and Schapire give polynomial-time algorithms for exactly learning read-once majority formulas and read-once positive NAND formulas of depth $O(\log n)$ with high probability using membership queries with high rates of persistent random noise or modest rates of persistent malicious noise [23]. It is an open question whether Kushilevitz and Mansour's algorithm that uses membership queries

and exactly learns logarithmic-depth decision trees with high probability in polynomial time can sustain nontrivial rates of persistent random noise in the answers to queries [29].

Learning algorithms for other classes of concepts using equivalence and membership queries may depend more strongly on the correctness of the answers to individual queries; in these cases, there is no guarantee of a learning algorithm for the class that can tolerate errors or omissions in the answers to membership queries.

A model introduced by Angluin and Slonim addresses these issues: equivalence queries are assumed to be answered correctly, while membership queries are either answered correctly or with "I don't know" and the answers are persistent. The "I don't know" answers are determined by independent coin flips the first time each query is made [13]. They give a polynomial-time algorithm to learn monotone DNF formulas with high probability in this setting. They also show that a variant of this algorithm can deal with one-sided errors, assuming that no negative point is classified as positive. Goldman and Mathias also consider this model [24].

Sloan and Turán [11, 36] introduce a model of equivalence and *limited membership queries* which is midway between that of Angluin and Slonim and our current model. They consider malicious omissions in membership queries but assume that the equivalence queries remain correct. More precisely, they explore two such models: a *strict* and a *nonstrict* one. In the strict model the final hypothesis of the algorithm must be equivalent to the target concept while in the nonstrict model the hypothesis and the target may differ on examples that are answered with "I don't know". Furthermore, the equivalence queries in the nonstrict model may not use an example previously classified as "I don't know" for a counterexample. Both models are proven equivalent for the purpose of polynomial-time learnability of concepts if

equivalence queries are used. Sloan and Turán, however, also consider the simpler strict and nonstrict models where equivalence queries are not available, only the membership queries with malicious omissions. They give a polynomial-time learning algorithm using limited membership queries in the nonstrict model for the class of monotone monomials and also prove a lower bound on the query complexity for this problem. They also give a polynomial time algorithm in this model for the class of k-term monotone DNF formulas. If equivalence queries are available, they show that general monotone DNF formulas can be learned.

Blum, Chalasani, Goldman and Slonim introduce the models of *incomplete boundary queries* and *unreliable boundary queries* [14]. These are like limited and malicious membership queries, respectively, but subject to the restriction that omissions or errors may occur only in the "boundary region" of the target concept. Furthermore, equivalence queries may not return counterexamples from this region, or in the case of PAC-learning, the example distribution has zero probability in this region. The running time of algorithms may not depend on the number of omissions or errors received. They show that intersections of two halfspaces are PAC-learnable with unreliable boundary queries (and thus also with incomplete boundary queries). Considering learning from equivalence and "false-positive-only" unreliable boundary queries, they show that read-once monotone DNF formulas with terms of size at least four are learnable for boundary radius 1 (this implies learnability from equivalence and incomplete boundary queries). They also show that $(r + 1)$-separable k-term monotone DNF formulas are learnable from equivalence and "false-positive-only" unreliable boundary queries for boundary radius r.

Frazier, Goldman, Mishra and Pitt [20] introduce a model of omissions in answers to membership queries, called learning from a consistently ignorant teacher. The

basic idea is to require that if the teacher gives answers to certain queries that would imply a particular answer to another query, the teacher cannot answer the latter query with "I don't know." For example, in the domain of monotone DNF formulas, if the teacher classifies a particular point as positive, then the teacher cannot answer "I don't know" about any of the ancestors of the point. The goal of the learner is to learn exactly the ternary classification of points into positive, negative, and "I don't know" that is presented by the teacher. Such a ternary classification can be represented by the *agreement* of a set of binary-valued concepts; the agreement classifies a point as positive (respectively, negative) if all the concepts in the set classify it as positive (respectively, negative), otherwise, the agreement classifies the point as "I don't know." They give efficient learning algorithms in this model for monomials with at least one positive example, concepts represented as the agreement of a constant number of monotone DNF formulas, k-term DNF formulas, DFA's, or decision trees, and concepts represented by an agreement of boxes with samplable intersection. Compared to the model of Sloan and Turán, this model has a different measure of the representational complexity of a concept with omissions, which allows a much higher rate of omissions to be compactly represented. It also differs in requiring the learner to reproduce exactly the "I don't know" labels of the teacher, whereas in the (nonstrict) model of limited membership queries such examples can be classified arbitrarily.

Chapter 10

Preliminaries

10.1 Concepts and Concept Classes

Our definitions for concepts and concept classes are a bit non-standard. We have explicitly introduced the domains of concepts in order to try to unify the treatment of fixed-length and variable-length domains. We take Σ and Γ to be two finite alphabets. Examples are represented by finite strings over Σ and concepts are represented by finite strings over Γ.

A *concept* consists of a pair (X, f), where $X \subseteq \Sigma^*$, and f maps X to $\{0, 1\}$. The set X is the *domain* of the concept. The *positive examples* of (X, f) are those $w \in X$ such that $f(w) = 1$, and the *negative examples* of (X, f) are those $w \in X$ such that $f(w) = 0$. Note that strings not in the domain of the concept are neither positive nor negative examples of it.

A *concept class* is a triple (R, Dom, μ), where R is a subset of Γ^*, Dom is a map from R to subsets of Σ^*, and for each $r \in R$, $\mu(r)$ is a function from $Dom(r)$ to

$\{0,1\}$. R is the set of legal representations of concepts. For each $r \in R$, the *concept represented by* r is $(Dom(r), \mu(r))$.

A concept (X, f) is *represented by* a concept class (R, Dom, μ) if and only if for some $r \in R$, (X, f) is the concept represented by r. The *size* of a concept (X, f) represented by (R, Dom, μ) is defined to be the length of the shortest string $r \in R$ such that r represents (X, f). The size of (X, f) is denoted by $|(X, f)|$; note that it depends on the concept class chosen.

The concept classes we consider in this part of the dissertation are boolean formulas and syntactically restricted subclasses of them, boolean decision trees, and DFA's. The representations are more or less standard, except each concept representation specifies the relevant domain. For DFA's, the domain of every concept is the set Σ^* itself. For boolean formulas and decision trees, we assume that $\Sigma = \{0,1\}$, and each concept representation specifies a domain of the form $\{0,1\}^n$.

For each finite set S of strings from Σ^*, we define its *table-size*, denoted $\|S\|$, as the sum of the lengths of the strings in S and the number of strings in S. Note that $\|S\| = 0$ if and only if $S = \emptyset$. The table-size of a set of strings is related in a straightforward way to an encoding of a list of the strings; see Chapter 12.

10.2 Queries

For a learning problem we assume that there is an unknown target concept r drawn from a known concept class (R, Dom, μ). Information about the target concept is available to the learning algorithm as the answers to two types of queries: equivalence queries and membership queries.

In an equivalence query, the algorithm gives as input a concept $r' \in R$ with the same domain as the target, and the answer depends on whether $\mu(r) = \mu(r')$. If so, the answer is "yes", and the learning algorithm has succeeded in its goal of exact identification of the target concept. Otherwise, the answer is a *counterexample*, any string $w \in Dom(r)$ on which the functions $\mu(r)$ and $\mu(r')$ differ. We denote an equivalence query on a hypothesis h by $\mathrm{EQ}(h)$.

The *label* for a counterexample $v = \mathrm{EQ}(r')$ is the value of $\mu(r)$ on v, giving its classification by the target concept. Since the hypothesized concept r' and the target concept r differ on the classification of the counterexample v, the label of v is also the complement of the value of $\mu(r')$ on v. *Positive counterexamples* are those with label 1 and *negative counterexamples* are those with label 0.

In a membership query, the learning algorithm gives as input a string $w \in Dom(r)$, and the answer is either 0 or 1. If the answer is equal to the value of $\mu(r)$ on w, then the answer is *correct*. If the answer is not equal to the value of $\mu(r)$ on w, then the answer is an *error* or a *lie*.

In a *standard membership query*, denoted MQ, all the answers are required to be correct. In a *malicious membership query*, denoted MMQ, each answer can be either correct or an error. The related model of *limited membership query* requires that answers of 0 and 1 are always correct but allows a third possible answer, $\perp$.

The answers to malicious membership queries are also restricted as follows.

1. They are *persistent*; that is, different membership queries with the same input string w receive the same answer. Note that non-persistent queries may reveal some information; in case two different queries to the same string receive different answers, the learning algorithm knows that there has been an error on

this string, though this will not in general determine the correct classification of the string. Every algorithm designed to work with persistent queries can be made to work with non-persistent ones by caching the queries and always using the first answer for subsequent queries of the same string.

2. In addition, we bound the quantity of errors permitted in answers to malicious membership queries. One natural quantity to bound would be the number of different strings whose membership queries can be answered incorrectly, and this works well in fixed-length domains. However, in variable-length domains, we wish to account for the lengths of the strings as well as their number.

   Therefore, in general the algorithm is given a bound L on the table-size, $\|S\|$, of the set S of strings whose malicious membership queries are answered erroneously during a single run. In the case of a fixed-length domain, $\{0,1\}^n$, we may instead give a bound ℓ on the number of different strings whose MMQ's are answered incorrectly Note that $L = \ell(n+1)$ is a bound on the table-size in this case.

   Note that when $L = 0$ or $\ell = 0$ there can be no errors in the answers to MMQ's and we have the usual model of standard membership queries as a special case.

We assume that an online adversary controls the choice of counterexamples in answers to equivalence queries and the choice of which elements of the domain will be answered with errors in malicious membership queries. When the learning algorithms we consider are deterministic, the adversary may be viewed as choosing in advance the set of strings for which it will give incorrect answers to membership queries, as well as all the counterexamples it will give to equivalence queries.

We extend the usual notion of polynomial-time learning to our model of equiv-

alence and malicious membership queries by allowing the polynomial bound on the running time to depend on three parameters, that is, $p(s, n, L)$. Here s is the usual parameter bounding the length of the representation of the target concept, n is the usual parameter bounding the length of the longest counterexample seen so far, and L is a new parameter, bounding the table-size of the set of strings on which MMQ gives incorrect answers.

The definitions are extended in the usual way to cover randomized learning algorithms and their expected running times, and also extended equivalence queries, in which the inputs to equivalence queries and the final result of the algorithm are allowed to come from a concept class different from (usually larger than) the concept class from which the target is drawn.

It is straightforward to transform any algorithm that uses malicious membership queries into one that uses limited membership queries. Every $\perp$ answer can be replaced by a 0 or a 1 arbitrarily and given to the learner. Therefore learning with malicious membership queries is at least as hard as learning with limited membership queries in the strict model. The same applies to learning from equivalence and malicious membership queries and learning from equivalence and limited membership queries in the strict model. Note that the most general kind of membership query is one in which both wrong and $\perp$ answers are possible, but such queries are not harder than the malicious ones. This work considers only malicious membership queries. It is possible, however, that limited membership queries or a mix of malicious and limited membership queries would allow other classes of concepts to be learned.

10.3 Monotone DNF Formulas

We assume a set of propositional variables V and denote its elements by x_1, x_2, $\ldots$, x_n, where n is the cardinality of V. A monotone DNF formula over V is a DNF formula over V where no literal is negated. The domain of such a formula is $\{0,1\}^n$. For example, for $n = 20$,

$$x_1 x_4 \vee x_2 x_{17} x_3 \vee x_9 x_5 x_{12} x_3 \vee x_8$$

is a monotone DNF formula (with domain $\{0,1\}^{20}$). We assume that the target formula h^* has been minimized, that is, it contains no redundant terms. (Incidentally, there is an efficient algorithm to minimize the number of terms of a monotone DNF formula.) We denote the number of terms by m.

We view the domain $\{0,1\}^n$ of monotone DNF formulas as a lattice, with componentwise "or" and "and" as the lattice operations. The top element is the vector of all 1's, and the bottom element is the vector of all 0's. The elements are partially ordered by $\leqslant$, where $v \leqslant w$ if and only if $v[i] \leqslant w[i]$ for all $1 \leqslant i \leqslant n$. Often we refer to the examples as points of the hypercube $\{0,1\}^n$. For a point v, all points w such that $w \leqslant v$ are called the *descendants* of v. Those descendants that can be obtained by changing exactly one coordinate of v from a 1 to a 0 are called the *children* of v. The *ancestors* and the *parents* are defined similarly. Note that v is both a descendant and ancestor of itself.

For convenience, we use a representation of monotone DNF formulas in which each term is represented by the minimum vector, in the ordering $\leqslant$, that satisfies the term. Thus, vector 10011 (where $n = 5$) denotes the term $x_1 x_4 x_5$. In this

representation, if h is a monotone DNF formula and v is a vector in the domain, v satisfies h if and only if for some term t of h, $t \leqslant v$. That is, a monotone DNF formula is satisfied only by the ancestors of its terms. In the other direction, we say that term t *covers* point v if and only if v satisfies t. For the sake of simplicity we often use in our algorithms something called "the empty DNF formula". This is the formula with no terms, which is not satisfied by any point, and is therefore the identically false formula.

For any n-argument boolean function f, we call point x a *local minimum point* of f if $f(x) = 1$ but for every child y of x in the lattice, $f(y) = 0$. The local minimum points of a minimized DNF formula represent its terms in our representation.

For two n-argument boolean functions f_1 and f_2 we define the set $Err(f_1, f_2)$ to be the set of points where they differ. I.e., $Err(f_1, f_2) = \{ x : f_1(x) \neq f_2(x) \}$. The cardinality of $Err(f_1, f_2)$ is called the *distance* between f_1 and f_2 and is denoted by $d(f_1, f_2)$.

There is a simple algorithm that learns monotone DNF formulas from equivalence and standard membership queries. It cannot tolerate any errors or omissions but has been the basis for many algorithms that can, including several in this thesis. This algorithm was given by Angluin [4] and is shown in Figure 10.1.

The idea for this algorithm is to always ask an equivalence query with a subset of the terms of the target formula, starting with the empty DNF formula. Thus, every counterexample it receives must be positive and lie above some local minimum point, corresponding to an unknown term of the target function. The algorithm uses a simple subroutine REDUCE, shown in Figure 10.2, to find such terms.

```
LearnStandard()
{
    h = "the empty DNF formula";

    While ((v = EQ(h)) ≠ "yes")
    {
        w = Reduce(v);
        Add term w to h;
    }

    Output h;
}
```

Figure 10.1 Algorithm for learning monotone DNF from EQ's and standard MQ's

```
Reduce(v)
{

    For (each child w of v)
        If (MQ(w) == 1)
            Return Reduce(w);

    Return v;
}
```

Figure 10.2 Subroutine Reduce

Chapter 11

Malicious Membership Queries

In this chapter we present and analyze an algorithm that uses equivalence and malicious membership queries to learn monotone DNF formulas. The key idea is to depend on equivalence queries as much as possible, since they are correct.

11.1 The Algorithm

The algorithm keeps track of all the counterexamples and their labels received through equivalence queries and consults them first, before asking a membership query. The pairs of counterexamples and their labels are kept in a set named *CounterExamples*. Obviously, for a positive counterexample v, if $x \geqslant v$ then it is not worth making a membership query about x; it must be a positive point. Similarly, for a negative counterexample v, if $x \leqslant v$ then x has to be a negative point of the target formula. For this reason we define a subroutine CheckedMQ and use it instead of a membership query. The subroutine is given in Figure 11.1.

```
CheckedMQ(x, CounterExamples)
{
    If (∃⟨v,1⟩ ∈ CounterExamples s.t. x ⩾ v)
        Return 1;

    If (∃⟨v,0⟩ ∈ CounterExamples s.t. x ⩽ v)
        Return 0;

    Return MMQ(x);
}
```

Figure 11.1 Subroutine CheckedMQ

```
Reduce(v, CounterExamples)
{
    For (each child w of v)
        If (CheckedMQ(w, CounterExamples) == 1)
            Return Reduce(w, CounterExamples);

    Return v;
}
```

Figure 11.2 Subroutine Reduce

Our algorithm is based on the algorithm LearnStandard, illustrated in Figure 10.1 and given by Angluin [4]. Thus, it also uses a subroutine Reduce in order to move down in the lattice from a positive counterexample. All the membership queries are done using the subroutine CheckedMQ, which possibly lets the algorithm avoid some incorrect answers. The subroutine Reduce for the new algorithm is given in Figure 11.2.

The algorithm for exactly identifying monotone DNF formulas using equivalence queries and malicious membership queries is given in Figure 11.3.

The algorithm is based on a few simple ideas. A positive counterexample is reduced to a point that is added as a term to the existing hypothesis h, which is a

```
LearnMonDNF()
{
    CounterExamples = ∅;
    h = "the empty DNF formula";

    While ((v = EQ(h)) ≠ "yes")
    {
        Add ⟨v, (1 − h(v))⟩ to CounterExamples;

        If (h(v) == 0)
        {
            w = Reduce(v, CounterExamples);
            Add term w to h;
        }
        Else
            For (each term t of h)
                If (t(v) == 1)
                    Delete term t from h;

    }

    Output h;
}
```

Figure 11.3 Algorithm for learning monotone DNF from EQ's and MMQ's

monotone DNF. That is, the new hypothesis classifies the latest counterexample and possibly some other points as positive.

Negative counterexamples are used to detect inconsistencies between membership and equivalence queries. They show that there have been errors in membership queries that have caused wrong terms to be added to the hypothesis. The algorithm reacts by removing all the terms that are inconsistent with the latest counterexample. These are the terms that have the negative counterexample above them. A term is removed only when there is a negative counterexample above it.

11.2 Analysis of LearnMonDNF

Theorem 8 LearnMonDNF *learns the class of monotone DNF formulas in polynomial time using equivalence and malicious membership queries.*

We need a definition and a simple lemma before proving the theorem.

Let h^* be a monotone boolean function on $\{0,1\}^n$, and let h' be an arbitrary boolean function on $\{0,1\}^n$. Let C be any subset of $\{0,1\}^n$. The *monotone correction of h' with h^* on C*, denoted $mc(h', h^*, C)$, is the boolean function h'' defined for each string $x \in \{0,1\}^n$ as follows:

$$h''(x) \stackrel{\text{def}}{=} \begin{cases} 1, & \text{if there exists } y \in C \text{ such that } y \leq x \text{ and } h^*(y) = 1; \\ 0, & \text{if there exists } y \in C \text{ such that } x \leq y \text{ and } h^*(y) = 0; \\ h'(x), & \text{otherwise.} \end{cases}$$

Note that since h^* is monotone, the first two cases above cannot hold simultaneously. It is also clear that if the value of $h''(x)$ is determined by one of the first two cases, $h''(x) = h^*(x)$. We prove a simple monotonicity property of the monotone correction operation.

Lemma 9 *Suppose h^* is a monotone boolean function and h' is an arbitrary boolean function on $\{0,1\}^n$. Let $C_1 \subseteq C_2$ be two subsets of $\{0,1\}^n$. Let $h_1 = mc(h', h^*, C_1)$ and $h_2 = mc(h', h^*, C_2)$. Then the set of points on which h_2 and h^* differ is contained in the set of points on which h_1 and h^* differ. That is, $Err(h_2, h^*) \subseteq Err(h_1, h^*)$.*

Proof: Let x be an arbitrary point on which $h_2(x) \neq h^*(x)$. Then it must be

that $h_2(x) = h'(x)$ and there does not exist any point $y \in C_2$ such that $x \leq y$ and $h^*(y) = 0$ or $y \leq x$ and $h^*(y) = 1$. Since C_1 is contained in C_2, there is no point $y \in C_1$ such that $x \leq y$ and $h^*(y) = 0$ or such that $y \leq x$ and $h^*(y) = 1$. Thus, $h_1(x) = h'(x)$ and $h_1(x) \neq h^*(x)$. Consequently, $Err(h_2, h^*) \subseteq Err(h_1, h^*)$. ■

Now we start the proof of Theorem 8.

Proof: Let h^* denote the target concept, an arbitrary monotone DNF formula over $\{0,1\}^n$ with m terms. Let ℓ be a bound on the number of strings whose MMQ's are answered incorrectly. Because equivalence queries are answered correctly, if the algorithm ever halts, the hypothesis output is correct, so we may focus on proving a polynomial bound on the running time.

Since LearnMonDNF is deterministic and the target concept h^* is fixed, we may assume that the adversary chooses in advance how to answer all the queries, that is, chooses a sequence $y_1, y_2, \ldots$ of counterexamples to equivalence queries and a set S of strings on which to answer MMQ's incorrectly. Note that $|S| \leqslant \ell$.

In turn, these choices determine a particular computation of LearnMonDNF which we now focus on. It suffices to bound the length of this computation. In this computation the answers to MMQ's agree with the boolean function h_0 defined as follows. $h_0(x) = h^*(x)$ for all strings $x \notin S$ and $h_0(x) = 1 - h^*(x)$ for all strings $x \in S$. Also, if CheckedMQ is called with arguments x and C, where x is some string and C is the current value of the set *CounterExamples*, its return value agrees with the value of the boolean function $mc(h_0, h^*, C)$ on string x.

