

THE UNIVERSITY OF CALGARY

Learning With a Minimal Number of Queries

by

Sleiman Matar

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

AUGUST, 1993

© Sleiman Matar 1993



National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services Branch

Direction des acquisitions et
des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

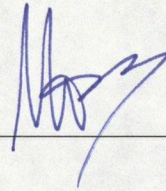
L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-88572-6

Canada

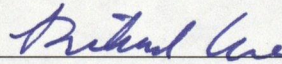
THE UNIVERSITY OF CALGARY
FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Learning With a Minimal Number of Queries" submitted by Sleiman Matar in partial fulfillment of the requirements for the degree of Master of Science.



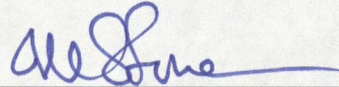
N. H. Bshouty,

Department of Computer Science



R. E. Cleve,

Department of Computer Science



M. G. Stone,

Department of Mathematics and Statistics

Date

6/8/93

Abstract

A number of efficient learning algorithms exactly identify an unknown concept taken from some class using membership and equivalence queries. Using a standard transformation, such algorithms can easily be converted to on-line learning algorithms that use membership queries. Under such a transformation, the number of equivalence queries made by the query algorithm directly corresponds to the number of mistakes made by the on-line algorithm. In this thesis we consider several of the classes known to be learnable in this setting, and investigate the minimum number of equivalence queries with accompanying counterexamples (or, equivalently, the minimum number of mistakes in the on-line model) that can be made by a learning algorithm that makes a polynomial number of membership queries and uses polynomial computation time. We are able both to reduce the number of equivalence queries used by the previous algorithms and often prove matching lower bounds.

Acknowledgments

first and foremost i want to thank you, nader, for being a great supervisor. thank you for your help, for keeping your office door open always, and for your support during my degree. i hope we will collaborate in the future.

i would like to thank vivian, nadera and nader for their warm hospitality and generosity. spending the saturdays with you was always a pleasure to me.

thank you, tino (tamon), for proof reading the thesis, for the squash games, for the cold bananas, for bob marley and santana, and for being there in the office. keep smiling.

i thank also mark james for his valuable help concerning latex problems.

thank you, danny (jaliff), for proof reading parts of the thesis, and for your valuable remarks.

most of the results of this thesis have appeared in a joint paper with nader bshouty, sally goldman and tom hancock, and i want to thank you nader, sally and tom for this collaboration.

i would like also to thank my examiners richard cleve and mike stone for serving in my defense committee and for their remarks.

Contents

Approval Sheet	ii
Abstract	iii
Acknowledgments	iv
Contents	v
List of Tables	viii
List of Figures	ix
Chapter 1. Introduction	1
1.1. Learning models	2
1.2. The oracles from a practical point of view	5
1.3. Our model	7
1.4. Boolean formulas	10
1.5. The goal of this thesis	12
1.6. Literature review and our results	13
Chapter 2. Definitions	19
2.1. Basic definitions	19
2.2. Deterministic finite automata	21
2.3. Measuring the running time	23
2.4. Miscellaneous definitions and facts	24
2.5. Exact learning definitions	24
Chapter 3. A generalization of the halving algorithm	28
3.1. The standard halving algorithm	28
3.2. Generalizing the halving algorithm	29
3.3. Applying the generalized halving algorithm to DNF formulas	32
Chapter 4. k-term DNF	35
4.1. Previous work and of our results	35
4.2. A general algorithm	36
4.2.1. The first improvement	39

4.2.1.1. A parallel greedy algorithm	39
4.2.1.2. Analysis	40
4.2.1.3. Reducing the number of equivalence queries	42
4.2.1.4. Testing the equivalence of two k -term DNF formulas	43
4.2.2. The second improvement	48
4.2.2.1. Analysis	49
4.2.3. Summary	50
4.3. A version of produce-terms based on Blum-Rudich's algorithm	50
4.4. Conclusion	50
Chapter 5. Read-k Sat-j DNF	54
5.1. A learning algorithm	54
5.1.1. An outline of Aizenstein and Pitt's algorithm	54
5.1.2. Our refinement	56
5.2. Number of terms in a Read- k Sat- j DNF	60
5.2.1. The case $k = 1$	60
5.2.2. The case $k > 1$	61
5.3. Lower bounds	64
5.3.1. The case $k > 1$	64
5.3.1.1. The definition of the target class	65
5.3.1.2. The adversary	67
5.3.2. The case $k = 1$	71
5.3.2.1. Definition of the target class C''	71
5.4. Remarks	72
Chapter 6. Monotone DNF Formulas	73
6.1. Lower Bounds	73
The adversary	74
6.2. Upper Bounds	78
6.2.1. Preliminaries	78
6.2.2. Angluin's algorithm	80
6.2.3. Our refinement	80
Chapter 7. Horn Sentences	84
7.1. Lower Bound	84
Chapter 8. Boolean read-once formulas over various bases	91
8.1. Generating justifying assignments with a minimal number of equivalence queries	92
8.1.1. Definitions	92
8.1.2. The standard transformation	93
8.1.3. Our refinement	94
Chapter 9. Arithmetic read-once formulas	98

Chapter 10. Deterministic Finite State Automaton	100
10.1. Lower bound	100
Chapter 11. Open Problems	103
Appendix A. k-term DNF formulas, continued	105
A.1. Angluin's algorithm and our refinement	105
A.1.1. An outline of Angluin's algorithm	106
A.1.2. Our refinement	107
A.2. Summary	108
Bibliography	110

List of Tables

1.1	Summary of the number of equivalence queries in our results	17
1.2	Summary of the number of membership queries of this thesis's results .	18
4.1	The results obtained when using produce-terms	51
4.2	Summary of the results when using produce-terms based on Blum and Rudich's algorithm	53
A.1	The results obtained when using produce-terms based on Angluin's algorithm	109

List of Figures

1.1	The relationships among the different models	6
1.2	The formula $f(m) = m^2$ for $m \in \{1, 2, 3, 4, 5\}$	10
1.3	Five boolean formulas that represent the formula $f(m) = m^2$ for $m \in \{1, 2, 3, 4, 5\}$	11
2.1	An example of a transition function	22
2.2	A graphical representation of the automaton in example 2.2	23
3.1	The standard halving algorithm	29
3.2	A generalization of the halving algorithm	30
4.1	Learning k -term DNF	37
4.2	The FORK macro	40
4.3	First improvement of learn1- k -term-dnf	41
4.4	An algorithm to test if a k -term DNF formula is a tautology	46
4.5	An algorithm to test if two k -terms formulas are logically equivalent. ..	46
4.6	The second improvement of learn1- k -term-dnf	49
5.1	Learning Read- k Sat- j DNF	56
6.1	Our refinement for learning the class of monotone DNF formulas	81
8.1	Generating justifying assignments using $n/\log n$ equivalence queries ...	96

CHAPTER 1

Introduction

One recognizes a particular concept over some domain if one has memorized a specific description for every concept in the domain. However, a human being is able also to identify a concept even if there is no explicit procedure in mind for doing so, or even if the procedure is incomplete or contains too many specifications that do not apply for that particular concept. We call the latter method for identifying concepts “learning”. It would be helpful to formalize some learning methods. If we succeed in doing this, we can install a formal description of these methods in a machine and expect the machine to accomplish things “skillfully”.

In this thesis we are interested in finding *efficient* methods for learning concepts. The following scenario informally describes the setting of the problems addressed in this thesis. In the setting there is a class of concepts (referred to by the *target class*), e.g. the set of all songs, the set of all chairs, etc. The concepts are defined over a *domain*, which is the set of all instances each of which is an assignment of values to the relevant attributes. There is also a data base that describes a specific concept from the class (we refer to this concept by the *target concept*). Our problem, as researchers in learning, is to write an algorithm that accesses the data base and after a *reasonable* length of time halts and outputs a description of the target concept. We refer to this algorithm by the *learning algorithm* or the *learner*. If we manage to write a learning algorithm for a class of concepts, then we say that this class is *learnable*.

As an example, consider a robot moving in a building. We are interested in “teach-

ing” the robot to identify the room it is in at the moment. The concepts that we have here are rooms, the target concept is the room in which the robot is, and the data base that describes the target concept is the properties of the room as obtained via the robot’s sensors. Our task, as the people who are responsible for the robot, is to install in the robot a program that monitors the environment using the robot’s sensors, and finds in a short time a unique description of the room.

Several questions arise here. What are the resources of the learning algorithm? More specifically, how do we formalize the data base that describes the target concept? What is the time and space allotted for the program? Also, are we asking the algorithm to exactly identify the target concept or just to approximate it? Finally, an important question is: for which classes of concepts can we find learning algorithms?

1.1. Learning models

The data base is represented by *oracles*. Each oracle answers one particular type of query. The learning algorithm makes a query to some oracle, and receives in the following time unit the answer for the query (note that we ignore here the time that the oracle requires to find the answer, which justifies the name “oracle”).

In his seminal paper [29], Valiant introduced the *EXAMPLES* oracle. This oracles does not get an input, but rather has a button with the following functionality: When the learning algorithm presses the button, *EXAMPLES* outputs an instance from the concepts’ domain, correctly classified as a positive or negative example of the target concept. Continuing with the robot example, suppose the relevant attributes in the domain are the color of the room, whether or not there is a piano in it, and whether or not it has windows. The *EXAMPLES* oracle may output the following instances together with their correct classification:

$\langle \text{room-is-red, room-has-windows, there-is-piano; no} \rangle,$

or

(room-is-green, room-has-windows, there-is-no-piano; yes).

At every request, the EXAMPLES oracle outputs an instance (together with its classification) according to some unknown probability distribution \mathcal{D} over all instances in the domain. That is, if the \mathcal{D} fixes a probability α on some instance then this instance will be output by the oracle with probability α , independently from previous requests. This setting of learning (i.e. only the EXAMPLES oracle provides information about the target concept) is called the *probably approximately correct (PAC) model*. In this model, The learning algorithm *always* halts after a polynomial time, and it outputs a description of a concept that, with *high* probability, is a *good* approximation to the target concept. The learning algorithm is given a parameter ϵ that determines the degree of approximation and in what probability the output concept is a good approximation. The running time typically depends on the number of attributes in the domain and on the ϵ parameter.

Observe that in the PAC model, the learning algorithm is completely passive. It just monitors the environment (the EXAMPLES's outputs) without being able to ask for the classification of some specific instance. Another oracle that allows the learner some activity is the *membership* oracle. Here, the learner provides the oracle with an instance, and the oracle responds by classifying the instance as a positive or negative example of the target concept. The PAC model was proven to be weak, in the sense that the setting it provides is not powerful enough for learning several “basic” classes of concepts. Adding the membership oracle to the setting of the PAC model was proven to be helpful. Many classes of concepts that are not learnable using the EXAMPLES oracle only, are learnable using both the EXAMPLES and the membership oracles.

Another well-studied oracle in the literature is the *equivalence* oracle. The learner supplies the oracle with a conjectured concept, and is told “yes” if the given concept is the target one, or “no” otherwise. Moreover, if the answer is “no” the oracle

outputs a *counterexample*, that is an instance that is misclassified by the conjectured concept. The learning setting in which a membership oracle and an equivalence oracle are provided is called the *minimally adequate teacher model* and it was developed by Angluin [5]. In this model the learner is required to *exactly identify* the target concept. This model is more powerful than the PAC model, in the sense that several classes of concepts that were proven to be unlearnable in the PAC model are learnable in the minimally adequate teacher model. There are two versions for the minimally adequate teacher model. The first is the *restricted* model. In this model the concepts given to the equivalence oracle must be from the target class. Moreover, the concept that the learner outputs must be also from the target class. The other version is the *unrestricted* model, in which the concepts given as input to the equivalence oracle and the output concept need not be in the target class. Obviously, if a class is learnable in the restricted model, it is learnable in the unrestricted model. The converse is not true. In this thesis we consider both versions, but most of the learning algorithms given here are in the restricted model.

One other model discussed in this introduction is the *on-line learning model* developed by Littlestone [24]. In the on-line learning model the learning session is divided into a sequence of trials, in each of which the learner is given an instance and is asked to predict — according to its conjecture — how the target concept classifies the given instance. After the prediction is made, the learner is told whether the prediction is correct and can then use a *polynomial* number of operations (designed to refine its conjecture) before proceeding to the next trial. When analyzing a learning algorithm in this model we consider the running time of the trial, and the number of mistakes (and ignore the number of instances given to the learner). More specifically, both must be polynomial in the number of attributes and in the complexities of the target formula.

In the literature there are other oracles, but in this introduction we consider only the ones mentioned above. We are interested also in the relationships among the

different models. Angluin [5, 4] showed that an equivalence query can be simulated in the PAC model. Littlestone [24] showed that a learning algorithm that uses equivalence queries can be transformed into an on-line algorithm. He has also shown that an algorithm in the on-line model can be transformed into a learning algorithm in Angluin’s unrestricted model. Observe also that if a class of concepts is learnable in some model, adding a new oracle to the model will not weaken it, so the class of concepts is also learnable in the new model. The diagram in figure 1.1 shows the relationships among the various models. The diagram is complete: if there are no arrows between a model M and a model M' then M and M' are not comparable. This means that there is a class of concepts that is learnable in one of them but not in the other, and vice versa.

1.2. The oracles from a practical point of view

We now give an intuition behind the defined oracles.

The intuition behind the EXAMPLES and membership oracles is fairly straightforward. The EXAMPLES oracle formalizes the experiment of the learner monitoring the environment while being completely passive. The membership query is a formalization of the experiment in which the learner is given an active choice in formulating queries. The following is a good example to help understand the intuition behind the membership oracle and the EXAMPLES oracle. In emergency situations a particular factory is faced with the dangerous task of extinguishing fires in labs that contain explosive materials. The task can be performed by a human expert; however, the factory management does not want to endanger its workers, so it decided to buy a machine to perform the task. The management is faced with the problem of how to “train” the machine to do the work. The purchased machine has a learning algorithm installed. That is, the machine makes queries and it accordingly updates the description it has of how to handle the task. It has sensors that monitor the environment. A call or prompt to the EXAMPLES oracle in this setting would be observing the

Angluin [5] has shown that one can simulate an equivalence query by asking *not too many* examples from the EXAMPLES oracle. This, on its own, is an intuition behind using the equivalence query. Continuing with our example, and using Angluin’s transformation, suppose the learner (the machine) has completed a good description of the task, and it makes an equivalence query (i.e. the learner has a conjecture and it wants to test it). In order to do this, the learner observes the expert performing the task. If the expert’s behavior does not fit the learner’s conjecture, then the learner has a counterexample to its conjecture. Otherwise, the learner makes further queries (for a certain time). The practical point of view of the on-line model is quite similar to this of the equivalence oracle. After having completed some conjecture (i.e. finished a trial) the learner observes and monitors the environment, and predicts how the expert will handle the situation. The learner, after that, observes the expert’s behavior in order to know whether the prediction is correct or not.

1.3. Our model

Our model is a variation of the minimally adequate teacher model, that is, the oracles available to the learner are the membership and equivalence oracles. Previous work generally made the assumption that both membership queries and equivalence queries have an equivalent cost to the learner (namely, a constant cost). Thus there was no reason to favor one type of query over the other. In reality, one type of query is often significantly less expensive to implement. In particular, we are interested in learning problems in which membership queries are relatively inexpensive to perform (i.e. a simple experiment that can be run by the learner) whereas equivalence queries are expensive (i.e. require a “teacher’s” supervision to provide a counterexample). If we examine the complexity of the learning algorithm under the on-line learning model, then reducing the number of equivalence queries directly corresponds to minimizing the number of prediction mistakes in the on-line algorithm [24].

We further describe our motivation for reducing the number of equivalence queries needed to obtain exact identification. In this work we are able to reduce (sometimes quite dramatically) the number of equivalence queries needed to obtain exact identification at the expense of increasing the computation time and number of membership queries (although they remain polynomial in the number of attributes and in the complexities of the target concept.) Being able to prove tight bounds on the number of equivalence queries needed for exact identification is, clearly, of great theoretical interest. We now argue that it is also of practical interest.

As one example, consider the situation in which the target concept f measures some observable consequence of the learner's action. For example, Rivest and Schapire [28] motivate the problem of learning an unknown deterministic finite automaton by considering the problem of a robot trying to learn to navigate in an environment described by a finite state machine. Here a membership query represents experimentation by the robot, followed by an observation of its perceived state after executing the experiment. Thus, in this context, membership queries can be made in an unsupervised manner by the learner interacting with the environment. On the other hand, an equivalence query requires the intervention of a teacher to provide a counterexample. For example, there are some states that the robot can reach only through specific sequences of actions that it cannot stumble on through its own experimentation, and of which it must be told by a teacher. In such settings, minimizing the number of equivalence queries allows the learner to minimize the supervision needed.

Another source of motivation comes from the goal of minimizing the number of prediction mistakes in an on-line (or incremental) learning model. As we have mentioned, the model of learning with membership and equivalence queries is essentially equivalent to the on-line learning model when the learner is provided with membership queries [5, 24]. The conversion of an algorithm A that uses membership queries and equivalence queries to an on-line algorithm A' works as follows. If algorithm A wants to perform some internal computation or perform a membership query then

algorithm A' will perform the same task. If A wants to make an equivalence query with hypothesis h , then A' can just use hypothesis h to make predictions. If hypothesis h is equivalent to the target no mistakes will occur and the learning session is done. Otherwise, if algorithm A' makes a prediction mistake on instance x , then this instance can be passed to A as a counterexample. Thus, the number of mistakes made by A' is just one less than the total number of equivalence queries made by A . Since the primary goal of an on-line learning algorithm is to reduce the number of mistakes, the learner would be willing to spend additional computation time and make additional membership queries to reduce the number of mistakes. Thus minimizing the number of equivalence queries is equivalent to minimizing prediction errors in an on-line learning model.

Another situation in which we would like to minimize equivalence queries is the case where the learning algorithm is being used to interpolate a function that is in fact already known in some sense (e.g. we have a black box oracle, or truth table) to obtain some desired representation (e.g. a read-once formula or equivalently a circuit of fan-out 1). In this situation, membership queries may be readily implementable as substitutions, yet implementing an equivalence query may be much more expensive.

The last source of motivation we mention is related to learning in parallel. Bshouty and Cleve [15] have established lower bounds on the number of membership and equivalence queries needed to learn some classes of concepts, when the learner is given an unlimited power of computation (that is, we do not count steps that do not involve calling the oracles). Let \mathcal{C} be the target class of concepts. They establish the following relation:

$$ME(p) \geq \mathcal{E}_U(\mathcal{C}, \lceil p \log |\mathcal{C}| \rceil),$$

where $ME(p)$ is the number of membership and equivalence queries needed to learn \mathcal{C} using p processors, and $\mathcal{E}_U(\mathcal{C}, \lceil p \log |\mathcal{C}| \rceil)$ is the number of equivalence queries needed to learn \mathcal{C} sequentially when number of membership queries used is $\lceil p \log |\mathcal{C}| \rceil$ (in both settings the learner is given unlimited computational power). Therefore, the

m	$f(m)$
0 0 1	0 0 0 0 1
0 1 0	0 0 1 0 0
0 1 1	0 1 0 0 1
1 0 0	1 0 0 0 0
1 0 1	1 1 0 0 1

FIGURE 1.2. The formula $f(m) = m^2$ for $m \in \{1, 2, 3, 4, 5\}$, when both the output and the input is represented in binary.

lower bounds of this thesis on the number of equivalence queries, can be used in the above relation to get a lower bound for parallel learning.

1.4. Boolean formulas

Most of the classes considered in this thesis are subclasses of boolean formulas. We now argue that we do not lose any generality by this restriction. The reason is that both the input and the output of any formula (over a finite domain) can be represented in binary. More specifically, suppose the output of the formula is represented in binary using ℓ bits. This defines ℓ boolean formulas, one formula for every bit in the output. We give an example to illustrate the concept.

Example: Suppose we have the formula $f(m) = m^2$, for $m \in \{1, 2, 3, 4, 5\}$. We construct a set of boolean formulas that represent the formula f . The table in figure 1.2 shows f when both the input and the output of f are represented in binary. We can now define five boolean formulas f_1, f_2, f_3, f_4 and f_5 , where the formula $f_i(m)$ (m represented in binary) gives the i th bit, from left, of $f(m)$. Figure 1.3 shows these formulas.

Representing formulas by a set of boolean formulas raises the interesting question of: how does the boolean representation of the non-boolean formula change the length of the representation? The issue, however, is beyond the scope of this thesis.

m	$f_1(m)$	$f_2(m)$	$f_3(m)$	$f_4(m)$	$f_5(m)$
0 0 1	0	0	0	0	1
0 1 0	0	0	1	0	0
0 1 1	0	1	0	0	1
1 0 0	1	0	0	0	0
1 0 1	1	1	0	0	1

FIGURE 1.3. Five boolean formulas that represent the formula $f(m) = m^2$ for $m \in \{1, 2, 3, 4, 5\}$.

Most of the classes considered here are subclasses of DNF formulas. Suppose the set of variables over which the formulas are defined is $V = \{v_1, \dots, v_n\}$. We denote the negation of a variable v by \bar{v} . Also, \wedge denotes the operator AND, and \vee denotes the operator OR, so $v \vee u$ and $v \wedge u$ denote v OR u and v AND u , respectively. The set of *literals* associated with V is $\{v_1, \dots, v_n, \bar{v}_1, \dots, \bar{v}_n\}$. A *term* is a conjunction of literals, e.g. $v_3, \bar{v}_1 \wedge v_5 \wedge v_2$ and $v_2 \wedge v_4 \wedge \bar{v}_1$ are all terms. A formula is in disjunctive normal form (DNF) if it is a disjunction of terms, for example,

$$v_3 \vee (\bar{v}_1 \wedge v_5 \wedge v_2) \vee (v_2 \wedge v_4 \wedge \bar{v}_1)$$

is a DNF formula. A *clause* is a disjunction of literals. e.g. $\bar{v}_7 \vee \bar{v}_9 \vee v_3$, and a formula is in conjunctive normal form (CNF) if it is a conjunction of clauses, e.g.

$$(\bar{v}_1 \vee v_2) \wedge (\bar{v}_2 \vee v_3 \vee \bar{v}_4).$$

Every boolean function can be represented as both a DNF formula and a CNF formula. For example, the above CNF formula can be represented by the following DNF formula

$$(\bar{v}_1 \wedge \bar{v}_2) \vee (\bar{v}_1 \wedge v_3) \vee (\bar{v}_1 \wedge \bar{v}_4) \vee (v_2 \wedge v_3) \vee (v_2 \wedge \bar{v}_4).$$

1.5. The goal of this thesis

Within the minimally adequate teacher model a number of interesting polynomial time algorithms have been presented to learn target classes such as deterministic finite automata [4], Horn sentences [7], read-once formulas [8, 16, 17], k -term DNF formulas [12], etc. (these classes will be defined later.) It is easy to show that membership queries alone are not sufficient for efficient learning of these classes. Angluin developed a technique called “approximate fingerprints” to show that equivalence queries alone are also not enough [6]. (In both cases the arguments are information theoretic, and hold even when only the number of queries, and not the computation time, is bounded.) Our research extends Angluin’s results to establish tight bounds on how many equivalence queries are required for a number of these classes.

Thus, the goal of our work is to establish tight bounds on how many equivalence queries are required. Unless otherwise stated, in all upper bounds we restrict the learner to an amount of time and a number of membership queries that are both polynomial in the the number of attributes (or variables) in the domain and in complexities of the target concept. However, all of our lower bounds place no restrictions on the computation time of the learning algorithm.

Maass and Turán have also studied upper and lower bounds on the number of equivalence queries required for learning, both with and without membership queries [25]. However, they count only the total number of queries rather than the individual number of queries of each type.