The set *CounterExamples* only changes when a new counterexample is received. Therefore, the successive distinct sets of counterexamples in this computation can

be denoted by $C_0, C_1, \ldots$, where $C_0 = \varnothing$ and $C_i = C_{i-1} \cup \{y_i\}$, for $i = 1, 2, \ldots$. If we also define $h_i = mc(h_0, h^*, C_i)$ for $i = 1, 2, \ldots$, then CheckedMQ answers according to h_0 until the first counterexample is received, then according to h_1 until the second counterexample is received, and so on.

Clearly, since h_0 disagrees with h^* on at most ℓ strings, $d(h_0, h^*) \leqslant \ell$. Since the sets $C_0, C_1, \ldots$ are monotonically nondecreasing, Lemma 9 shows that $Err(h_i, h^*) \subseteq Err(h_{i-1}, h^*)$ for $i = 1, 2, \ldots$.

We say that a counterexample y_i *corrects a positive error* at point x if $h_{i-1}(x) = 1$ but $h_i(x) = h^*(x) = 0$. We say that a counterexample y_i *corrects a negative error* at point x if $h_{i-1}(x) = 0$ but $h_i(x) = h^*(x) = 1$. Note that from the construction of CheckedMQ it follows that positive errors can be corrected only by negative counterexamples and negative errors can be corrected only by positive counterexamples. Let there be ℓ_p positive and ℓ_n negative errors corrected in the whole computation. Of course, $\ell_p + \ell_n \leqslant \ell$.

Claim 11 *If* Reduce *is called after counterexample y_i and before counterexample y_{i+1}, it returns a local minimum point of h_i.*

Proof: After y_i is added to *CounterExamples*, CheckedMQ answers according to h_i. The claim follows from the construction of Reduce. ■

Claim 12 *The following condition is preserved. At the $i+1$-th equivalence query* EQ(h)*, each term of h is a positive point of h_i.*

Proof: We prove the claim by induction.

Induction Basis: The first EQ is made on an empty formula. Thus, the claim is vacuously true.

Induction Step: Suppose the claim is true up to the i-th EQ. Let h' be the hypothesis h at the i-th EQ and h'' be the hypothesis h at the $i+1$-th EQ. There are two cases to consider.

Case 1: y_i is a positive counterexample. Then $h_i(x) = 1$ if and only if $h_{i-1}(x) = 1$ or $x \geqslant y_i$. Let t be the term returned by REDUCE with parameters y_i and *CounterExamples*. Then $h'' = h' \vee t$. Let t'' be a term in h''. Then either t'' is a term of h' or $t'' = t$. If t'' is a term of h' then $h_{i-1}(t'') = 1$ by the inductive assumption and therefore $h_i(t'') = 1$. If $t'' = t$ then $h_i(t'') = 1$ since t was returned by REDUCE(y_i, *CounterExamples*) which used CHECKEDMQ, which answered according to h_i.

Case 2: y_i is a negative counterexample. Then $h_i(x) = 1$ if and only if $h_{i-1}(x) = 1$ and $x \not\leqslant y_i$. Let t'' be a term in h'', which consists of all those terms t' of h' such that $t' \not\leqslant y_i$. Therefore, $t'' \not\leqslant y_i$ and by the inductive assumption $h_{i-1}(t'') = 1$. It follows that $h_i(t'') = 1$. ■

Claim 13 *Once a term x is deleted from hypothesis h, it can never reappear in it.*

Proof: Since x was deleted, there must have been a negative counterexample y_i such that $y_i \geqslant x$. But then $(y_i, 0)$ belongs to *CounterExamples* and the subroutine call CHECKEDMQ(x, *CounterExamples*) can never return 1 again, which is necessary for x to be added to h. ■

We divide the run of the algorithm into non-overlapping *stages*. A new stage begins either at the beginning of the run or with a new negative counterexample. Thus

with each new stage *CounterExamples* contains one more negative counterexample and possibly (but not necessarily) some new positive counterexamples. The following claim establishes that the distance $d(h_i, h^*)$ decreases with every new stage.

Claim 14 *Every negative counterexample corrects at least one error. More formally, if y_i is a negative counterexample, then there exists $x \in \{0,1\}^n$ such that $h_{i-1}(x) = 1$ and $h_i(x) = h^*(x) = 0$.*

Proof: Let y_i be a negative counterexample returned by EQ(h). Hence $h(y_i) = 1$, and there is some term $x \leqslant y_i$ in h. By Claim 12, $h_{i-1}(x) = 1$.

Since $h^*(y_i) = 0$ and $y_i \geqslant x$ it follows that $h^*(x) = 0$. By the definition of h_i it follows that $h_i(x) = 0$. ■

From Claim 14 it follows that there are at most ℓ_p negative counterexamples. Hence there are at most $\ell_p + 1$ stages in the run of the algorithm.

We divide each stage of the algorithm into non-overlapping *substages*. A substage begins either at the beginning of a stage or with a new positive counterexample that corrects an error. Obviously there can be no more than ℓ_n positive counterexamples that correct errors and hence no more than $\ell_p + \ell_n + 1$ substages in the whole run of the algorithm. The distance $d(h_i, h^*)$ decreases with every new substage. If, however, functions h_i and h_j belong to the same substage, they are equivalent and their local minima are the same. This allows us to bound the total number of positive counterexamples.

Claim 15 *Every new positive counterexample is reduced to a local minimum point of $h_0, h_1, \ldots$ that has not been found earlier.*

Proof: Let v be a positive counterexample that REDUCE is started with. Let t be the point REDUCE(v, *CounterExamples*) returns. Assume, by way of contradiction, that t has already been found before. From Claim 13 it follows that t is a term in h. Since $v \geqslant t$, it follows that $h(v) = 1$. This is a contradiction to the assumption that v is a positive counterexample. ■

We denote the set of local minimum points of a boolean function f by $Lmp(f)$. We bound the total number of different local minima of the functions $h_0, h_1, \ldots$.

Lemma 10 *Let f and f' be n-argument boolean functions such that $Err(f, f') = \{x\}$. Then*

(a) If $f'(x) = 1$ then $\left|Lmp(f') - Lmp(f)\right| \leqslant 1$.

(b) If $f'(x) = 0$ then $\left|Lmp(f') - Lmp(f)\right| \leqslant n$.

Proof:

(a) The only point that can be a local minimum of f' and is not a local minimum of f, is x itself. The claim follows immediately.

(b) Any point which is a local minimum of f' but not of f is a parent of x. Since x has at most n parents, the claim follows. ■

Corollary 11 *Let f and f' be n-argument boolean functions such that $Err(f, f')$ contains d_p positive points of f' and d_n negative points of f'. Then*

$$\left|Lmp(f') - Lmp(f)\right| \leqslant nd_n + d_p.$$

Corollary 12 *Let $g_0, g_1, \ldots, g_r$ be the subsequence of $h_0, h_1, \ldots$, such that each g_i is the first of all the h_j's in its substage. Let $Err(h^*, g_{i-1}) - Err(h^*, g_i)$ contain $\ell_{p,i-1}$ positive and $\ell_{n,i-1}$ negative points of h^* for all $i = 1, 2, \ldots, r$. Let $Err(h^*, g_r)$ contain $\ell_{p,r}$ positive and $\ell_{n,r}$ negative points of h^*. Then the total number of different local minima of functions $g_0, g_1, \ldots, g_r, h^*$ is bounded above by $m + n\sum_{i=0}^{r} \ell_{n,i} + \sum_{i=0}^{r} \ell_{p,i}$.*

Proof: Note that $g_0, g_1, \ldots, g_r$ are the different functions in $h_0, h_1, \ldots$, and that CheckedMQ first answers according to g_0, then according to g_1 and so on. Obviously, $Err(h^*, g_i) \subseteq Err(h^*, g_{i-1})$ and $Err(g_{i-1}, g_i) = Err(h^*, g_{i-1}) - Err(h^*, g_i)$ for all $i = 1, 2, \ldots, r$. Also note that for each $i = 0, 1, \ldots, r-1$, one of $\ell_{p,i}$ and $\ell_{n,i}$ is 0, but $\ell_{p,r}$ and $\ell_{n,r}$ may both be positive.

We want to find $\left|\bigcup_{i=0}^{r} Lmp(g_i) \cup Lmp(h^*)\right|$, knowing that $\left|Lmp(h^*)\right| = m$. Since

$$\bigcup_{i=0}^{r} Lmp(g_i) \cup Lmp(h^*) \subseteq Lmp(h^*) \cup \big(Lmp(g_r) - Lmp(h^*)\big) \cup \bigcup_{i=0}^{r-1} \big(Lmp(g_i) - Lmp(g_{i+1})\big),$$

from Corollary 11 it follows that

$$\left|\bigcup_{i=0}^{r} Lmp(g_i) \cup Lmp(h^*)\right| \leqslant \left|Lmp(h^*)\right| + (n\ell_{n,r} + \ell_{p,r}) + \sum_{i=0}^{r-1} (n\ell_{n,i} + \ell_{p,i})$$

and the bound follows. ■

Since each error can be corrected at most once, it follows that $\sum_{i=0}^{r} \ell_{n,i} \leqslant \ell_n$ and $\sum_{i=0}^{r} \ell_{p,i} \leqslant \ell_p$. Hence the total number of the local minima and the total number of positive counterexamples that can be found in a computation is bounded by $m + n\ell_n + \ell_p$. The number of negative counterexamples in a complete run is

bounded by the number of positive errors. The total number of counterexamples is therefore bounded by $m + \ell_n n + \ell_p + \ell_p \leqslant m + \ell(n+1) = O(m + \ell n)$.

We now count the number of membership queries in a complete run of the algorithm. Each positive counterexample v may cause at most $n(n+1)/2$ membership queries, before Reduce(v, *CounterExamples*) returns. Therefore there can be at most $O(mn^2 + \ell n^3)$ membership queries in a complete run of the algorithm.

It is also clear that the running time of the algorithm is polynomial in m, n and ℓ. This concludes the proof of Theorem 8. ■

Comparing LearnMonDNF with the algorithm for learning monotone DNF formulas from equivalence and limited membership queries, given by Sloan and Turán [36], we see that LearnMonDNF is able to cope with MMQ's instead of the more benign limited membership queries, but at a cost of making more queries overall. In particular, it uses $O(m + \ell n)$ equivalence queries while the algorithm of Sloan and Turán only $m + \ell + 1$, and it uses $O(mn^2 + \ell n^3)$ malicious membership queries while the algorithm of Sloan and Turán uses $mn + \ell n$ limited membership queries.

Chapter 12

Finite Exceptions

In this chapter we define concepts with exceptions and introduce concept classes that are polynomially closed under finite exceptions. We present several examples of such classes. Finally, we develop a polynomial-time algorithm that learns the class of monotone DNF formulas with finite exceptions from equivalence and membership queries.

12.1 Exceptions

For a concept (X, f) and a finite set $S \subseteq X$, we define *the concept* (X, f) *with exceptions* S, denoted *xcpt*$((X, f), S)$, as the concept (X, f') where $f'(w) = f(w)$ for strings in $X - S$, and $f'(w) = 1 - f(w)$ for strings in S. (Thus f and f' have the same domain, and are equal except on the set of strings S, which is a subset of their common domain.) It is useful to note that S is partitioned by (X, f) into the set of *positive exceptions* S_+ that are classified as negative by f, and the set of

negative exceptions S_- that are classified as positive by f. When the domain X of a function f is clearly understood and we do not wish to mention it explicitly, we often call this function itself a concept. In this case we also use a shorthand notation for $xcpt((X, f), S)$, namely, we just write $xcpt(f, S)$.

A concept class (R, Dom, μ) is *closed under finite exceptions* provided that for every concept (X, f) represented by (R, Dom, μ) and every finite set $S \subseteq X$, the concept $xcpt((X, f), S)$ is also represented by (R, Dom, μ). If, in addition, there is a fixed polynomial of two arguments such that the concept $xcpt((X, f), S)$ is of size bounded by this polynomial in the size of (X, f) and $\|S\|$, we say that (R, Dom, μ) is *polynomially closed under finite exceptions.*

This definition differs from a similar earlier definition given by Board and Pitt [16] in that we do not require the existence of a polynomial-time algorithm that produces the new concept given the old concept and a list of exceptions. However, for the classes that we consider there are such algorithms.

We define a natural operation of adding finite exception tables to a class of concepts to produce another class of concepts that "embeds" the first and is polynomially closed under finite exceptions.

We assume $\Sigma \subseteq \Gamma$ and $|\Gamma| \geq 2$. We define a simple encoding e that takes a string r from Γ^* and a finite set of strings $S \subseteq \Sigma^*$ and produces a string r' in Γ^* from which r and the elements of S can easily be recovered, and is such that $|r'| = 2(1+|r|+\|S\|)$. The details of the encoding are as follows.

Assume that 0 and 1 are distinct symbols in Γ. We define

$$e_b(b_1 b_2 \ldots b_j) \stackrel{\text{def}}{=} bbbb_1bb_2 \ldots bb_j,$$

for $b \in \{0,1\}$ and $b_1, b_2, \ldots, b_j \in \Gamma$. Note that $|e_b(w)| = 2(1+|w|)$ for every string $w \in \Gamma^*$. We then define the encoding of r and S as

$$r' = e(r,S) \stackrel{\text{def}}{=} e_0(r)e_1(s_1)e_0(s_2)\ldots e_{k \bmod 2}(s_k),$$

where $s_1, s_2, \ldots, s_k$ are the strings in S.

Given a concept class (R, Dom, μ), we define the *class obtained from it by adding exception tables* to be $(R', \mathit{Dom}', \mu')$, where R' is the set of all strings of the form $e(r,S)$ such that $r \in R$ and S is a finite subset of $\mathit{Dom}(r)$, and for each $r' \in R'$, the concept represented by $r' = e(r,S)$ is the concept represented by r with exceptions S, that is, $(\mathit{Dom}'(r'), \mu'(r')) = \mathit{xcpt}((\mathit{Dom}(r), \mu(r)), S)$.

For example, adding exception tables to the monotone DNF formulas produces a concept class that we call *monotone DNF formulas with finite exceptions*. More detailed discussion of classes obtained by adding exception tables and of polynomial closure under finite exceptions can be found in the next section.

12.2 Examples and Lemmas

Example 2 The class of regular languages represented by DFA's is polynomially closed under finite exceptions. Board and Pitt give an algorithm that takes as input a DFA M and an exception set S, and produces a new DFA for $\mathit{xcpt}(M,S)$ [16]. The DFA's size is polynomial in the size of M and S.

Example 3 Another example of a class that is polynomially closed under finite exceptions is the class of boolean decision trees. A boolean decision tree over a set

of propositional variables V is a full (but not necessarily complete) labeled binary tree. Each internal node of it is labeled with a boolean variable from V and each leaf is labeled with 0 or 1. The value of a boolean decision tree on a domain point is computed recursively starting at the root. Each internal node tests the variable that it is labeled with. If the variable is set to 1, the result is the value of the right subtree, otherwise it is the value of the left subtree. The value of a leaf node is its label. The following result is taken from the paper by Board and Pitt [16] but since the construction is not given there, we sketch it here.

Lemma 11 *The class of boolean decision trees is polynomially closed under finite exceptions.*

Proof: Let T be a decision tree on n variables. Let S be the exception set for T. We construct the decision tree for $xcpt(T, S)$ as follows. We treat each exception point $x \in S$ individually. First we walk down from the root of the original tree T to see where x is located in it. If this leads us to a leaf with depth n (i.e., if all variables are tested on this path), then we just reverse the value of the leaf, because this path is for x only. However, if we find ourselves at a leaf with depth less than n, we have to add new internal nodes to the tree. Denote the value of this leaf by b. We then continue the path that led us to this leaf with a path in which all the remaining variables are tested. We move to the right if the variable is 1 and to the left if it is 0. We end the path by a leaf with value $1 - b$. For each new internal node on the path, we make the other child (the one not on the path) a leaf, and give it the original value b. Thus, each counterexample adds at most n new internal nodes to the tree. The size of the new tree, measured as the number of internal nodes, is bounded by $|T| + n \times |S| = |T| + \|S\|$. ■

Example 4 One more interesting example is the class of DNF formulas.

Lemma 12 *The class of DNF formulas is polynomially closed under finite exceptions.*

Proof: Let f be an m-term DNF formula over n variables and S be an exception set for it. Let S be partitioned into the sets of positive and negative exceptions (S_+ and S_-, respectively), as described in Section 12.1. We construct a DNF formula for $xcpt(f, S)$ from the formula $(f \wedge f_-) \vee f_+$, where f_- is a DNF formula which is true on all the points in its domain except the ones in S_-, and f_+ is a DNF formula which is true exactly on the points in S_+. The domain for all these formulas is $\{0, 1\}^n$.

Obtaining f_+ is easy—straightforward disjunction of all the *terms* in S_+, where we make terms from points by substituting the respective variable for a 1 value of a coordinate and its negation for a 0 value. Obtaining f_- is harder. First we make a decision tree corresponding to f_-. We put each point from S_- individually in the tree as a 0-valued leaf at the end of a path of length n. All the remaining leaves get value 1. Then for each leaf with value 1 we make a term that will go into f_- by following the path from this leaf to the root. Obviously f_- has at most $n \times |S_-|$ terms. Thus, after "multiplying" the terms out, the formula $(f \wedge f_-) \vee f_+$ will have at most $mn \times |S_-| + |S_+| \leqslant (mn + 1) \times |S|$ terms. ■

For example, if $f = x_1\overline{x}_2 \vee x_3$ is a DNF formula over $\{0, 1\}^3$ and the set of exceptions S is $\{010, 001, 101\}$, then $S_+ = \{010\}$ and $S_- = \{001, 101\}$. The DNF formula f_+ that is true only on points from S_+ is $\overline{x}_1 x_2 \overline{x}_3$ and a possible decision tree corresponding to the function that is true on all points except the ones in S_- is given

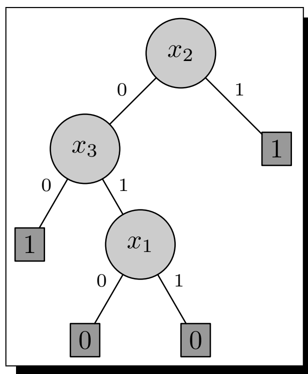

Figure 12.1 A decision tree corresponding to the formula f_-

in Figure 12.1. Formula f_- is then $\overline{x}_3\overline{x}_2 \vee x_2$. Hence,

$$\begin{aligned} xcpt(f, S) &= \big((x_1\overline{x}_2 \vee x_3) \wedge (\overline{x}_3\overline{x}_2 \vee x_2)\big) \vee \overline{x}_1 x_2 \overline{x}_3 \\ &= x_1\overline{x}_2\overline{x}_3 \vee x_2 x_3 \vee \overline{x}_1 x_2 \overline{x}_3. \end{aligned}$$

Example 5 By duality it follows that the class of CNF formulas is polynomially closed under finite exceptions.

Note that stronger bounds on the size of the new formula can be obtained by using the result by Zhuravlev and Kogan [41]. We, however, chose to present a simpler argument. Also note that the size bound is insufficient for *strong polynomial closure under exception lists* as defined by Board and Pitt [16].

Example 6 As our final example we show that any class that is obtained by adding exception tables to another class is polynomially closed under finite exceptions.

Lemma 13 *Let* (R, Dom, μ) *be any class of concepts. Then the concept class obtained from it by adding exception tables is polynomially closed under finite exceptions.*

Proof: Let (R', Dom', μ') be the class of concepts obtained from (R, Dom, μ) by adding exception tables, as defined in Section 12.1. Let (X', f') be any concept from (R', Dom', μ') and let $r' \in R'$ be a shortest representation of (X', f'). Then there exists a concept $r \in R$ and a finite set $S \subseteq Dom(r)$, such that $(Dom'(r'), \mu'(r'))$, the concept represented by r', is equal to $xcpt((Dom(r), \mu(r)), S)$, the concept represented by r with exceptions S, and $|r'| = 2(1 + |r| + \|S\|)$. Let $S' \subseteq Dom'(r') = Dom(r)$ be any finite set. Let concept h'' be defined as $h'' \stackrel{\text{def}}{=} xcpt((Dom'(r'), \mu'(r')), S')$. It is easy to see that $h'' = xcpt((Dom(r), \mu(r)), S \triangle S')$ and thus h'' is represented by some $r'' \in R'$ with size $2(1+|r|+\|S \triangle S'\|) \leqslant 2(1+|r|+\|S\|+\|S'\|) = |r'|+2\|S'\|$. ■

Corollary 13 *The class of monotone DNF formulas with finite exceptions is polynomially closed under finite exceptions.*

12.3 The Learning Algorithm

In this section, we present an algorithm that learns the class of monotone DNF formulas with finite exceptions. The target concept is a boolean function on n variables $h^* \stackrel{\text{def}}{=} xcpt(h^*_M, S^*)$, where h^*_M is some monotone DNF formula and S^* is a set of exceptions for it. The domain of the target concept is $\{0, 1\}^n$.

We assume that we have an upper bound on the cardinality of S^* and denote it by l (i.e., $|S^*| \leqslant l$). If this bound is not known, we can start out by assuming it to be any positive integer and doubling it whenever convergence is not achieved within the proper time bound, which will be given later. We assume that h^*_M is minimized and has m terms.

Like LEARNMONDNF, our current algorithm also has a set *CounterExamples*

```
GetExceptions(h, CounterExamples)
{
    S = ∅;

    For (each ⟨x, b⟩ ∈ CounterExamples)
        If ((h(x) ≠ b))
            Add x to S;

    Return S;
}
```

Figure 12.2 Subroutine GetExceptions

that stores all labeled counterexamples received from equivalence queries. The purpose of it is slightly different: it lets the algorithm conclude that some points cannot be classified by h^*_M alone, and, therefore, have to be included in the exception set.

The algorithm tries to find a suitable monotone DNF formula, which, coupled with a proper exception set, would give the target concept. The equivalence queries are made on a pair $\langle h, S\rangle$ of a monotone DNF formula h and a set of exceptions S. The algorithm focuses only on building h, and sets S to be those elements of the set *CounterExamples* that are currently misclassified by h. It uses a simple subroutine GetExceptions for building S. The subroutine is given in Figure 12.2.

In order to classify the counterexamples received, the algorithm needs to evaluate the current function $xcpt(h, S)$. This is done by another very simple subroutine TheFunction, given in Figure 12.3.

Our algorithm also uses a subroutine Reduce similar to those used in Chapter 11 and by the algorithm LearnStandard shown in Figure 10.1. With the help of this subroutine it can move down in the lattice from a positive counterexample. Its goal is to reduce the positive counterexample to some point that can be added as a term to the formula h. Then the new hypothesis would classify the counterexample and

```
TheFunction(h, S, x)
{
    If (x ∈ S)
        Return 1 − h(x);
    Else
        Return h(x);
}
```

Figure 12.3 Subroutine TheFunction

```
Reduce(v, CounterExamples)
{
    For (each child w of v)
        If ((MQ(w) == 1) and (|{ y ⩾ w : ⟨y, 0⟩ ∈ CounterExamples }| ⩽ l))
            Return Reduce(w, CounterExamples);
    Return v;
}
```

Figure 12.4 Subroutine Reduce

possibly some other points as positive. However, this may not always be possible. There can be overwhelming evidence that the candidate point is just a positive exception and thus should not be added to h. More precisely, if there are more than l negative counterexamples above a term of h, then they all have to be in the exception set, which is then too big. Therefore the current subroutine Reduce is somewhat more complex and checks whether a point has enough evidence to be an undoubted exception point or not. The subroutine is given in Figure 12.4.

The algorithm for learning monotone DNF formulas with at most l exceptions using equivalence queries and membership queries is given in Figure 12.5.

The algorithm is based on the following ideas. Each positive counterexample is reduced if possible to a new term to be added to the formula, as was explained above.

```
LearnMonDNFwithFX()
{
    S = CounterExamples = ∅;
    h = "the empty DNF formula";

    While ((v = EQ(⟨h, S⟩)) ≠ "yes")
    {
        Add ⟨v, (1 − TheFunction(h, S, v))⟩ to CounterExamples;

        If (TheFunction(h, S, v) == 1)
            For (each term t of h)
                If (|{ w ⩾ t : (w, 0) ∈ CounterExamples }| > l)
                    Delete term t from h;

        For (each ⟨x, 1⟩ ∈ CounterExamples)
            If ((h(x) == 0) and (|{ y ⩾ x : ⟨y, 0⟩ ∈ CounterExamples }| ⩽ l))
            {
                w = Reduce(x, CounterExamples);
                Add term w to h;
            }

        S = GetExceptions(h, CounterExamples);
    }

    Output ⟨h, S⟩;
}
```

Figure 12.5 Algorithm for learning monotone DNF with finite exceptions

In case this is not possible, the algorithm benefits anyway by storing it in the set *CounterExamples*.

Negative counterexamples imply that there are not as many positive points in the target concept as we thought. Sometimes more exception points are necessary for the hypothesis to be correct. Other times some terms have to be removed from the formula. Deleting a term happens only when there is enough evidence that a term is wrong, namely, when there are more than l negative counterexamples above it.

12.4 Analysis of the Algorithm

Theorem 9 LearnMonDNFwithFX *learns the class of monotone DNF formulas with exceptions in polynomial time using equivalence and standard membership queries.*

Proof: We begin the analysis with this simple claim.

Claim 16 *Once a term t is deleted from hypothesis h, it can never reappear in it.*

Proof: A term t can be deleted only if there are more than l negative counterexamples above it. To reappear, t must be returned by Reduce. But every point returned by Reduce must have at most l negative counterexamples above it at the time it is returned, so Reduce cannot return t again. ■

The following lemma shows what points Reduce can return.

Lemma 14 Reduce *always returns either a local minimum of h^* or a parent of a positive exception in S^*.*

Proof: First note that Reduce can only be called on points x such that $h^*(x) = 1$ and can only return points w such that $h^*(w) = 1$. Let w be a point returned by Reduce. Assume w is not a local minimum point of h^*. Then there is some child y of w such that $h^*(y) = 1$, and the number of negative counterexamples above y must exceed l (or else Reduce would have been called recursively on y). Hence, y cannot be above any term t of h^*_M, since each term t can have at most l negative counterexamples above it. Therefore, y is a positive exception in S^*. ■

Now we are ready to bound the number of different points that can be returned by the subroutine REDUCE.

Claim 17 *The number of different points that* REDUCE *can return is at most* $m + (n+1)l$.

Proof: By Lemma 14, the number of different points that can be returned by REDUCE is at most the number of points that are local minima of h^* or parents of positive exceptions in S^*. Let S^* contain l_p positive exceptions and l_n negative exceptions, where $l_p + l_n \leqslant l$. The formula h^*_M has m terms and therefore m local minima. By Lemma 10, the number of local minima of h^* is at most $m + l_p + nl_n$. Each positive exception has at most n parents, so the number of parents of positive exceptions is bounded by nl_p. Thus, the number of different points REDUCE can return, and the number of calls to REDUCE, is bounded by $m + (n+1)l_p + nl_n \leqslant m + (n+1)l$. ■

All equivalence queries are asked about the current hypothesis *xcpt*(h, S). Since S is computed right before each equivalence query, the argument of an equivalence query is always consistent with all the counterexamples seen to that point. Let h_i and h_j denote the function *xcpt*(h, S) at the time when i-th and j-th equivalence query is asked, respectively, and let $i < j$. Let v_i be the counterexample returned by the i-th equivalence query. Clearly, the values of $h_i(v_i)$ and $h_j(v_i)$ must be different. Thus, the function *xcpt*(h, S) is different for each equivalence query. This allows us to bound the total number of equivalence queries.

Claim 18 *The number of equivalence queries made before success is bounded by* $O(m^2n^2l^3)$.

Proof: We examine how $xcpt(h, S)$ changes. Either h itself changes, or h remains the same and S changes; namely, it contains exactly one more point, the most recent counterexample.

By Claim 16, each term of h can appear in h or disappear from it only once. Thus each possible term can induce at most two changes in formula h—first by appearing in it and then by disappearing. Thus, h can only change twice as many times as the number of terms that REDUCE can return. Therefore, by Claim 17, there can be at most $2(m + (n + 1)l) + 1$ different functions h in a complete run of the algorithm.

We now count the number of times that S can change while h remains the same. Set S grows larger by one with each new counterexample. It may contain some points x such that $h(x) = 1$ and possibly some points x such that $h(x) = 0$. We bound the number of each of these separately.

Each point $x \in S$ such that $h(x) = 1$ is above some term of h. No term can have more than l negative counterexamples above it. Therefore, the number of points $x \in S$ such that $h(x) = 1$ can be bounded by l times the bound $m + (n + 1)l$ on the number of different terms of h, that is, by $ml + (n + 1)l^2$.

Each point $x \in S$ such that $h(x) = 0$ is a positive counterexample, and thus is not above any term in h. Such an x must have more than l negative counterexamples above it. Otherwise, the algorithm would have called REDUCE on x and added a new term $t \leqslant x$ to h. If x has more than l negative counterexamples above it, then it cannot be above a term in h^*_M and thus has to be a positive exception in S^*. Hence we have a bound of l_p on the number of points $x \in S$ such that $h(x) = 0$.

Altogether, we can bound the cardinality of S by $|S| \leqslant ml + (n + 1)l^2 + l_p \leqslant (m + 1)l + (n + 1)l^2$. While h stays the same, the number of possible different sets

S is at most $(m+1)l + (n+1)l^2 + 1$.

Hence, the total number of equivalence queries in a complete run of the algorithm is bounded by $(2(m + (n+1)l) + 1) \times ((m+1)l + (n+1)l^2 + 1) = O(m^2n^2l^3)$. ■

We now count the total number of membership queries. Membership queries are made only in REDUCE, at most $n(n+1)/2$ per call to REDUCE. Claim 17 bounds the number of different points that REDUCE can return by $m + (n+1)l$. By Claim 16, the number of calls to REDUCE is bounded by the number of different points that it can return. Therefore, the total number of membership queries is bounded by $O(mn^2 + n^3l)$.

It is not difficult to see that the total running time of the algorithm is polynomial in n, m and l. This concludes the proof of Theorem 9. ■

Chapter 13

Exceptions and Errors

In this chapter, we exhibit a relation between learning concepts with exceptions and learning with malicious membership queries. We give a generic algorithm transformation. This transformation shows that any class of concepts that is polynomially closed under finite exceptions and learnable in polynomial time with equivalence and standard membership queries is also learnable in polynomial time using equivalence and malicious membership queries.

Theorem 10 *Let H be a class of concepts that is polynomially closed under finite exceptions and learnable in polynomial time with equivalence and standard membership queries. Then H is learnable in polynomial time with equivalence and malicious membership queries.*

Proof: Let $H = (R, Dom, \mu)$ be a target class of concepts that is polynomially closed under finite exceptions. We assume that LEARN is an algorithm to learn H using equivalence (EQ) and standard membership queries (MQ) in time $p_A(s, n)$, for some polynomial p_A. Without loss of generality, p_A is non-decreasing in both

arguments. We transform this algorithm into algorithm LearnwithMMQ, which learns any concept $h^* \in H$ using equivalence and malicious membership queries in time polynomial in $|h^*|$, n and the table-size L of the set of strings on which MMQ may lie.

As in Chapters 11 and 12 the main idea is to keep track of all the counterexamples seen and to use them to avoid unnecessary membership queries. For this purpose we use a set *CounterExamples* again. As before it stores pairs of counterexamples and their labels. Now, before asking a membership query about string x, we scan *CounterExamples* to see whether it already contains x and a label for it. If x and the label are found, the algorithm knows the answer and does not make the query. (For some concept classes, such as monotone DNF formulas, it might be possible to infer the classification of x according to the target concept h^* even though x and its label are not contained in *CounterExamples*. However, this simple checking suffices for our algorithm and, what is more important, works in the general case.)

Another idea is to keep track of the answers received from membership queries, and to use them to conclude that MMQ has lied. For this purpose LearnwithMMQ has a set *MembershipAnswers*. This set stores pairs $\langle x, b\rangle$ for which MMQ was called on string x and returned answer b. After receiving a new counterexample from EQ, the algorithm stores it in *CounterExamples* and checks whether this counterexample is already contained in *MembershipAnswers*. If it is present in *MembershipAnswers* with the wrong label, the algorithm discards everything except the set *CounterExamples* and starts from scratch. If this is not the case, the algorithm continues the simulation of Learn, which we now describe in detail.

The new algorithm simulates Learn on the target concept, but modifies Learn's queries as follows:

```
NewMQ(x, CounterExamples, MembershipAnswers)
{
    If (⟨x, b⟩ ∈ CounterExamples)
        Return b;

    b = MMQ(x);
    Add ⟨x, b⟩ to MembershipAnswers;
    Return b;
}
```

Figure 13.1 Subroutine NewMQ

- Each membership query of Learn, MQ(x), is replaced by a subroutine call NewMQ(x, *CounterExamples*, *MembershipAnswers*). The subroutine is given in Figure 13.1.

- Each equivalence query of Learn, $x = \mathrm{EQ}(h)$, as well as the output statement, **Output** h, is replaced by the block of code given in Figure 13.2.

Note that when the simulation is restarted, only the set *CounterExamples* reflects any work done so far. We now show that LearnwithMMQ is correct and runs in time polynomial in $|h^*|$, n, and L. We partition the run of the algorithm into *stages*, where a stage begins with a new simulation of Learn. First we show that a stage cannot last forever.

Claim 19 *Every stage ends in time polynomial in $|h^*|$, n, and L.*

Proof: Note that H is polynomially closed under finite exceptions, which means that there is a polynomial $p(\cdot,\cdot)$ such that for every concept $h \in H$ and every finite set $S \subseteq \mathit{Dom}(h)$ there exists a concept $h' \in H$ equal to $\mathit{xcpt}(h, S)$ such that size $|h'| \leqslant p(|h|, \|S\|)$. Without loss of generality we can assume that p is non-decreasing in both arguments. We now prove that each stage ends in time bounded

```
{
    x = EQ(h);

    If (x == "yes")
    {
        Output h;
        Return;
    }

    Add ⟨x, (1 − h(x))⟩ to CounterExamples;

    If (⟨x, h(x)⟩ ∈ MembershipAnswers)
    {
        MembershipAnswers = ∅;
        Restart Simulation, retaining CounterExamples;
    }

}
```

Figure 13.2 The block of code replacing "$x = \mathrm{EQ}(h)$" or "**Output** h"

by $p_A(p(|h^*|, L), n)$, where we count only the time spent on LEARN operations (i.e., we do not count the simulation and bookkeeping overhead).

We prove this by contradiction. Assume that stage i goes over the limit. Let us look at the situation right after the number of simulated steps of LEARN exceeds our stated time bound. Let S_i denote the set of strings the MMQ has lied about during this stage, up to the time bound. Let n denote the length of the longest counterexample received during this stage, up to the time bound.

None of the strings in S_i can belong to *CounterExamples*. Assume by way of contradiction otherwise. Let $x \in S_i$ be a string contained in *CounterExamples* with some label. Set S_i contains exactly the strings that the MMQ lied on in this stage and time bound, so there was a query MMQ(x). It must have happened before x was added to *CounterExamples*. But then at the moment x was added to *CounterExamples* it already belonged to *MembershipAnswers* and an inconsistency had to be found. The

stage had to end.

Therefore, considering S_i as an exception set, all the information received by LEARN in this stage and within the given time bound is consistent with the concept $h' = xcpt(h^*, S_i) \in H$. LEARN either has to output h' in time bounded by $p_L(p(|h^*|, \|S_i\|), n) \leqslant p_L(p(|h^*|, L), n)$, or it has to receive a counterexample $x \in S_i$. In the former case, LEARNWITHMMQ makes an equivalence query $\mathrm{EQ}(h')$ and receives a counterexample $x \in S_i$, since only counterexamples from S_i are possible at that point. In either case, an element of S_i is added to *CounterExamples* by the above time bound, which we showed above was impossible. This is a contradiction to the assumption that stage i goes over this bound. ■

What remains is to show that there can be only a polynomial number of stages. That is, we do not restart the simulation too many times.

Claim 20 *There are at most $L + 1$ stages in the run of the algorithm* LEARNWITHMMQ.

Proof: At the beginning of each stage (except the first one) the algorithm discovers a new string where the MMQ lies and from then on MMQ can never lie on this string again, because it is added to *CounterExamples*. To be more precise, MMQ does not get a chance to lie on this string because it is never asked about it again. Let S be the set of the strings that MMQ lies on. Since $|S| \leqslant \|S\| \leqslant L$, in stage $L + 1$ the MMQ can lie on no strings (i.e., it is not asked queries about any of the strings where it may lie). Therefore LEARN has to converge to the target concept h^*. ■

The time spent on simulation and bookkeeping is clearly polynomial in $|h^*|$, n, and

L. Thus, LEARNWITHMMQ is a polynomial-time algorithm that uses equivalence and malicious membership queries to learn the class of concepts $H = (R, Dom, \mu)$. This concludes the proof of Theorem 10. ■

As corollaries of Theorem 10 we have the following.

Corollary 14 *The class of regular languages, represented by DFA's, is learnable in polynomial time with equivalence and malicious membership queries.*

Proof: Board and Pitt [16] have shown that this class of concepts is polynomially closed under finite exceptions. Angluin [3] has shown that it is learnable in polynomial time using membership and equivalence queries. ■

Corollary 15 *The class of boolean decision trees is learnable in polynomial time with extended equivalence and malicious membership queries.*

Proof: Lemma 11 shows that the class of boolean decision trees is polynomially closed under finite exceptions. Bshouty [17] has shown that it is learnable in polynomial time using membership and extended equivalence queries. ■

Corollary 16 *The class of monotone DNF formulas with finite exceptions is learnable in polynomial time with equivalence and malicious membership queries.*

Proof: Corollary 13 shows that the class of monotone DNF formulas with exceptions is polynomially closed under finite exceptions. In Chapter 12 we gave an algorithm that learns this class in polynomial time with membership and equivalence queries. ■

Note that we can also learn the class of monotone DNF formulas without any exceptions with this generic algorithm, using extended equivalence and malicious membership queries, since it is just a subclass of the class that allows exceptions. However, the algorithm is much less efficient than the one described in Chapter 11.

Chapter 14

Discussion and Open Problems

In Chapter 11 we showed how to learn monotone DNF formulas in polynomial time from equivalence and malicious membership queries. It would also be nice to give a lower bound result for this problem. So far we can only prove the trivial fact that there must be more membership queries than lies (otherwise membership queries do not reveal any information and equivalence queries alone do not suffice [5]).

There are many other classes of concepts besides monotone DNF formulas that do not have any lower bound results in the model of equivalence and malicious membership queries. Sloan and Turán [36] have proved such lower bounds in their model of equivalence and limited membership queries for the class of monotone monomials and for another specially constructed class. These bounds apply to the model of equivalence and malicious membership queries as well. These, however, are the only two lower bound results known for these models.

Another question regarding lower bounds for the model of equivalence and malicious membership queries is whether this model is harder than or as hard as the

model of equivalence and limited membership queries, from the point of view of polynomial-time learnability. Sloan and Turán exhibited a concept class that requires so many more examples to learn from equivalence and malicious membership queries than it takes to learn from equivalence and limited membership queries that no polynomial can bound the former number in terms of the latter. However, in both models the number of queries required is exponential in the number of lies and therefore does not answer the question whether one model is harder with respect to polynomial-time learnability.

In Chapter 12 we gave a polynomial-time algorithm using equivalence and (error-free) membership queries to learn the class of monotone DNF formulas with exceptions. Among the open problems regarding learning with exceptions are finding polynomial-time algorithms for other classes of concepts and proving lower bounds for any of the classes. Also, it may be possible to improve the running time of the algorithm given in Chapter 12.

In Chapter 13 we showed that there is a polynomial-time algorithm using equivalence and malicious membership queries for learning any concept class that is polynomially closed under finite exceptions and can be learned in polynomial time using equivalence and (error-free) membership queries. Thus, learning with exceptions is not easier than learning with lies if polynomial-time identification is the only goal and actual running times are not considered. An immediate question is whether it really is harder than learning with lies or they both are equally hard. In other words, is there a class that is polynomially closed under finite exceptions, is learnable with malicious membership queries in polynomial time, and is not polynomial-time learnable with exceptions?

The generic method of Chapter 13 allows us to learn new classes with equiva-

lence and malicious membership queries. These include DFA's and boolean decision trees. However, this result leaves the question open for other classes, polynomial-time learnable with equivalence and (error-free) membership queries, such as read-once formulas, that are not polynomially closed under finite exceptions. A start in this direction is made by Angluin [6], who gives a randomized polynomial-time algorithm to learn read-once DNF formulas with equivalence and malicious membership queries.

Part III

Learning with Random Errors in Queries

Chapter 15

Introduction and Definitions

In this part of the thesis we continue to explore the effect of errors on the learning model that uses equivalence and membership queries. In this model, introduced by Angluin [3], a learner tries to exactly identify an unknown target concept belonging to a known class of possible concepts. While being an appealing and well-studied model [1, 2, 3, 7, 32], it is an error-free model, and thus more susceptible to failures in practical applications. Several attempts have been made to augment this model with various kinds of errors; see the discussion in Section 9.2 of Part 2 for more details. Typically, when adding errors to the model, the equivalence queries remain error-free, while membership queries are allowed to be a (limited) source of errors or omissions [8, 12, 13, 24, 36].

Obviously, errors are harder to correct for than omissions which point out that reliable classification of certain examples is not available. This part of the thesis considers only errors in answers to membership queries. An important issue is whether these errors are *persistent* or *non-persistent*. The former kind is harder to overcome, as repeated queries do not reveal any more information. Another issue is how the

errors are distributed. Two popular approaches are to treat them as either *malicious* or as *random*. Malicious errors are introduced by an online adversary and were the focus of Part 2. In this part we look at random errors in membership queries. Random errors are distributed uniformly at random among all possible examples according to a certain error rate (or fraction) $p < 1/2$.