1.6. Literature review and our results

The *halving algorithm* [11, 24, 5] is an algorithm that learns any class \mathcal{C} of concepts using $\log |\mathcal{C}|$ equivalence queries and unlimited computational power. In chapter 3 we present the standard halving algorithm and we develop a generalization of it by giving an algorithm that uses both membership queries and equivalence queries, and unlimited computational power. We are able to reduce the number of equivalence queries to $\frac{\log |\mathcal{C}|}{d \log \log |\mathcal{C}|}$ by using $\frac{\log^{d+1} |\mathcal{C}|}{\log e}$ membership queries (for any constant $d \geq 1$).

In this thesis we establish several lower bounds on the number of equivalence queries required for learning some classes of concepts (some lower bounds are cited from the paper by Bshouty and Cleve [15]). *All* these lower bounds hold even if the concepts given to the equivalence oracle are *any* concepts, and provided that the learner is given unlimited computational power.

Learning the class of disjunctive normal form (DNF) boolean formulas using membership queries and equivalence queries is still an open problem, and intensive research is devoted to learning various subclasses of DNF. The class of k -term DNF formulas is the class of DNF formulas in which there are at most k terms. Blum and Rudich [12] have given an algorithm that learns the class of k -term DNF formulas in

the unrestricted model in time¹ $2^{O(k)}n(\log n)^{O(1)}$ using $2^{O(k)}n(\log n)^{O(1)}$ equivalence and membership queries. We improve their results by giving two new algorithms. The first is a learning algorithm in the unrestricted model that learns the class of k -term DNF formulas in time $2^{O(k)}n^{O(1)}(\log n)^{O(1)}$ using $2^{O(k)}n^{O(1)}(\log n)^{O(1)}$ membership queries and $k + 1$ equivalence queries (if k is known to the learner *a priori*, then the number of equivalence queries is k). Note that these complexities are polynomial in n for $k = O(\log n)$. The second algorithm is in the restricted model. Its running time and the number of membership queries it uses are $2^{O(k^2)}n^{O(1)}(\log n)^{O(k)}$. The number of equivalence queries is again $k + 1$ (here, again, if k is known to the learner *a priori*, then the number of equivalence queries is k). These complexities are polynomial in n for $k = O(\sqrt{\log n})$. By the results of Bshouty and Cleve [15] one cannot reduce the number of equivalence queries. That is, if k is not known, the learner must ask at least $k + 1$ equivalence queries, and if it is known, the learner must ask at least k equivalence queries.

A DNF formula is *Read- k Sat- j* if every variable appears in it at most k times, and every assignment satisfies at most j terms in it. Aizenstein and Pitt [2] have described an algorithm that learns the class of *Read- k Sat- j* DNF formulas in time $n^{O(kj)}$. The number of equivalence queries that their algorithm uses is also $n^{O(kj)}$. In this thesis we are able to drop the number of equivalence queries used to learn the class of *Read- k Sat- j* DNF formulas to $m + 1$, where m is the number of terms in the target formula (if m is known *a priori*, then the number of equivalence queries is m). This is an improvement over their algorithm since we show in section 5.2 that m cannot exceed $4\sqrt{jk(k-1)n}$. We also show in section 5.3 matching lower bounds on the number of equivalence queries.

A DNF formula is *monotone* if it does not contain negations. Angluin [5] has given an algorithm that learns the class of monotone DNF formulas using $m + 1$ equivalence

¹Henceforth, unless otherwise stated, n refers to the number of variables. We refer the reader to section 2.3 for formal definitions of the notations $O()$, $\Omega()$, and $\Theta()$.

queries, where m is the number of terms in the target formula. In this thesis, we find an algorithm that uses $m - \Theta\left(\frac{\log m + \log n}{\log n - \log \log m}\right)$ equivalence queries, provided that m is known *a priori*, and we prove one cannot do better than this. If m is not known *a priori* then we exhibit an algorithm that uses $m - c$ equivalence queries, for any given constant c . This is tight for m superpolynomial in n .

The Horn sentences are a special case of the conjunctive normal form (CNF) formulas. Every clause in a Horn sentence is either of the form $\text{true} \rightarrow v$ or of the form $v_{i_1} \wedge v_{i_2} \wedge \cdots \wedge v_{i_j} \rightarrow l$, where the v 's are variables, and l can be either a variable, false or true. Angluin, Frazier and Pitt [7] have shown that the class of Horn sentences is learnable using $m(2n + 1)$ equivalence queries, where m is the number of clauses in the target formula. We find a lower bound of $\Omega\left(\frac{mn}{\log m + \log n}\right)$ on the number of equivalence queries.

A formula is Read-Once if every variable appears in it at most once. The class of boolean Read-Once formulas over the operators AND, OR and NOT, was proved to be learnable by Angluin, Hellerstein and Karpinski [8]. The number of equivalence queries use by their algorithm is $O(n)$. Since then Hancock and Hellerstein [21], Bshouty, Hancock and Hellerstein [17] and Bshouty, Hancock, Hellerstein and Karpinski [18] have generalized the algorithm to learn Read-Once formulas over various bases. Bshouty, Hancock and Hellerstein [16] have also shown an algorithm for learning Arithmetic Read-Once formulas over the basis $\{+, -, \times, \div\}$ over a field \mathcal{F} . In this thesis we show how to reduce the number of equivalence queries in the boolean case to $O\left(\frac{n}{\log n}\right)$, and to $O\left(\frac{n \log |\mathcal{F}|}{\log n}\right)$ in the arithmetic case. In both cases, these bounds are tight, by the work Bshouty and Cleve [15].

Angluin [4] has exhibited an algorithm that learns the class of deterministic finite automata (DFA) representing regular languages over some alphabet Σ . The algorithm she explores uses $n - 1$ equivalence queries where n is the number of states in the target automaton. We prove here a lower bound of $\Omega\left(\frac{n}{\log n}\right)$ on the number of equivalence queries.

The number of equivalence queries of the results established in this thesis are shown in table 1.1. As stated above, reducing the number of equivalence queries is on the expense of using an additional polynomial number of membership queries. Table 1.2 shows the number of membership queries used in the learning algorithms that are shown in table 1.1.

Representation Class	Previous Upper bounds	Upper Bound	Lower Bound
k -term DNF ($k=O(\log n)$)* k not known k known	$n2^{O(k)} \log n$ [12]	$k+1$ k	$k+1$ [15] k [15]
Read- k Sat- j DNF m not known m known	$n^{O(kj)}$ [2]	$m+1$ m	$m+1$ m ‡
Monotone DNF m not known m known	$m+1$ [5]	$m - \Theta(1)$ $m - \Theta\left(\frac{\log m + \log n}{\log n - \log \log m}\right)$	$m - \omega(1)$ $m - \Theta\left(\frac{\log m + \log n}{\log n - \log \log m}\right)$ §
Horn Sentences m not known m known	$m(2n+1)$ [7]	$O\left(\frac{mn}{\log m + \log n}\right)$ †	$\Omega\left(\frac{mn}{\log m + \log n}\right)$
Read-once formulas over \vee, \wedge, \neg over B_k	n [8] n [17]	$O\left(\frac{n}{\log n}\right)$ $O\left(\frac{n}{\log n}\right)$	$\Omega\left(\frac{n}{\log n}\right)$ [15] $\Omega\left(\frac{n}{\log n}\right)$ [15]
Arithmetic read-once formulas for $ \mathcal{F} = o(n/\log n)$	n [16]	$O\left(\frac{n \log \mathcal{F} }{\log n}\right)$	$\Omega\left(\frac{n \log \mathcal{F} }{\log n}\right)$ [15]
DFA with n states n not known n known	$n-1$ [4, 28]		$\Omega\left(\frac{n}{\log n}\right)$

*For $k = O(\sqrt{\log n})$ we can obtain this result using k -term DNF formulas as the hypotheses for the equivalence queries. For the remaining cases, general DNF formulas are used for the equivalence queries.

†With unlimited computation, for $m \leq 2^{\epsilon n}$, $\epsilon < 1/2$. Furthermore, we note that both this upper bound and the matching lower bound hold for arbitrary DNF formulas.

‡This bound holds for $k > 1$ provided that $jk = o\left(\frac{n}{(\log n)^2}\right)$. For $k = 1$, we have the lower bounds for learning j -term DNF.

§This lower bound holds for $m \leq 2^{\sqrt{n}}$.

TABLE 1.1. This table summarizes the number of equivalence queries in our results. All lower bounds allow the learning algorithm to use unbounded computation time and to propose any hypothesis. Unless stated otherwise, all upper bounds are for algorithms that use polynomial computation time. For the Boolean classes n is the number of variables and m is the number of terms/clauses. For k -term DNF, k is the number of terms. Note that all upper bounds for an unknown size parameter can be used when the size parameter is known. Likewise, all lower bounds for a known size parameter apply when the size parameter is unknown.

Representation Class	Previous Upper bounds	Our Upper Bounds
k -term DNF		
Unrestricted model	$n^2 2^{O(k)} \log n$ [12]	$n^2 2^{O(k)} \log n$
Restricted model	n^{k^2} [3]	$n^2 2^{O(k^2)} (\log n)^{O(k)}$
Read- k Sat- j DNF	$O(kn^{j+2})$ [2]	$O(n^{4kj+2j})$
Monotone DNF	$n(m+1)$ [5]	$\text{poly}(m, n)$ [†]
Horn Sentences	$O(m^2 n)$ [7]	$O(mn)$ [*]
Read-once formulas		
over \vee, \wedge, \neg	$O(n^3)$ [8]	$O(n^5)$
over B_k	$O(n^{k+3} + n^6)$ [17]	$O(n^{k+5} + n^8)$
Arithmetic read-once formulas for $ \mathcal{F} = o(n/\log n)$	$O(n^3)$ [16]	$O(n^5)$

^{*}Using unlimited computational power.

[†]There is a tradeoff between the degree of this polynomial and the number of equivalence queries in our upper bound: the larger the constant in the $\Theta()$ expression in the corresponding entry of table 1.1, the larger the degree of this polynomial.

TABLE 1.2. *This table summarizes the number of membership queries in our results.*

CHAPTER 2

Definitions

2.1. Basic definitions

Let \mathcal{C} be a class of concepts each mapping a domain \mathcal{X} into a range \mathcal{Y} . For most of classes studied here \mathcal{C} is some set of boolean formulas, n is the number of variables, $\mathcal{X} = \{0, 1\}^n$ and $\mathcal{Y} = \{0, 1\}$. We also use the term *assignment* to denote an instance in \mathcal{X} . We assume that the n variables are v_1, v_2, \dots, v_n where the value of v_i is given by the i th bit of the assignment. We use the bar over a variable to denote the operator NOT applied to the variable, so \bar{v} denotes “NOT v ”, and we call it also the *negation* of v , or v *negated*. A *literal* is either a variable or its negation, thus, the set of literals is $\{v_1, \dots, v_n, \bar{v}_1, \dots, \bar{v}_n\}$. We say that a variable v *appears negatively* in a formula f if the literal \bar{v} appears in f . Dually, v *appears positively* in f if f contains the literal v (obviously, a variable can appear both negatively and positively in a formula). We use \vee and \wedge to denote the operators OR and AND, respectively. A boolean formula is said to be *monotone* if it can be represented using only AND/OR gates (with no negations). A *term* is a conjunction of literals, so an assignment satisfies a term if and only if it satisfies every literal in it. A formula is in disjunctive normal form (DNF) if it is written as the disjunction of terms. We often use $v_1 \cdots v_k$ to denote the term $v_1 \wedge \cdots \wedge v_k$. A *clause* is a disjunction of literals. A boolean formula is in conjunctive normal form (CNF) if it is a conjunction of clauses.

Example: *Let*

$$f(v_1, v_2, v_3, v_4, v_5) = (\bar{v}_2 \wedge \bar{v}_3 \wedge v_1) \vee (v_4 \wedge \bar{v}_5) \vee (v_1 \wedge v_2 \wedge \bar{v}_5 \wedge \bar{v}_4).$$

Each of $\bar{v}_2 \wedge \bar{v}_3 \wedge v_1$, $v_4 \wedge \bar{v}_5$, and $v_1 \wedge v_2 \wedge \bar{v}_5 \wedge \bar{v}_4$ is a term. The formula f is a DNF formula, but it is not monotone, because it contains negated variables. Another way to write f is the following

$$f(v_1, v_2, v_3, v_4, v_5) = \bar{v}_2 \bar{v}_3 v_1 \vee v_4 \bar{v}_5 \vee v_1 v_2 \bar{v}_5 \bar{v}_4.$$

Henceforth, we deal with definitions that are specific for concepts that are formulas, so we will use the term “formula” rather than “concept”.

If for a boolean formula f and an instance $x \in \mathcal{X}$ $f(x) = 1$, we say that f is *true on x* , and that x *satisfies f* . If $f(x) = 0$ we say that f is *false on this instance*, and that x *falsifies f* . For $x \in \mathcal{X}$ and variable v , we use $x[v]$ to denote the value assigned to v by x . We also use x_i and $x[i]$, for $1 \leq i \leq n$, to denote $x[v_i]$. Let ℓ be a literal that corresponds to the variable v_i , that is ℓ is either v_i or \bar{v}_i . Then we define $x[\ell]$ as follows

$$x[\ell] = \begin{cases} x[i] & \text{if } \ell \text{ is } v_i \\ \overline{x[i]} & \text{if } \ell \text{ is } \bar{v}_i. \end{cases}$$

We sometimes let the words *true* and *false* denote the constants 1 and 0, respectively.

Several algorithms in this thesis involve looking at a *projection* of a formula. Informally, a projection is obtained from a formula f by fixing some variables in f to some constants. Formally, let p be a vector of length n whose entries are from the set

$\{0, 1, *\}$. For an assignment $x \in \mathcal{X}$, we define the assignment $p|x$ by

$$(p|x)[i] = \begin{cases} x[i] & \text{if } p[i] = *, \\ p[i] & \text{otherwise.} \end{cases}$$

The vector p is called a *partial assignment*, and it induces a projection. Informally, the $*$'s in p indicate the variables that are not fixed. For a partial assignment p and a formula f , we define the projection f_p as follows: for every $x \in \mathcal{X}$, $f_p(x) = f(p|x)$.

2.2. Deterministic finite automata

The principal non-Boolean class considered here is the class of deterministic finite automata (DFA). In this case n is the number of states in the target DFA, \mathcal{X} consists of all strings from the given alphabet and \mathcal{Y} is $\{0, 1\}$. A DFA is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$, where

Q is a finite set of *states*,

Σ is a finite set of symbols, called the *alphabet*,

$q_0 \in Q$ is the *initial state*,

$F \subseteq Q$ is the set of *final states*,

and δ is the *transition function*, a function from $Q \times \Sigma$ to Q . In addition, the automaton reads an *input* from the *input tape*. The input is some string from the set of all finite strings over Σ (this set is denoted by Σ^*). The automaton M is initially in its initial state. The rules according to which M picks its next state are encoded into the transition function. Thus, if M is in state $q \in Q$ and the symbol read from the input tape is $\sigma \in \Sigma$, then $\delta(q, \sigma) \in Q$ is the uniquely determined state to which M passes. The definition of δ is extended to the domain $Q \times \Sigma^*$ in the standard way, that is $\delta(q, \lambda) = q$, where λ is the empty string, and for every $\sigma \in \Sigma$ and for every $w \in \Sigma^*$, $\delta(q, \sigma w) = \delta(\delta(q, \sigma), w)$. There is a convenient way to represent a DFA, which is the graphical representation. A directed graph $G(V, E)$ represents the automaton $M = (Q, \Sigma, \delta, q_0, F)$, if it has the following properties

q	σ	$\delta(q, \sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_1
q_1	b	q_0

FIGURE 2.1. The transition function of the automaton in example 2.2.

- (1) The set of vertices V is the set of states Q in M .
- (2) An edge $(u, v) \in E$ if and only if there is $\sigma \in \Sigma$ such that $\delta(u, \sigma) = v$. The edge (u, v) will be then labeled with σ .
- (3) The vertex that corresponds to the initial state q_0 has a special sign to designate it from all other vertices. For example, it can be designated by drawing an arrow pointing to it.
- (4) The final vertices are designated with a special sign.

Example: As an example, consider the automaton $M = (Q, \Sigma, \delta, q_0, F)$, where

$$Q = \{q_0, q_1\},$$

$$\Sigma = \{a, b\},$$

$$F = \{q_0\},$$

and the transition function δ is given by the table in figure 2.1. The graph in figure 2.2 is a graphical representation of the above automaton.

We say that an automaton $M = (Q, \Sigma, \delta, q_0, F)$ accepts a string $w \in \Sigma^*$ if $\delta(q_0, w)$ is a final state. Continuing with our example, the automaton in figure 2.2 accepts the empty string λ . It also accepts the string a , aa , and $abab$. The reader can verify that the automaton in figure 2.2 accepts a string w if and only if the number of b 's in w is even.

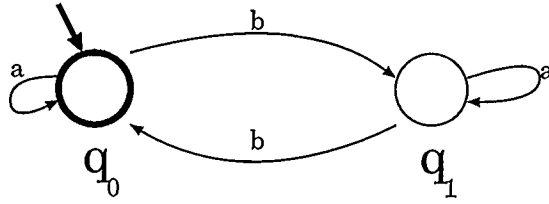


FIGURE 2.2. A graphical representation of the automaton in example 2.2. q_0 is the initial state, and this is denoted by the dark arrow. The final state (q_0) is denoted by a dark circle.

2.3. Measuring the running time

We often use the notations $O()$, $\Theta()$, $\Omega()$, $o()$ and $\omega()$ to measure the various complexities (e.g. the running time and the number of queries) of an algorithm.

Let $p(n)$ and $q(n)$ be two functions whose argument is the natural numbers, and whose values are always greater than zero. Then we have the following definitions:

$O()$: We write $p = O(q)$, if there exists a natural number n_0 and a constant $c > 0$, such that for every $n > n_0$, $p(n) \leq cq(n)$. Observe that if $p = O(q)$, then this means that $p(n)$ is bounded above by $cq(n)$, for sufficiently large n 's (or briefly, p is bounded above by q). As an example, $n + 5 = O(n^2)$, and $n^2 = O(n^2)$.

$\Omega()$: We write $p = \Omega(q)$, if there exists a natural number n_0 and a constant $c > 0$, such that for every $n > n_0$, $p(n) \geq cq(n)$. Another way to define $\Omega()$, is by saying that $p = \Omega(q)$ if and only if $q = O(p)$. As an example we have that $n^2 = \Omega(n + 5)$.

$\Theta()$: $p = \Theta(q)$ if both $p = O(q)$ and $p = \Omega(q)$. In other words, $p = \Theta(q)$ if there exist a natural number n_0 and two constants c_0 and c_1 greater than zero, such that for every $n > n_0$, $c_0q(n) \leq p(n) \leq c_1q(n)$. As an example, consider $3n = \Theta(n)$.

$o()$: Informally, $p = o(q)$ if p is bounded above by q and the bound is not tight*.

*A bound $b(n)$ on $p(n)$ is *tight* if $b(n) \leq cp(n)$, for some constant c and for all sufficiently large n 's.

Formally, $p = o(q)$ if

$$\lim_{n \rightarrow \infty} \frac{p(n)}{q(n)} = 0.$$

So, for example, $n + 5 = o(n^2)$.

$\omega()$: $p = \omega(q)$ if p is bounded below by q and the bound is not tight. Formally, $p = \omega(q)$, if

$$\lim_{n \rightarrow \infty} \frac{p(n)}{q(n)} = \infty.$$

Note that if $p(n)$ is bounded above by $q(n)$ and the bound is not tight, then $q(n)$ is bounded below by $p(n)$ and the bound is not tight, and vice versa. This yields that $p = \omega(q)$ if and only if $q = o(p)$.

We later use the $\omega()$ notation to indicate that $n^{\omega(1)}$ is not bounded by any polynomial, or, in other words, is *super-polynomial*. More formally, an expression $p(n)$ is super-polynomial if for any constant k , $p(n) > n^k$, for infinitely many n 's.

2.4. Miscellaneous definitions and facts

Let a and b be two positive integers such that $a \geq b$. Then $\binom{a}{b} = \frac{a!}{b!(a-b)!}$. For $a \geq b$, $\binom{a}{b}$ is the number of possible subsets of size b out of a distinct elements. It is known that

$$\left(\frac{a}{b}\right)^b \leq \binom{a}{b} \leq \left(\frac{ea}{b}\right)^b,$$

where e is Euler's constant ($e = 2.718281828\dots$). The bound holds for any integer $a > 0$, and any integer b such that $0 \leq b \leq a$.

In this thesis \log denotes the logarithm base 2, and \ln denotes the natural logarithm.

2.5. Exact learning definitions

In this thesis we are interested in a specific learning model, namely, the minimally adequate teacher model introduced by Angluin [4]. In this model, there are two oracles: the membership oracle, and the equivalence oracle. Let \mathcal{C} be the target class of concepts, and let $f \in \mathcal{C}$ be the target concept. The membership oracle (representing f) gets as input an instance $x \in \mathcal{X}$, and answers with $f(x)$. The

equivalence oracle (representing f) takes as input a concept h , and answers whether or not $h \equiv f$. If they are not equivalent, the oracle also outputs a counterexample, that is, an instance $a \in \mathcal{X}$ on which f and h disagree. We assume that a call to an oracle and getting its reply takes only one time unit. Observe that given access to a membership oracle representing a concept f (that is, the oracle replies with the value of f on the input instance), one can simulate membership queries for any projection f_p of f given the partial assignment p . This is true because $f_p(x) = f(p|x)$ (by definition of f_p).

When the target concept is a boolean formula, then we say that an example is *positive* (*negative*) if it satisfies (falsifies) the target formula. When the target concept is a DFA U , then an example a is a positive example if it is recognized by U , that is, is in the regular set defined by U , and a negative example otherwise.

The size of a DNF (CNF) formula is the number of terms (clauses) in it. The size of a DFA is the number of the states in the automaton. A read-once formula can be represented by a tree in which the leaves contain variables and constants, and the internal nodes contain operators (taken from some known set of operators). The size of a read-once formula is measured in a different way: the number of nodes in the tree representing it (it is known that a read-once formula has a unique tree representation up to some isomorphism that preserves the size of the tree).

A class of formulas \mathcal{C} is *exactly learnable* by a class of formulas \mathcal{C}' , if there exists an algorithm A with the following property: for any $f \in \mathcal{C}$, given access to membership and equivalence oracles representing f , A outputs a formula g that is logically equivalent to f . Both the input formulas to the equivalence oracle and the output formula g must be from the class \mathcal{C}' . Also, the running time of A must be polynomial in the number of variables n and in the size of the smallest formula in \mathcal{C}' that is logically equivalent to f . We refer to A as the *learner* or the *learning algorithm*. When the target class \mathcal{C} is not a class of formulas, then the definition may differ slightly. For example, when \mathcal{C} is a class of DFAs (that is, regular sets), then the restriction on the

running time of A is that it is polynomial in the length of the longest counterexample returned by the equivalence oracle, and in the size of the smallest concept in \mathcal{C}' equivalent to the target formula.

We say a class of concepts \mathcal{C} is *exactly learnable* if there exists a class of concepts \mathcal{C}' such that \mathcal{C} is exactly learnable by \mathcal{C}' . Since we are going to consider only algorithms in the minimally adequate teacher model, we will use the terms “learnable by” and “learnable” to stand for “exactly learnable by” and “learnable”, respectively. We may also use the term “polynomial time” without specifying the arguments of the polynomial, when there is no ambiguity.

As mentioned in the introduction, our goal is to reduce the number of equivalence queries used to learn the target class of concepts. We are interested in proving tight bounds on the number of equivalence queries needed to learn some classes of concept. In other words, we are interested in proving lower bounds on the number of equivalence queries needed and to show matching upper bounds.