When considering models that involve randomness it may happen that the learning algorithm cannot exactly identify the target concept. For example, in the PAC model it is possible that the distribution of the examples seen is significantly different from the real distribution. In models that allow errors in queries it may happen that the error rate that the algorithm has to cope with is much higher than it is supposed to be. It is even possible (although unlikely in all realistic models) that all the queries are answered incorrectly. For these reasons, the goal of the learning algorithm cannot be exact identification of the target concept under all circumstances. Instead, algorithms are required to identify the target concept with *high probability*, i.e., with probability at least $1-\delta$, for any $\delta > 0$. Algorithms are considered polynomial-time if their running times can be bounded by a polynomial in the size of the target concept, the size of the longest counterexample seen and in $1/\delta$. Algorithms that have to overcome errors (or omissions) occurring with rate $p < 1/2$ are in addition allowed to have their running times depend polynomially on $\frac{1}{1/2-p}$, i.e., on the inverse of how far the error rate is from a fair coin toss.

Malicious errors can be expected to be harder to overcome than random errors, and this is often seen from the fraction of errors that polynomial-time algorithms for each model can sustain. For example, the algorithm LearnMonDNF from Section 11.1 in Part 2 can overcome very few errors if its running time may not depend on the actual number of errors received, as was the case in the model of

equivalence and malicious membership queries. Each error it receives may result in as many as $\Omega(n^2)$ extra membership queries done, each of which may contain another error. Thus, saying that it could perhaps cope with an error rate of about $1/n^2$ is probably a very optimistic claim.

Monotone DNF formulas are a promising target concept class for models allowing errors in membership queries, since the monotonicity may be useful in error detection and correction. The simplicity of the algorithm LearnMonDNF given in Section 11.1 of Part 2 suggests that with minor modifications it may be applicable to the model of random errors as well. In this part of the thesis we randomize the algorithm in a straightforward way and perform a thorough analysis of its probability of success. After obtaining both upper and lower bounds on the probability of success, we compare it to a randomized version of the algorithm that learns monotone DNF formulas from equivalence and (error-free) membership queries [4], illustrated in figure 10.1 in Part 2. This algorithm is very simple and efficient but depends on all the answers being correct. It appears that the more complex algorithm is in general more promising than the simple one, although its advantages are best seen for a very limited range of the parameters representing error rate and term size.

Many of the relevant definitions and notation have already been presented in Chapter 10 of Part 2. The most important concepts as well as notation and terminology unique to this part are briefly explained below.

Boolean formulas are defined in terms of variables, each of which may be assigned a value that is *true* or *false*. It is very common to denote the assignment of these values to each of the variables by a string in $\{0,1\}^n$, also called boolean n-vector, where n is the number of variables in the formula. By convention, 1 stands for *true* and 0 for *false*. This applies also to the values of formulas on particular assignments;

formulas are viewed as functions from the domain of all boolean n-vectors and having a range of $\{0, 1\}$.

The individual digits in boolean n-vectors are called *bits* throughout this part of the thesis. The *sample space* of the target formula consists of all the possible assignments that can be applied to its n variables, i.e., of all the 2^n boolean n-vectors. We view this sample space as a lattice and consider each of the possible assignments to variables as a *point* in this lattice. The top element is the vector of all 1's and the bottom element is the vector of all 0's. Componentwise "or" and "and" are the lattice operations. The partial order in the lattice is tied to the representations of points by boolean n-vectors. A point x is *above* point y if and only if each bit of x is not smaller than the corresponding bit of y and there is at least one bit that is strictly greater. More formally, $x[i] \geqslant y[i]$ for all $1 \leqslant i \leqslant n$ and there is an $1 \leqslant i \leqslant n$ such that $x[i] > y[i]$. In this case we may also say that point y is *below* point x. All the points that are above point x are called the *proper ancestors* of x, while those that are below x are called its *proper descendants*. Proper ancestors of x together with point x itself form the *ancestors* of x. Similarly for the *descendants*. Proper ancestors of x that differ from it only in the value of one bit are called the *parents* of x. Proper descendants of x that differ only in the value of one bit are called its *children*.

In this part of the thesis we analyze the performance of algorithms on the target concept class of *monotone monomials*. These are monotone DNF formulas that contain exactly one term. Each monotone monomial can be represented conveniently by its *minimum point*. This is the lowest point in the lattice such that its corresponding assignment makes the monomial true. In other words, the value of the monomial is 1 (or true) on this minimum point but false on all its children. Due to the mono-

tonicity, it is 1 only on all the ancestors of the minimum point. The points on which the value of the monomial is 1 are called its *positive* points. All other points in the lattice are called *negative*.

More generally, each monotone DNF formula can be described in this way by a set of its *local minimum points*, each of which corresponds to one term of the formula (assuming that it has been minimized). A natural way to identify monotone DNF formulas (at least in the error free model of equivalence and membership queries) is to start with a conjecture of the *empty DNF formula* which contains no terms and is therefore the identically false formula. Unless this trivial hypothesis is correct, a positive point will be returned as a counterexample, which, due to the monotonicity, is a point that is above one of the local minimum points of the formula. All that remains is to work one's way down the lattice using membership queries to inquire about the truth value of each point until a local minimum point is reached. This is exactly the approach taken by the algorithm LEARNSTANDARD given by Angluin [4] and illustrated in Figure 10.1 of Part 2. Each new term found is an increment of progress towards the identification of the monotone DNF formula. In case the target formula is a monotone monomial only one term needs to be found.

Clearly, when we have a positive point x that is above a minimum point y (which is also positive) and we wish to traverse the lattice down to the minimum point, there are certain bits of x that have values 1 and need to be set to 0 in order to reach y. Such bits are called the *unwanted 1-bits*. There may be other bits that have values 1 in both x and y. These bits are called the *needed 1-bits* and may not be set to 0 while moving down the lattice (or else we will miss point y and find ourselves on a negative point).

Our model of random persistent errors in the membership queries can be described

as follows. Let the target function be denoted by f. A parameter p defines the probability of an error in each individual membership query, independently of others. This is also called the *error rate*. For each point x in the lattice, if a membership query is asked about it and has not been asked before, the answer corresponds to $f(x)$ (and is not an error) with probability $1-p$ and it is $1-f(x)$ (an error) with probability p. Subsequent queries about point x receive the same answer. Since the errors are introduced independently of each other, we may consider that their location is decided before the actual start of the learning algorithm. When membership queries are subject to this kind of errors, we often refer to them as the *random membership queries* and denote by RMQ.

In the next chapter we define the randomized algorithm that we want to analyze and introduce an analogy between the algorithm and an imagined "game", thus providing a somewhat more entertaining and less technical way to discuss the algorithm. After that, in Chapter 17 we derive the exact formula for the probability of success of the algorithm and give some sample values according to the formula. Subsequently, in Chapter 18 we derive the lower and upper bounds on the formula and finally, in Chapter 19 we compare the algorithm to a simpler one.

Chapter 16

The Algorithm and the Game

We would like to find out how well the algorithm LearnMonDNF from Section 11.1 of Part 2 performs in the setting of random and persistent errors. To simplify the analysis, we focus on learning monotone monomials. We also restrict our attention to only the first call of its subroutine Reduce. That is, we are interested in whether Reduce will return the correct monomial or not. To be more precise, we also consider it acceptable for Reduce to return a point above the monomial, that is, to classify a subset of the positive points of the monomial as being positive. This is not a serious error, since a subsequent equivalence query in such case would have to present another positive counterexample, thus giving an opportunity to improve the former result. As we will see in Chapter 18, if a point above the target monomial is returned, then it is very likely to be close to the monomial, requiring few additional counterexamples. We do not, however, want Reduce to return a negative point of the monomial, as such would have to be discarded eventually. This would result in a very inefficient way of finding the correct point since there seems to be no useful bound on the number of such negative points that can be returned.

```
RandomReduce(v)
{
    While (unmarked child w of v exists)
    {
        Uniformly select a random unmarked child w;

        If (RMQ(w) == 1)
            Return RandomReduce(w);
        Else
            Mark w;

    }

    Return v;
}
```

Figure 16.1 Algorithm RandomReduce

Since we only look at the first call of Reduce, there are no useful counterexamples to consider and thus the other subroutine CheckedMQ can be ignored (i.e., substituted by a call to MMQ in the model of malicious errors). The algorithm basically consists of getting a positive point v from EQ and calling Reduce on it. Note that Reduce has not been completely specified yet. In particular, there is freedom in determining the order in which to examine the children of v. Since the errors are now assumed to happen randomly, it seems natural and helps in analysis to ask that the children of v are selected randomly (excluding the ones already queried, if any). Thus, the interesting part of the algorithm LearnMonDNF in the setting of random persistent errors can be described by the randomized algorithm RandomReduce given in Figure 16.1.

Algorithm RandomReduce is started on a positive point v and it tries to move down recursively from this point until it finds a local minimum point—one that is positive but has only negative-valued children. Marking the children of v helps it avoid repeated queries of the same point. If all the children have been queried and

no positive ones found, the algorithm terminates returning its current point as its hypothesis for the monotone monomial. In order to query the children of v it uses the random membership query, denoted by RMQ, which may make random errors at a certain rate. Thus there is a chance of the algorithm stopping above the actual monomial and there is also a chance of it making a recursive call on a negative point—a mistake that will inevitably lead to returning a negative point as the hypothesis monomial. We now analyze the probabilities associated with the possible outcomes of the call to RandomReduce.

We assume that RandomReduce is started on a positive point of the target monomial. We are interested in the probability of it returning this monomial. We call a *success* the event where RandomReduce returns any point above the target term. We call a *complete success* the event where RandomReduce returns exactly the target monomial. If it returns a point not above the target, we call it a *failure*. In order to calculate the probabilities corresponding to these events, we need another description of how RandomReduce works. Let the point v on which RandomReduce is started be encoded by a string containing T needed 1-bits and F unwanted 1-bits. Obviously, all the unwanted 1-bits have to be set to 0 in order to achieve complete success. In the following description of RandomReduce we focus on the bits that the boolean n-vectors (or points in the lattice) consist of. We mark and unmark individual bits, not the points themselves.

RandomReduce is initially invoked on a point containing some mix of 0-bits and 1-bits. All 1-bits are considered *unmarked*. It sets to 0 a random unmarked 1-bit and asks RMQ whether the new string is a positive point of the target. If the answer is 1 (i.e., “yes”), RandomReduce calls itself on this new point, unmarking all marked 1-bits (if any) before that. If the answer is 0 (i.e., “no”), it flips the newly

created 0-bit back to 1, *marks* it and tries a different unmarked 1-bit. When all the unmarked 1-bits (if any) are tried, it terminates. It is easy to see that if any of the T needed 1-bits get set to 0 and the RMQ classifies the new string as a positive point, a failure is inevitable.

This description differs somewhat from the pseudocode of RandomReduce, given in Figure 16.1. In the pseudocode points themselves become marked and unmarking is not necessary. In the new description, however, we depend on points being boolean n-vectors and we mark the position (or index, coordinate) where the current point and its child being queried differ, as opposed to marking the child itself. This way we can keep track of the progress of the algorithm by storing only a string of n characters, where each character stands for a 0-bit, a marked 1-bit or an unmarked 1-bit. If we wanted to mark points themselves, as given in the pseudocode, we would need to store all the 2^n points plus a flag for each point describing whether the point is marked.

We now draw parallels between the behavior of the algorithm RandomReduce and the following "Bottle Shooting Game." The game proceeds through *states*, each characterized by the number and kind of bottles in it. There are two kinds of bottles—the *fat* ones and the *thin* ones. The fat bottles are called fat because they are easier to hit than the thin ones. They correspond to the unwanted 1-bits in the current point that RandomReduce is being called (or is calling itself) on. The thin bottles correspond to the needed 1-bits in the current point. Thus, initially there are F fat bottles and T thin bottles. During the game, some bottles, either fat or thin, may get marked (as well as unmarked). Due to this there are really four possible kinds of bottles in a state: "fat unmarked", "fat marked", "thin unmarked" and "thin marked". In the beginning no bottles are marked.

The game consists of repeatedly choosing an unmarked bottle, shooting at it and moving to a different state depending on whether the bottle was hit or missed. Shooting at a bottle corresponds to querying a child of the current point in RandomReduce. The marked bottles are "protected", i.e., ineligible for shooting at, which corresponds to a certain child of the current point having been queried already. Hitting a bottle in the game corresponds to RMQ returning answer 1 on the queried child of the current point. This would lead to a recursive call of RandomReduce and in the game this leads to unmarking all the bottles (making them eligible for shooting at). Missing a bottle corresponds to RMQ returning 0, which leads to the child being marked and another unmarked child of the current point being selected. In the game this leads to marking the bottle that was missed, i.e., protecting it temporarily. Here it is worth pointing out that when shooting at a particular bottle it is impossible to hit a different one (imagine, for example, that they are sufficiently far apart from each other). Also note that if a bottle is hit, it becomes a pile of broken glass that is thrown away at that very moment.

The difficulty of hitting a fat bottle and that of hitting a thin one is not the same. As mentioned above, fat bottles correspond to the unwanted 1-bits in the current point and a query of a child which differs from the current string in an unwanted 1-bit should return 1, if there is no error. This happens with probability $1-p$ in the algorithm and therefore the probability of hitting a fat bottle is set to $1-p$ in the game. Naturally, the probability of missing a fat bottle is p, which corresponds to the probability of RMQ making an error on the queried child. Thin bottles correspond to the needed 1-bits and a query of a child differing from the current point in a needed 1-bit should return 0, if no error occurs. Therefore, the probability of missing a thin bottle is set to $1-p$ in the game and the probability of hitting it is p. Since $p < 1/2$,

it happens that fat bottles are easier to hit than the thin ones and thin are easier to miss than the fat ones.

Hitting a thin bottle actually terminates the game after changing its state to a special *Failure* state, since this corresponds to calling RandomReduce recursively on a negative point, due to an unfortunate error in the RMQ. The game also ends if there are no more bottles eligible for shooting at. In this latter case the game ends in one of the *Success* states. These states have T marked thin bottles in them and no unmarked fat bottles. The Success state that has no fat bottles at all is special, for it leads to yet another state, the *Complete Success* state. In order to simplify references to the beginning of the game, we introduce a special *Start* state. This is the state that the game starts at and from this state it immediately moves to the state with T thin and F fat bottles, all of which are unmarked. It is impossible to move to any other state than this from the Start state.

Since the total number of thin bottles can never change in the game (excluding Start, Failure and Success states), we do not consider it as a useful parameter of each state. Instead, we treat it as a parameter of the game. Thus, the state with f fat bottles, of which f' are unmarked, and T thin bottles, of which t' are unmarked, is denoted by $S_T(f, t', f')$. The rules of the game are schematically presented in Figure 16.2, where arrows denote the possible transitions between the states and the numbers on arrows denote the probabilities associated with these transitions. The Failure state and the transitions leading to it are omitted to save space—for every state shown the probabilities on its outgoing arrows should add to 1; if they do not, the remaining probability is for the omitted transition to the Failure state.

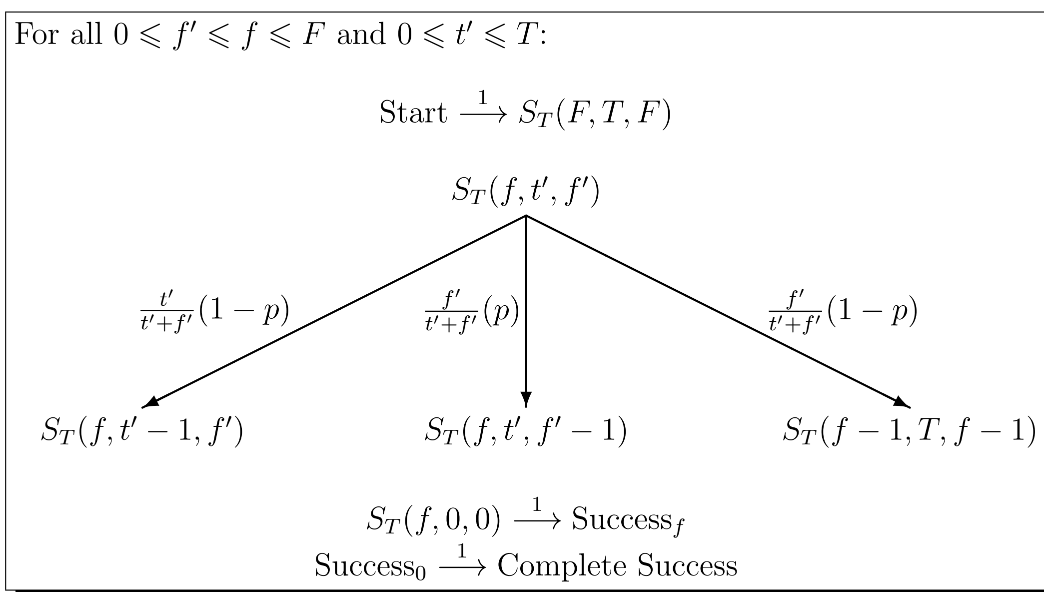

Figure 16.2 States, transitions and their probabilities in the game

Chapter 17

Probabilities Associated with the Game Events

We are interested in the probability of getting from the Start state to any of the Success states, and especially to the Complete Success state. This, of course, corresponds to getting from state $S_T(F, T, F)$ to one of the states $S_T(f, 0, 0)$, where $0 \leqslant f \leqslant F$, and in the ideal case—to the state $S_T(0, 0, 0)$. It is possible to express these probabilities explicitly, that is, in terms of T, F, f and p. Before we do this, however, let us introduce some new notation and a convenient way to analyze the game.

Let us imagine that the states of the game are positioned on a three-dimensional grid according to their three parameters. The first parameter, the total number of fat bottles, defines the *level* of a state. The game starts on level F, the top level. The game either ends on this level, meaning that it goes to the Failure state or to the Success_F state, or it eventually moves down to level $F - 1$. From there it can end again or move down one more level. Level 0 is the bottom level and its states

have no fat bottles in them. Obviously, moving down one level is achieved by hitting a fat bottle which is possible from most states of each level above level 0.

Within each level, the states are positioned from left to right according to the second parameter, the number of unmarked thin bottles. States with more unmarked thin bottles are to the left of those with less. They are also positioned closer or farther according to the third parameter, the number of unmarked fat bottles. States with more unmarked fat bottles are farther away than those with less.

There are also the special states in our game: Start, Failure, Complete Success and Success_f, for all $0 \leqslant f \leqslant F$. Let us imagine that these states are kept away from our three-dimensional grid where the other states are positioned. It is, however, convenient to imagine that Success_f state "belongs" to level f, for all $0 \leqslant f \leqslant F$. That is, they are away from the grid but stacked on each other at the appropriate heights. Thus, Success_f is often referred to as the *level f Success state* and the event of reaching it is called *success on level f*. Figure 17.1 shows all the states of some level f.

Now let us take an arbitrary state $S_T(f, t', f')$, where $t' \neq 0$ or $f' \neq 0$, and analyze the possible transitions and their respective probabilities from this state (if both t' and f' are 0, then the only transition possible is to Success_f state):

1. Suppose a fat bottle is chosen and hit. This happens with probability $\frac{f'}{t'+f'}(1-p)$ and takes us to state $S_T(f-1, T, f-1)$, which is the farthest leftmost state of the next (one down) level.

2. Now suppose that a fat bottle is chosen and missed. This happens with probability $\frac{f'}{t'+f'}(p)$ and takes us to state $S_T(f, t', f'-1)$, which is on the same level and one grid line closer than the original state.

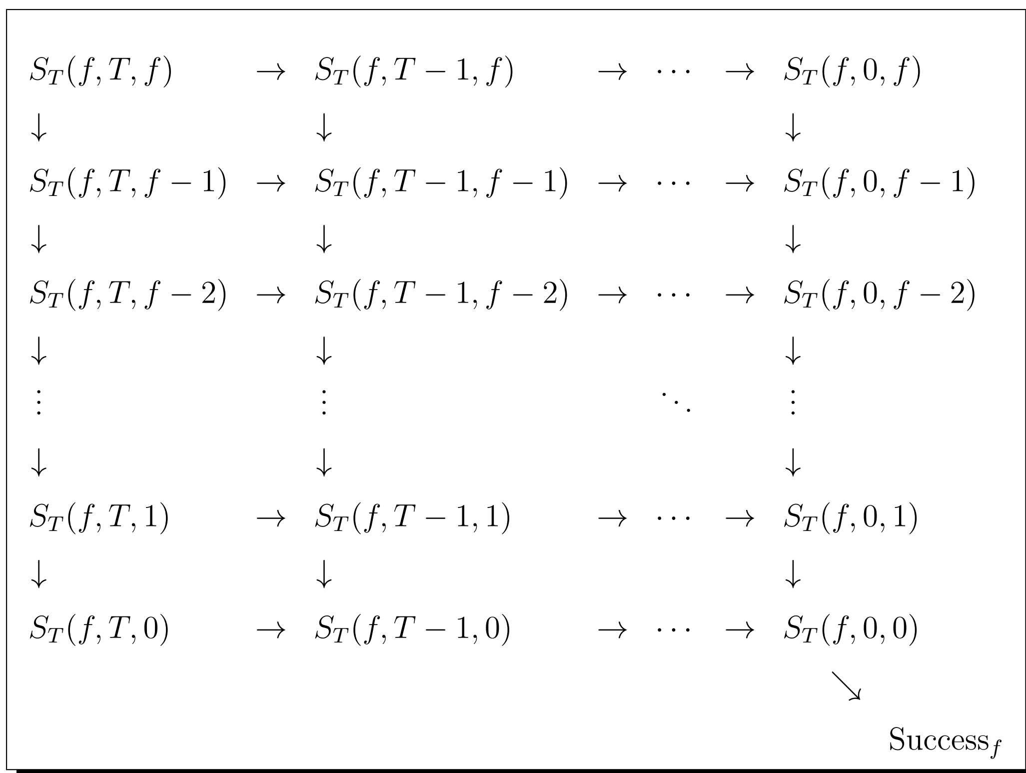

Figure 17.1 The states of level f and transitions between them

3. It is also possible to choose a thin bottle and miss. The probability of this event is $\frac{t'}{t'+f'}(1-p)$ and the next state in such case is $S_T(f, t'-1, f')$, which is on the same level and just to the right of the original state.

4. The final possibility is to choose a thin bottle and hit it. This event has probability $\frac{t'}{t'+f'}(p)$ and it leads to the Failure state.

It can be seen from the above description that the game proceeds as follows. It starts at the farthest leftmost state of the top level. It possibly moves to some other state on the same level, always moving only right or closer and never leaving the level. The arrows in Figure 17.1 show how the game can change states within the same level. If it manages to reach the closest rightmost state of the level, then it goes straight to the Success_F state and stops there. Otherwise, at some point it

either goes to the Failure state (i.e., "fails") or moves to the farthest leftmost state of the next (one lower) level. From there the same behavior continues. The farthest leftmost state of each level is called its *entry state*—moving to a lower level implies moving to the entry state of that level.

For example, if there were 5 fat and 3 thin bottles originally, the game could proceed as follows. It would start in state $S_3(5,3,5)$, the entry state of level 3, the top level. From there it could go to state $S_3(5,2,5)$ due to a thin bottle chosen and missed. From this state it could proceed to state $S_3(5,2,4)$ due to a fat bottle chosen and missed. After that the game could move to state $S_3(4,3,4)$ of the level below, if a fat bottle is chosen and hit. This is the entry state of level 4. After that the game could proceed through states $S_3(4,3,3)$, $S_3(4,2,3)$, $S_3(4,1,3)$ before moving to the entry state of level 3, which is state $S_3(3,3,3)$. These transitions correspond to a fat bottle being missed, two thin bottles in a row being missed and then a fat bottle being hit. After that the game could move through the states $S_3(3,3,2)$ and $S_3(3,3,1)$ and then suddenly end in the Failure state. These transitions correspond to two fat bottles being missed in a row and then a thin bottle being hit.

Returning to the analysis of the game in general, let us denote by $\mathbf{P}\{S \rightsquigarrow S'\}$ the probability of reaching state S' from state S, possibly via other states. Let us denote by $\mathbf{P}\{S \to S'\}$ the probability of reaching state S' from state S directly; this is possible only if this transition has a nonzero probability in the game. According to this notation, we are interested in finding

$$\sum_{f=0}^{F} \mathbf{P}\{\text{Start} \rightsquigarrow \text{Success}_f\} = \sum_{f=0}^{F} \mathbf{P}\{S_T(F,T,F) \rightsquigarrow S_T(f,0,0)\}.$$

We are also interested in finding

$$\mathbf{P}\{\text{Start}\rightsquigarrow\text{Complete Success}\} = \mathbf{P}\{S_T(F,T,F)\rightsquigarrow S_T(0,0,0)\}.$$

Clearly, it would be very helpful to express

$$\mathbf{P}\{S_T(F,T,F)\rightsquigarrow S_T(f,0,0)\} = \mathbf{P}\{\text{Start}\rightsquigarrow\text{Success}_f\}$$

in terms of T, F, f and p, that is, the probability of starting at the Start state and reaching the level f Success state. The following theorem accomplishes this goal.

Theorem 11 *For every* $0 \leqslant f \leqslant F$*, the probability* $\mathbf{P}\{S_T(F,T,F)\rightsquigarrow S_T(f,0,0)\}$ *can be expressed as*

$$(1-p)^{T+F-f}p^f \prod_{g=f+1}^{F} \frac{\sum_{t=0}^{T}\sum_{h=1}^{g}\binom{T-t+g-h}{T-t}\binom{t+h-1}{t}(1-p)^{T-t}p^{g-h}}{\binom{T+g}{T}}.$$

Proof: Reaching state $S_T(f,0,0)$ from state $S_T(F,T,F)$ involves first reaching the entry state of level f and only then moving to state $S_T(f,0,0)$. Therefore, we may write

$$\mathbf{P}\{S_T(F,T,F)\rightsquigarrow S_T(f,0,0)\} \\ = \mathbf{P}\{S_T(F,T,F)\rightsquigarrow S_T(f,T,f)\} \cdot \mathbf{P}\{S_T(f,T,f)\rightsquigarrow S_T(f,0,0)\}.$$

Of course, reaching a particular level f from level F involves going through all the levels in between, therefore

$$\mathbf{P}\{S_T(F,T,F)\rightsquigarrow S_T(f,T,f)\} = \prod_{g=f+1}^{F} \mathbf{P}\{S_T(g,T,g)\rightsquigarrow S_T(g-1,T,g-1)\}.$$

There are many different ways to get from the entry state of level g, $S_T(g,T,g)$, to the entry state of level $g-1$, $S_T(g-1,T,g-1)$. That is, on level g the game can move to some arbitrary state $S_T(g,t,h)$, where $t \leqslant T$ and $h \leqslant g$, and from there go directly to $S_T(g-1,T,g-1)$, as long as $h > 0$. Therefore,

$$\begin{aligned}&\mathbf{P}\{S_T(g,T,g) \rightsquigarrow S_T(g-1,T,g-1)\}\\&\qquad = \sum_{t=0}^{T}\sum_{h=1}^{g} \mathbf{P}\{S_T(g,T,g) \rightsquigarrow S_T(g,t,h)\} \cdot \mathbf{P}\{S_T(g,t,h) \rightarrow S_T(g-1,T,g-1)\}.\end{aligned}$$

The following lemma is the most important component in the proof of Theorem 11.

Lemma 15 *The probability of reaching state $S_T(g,t,h)$ from state $S_T(g,T,g)$, where $t \leqslant T$ and $h \leqslant g$, is given by*

$$\mathbf{P}\{S_T(g,T,g) \rightsquigarrow S_T(g,t,h)\} = \frac{\binom{T-t+g-h}{T-t}\binom{t+h}{t}}{\binom{T+g}{T}}(1-p)^{T-t}p^{g-h}.$$

Proof: In order to reach state $S_T(g,t,h)$ from state $S_T(g,T,g)$, the game must do $T-t+g-h$ transitions from state to state. Furthermore, $T-t$ of these transitions must go one state to the right on the same left-right grid line, and $g-h$ of them must move one state closer, staying on the same farther-closer grid line. There are two possible directions for movement and each sequence of $T-t+g-h$ transitions of which $T-t$ transitions move to the right and $g-h$ transitions move closer specifies a distinct way of getting from $S_T(g,T,g)$ to $S_T(g,t,h)$. Thus, there are $\binom{T-t+g-h}{T-t}$ such ways altogether. Not surprisingly, the probability that the game proceeds in any one of these ways is exactly the same. This can be best seen as follows.

Suppose the game is in state $S_T(g,t',f')$ at the moment. The probability of

moving right is then $\frac{t'(1-p)}{t'+f'}$ and it leads to state $S_T(g, t'-1, f')$. The probability of moving closer is $\frac{f'p}{t'+f'}$ and leads to state $S_T(g, t', f'-1)$. Suppose we fix some way of getting from $S_T(g, T, g)$ to $S_T(g, t, h)$ and write down the probabilities of all the transitions in the same order as they happen. Regardless of the direction of movement, the total number of unmarked bottles decreases by 1 with each transition. Thus, the denominators of these probabilities form the sequence $T+g$, $T+g-1$, $T+g-2$, $\ldots$, $t+h+2$, $t+h+1$.

The numerators of these probabilities form a sequence which is a merge of two sequences: $T(1-p)$, $(T-1)(1-p)$, $(T-2)(1-p)$, $\ldots$, $(t+2)(1-p)$, $(t+1)(1-p)$ and gp, $(g-1)p$, $(g-2)p$, $\ldots$, $(h+2)p$, $(h+1)p$.This can be best seen as follows. The numerator of the probability for the first transition to the right is definitely $T(1-p)$, regardless of what the denominator is (which will depend on how many moves in the other direction have been performed already). The numerator of the probability for the next transition to the right is $(T-1)(1-p)$, again regardless of whether it immediately follows the first transition to the right or has transitions that move closer in between. So it continues, until all $T-t$ transitions to the right are done. Similarly, the probability for the first transition that moves closer has numerator gp, regardless of how many transitions to the right have been performed before it. The next one will have numerator $(g-1)p$ and so on. In general, the numerator of the probability for the i-th transition to the right is $(T-i+1)(1-p)$ and the numerator of the probability for the i-th transition that moves closer is $(g-i+1)p$. Altogether, no matter which way the game moves from $S_T(g, T, g)$ to $S_T(g, t, h)$, the numerators of the sequence of probabilities corresponding to these transitions will contain each of the elements $T(1-p)$, $(T-1)(1-p)$, $\ldots$, $(t+1)(1-p)$ and gp, $(g-1)p$, $\ldots$, $(h+1)p$ exactly once.

According to the above, the probability that the game will follow any particular way of getting from $S_T(g,T,g)$ to $S_T(g,t,h)$ is equal to

$$\frac{\left(\frac{T!}{t!}\right)(1-p)^{T-t}\left(\frac{g!}{h!}\right)p^{g-h}}{\left(\frac{(T+g)!}{(t+h)!}\right)} = \frac{T!g!}{(T+g)!}\frac{(t+h)!}{t!h!}(1-p)^{T-t}p^{g-h} = \frac{\binom{t+h}{t}}{\binom{T+g}{T}}(1-p)^{T-t}p^{g-h}.$$

Multiplying this by the number of possible ways to get from $S_T(g,T,g)$ to $S_T(g,t,h)$ gives the desired result. ■

Now we are ready to continue with the proof of Theorem 11. The probability of moving from some state $S_T(g,t,h)$ directly to the entry state of the level below is

$$\mathbf{P}\{S_T(g,t,h) \rightarrow S_T(g-1,T,g-1)\} = \frac{h}{t+h}(1-p).$$

Therefore, by Lemma 15, the probability to get to level $g-1$ when starting at the entry state of level g is

$$\begin{aligned}
&\mathbf{P}\{S_T(g,T,g) \rightsquigarrow S_T(g-1,T,g-1)\} \\
&\qquad = \sum_{t=0}^{T}\sum_{h=1}^{g} \frac{\binom{T-t+g-h}{T-t}\binom{t+h}{t}}{\binom{T+g}{T}}(1-p)^{T-t}p^{g-h} \cdot \frac{h}{t+h}(1-p) \\
&\qquad = \frac{(1-p)}{\binom{T+g}{T}} \sum_{t=0}^{T}\sum_{h=1}^{g} \binom{T-t+g-h}{T-t}\binom{t+h-1}{t}(1-p)^{T-t}p^{g-h}.
\end{aligned}$$

Using the above we can express the probability of reaching level f from the Start

state (or, equivalently, from the entry state of the (highest) level F):

$$\begin{aligned}
&\mathbf{P}\{S_T(F,T,F)\rightsquigarrow S_T(f,T,f)\} \\
&\qquad = \prod_{g=f+1}^{F} \frac{(1-p)}{\binom{T+g}{T}} \sum_{t=0}^{T} \sum_{h=1}^{g} \binom{T-t+g-h}{T-t} \binom{t+h-1}{t} (1-p)^{T-t} p^{g-h} \\
&\qquad = (1-p)^{F-f} \prod_{g=f+1}^{F} \frac{\sum_{t=0}^{T} \sum_{h=1}^{g} \binom{T-t+g-h}{T-t} \binom{t+h-1}{t} (1-p)^{T-t} p^{g-h}}{\binom{T+g}{T}}.
\end{aligned}$$

The probability of moving from state $S_T(f,T,f)$ to state $S_T(f,0,0)$, i.e., the probability of ending the game with a success in level f when starting at this very level, is

$$\mathbf{P}\{S_T(f,T,f)\rightsquigarrow S_T(f,0,0)\} = (1-p)^T p^f,$$

as can be easily seen either from Lemma 15 or by observing that no matter in what order the fat and thin bottles are chosen, each one of them has to be missed. Using this knowledge, we can express the probability of starting in the Start state and reaching the level f Success state, that is, state Success$_f$, which is reachable only from $S_T(f,0,0)$:

$$\begin{aligned}
&\mathbf{P}\{S_T(F,T,F)\rightsquigarrow S_T(f,0,0)\} \\
&\qquad = (1-p)^T (1-p)^{F-f} p^f \prod_{g=f+1}^{F} \frac{\sum_{t=0}^{T} \sum_{h=1}^{g} \binom{T-t+g-h}{T-t} \binom{t+h-1}{t} (1-p)^{T-t} p^{g-h}}{\binom{T+g}{T}}.
\end{aligned}$$

This completes the proof of Theorem 11. ■

Now we are ready to express the probabilities of success and complete success in the game, which are simple corollaries of Theorem 11.

Corollary 17 *The probability of success in the game is:*

$$(1-p)^T \sum_{f=0}^{F} p^f (1-p)^{F-f} \prod_{g=f+1}^{F} \frac{\sum_{t=0}^{T} \sum_{h=1}^{g} \binom{T-t+g-h}{T-t} \binom{h+t-1}{t} (1-p)^{T-t} p^{g-h}}{\binom{T+g}{T}}$$

Proof: As mentioned above, this is the same as $\sum_{f=0}^{F} \mathbf{P}\{\text{Start} \rightsquigarrow \text{Success}_f\}$, which equals $\sum_{f=0}^{F} \mathbf{P}\{S_T(F,T,F) \rightsquigarrow S_T(f,0,0)\}$. The result follows easily from Theorem 11.

■

Corollary 18 *The probability of complete success in the game is:*

$$(1-p)^{T+F} \prod_{g=1}^{F} \frac{\sum_{t=0}^{T} \sum_{h=1}^{g} \binom{T-t+g-h}{T-t} \binom{t+h-1}{t} (1-p)^{T-t} p^{g-h}}{\binom{T+g}{T}}$$

Proof: As mentioned above, this is the same as $\mathbf{P}\{S_T(F,T,F) \rightsquigarrow S_T(0,0,0)\}$. Setting $f = 0$ in the formula given by Theorem 11 completes the proof. ■

Unfortunately the formulas given in Corollaries 17 and 18 do not give a good insight about the asymptotics of the probabilities of success or complete success in the game. Tables 17.1, 17.2 and 17.3 illustrate the values of these probabilities for different T, F and p.

	$F=0$	$F=20$	$F=40$	$F=60$	$F=80$	$F=100$
$T=0$	1.000000	0.998999	0.998999	0.998999	0.998999	0.998999
$T=100$	0.904792	0.694685	0.650498	0.625407	0.608038	0.594833
$T=200$	0.818649	0.484561	0.424966	0.392844	0.371341	0.355397
$T=300$	0.740707	0.339025	0.278532	0.247585	0.227551	0.213061
$T=400$	0.670186	0.237916	0.183145	0.156551	0.139904	0.128159
$T=500$	0.606379	0.167458	0.120807	0.099312	0.086300	0.077346
$T=600$	0.548647	0.118214	0.079939	0.063204	0.053408	0.046832
$T=700$	0.496411	0.083693	0.053060	0.040352	0.033159	0.028449
$T=800$	0.449149	0.059423	0.035328	0.025844	0.020652	0.017337
$T=900$	0.406387	0.042311	0.023593	0.016603	0.012904	0.010599
$T=1000$	0.367695	0.030211	0.015803	0.010699	0.008087	0.006500

Table 17.1 Probability of Complete Success in the Game for $p = 0.001$

	$F=0$	$F=20$	$F=40$	$F=60$	$F=80$	$F=100$
$T=0$	1.000000	0.999900	0.999900	0.999900	0.999900	0.999900
$T=100$	0.990049	0.964118	0.957800	0.954040	0.951357	0.949272
$T=200$	0.980198	0.929646	0.917503	0.910314	0.905202	0.901238
$T=300$	0.970444	0.896434	0.878930	0.868621	0.861316	0.855664
$T=400$	0.960788	0.864436	0.842007	0.828866	0.819585	0.812423
$T=500$	0.951227	0.833605	0.806661	0.790958	0.779903	0.771394
$T=600$	0.941762	0.803899	0.772825	0.754808	0.742168	0.732462
$T=700$	0.932391	0.775275	0.740432	0.720335	0.706282	0.695518
$T=800$	0.923113	0.747694	0.709421	0.687460	0.672155	0.660461
$T=900$	0.913927	0.721116	0.679730	0.656107	0.639698	0.627192
$T=1000$	0.904833	0.695504	0.651304	0.626205	0.608829	0.595619

Table 17.2 Probability of Complete Success in the Game for $p = 0.0001$

	$F=0$	$F=20$	$F=40$	$F=60$	$F=80$	$F=100$
$T=0$	1.000000	0.999990	0.999990	0.999990	0.999990	0.999990
$T=100$	0.999000	0.996351	0.995696	0.995305	0.995025	0.994806
$T=200$	0.998002	0.992726	0.991422	0.990642	0.990084	0.989650
$T=300$	0.997004	0.989115	0.987166	0.986002	0.985169	0.984521
$T=400$	0.996008	0.985517	0.982928	0.981383	0.980278	0.979418
$T=500$	0.995012	0.981932	0.978709	0.976786	0.975412	0.974343
$T=600$	0.994018	0.978361	0.974509	0.972212	0.970571	0.969294
$T=700$	0.993024	0.974803	0.970326	0.967659	0.965753	0.964271
$T=800$	0.992032	0.971258	0.966163	0.963127	0.960960	0.959275
$T=900$	0.991040	0.967726	0.962017	0.958618	0.956192	0.954305
$T=1000$	0.990050	0.964208	0.957889	0.954129	0.951447	0.949361

Table 17.3 Probability of Complete Success in the Game for $p=0.00001$

Chapter 18

Bounds on Probabilities

In this chapter we develop a lower bound on the probability of success and both upper and lower bounds on the probability of complete success in the game. We start with the upper bound.

18.1 Upper Bound

There are several relatively easy upper bounds that can be obtained for the probability of complete success. The following theorem presents one which is best for the majority of interesting p, T and F values and is also very simple to prove.

Theorem 12 *The probability* $\mathbf{P}\{\text{Start}\rightsquigarrow\text{Complete Success}\}$ *of reaching the Complete Success state when starting at the Start state is bounded from above by*

$$\left(1 - \frac{pT}{T+F}\right)^F (1-p)^T.$$

Proof: In order to reach state $S_T(0,0,0)$ the game must necessarily go through the entry states of all the levels, $S_T(F,T,F)$, $S_T(F-1,T,F-1)$, $\ldots$, $S_T(1,T,1)$. In each state $S_T(f,T,f)$, where $1 \leqslant f \leqslant F$, there is a possibility of immediate failure, which has probability $\frac{pT}{T+f}$. The game must also go through all of the states $S_T(0,T,0)$, $S_T(0,T-1,0)$, $\ldots$, $S_T(0,1,0)$. In each of these states there is a possibility of immediate failure which has probability p. Thus, the probability of reaching the Complete Success state from the Start state is bounded from above by

$$\left(\prod_{f=1}^{F}\left(1-\frac{pT}{T+f}\right)\right)\cdot(1-p)^T \leqslant \left(1-\frac{pT}{T+F}\right)^F(1-p)^T,$$

and the proof of Theorem 12 is complete. ■

18.2 Lower Bound for Success

Reasonable lower bounds for the probability of success or complete success are much harder to obtain. In this section, we present a lower bound for the probability of success in the game and give a proof of it in a "top-down" fashion. Certain components of the proof are discussed in later sections. A lower bound for the probability of complete success can be developed from our lower bound for the probability of success; this is also done in a separate section.

Theorem 13 *The probability* $\sum_{f=0}^{F}\mathbf{P}\{\text{Start}\rightsquigarrow\text{Success}_f\}$ *of reaching any of the Success states when starting at the Start state is bounded from below by*

$$1-2pT\left(\ln\left(1+\frac{F-1}{pT+1}\right)+1.5\right),$$

for all $T \geqslant 2$.