If \mathcal{C} is a representation class, we define $\mathcal{E}(\mathcal{C}, q)$ to be the minimum worst case number of equivalence queries made by any polynomial time algorithm that uses at most q membership queries to identify \mathcal{C} . (That is, an algorithm A that exactly identifies \mathcal{C} and never makes more than q membership queries when doing so, must make at least $\mathcal{E}(\mathcal{C}, q)$ equivalence queries when run for some target $f \in \mathcal{C}$. Furthermore there is some such A that achieves this bound when run on any target from \mathcal{C} .) Observe that this quantity typically decreases as q increases. We let $\mathcal{E}(\mathcal{C})$ denote the minimum number of equivalence queries made by any polynomial time algorithm to identify \mathcal{C} (making a polynomial number of membership queries). Likewise, when the learner is not restricted to use polynomial time we let $\mathcal{E}_U(\mathcal{C}, q)$ denote the minimum number of equivalence queries needed to obtain exact identification when at most q membership queries are made. Finally, $\mathcal{E}_U(\mathcal{C})$ denotes the number of equivalence queries needed to obtain exact identification using unlimited time when a polynomial number of membership queries can be made.

A variation that we explore here is whether the learner is given the size of the target concept before the learning session begins. For previous work aimed mainly at proving tractability, this is not an important distinction, since a standard technique allows conversion from an algorithm that knows the size of the target to one that does not [22]. However for our precise bounds this difference can be important, and for some classes we obtain different results depending on whether or not the size of the target concept is known *a priori*.

The numbering of lines in the figures that include algorithms are merely for ease of reference in the text.

CHAPTER 3

A generalization of the halving algorithm

In this section we consider a generalization of the halving algorithm [11, 24, 5] in which we can reduce the number of equivalence queries required by allowing the learner to make membership queries. Unlike all other positive results presented in this thesis, in this chapter we shall not bound the computation time of the learner. However, the learner is still limited to make a polynomial number of membership queries.

The problem with which we deal in this chapter is learning a class of concepts when the learner is given an infinite computational power. That is, we do not count the steps that do not involve calling the oracles; only oracle calls are counted.

3.1. The standard halving algorithm

We first present the standard halving algorithm, shown in figure 3.1. At each iteration of the main loop in the halving algorithm, there is a set \mathcal{C}_i of formulas one of which is the target formula (i refers here to the index of the iteration). The algorithm uses a *majority vote* function g . For an instance x let $\mathcal{C}_i^0(x)$ be the set of all formulas in \mathcal{C}_i that are 0 on x . $\mathcal{C}_i^1(x)$ is defined analogously (so $\mathcal{C}_i^0(x) \cup \mathcal{C}_i^1(x) = \mathcal{C}_i$, for every instance x). The majority vote function g is defined as follows. For an instance x

$$g(x) = \begin{cases} 0 & \text{if } |\mathcal{C}_i^0(x)| \geq |\mathcal{C}_i^1(x)|, \\ 1 & \text{otherwise.} \end{cases}$$

```

halving( $\mathcal{C}$ )
1 initialize  $\mathcal{C}_0$  to  $\mathcal{C}$  and  $i$  to 0.
2 while  $|\mathcal{C}_i| > 1$ 
3   for all  $x \in \mathcal{X}$ 
4     if  $|\mathcal{C}_i^0(x)| \geq |\mathcal{C}_i^1(x)|$ 
5       then  $g(x) \leftarrow 0$ .
6       else  $g(x) \leftarrow 1$ .
7   if Equiv( $g$ ) = "yes" then return  $g$ 
8   let  $a$  be the counterexample.
9    $\mathcal{C}_{i+1} \leftarrow \mathcal{C}_i^{\neg g(a)}(a)$ .
10   $i \leftarrow i + 1$ .
11 return the (unique) formula in  $\mathcal{C}_i$ .

```

FIGURE 3.1. *The standard halving algorithm that uses only equivalence queries.*

We now ask an equivalence query with g , and let a be the counterexample returned. Suppose, without loss of generality, $g(a) = 0$, so $f(a) = 1$, where f is the target formula. By definition of g , we know that $|\mathcal{C}_i^0(a)| \geq |\mathcal{C}_i^1(a)|$. Since $f(a) = 1$, f is in the set $\mathcal{C}_i^1(a)$, so we update \mathcal{C}_{i+1} to be $\mathcal{C}_i^1(a)$, and iterate again. We stop iterating when we reach the stage where \mathcal{C}_i contains only one formula, which is the target formula. Observe that $\mathcal{C}_i^1(a)$ is smaller than $\mathcal{C}_i^0(a)$. Thus the size of \mathcal{C}_{i+1} is at most half the size of \mathcal{C}_i . We conclude that $|\mathcal{C}_i| \leq |\mathcal{C}| (1/2)^i$. Solving this inequality will give us that there are at most $\log |\mathcal{C}|$ iterations.

This establishes the following theorem.

THEOREM 3.1. [11, 24, 5] *For any concept class \mathcal{C} , $\mathcal{E}_U(\mathcal{C}, 0) \leq \log |\mathcal{C}|$.*

3.2. Generalizing the halving algorithm

In this section we show how to add membership queries to the halving algorithm in order to reduce the number of equivalence queries used.

THEOREM 3.2. *For any concept class \mathcal{C} and any $q \geq 2 \ln |\mathcal{C}|$,*

$$\mathcal{E}_U(\mathcal{C}, q) \leq \frac{\log |\mathcal{C}|}{\log q - \log \ln |\mathcal{C}|}.$$

```

generalized-halving( $\mathcal{C}, \alpha$ )
1 initialize  $\mathcal{C}_0$  to  $\mathcal{C}$  and  $i$  to 0.
2 while  $|\mathcal{C}_i| > 1$ 
3     if there exists  $x \in \mathcal{X}$  such that  $\min \{|\mathcal{C}_i^0(x)|, |\mathcal{C}_i^1(x)|\} \geq \alpha|\mathcal{C}_i|$ 
4         then
5              $\xi \leftarrow \text{MQ}(x)$ .
6              $\mathcal{C}_{i+1} \leftarrow \mathcal{C}_i^\xi(x)$ .
7         else
8             for all  $x \in \mathcal{X}$ 
9                 if  $|\mathcal{C}_i^1(x)| \geq |\mathcal{C}_i^0(x)|$ 
10                     then  $g(x) \leftarrow 1$ .
11                     else  $g(x) \leftarrow 0$ .
12             if  $\text{Equiv}(g) = \text{"yes"}$ 
13                 then return  $g$ .
14             else let  $a$  be the counterexample.
15              $\mathcal{C}_{i+1} \leftarrow \mathcal{C}_i^{\neg g(a)}(a)$ .
16      $i \leftarrow i + 1$ .
17 return the (unique) formula in  $\mathcal{C}_i$ .

```

FIGURE 3.2. A generalization of the halving algorithm that uses membership queries to reduce the number of equivalence queries.

Proof: In figure 3.2 we give an algorithm to learn any class \mathcal{C} with unlimited computation time using at most $\log |\mathcal{C}| / \log \frac{1}{1-\alpha}$ membership queries and $\log |\mathcal{C}| / \log \frac{1}{\alpha}$ equivalence queries for any $0 < \alpha \leq 1/2$.

A membership query is performed if there exists an instance $x \in \mathcal{X}$ for which both $\mathcal{C}_i^0(x)$ and $\mathcal{C}_i^1(x)$ have cardinality at least $\alpha|\mathcal{C}_i|$. Thus, each membership query allows the learner to eliminate at least $\alpha|\mathcal{C}_i|$ of the remaining concepts. If for all $x \in \mathcal{X}$ either $\mathcal{C}_i^0(x)$ or $\mathcal{C}_i^1(x)$ has cardinality less than $\alpha|\mathcal{C}_i|$ then, just like in the standard halving algorithm, the learner uses the majority vote hypothesis. However, instead of just being assured that half of the elements of \mathcal{C}_i are eliminated, here, at least $(1 - \alpha)|\mathcal{C}_i|$ concepts are eliminated.

Let q and p be the number of membership queries and equivalence queries, respectively, used by the generalized halving algorithm. Since every membership query eliminates at least $\alpha|\mathcal{C}_i|$ concepts from \mathcal{C}_i , we obtain the following inequality

$$(1 - \alpha)^q |\mathcal{C}| \geq 1.$$

Solving this inequality we get that the algorithm uses at most $\log |\mathcal{C}| / \log \frac{1}{1-\alpha}$ membership queries. Similarly, since every equivalence query eliminates at least $(1 - \alpha)|\mathcal{C}_i|$ of \mathcal{C}_i , we get the inequality

$$\alpha^p |\mathcal{C}| \geq 1,$$

which gives the bound $p \leq \log |\mathcal{C}| / \log(1/\alpha)$.

We now use the standard inequality $(1 - y)^{1/y} \leq e^{-1}$, for every $y > 0$. Another way to write this inequality is $\log \frac{1}{1-y} \geq y \log e$. Applying this inequality in the bound on q (for $y = \alpha$) we get

$$q \leq \frac{\log |\mathcal{C}|}{\log(1/(1 - \alpha))} \leq \frac{\log |\mathcal{C}|}{\alpha \log e},$$

and hence $\alpha \leq \frac{\log |\mathcal{C}|}{q \log e} = \frac{\ln |\mathcal{C}|}{q}$. Note that since $\alpha \leq 1/2$ we must have $q \leq 2 \ln |\mathcal{C}|$. We now use this bound on α in the bound on p to get

$$p \leq \frac{\log |\mathcal{C}|}{\log(1/\alpha)} \leq \frac{\log |\mathcal{C}|}{\log q - \log \ln |\mathcal{C}|}.$$

□

COROLLARY 3.3. For any class \mathcal{C} , and for $q = \frac{\log^{d+1} |\mathcal{C}|}{\log e}$ ($d \geq 1$),

$$\mathcal{E}_U(\mathcal{C}, q) \leq \frac{\log |\mathcal{C}|}{d \log \log |\mathcal{C}|}.$$

Proof: This immediately follows from the above theorem, after substituting $q = \frac{\log^{d+1} |\mathcal{C}|}{\log e}$. □

3.3. Applying the generalized halving algorithm to DNF formulas

We remind the reader that a DNF formula is a formula in disjunctive normal form (that is, a disjunction of terms).

In this section we establish the following upper bound.

THEOREM 3.4. For \mathcal{C} the class of m -term DNF formulas,

$$\mathcal{E}_U(\mathcal{C}) = O\left(\frac{mn}{\log n + \log m}\right),$$

provided that $m \leq 2^{\epsilon n}$, for some $\epsilon < 1/2$.

Proof: In lemma 3.5 (below) we prove that the class of m -term DNF formulas (for $m \leq 2^{\epsilon n}$, $\epsilon < 1/2$), is of size at least $2^{c mn}$ for some constant $c > 0$. In lemma 3.6 we prove that there are at most $O(3^{mn})$ DNF formulas with m terms. Combining these two bounds together, and by using corollary 3.3 we obtain for \mathcal{C} the class of m -term DNF formulas we obtain the following result:

$$\mathcal{E}_U(\mathcal{C}) \leq \frac{\log |\mathcal{C}|}{\log \log |\mathcal{C}|} \leq \frac{\log 3^{mn}}{\log \log 2^{c mn}} \leq \frac{mn \log 3}{\log c + \log m + \log n} = O\left(\frac{mn}{\log n + \log m}\right).$$

□

Note that by duality we get a similar result for the class of m -clause CNF formulas. The same result applies also to m -clause Horn sentences, since a Horn sentence is a CNF formula.

We now prove the lower bound on the number of m -term DNF formulas. We obtain a lower bound by proving that the number of different monotone m -term DNF

formulas (for $m \leq 2^{\epsilon n}$, $\epsilon < 1/2$) is at least 2^{cmn} for some constant $c > 0$. Observe that the class of monotone DNF formulas is a subclass of the general DNF formulas class.

LEMMA 3.5. *Let \mathcal{C} be the class of monotone DNF formulas each containing exactly m terms, where $m \leq 2^{\epsilon n}$ for some constant $\epsilon < 1/2$. Then $|\mathcal{C}| > 2^{cmn}$, for some constant $c > 0$.*

Proof: Let \mathcal{T} be the set of all monotone terms each containing exactly $\lfloor n/2 \rfloor$ variables. The size of \mathcal{T} is $\binom{n}{\lfloor n/2 \rfloor}$. Let \mathcal{C}' be the set of DNF formulas each having exactly m terms from \mathcal{T} . The size of \mathcal{C}' is

$$\binom{\binom{n}{\lfloor n/2 \rfloor}}{m}.$$

We now use twice the inequality $\binom{n}{k} \geq (n/k)^k$, to get

$$\begin{aligned} (1) \quad |\mathcal{C}'| &= \binom{\binom{n}{\lfloor n/2 \rfloor}}{m} \geq \left(\frac{\binom{n}{\lfloor n/2 \rfloor}}{m} \right)^m \geq \left(\frac{\left(\frac{n}{\lfloor n/2 \rfloor} \right)^{\lfloor n/2 \rfloor}}{m} \right)^m \\ (2) \quad &\geq \left(\frac{2^{\lfloor n/2 \rfloor}}{m} \right)^m \geq \left(2^{\lfloor n/2 \rfloor - \epsilon n} \right)^m \geq \left(2^{\frac{n}{2} - 1 - \epsilon n} \right)^m \\ (3) \quad &\geq \left(2^{c'n - 1} \right)^m \text{ for } c' = (1/2) - \epsilon \\ (4) \quad &\geq (2^{\epsilon n})^m = 2^{cmn} \text{ for } c = c'/2 \end{aligned}$$

In line 1 we used twice the inequality mentioned above. In line 2 we used the fact that $\frac{n}{\lfloor n/2 \rfloor} \geq 2$, the fact that $\lfloor n/2 \rfloor \geq (n/2) - 1$, and the fact that $m \leq 2^{\epsilon n}$. In line 3 we used the fact that $\epsilon < 1/2$ so $c' = 1/2 - \epsilon$ is a strictly positive number. We obtain line 4 by observing that for sufficiently large n it is the case that $c'n - 1 > \frac{\epsilon}{2}n$. This proves the lemma. \square

We now prove an upper bound on the number of m -term DNF formulas (the bound holds for every m).

LEMMA 3.6. *The number of m -term DNF formulas is at most 3^{mn} .*

Proof: Fix a term t and a variable v . The variable v either appears negated in t , unnegated or does not appear. For each variable there are these three possibilities, so the number of all terms is at most 3^n . Another way to see this is by representing a term t using a vector of n positions: position i , $1 \leq i \leq n$, is either 1 (if the variable v_i appears unnegated in t), 0 (if v_i appears negated in t), or $*$ (if v_i does not appear in t). Every such vector represents a unique term, and every term is represented by a unique vector. The number of these vectors (and, hence, the number of all possible terms) is 3^n . An m -term DNF formula is a disjunction of a subset of m of these terms, so the number of all possible such formulas is at most

$$\binom{3^n}{m} \leq (3^n)^m = 3^{mn}.$$

□

CHAPTER 4

k-term DNF

Learning DNF formulas is still an open problem, and an intensive research is devoted to learning subclasses of the DNF formulas, e.g. monotone DNF, Read-Twice DNF, Read-Thrice DNF, *k*-DNF and *k*-term DNF. In this chapter we consider the problem of learning *k*-term DNF with membership queries and equivalence queries.

In section 4.1 we give the main known results regarding exact learning of *k*-term DNF formulas. In section 4.2 we present three general techniques for learning *k*-term DNF, that use a procedure (**produce-terms**). In section 4.3 we show a version of **produce-terms** that is based on Blum and Rudich paper [12]. In section 4.4 we summarize our results. The techniques of this chapter can be applied also to Angluin's algorithm for learning *k*-term DNF formulas [3]. However, the complexities obtained from applying our techniques to Blum and Rudich's algorithm [12] are far better, so we have placed the improvement of Angluin's algorithm in appendix A.

4.1. Previous work and of our results

Pitt and Valiant [26] and Kearns et. al [23] have proved that *k*-term DNF, for $k \geq 2$, is unlearnable in the PAC model (when the output formula is also *k*-term DNF) unless $RP = NP$. * Using Angluin's transformation from an equivalence query

*NP is the class of all decision problems that can be solved by nondeterministic polynomial Turing machines, and RP is the class of all decision problems that can be solved by probabilistic polynomial Turing machines, with one-sided error [10]. Obviously, $RP \subseteq NP$, but it is still an open problem whether or not the containment is proper.

algorithm to a PAC algorithm [5], the unlearnability of k -term DNF formulas in the PAC model implies that k -term DNF formulas are not learnable using equivalence queries alone. Therefore, the use of membership queries is essential in order to exactly identify k -term DNF formulas by k -term DNF formulas.

The first algorithm in the literature for learning k -term DNF formulas in the minimally adequate teacher model is due to Angluin [3]. The running time, the number of membership queries and the number of equivalence queries are each $O(n^{k^2})$. These complexities are polynomial when k is constant. Angluin's algorithm is in the restricted model, that is both the output formula and the formulas given as input to the equivalence query oracle are k -term DNF formulas.

Blum and Rudich [12] have significantly refined this to a learning algorithm that uses $O(n(\log n)^{O(1)}2^k)$ equivalence queries and membership queries. This complexity is polynomial for $k = O(\log n)$. However, their algorithm is in the unrestricted model.

Applying our techniques to Blum and Rudich's algorithm we reduce the number of equivalence queries to $k + 1$ (or k , when k is known *a priori*) while the number of membership queries is either $O(n(\log n)^{O(1)}2^k)$ (when using unrestricted equivalence queries) or $O(n(\log n)^{O(k)}2^{k^2})$ (when using restricted equivalence queries). The number of membership queries of the algorithm in the unrestricted model is polynomial for $k = O(\log n)$, and the number of membership queries of the other algorithm is polynomial for $k = O(\sqrt{\log n})$.

4.2. A general algorithm

In this section we present three general techniques for learning k -term DNF formulas. We begin with a straightforward algorithm that formalizes both algorithms of Angluin [3] and Blum and Rudich [12]. We then show two improvements of it.

All three algorithm use a procedure, **produce-terms**, that gets as an argument a positive example x (for the target formula f), and uses only membership queries to produce c terms one of which is in f and is satisfied by x . Let q be an upper bound

learn1- k -term-dnf

```

1  $\mathcal{T} \leftarrow \emptyset$ .
2 repeat
3   let  $h$  be the disjunction of all terms in  $\mathcal{T}$  (if  $\mathcal{T} = \emptyset$  then  $h \leftarrow \text{false}$ ).
4    $a \leftarrow \text{Equiv}(h)$ .
5   if  $a = \text{"yes"}$  then return  $h$ .
6   if  $a$  is a positive counterexample
7     then call produce-terms( $a$ ) and add the terms it returns to  $\mathcal{T}$ .
8   else ( $a$  is a negative counterexample)  $\mathcal{T} \leftarrow \mathcal{T} - \{t \in \mathcal{T} : t(a) = 1\}$ .
9 until done.
```

FIGURE 4.1. A general algorithm for learning k -term DNF in time $O(kt + nk^2c^2)$, using at most kc equivalence queries and at most $kc + kq$ membership queries.

on the number of membership queries that **produce-terms** uses. Also, let t be an upper bound on the running time of **produce-terms**.

Let $f(x_1, \dots, x_n) = t_1 \vee \dots \vee t_k$ be the target k -term DNF formula, where t_1, \dots, t_k are terms. We assume that f is reduced; that is, we cannot drop any term from f or any literal from any term without logically changing f .

The idea behind the algorithm **learn1- k -term-dnf** is the following. We first ask an equivalence query with the **false** concept, to get a positive counterexample a (unless the target formula is identically false, then we are done). We call **produce-terms**(a) to produce c terms t_1, \dots, t_c , one of which is in the target formula f and is satisfied by a . We define h to be the conjunction of the terms t_1, \dots, t_c , and ask an equivalence query with it. If the counterexample a is negative (that is $f(a) = 0$) then we know that there is at least one term in h that is not in f , in particular, all terms in h that are satisfied by a . We drop these terms and ask another equivalence query. On the other hand, if a is a positive example, then we call **produce-terms**(a) to get another c terms one of which is a term t' in f . Since all terms in h are falsified by a , and t' is satisfied by a , t' cannot be in h , so the c new terms added to h contain a *new* term from f . So far, h contains two term from f . We continue in this process:

a negative counterexample drops “bad” terms from h and a positive counterexample adds c terms to h one of which is a new term from f .

THEOREM 4.1. *Given the procedure **produce-terms** described above, the algorithm **learn1-k-term-dnf** learns the target formula in time $O(kt + nk^2c^2)$, using at most kc equivalence queries and at most $kc + kq$ membership queries.*

We first prove the next lemma.

LEMMA 4.2. *The procedure **produce-terms** is called at most k times.*

Proof: Every time **learn1-k-term-dnf** calls **produce-terms**, with the positive counterexample a , c terms are added to \mathcal{T} one of which is in f and is satisfied by a (t_i without loss of generality). Since $h(a) = 0$, all terms in \mathcal{T} are falsified, whereas $t_i(a) = 1$. Therefore, t_i is not in \mathcal{T} . Moreover, t_i will be never deleted by a negative counterexample. The reason is that if a is a negative counterexample then every term in f including t_i is falsified by a , so t_i will not be dropped from \mathcal{T} in step 8. After k calls to **produce-terms** all the terms of f will be in \mathcal{T} , so $f \Rightarrow h$. Therefore, the following counterexamples will all be negative. \square

Proof of theorem 4.1: By the above lemma, \mathcal{T} will contain at most kc terms, k of which are the terms that appear in f . Each negative counterexample drops at least one term from \mathcal{T} . Therefore, the number of negative counterexamples is at most $kc - k$, implying that the number of equivalence queries is at most kc . Membership queries are needed for **produce-terms** and for knowing the classification of a in step 6 of the algorithm. Therefore, the number of membership queries is at most $kq + kc$ (recall that q is the number of membership queries used by **produce-terms**). Note that it is easy to determine (in time $O(n)$) if a term is satisfied by a given assignment. To find the running time of the algorithm, note that if a is a positive example **learn1-k-term-dnf**

spends time t in **produce-terms**, and when a is a negative counterexample, **learn1- k -term-dnf** spends time $O(nkc)$ dropping “bad” terms from \mathcal{T} . Since the number of positive counterexamples is at most k and the number of negative counterexamples is at most kc , the running time is $O(kt + nk^2c^2)$. \square

4.2.1. The first improvement. We now give an algorithm that meets the lower bound of Bshouty and Cleve [15] regarding the number of equivalence queries needed to identify k -term DNF formulas.

In this section, we present an algorithm for learning k -term DNF formulas in which the running time is $O(kt + nc^{O(k)})$ and the number of membership queries is at most $kq + c^{k+1}$. The number of equivalence queries when k is known is k , and $k+1$ otherwise. Furthermore, the formulas used for the equivalence queries and the output formula are k -term DNF formulas.

We first describe a parallel version of our algorithm in which the number of parallel steps of equivalence queries is $k + 1$ and the total number of equivalence queries is at most c^{k+1} . We then show how to make the algorithm sequential in such a way to reduce the number of equivalence queries to $k + 1$.

4.2.1.1. A parallel greedy algorithm. We begin with an informal description of our parallel algorithm. Let x be a positive example of f . If we call **produce-terms**(x), we get c terms $\mathcal{T}^{(1)} = \{t_1^{(1)}, \dots, t_c^{(1)}\}$, one of which is guaranteed to be a term in f (without loss of generality assume that $\mathcal{T}^{(1)}$ contains the term t_1 from f). We now continue performing the following step in parallel on all these terms. For each $t \in \mathcal{T}^{(1)}$ make the equivalence query **Equiv**(t). If the counterexample x is negative then t is a “bad” term and we quit working on it. Otherwise, $f(x) = 1$, call **produce-terms**(x) to get another c terms, $\mathcal{T}^{(2)} = \{t_1^{(2)}, \dots, t_c^{(2)}\}$, one of which is guaranteed to be a term in f . Furthermore, since $t_1 \in \mathcal{T}^{(1)}$, it follows that some *other* term from f (say t_2) is in $\mathcal{T}^{(2)}$.

```

FORK( $H_1, H_2, \dots, H_c$ , phase)
1  allocate  $c$  processors, and for each processor  $j = 1, \dots, c$  do
2       $x^{(j)} \leftarrow \text{Equiv}(H_j)$ .
3      if  $x^{(j)}$  is “yes”
4          then return  $H_j$ , and we are done.
5      if  $x^{(j)}$  is a negative counterexample
6          then stop working on  $H_j$ .
7      else
8          call produce-terms( $x^{(j)}$ ), and let  $t_1^{(j)}, t_2^{(j)}, \dots, t_c^{(j)}$  be the
          terms returned.
9      FORK( $H_j \vee t_1^{(j)}, H_j \vee t_2^{(j)}, \dots, H_j \vee t_c^{(j)}$ , phase+1).

```

FIGURE 4.2. A macro that the main routine, *learn2-k-term-dnf*, uses

We now work in parallel on all formulas of the form $t_i^{(1)} \vee t_j^{(2)}$ for $1 \leq i, j \leq c$ making an equivalence query for each one. As before, if the counterexample is negative we stop working on that formula. Otherwise we give the counterexample as input to **produce-terms** and get another c terms one of which is a *new* term in f . After k such parallel phases we will have a set of k -term DNF formulas, one of which is the target formula. Finally, we use equivalence queries to find which formula is the target formula. To summarize, there are k phases and in phase i there are at most c^i i -term DNF formulas, one of which contains i terms from f . In addition, note that we get these formulas independently, in the sense that getting some i -term DNF formula does not depend on getting other i -term DNF formulas. Figure 4.2 formalizes the above discussion, and figure 4.3 shows **learn2-k-term-dnf**.

4.2.1.2. Analysis. We now analyze the complexity of this parallel algorithm. There are at most k phases. In phase i , $1 \leq i \leq k$, **learn2-k-term-dnf** asks at most c^i equivalence queries. Therefore, the total number of equivalence queries made is at most $1 + \sum_{i=1}^k c^i \leq c^{k+1}$ (we need one equivalence query before calling **FORK**). Also, in phase i , $1 \leq i \leq k$, **learn2-k-term-dnf** calls **produce-terms** at most c^i times. Therefore, the total number of membership queries is $q + \sum_{i=1}^k qc^i \leq qc^{k+1}$ (here,

```

learn2- $k$ -term-dnf
1  $x \leftarrow \text{Equiv}(\text{false})$ .
2 if  $x$  is “yes”
3     then return false) and we are done.
4 call produce-terms( $x$ ), and let  $t'_1, t'_2, \dots, t'_e$  be the terms returned.
5 FORK( $t'_1, \dots, t'_e, 1$ ).

```

FIGURE 4.3. An algorithm for learning k -term DNF formulas in time $O(kt + nc^{O(k)})$ using at most $kq + c^{k+1}$ membership queries and k equivalence queries (or $k + 1$, if k is not known a priori)

again, we call **produce-terms** before calling FORK), and the sequential running time is $tc^{O(k)}$.

4.2.1.3. *Reducing the number of equivalence queries.* Our idea in reducing the number of equivalence queries is the following. Suppose we have two i -term DNF formulas h and h' , and we want to run an equivalence query for both. Instead, we test whether $h \equiv h'$. If this is the case then we can drop one of them. Otherwise, we find an assignment y for which (without loss of generality) $h(y) = 0$ and $h'(y) = 1$. We then perform a membership query to see if y is a negative or a positive example (of f). If y is a negative example then h' can be discarded because $h'(y) = 1$ implies that h' contains a term not in f . In this case we ask an equivalence query with h . Otherwise y is a positive counterexample for h and we perform an equivalence query for h' .

Using this idea we reduce the number of equivalence queries in phase i from c^i to 1 (the last i -term DNF formula has no other i -term DNF formula to be compared with, so we ask an equivalence query with it). On the other hand, the number of membership queries is increased by $c^i - 1$. If k is known then there is no need to ask an equivalence query in the k th phase, because the formula to pass the last test is guaranteed to be the target formula. Otherwise, we need an equivalence query for the k th phase as well, and then the number of equivalence queries is $k+1$.

All that remains now is to give a procedure that tests if two i -term DNF formulas are logically equivalent. In the next subsection we show a procedure, **are-equivalent**, that handles this task in time $O(ni^3 + 2^{O(i)})$. Having this procedure with the claimed running time we can now prove the next theorem.

THEOREM 4.3. *Given **produce-terms** as described above, **learn2-k-term-dnf** exactly identifies an unknown k -term DNF formula in time $O(kt + nc^{O(k)})$ using at most $kq + c^{k+1}$ membership queries and k equivalence queries (or $k + 1$, if k is not*

given as an input to the algorithm). Moreover, the learning is in Angluin's restricted model.

Proof: We need only find the running time and the number of membership queries.

The sequential version of **learn2- k -term-dnf** (i.e. after dropping the number of i -term DNF formulas in phase i to 1) calls **produce-terms** k times, once in each phase. The procedure **are-equivalent** is used $c^i - 1$ times in phase i , so it is called at most c^{k+1} times. Therefore, the time spent for dropping the intermediate formulas is $O((nk^3 + 2^{k+1})c^{k+1}) = nc^{O(k)}$, and the total time is $O(kt + nc^{O(k)})$.

The number of membership queries is at most $kq + c^{k+1}$, where the first factor is due to calling **produce-terms** k times and the second is due to the dropping of the intermediate DNF formulas in the k stages. \square

4.2.1.4. *Testing the equivalence of two k -term DNF formulas.* All that remains now is to give a function that tests whether two k -term DNF formulas are equivalent. We first give few lemmas that will establish the correctness of **are-equivalent** which takes two k -term DNF formulas and returns true if and only if they are logically equivalent.

Let t be a term (not equivalent to false or true) and let h be a DNF formula. We define the partial assignment $p_t \in \{0, 1, *\}$ as follows:

$$p_t[i] = \begin{cases} 1 & \text{if } v_i \text{ appears unnegated in } t \\ 0 & \text{if } v_i \text{ appears negated in } t \\ * & \text{if } v_i \text{ does not appear in } t, \end{cases}$$

for every $i = 1, \dots, n$. Note that, no matter how the $*$'s in p_t are fixed, p_t satisfies t . This partial assignment induces a projection h_{p_t} defined as: for every assignment a ,

$h_{p_t}(a) = h(p_t|a)$. We remind the reader that the assignment $p_t|a$ is defined as follows:

$$(p_t|a)[i] = \begin{cases} p_t[i] & \text{if } p_t[i] \neq * \\ a[i] & \text{if } p_t[i] = *, \end{cases}$$

for every $i = 1, \dots, n$. For ease of notation we let h_t to denote h_{p_t} . A boolean function h is a *tautology* if $h(a) = 1$ for every assignment a .

We first establish few lemmas that will be used to prove the correctness of **are-equivalent**.

LEMMA 4.4. *Let h be a DNF formula and t a term. Then $t \Rightarrow h$ if and only if h_t is a tautology.*

Proof: The term $t \Rightarrow h$ if and only if $t(a) = 1$ implies $h(a) = 1$, for every assignment. This means that $t \Rightarrow h$ if and only if $h(a) = 1$, for every assignment a that assign 1 to all literals in t . By definition of p_t , p_t assigns 1 to all literals in t , so $t \Rightarrow h$ if and only if $h(p_t|a) = 1$, for every assignment a . Using the fact that $h(p_t|a) = h_t(a)$, we get the claim of the lemma. \square

LEMMA 4.5. *Let $h = t_1 \vee \dots \vee t_j$ and $h' = t'_1 \vee \dots \vee t'_{j'}$ be DNF formulas. Then $h \equiv h'$ if and only if $t_i \Rightarrow h'$ and $t'_{i'} \Rightarrow h$, for each $1 \leq i \leq j$ and $1 \leq i' \leq j'$.*

Proof: We have the following straightforward implications:

$$\begin{aligned} h \equiv h' &\iff h \Rightarrow h' \text{ and } h' \Rightarrow h \\ &\iff t_1 \vee \dots \vee t_j \Rightarrow h' \text{ and } t'_1 \vee \dots \vee t'_{j'} \Rightarrow h \\ &\iff t_1 \Rightarrow h', \dots, t_j \Rightarrow h', t'_1 \Rightarrow h, \dots, \text{ and } t'_{j'} \Rightarrow h \text{ (a simple fact from logic)} \\ &\iff t_i \Rightarrow h' \text{ and } t'_{i'} \Rightarrow h, \text{ for each } 1 \leq i \leq j \text{ and } 1 \leq i' \leq j'. \end{aligned}$$

\square

LEMMA 4.6. *Let h be a DNF formula that does not contain any term equivalent to **true**. If h is a tautology then there exists some variable v such that both v and \bar{v} appear in h .*

Proof: Suppose this is not the case. For a variable v , we say it appears *positively* in h if v appears unnegated in h , and we say v appears *negatively* if v appears negated in h . Since we assumed that no variable appears both negated and unnegated in h , a variable appears in h either positively or negatively, but not both. Therefore, we can define the following assignment y

$$y[i] = \begin{cases} 0 & \text{if } v_i \text{ appears positively in } h \\ 1 & \text{if } v_i \text{ appears negatively in } h. \end{cases}$$

Note that, no matter what are the values in y of the variables that do not appear in h , y falsifies *every* literal in h , and therefore it falsifies every term in h (by assumption, h does not contain a term that is equivalent to **true**). It follows that $h(y) = 0$, which contradicts the fact that h is a tautology. \square

LEMMA 4.7. *Let h be a DNF formula, and let v be any variable in h . Then h is a tautology if and only if h_v and $h_{\bar{v}}$ are tautologies.*

Proof: It is clear that if h is a tautology, then every projection of h is a tautology too, in particular h_v and $h_{\bar{v}}$, for every variable v .

Suppose h_v and $h_{\bar{v}}$ are tautologies, for some variable v . The function h can be represented by $h = (v \wedge h_v) \vee (\bar{v} \wedge h_{\bar{v}})$. Since both h_v and $h_{\bar{v}}$ are tautologies we get that $h = v \vee \bar{v} = \text{true}$. This proves the other direction. \square

THEOREM 4.8. *Given an i -term DNF formula h , the procedure **is-tautology** returns **true** if and only if h is a tautology. The running time of **is-tautology** is $O(i^2n + 2^i)$.*

Proof: If h contains no variables (i.e. only constants) then it is straightforward to find if $h \equiv 1$. This is done in step 1 in **is-tautology**. Suppose that h does contain

```

is-tautology( $h$ )
1 if  $h$  contains no variables
2     then
3         if  $h \equiv 1$ 
4             then return true.
5             else return false.
6 if no variables appear in  $h$  both negated and unnegated
7     then return false.
8 let  $v_i$  be a variable that appears in  $h$  both negated and unnegated.
9 if is-tautology( $h_{v_i}$ ) = false or is-tautology( $h_{\bar{v}_i}$ ) = false
10     then return false.
11 return true.

```

FIGURE 4.4. An algorithm to test if h is a tautology.

```

are-equivalent( $h, h'$ )
1 let  $h = t_1 \vee \dots \vee t_k$  and  $h' = t'_1 \vee \dots \vee t'_k$ .
2 for each  $i = 1, \dots, k$  do
3     if is-tautology( $h_{t'_i}$ ) = false
4         then return false.
5 for each  $i = 1, \dots, k$  do
6     if is-tautology( $h'_{t_i}$ ) = false
7         then return false.
8 return true.

```

FIGURE 4.5. An algorithm to test if two k -terms formulas are logically equivalent.

variables. Then, we check if it contains some variable both negated and unnegated. If not, then by lemma 4.6 h is not a tautology, so **is-tautology** returns **false** in step 7. If h contains some variable (say v) both negated and unnegated, then, by lemma 4.7, h is a tautology if and only if both h_v and $h_{\bar{v}}$ are tautologies. This is done in step 9.

In order to analyze the running time of **is-tautology**, note that if v appears both positively and negatively in h then the number of terms in each of h_v and $h_{\bar{v}}$ is strictly less than the number of terms in h . The reason is that when fixing the value of v to 1 in h to get h_v , we eliminate the term in which \bar{v} appears. Similarly with $h_{\bar{v}}$: fixing the value of \bar{v} to 0 in h to get $h_{\bar{v}}$ eliminates the term in which v appears. Observe that the size of h is at most $O(in)$, since it contains i terms each containing at most n literals. Therefore, the time spent in steps 6 and 8 is $O(in)$. The following recurrence gives a bound on the running time of **is-tautology** when given an i -term DNF formula.

$$T(i) = \begin{cases} \text{constant} & \text{if } i = 0 \\ 2T(i-1) + O(in) & \text{otherwise.} \end{cases}$$

Solving this recurrence gives $T(i) = O(i^2n + 2^i)$. \square

We now prove the correctness of **are-equivalent**.

THEOREM 4.9. *Given two k -term DNF formulas h and h' , the procedure **are-equivalent** returns true if and only if h and h' are logically equivalent. The running time of **are-equivalent** is $O(k^3n + 2^{O(k)})$.*

Proof: Let $h = t_1 \vee \dots \vee t_k$ and $h' = t'_1 \vee \dots \vee t'_k$. According to lemma 4.5, h and h' are logically equivalent if and only if $t_i \Rightarrow h'$ and $t'_i \Rightarrow h$, for each $1 \leq i \leq k$. Using lemma 4.4, h and h' are logically equivalent if and only if $h_{t'_i}$ and h'_{t_i} are tautologies for every $i = 1, \dots, k$. This is performed in steps 2 and 5.

We now analyze the running time of **are-equivalent**. The procedure **is-tautology** is called at most $2k$ times, each time with a k -term DNF formula. Therefore, the total running time of **are-equivalent** is $O(2k(k^2n + 2^k)) = O(k^3n + 2^{O(k)})$. \square

4.2.2. The second improvement. In this section we present another improvement of the algorithm `learn1-k-term-dnf` that reduces the number of equivalence queries to k (or $k + 1$, if k is not known *a priori*). The result presented here is in Angluin's unrestricted model. We start with an informal discussion.

Suppose we call `produce-terms`(x), for a positive counterexample x , and let $\mathcal{T}' = \{t'_1, \dots, t'_c\}$ be the terms returned. Our goal is to drop the terms in \mathcal{T}' that do not imply the target formula f . If we succeed with our task, then when we add those terms that imply f to h and ask `Equiv`(h), we are guaranteed to get a positive counterexample.

In a previous section we presented a criterion (`is-tautology`) for testing if a k -term DNF formula is a tautology. We now present another criterion for testing if a k -term DNF formula is a tautology. Recall that an (n, k) -universal set is a set of assignments $\{b_1, \dots, b_t\} \subseteq \{0, 1\}^n$ such that every subset of k variables assumes all of its 2^k possible assignments in the b_i 's.

We have the following lemma:

LEMMA 4.10. *Let S be an (n, k) -universal set, and let f be a k -term DNF formula. Then*

$$f \text{ is a tautology if and only if } f(a) = 1, \text{ for every } a \text{ in } S.$$

Proof: If f is a tautology then it is satisfied by all assignments, in particular by those in the (n, k) -universal set.

It is known that every k -term DNF formula is a k -CNF formula. So let h be a k -CNF formula equivalent to f . Suppose h is not a tautology, so there is an assignment a that falsifies h . Let C be a clause in h falsified by a . Since h is a k -CNF formula, the number of literals in C is at most k . Therefore, by definition of an (n, k) -universal set, there exists an assignment b in the (n, k) -universal set S , that assigns to the variables of C the values that a assigns. This assignment b falsifies h . \square

Based on the above lemma and lemma 4.4, we show how to drop the terms in \mathcal{T}' that do not imply f . Let t be a term in \mathcal{T}' . By lemma 4.4, $t \Rightarrow f$ if and only if f_t is a

learn3- k -term-dnf

```

1 let  $x$  be a positive example of  $f$ .
2  $h \leftarrow \text{false}$ .
3 repeat
4   call produce-terms( $x$ ), and let  $\mathcal{T}' = \{t'_1, \dots, t'_c\}$  be the terms returned.
5   add to  $h$  those terms in  $\mathcal{T}'$  that imply  $f$ .
6    $x \leftarrow \text{Equiv}(h)$ .
7   if  $x$  is "yes"
8     then we are done.
9 until done.
```

FIGURE 4.6. An algorithm that learns k -term DNF in time $O(kt + c2^{O(k)} \log n)$ using $k + 1$ equivalence queries if k is not known, and k if it is known, and at most $kq + c2^{O(k)} \log n$.

tautology. Since f is a k -term DNF formula, any projection of it cannot contain more than k terms, so f_t is a k -term DNF formula too. By lemma 4.10, f_t is a tautology if and only if $f_t(a) = 1$ for every assignment a in an (n, k) -universal set S . It is easy to simulate a membership query for f_t , since, by definition of f_t , $f_t(a) = f(p_t|a)$, where p_t is defined as follows

$$(p_t)[i] = \begin{cases} 1 & \text{if } v_i \text{ appears positively in } t \\ 0 & \text{if } v_i \text{ appears negatively in } t \\ \star & \text{if } v_i \text{ does not appear in } t. \end{cases}$$

The number of membership queries to check if a term t implies f is $k^{O(1)} 2^{2k} \log n$ which is the size of the (n, k) -universal set mentioned in lemma 4.10 [12, 19]. Figure 4.6 describes the algorithm.

4.2.2.1. Analysis. Since the terms in h imply f , in each iteration of **learn3- k -term-dnf** h implies f . After k calls to **produce-terms** h will contain all terms of f , therefore f implies h , so $h \equiv f$. If k is known then there is no need for the $(k + 1)$ st equivalence query. The running time is $O(kt + c2^{O(k)} \log n)$, where $O(kt)$

is the time of k calls to **produce-terms**, and $O(c2^{O(k)} \log n)$ is the time spent on applying lemma 4.10 for kc terms. The number of membership queries that **learn3- k -term-dnf** uses is at most $kq + c2^{O(k)} \log n$. Here, again, kq membership queries are needed for k calls to **produce-terms**, and $c2^{O(k)} \log n$ membership queries are needed to apply lemma 4.10 to kc terms.

We thus have proved the following theorem.

THEOREM 4.11. *Given **produce-terms** as described above, **learn3- k -term-dnf** exactly identifies the unknown k -term DNF formula in time $O(kt + c2^{O(k)} \log n)$ using at most $kq + c2^{O(k)} \log n$ membership queries and $k + 1$ equivalence queries if k is not known, and k if it is known.*

4.2.3. Summary. Table 4.1 summarizes the results of the above section.

4.3. A version of produce-terms based on Blum-Rudich's algorithm

Blum and Rudich's [12] algorithm is essentially **learn1- k -term-dnf**. In this section we briefly describe the version of **produce-terms** that is based on their paper.

The original algorithm is quite involved. We, therefore, merely give a general outline of it. The procedure **produce-terms** is divided into two parts. Given a positive example x , the first part produces a collection of $O(2^{O(k)}(\log n)^2)$ assignments one of which, z , satisfies exactly one term in f (that is also satisfied by x). The second part, takes these assignments and produces $O(2^{O(k)}(\log n)^3)$ terms one of which is in f , and is satisfied by z . Both the running time and the number of membership queries used are $O(n^2 2^{O(k)}(\log n)^3)$.

4.4. Conclusion

Using the techniques of the section 4.2 and the version of **produce-terms** presented in 4.3 we get the results shown in the table 4.2.

Note that the complexities of **learn3- k -term-dnf** shown in figure 4.2 are polynomial for $k = O(\log n)$. Moreover, the number of equivalence queries that it uses is

	Running Time	Membership Queries	Equivalence Queries
learn1- k -term-dnf	$O(kt + nk^2c^2)$	$kq + kc$	kc
learn2- k -term-dnf	$O(kt + nc^{O(k)})$	$kq + c^{k+1}$	k or $k + 1$
learn3- k -term-dnf	$O(kt + c2^{O(k)} \log n)$	$kq + c2^{O(k)} \log n$	k or $k + 1$

TABLE 4.1. Summary of results for learning k -term DNF using a general procedure **produce-terms**. c is the number of terms produced by **produce-terms**, q is the number of membership queries it uses, and t is a bound on its running time.

optimal (by the result of Bshouty and Cleve [15]). The result, however, is in Angluin's unrestricted model.

The complexities of **learn2- k -term-dnf** shown in figure 4.2 are polynomial for $k = O(\sqrt{\log n})$, and the result is in Angluin's restricted model. Moreover, the number of equivalence queries is optimal.

Both results of **learn2- k -term-dnf** and **learn3- k -term-dnf** are the best known results for learning k -term DNF formulas in the corresponding model (either the restrictive or the unrestrictive).

	Running Time	Membership Queries	Equivalence Queries
learn1-k-term-dnf [12]	$n^2(\log n)^{O(1)}2^{O(k)}$	$n^2(\log n)^{O(1)}2^{O(k)}$	$(\log n)^{O(1)}2^{O(k)}$
learn2-k-term-dnf	$n^2(\log n)^{O(k)}2^{O(k^2)}$	$n^2(\log n)^{O(k)}2^{O(k^2)}$	k or $k + 1$
learn3-k-term-dnf	$n^2(\log n)^{O(1)}2^{O(k)}$	$n^2(\log n)^{O(1)}2^{O(k)}$	k or $k + 1$

TABLE 4.2. *Summary of results when using the second version of **produce-terms** based on Blum-Rudich's algorithm. The first line states the results of Blum and Rudich, and the other two lines state our results.*

CHAPTER 5

Read- k Sat- j DNF

A DNF formula is *Read- k* if every variable appears in it at most k times, and it is *Sat- j* if every assignment satisfies at most j terms in it. We say that a DNF formula is *Read- k Sat- j* if it is both *Read- k* and *Sat- j* . The class of *Read- k Sat- j* DNF formulas was proven to be learnable by Aizenstein and Pitt [2]. The running time of their algorithm is $O(n^{4kj+2j+2})$ and it uses at most $k(n^{j+2} + n^{j+1})$ membership queries and at most $kn^{2kj+j+1}$ equivalence queries. In section 5.1 we give an outline of their algorithm and we show how to reduce the number of equivalence queries to m (the number of terms in the target formula), if m is given *a priori* to the learner, or to $m+1$ if m is not known. This is a dramatic improvement over the algorithm of Aizenstein and Pitt, since we prove in section 5.2 that m cannot exceed $4\sqrt{jk(k-1)n}$. Both the running time and the number of membership queries in our algorithm are $n^{\Theta(kj)}$. In section 5.3 we establish lower bounds that match the stated upper bounds.

5.1. A learning algorithm

5.1.1. An outline of Aizenstein and Pitt's algorithm. The algorithm of Aizenstein and Pitt [2] is essentially **learn1- k -term-dnf**. The algorithm (shown in figure 5.1) maintains a set \mathcal{T} of terms (initialized to the empty set). At the beginning of the main loop, the algorithm makes an equivalence query with the hypothesis h that is a disjunction of the terms in \mathcal{T} . If the counterexample a is positive, **produce-terms** is called with the input a , and the terms it returns are added to \mathcal{T} . If the

counterexample is negative, then it is used to eliminate bad terms from \mathcal{T} . This loop is repeated until the equivalence query oracle returns “yes”.