Note that a similar expression gives a lower bound for all $T \geqslant 1$:

$$1 - 2pT\left(\ln\left(1 + \frac{F-2}{pT+2}\right) + 2.5\right).$$

We prove the correctness of both formulas but focus more on the former, since we are mostly interested in understanding the asymptotics of these bounds.

Proof: The main idea of the proof is changing the game to a harder one (in terms of the probability of success) which is easier to analyze. We call this the *new game* and refer to the original game as the *old game*. The new game differs from the old one only on some (higher) levels, namely on levels F, $F-1$, ..., f_0+1. The value of f_0 will be determined later, so as to make the bound as strong as possible. On lower levels (i.e., f_0, f_0-1, ..., 0) both games are essentially the same. Nevertheless, we do introduce new state symbols for the new game even on levels where it does not differ from the old game. For the top levels, the states are denoted by $S'_T(f, f')$, since, as we shall see, the game does not depend on the number of unmarked thin bottles at this stage. For the bottom levels, the states are denoted by $S'_T(f, t', f')$ with the same interpretation as $S_T(f, t', f')$ in the old game. Analogously to the old game, there are also the Start$'$, Failure$'$, Complete Success$'$ and Success$'_f$ states.

The first step of the proof of Theorem 13 is to introduce the new game with all the details. This is done in Section 18.3. The next step is to prove that the new game is really no easier than the old one. Formally, we have to prove that

$$\sum_{f=0}^{F} \mathbf{P}\{\text{Start} \rightsquigarrow \text{Success}_f\} \geqslant \sum_{f=0}^{F} \mathbf{P}\{\text{Start}' \rightsquigarrow \text{Success}'_f\},$$

and Claim 22 in Section 18.5 accomplishes this. The proof strongly depends on the fact that

$$\sum_{g=0}^{f} \mathbf{P}\{S_T(f,t',f') \rightsquigarrow \text{Success}_g\} < \sum_{g=0}^{f} \mathbf{P}\{S_T(f,t'-1,f') \rightsquigarrow \text{Success}_g\},$$

that is, picking a thin bottle to shoot at and missing it increases the probability of success (in the old game). The above inequality is proved by Claim 21 in Section 18.4.

The next step is to bound the probability of success in the new game. We assume that $F \geqslant f_0$. The possibility of reaching the upper level success states $\text{Success}'_F$, $\text{Success}'_{F-1}$, ..., $\text{Success}'_{f_0+1}$ is ignored, since the corresponding probabilities are very small. (This can be seen from the proof of the lower bound for complete success, in Section 18.8.) Formally, we say that

$$\sum_{f=0}^{F} \mathbf{P}\{\text{Start}' \rightsquigarrow \text{Success}'_f\} \geqslant \sum_{f=0}^{f_0} \mathbf{P}\{\text{Start}' \rightsquigarrow \text{Success}'_f\},$$

if $F \geqslant f_0$. Now we notice that the game will necessarily move through the state $S'_T(f_0,T,f_0)$ if it reaches a success state on levels 0 through f_0. Formally,

$$\sum_{f=0}^{f_0} \mathbf{P}\{\text{Start}' \rightsquigarrow \text{Success}'_f\}$$
$$= \mathbf{P}\{\text{Start}' \rightsquigarrow S'_T(f_0,T,f_0)\} \cdot \sum_{f=0}^{f_0} \mathbf{P}\{S'_T(f_0,T,f_0) \rightsquigarrow \text{Success}'_f\},$$

if $F \geqslant f_0$.

What remains is to bound the probability of reaching state $S'_T(f_0,T,f_0)$ from the Start$'$ state and to bound the probability of success from state $S'_T(f_0,T,f_0)$. Claim 23

in Section 18.6 proves that

$$\mathbf{P}\{\text{Start}'\rightsquigarrow S'_T(f_0,T,f_0)\} \geqslant 1 - 2pT\ln\frac{pT+F}{pT+f_0},$$

if $F \geqslant f_0$. The only other constraint imposed by this claim is that $f_0 \geqslant 0$ if $T \geqslant 2$ and $f_0 \geqslant 2$ if $T = 1$. The statement of the claim does not hold for $T = 0$ but the game is not very interesting in this case and easy to find the exact probability of success for. From the point of view of this claim, we would like f_0 to be as large as possible, but this would in fact harm the overall bound that we are proving. Therefore, f_0 is not chosen yet. Claim 24 in Section 18.7 proves that

$$\sum_{g=0}^{f}\mathbf{P}\{S'_T(g,T,g)\rightsquigarrow \text{Success}'_g\} \geqslant 1 - (pTg + pT + pg),$$

for all $f \leqslant f_0$. Obviously, this gives a lower bound for all $F \leqslant f_0$, a case that we had not considered yet. Of course, in this case we may want to use the exact formula for the probability of success, especially if f_0 is sufficiently small (which will be the case). However, since the ultimate goal of this analysis is to gain understanding about the asymptotics of success probabilities, we now return to the case when $F \geqslant f_0$.

What remains for the proof to be complete is to combine Claims 23 and 24 and

thus obtain the bound given by Theorem 13. We have that

$$\begin{aligned}
&\sum_{f=0}^{F} \mathbf{P}\{\text{Start}'\rightsquigarrow\text{Success}'_f\} \\
&\qquad \geqslant \mathbf{P}\{\text{Start}'\rightsquigarrow S'_T(f_0,T,f_0)\} \cdot \sum_{g=0}^{f_0} \mathbf{P}\{S'_T(f_0,T,f_0)\rightsquigarrow\text{Success}'_g\} \\
&\qquad = \left(1 - 2pT \ln \frac{pT+F}{pT+f_0}\right) \cdot \big(1 - (pTf_0 + pT + pf_0)\big) \\
&\qquad \geqslant 1 - 2pT \ln \frac{pT+F}{pT+f_0} - pTf_0 - pT - pf_0 \\
&\qquad = 1 - 2pT \ln \frac{pT+F}{pT+f_0} - pT(f_0+1) - pf_0 \\
&\qquad \geqslant 1 - 2pT \left(\ln\left(1 + \frac{F-f_0}{pT+f_0}\right) + f_0 + 0.5\right),
\end{aligned}$$

for all $F \geqslant f_0$ and respecting all the constraints introduced by Claim 23. At this point we can choose f_0 so as to make the bound as strong as possible. If we treat p, T and F as constants for a while and only f_0 as a variable, we can see that our goal is to minimize

$$\ln \frac{pT+F}{pT+f_0} + f_0,$$

for which it is best to take $f_0 = 1 - pT$. If $T = 1$ then the smallest $f_0 \geqslant 2$ will give the best bound, while if $T \geqslant 2$, we have a choice between 0 and 1 as the values for f_0. It can be seen however that this lower bound can be useful only in cases when pT is small, in fact, quite small, thus $f_0 = 1$ appears to be the better of the two choices. Setting $f_0 = 1$ completes the proof of Theorem 13. ■

18.3 The New Game

In this section we describe the new game with all the details. It is derived from the old game by disallowing the marking of thin bottles, as long as there are any unmarked fat bottles remaining. Thus, picking a thin bottle to shoot at and missing it is usually a wasted shot—nothing changes and the game is at the same state. In the next chapter we prove that marking a thin bottle in this case would actually increase the probability of success (in the old game), but we don't do it here, in the new game. One more section later we prove that this indeed makes the new game harder (in terms of the probability of success) than the old game. The reason we change the game is that the probabilities in the new game essentially lose their dependence on the number of unmarked thin bottles (because we do not mark any), and, because of this, are easier to analyze.

The above discussion is a little inaccurate. We really only change the game in the "upper" levels, that is, in levels F, $F-1$, ..., f_0+1, where f_0 will be determined later. On the "lower" levels, (i.e., f_0, f_0-1, ..., 0) both games are exactly the same.

The states in the new game are derived from the ones in the old game by adding a "prime" symbol ($'$). The interpretation of the parameters is the same. However, since the thin bottles are not marked as long as (1) the level is high (i.e., $f > f_0$), and (2) there are unmarked fat bottles (i.e., $f' > 0$), there are many states in the old game which do not have their counterparts in the new game. For example, states $S'_T(f, t', f')$, where $T > t'$, $f' > 0$ and $f > f_0$, simply do not exist. The only existing state for $f' > 0$ and $f > f_0$ is $S'_T(f, T, f')$ and we denote it by $S'_T(f, f')$ to keep the unnecessary parameter out. In effect, the transition from state $S_T(f, T, f')$ to state $S_T(f, T-1, f')$ in the old game has been replaced by a self loop back to

state $S_T(f, T, f')$.

This replacement has an effect on the probabilities associated with direct transitions from state $S'_T(f, f')$. In the old game we had the following four possibilities associated with state $S_T(f, T, f')$:

1. Pick a fat bottle, shoot and hit it, moving on to state $S_T(f-1, T, f-1)$ one level below. The probability of this happening was $\frac{f'}{T+f'}(1-p)$;
2. Pick a fat bottle, shoot and miss it, moving to state $S_T(f, T, f'-1)$ on the same level, with probability $\frac{f'}{T+f'}p$;
3. Pick a thin bottle, shoot and hit it, terminating the game in the Failure$'$ state, which could happen with probability $\frac{T}{T+f'}p$;
4. Pick a thin bottle, shoot and miss it, moving to state $S_T(f, t'-1, f')$ on the same level. The probability of this was $\frac{T}{T+f'}(1-p)$.

Now this last option is eliminated and the probabilities for the former three possibilities have to be adjusted. We get the corrected probabilities if we divide each one of them by

$$1 - \frac{T}{T+f'}(1-p) = \frac{pT+f'}{T+f'},$$

which is the combined probability of the former three possibilities. After this adjustment is done, we have the following possible transitions from state $S'_T(f, f')$ in the new game:

1. Pick a fat bottle, shoot and hit it, moving on to state $S'_T(f-1, f-1)$ one level below. The probability of this happening is $\frac{f'}{pT+f'}(1-p)$;

2. Pick a fat bottle, shoot and miss it, moving to state $S'_T(f, f'-1)$ on the same level, with probability $\frac{f'}{pT+f'}p$;

3. Pick a thin bottle, shoot and hit it, terminating the game in the Failure$'$ state, which could happen with probability $\frac{T}{pT+f'}p$.

As we see, all these probabilities add to 1, as needed. The self loop has been eliminated by recalculating the probabilities.

When $f > f_0$ but $f' = 0$, marking of thin bottles is resumed. At this point, however, there are not many possibilities how the game can end. Namely, if all T thin bottles are now shot at and missed, the game ends in the Success$'_f$ state, while if any one of them is hit, the game ends in the Failure$'$ state. Therefore, we can simplify the game some more and just say that from state $S'_T(f, 0)$ (which obviously corresponds to state $S_T(f, T, 0)$ in the old game) there is a probability of $(1-p)^T$ associated with a direct transition to the Success$'_f$ state and there are no other transitions from this state except to the Failure$'$ state.

As mentioned above, when the new game reaches level f_0, the rules of the old game are restored. The lower portion of its state diagram is isomorphic to the one for the old game, and the probabilities associated with the transitions are the same. The new game has essentially the same states, the same transitions and the same probabilities on these levels. The state with T total thin bottles, f total fat bottles, t' unmarked thin bottles and f' unmarked fat bottles is denoted by $S'_T(f, t', f')$. The prime symbol is used just as a reminder that it is a “different” game. A state diagram of the new game is given in Figure 18.1.

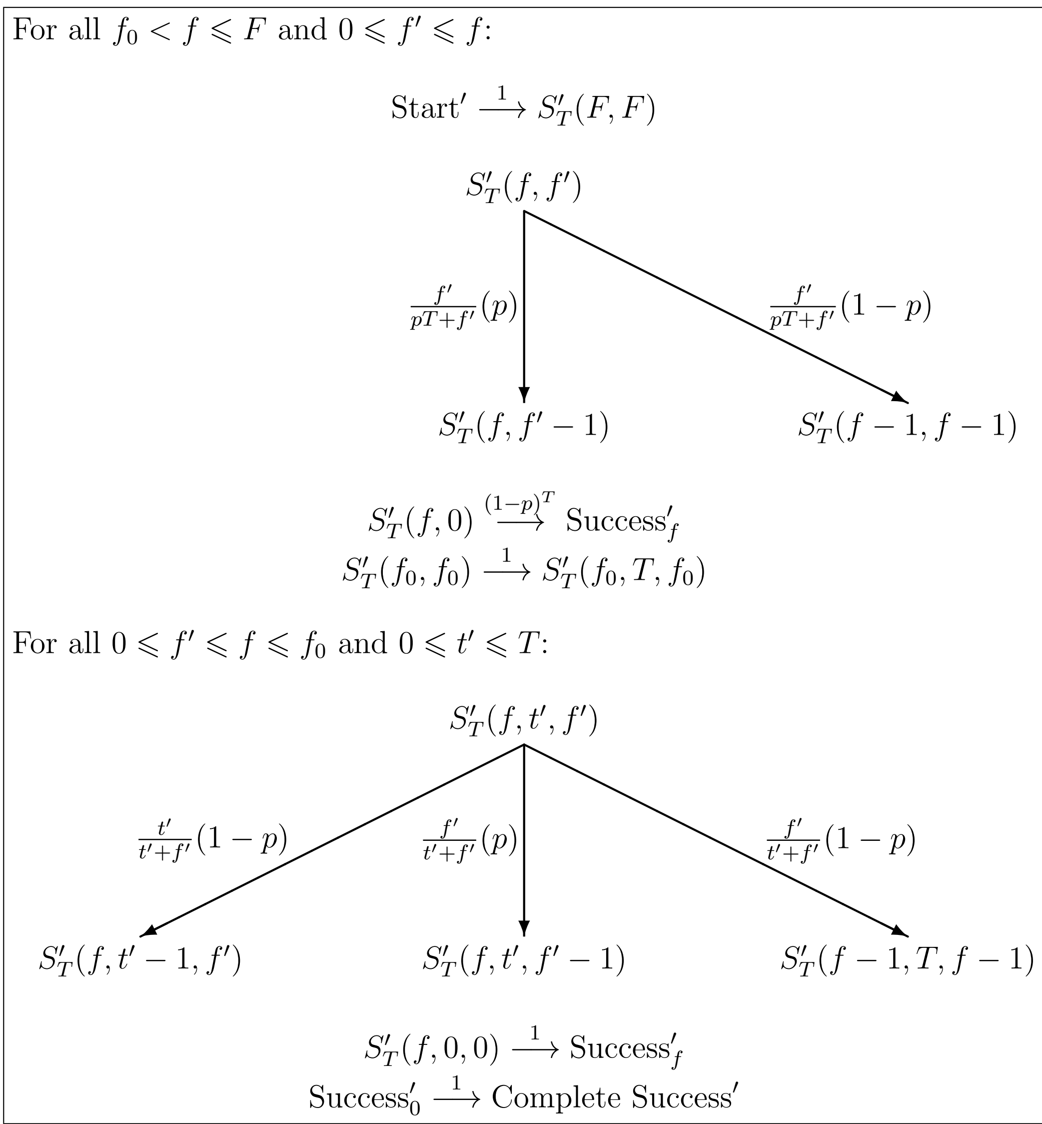

Figure 18.1 States, transitions and their probabilities in the new game

18.4 Missing Thin Bottles

This section introduces the idea behind the invention of the new game. It appears that every time when a thin bottle is picked, shot at, missed and marked in the old game, the probability of success increases (assuming that $0 < p < 1/2$). This leads to the introduction of the new game where the thin bottles are not marked temporarily, as already described in Section 18.3. A full proof that the probability of success in the new game does not exceed that in the old game is given in Section 18.5. Now we present the main result of this section.

Claim 21 *Picking a thin bottle, shooting at it and missing increases the probability of success in the old game:*

$$\sum_{g=0}^{f} \mathbf{P}\{S_T(f,t',f') \rightsquigarrow \text{Success}_g\} < \sum_{g=0}^{f} \mathbf{P}\{S_T(f,t'-1,f') \rightsquigarrow \text{Success}_g\},$$

for all $0 \leqslant f' \leqslant f \leqslant F$ *and* $1 \leqslant t' \leqslant T$.

Proof: We prove this claim by induction on f', the number of unmarked fat bottles.

Induction Basis: Claim 21 holds if there are no unmarked fat bottles:

$$\sum_{g=0}^{f} \mathbf{P}\{S_T(f,t',0) \rightsquigarrow \text{Success}_g\} < \sum_{g=0}^{f} \mathbf{P}\{S_T(f,t'-1,0) \rightsquigarrow \text{Success}_g\},$$

for all $0 \leqslant f \leqslant F$ and $1 \leqslant t' \leqslant T$.

Proof: In this case the only bottles that can be picked and shot at are thin. If all of them are missed, the game ends in the Success_f state. If any of them is hit, the game ends in the Failure state. Therefore, it is not possible to get to any of the

Success$_g$ states for $g < f$ and, quite obviously,

$$\begin{aligned}\sum_{g=0}^{f} \mathbf{P}\{S_T(f,t',0) \rightsquigarrow \text{Success}_g\} &= \mathbf{P}\{S_T(f,t',0) \rightsquigarrow \text{Success}_f\} \\ &= (1-p)^{t'} \\ &< (1-p)^{t'-1} = \mathbf{P}\{S_T(f,t'-1,0) \rightsquigarrow \text{Success}_f\} \\ &\qquad\qquad = \sum_{g=0}^{f} \mathbf{P}\{S_T(f,t'-1,0) \rightsquigarrow \text{Success}_g\},\end{aligned}$$

for all $0 \leqslant f \leqslant F$ and $1 \leqslant t' \leqslant T$. ■

Induction Hypothesis: We assume that Claim 21 holds when the number of unmarked fat bottles is $f' - 1$:

$$\sum_{g=0}^{f} \mathbf{P}\{S_T(f,t',f'-1) \rightsquigarrow \text{Success}_g\} < \sum_{g=0}^{f} \mathbf{P}\{S_T(f,t'-1,f'-1) \rightsquigarrow \text{Success}_g\},$$

where $1 \leqslant f' \leqslant f \leqslant F$ and $1 \leqslant t' \leqslant T$

Inductive Step: To complete the induction we need to prove that Claim 21 holds when the number of unmarked fat bottles is f'. The proof is again by induction, this time on the number of remaining unmarked thin bottles, t'.

Induction Basis: Claim 21 holds when only one unmarked thin bottle and f' unmarked fat bottles remain:

$$\sum_{g=0}^{f} \mathbf{P}\{S_T(f,1,f') \rightsquigarrow \text{Success}_g\} < \sum_{g=0}^{f} \mathbf{P}\{S_T(f,0,f') \rightsquigarrow \text{Success}_g\},$$

for all $0 \leqslant f' \leqslant f \leqslant F$.

Proof: In order to make the displayed material shorter and improve its readabil-

ity we introduce the following notation for the probability of success from state $S_T(f, t', f')$:

$$P_T(f, t', f') \stackrel{\text{def}}{=} \sum_{g=0}^{f} \mathbf{P}\{S_T(f, t', f') \rightsquigarrow \text{Success}_g\}, \tag{$*$}$$

for all $0 \leqslant f' \leqslant f \leqslant F$ and $0 \leqslant t' \leqslant T$.

According to this notation, we may write that

$$\begin{aligned} P_T(f, 1, f') = \frac{f'(1-p)}{f'+1} P_T(f-1, T, f-1) + \frac{f'p}{f'+1} P_T(f, 1, f'-1) \\ + \frac{(1-p)}{f'+1} P_T(f, 0, f'), \end{aligned}$$

and that

$$P_T(f, 0, f') = (1-p) P_T(f-1, T, f-1) + p P_T(f, 0, f'-1).$$

Therefore, we have to show that

$$\begin{aligned} \frac{f'(1-p)}{f'+1} P_T(f-1, T, f-1) + \frac{f'p}{f'+1} P_T(f, 1, f'-1) + \frac{(1-p)}{f'+1} P_T(f, 0, f') \\ < (1-p) P_T(f-1, T, f-1) + p P_T(f, 0, f'-1), \end{aligned}$$

or that

$$\begin{aligned} \frac{f'(1-p)}{f'+1} & P_T(f-1, T, f-1) + \frac{f'p}{f'+1} P_T(f, 1, f'-1) \\ & < \big((1-p) P_T(f-1, T, f-1) + p P_T(f, 0, f'-1)\big) \left(1 - \frac{1-p}{f'+1}\right) \\ & = \big((1-p) P_T(f-1, T, f-1) + p P_T(f, 0, f'-1)\big) \frac{f'+p}{f'+1} \\ & = \frac{(f'+p)(1-p)}{f'+1} P_T(f-1, T, f-1) + \frac{(f'+p)p}{f'+1} P_T(f, 0, f'-1), \end{aligned}$$

which is easy because $P_T(f,1,f'-1) < P_T(f,0,f'-1)$ by the induction hypothesis on unmarked fat bottles. ■

Induction Hypothesis: We assume that Claim 21 holds when the number of remaining unmarked thin bottles is $t'-1$ and the number of unmarked fat bottles is f':

$$\sum_{g=0}^{f} \mathbf{P}\{S_T(f,t'-1,f') \rightsquigarrow \text{Success}_g\} < \sum_{g=0}^{f} \mathbf{P}\{S_T(f,t'-2,f') \rightsquigarrow \text{Success}_g\},$$

where $0 \leqslant f' \leqslant f \leqslant F$ and $1 \leqslant t' \leqslant T$.