We need now to describe the procedure **produce-terms** that encapsulates Aizensteins and Pitt’s technique for producing terms. We start with a few definitions. A term t is *almost satisfied by an assignment a with respect to a literal x* if x is the only literal in t assigned 0 by a . We denote an assignment a with the literal x fixed to be 0 (respectively 1) by $a_{x \leftarrow 0}$ (respectively, $a_{x \leftarrow 1}$). The *sensitive set* of a is defined by

$$\text{sensitive}(a) = \{\text{literal } x \mid a \text{ assigns 1 to } x \text{ and } f(a) \neq f(a_{x \leftarrow 0})\}.$$

Thus, if $x \in \text{sensitive}(a)$, flipping x in a will cause the value of f to change. Let S and S' be sets of literals and let i be an integer. We say that S' is an *i -variant* of S if $S \subseteq S'$ and $|S' - S| \leq i$. An assignment a' is said to be an *i -variant* of an assignment a if the number of bits on which a and a' disagree is at most i . For a term t , let $\text{lits}(t)$ denote the set of literals in t .

The following theorem (lemma 8 in the original paper [2]) suggests a way to implement **produce-terms**.

THEOREM 5.1. *If a is a satisfying assignment for the target Read- k Sat- j DNF formula f satisfying the set of terms $T = \{t_1, t_2, \dots, t_q\}$ in f , then there exists an assignment a' which is a j -variant of a such that for some term t in T $\text{lits}(t)$ is a $2kj$ -variant of $\text{sensitive}(a')$.*

The procedure **produce-terms** does the following. Given a positive example a , it produces all j -variant assignments of a , and for each such one a' , it produces all terms t such that $\text{lits}(t)$ is a $2kj$ -variant of $\text{sensitive}(a')$. The procedure **produce-terms** returns all the terms produced. The above theorem guarantees that at least one of the terms returned appears in the target formula and is satisfied by a , which fulfills the requirement from **produce-terms**.

```

learn-read- $k$ -sat- $j$ -dnf
1  $\mathcal{T} \leftarrow \emptyset$ .
2 repeat
3   let  $h$  be the disjunction of all terms in  $\mathcal{T}$  (if  $\mathcal{T} = \emptyset$  then  $h \leftarrow \text{false}$ ).
4    $a \leftarrow \text{Equiv}(h)$ .
5   if  $a = \text{"yes"}$  then return  $h$ .
6   if  $a$  is a positive counterexample
7     then call produce-terms( $a$ ) and add the terms it returns to  $\mathcal{T}$ .
8   else ( $a$  is a negative counterexample)  $\mathcal{T} \leftarrow \mathcal{T} - \{t \in \mathcal{T} \mid t(a) = 1\}$ .
9 until done.

```

FIGURE 5.1. An algorithm for learning Read- k Sat- j DNF

We now analyze the running time of **produce-terms** and find the number of membership queries it uses. There are $\binom{n}{j}$ j -variants of a . For each one we find the sensitive set, each of which requires $n + 1$ membership queries. Therefore the number of membership queries of **produce-terms** is $(n + 1)\binom{n}{j} \leq n^{j+1}$. For each j -variant of a **produce-terms** produces at most $\binom{n}{2kj}$ terms, so the number of terms **produce-terms** returns is at most $\binom{n}{2kj}\binom{n}{j} \leq n^{2kj}n^j$. The running time is $n^{\Theta(kj)}$.

5.1.2. Our refinement. Let m be the number of terms in the target Read- k Sat- j DNF formula. In this section we show how to refine the algorithm of Aizenstein and Pitt by reducing the number of equivalence queries to m if m is known *a priori* or to $m + 1$ otherwise.

Our refinement is essentially **learn3- k -term-dnf**, that is, in step 7 of **learn-read- k -sat- j -dnf** (figure 5.1) we add to \mathcal{T} only those terms that imply the target formula. If we accomplish this task, the counterexamples that we get will be all positive. After at most m calls to **produce-terms**, h contains all terms of f so $f \Rightarrow h$. Since we add to \mathcal{T} only terms that imply f , we have that $h \Rightarrow f$ in each iteration of the main loop. Therefore, after adding all m terms of f to \mathcal{T} , we get that $f \Rightarrow h$ and $h \Rightarrow f$ which implies that $h \equiv f$. If m is known then there is no need to ask the $(m + 1)$ st equivalence query (which will return the answer “yes”). Otherwise, we do not know

when to stop and the $(m + 1)$ st is needed.

We now show how to find if a term $t \Rightarrow f$. We remind the reader, that given a term t (not equivalent to false or true) the partial assignment p_t is defined as follows:

$$p_t[i] = \begin{cases} 1 & \text{if } v_i \text{ appears unnegated in } t \\ 0 & \text{if } v_i \text{ appears negated in } t \\ * & \text{if } v_i \text{ does not appear in } t, \end{cases}$$

for every $i = 1, \dots, n$. This partial assignment induces a projection h_{p_t} defined by: for every assignment a , $h_{p_t}(a) = h(p_t|a)$, where $p_t|a$ is defined by:

$$(p_t|a)[i] = \begin{cases} p_t[i] & \text{if } p_t[i] \neq * \\ a[i] & \text{if } p_t[i] = *, \end{cases}$$

for every $i = 1, \dots, n$. For ease of notation we let h_t to denote h_{p_t} . A boolean function h is a *tautology* if $h(a) = 1$ for every assignment a .

Let t be a term and let f be the target Read- k Sat- j DNF formula. By lemma 4.4 $t \Rightarrow f$ if and only if f_t is a tautology. The next lemma shows that f_t is also a Read- k Sat- j DNF formula.

LEMMA 5.2. *Let f be a Read- k Sat- j DNF formula and let t be a term. Then f_t is a Read- k Sat- j DNF formula.*

Proof: It is clear that f_t is Read- k because we do not add literals to the f in order to get f_t . We show it is Sat- j . Let $f = t_1 \vee \dots \vee t_m$, then f_t would be $t'_1 \vee \dots \vee t'_m$, where every t'_i is the projection $(t_i)_t$ (the terms of f_t are not necessarily distinct, and, moreover, some of the terms t'_1, \dots, t'_m may be equivalent to true or false). Let a be an assignment that satisfies some term t'_i . Since t'_i is $(t_i)_t$ it is true that $t'_i(a) = t_i(p_t|a)$, so the assignment $p_t|a$ satisfies the term t_i . The formula f is Sat- j so the assignment

$p_t|a$ satisfies at most j terms, and thus, a satisfies at most j term in f_t . \square

Thus, our problem is reduced to testing whether a Read- k Sat- j DNF formula is a tautology.

LEMMA 5.3. *Let f be a Read- k Sat- j DNF formula and let a be any assignment. Then f is a tautology iff $f(a') = 1$ for every $2(k+1)j$ -variant a' of a .*

In order to prove this lemma, we will use lemma 7 in [2].

LEMMA 5.4. *Let a be any assignment satisfying a Read- k Sat- j DNF formula f , then there are at most $2kj$ literals y such that there is a term in f that is almost satisfied by a with respect to \bar{y} .*

The following lemma handles the case when a falsifies f .

LEMMA 5.5. *Let a be any assignment falsifying a Read- k Sat- j DNF formula f , then there are at most $2(k+1)j$ literals y such that there is a term in f that is almost satisfied by a with respect to \bar{y} .*

Proof: Let t be a term that contains n literals defined as follows. If $a[i] = 1$ then the literal v_i appears in t , otherwise ($a[i] = 0$) the literal \bar{v}_i appears in t . Observe that t is satisfied *only* by the assignment a . Also let $f' = f \vee t$. Note that f' is Read- $(k+1)$ since f is Read- k and by adding t we added one occurrence for every variable. Also note that f' is Sat- j . This follows from two facts: The first is that if an assignment satisfies t , this assignment must be the above a , and we know that a does not satisfy any other term in f' (otherwise, a would be a satisfying assignment for f). The second fact is that if an assignment a' satisfies *other* terms in f' then it satisfies the same terms in f , so it satisfies at most j terms. In both cases, every assignment satisfies at most j terms in f' , so f' is Sat- j . Using lemma 5.4, we get that there are at most $2(k+1)j$ literal y such that there exists a term in f' that is almost satisfied by a with respect to \bar{y} .

Define the following:

$$Y'(a) = \{\text{literal } y \mid \text{there is a term in } f' \text{ that is almost satisfied by } a \text{ with respect to } \bar{y}\}.$$

In the above paragraph, we have proved that $|Y'(a)| \leq 2(k+1)j$. Let y be a literal such that there is term in f that is almost satisfied by a with respect to \bar{y} . The terms in f that are almost satisfied by a with respect to \bar{y} are also in f' , therefore $y \in Y'(a)$. Since $|Y'(a)| \leq 2(k+1)j$, we get that the number of such y 's is at most $2(k+1)j$, which proves the lemma. \square

Proof of lemma 5.3: We consider the two directions of the claim.

(\Rightarrow) This direction is trivial, for if f is a tautology then its value is 1 on every assignment.

(\Leftarrow) Suppose f is not a tautology. We show a $2(k+1)j$ -variant of a that falsifies f .

Since f is not a tautology, there is an assignment w for which $f(w) = 0$. If w is a $2(k+1)j$ -variant of a then we are done. So assume that w is not a $2kj$ -variant of a . We will use the following definitions.

$$Y(w) = \{\text{literal } y \mid \text{there is a term in } f \text{ that is almost satisfied by } w \text{ with respect to } \bar{y}\}.$$

$$D(a, w) = \{\text{literal } y \mid w \text{ assigns 1 to } y \text{ and } a \text{ assigns 0 to it}\}.$$

By lemma 5.5 we have that $|Y(w)| \leq 2(k+1)j$, and by our assumption that w is not a $2(k+1)j$ -variant of a , $|D(a, w)| > 2(k+1)j$. Therefore, there exists a literal $y \in D(a, w) - Y(w)$. Since w falsifies all terms in f and since $y \notin Y(w)$, the assignment $w' = w_{y \leftarrow 0}$ still falsifies f . Moreover, by lemma 5.5, we know that $|Y(w')| \leq 2(k+1)j$, and $|D(a, w')| = |D(a, w)| - 1$, since w' and a both assign 0 to y . If $|D(a, w')| > 2(k+1)j$, we can repeat the same process: find a literal $y \in D(a, w') - Y(w')$, flip it in w' to get a new assignment w'' that falsifies f . For the new w'' we know that $|Y(w'')| \leq 2(k+1)j$ and $|D(a, w'')| = |D(a, w')| - 1$. This process can be repeated until we get an assignment a' that falsifies f and for which $|D(a, a')| \leq 2(k+1)j$. This a' is a $2(k+1)j$ -variant of a (since $|D(a, a')| \leq 2(k+1)j$), and it falsifies f , so we are done. \square

To summarize, in step 7 we add to \mathcal{T} only those terms that imply the target formula. In order to test if a term t implies f , we check if $f_t(a') = 1$ for every $2(k+1)j$ -variant a' of *any* assignment a (e.g. a is all 0's). The number of membership queries needed for every test is $\binom{n}{2(k+1)j} \leq n^{2(k+1)j}$. The procedure **produce-terms** is called $m \leq kn$ times (m is the number of terms in f), and every time it returns at most $\binom{n}{2kj}$ terms. Therefore, the number of additional membership queries needed in step 7 is at most $n^{2(k+1)j} n^{2kj} = n^{4kj+2j}$. The additional running time is clearly $n^{\Theta(kj)}$.

We thus have proved the following theorem.

THEOREM 5.6. *The class of Read- k Sat- j DNF formulas is learnable using m equivalence queries, if m is known a priori, or $m + 1$ otherwise, where m is the number of terms in the target formula. Both the running time and the number of membership queries are $n^{\Theta(jk)}$.*

5.2. Number of terms in a Read- k Sat- j DNF

In this section we show that the number of terms in a Read- k Sat- j DNF formula ($k > 1$) cannot exceed $4\sqrt{jk(k-1)n}$. For $k = 1$, the number of terms cannot exceed j .

We use the following definition. A *disjoiner between* two terms t and t' is a literal that appears positively in one of them but negatively in the other. If a literal ℓ is a disjoiner between t and t' , then we say that ℓ *disjoins* t and t' . Obviously, if there is a disjoiner between two terms, there cannot be an assignment that satisfies both. Conversely, if there is no disjoiner between any two terms of a set S of terms, then there is an assignment that satisfies all the terms in the set S .

5.2.1. The case $k = 1$. Observe that since the Read-Once Sat- j DNF formula f is Read-Once, there cannot be a disjoiner between any two terms in f . Otherwise, this would mean that a variable appears twice, which contradicts the fact that f is Read-Once. Therefore, there cannot be more than j terms. For, if there are more than j terms in f , we can find an assignment that satisfies all of them, since there

are no disjointers between any two terms in f . This would contradict the fact that f is Sat- j . We thus have proved the following theorem.

THEOREM 5.7. *The number of terms in a Read-Once Sat- j DNF formula is at most j .*

5.2.2. The case $k > 1$. In this section we prove that the number of terms in a Read- k Sat- j DNF formula ($k > 1$) is at most $4\sqrt{jk(k-1)n}$.

Let $G_f = (V, E)$ be the graph, induced by f , defined as follows. V is the set of terms in f , and $(u, v) \in E, u \neq v$, if and only if the terms u and v share some variable. An edge $(u, v) \in E$ is labeled with the set of variables that is shared by u and v . A graph is *simple* if it does not contain an edge of the form (u, u) , and any two vertices are connected by at most one edge. Observe that G_f is simple.

LEMMA 5.8. *G_f has the property that any $j + 1$ distinct vertices in G_f contain a pair of vertices that are connected by an edge.*

Proof: Suppose the claim is not true. This means that there are $j + 1$ terms in f for which no two share a variable. Therefore we can define an assignment that satisfies all of these $j + 1$ terms, contradicting the fact that f is Sat- j . \square

We are interested in bounding from below the number of edges in G_f (i.e. $|E|$). For this purpose we look at the complement graph of G_f , $\overline{G}_f = (V, \overline{E})$. Let K_i denote the clique on i vertices, that is, a simple graph with i vertices in which there is an edge between any two vertices. By lemma 5.8 we obtain that \overline{G}_f does not contain K_{j+1} .

If we find an upper bound on the $|\overline{E}|$, we can find a lower bound on $|E|$, i.e. the number of edges in G_f .

A graph is *i -partite* if its vertices can be partitioned into i subsets so that no edge has both ends in any one subset (we refer to the subsets as *partitions*). A graph is *complete i -partite* if it is simple, i -partite and if every vertex in any partition is connected to all vertices outside the partition. Let $T_{i,p}$ denote the complete i -partite

graph on p vertices in which each partition has either $\lfloor p/i \rfloor$ or $\lceil p/i \rceil$ vertices. For a graph G , we use $\epsilon(G)$ to denote the number of edges in G .

We use the following lemmas. The first lemma is theorem 7.9 in [13]. The second lemma is exercise 1.2.9 in the same reference.

LEMMA 5.9. *If a graph G is simple and contains no K_{j+1} , then $\epsilon(G) \leq \epsilon(T_{j,m})$, where m is the number of vertices in G .*

LEMMA 5.10.

$$\epsilon(T_{j,m}) = \binom{m-h}{2} + (j-1) \binom{h+1}{2},$$

where $h = \lfloor m/j \rfloor$.

We are now ready to find an upper bound on the number of edges in \overline{G}_f .

LEMMA 5.11.

$$\epsilon(\overline{G}_f) \leq \frac{m(m+2)(j-1)}{2j}.$$

Proof: By lemma 5.8, \overline{G}_f does not contain a clique K_{j+1} , so we can use lemma 5.9 and lemma 5.10 to bound $\epsilon(\overline{G}_f)$ as follows.

$$\begin{aligned}
 (5) \quad \epsilon(\overline{G}_f) &\leq \binom{m-h}{2} + (j-1) \binom{h+1}{2} \\
 (6) \quad &= \binom{m - \lfloor m/j \rfloor}{2} + (j-1) \binom{\lfloor m/j \rfloor + 1}{2} \\
 (7) \quad &= \frac{(m - \lfloor m/j \rfloor)(m - \lfloor m/j \rfloor - 1)}{2} + (j-1) \frac{(\lfloor m/j \rfloor + 1)(\lfloor m/j \rfloor)}{2} \\
 (8) \quad &\leq \frac{(m - (m/j) + 1)(m - (m/j))}{2} + (j-1) \frac{(m/j + 1)(m/j)}{2} \\
 (9) \quad &= \left(\frac{(jm - m + j)(jm - m)}{2j^2} \right) + \left(\frac{(j-1)(m+j)m}{2j^2} \right) \\
 &= \frac{m(j-1)}{2j^2} ((jm - m + j) + (m + j)) \\
 &= \frac{m(j-1)}{2j^2} (jm + 2j) \\
 &= \frac{m(m+2)(j-1)}{2j}.
 \end{aligned}$$

Inequality 5 follows immediately from lemmas 5.9 and lemma 5.10. In line 6 we just substitute the value of h . Equality 7 follows from the fact that $\binom{a}{2} = \frac{a(a-1)}{2}$. We get line 8 using the fact that $\frac{m}{j} \geq \left\lfloor \frac{m}{j} \right\rfloor \geq \frac{m}{j} - 1$. We multiplied both numerator and denominator of the fractions in 8 to get 9. The rest of the equalities are straightforward. \square

We are now ready to get a lower bound on $\epsilon(G_f)$.

LEMMA 5.12.

$$\epsilon(G_f) \geq \frac{m(m-3j+2)}{2j}.$$

Proof: Recall that the number of vertices in G_f is m . Since G_f is the complement graph of \overline{G}_f , it is the case that $\epsilon(G_f) + \epsilon(\overline{G}_f) = \binom{m}{2}$. Using the upper bound on $\epsilon(\overline{G}_f)$ from lemma 5.11 we obtain

$$\begin{aligned} \epsilon(G_f) &\geq \binom{m}{2} - \frac{m(m+2)(j-1)}{2j} \\ &= \frac{m}{2j} (j(m-1) - (m+2)(j-1)) \\ &= \frac{m(m-3j+2)}{2j}. \end{aligned}$$

\square

The next lemma gives an upper bound on $\epsilon(G_f)$, and together with the previous lemma, it provides an upper bound on m .

LEMMA 5.13. $\epsilon(G_f) \leq \binom{k}{2}n$.

Proof: Every variable appears in at most k terms (and consequently, in at most k vertices in the graph G_f). Together with the fact that in a simple graph of k vertices there can be at most $\binom{k}{2}$ edges, this implies that every variable appears in at most $\binom{k}{2}$ labels. Therefore the total number of labels does not exceed $\binom{k}{2}n$, which proves the claim. \square

THEOREM 5.14. *Let m be the number of terms in a Read- k Sat- j DNF formula f ($k > 1$), then $m \leq 4\sqrt{jk(k-1)n}$.*

Proof: By lemma 5.12 $\epsilon(G_f) \geq \frac{m(m-3j+2)}{2j}$, and by lemma 5.13 $\epsilon(G_f) \leq \binom{k}{2}n$. Combining these two inequalities yields

$$\begin{aligned}
 \frac{m(m-3j+2)}{2j} &\leq \binom{k}{2}n \\
 m(m-3j) &\leq jk(k-1)n \\
 m^2 - 3mj + \frac{9}{4}j^2 &\leq jk(k-1)n + \frac{9}{4}j^2 \\
 m - \frac{3}{2}j &\leq \sqrt{jk(k-1)n + \frac{9}{4}j^2} \\
 (10) \quad m &\leq \sqrt{jk(k-1)n + \frac{9}{4}j^2} + \frac{3}{2}j \\
 (11) \quad &\leq \sqrt{jk(k-1)n + \frac{9}{4}jkn} + \frac{3}{2}\sqrt{jkn} \\
 &\leq \sqrt{jk(k-1)n + \frac{9}{4}jk(k-1)n} + \frac{3}{2}\sqrt{jk(k-1)n} \\
 &\leq \sqrt{4jk(k-1)n} + \frac{3}{2}\sqrt{jk(k-1)n} \\
 &\leq 4\sqrt{jk(k-1)n}.
 \end{aligned}$$

Line 11 follows from line 10 since $j \leq m$, and m cannot exceed the number of literals in the formula. The number of literals is at most kn , because the formula is read- k . \square

5.3. Lower bounds

In this section we prove lower bounds on the number of equivalence queries needed to identify the target Read- k Sat- j DNF formula. We separate between two cases. The first case is Read- k Sat- j DNF, when $k > 1$, and the second case is Read-Once Sat- j DNF.

5.3.1. The case $k > 1$. We remind the reader that the variables over which the formulas are defined are v_1, \dots, v_n . We use v_i^0 to denote v_i , and we use v_i^1 to denote

\bar{v}_i . Thus $v_i^{c_i}$ is v_i , if $c_i = 0$ and is \bar{v}_i if $c_i = 1$. If a variable is negated then we say its *sign* is 1, otherwise its sign is 0. Also, recall that a disjoiner between two terms is a variable that appears positively in one of them but negatively in the other.

We define a subclass \mathcal{C}' of Read- k Sat- j DNF formulas ($k > 1$) and prove that the learner must ask at least $\sqrt{j(k-1)n/2}$ equivalence queries to learn \mathcal{C}' .

5.3.1.1. *The definition of the target class.* Let $s = \left\lfloor \sqrt{\frac{n}{2j(k-1)}} \right\rfloor$, and let $m' = 1 + (k-1)s$. A formula $f \in \mathcal{C}'$ is of the form $f(x) = f_1(x) \vee f_2(x) \vee \dots \vee f_j(x)$, where each f_i is a Read- k Sat-Once DNF formula. We refer to every f_i as a *subformula*. A variable cannot appear in more than one subformula. In every subformula there are m' terms. Since we want every subformula f_i to be Sat-Once, every term in f_i must contain a literal that disjoins it from every other term in f_i . We first describe f_1 . The other subformulas are defined similarly *using a different set of variables*.

Every term t in f_1 is of two parts: the *necessary* part and the *new* part. The necessary part contains only disjoiners, i.e. literals that disjoin t from other terms in f_1 . The new part contains s literals none of which appears in the new part of any other term. Let $t_1, \dots, t_{m'}$ be the terms in f_1 . We use the letter Π to refer to the new part of the term. More specifically, Π_i is the new part of term t_i , and it is

$$\Pi_i = \bigwedge_{j=(i-1)s+1}^{is} v_j^{c_j},$$

for some $c_{(i-1)s+j} \in \{0, 1\}$, $j = 1, \dots, s$. The term t_i contains, besides Π_i , literals that disjoins it from all the terms that precede it (i.e. t_j , $1 \leq j < i$). The first literal in Π_i (i.e. $v_{(i-1)s+1}^{c_{(i-1)s+1}}$) is used to disjoin t_i from the next consecutive $k-1$ terms. Then we cannot use this variable anymore because it has appeared k times (once in the new part of t_i , that is Π_i , and $k-1$ other times as a disjoiner in $k-1$ terms). In order to disjoin t_i from the following $k-1$ terms we use the second literal in Π_i , that is $v_{(i-1)s+2}^{c_{(i-1)s+2}}$, and then we use the third literal, etc. For ease of notation we let Π^p denote the negation of the p th literal in Π .

To summarize, the terms in f_1 are:

$$\begin{aligned}
t_1 &= \Pi_1 \\
t_2 &= \Pi_1^1 \wedge \Pi_2 \\
t_3 &= \Pi_1^1 \wedge \Pi_2^1 \wedge \Pi_3 \\
&\dots \\
t_{(k-1)} &= \Pi_1^1 \wedge \Pi_2^1 \wedge \Pi_{k-2}^1 \wedge \Pi_{k-1} \\
t_k &= \Pi_1^2 \wedge \Pi_2^1 \wedge \Pi_{k-2}^1 \wedge \Pi_{k-1}^1 \wedge \Pi_k \\
t_{k+1} &= \Pi_1^2 \wedge \Pi_2^2 \wedge \Pi_{k-2}^1 \wedge \Pi_{k-1}^1 \wedge \Pi_k^1 \wedge \Pi_{k+1} \\
&\dots \\
t_{m'} &= \Pi_1^s \wedge \Pi_2^s \wedge \dots \wedge \Pi_{m'-1}^1 \wedge \Pi_{m'}.
\end{aligned}$$

Note that every literal in t_1 appears in t_1 and in other $k-1$ terms. So the number of terms is $1+(k-1)s$ which is m' . Note also that there is no assignment that can satisfy any two of the above terms, because each two are disjointed by a disjoiner. Therefore, f_1 is indeed Sat-Once. f_1 is Read- k because no variable appears more than k times (once in the new part of some term, and at most $k-1$ other times as a disjoiner). Finally, observe that the number of variables appearing in f_1 is sm' .

The other subformulas (f_2, \dots, f_j) are defined similarly to f_1 but with a distinct set of variables. More specifically, f_1 is defined over the variables $v_1, \dots, v_{sm'}$, f_2 is defined over the next consecutive sm' variables, that is $v_{sm'+1}, \dots, v_{2sm'}$, etc.

Observe that since each subformula is Sat-Once, $f = f_1 \vee \dots \vee f_j$ is Sat- j . Also, since each subformula is defined over a distinct set of variables, and since every one is Read- k , it is the case that f is Read- k . Finally, the number of variables used in f

is

$$\begin{aligned}
jsm' &= js(1 + (k-1)s) = js + j(k-1)s^2 \\
&\leq j\sqrt{\frac{n}{2j(k-1)}} + j(k-1) \left(\sqrt{\frac{n}{2j(k-1)}} \right)^2 \\
&= \sqrt{\frac{n}{2(k-1)}} + \frac{n}{2}.
\end{aligned}$$

For $k \geq 2$ it is true that $\sqrt{\frac{n}{2(k-1)}} \leq \frac{n}{2(k-1)} \leq \frac{n}{2}$, therefore the number of variables used in f is less than n , so f is well-defined.

Observe also that the number of terms in f is

$$\begin{aligned}
m &= jm' = j(1 + (k-1)s) \\
&= j \left(1 + (k-1) \sqrt{\frac{n}{2j(k-1)}} \right) \\
&> j(k-1) \sqrt{\frac{n}{2j(k-1)}} = \sqrt{j(k-1)n/2}.
\end{aligned}$$

Example: Let $n = 100$, $j = 2$ and $k = 3$. We have $s = 3$ and $m' = 7$. We show some formula $f = f_1 \vee f_2$ in \mathcal{C}' (for the mentioned parameters). The Π 's of f_1 are as follows.

$$\Pi_1 = v_1 \bar{v}_2 \bar{v}_3, \Pi_2 = v_4 v_5 \bar{v}_6, \Pi_3 = \bar{v}_7 v_8 v_9, \Pi_4 = v_{10} v_{11} v_{12},$$

$$\Pi_5 = \bar{v}_{13} \bar{v}_{14} \bar{v}_{15}, \Pi_6 = v_{16} \bar{v}_{17} \bar{v}_{18}, \text{ and } \Pi_7 = \bar{v}_{19} \bar{v}_{20} v_{21}.$$

Therefore, f_1 is:

$$f_1 = \Pi_1 \vee \bar{v}_1 \Pi_2 \vee \bar{v}_1 \bar{v}_4 \Pi_3 \vee v_2 \bar{v}_4 v_7 \Pi_4 \vee v_2 \bar{v}_5 v_7 \bar{v}_{10} \Pi_5 \vee v_3 \bar{v}_5 \bar{v}_8 \bar{v}_{10} v_{13} \Pi_6 \vee v_3 v_6 \bar{v}_8 \bar{v}_{11} v_{13} \bar{v}_{16} \Pi_7.$$

The subformula f_2 is defined over the variables v_{22}, \dots, v_{42} .

5.3.1.2. The adversary. The lower bound here holds for $jk = o(n/(\log n)^2)$. We establish this bound using the techniques of Bshouty and Cleve [15].

THEOREM 5.15. For $jk = o(n/(\log n)^2)$, $k > 1$, and $m \leq \sqrt{j(k-1)n/2}$,

$$\mathcal{E}_U(\text{Read-}k \text{ Sat-}j \text{ DNF}_{n,m}) \geq m$$

$$\mathcal{E}_U(\text{Read-}k \text{ Sat-}j \text{ DNF}_n) \geq m + 1.$$

Proof: We prove that the lower bound holds for the class \mathcal{C}' defined above, by showing that the learner must ask at least m equivalence queries (or $m + 1$ if m is not known *a priori*). (If m is less than the number of terms in the formulas in \mathcal{C}' , we delete all the extra terms from each formula.)

Let f be the target formula in \mathcal{C}' . The goal of the adversary is to ensure that each equivalence query (accompanied with a polynomial number of membership query) will only help the learner to know at most one term in f .

Let m denote the number of terms in f . The learner's task is to find the signs of the variables in each of the formula's Π 's. Once this is done, the learner will be able to exactly identify the formula. Recall that the size of every Π is $s = \left\lfloor \sqrt{\frac{n}{2j(k-1)}} \right\rfloor$. Suppose the order of the literals in every Π is fixed. A vector $w \in \{0, 1\}^s$ is the *sign vector* of Π if the i th bit of w gives the sign of the i th literal in Π . As stated above, the learner's task is to learn the sign vectors of the Π 's.

Henceforth, we number the Π 's of the target formula: Π_1, \dots, Π_m . Let $\mathcal{P}_i \subseteq \{0, 1\}^s$ be the set of sign vectors each of which is a candidate for being the sign vector of Π_i . In other words, every vector in \mathcal{P}_i is consistent with the adversary's replies so far. At the beginning of the learning session the learner does not know the sign of any variable in any Π , so $\mathcal{P}_i = \{0, 1\}^s$, for $i = 1, \dots, m$. Lemma 5.16 (follows after this proof) shows that the initial size of every \mathcal{P}_i , $i = 1, \dots, m$, is super polynomial, provided that $jk = o(n/(\log n)^2)$. Later we will show that every membership query decreases the size of every \mathcal{P}_i by at most 1. These two facts imply that a polynomial number of membership queries cannot decrease the size of any \mathcal{P}_i to 1. In other words, a polynomial number of membership queries does not suffice to determine the sign vector of any Π .

The adversary's strategy in answering the membership and equivalence queries will be such that after e equivalence queries the learner knows only Π_1, \dots, Π_e but has gained no information about Π_{e+1}, \dots, Π_m .

Let e be the number of equivalence queries that have been answered so far in the learning session. The adversary maintains the following invariants:

- (1) For $1 \leq \ell \leq e$, \mathcal{P}_ℓ contains exactly one element (the sign vector of Π_ℓ , as known to the learner). In addition, \mathcal{P}_ℓ , $1 \leq \ell \leq e$, does not change in the future (so the adversary remains consistent).
- (2) For $\ell > e$, $|\mathcal{P}_\ell|$ is super-polynomial.

Let $g(x)$ be the disjunction of the first (known) e terms in the target formula f .

Answering membership queries: Suppose the learner asks $\text{MQ}(a)$. The adversary answers as follows.

MQ1: $g(a) = 1$.

In this case, since $g(a) = 1$, it is the case that $f(a) = 1$, so the adversary answers 1. The learner has gained no information by this reply.

MQ2: $g(a) = 0$.

In this case the adversary answers 0, and updates the candidate sets $\mathcal{P}_{e+1}, \dots, \mathcal{P}_m$ as follows. Recall that the length of the sign vector of each Π is s . Let b_i , $e < i \leq m$ be the i th block of size s in a . Note that a satisfies some Π_i if and only if the sign vector of Π_i is the complement vector of the block b_i . Therefore, in order to ensure that a falsifies all Π_i , $e < i \leq m$, we eliminate the complement of b_i from \mathcal{P}_i . Thus, we eliminate at most one element from each \mathcal{P}_i , $e < i \leq m$.

Answering equivalence queries: For each equivalence query $\text{Equiv}(h)$, the adversary answers as follows.

EQ1: $g \not\equiv h$, that is, there exists an assignment a such that $h(a) = 0$ and $g(a) = 1$.

In this case the adversary returns “no” accompanied with the counterexample a . The learner has gained no information because of this reply. (This case is similar to case **MQ1**.)

In order to maintain the invariants, the adversary picks an arbitrary element π in \mathcal{P}_{e+1} , and updates \mathcal{P}_{e+1} to be exactly $\{\pi\}$.

EQ2: $h \not\equiv g$, that is, there exists an assignment a such that $h(a) = 1$ and $g(a) = 0$.

In this case the adversary answers “no” accompanied with a as a counterexample. In order to be consistent in future replies, the adversary updates the \mathcal{P}_i 's, $e < i \leq m$, as in case **MQ2**.

In addition, in order to maintain the invariants, the adversary picks an arbitrary element π in \mathcal{P}_{e+1} , and updates \mathcal{P}_{e+1} to be exactly $\{\pi\}$.

EQ3: $g \equiv h$.

In this case, the adversary discloses a new term as follows. The adversary picks some element $\pi \in \mathcal{P}_{e+1}$, and returns the answer “no” accompanied with the counterexample a built in the following manner. The i th block, $1 \leq i \leq e$ is chosen to be the (unique) element in \mathcal{P}_i . The $(e+1)$ st block is π , and the other blocks are fixed arbitrarily. The adversary updates \mathcal{P}_{e+1} by setting it to be the complement of π . Observe that, by the way it was constructed, a falsifies g , so it falsifies h . However, the adversary disclosed the $(e+1)$ st term and it is satisfied by a , so it satisfies the target formula.

Observe that an equivalence query discloses exactly one term from the target formula, so the first invariant is maintained. Observe also that as a result of a membership query or an equivalence query, the size of every \mathcal{P}_i , $e+1 < i \leq m$, decreases by at most 1. Since the initial size of every \mathcal{P}_i is super-polynomial, and since the learner is allowed only a polynomial number of membership queries (and of course equivalence queries) the size of \mathcal{P}_i , $e+1 < i \leq m$, remain super-polynomial. So, the

second invariant is maintained as well. \square

LEMMA 5.16. *The initial size of every \mathcal{P}_i is super polynomial in n , provided that $jk = o(n/(\log n)^2)$. That is, if the initial size of \mathcal{P}_i is expressed by $S(n)$, then $S(n) = n^{\omega(1)}$.*

Proof: We know that $s = \sqrt{\frac{n}{2j(k-1)}} > \sqrt{\frac{n}{2jk}}$. We have the following implications:

$$\begin{aligned}
 s &> \sqrt{\frac{n}{2jk}} \\
 jk &> \frac{n}{2s^2} \\
 \frac{n}{2s^2} &= o\left(\frac{n}{(\log n)^2}\right) \\
 s &= \omega(\log n).
 \end{aligned}
 \tag{12}$$

Line 12 follows from the preceding line, by the fact that $jk = o(n/(\log n)^2)$ (the greater-than sign in the preceding line is implicit in the $o()$ notation).

We have thus proved that $S(n) = 2^{\omega(\log n)}$. This quantity is super-polynomial in n . An easy way to see this is by comparing $\omega(\log n)$ to $(\log n)\omega(1)$. Then we get that $S(n) = 2^{(\log n)\omega(1)} = n^{\omega(1)}$ which is super-polynomial (the power of n is greater than any constant). \square

5.3.2. The case $k = 1$. By theorem 5.7, the number of terms in a Read-Once Sat- j DNF formula is at most j terms. We display a class \mathcal{C}'' of Read-Once Sat- j DNF formulas, and show that the learner must ask at least j equivalence queries in order to identify the target formula. This bound holds for $j = o(n/\log n)$.

5.3.2.1. Definition of the target class \mathcal{C}'' . A formula $f \in \mathcal{C}''$ contains exactly j terms, each of size $s = \lfloor \frac{n}{j} \rfloor$. The i th term t_i , $1 \leq i \leq j$, is

$$t_i = \bigwedge_{p=(i-1)s+1}^{is} x_p^{c_p},$$

for some signs $c_{(i-1)s+1}, \dots, c_{is} \in \{0, 1\}$.

Here, as in the previous lower bound, the learner's task is to identify the sign vector of every term. The adversary replies to the learner are the same as those in the lower bound for the case $k > 1$. Note that since $j = o(n/\log n)$, the initial size of every candidate set is $2^s = 2^{\lceil n/j \rceil} = 2^{\omega(\log n)}$ which is super polynomial as claimed above.

We have thus proved:

THEOREM 5.17. *For $j = o(n/\log n)$,*

$$\mathcal{E}_U(\text{Read-Once Sat-}j \text{ DNF}) = j.$$

5.4. Remarks

We have found the upper bound $m \leq \sqrt{jk(k-1)n}$ on the number of terms in a Read- k Sat- j DNF formula. The lower bound that we established assumes that $m \leq \sqrt{j(k-1)n/2}$, so there is a gap of a factor \sqrt{k} between the bounds. However, the algorithm of Aizenstein and Pitt [2] and our refinement are polynomial only for j and k constants. Therefore the gap between the presented bounds is a constant. To conclude, we have presented upper bounds that are polynomial when j and k are constants. Also, we have presented lower bounds that match the upper bounds (within a constant factor) when j and k are constant.

CHAPTER 6

Monotone DNF Formulas

In this section we let Monotone DNF_n represent the class of monotone DNF formulas on n variables, and we let $\text{Monotone DNF}_{n,m}$ be the subset of those formulas that have at most m terms. A learning algorithm is allowed time and membership queries polynomial in n and m (although for the former class m is not known to the learner *a priori*). We are able to prove a tight bound on the number of equivalence queries needed to exactly identify the target monotone DNF formula. We, first, prove two lower bounds, one for the case when the number of terms in the target formula m is known *a priori*, and the other case is when m is not known *a priori*. Later, we show two upper bounds, one for every case.

6.1. Lower Bounds

To prove lower bounds on the number of equivalence queries required to learn monotone DNF formulas, we prove the following key lemma demonstrating a trade off between membership and equivalence queries. The proof uses an adversary argument to show that for a certain subclass of monotone DNF formulas, membership queries reveal relatively little information.

LEMMA 6.1.

$$\mathcal{E}(\text{Monotone DNF}_{n,m}, q) \geq m - d$$

for any $0 < d < n$ satisfying $\left(\left\lfloor \frac{n}{d} \right\rfloor\right)^d > q + m - d$.

Proof: For ease of exposition we consider the case where d divides n evenly. We prove the result holds for the following subclass of monotone DNF formulas. The target formula includes the following d terms (which we call the “fixed” terms, since we give them to the learner in advance).

$$\begin{aligned} t_1 &= v_1 v_2 \cdots v_{\frac{n}{d}} \\ t_2 &= v_{\frac{n}{d}+1} v_{\frac{n}{d}+2} \cdots v_{2\frac{n}{d}} \\ &\vdots \\ t_d &= v_{\frac{d-1}{d}n+1} \cdots v_n. \end{aligned}$$

The remaining terms t_{d+1}, \dots, t_m will each include all but one of the variables from each fixed term, that is from each t_i with $i \leq d$ (so each such term contains $n - d$ variables). All the monotone DNF formulas obtained in this fashion represent distinct functions. The task of the learner, then, is to decide whether each of the possible $(n/d)^d$ terms of the specified form are in the target formula. Let \mathcal{T} denote the set of unknown terms t_{d+1}, \dots, t_m .

The adversary. The adversary’s replies to the learner are such that every membership query eliminates at most one term from \mathcal{T} , and every equivalence query either eliminates at most one term from \mathcal{T} or discloses at most one unknown term from the target formula. The target formula contains up to $m - d$ initially unknown terms. The membership queries may eliminate up to q of the possible terms, but there are at least $(n/d)^d - q > m - d$ (by the choice of d) remaining terms about which the learner has no information. And $m - d$ of these terms may appear in f in any combination. The following is the adversary’s reply to the learner. Let g be a disjunction of the fixed terms and the known terms from \mathcal{T} , i.e.

$$g(x) = t_1 \vee \cdots \vee t_d \vee t_{d+1} \vee \cdots \vee t_{d+i},$$

where the terms t_{d+1}, \dots, t_{d+i} are already known to the learner.

Answering membership queries: Suppose the learner has asked $\text{MQ}(a)$, for some assignment a . There are two cases.

MQ1: $g(a) = 1$.

In this case, since all the terms of g appear in the target formula f , it is the case that $f(a) = 1$ independent of the other unknown terms. The adversary answers 1, and the learner has gained no information.

MQ2: $g(a) = 0$.

a contains at least d 0's (if it contains less than d 0's, then there would be some fixed term that is satisfied by a). If a contains exactly d 0's then a satisfies exactly one term t from \mathcal{T} . The adversary answers 0 and eliminates t from \mathcal{T} . If a contains more than d 0's, then all terms in \mathcal{T} are falsified by a , so by answering 0 to the learner, we do not eliminate any term from \mathcal{T} .

Answering equivalence queries: Suppose the learner has asked $\text{Equiv}(h)$. Here again we have multiple cases.

EQ1: $g \not\equiv h$, that is, there exists an assignment a such that $g(a) = 1$ and $h(a) = 0$. The adversary, answers “no” accompanied with the assignment a . Like case MQ1 the learner has gained no information from this counterexample.

EQ2: $h \not\equiv g$, that is, there exists an assignment a such that $h(a) = 1$ and $g(a) = 0$. This case is similar to case MQ2. The adversary answers “no” with the counterexample a . It updates the set of candidates \mathcal{T} as in case MQ2, that is, by eliminating at most one term from \mathcal{T} .

EQ3: $g \equiv h$.

In this case the adversary discloses a new unknown term t from \mathcal{T} , and it outputs as a counterexample the assignment a that fixes 1 on all the variables in t but 0 elsewhere (so a falsifies both the fixed terms and any other term in g). Since g is logically equivalent to h , and since a falsifies g , it falsifies h , whereas it satisfies the disclosed term t , so it serves as a

counterexample.

□

We apply this lemma to prove lower bounds for both cases where m is known or unknown. We first consider the case in which m is a known input parameter for the learner.

THEOREM 6.2. *For $m < 2^{\sqrt{n}}$,*

$$\mathcal{E}(\text{Monotone DNF}_{n,m}) \geq m - \theta \left(\frac{\log m + \log n}{\log n - \log \log m} \right)$$

Proof: Let the number of membership queries used by the learner be $q < (mn)^k$ for some constant k . The condition for applying lemma 6.1 is that the chosen d satisfies

$$(14) \quad \left(\frac{n}{d} \right)^d + d > m + q.$$

We will choose d such that it satisfies the condition

$$(15) \quad \left(\frac{n}{d} \right)^d > (mn)^{k+1}.$$

Such d will obviously satisfy the primary condition 14, since $q < (mn)^k$, $m < mn$, and $d > 0$. By taking the logarithm of both sides of 15, we get that d must satisfy:

$$d(\log n - \log d) > (k+1)(\log m + \log n).$$

By dividing both sides by $\log n - \log d$, we get that the condition on d is

$$(16) \quad d > \frac{(k+1)(\log m + \log n)}{(\log n - \log d)}.$$

Choose d to be

$$d = \frac{2(k+1)(\log m + \log n)}{(\log n - 2 \log \log m)}.$$

(Note that since $m < 2^{\sqrt{n}}$, $\log n - 2 \log \log m > 0$, so $d > 0$). We show that this d satisfies condition 16, for $m < 2^{\sqrt{n}}$. We need to verify that

$$\frac{2(k+1)(\log m + \log n)}{(\log n - 2 \log \log m)} > \frac{(k+1)(\log m + \log n)}{(\log n - \log d)}.$$

By cancelling similar factors and terms we get that we need to verify that

$$(17) \quad \log d < \frac{\log n}{2} + \log \log m.$$

Taking the logarithm of d we obtain that

$$(18) \quad \log d = 1 + \log(k+1) + \log(\log m + \log n) - \log(\log n - 2 \log \log m).$$

For $a, b \geq 2$, we have that $a + b \leq ab$, so $\log(a + b) \leq \log(ab) = \log a + \log b$. Applying this to $\log(\log m + \log n)$, we get that $\log(\log m + \log n) \leq \log \log m + \log \log n$, when $\log m \geq 2$, and $\log n \geq 2$. For $\log m < 2$ (that is, $m < 4$) it is the case that $\log(\log m + \log n) \leq \log(2 + \log n) \leq \log(2 \log n) = 1 + \log \log n$. So, for sufficiently large n it is true that

$$\log(\log m + \log n) \leq 1 + \log \log m + \log \log n \leq \log \log m + \frac{\log n}{4}.$$

Also, since $m < 2^{\sqrt{n}}$, it is true that $\log m < \sqrt{n}$, and that $\log \log m < \frac{\log n}{2}$. Therefore $\log n - \log \log m > 1$, for $n \geq 4$. This implies that $\log(\log n - \log \log m) > 0$, for $n \geq 4$. In addition, since k is a constant, for sufficiently large n we have that $1 + \log(k+1) < \frac{\log n}{4}$. Combining all this together in 18 we obtain that

$$\begin{aligned} \log d &= 1 + \log(k+1) + \log(\log m + \log n) - \log(\log n - \log \log m) \\ &< \frac{\log n}{4} + \log \log m + \frac{\log n}{4} \\ &= \frac{\log n}{2} + \log \log m, \end{aligned}$$

for sufficiently large n . This proves that d does satisfy condition 17, and thus it satisfies condition 14. \square

Note that we have not attempted to choose d such that $(n/d)^d$ is superpolynomial in n and m . What we showed in the above theorem is that for *any* polynomial $p(n, m)$, there exists a constant c such that for

$$(19) \quad d = \frac{c(\log m + \log n)}{\log n - \log \log m},$$

the quantity $(n/d)^d$ is greater than $p(n, m)$. A calculation similar to the one above shows that for any constant c , and for the d in 19, the quantity $(n/d)^d$ is bounded by some polynomial. We will use this to establish an upper bound later.

In the case where the learning algorithm is not given an *a priori* upper bound on the number of terms, we may prove a slightly stronger result.

THEOREM 6.3. *For any $0 < k_n < n - \omega(\log n)$ with $k = \omega(1)$, and n sufficiently large*

$$\mathcal{E}(\text{Monotone DNF}_n) \geq m - k_n$$

Proof: Pick $d = k_n$. If at any point after having made e equivalence queries the algorithm has made a number of membership queries superpolynomial in n and e (answered by the strategy above), the adversary decides there is only one more term in f , which means the algorithm has made superpolynomial number of membership queries. Thus the algorithm can only ever make a number of membership queries polynomial in n . The result follows since $(n/k_n)^{k_n}$ grows superpolynomially in n . \square

6.2. Upper Bounds

In this section we describe an algorithm that matches the above lower bounds. We begin by briefly describing Angluin's algorithm [5] for learning a monotone DNF formula using at most $m + 1$ equivalence queries (or m equivalence queries, if m is known *a priori*).

6.2.1. Preliminaries. A *prime implicant* of a boolean formula f is a conjunction t (not containing contradictory literals) such that t implies f , but no proper subterm of t implies f . For general DNF formulas the number of prime implicants may be exponentially larger than the number of terms. However, the next two lemmas establish that if f is a monotone DNF formula then the number of its prime implicants is at most m , where m is the number of terms in f , and that every prime

implicant of f contains no negated variables. For two assignments a and a' , we say that $a \leq a'$ if and only if $a[i] \leq a'[i]$, where we assume that $0 \leq 1$.