Inductive Step: To complete the induction, we need to prove that Claim 21 holds when the number of unmarked thin bottles is t' and the number of unmarked fat bottles is f'. According to the notation used in the proof of the base case of this induction (on unmarked thin bottles), we have to prove that $P_T(f,t',f') < P_T(f,t'-1,f')$, where $0 \leqslant f' \leqslant f \leqslant F$ and $1 \leqslant t' \leqslant T$;

Proof: We know that

$$P_T(f,t'-1,f') = \frac{f'(1-p)}{f'+t'-1} P_T(f-1,T,f-1) + \frac{f'p}{f'+t'-1} P_T(f,t'-1,f'-1) + \frac{(t'-1)(1-p)}{f'+t'-1} P_T(f,t'-2,f').$$

Using the inductive hypothesis on the number of remaining unmarked thin bottles, we can rewrite it as

$$P_T(f,t'-1,f') > \frac{f'(1-p)}{f'+t'-1} P_T(f-1,T,f-1) + \frac{f'p}{f'+t'-1} P_T(f,t'-1,f'-1) + \frac{(t'-1)(1-p)}{f'+t'-1} P_T(f,t'-1,f').$$

If we try to solve it for $P_T(f, t'-1, f')$, we get the following:

$$
\begin{aligned}
&P_T(f,t'-1,f')\left(1-\frac{(t'-1)(1-p)}{f'+t'-1}\right)\\
&\qquad > \frac{f'(1-p)}{f'+t'-1}P_T(f-1,T,f-1)+\frac{f'p}{f'+t'-1}P_T(f,t'-1,f'-1),\\
&P_T(f,t'-1,f')\frac{f'+t'p-p}{f'+t'-1}\\
&\qquad > \tfrac{f'(1-p)}{f'+t'-1}P_T(f-1,T,f-1)+\tfrac{f'p}{f'+t'-1}P_T(f,t'-1,f'-1),\\
&P_T(f,t'-1,f')\\
&\qquad > \frac{f'(1-p)}{f'+t'p-p}P_T(f-1,T,f-1)+\frac{f'p}{f'+t'p-p}P_T(f,t'-1,f'-1).
\end{aligned}
$$

Since

$$
\begin{aligned}
P_T(f,t',f') = \frac{f'(1-p)}{f'+t'}P_T(f-1,T,f-1) &+ \frac{f'p}{f'+t'}P_T(f,t',f'-1)\\
&+\frac{t'(1-p)}{f'+t'}P_T(f,t'-1,f'),
\end{aligned}
$$

we have to show that

$$
\begin{aligned}
&\frac{f'(1-p)}{f'+t'}P_T(f-1,T,f-1)+\frac{f'p}{f'+t'}P_T(f,t',f'-1)\\
&\qquad\qquad < P_T(f,t'-1,f')\left(1-\frac{t'(1-p)}{f'+t'}\right)\\
&\qquad\qquad = P_T(f,t'-1,f')\frac{f'+t'p}{f'+t'}.
\end{aligned}
$$

Now it suffices to show that

$$\begin{aligned}
&\frac{f'(1-p)}{f'+t'}P_T(f-1,T,f-1)+\frac{f'p}{f'+t'}P_T(f,t',f'-1)\\
&<\left(\frac{f'(1-p)}{f'+t'p-p}P_T(f-1,T,f-1)+\frac{f'p}{f'+t'p-p}P_T(f,t'-1,f'-1)\right)\frac{f'+t'p}{f'+t'}\\
&=\frac{f'(1-p)(f'+t'p)}{(f'+t')(f'+t'p-p)}P_T(f-1,T,f-1)\\
&\quad+\frac{f'p(f'+t'p)}{(f'+t')(f'+t'p-p)}P_T(f,t'-1,f'-1).
\end{aligned}$$

This is not hard, using the inductive hypothesis on the number of unmarked fat bottles, which says that $P_T(f,t',f'-1) < P_T(f,t'-1,f'-1)$. ■

With this we have concluded the induction on the remaining number of unmarked thin bottles, t', and thus proved the induction step on the number of unmarked fat bottles, f'. Therefore, the induction on the number of unmarked fat bottles is complete and Claim 21 is proved. ■

18.5 The New Game is Not Easier

In this section we use Claim 21 to prove that the probability of success in the new game does not exceed the probability of success in the old game. In other words, the new game is no easier than the old one. We already know that on levels f_0, f_0-1, ..., 0 both games are the same, thus, we only need to prove that for $F > f_0$ the following holds:

$$\sum_{g=0}^{F}\mathbf{P}\{S'_T(F,F)\rightsquigarrow\text{Success}'_g\}\leqslant\sum_{g=0}^{F}\mathbf{P}\{S_T(F,T,F)\rightsquigarrow\text{Success}_g\}.$$

The claim that we prove in this section is actually stronger. It proves that for every state of the new game, the probability of reaching the success states will not exceed the probability of success from the corresponding state in the old game. There is no need for the claim to be so strong, it is merely a side effect of the method of proving it.

Claim 22 *The probability of reaching any of the success states when starting at state* $S'_T(f, f')$ *in the new game is less than or equal to the probability of reaching any of the success states when starting at state* $S_T(f, T, f')$ *in the old game:*

$$\sum_{g=0}^{f} \mathbf{P}\{S'_T(f, f') \rightsquigarrow \text{Success}'_g\} \leqslant \sum_{g=0}^{f} \mathbf{P}\{S_T(f, T, f') \rightsquigarrow \text{Success}_g\},$$

for all $f_0 < f \leqslant F$ *and* $0 \leqslant f' \leqslant f$.

Proof: The proof is by induction on the total number of fat bottles, f.

Induction Basis: There is a transition with probability 1 from state $S'_T(f_0, f_0)$ to state $S'_T(f_0, T, f_0)$ in the new game and from then on it proceeds just as the old game. Therefore,

$$\sum_{g=0}^{f} \mathbf{P}\{S'_T(f_0, f_0) \rightsquigarrow \text{Success}'_g\} = \sum_{g=0}^{f} \mathbf{P}\{S_T(f_0, T, f_0) \rightsquigarrow \text{Success}_g\}.$$

Induction Hypothesis: We assume that the following inequality holds:

$$\sum_{g=0}^{f-1} \mathbf{P}\{S'_T(f-1, f-1) \rightsquigarrow \text{Success}'_g\} \leqslant \sum_{g=0}^{f-1} \mathbf{P}\{S_T(f-1, T, f-1) \rightsquigarrow \text{Success}_g\}.$$

Inductive Step: Now we just need to prove that Claim 22 holds when there are

f total fat bottles. This is also done by induction; this time the induction is on the number of remaining unmarked fat bottles, f'.

Induction Basis: When there are f total fat bottles but none of them are unmarked, the following holds:

$$\sum_{g=0}^{f} \mathbf{P}\{S'_T(f,0) \rightsquigarrow \text{Success}'_g\} = \sum_{g=0}^{f} \mathbf{P}\{S_T(f,T,0) \rightsquigarrow \text{Success}_g\}.$$

Proof: In this case the only bottles that can be picked and shot at in the old game are the thin ones. If all of them are missed, then the game ends in the Success_f state. Otherwise it ends in the Failure state. Therefore,

$$\begin{aligned}\sum_{g=0}^{f} \mathbf{P}\{S_T(f,T,0) \rightsquigarrow \text{Success}_g\} &= \mathbf{P}\{S_T(f,T,0) \rightsquigarrow \text{Success}_f\} \\ &= (1-p)^T \\ &= \mathbf{P}\{S'_T(f,0) \rightarrow \text{Success}'_f\} \\ &= \sum_{g=0}^{f} \mathbf{P}\{S'_T(f,0) \rightsquigarrow \text{Success}'_g\}. \qquad \blacksquare\end{aligned}$$

Induction Hypothesis: We assume that Claim 22 holds when from f total fat bottles $f'-1$ remain unmarked:

$$\sum_{g=0}^{f} \mathbf{P}\{S'_T(f,f'-1) \rightsquigarrow \text{Success}'_g\} \leqslant \sum_{g=0}^{f} \mathbf{P}\{S_T(f,T,f'-1) \rightsquigarrow \text{Success}_g\}.$$

Inductive Step: To complete the induction we need to prove that the claim holds when there are f total fat bottles and f' of them are unmarked. At this point we again use the notation $(*)$ introduced in the proof of Claim 21 which allows us to express the probability of success in the old game in a more compact way. We also

introduce similar notation for the probability of success from state $S'_T(f, f')$ in the new game:

$$P'_T(f, f') \stackrel{\text{def}}{=} \sum_{g=0}^{f} \mathbf{P}\{S'_T(f, f') \rightsquigarrow \text{Success}'_g\},$$

for all $f_0 < f \leqslant F$ and $0 \leqslant f' \leqslant f$, as well as for $f = f' = f_0$. According to this notation, we now have to prove that $P'_T(f, f') \leqslant P_T(f, T, f')$.

Proof: We can express the probabilities of success in each game as:

$$\begin{aligned}
P'_T(f, f') &= \frac{f'p}{pT + f'} P'_T(f, f' - 1) + \frac{f'(1-p)}{pT + f'} P'_T(f - 1, f - 1), \\
P_T(f, T, f') &= \frac{f'p}{f' + T} P_T(f, T, f' - 1) + \frac{f'(1-p)}{f' + T} P_T(f - 1, T, f - 1) \\
&\qquad + \frac{T(1-p)}{f' + T} P_T(f, T - 1, f')
\end{aligned}$$

By the inductive assumption on the number of remaining unmarked fat bottles we have that $P'_T(f, f' - 1) \leqslant P_T(f, T, f' - 1)$. By the inductive assumption on the number of total fat bottles we have that $P'_T(f - 1, f - 1) \leqslant P_T(f - 1, T, f - 1)$. By Claim 21 we know that $P_T(f, T, f') < P_T(f, T - 1, f')$. Therefore, we can write that

$$\begin{aligned}
P_T(f, T, f') &> \frac{f'p}{f' + T} P'_T(f, f' - 1) + \frac{f'(1-p)}{f' + T} P'_T(f - 1, f - 1) \\
&\qquad + \frac{T(1-p)}{f' + T} P_T(f, T, f').
\end{aligned}$$

If we solve this for $P_T(f,T,f')$ we get that

$$\begin{aligned} P_T(f,T,f')\left(1-\frac{T(1-p)}{f'+T}\right) &> \frac{f'p}{f'+T}P'_T(f,f'-1)+\frac{f'(1-p)}{f'+T}P'_T(f-1,f-1), \\ P_T(f,T,f')\frac{pT+f'}{f'+T} &> \frac{f'p}{f'+T}P'_T(f,f'-1)+\frac{f'(1-p)}{f'+T}P'_T(f-1,f-1), \\ P_T(f,T,f') &> \frac{f'p}{pT+f'}P'_T(f,f'-1)+\frac{f'(1-p)}{pT+f'}P'_T(f-1,f-1), \\ P_T(f,T,f') &> P'_T(f,f'). \end{aligned}$$

■

This concludes the induction on the number of remaining unmarked fat bottles and thus also the induction on the number of total fat bottles. Claim 22 is proved. ■

18.6 Getting Through the Top Levels

In this section we bound the probability of reaching level f_0 in the new game from a start state above it.

Claim 23 *The probability* $\mathbf{P}\{\mathrm{Start}'\rightsquigarrow S'_T(f_0,T,f_0)\}$ *of reaching level* f_0 *in the new game when starting at state* Start' *is bounded from below by*

$$1-2pT\ln\frac{pT+F}{pT+f_0},$$

for all $F \geqslant f_0$*, assuming that* $f_0 \geqslant 2$ *if* $T=1$ *and* $f_0 \geqslant 0$ *if* $T \geqslant 2$*.*

Proof: We start by deriving a lower bound on the probability of reaching level $f-1$ from the entry state of level f. That is, we are going to bound the value of $\mathbf{P}\{S'_T(f,f)\rightsquigarrow S'_T(f-1,f-1)\}$ from below. In order to get to level $f-1$ we have to

hit any of the unmarked fat bottles, of which there are f initially. If we miss all of them, then we cannot get to the next level any more. Therefore, to get to the next level, one of the following must happen: we hit the first fat bottle that we shoot at (this can happen with probability $\frac{f(1-p)}{pT+f}$); we miss the first one but hit the second one (probability $\frac{fp(f-1)(1-p)}{(pT+f)(pT+f-1)}$); and so on. In general, the probability of missing the first i fat bottles picked and hitting the $i+1$-th bottle is

$$p^i(1-p)\frac{f(f-1)\cdots(f-i)}{(pT+f)(pT+f-1)\cdots(pT+f-i)}.$$

Therefore,

$$\begin{aligned}
&\mathbf{P}\{S'_T(f,f)\rightsquigarrow S'_T(f-1,f-1)\}\\
&\qquad= \sum_{i=0}^{f-1} p^i(1-p)\frac{f(f-1)\cdots(f-i)}{(pT+f)(pT+f-1)\cdots(pT+f-i)}\\
&\qquad= (1-p)\sum_{i=0}^{f-1} p^i\frac{f}{pT+f}\cdot\frac{f-1}{pT+f-1}\cdot\cdots\cdot\frac{f-i}{pT+f-i}\\
&\qquad\geqslant (1-p)\sum_{i=0}^{f-1} p^i\left(\frac{1}{pT+1}\right)^i\\
&\qquad= (1-p)\frac{1-\left(\frac{p}{pT+1}\right)^f}{1-\frac{p}{pT+1}}.
\end{aligned}$$

If we denote $\alpha \overset{\text{def}}{=} \frac{1}{pT+1}$, then $\alpha < 1$ and the inequality above may be rewritten as

$$\begin{aligned}
\mathbf{P}\{S'_T(f,f)\rightsquigarrow S'_T(f-1,f-1)\} &\geqslant \frac{1-p}{1-\alpha p}\big(1-(\alpha p)^f\big)\\
&\geqslant \frac{1-p}{1-\alpha p}(1-p^f). \qquad (**)
\end{aligned}$$

Now we prove two simple lemmas.

Lemma 16 *For* $\alpha = \frac{1}{pT+1}$ *and* $f \geqslant 1$ *it holds that*

$$\frac{1-p}{1-\alpha p} \geqslant 1 - \frac{pT}{pT+f}.$$

Proof: First we prove that $\frac{1-p}{1-\alpha p} \geqslant 1 - 2p(1-\alpha)$. Assume the opposite, i.e., that $\frac{1-p}{1-\alpha p} < 1 - 2p(1-\alpha)$. Then

$$\begin{aligned}
1-p &< \big(1-2p(1-\alpha)\big)(1-\alpha p) \\
&= 1 - p + 2\alpha p^2(1-\alpha) - (1-\alpha)p, \\
0 &< 2\alpha p^2(1-\alpha) - (1-\alpha)p, \\
(1-\alpha)p &< 2\alpha p^2(1-\alpha), \\
1 &< 2\alpha p, \\
1/2 &< \alpha p,
\end{aligned}$$

which is a contradiction since $p < 1/2$ and $\alpha < 1$. Therefore $\frac{1-p}{1-\alpha p} \geqslant 1 - 2p(1-\alpha)$ and now it suffices to prove that $1 - 2p(1-\alpha) \geqslant 1 - \frac{pT}{pT+f}$. We know that $1 - \alpha = \frac{pT}{pT+1} \leqslant \frac{pT}{pT+f}$, for $f \geqslant 1$. Therefore, since $p < 1/2$, we have that

$$\begin{aligned}
2p(1-\alpha) &< \frac{pT}{pT+f}, \\
1 - 2p(1-\alpha) &> 1 - \frac{pT}{pT+f}.
\end{aligned}$$

■

Lemma 17 *For all* $f \geqslant 3$, $T \geqslant 1$ *and* $f \geqslant 1$, $T \geqslant 2$ *it holds that*

$$1 - p^f > 1 - \frac{pT}{pT+f}.$$

Proof: We need to prove that $p^f < \frac{pT}{pT+f}$ or that $fp^{f-1} < T(1-p^f)$. For this it suffices to prove that $f(1/2)^{f-1} \leqslant T(1-(1/2)^f)$.

If we assume that $T \geqslant 1$, it suffices to prove that

$$\begin{aligned} f(1/2)^{f-1} &\leqslant 1-(1/2)^f \\ (f+1/2)(1/2)^{f-1} &\leqslant 1 \\ f+1/2 &\leqslant 2^{f-1}. \end{aligned}$$

The last inequality holds for all $f \geqslant 3$.

If, however, we assume that $T \geqslant 2$, it suffices to prove that

$$\begin{aligned} f(1/2)^{f-1} &\leqslant 2-(1/2)^{f-1} \\ (f+1)(1/2)^{f-1} &\leqslant 2 \\ f+1 &\leqslant 2^f. \end{aligned}$$

The last inequality holds for all $f \geqslant 1$. ■

Two more lemmas are given here without proofs. Lemma 18 can be easily proved by induction while Lemma 19 can be proved by merely looking at the geometric interpretation of integrals.

Lemma 18 *For all nonnegative $a_1, a_2, \ldots, a_n$ the following inequalities hold:*

$$1-\sum_{i=1}^{n} a_i \leqslant \prod_{i=1}^{n}(1-a_i) \leqslant 1-\sum_{i=1}^{n} a_i + \sum_{1\leqslant i<j\leqslant n} a_i a_j.$$

Lemma 19 *For all $X > 1$ and $Y \geqslant 0$ the following inequalities hold:*

$$\ln\frac{X+Y+1}{X} = \int_{X-1}^{X+Y}\frac{dz}{z+1} < \sum_{z=0}^{Y}\frac{1}{X+z} < \int_{X-1}^{X+Y}\frac{dz}{z} = \ln\frac{X+Y}{X-1}.$$

We now continue with the proof of Claim 23. Using Lemmas 16, 17 and 18 and equation $(**)$ we may now write that

$$\begin{aligned}\mathbf{P}\{S'_T(f,f)\rightsquigarrow S'_T(f-1,f-1)\} &> \left(1-\frac{pT}{pT+f}\right)^2\\ &\geqslant 1-\frac{2pT}{pT+f},\end{aligned}$$

which holds for all $T \geqslant 1$, $f \geqslant 3$ and for all $T \geqslant 2$, $f \geqslant 1$.

The probability $\mathbf{P}\{\text{Start}'\rightsquigarrow S'_T(f_0,T,f_0)\}$ of reaching level f_0 from the Start$'$ state in the new game can now be expressed as

$$\begin{aligned}\mathbf{P}\{\text{Start}'\rightsquigarrow S'_T(f_0,T,f_0)\} &= \mathbf{P}\{S'_T(F,F)\rightsquigarrow S'_T(f_0,f_0)\}\\ &= \prod_{f=f_0+1}^{F} \mathbf{P}\{S'_T(f,f)\rightsquigarrow S'_T(f-1,f-1)\}\\ &> \prod_{f=f_0+1}^{F}\left(1-\frac{2pT}{pT+f}\right).\end{aligned}$$

Using Lemma 18 we may rewrite this as

$$\begin{aligned}\mathbf{P}\{\text{Start}'\rightsquigarrow S'_T(f_0,T,f_0)\} &> 1-\sum_{f=f_0+1}^{F}\frac{2pT}{pT+f}\\ &= 1-2pT\sum_{f=f_0+1}^{F}\frac{1}{pT+f}.\end{aligned}$$

Using Lemma 19 we get that

$$\mathbf{P}\{\text{Start}'\rightsquigarrow S'_T(f_0,T,f_0)\} > 1-2pT\ln\frac{pT+F}{pT+f_0},$$

which concludes the proof of Claim 23. ■

18.7 Succeeding on Lower Levels

In this section we bound from below the probability of reaching any of the success states on levels f_0 and lower in the new game, assuming that we have already reached state $S'_T(f_0, T, f_0)$. As we know, from this level both games are the same, so we analyze the new game just as we would analyze the old one.