LEMMA 6.4. *A prime implicant t of a monotone DNF formula f contains no negated variables.*

Proof: Assume for contradiction that t does contain a negated variable \bar{v}_i , and let t' be a term that is obtained from t by dropping \bar{v}_i . We prove that t' implies f , which contradicts the fact that t is a prime implicant of f . Let a' be an assignment that satisfies t' , we show that it satisfies f . If a' assigns 0 to v_i then a' assigns 1 to \bar{v}_i , so $t(a') = 1$ implying $f(a') = 1$ (because $t \Rightarrow f$). Otherwise, a' assigns 1 to v_i . Consider the assignment a obtained from a' by flipping bit i in a' (so $a[i] = 0$). The assignment a satisfies t , so $f(a) = 1$. But, since $a \leq a'$ and f is monotone, it must be that $f(a') = 1$. We thus have showed that $t' \Rightarrow f$. \square

The following lemma is the key lemma behind Angluin's algorithm.

LEMMA 6.5. *Let f be a monotone DNF formula with m terms. Then the number of prime implicants of f is at most m .*

Proof: Assume for contradiction that the number of prime implicants of f is greater than m , so there is some prime implicant T that does not appear in f . By definition of a prime implicant $T \Rightarrow f$. This implies, by lemma 4.4, that f_{p_T} is a tautology, where p_T is defined by: $p_T[i]$ is 1 if the variable v_i appears in T , and \star otherwise (from the previous lemma, T does not contain a negated variables, so p_T does not contain 0's). We now show an assignment that falsifies f_{p_T} . This implies that our primary assumption, that the number of prime implicants of f is greater than the number of the terms in f , is incorrect.

Let $f = t_1 \vee \dots \vee t_m$, so $f_{p_T} = t'_1 \vee \dots \vee t'_m$, where t'_i is the projection of t_i induced by p_T (that is, $t'_i = (t_i)_{p_T}$). None of the t'_i 's is equivalently true, because this would mean that p_T assigns 1's to all the variables of some t_i in f implying that t_i is a subterm of T . t_i cannot be equal to T since we have assumed that T is not a term

in f . Therefore, if t_i is a subterm, it must be a proper subterm. But this contradicts the fact that T is a prime implicant (because t_i implies f and it is a proper subterm of T). We conclude that every term in f_{p_T} is not equivalent to true. Also, since f is monotone, none of the terms in f_{p_T} contains a negated variable. Now consider the value of f_{p_T} on the assignment a that is all 0's (that is $a = 0_n$). The assignment a falsifies every term in f_{p_T} because every term in f_{p_T} contains an unnegated variable. Therefore, $f_{p_T}(a) = 0$, contradicting the fact that f_{p_T} is a tautology. \square

6.2.2. Angluin's algorithm. Given lemma 6.5, there is a fairly straightforward exact identification algorithm due to Angluin [5] (based on a previous PAC learning algorithm of Valiant [29]). We use each equivalence query to find a prime implicant of the target formula f . Our current hypothesis h is the disjunction of all known prime implicants (initially the always false hypothesis). Then each counterexample a can be used to find a new prime implicant by walking towards the all zeros example (using membership queries to decide which variables should be set in the counterexample to 0). Let a' be the assignment that we get from a by walking it towards 0_n (while preserving the condition that we do not flip a 1 in a unless the new assignment keeps satisfying f). It is easy to see that the resulting example a' will satisfy exactly the variables of some new prime implicant. So the disjunction of the variables set to 1 in a' gives a prime implicant t of f (that is satisfied by a'). The new implicant t is not found in h , because $h(a') = 0$ (this follows from the facts that $a' \leq a$, h is monotone and $h(a) = 0$). This technique requires $m + 1$ equivalence queries and mn membership queries.

6.2.3. Our refinement. A simple optimization allows us to find the first prime implicant without making an equivalence query. Monotonicity implies that if the target formula is not identically 0 then $f(1_n) = 1$ (1_n is the all 1's example). This can be used to find the first term, reducing our equivalence query requirement to m . That observation gives us the special case (for $k = 0$) of an algorithm we present

```

learn-monotone-dnf( $n, k$ )
1 let  $h$  be identically false.
2 repeat
3     find a positive counterexample  $a$  as follows.
4         if  $h$  contains at most  $k$  terms
5             then search exhaustively for an  $a$  with at most  $k$  bits set to 0
              such that  $h(a) = 0$  and  $f(a) = 1$  (return  $h$  if none is found).
6             else ( $h$  contains more than  $k$  terms)
7                  $a \leftarrow \text{Equiv}(h)$ .
8                 if  $a$  is “yes” then return  $h$ .
9     walk  $a$  towards  $0_n$  while preserving  $f(a) = 1$ .
10    let  $t$  be the conjunction of all variables in  $a$  set to 1.
11     $h \leftarrow h \vee t$ .
12 until done.

```

FIGURE 6.1. An algorithm to learn monotone DNF with $m - k$ equivalence queries and $O\left(k \binom{n}{k}^k + mn\right)$ membership queries.

in figure 6.1. This new algorithm can reduce the number of equivalence queries by an arbitrary number k . This savings is at a cost of time and membership queries exponential in k , but this will be enough to show that our previous lower bounds are tight.

LEMMA 6.6. *There is an exact identification algorithm for monotone DNF formulas that takes as input n and a non-negative integer $k < m$, and learns the target formula using $m - k$ equivalence queries and $O\left(k \left(\frac{n}{k}\right)^k + mn\right)$ time and membership queries.*

Proof: This algorithm (shown in figure 6.1) finds $k + 1$ prime implicants of f before making any equivalence queries. The key observation is that as long as we have discovered at most k prime implicants, then if there is any counterexample there will be one that has only k variables set to 0 (and we can exhaustively test all possible such counterexamples, of which there are at most $\binom{n}{k} = O((n/k)^k)$). This is because any positive counterexample fails to satisfy our k prime implicants. Given that such a counterexample exists, there is some set of k or fewer variables covering our prime implicants that are set to 0 in the counterexample, and given that those variables are 0 in some positive counterexample, the example that has only those variables set to 0 will still be both a positive example and a counterexample. Thus for the first $k + 1$ terms, we use brute force enumeration to find counterexamples. After this we use $m - k$ equivalence queries to learn the remaining $m - k - 1$ terms in the standard manner. \square

Based on this technique we prove two upper bounds for learning monotone DNF formulas. In the case where m is not known, the learner needs $m - \Theta(1)$ equivalence queries. When m is known, we prove that the number of queries is reduced to $m - \Theta\left(\frac{\log m + \log n}{\log n - \log \log m}\right)$. Note that these bounds differ only when m is superpolynomial in n . They both follow from lemma 6.6 by substituting the appropriate quantities for k .

THEOREM 6.7. For any constant $c > 0$,

$$\mathcal{E}(\text{Monotone DNF}_n) \leq m - c.$$

Proof: Choose k to be c . Obviously $c(n/c)^c$ is polynomial for every constant c . \square

THEOREM 6.8. For any constant $c > 0$,

$$\mathcal{E}(\text{Monotone DNF}_{n,m}) \leq m - c \left(\frac{\log m + \log n}{\log n - \log \log m} \right).$$

Proof: For any constant c set k to be

$$k = c \left(\frac{\log m + \log n}{\log n - \log \log m} \right).$$

An argument similar to that in the proof of theorem 6.2 shows that $k(n/k)^k$ is bounded by some polynomial. \square

CHAPTER 7

Horn Sentences

We remind the reader that a Horn sentence* is a CNF formula in which every clause is either of the form $\text{true} \rightarrow v$ or of the form $v_{i_1} v_{i_2} \cdots v_{i_j} \rightarrow \ell$, where $v, v_{i_1}, \dots, v_{i_j}$ are variables and ℓ is either a variable, false, or true (we refer to clause in either of these two forms by *Horn clause*). The left part of a Horn clause is called the *antecedent* and the right part is called the *consequent*. A Horn clause is falsified by an assignment a if and only if a satisfies its antecedent and falsifies its consequent.

In this chapter we let Horn Sentence_n represent the set of Horn sentences over n variables, and we let $\text{Horn Sentence}_{n,m}$ be the subset of those formulas that have at most m clauses. We show a lower bound of $\Omega(\frac{mn}{\log m + \log n})$ on the number of equivalence queries needed to identify the target Horn sentence.

7.1. Lower Bound

In this section we prove our lower bound by showing that it holds for the following subclass of Horn Sentences. Suppose there is an algorithm that learns the class of Horn sentences in time less than $(mn)^c$. Let $d = \lceil (c+1)(\log n + \log m) \rceil$ and $q = \lfloor n/2d \rfloor$. Divide the $2dq$ variables v_1, \dots, v_{2dq} into q blocks each of which contains $2d$ variables. Specifically, for $1 \leq i \leq q$, block B_i contains variables $v_{2d(i-1)+1}, \dots, v_{2di}$. Given a

*We could equivalently use the term “formula” rather than “sentence”, however, the notion “Horn sentence” is more common in the literature.

vector x we use $x[B_i]$ to denote the portion of x that corresponds to block B_i . That is, $x[B_i]$ is $(x_{2d(i-1)+1}, \dots, x_{2di})$.

For each of the q blocks of variables we construct a Horn sentence in the following manner. Let y_1, \dots, y_{2d} be the $2d$ variables in block B_i . We define[†]

$$P_i = \bigwedge_{j=1}^d (y_{2j-1}y_{2j} \rightarrow y_{(2j+1) \bmod 2d}) \wedge (y_{2j-1}y_{2j} \rightarrow y_{(2j+2) \bmod 2d}).$$

So for example if $d = 3$ we have

$$P_i = (y_1y_2 \rightarrow y_3)(y_1y_2 \rightarrow y_4) \cdots (y_5y_6 \rightarrow y_1)(y_5y_6 \rightarrow y_2).$$

OBSERVATION 7.1. *For $1 \leq i \leq q$, P_i has the property that if both variables in any pair y_{2j-1}, y_{2j} are 1 then it will be false unless all $2d$ variables are 1.*

Proof: Fix $1 \leq i \leq q$. Suppose for some pair y_{2j_0-1}, y_{2j_0} , $1 \leq j_0 \leq d$, both variables are assigned 1 by some assignment a , and $P_i(a) = 1$. We prove that a assigns 1 to all $2d$ variables. Since $P_i(a) = 1$, all its clauses are satisfied by a , including the clauses $y_{2j_0-1}y_{2j_0} \rightarrow y_{(2j_0+1) \bmod 2d}$ and $y_{2j_0-1}y_{2j_0} \rightarrow y_{(2j_0+2) \bmod 2d}$. Since both y_{2j_0-1} , and y_{2j_0} are assigned 1 by a , then $y_{(2j_0+1) \bmod 2d}$ and $y_{(2j_0+2) \bmod 2d}$ must be also assigned 1 by a . We now look at the clauses

$$y_{(2j_0+1) \bmod 2d} \wedge y_{(2j_0+2) \bmod 2d} \rightarrow y_{(2j_0+3) \bmod 2d}, \text{ and } y_{(2j_0+1) \bmod 2d} \wedge y_{(2j_0+2) \bmod 2d} \rightarrow y_{(2j_0+4) \bmod 2d},$$

and similarly prove that the next pair of variables (that is $y_{(2j_0+3) \bmod 2d}$ and $y_{(2j_0+4) \bmod 2d}$) are assigned 1 by a . We continue this inductively, showing that all variables are assigned 1 by a . \square

Let $\mathcal{S}^{(dq)} \subset \{0, 1\}^{2dq}$ be the set of bit strings for which each consecutive pair consists of a 1 and 0. That is: $\mathcal{S}^{(dq)} = \{(s_1, \dots, s_{2dq}) \mid (s_{2j-1}, s_{2j}) \text{ is } (0, 1) \text{ or } (1, 0) \text{ for } 1 \leq j \leq dq\}$. For any vector $s \in \mathcal{S}^{(dq)}$ define $I(s) = \{j \mid s_j = 1\}$, and for each $s \in \mathcal{S}^{(dq)}$ let

$$R_s = \left(\bigwedge_{j \in I(s)} v_j \right) \rightarrow 0.$$

[†]Although P_i is a formula defined over the $2d$ variables in block B_i , we use $P_i(x)$ to denote $P_i(x[B_i])$.

OBSERVATION 7.2. For any $x \in \{0, 1\}^{2dq}$ and $s \in \mathcal{S}^{(dq)}$, $R_s(x) = 0$ if and only if $x_j = 1$ for all $j \in I(s)$.

Finally, for $s_1, \dots, s_t \in \mathcal{S}^{(dq)}$ (t will be determined later), let

$$F_{s_1, \dots, s_t} = P_1 \wedge \dots \wedge P_q \wedge R_{s_1} \wedge \dots \wedge R_{s_t},$$

and let

$$\mathcal{C} = \left\{ F_{s_1, \dots, s_t} \mid s_1, \dots, s_t \text{ are in } \mathcal{S}^{(dq)} \text{ and are distinct} \right\}.$$

THEOREM 7.3. For $m - n = \Omega(m)$, $\mathcal{E}(\text{Horn Sentences}_{n,m}) = \Omega\left(\frac{mn}{\log n + \log m}\right)$.

Proof: We prove that the above lower bound holds for the class \mathcal{C} defined above. Since the number of clauses in each P_i is $2d$, there are $2dq < n$ clauses in $P_1 \wedge \dots \wedge P_q$ so fix $t = m - 2dq$, to get that in every F_{s_1, \dots, s_t} in \mathcal{C} , the number of clauses is at most m . Since

$$tq = (m - 2dq)q \geq (m - n)q = \Omega\left(\frac{mn}{\log n + \log m}\right),$$

the desired result will follow if the adversary can force the learner to make tq equivalence queries before obtaining exact identification.

Let f be the target function. Observe that the learner knows $P_1 \wedge \dots \wedge P_q$ before the learning session begins. The goal of the adversary is to ensure that each equivalence query (combined with a polynomial number of membership queries) will only help the learner to determine one block of some s_i (i.e. one of $s_i[B_1], \dots, s_i[B_q]$). Since there are tq such blocks, once this goal is achieved the result will follow.

For ease of exposition, we further divide s_1, \dots, s_t each into q blocks each containing $2d$ bits. We denote these blocks by $b_1, \dots, b_q, b_{q+1}, \dots, b_{(t-1)q}, b_{(t-1)q+1}, \dots, b_{tq}$. The adversary's strategy in answering the membership and equivalence queries will be such that after e equivalence queries the learner will know only b_1, \dots, b_e but has gained no information about b_{e+1}, \dots, b_{tq} . We say that b_ℓ is *known* if $\ell \leq e$ and *unknown* if $\ell > e$.

Let $D_\ell^{(e)}$ denote the values for b_ℓ that are consistent with all examples seen by the learner after e equivalence queries have been answered. During the proof we will often focus on the elements of $D_\ell^{(e)}$ that are in block i of some s_j . Thus for $1 \leq i \leq q$, let

$$\mathcal{D}_{B_i}^{(e)} = \{D_\ell^{(e)} \mid \ell = (j-1)q + i \text{ for } 1 \leq j \leq t\}.$$

Note that for all j and before asking any membership query, $D_j^{(0)} = \mathcal{S}^{(d)}$ and thus $|D_j^{(0)}| = 2^d \geq (mn)^{c+1}$ at the beginning of the learning session.

Let e be the number of equivalence queries that have been answered so far in the learning session. The adversary will maintain the following invariants.

- (1) For $1 \leq \ell \leq e$, $D_\ell^{(e)} = \{b_\ell\}$. That is, b_1, \dots, b_e are known.
- (2) For $1 \leq \ell_1 < \ell_2 \leq e$, $D_{\ell_1}^{(e)} \cap D_{\ell_2}^{(e)} = \emptyset$. That is, b_1, \dots, b_e are distinct.
- (3) For $\ell > e$, $(D_1^{(e)} \cup \dots \cup D_e^{(e)}) \cap D_\ell^{(e)} = \emptyset$. That is, b_1, \dots, b_e are not included in the set of candidates for b_{e+1}, \dots, b_{tq} .
- (4) For $D_1, D_2 \in \mathcal{D}_{B_i}^{(e)}$ such that $|D_1| > 1$ and $|D_2| > 1$, $D_1 = D_2$. That is, all unknown values in a given block have the same set of candidates remaining.
- (5) For any ℓ , if $|D_\ell^{(e)}| = 1$ then $D_\ell^{(w)} = D_\ell^{(e)}$ for $w > e$. That is, once b_ℓ is known \mathcal{D}_ℓ does not change.
- (6) Let Q_e be the number of membership and equivalence queries asked by the learner up to (and including) the e^{th} equivalence query. Then

$$|D_\ell^{(e)}| \geq (mn)^{c+1} - Q_e \text{ for } \ell > e.$$

Notice that since the running time is assumed to be less than $(mn)^c$, we have that $Q_e < (mn)^c$, it follows that $|D_\ell^{(e)}| \geq (mn)^c$ for $\ell > e$. We now define the strategy that will be used by the adversary to respond to the queries. Each query will enable the learner to determine only one of the tq blocks b_1, \dots, b_{tq} and further can eliminate at most one element from each $D_\ell^{(e)}$ for $\ell > e$. Thus adversary can force tq equivalence queries as desired.

After e equivalence queries have been answered, $r = \lfloor e/q \rfloor$ is the largest j such that

s_j is completely known, and $p = e - qr$ is the index of the last known block within s_{r+1} . Let I_e be the indices of the elements of s_{r+1} that are known to be 1. That is $I_e = \{j \mid j \in I(s_{r+1}[B_i]) \text{ for } 1 \leq i \leq p.\}$ Now let

$$R_e^* = \left(\bigwedge_{j \in I_e} v_j \right) \wedge \left(\bigwedge_{j=2dp+1}^{2dq} v_j \right) \rightarrow 0.$$

Thus R_e^* contains all variables whose corresponding indices in s_{r+1} are known to be 1 and all variables corresponding to the unknown elements in s_{r+1} .

OBSERVATION 7.4. *The antecedent of R_e^* is a superset of the antecedent of $R_{s_{r+1}}$ and thus $R_e^*(x) = 0$ implies that $R_{s_{r+1}}(x) = 0$.*

Let $g_e(x) = P_1(x) \wedge \cdots \wedge P_q(x) \wedge R_{s_1}(x) \wedge \cdots \wedge R_{s_r}(x) \wedge R_e^*(x)$. Applying Observation 7.4 it follows that for any $1 \leq e \leq tq$, if $g_e(x) = 0$ then $f(x) = 0$.

Answering a membership query: For each membership query, $\text{MQ}(a)$, the adversary responds as follows.

Case MQ1: $g_e(a) = 0$.

In this case the adversary replies 0. Since $g_e(a) = 0$ implies $f(a) = 0$ no information is given to the learner by this answer.

Case MQ2: $g_e(a) = 1$ and there exists $i \in \{1, \dots, dq\}$ such that $(a_{2i-1}, a_{2i}) = (0, 0)$.

In this case the adversary returns 1. Since $(a_{2i-1}, a_{2i}) = (0, 0)$ it follows that $R_s(a) = 1$ for any s , and thus no information is given to the learner by this answer.

Case MQ3: $g_e(a) = 1$ and for all $i \in \{1, \dots, dq\}$, $(a_{2i-1}, a_{2i}) \neq (0, 0)$.

Since $P_i(a[B_i]) = 1$ for all blocks i , by observation 7.1 we know that $a[B_i]$ is either all 1's or an element of $\mathcal{S}^{(d)}$. If $a[B_i]$ contained all 1s for all $1 \leq i \leq q$, it would follow that $R_e^*(a) = 0$. Thus, there exists an i_0 such that $a[B_{i_0}] \in \mathcal{S}^{(d)}$. The adversary returns 1 and removes $a[B_{i_0}]$ from the set of candidates for all blocks b_ℓ that are not known and correspond to B_{i_0} . That is for all $D \in \mathcal{D}_{B_{i_0}}^{(e)}$

such that $|D| > 1$, update $D \leftarrow D \setminus a[B_{i_0}]$. Note that after this update if each unknown r_j is selected from its associated D then we are assured that $g_e(x) = 1$ for all vectors x .

Answering the $(e + 1)$ st equivalence query: For each equivalence query, $\text{Equiv}(h)$, the adversary responds as follows.

Case EQ1: $h \not\equiv g_e$.

That is, there exists a vector a such that $h(a) = 1$ and $g_e(a) = 0$.

The adversary will handle this situation just as it did in case MQ1 where the learner asked the membership query $\text{MQ}(a)$ for which $g_e(a) = 0$. Finally, to maintain the invariants, the adversary selects an arbitrary $u \in D_{e+1}^{(e)}$ and sets $D_{e+1}^{(e+1)} \leftarrow u$. Let $p = (e + 1) - q \lfloor e/q \rfloor$. For each $D_\ell^{(e)} \in \mathcal{D}_{B_p}^{(e)}$ such that $|D_\ell^{(e)}| > 1$, the adversary sets $D_\ell^{(e+1)} \leftarrow D_\ell^{(e)} \setminus \{u\}$. Also e is incremented in all other $D_\ell^{(e)}$.

Case EQ2: $g_e \not\equiv h$.

That is, there exists a vector a such that $h(a) = 0$ and $g_e(a) = 1$.

The adversary will handle this situation just as it did in cases MQ2 and MQ3 where the learner asked the membership query $\text{MQ}(a)$ for which $g_e(a) = 1$. As in case EQ1, the adversary then updates the candidate sets to maintain the invariants.

Case EQ3: $h \equiv g_e$.

In this case we will take advantage of the fact for all ℓ , $D_\ell^{(e)}$ satisfies the invariants. Observe that all updates made in the above cases preserve these invariants. By invariant 4 it follows that for all $D_1, D_2 \in \mathcal{D}_{B_p}^{(e)}$ for which $|D_1| > 1$ and $|D_2| > 1$, $D_1 = D_2$ where p is the block number of b_{e+1} . That is, b_{e+1} corresponds to $s_{r+1}[B_p]$ where $r = \lfloor (e + 1)/q \rfloor$. For any such $D \in \mathcal{D}_{B_p}^{(e)}$ for which $|D| > 1$ select some $u \in D$ and set $s_{r+1}[B_p] = u$. Consider the example x in which $x[B_i] = 1_{2d}$ for $i \neq p$ and $x[B_p] = u$. By Invariant 3, $u \notin D_j^{(e)}$ for $j < e$, and thus it follows that $h(x) = 1$. Since

$R_{e+1}^*(x) = 0$ it follows that $g_{e+1}(x) = 0$ and thus x can be returned as the counterexample. Finally, as in case EQ1, the adversary updates the candidate sets to maintain the invariants. \square

CHAPTER 8

Boolean read-once formulas over various bases

In this chapter we prove an upper bound on the number of equivalence queries needed to identify read-once formulas. This is achieved as a consequence of a more general result, showing that an algorithm that makes use of equivalence queries only to generate justifying assignments (defined below) needs to make only $O(n/\log n)$ queries. This is an improvement from a previous technique that uses n queries [8, 18], and immediately gives us improved upper bounds for various classes of read-once formulas and non-monotone switch configurations. These upper bounds are tight by the work of Bshouty and Cleve [15].

In this chapter we consider the following classes of read-once formulas. Let $\text{ROF}_n(B)$ denote the set of read once-formulas whose gates are labeled with functions from B (the “basis”). Let B_k denote the basis of all boolean functions over k inputs, for a constant k . A *switch configuration* can be informally described as a black box with n electrical switches. The invisible part of the black box is an arbitrary interconnection of the switches in the box. The learner tries to learn this interconnection or an “equivalent” one using *switch operations*. A switch operation consists of setting an individual switch to either ON or OFF, and observing the output of the black box as displayed by a lamp: if the lamp lights up then the output of the box is 1, otherwise it is 0. The learner is required to return an interconnection I of the switches that has the property that a set of ON switches causes the lamp to light up in the black box if and only if the corresponding set of switches does so in the interconnection I .