Claim 24 *The probability $\sum_{g=0}^{f} \mathbf{P}\{S'_T(f,T,f) \rightsquigarrow \mathrm{Success}'_g\}$ of reaching success in the new game when starting in state $S'_T(f,T,f)$ is bounded from below by $1 - (pTf + pT + pf)$, for all $0 \leqslant f \leqslant f_0$.*

Proof: Suppose that on each level g we miss all the thin bottles that we choose and hit the first fat bottle. This way we inevitably get to the entry state of level $g-1$ below. The probability of this happening is $\sum_{t=0}^{T}(\alpha(g,T,t)(1-p)^t(1-p)$, where $\alpha(g,T,t)$ denotes the probability of choosing exactly t thin bottles in a row, when given T thin and g fat bottles. Since $\sum_{t=0}^{T} \alpha(g,T,t) = 1$, we can easily conclude that the probability to reach level $g-1$ from level g is at least $(1-p)^{T+1}$. Obviously, if we reach the lowest level and then miss all the thin bottles there, we will have reached a success state. Therefore

$$\begin{aligned}\sum_{g=0}^{f} \mathbf{P}\{S'_T(f,T,f) \rightsquigarrow \mathrm{Success}'_g\} &\geqslant (1-p)^{(T+1)f}\mathbf{P}\{S'_T(0,T,0) \rightsquigarrow \mathrm{Success}'_0\} \\ &= (1-p)^{(T+1)f}(1-p)^T \\ &= (1-p)^{Tf+f+T}.\end{aligned}$$

We now apply Lemma 18 and the claim follows. ■

18.8 Lower Bound for Complete Success

In this section we prove a lower bound for the probability of complete success in the game. The bound seems quite close to the lower bound for the probability of success.

Theorem 14 *The probability* $\mathbf{P}\{\text{Start}\rightsquigarrow\text{Complete Success}\}$ *of reaching the* Complete Success *state when starting at the* Start *state is bounded from below by*

$$1 - 2pT\left(\ln\left(1 + \frac{F-1}{pT+1}\right) + 2\right),$$

for all $T \geqslant 2$.

Proof: We prove an easy upper bound on the probability of reaching any of the states Success_f where $f > 0$ and then subtract it from our existing lower bound on the probability of success. In order to reach state Success_f, for any $0 < f \leqslant F$ it is necessary to first reach state $S_T(f, T, f)$ and then shoot at and miss every single bottle on level f. Therefore,

$$\begin{aligned}
\sum_{f=1}^{F} \mathbf{P}\{\text{Start}\rightsquigarrow\text{Success}_f\} &= \sum_{f=1}^{F} \mathbf{P}\{\text{Start}\rightsquigarrow S_T(f,T,f)\} p^f (1-p)^T \\
&\leqslant \sum_{f=1}^{F} p^f (1-p)^T \\
&= (1-p)^T p \frac{1-p^F}{1-p} \\
&\leqslant (1-p)^{T-1} p,
\end{aligned}$$

which is bounded from above by p because $T \geqslant 1$. All that remains is to subtract p from the bound given by Theorem 13. We subtract pT instead of p because it makes the final bound shorter and more similar to the bound given in Theorem 13. ■

Obviously, if we take the weaker lower bound on the probability of success which holds for all $T \geqslant 1$, we get a weakened lower bound for the probability of complete success:

$$1 - 2pT\left(\ln\left(1 + \frac{F-2}{pT+2}\right) + 3\right),$$

for all $T \geqslant 1$.

Chapter 19

Comparison to a Simpler Algorithm and Conclusions

The lower bound on complete success of algorithm RandomReduce learning monotone monomials exhibits the same asymptotics as the sample values tabulated in the tables given in the end of Chapter 17. That is, the probability of complete success (and success) seems to drop proportionally to the increase in the number of needed 1-bits (thin bottles) and to drop logarithmically with respect to the increase in the number of unwanted 1-bits (fat bottles). In other words, starting RandomReduce high in the lattice, far from the minimum point of the monomial does not pose a big problem for the algorithm. However, trying to learn a monomial containing many variables is hard for RandomReduce. And, of course, the higher the error rate p, the harder it is for the algorithm to find the minimum point.

The purpose of this part of the thesis was not to investigate the learnability of monotone monomials from equivalence queries and membership queries with random persistent errors. Monotone monomials can be learned in polynomial time from

equivalence queries alone, thus errors in membership queries do not matter. The goal was to estimate whether it would be practical to use RandomReduce in algorithm LearnMonDNF from Section 11.1 of Part 2 instead of its original subroutine Reduce in order to apply it to the model of equivalence and random membership queries. Unfortunately, the lower bound suggests that the usefulness of this modified algorithm LearnMonDNF would be very limited (assuming no further modifications are made). A polynomial-time learning algorithm for this model is required to have probability $1-\delta$ of exactly identifying the target concept in time polynomial in the size of the concept, the size of the longest counterexample, $1/\delta$ and the inverse of $1/2-p$, for any $\delta > 0$. The closer the error rate p is to $1/2$, the more time is the algorithm allowed.

If such running times are what we need to achieve then the lower bound on the probability of complete success does not give rise to much hope. Each "failure" in RandomReduce (i.e., adding a negative point as a term to the formula) would eventually be corrected by a negative counterexample resulting in the removal of the offending term(s), but a rather high number of runs of RandomReduce would be required to find the correct term. Having RandomReduce return a positive point which is not a minimum point would not cause a subsequent negative counterexample but we would still need to repeat RandomReduce for the same term until it is found. If we ran RandomReduce for each term a number of times that is the inverse of the lower bound of complete success, we could be sure that with very high probability we have found all the terms. This number however is not polynomial in the inverse of $1/2-p$.

Despite the fact that the applicability of RandomReduce in the general case appears limited, it may be useful if p, T and F are within certain bounds. For

example, if $p \approx \frac{\log n}{n}$ and $T \approx \log n$, where n is the number of variables in the target monotone DNF formula, RandomReduce performs clearly better than a different randomized algorithm SimpleReduce that assumes all membership queries to be correct and therefore never unmarks any bits. SimpleReduce can be obtained by randomizing an efficient implementation of subroutine Reduce for the error-free model, i.e., one that marks each 1-bit tried and never unmarks them or tries an already marked bit. When SimpleReduce is first called on a positive point of the target formula (and no bits are marked yet), it sets a random unmarked 1-bit to 0 and asks a membership query about the new point. If the answer is 1, it calls itself recursively, otherwise, it restores the bit to 1, marks it and tries another unmarked 1-bit. When no more unmarked 1-bits remain, it returns the current point. Clearly, SimpleReduce is faster than RandomReduce but it needs to get correct answers about all the points queried, or it will not find the correct minimum point. This happens with probability $(1-p)^{T+F}$, where, as before, T stands for the number of needed 1-bits or variables in the term and F stands for the unwanted 1-bits.

If we take p and T as in the example above, n large, and F close to $n-T$, i.e., almost n, the probability that SimpleReduce reaches the minimum point nearly vanishes, since

$$\lim_{n\to\infty}\left(1-\frac{\log n}{n}\right)^n = 0.$$

However, the probability of complete success for RandomReduce in the above example is not negligible. Suppose we wanted to determine how big should n be before the lower bound on the probability of complete success exceeds some fixed constant l, where $0 < l < 1$:

$$1 - 2pT\left(\ln\left(1+\frac{F-1}{pT+1}\right)+2\right) > l.$$

For this it would suffice if we found c such that $\ln \frac{pT+F}{pT+1} < c$ and that $pT(4+2c) < 1-l$. The first inequality would be satisfied if $F \leqslant e^c$, so we can take $c = \lceil \ln F \rceil \leqslant \ln F + 1$. The second inequality would hold if $pT < \frac{1-l}{6+2\ln n}$. Recalling that $pT \approx \frac{\log^2 n}{n}$, we are interested in finding n big enough to satisfy $(\log^2 n)(6+2\ln n) < n(1-l)$, which is clearly achievable.

To summarize the results derived in this part of the thesis, the algorithm RandomReduce does perform better than the simple algorithm SimpleReduce, which is ideal (in terms of efficiency) for the error-free model. However, direct applicability of RandomReduce as the key subroutine of algorithm LearnMonDNF is questionable, as the lower bound from Theorem 14 is not strong enough. Other algorithms or further modifications to LearnMonDNF are necessary for polynomial-time learning of monotone DNF formulas from equivalence and random membership queries.

Appendix

This appendix describes some of the less obvious constructs of the "C" programming language that are used in the algorithm pseudo-code throughout the thesis. An excellent reference to the language in general is [28].

Comments. Comments in C are put between the symbols `/*` and `*/`. They do not have any effect on the execution of the code statements.

Assignment. The assignment in C is denoted by a single equals sign. For example, $x = y$ assigns the value of variable y to variable x. Assignment also *returns* the value assigned, that is, it may be used as part of a more complicated expression, such as a comparison or another assignment.

Comparisons and Logical Operators. C has six comparison operators and three logical ones for performing boolean operations. In this thesis the relational operators for expressing various inequalities have been replaced with the better looking mathematical symbols $>$, $\geqslant$, $<$, $\leqslant$ and $\neq$, the latter testing whether the operands are different. The equality operator $==$ should be read as "is equal to" and would perhaps look better as a single equals sign but that is already used for assignments.

The logical operators of C have been put in words for readability.

Increment. A very popular C construct is ++. It is the increment operator, for example, x++ increments the value of variable x by one. It has the peculiarity that it returns the *former* value of x but the pseudo-code used in this dissertation does not depend on this feature.

If-Else Statements. The **If** statement evaluates the immediately following parenthesized expression, called the *test condition*, and assigns a *true* or a *false* value to it. If the value is true, the following block in figure braces, called the *if-block*, is executed. If the value of the test condition is false and there is an **Else** statement immediately following the if-block, then the block in figure braces immediately following the **Else** statement is executed. This block is known as the *else-block*. The braces are not required around either block if it consists of a single statement. Only one of the two blocks can be executed as a result of one evaluation of the **If-Else** construct.

While Loops. C has several looping constructs, the simplest of which is **While**. It evaluates the immediately following parenthesized test condition, just like the **If** statement. In case the expression is true, it executes the following block in figure braces, called the *while-block*. Braces may be omitted if the block contains only one statement. After executing the while-block, the control is returned back to the **While** statement, that is, the test condition is re-evaluated and the while-block executed again in case the condition is true. Thus, the **While** loop is reiterated while the test condition stays true. When it becomes false, the while-block is skipped and control is returned to the statement following the block. Note that the variables

involved in the test condition may be modified from inside the while-block as well. This makes statements such as the following possible (although not strictly in C): **While** (unmarked child exists).

For Loops. Another popular looping construct in C is **For**. It is more complicated than **While** since it has separate *initialization*, *test* and *reiteration* statements. These three statements, separated by semicolons and surrounded by parentheses come immediately after the keyword **For**. For example, **For** ($x = 0$; $x < y$; x++). The first statement is executed when the control is passed to the **For** statement. It is the initialization statement. It may happen that the initialization is already done or not needed and this statement may be missing. The semicolon separating it from the next statement must remain, however. After evaluating the initialization statement, the next statement is evaluated. This is the test statement. This statement may also be missing in which case it is considered to be true. The semicolon separating it from the next statement must exist. If the condition is true, the following block of code, enclosed in figure braces and known as the *for-block* is executed. Braces may be omitted if the block contains only one statement. After that, the control is passed back to the **For** statement but not quite the same way. This time the initialization statement is skipped and instead the third statement in parenthesis, the reiteration statement, is executed. This statement may be missing, just like the former two statements. After evaluating this statement, the test statement is evaluated and, if true, the for-block is executed again. A **For** loop is repeated this way as long as the test statement returns a true value. The variables used in initialization, test and reiteration statements may be modified from within the for-block as well.

Some algorithms in this thesis use a modified version of **For** loop that has no

analogies in the C programming language (but resembles **Foreach** of Perl [40]). When the parenthesized expression following the key word **For** does not contain any semicolons the reader should just follow his or her intuition about how this loop is executed. For example, **For** (each child w of v) means that the loop is executed once for each child of v, and that in each iteration the variable w is set to a different child of v. There is no test condition as such in this loop, rather a set consisting of all children of v is created once and w then iterates over this set.

Break and Continue Statements. Additional flexibility in executing looping constructs is achieved by using the **Break** and **Continue** statements. The former statement, **Break**, passes the control to the statement immediately following the innermost loop it is in, i.e., to the statement after the for-block or while-block. Note that this does not apply to if-blocks or else-blocks. Thus, it allows a loop to be terminated early. The latter statement, **Continue**, passes the control immediately to the loop control statement of the innermost loop that it is in, i.e., to the **While** or **For** statement. In case of the **For** statement, the reiteration and test statements, not the initialization statement, are executed. Thus, it allows the loop to be reiterated immediately, bypassing all the statements of the loop block that come after the **Continue**.

Subroutines and the Return Statement. In C all the subroutines are called functions. There is no difference in calling conventions between built-in (i.e., library) functions and user defined functions. When defined, each function must have its name (typeset in SmallCaps font in this thesis), a parenthesized list of its formal parameters (possibly empty, if it does not take parameters), and the body of the function which is a block of statements enclosed in mandatory figure braces. When

called, a function is specified by its name followed by a parenthesized list of its arguments. The values of these arguments are copied into the formal parameters of the function which become its local variables, i.e., they "go out of scope" or disappear once the control is passed out of the function. The control is passed out of the function after the last statement in its body is executed or when a **Return** statement is encountered. A **Return** statement may specify a value to be returned by the function to the calling function. This value is typically used on the right hand side of an assignment operator. For example, $x = \max(y, z)$. The value may be quite complex; returning pairs or arrays is acceptable (at least in the pseudo-code used in this thesis).

Arrays and Multi-Dimensional Arrays The arrays in C are indexed from 0 and square brackets are used to refer to a cell in an array, for example, $Tablerow[s]$. Arrays can be initialized by listing all the values inside figure braces, for example, $step[2] = \{ 0, 0 \}$. In this case the square brackets after the array name and the value (if any) inside them do not refer to a specific cell but instead specify the size of the array.

Multi-dimensional arrays are also allowed, although in reality they are just one dimensional arrays each cell of which is an array of one dimension less having the same cardinalities as those of the arrays contained in other cells. For example, $Table[i][x][s]$ denotes a cell in a three-dimensional array which is found at offset $i(d_2 d_3) + x(d_3) + s$ from its beginning, where d_2 and d_3 denote the cardinalities of the second and third dimension, respectively.

The pseudo-code in this thesis extends arrays and multi-dimensional arrays to allow infinite cardinalities. That is, it uses concepts such as, for example, *infinite*

arrays and *infinite three-dimensional arrays*. Such data structures are unlikely to exist in most of today's programming languages but they serve as useful abstractions in the pseudo-code. It would perhaps be more precise to refer to them as *hash tables* but that would make the pseudo-code less convenient. The reader should not worry about the issue that addressing a cell in a multi-dimensional array does not work in C without knowing exact cardinality of each dimension, an impossible requirement for infinite arrays. If the three coordinates are viewed as parameters to a hash function, the problem does not exist.

Bibliography

[1] H. Aizenstein and L. Pitt. Exact learning of read-twice DNF formulas. In *Proc. 32th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 170–179. IEEE Computer Society Press, 1991.

[2] D. Angluin. Learning k-term DNF formulas using queries and counterexamples. Technical Report YALEU/DCS/RR-559, Department of Computer Science, Yale University, August 1987.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Inform. Comput.*, 75(2):87–106, November 1987.

[4] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, April 1988.

[5] D. Angluin. Negative results for equivalence queries. *Machine Learning*, 5:121–150, 1990.

[6] D. Angluin. Exact learning of μ-DNF formulas with malicious membership queries. Technical Report YALEU/DCS/TR-1020, Yale University Department of Computer Science, March 1994.

[7] D. Angluin, L. Hellerstein, and M. Karpinski. Learning read-once formulas with queries. *J. ACM*, 40:185–210, 1993.

[8] D. Angluin and M. Kriķis. Malicious membership queries and exceptions. Technical Report YALEU/DCS/TR-1019, Yale University Department of Computer Science, March 1994.

[9] D. Angluin and M. Kriķis. Teachers, learners and black boxes. Technical Report YALEU/DCS/TR-1130, Yale University Department of Computer Science, October 1997.

[10] D. Angluin and M. Kriķis. Teachers, learners and black boxes. In *Proc. 10th Annu. Conf. on Comput. Learning Theory*, pages 285–297. ACM Press, New York, NY, 1997.

[11] D. Angluin, M. Kriķis, R. Sloan, and G. Turán. Malicious omissions and errors in answers to membership queries. *Machine Learning*, 28:211–255, 1997.

[12] D. Angluin and M. Kriķis. Learning with malicious membership queries and exceptions. In *Proc. 7th Annu. ACM Workshop on Comput. Learning Theory*, pages 57–66. ACM Press, New York, NY, 1994.

[13] D. Angluin and D. K. Slonim. Randomly fallible teachers: learning monotone DNF with an incomplete membership oracle. *Machine Learning*, 14(1):7–26, 1994.

[14] A. Blum, P. Chalasani, S. A. Goldman, and D. K. Slonim. Learning with unreliable boundary queries. In *Proc. 8th Annu. Conf. on Comput. Learning Theory*, pages 98–107. ACM Press, New York, NY, 1995.

[15] L. Blum and M. Blum. Toward a mathematical theory of inductive inference. *Inform. Control*, 28(2):125–155, June 1975.

[16] R. Board and L. Pitt. On the necessity of Occam algorithms. *Theoret. Comput. Sci.*, 100:157–184, 1992.

[17] N. H. Bshouty. Exact learning via the monotone theory. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 302–311. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[18] W. J. Bultman. *Topics in the Theory of Machine Learning and Neural Computing*. PhD thesis, University of Illinois at Chicago Mathematics Department, 1991.

[19] T. Dean, D. Angluin, K. Basye, S. Engelson, L. Kaelbling, E. Kokkevis, and O. Maron. Learning finite automata with stochastic output functions and an application to map learning. *Machine Learning*, 18(1):81–108, 1995.

[20] M. Frazier, S. Goldman, N. Mishra, and L. Pitt. Learning from a consistently ignorant teacher. In *Proc. 7th Annu. ACM Workshop on Comput. Learning Theory*, pages 328–339. ACM Press, New York, NY, 1994.

[21] R. Freivalds, E. B. Kinber, and R. Wiehagen. Inductive inference from good examples. In *Proceedings of the Second International Workshop on Analogical and Inductive Inference*, pages 1–17, 1989.

[22] S. A. Goldman and M. J. Kearns. On the complexity of teaching. *J. of Comput. Syst. Sci.*, 50(1):20–31, 1995. Earlier version in 4th COLT, 1991.

[23] S. A. Goldman, M. J. Kearns, and R. E. Schapire. Exact identification of read-once formulas using fixed points of amplification functions. *SIAM J. Comput.*, 22(4):705–726, 1993.

[24] S. A. Goldman and H. D. Mathias. Learning k-term DNF formulas with an incomplete membership oracle. In *Proc. 5th Annu. Workshop on Comput. Learning Theory*, pages 77–84. ACM Press, New York, NY, 1992.

[25] S. A. Goldman and H. D. Mathias. Teaching a smarter learner. *J. of Comput. Syst. Sci.*, 52(2):255–267, 1996. Earlier version in 6th COLT, 1993.

[26] J. Jackson and A. Tomkins. A computational model of teaching. In *Proc. 5th Annu. Workshop on Comput. Learning Theory*, pages 319–326. ACM Press, New York, NY, 1992.

[27] M. Kearns. Efficient noise-tolerant learning from statistical queries. In *Proc. 25th Annu. ACM Sympos. Theory Comput.*, pages 392–401. ACM Press, New York, NY, 1993.

[28] B. W. Kernighan and D. M. Ritchie. *The C Programming Language: ANSI C Version.* Prentice-Hall, Inc., 1988.

[29] E. Kushilevitz and Y. Mansour. Learning decision trees using the Fourier spectrum. *SIAM J. Comput.*, 22(6):1331–1348, 1993. Earlier version appeared in STOC 1991.

[30] M. Machtey and P. Young. *An Introduction to the General Theory of Algorithms.* Theory of Computation Series. Elsevier North-Holland, Inc., 1978.

[31] E. Minicozzi. Some natural properties of strong identification in inductive inference. *Theoret. Comput. Sci.*, 2:345–360, 1976.

[32] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Inform. Comput.*, 103(2):299–347, April 1993.

[33] D. Ron and R. Rubinfeld. Learning fallible deterministic finite automata. *Machine Learning*, 18(2/3):149–185, 1995.

[34] Y. Sakakibara. On learning from queries and counterexamples in the presence of noise. *Inform. Proc. Lett.*, 37(5):279–284, March 1991.

[35] A. Shinohara and S. Miyano. Teachability in computational learning. *New Generation Computing*, 8:337–347, 1991.

[36] R. H. Sloan and G. Turán. Learning with queries but incomplete information. In *Proc. 7th Annu. ACM Workshop on Comput. Learning Theory*, pages 237–245. ACM Press, New York, NY, 1994.

[37] C. Smith. *A Recursive Introduction to the Theory of Computation.* Springer-Verlag New York, Inc., 1994.

[38] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, November 1984.

[39] L. G. Valiant. Learning disjunctions of conjunctions. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence, vol. 1*, pages 560–566, Los Angeles, California, 1985. International Joint Committee for Artificial Intelligence.

[40] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., 1996.

[41] Y. Zhuravlev and Y. Kogan. Realization of boolean functions with a small number of zeros by disjunctive normal forms, and related problems. *Soviet Math. Doklady*, 32:771–775, 1985.