Raghavan and Schach [27] have shown how to represent a switch configuration by a monotone boolean formula, so the class of switch configurations is a (proper) subset of monotone boolean formulas. They have also proved that every monotone read-once formula can be represented by a switch configuration. They have shown an algorithm that uses only membership queries and learns switch configurations. When the parity of the switches is not known then the switch configuration corresponds to a non-monotone boolean read-once formula. These switch configurations can be learned using membership queries and equivalence queries. The equivalence queries are needed only to find justifying assignments for the variables.

Let $\text{Switch Configurations}_n$ denote the set of n element switch configurations (in the general non-monotone case where the parity of each switch is not known a priori).

It follows from the work of Bshouty and Cleve [15] that

$$\begin{aligned}\mathcal{E}(\text{ROF}_n(\text{AND}, \text{OR}, \text{NOT})) &= \Omega\left(\frac{n}{\log n}\right), \\ \mathcal{E}(\text{ROF}_n(B_k)) &= \Omega\left(\frac{n}{\log n}\right), \\ \mathcal{E}(\text{Switch Configurations}_n) &= \Omega\left(\frac{n}{\log n}\right).\end{aligned}$$

8.1. Generating justifying assignments with a minimal number of equivalence queries

In this section we describe a technique of generating justifying assignments with $O(n/\log n)$ equivalence queries that can then be used to get algorithms that match the above lower bounds.

8.1.1. Definitions. We start with few definitions. A class \mathcal{C} is *closed under zero projection* if for any function $f \in \mathcal{C}$, fixing some variables of f to 0 produces a function still in \mathcal{C} . A *justifying assignment* for a variable v is an assignment whose classification changes if the value of the variable v is changed. Among other things, the justifying assignment is a witness to the fact that the given variable is relevant.

Define the vector l_m to be $(\overbrace{l, \dots, l}^m)$ where $l \in \{0, 1, \star\}$. For an input vector $x = (x_1, \dots, x_n)$ and a set of variables $V = \{v_{i_1}, \dots, v_{i_k}\}$, we denote $x(V) = (x_{i_1}, \dots, x_{i_k})$. Recall that a partial assignment is an input setting that assigns \star to some of the variables (to indicate the variable is unassigned). For a partial assignment p and an assignment a , $p|a$ denotes the assignment that replaces the stars of p with the corresponding values in a . For a partial assignment p and a boolean function f , f_p is the projection of f induced by p and is defined by $f_p(a) = f(p|a)$. For an assignment a and a variable v the assignment $b = a_{\neg v}$ is the assignment that satisfies $b[v] = \neg a[v]$ and $b[v'] = a[v']$ for any variable $v' \neq v$.

Given two assignments a and b such that $f(a) \neq f(b)$, the procedure $\text{walk}(a, b)$ is a procedure that continues to flip bits in a that are different from b , while keeping $f(a) \neq f(b)$. It works as follows. It keeps scanning a attempting to flip bits on which a and b disagree, while keeping $f(a) \neq f(b)$, until it can no longer do so, and then it returns the new a . Every time $\text{walk}(a, b)$ scans a , it succeeds in flipping some bit in a . Since there are at most n bits on which a and b disagree, walk scans a at most n times. In each scan it attempts to flip at most n bits (for each flipping it asks a membership query to see if the flipping is successful or not). The number of membership queries and the running time are, therefore, $O(n^2)$. The procedure generates a new assignment a' such that for any variable v , if $a'(v) \neq b(v)$ then $f(a'_{\neg v}) \neq f(a')$, so a' is a justifying assignment for v .

8.1.2. The standard transformation. We now describe the standard transformation to produce a set of justifying assignments using n equivalence queries. We assume that we are given a procedure that learns the class \mathcal{C} using only membership queries and justifying assignments for the variables in the target formula (such a procedure exists for all the classes mentioned above [8, 16, 18, 27]).

Suppose we have justifying assignments for some subset Y of the variables (initially empty), and suppose those justifying assignments all agree on setting the variables

in $V \setminus Y$ to 0. Then if p is the partial assignment that sets \star to all variables in Y and 0 to all variables in $V \setminus Y$, we can use the membership and justifying assignment algorithm for \mathcal{C} to learn a hypothesis h equivalent to f_p (the condition that \mathcal{C} is closed under zero projections implies that f_p is in \mathcal{C}). We make an equivalence query on h . If we get a counterexample y then $h(y) \neq f(y)$, but $h \equiv f_p$, so $f_p(y) \neq f(y)$. By definition of f_p , it is true that $f_p(y) = f(p|y) \neq f(y)$ so we can use $\text{walk}(y, p|y)$ to find a justifying assignment for one or more new variables. We then repeat with a p that assigns values (i.e. 0's) to strictly fewer variables, and when $p = \star_n$ we are done.

8.1.3. Our refinement. We now present an improved transformation that finds at least $\Omega(\log n)$ new variables with each equivalence query. Recall that an (n, k) -universal set is a set $\{b_1, \dots, b_t\} \subseteq \{0, 1\}^n$ such that every subset of k variables assumes all of its 2^k possible assignments in the b_i 's. Cohen and Zémor [19] have shown how to build an (n, k) -universal set of size $k^{O(1)}2^{2k} \log n$.

THEOREM 8.1. *Let \mathcal{C} be a class of boolean formulas that is closed under zero projections. If \mathcal{C} is learnable in polynomial time from $M(n)$ membership queries, given justifying assignments for all the relevant variables, then for any $\epsilon > 0$ there is a $q = O(n^{1+\epsilon}M(n) + n^3)$ such that*

$$\mathcal{E}(\mathcal{C}, q) \leq \frac{n}{\lfloor (\epsilon/4) \log n \rfloor}.$$

Proof: The algorithm for this reduction is shown in figure 8.1. As before, there is a main loop where each iteration begins by running the membership query and justifying assignment subroutine to learn an $h \equiv f_p$ for the p that assigns 0 to the variables in $V \setminus Y$, using known justifying assignments A for the variables in Y . But before we ask the equivalence query $h \equiv f$, we also learn a family of f_{p_i} 's determined by an $(n, \lfloor (\epsilon/4) \log n \rfloor)$ -universal set of size $t = (\lfloor (\epsilon/4) \log n \rfloor)^{O(1)} 2^{\lfloor (2\epsilon/4) \log n \rfloor} \log n \leq n^\epsilon$. We define f_{p_i} (for $i = 1$ to t) to be the partial assignment that sets the variables in $V \setminus Y$ as in the i 'th element of the universal set. In other words, every possible assignment of values to some subset of $\lfloor (\epsilon/4) \log n \rfloor$ variables from $V \setminus Y$ is realized

by some f_{p_i} . To learn each f_{p_i} , we test whether $f_p(x) = f_{p_i}(x)$ for all justifying assignments in A and for all the membership queries made by the justifying assignment algorithm when learning f_p . If this is the case then the justifying assignments algorithm outputs the same hypothesis for both target functions, and the correctness of the algorithm implies that $f_p \equiv f_{p_i}$. If however we find some $f_p(x) \neq f_{p_i}(x)$ this implies (by definition) that $f(p|x) \neq f(p_i|x)$, and since those examples agree on all variables in Y , using **walk** we can find a new justifying assignment without making any equivalence queries.

Now we argue that if all f_{p_i} 's are equivalent to f_p , and we make an equivalence query on h , then we are able to find $\Omega(\log n)$ new justifying assignments from the counterexample. Let y be the counterexample returned by the equivalence oracle and let y' be y (we need to keep y unchanged, so we will do all our work on y'). We start by walking our counterexample y towards $p|y$, to give us at least one new justifying assignment. After this walk y' satisfies $f(y') \neq f_p(y)$, and y' is a justifying assignment for all the variables $Y_1 \subset V \setminus Y$ on which y' differs from $p|y$. If $|Y_1| \geq \lfloor (\epsilon/4) \log n \rfloor$, we are done. If not, then there is some p_i that agrees with y' on all the variables in Y_1 . But we know $f_{p_i} \equiv f_p$, so $f_{p_i}(y) = f_p(y)$ and $f(y') \neq f_{p_i}(y)$. We now call **walk**($y', p_i|y$), and we are guaranteed that we find justifying assignments for variables in $V \setminus (Y \cup Y_1)$. Call these variables Y_2 . If $|Y_1| + |Y_2| \geq \lfloor (\epsilon/4) \log n \rfloor$, again, we are done. If not we can repeat with a different p_i that agrees with y' on $Y_1 \cup Y_2$, and so on.

To establish the bound on the number of membership queries stated in the theorem, observe that the main loop in the algorithm **find-justifying-assignments** has at most $\frac{n}{\lfloor (\epsilon/4) \log n \rfloor}$ iterations, since every iteration finds justifying assignments for at least $\lfloor (\epsilon/4) \log n \rfloor$ new variables. So the number of equivalence queries is at most $\frac{n}{\lfloor (\epsilon/4) \log n \rfloor}$. Step 5 uses $M(n)$ membership queries to learn f_p . Another $t(M(n) + n) \leq n^\epsilon(M(n) + n)$ membership queries are needed in step 9, to test every f_{p_i} on at most $M(n) + n$ assignments. There are at most $\frac{n}{\lfloor (\epsilon/4) \log n \rfloor}$ iterations, so the total number

```

find-justifying-assignments( $n$ )
1  initialize  $Y = \emptyset$ ,  $A = \emptyset$ .
2  let  $\{b_1, \dots, b_t\}$  be an  $(n, \lfloor (\epsilon/4) \log n \rfloor)$ -universal set of size  $t \leq n^\epsilon$ .
3  repeat
4      let  $p$  the partial assignment that assigns  $\star$ 's to all variables in  $Y$  and
        0's to all other variables.
5      learn a hypothesis  $h \equiv f_p$  (using membership queries and the justifying
        assignments  $A$  for the relevant variables  $Y$ ).
6      for  $i = 1$  to  $t$  do
7          define the partial assignment  $p_i$  as follows:
8              
$$p_i[j] = \begin{cases} \star & v_j \in Y \\ b_i[j] & \text{otherwise.} \end{cases}$$

9              test  $f_{p_i}$  on all justifying assignments in  $A$  and on all the membership
                queries made by the algorithm to learn  $f_p$ .
10             if on some point  $x$ ,  $f_{p_i}(x) \neq f_p(x)$ 
11                 then
12                     call walk( $p_i|x, p|x$ ) to find one or more new justifying
                        assignments.
13                     update  $Y$  and  $A$ , and iterate, that is goto step 3a
                        (having made no equivalence queries).
14              $y \leftarrow \text{Equiv}(h)$ .
15             if  $y$  is "yes" then return  $h$  and halt.
16             let  $y'$  be  $y$ .
17             repeat
18                 pick an  $i$  for which  $p_i$  and  $y'$  agree on all the new variables
                    added so far.
19                 call walk( $y', p_i|y$ ) to find new justifying assignments.
20                 update  $Y$  and  $A$ .
21             till this loop has added  $\lfloor \frac{\epsilon}{4} \log n \rfloor$  new variables to  $Y$ .
22  until done.

```

FIGURE 8.1. An algorithm to use only $n/\log n$ equivalence queries to generate justifying assignments.

of membership queries used in these two steps is at most

$$\begin{aligned} \left(\frac{n}{\lfloor (\epsilon/4) \log n \rfloor} \right) (n^\epsilon (M(n) + n) + M(n)) &\leq n (n^\epsilon (M(n) + n) + M(n)) \\ &= O(n^{1+\epsilon} M(n) + n^2). \end{aligned}$$

Membership queries are also needed in the **walk** procedure in steps 12 and 19. But every call to **walk** finds a justifying assignment for at least one new variables, so these step are performed at most n times. The total number of membership queries needed for these two steps is therefore $O(n^3)$. Adding this to the number of membership queries used in the other steps, we get that the total number is as claimed in the theorem. \square

Applying the above technique to previous algorithms [8, 16, 18, 27] we obtain the following result.

THEOREM 8.2. *The quantities $\mathcal{E}(\text{ROF}_n(\text{AND}, \text{OR}, \text{NOT}))$, $\mathcal{E}(\text{ROF}_n(B_k))$, and $\mathcal{E}(\text{Switch Configurations}_n)$ are all $O(n/\log n)$.*

In all these cases the transformation adds a factor of $O(n^{1+\epsilon})$ membership queries and running time to the original algorithm and saves a factor of $\lfloor (\epsilon/4) \log n \rfloor$ equivalence queries.

CHAPTER 9

Arithmetic read-once formulas

Let $\text{AROF}_n^{(\mathcal{F})}(+, \times, /, -)$ denote the class of n variable *arithmetic* read-once formulas over the basis of addition, subtraction, multiplication, and division over a field \mathcal{F} . The inputs are constants from \mathcal{F} , and the output is a value in $\mathcal{F} \cup \{\infty, 0/0\}$.

There is a polynomial time identification algorithm for this class that uses membership queries and n equivalence queries [16]. When the size of the field \mathcal{F} is at least $2n + 5$ then the algorithm does not use equivalence queries at all (however, the algorithm is randomized in this case). The lower bound, established by Bshouty and Cleve [15], of $\Omega(n \log |\mathcal{F}| / \log n)$ on the number of equivalence queries holds when the size of \mathcal{F} is $o(n / \log n)$. It is an open problem whether equivalence queries are essential when the size of \mathcal{F} falls in the gap between $\Theta(n)$ and $\Theta(n / \log n)$.

In this chapter we show an upper bound that meets the above lower bound. The tight bound on the number of equivalence queries proved here is when the size of \mathcal{F} is $o(n / \log)$. The algorithm uses equivalence queries only to generate justifying assignments, but it is not immediately obvious that we can apply the techniques we used for the boolean case, because of the difficulty of non-boolean variables.

We need to make only a slight change in our algorithm **find-justifying-assignments** to make it work for arithmetic read-once formulas. In step 2 we want the universal set to be over all values in \mathcal{F} . That is, every subset of k variables assumes all its $|\mathcal{F}|^k$ values in the universal set. Having made this change, the algorithm **find-justifying-assignments** learns the target arithmetic read-once formula.

Instead of dealing with a universal set that contains values from \mathcal{F} , we work with a universal set that contains only 0's and 1's, in which the values of \mathcal{F} are represented in binary. Since we want the universal set to contain all values in \mathcal{F} , we need $\log |\mathcal{F}|$ bits to represent every value. We look at the columns of the set as being grouped into blocks of $\log |\mathcal{F}|$ columns each, each block corresponds to a value in \mathcal{F} . The number of columns needed in the universal set is therefore $n \log |\mathcal{F}|$. We want that every subset of size k of the variables assumes all its possible field values, so we require that every $k \log |\mathcal{F}|$ columns in the universal set assume all the possible (binary) values. Thus, the universal set needed is an $(n \log |\mathcal{F}|, k \log |\mathcal{F}|)$ -universal set, and, by the work of Cohen and Zémor [19], its size is

$$(k \log |\mathcal{F}|)^{O(1)} 2^{2k \log |\mathcal{F}|} \log(n \log |\mathcal{F}|).$$

We want this quantity to be polynomial in n , so:

$$(k \log |\mathcal{F}|)^{O(1)} 2^{2k \log |\mathcal{F}|} \log(n \log |\mathcal{F}|) \leq n^c,$$

for some constant c . Taking the logarithm of both size, and canceling small terms, we get that k must satisfy:

$$k = \frac{c \log n}{\log |\mathcal{F}|}.$$

Using this k , every iteration of the **find-justifying-assignments** finds justifying assignments for k new variables. The number of iterations (or, equivalently, the number of equivalence queries) is at most

$$\frac{n}{k} = \frac{n \log |\mathcal{F}|}{c \log n},$$

which matches the lower bound.

One last remark is that after building the $(n \log |\mathcal{F}|, k \log |\mathcal{F}|)$ -universal set in step 2, we go over the set, translating the strings in every block to values of \mathcal{F} (this would make the rest of the algorithm cleaner).

THEOREM 9.1. *For any field \mathcal{F} , $\mathcal{E} \left(\text{AROF}_n^{(\mathcal{F})}(+, \times, /, -) \right)$ is $O(n \log |\mathcal{F}| / \log n)$.*

CHAPTER 10

Deterministic Finite State Automaton

In this section we present a lower bound on the number of equivalence queries needed to learn DFAs. Recall that n denotes the number of states in the target DFA, and let $k = |\Sigma|$, where Σ is the alphabet of the target DFA. To distinguish between the situations in which n is known or unknown to the learner, let $DFA_{n,\Sigma}$ denote the case when n is known and DFA_Σ denote the case in which n is not known.

Given that there is a polynomial-time algorithm that can determine if two DFAs are equivalent and in addition output a minimum length counterexample if they are not equivalent, it seems reasonable to assume that the equivalence oracle uses such an example. Thus the results presented here all assume that the counterexamples returned by the equivalence oracle have length at most n . These results can easily be generalized to have the complexity depend on the length of the longest counterexample received by the learner.

10.1. Lower bound

In this section we present our lower bound. Observe that this bound holds even if the learner knows n *a priori*. Also there is no restriction on the time used by the algorithm, just on the number of membership queries and counterexamples.

THEOREM 10.1. *For any constant $c \geq 1$,*

$$\mathcal{E}(DFA_{n,\Sigma}, n^c - 1) \geq \frac{n - 2}{c \log_k n}.$$

Proof Sketch: Let T be an arbitrary string of $n - 2$ elements from Σ . We shall let $T[i]$ denote the i th element of T . For ease of exposition when defining the transition function, let $T[n - 1]$ denote some special symbol that is not in Σ . Let $A \subseteq \{1, 2, \dots, \frac{n-2}{c \log_k n}\}$. The adversary will select T and A as the learning session progresses. The target DFA, U , will be defined as follows.

- The state set, $Q = \{0, \dots, n - 1\}$,
- the initial state $q_0 = 1$,
- the accepting state set $F = \{i \cdot c \log_k n + 1 \mid i \in A\}$,
- the transition function δ is defined as follows:
 - $\delta(0, \sigma) = 0$, for all $\sigma \in \Sigma$ (so state 0 is a dead state)
 - $\delta(q, \sigma) = 0$, for all $q \in Q$ and $\sigma \neq T[q]$
 - $\delta(q, \sigma) = q + 1$ for all $q \in Q$ and $\sigma = T[q]$.

At the beginning both T and A are empty. T is appended by a block of $s = c \log_k n$ symbols at a time. Let $S^{(i)} \subseteq \Sigma^s$, $i = 1, \dots, \frac{n-2}{s}$, be the set of strings each of which is candidate to be the i th block in T of size s . Initially, before asking any query, $S^{(i)} = \Sigma^s$ for every $i = 1, \dots, \frac{n-2}{s}$.

The adversary replies to queries as follows. Suppose the learner has already asked $i - 1$ equivalence queries (so the first $i - 1$ blocks of T has been fixed).

Answering membership queries: Let a be the instance with which the learner asked a membership query. The adversary replies “no”. In order to be consistent, the adversary looks at the i th block in a (let it be the string t) and eliminates t from $S^{(i)}$. This guarantees (together with the adversary’s reply to equivalence queries) that whenever the i th block of T is determined, U will reject a .

Answering equivalence queries: Suppose the adversary has asked the i th equivalence query with the hypothesis H . Observe that the size of $S^{(i)}$ is initially $k^s = n^c$. Since the number of membership queries is at most $n^c - 1$ and since every membership query eliminates at most one element from $S^{(i)}$, there must be at least one element in $S^{(i)}$. Let t be an element in $S^{(i)}$. Choose the i th block of T to be t . This choice is consistent with the membership queries. The adversary answers “no” to the learner, and outputs the prefix of i blocks from T as the

counterexample. However, we need to guarantee that this string (call it T') is really a counterexample. If H rejects T' , then we want T' to be accepted by U , so we place the index i in the set A . Otherwise, if H accepts T' , we want U to reject T' , so we do not put i in A . Observe that U is designed so that on input T' it ends in state $q_i = 1 + is$ and q_i is in the final states set F if and only if i is in A .

The above argument applies as long as $i \leq (n - 2)n/s$, which establishes the lower bound \square

CHAPTER 11

Open Problems

This work has established some sharp bounds for a variety of the most basic classes for which exact identification algorithms are known. It is perhaps surprising that we had such great success in proving matching lower and upper bounds, particularly since the lower bounds hold under the most favorable conditions for learning (arbitrary hypotheses and superpolynomial time), while the upper bounds holds under the most restrictive (hypotheses must be in the class to be learned, polynomial time is required).

We conclude this thesis by pointing out open problems.

Horn sentences and DFAs: There are two exceptions to the sharp bounds established in this thesis. The first is the case of Horn sentences. The lower bound established here is $\Omega\left(\frac{mn}{\log n + \log m}\right)$. The learning algorithm of Angluin, Frazier, and Pitt [7] learns the class of Horn sentences using $\Theta(mn)$ equivalence queries. So there is a gap of $(\log n + \log m)$ between the two. The other exception to the sharp bounds is the case of DFAs. The class of DFAs was proven to be learnable by Angluin [4] using $n - 1$ equivalence queries, where n is the number of states in the target DFA. The lower bound established here is $\Omega\left(\frac{n}{\log n}\right)$. So there is a gap of a $\log n$ factor. An open problem is to close these two gaps.

Read-Twice DNF: The class of Read-Twice DNF formulas was proven to be learnable by Aizenstein and Pitt [1]. The number of equivalence queries used by their algorithm is $O(n^2)$. Can one apply our techniques to their algorithm and prove a sharp bound?

k -DNF and k -CNF: Another interesting problem is how much the use of membership queries can reduce the number of equivalence queries needed to learn classes that can in fact be learned with equivalence queries alone, such as k -DNF and k -CNF. For k -DNF, Littlestone's algorithm uses $O(km \log n)$ equivalence queries where m is the number of terms [24]. A lower bound is $n \log n$. The gap is still large, even using membership queries (though if m is known, our generalized halving algorithm uses $O\left(\frac{km \log n}{d(\log km + \log \log n)}\right)$ queries).

APPENDIX A

k -term DNF formulas, continued

In this appendix we exhibit a version of **produce-terms** that can be used to learn k -term DNF formulas. The version is based on Angluin's algorithm [3].

A.1. Angluin's algorithm and our refinement

We begin with a few definitions. For an assignment a and a set of literals L , we define $a_{L \leftarrow 0}$ to be

$$a_{L \leftarrow 0}[\ell] = \begin{cases} 0 & \text{if } \ell \in L \\ a[\ell] & \text{otherwise.} \end{cases}$$

for every literal ℓ . For L that is a singleton we will use the shorthand $a_{\ell \leftarrow 0}$ to denote $a_{\{\ell\} \leftarrow 0}$, where $L = \{\ell\}$. For a positive example a we define the *sensitive set* of a , $sensitive(a)$, to be the set of literals ℓ for which $a_{\ell \leftarrow 0}$ is a negative example. Note that we need only $n+1$ membership queries to find the sensitive set of an assignment. For each i the literals ℓ_i and $\bar{\ell}_i$ are called *complementary*. A k -term DNF function f is called *reduced* if we cannot drop any term from f or any literal from any term in f and get a function equivalent to f . Let $k \geq 1$ and define the set

$$I_k = \{(i, j) \mid 1 \leq i \neq j \leq k\}.$$

A *discriminant* of a reduced k -term DNF function $f = t_1 \vee \cdots \vee t_k$ is an indexed collection of literals ℓ_{ij} for $(i, j) \in I_k$, such that

- (1) For every $(i, j) \in I_k$, ℓ_{ij} is a literal in t_i and not in t_j .
- (2) If t_i and t_j contain a complementary pair of literals, then ℓ_{ij} and ℓ_{ji} are a complementary pair of literals.
- (3) For each $i = 1, \dots, k$, the set $\{\ell_{ji} \mid j \neq i\}$ does not contain a complementary pair.

Let L_{*i} denote the set $\{\ell_{ji} \mid \text{for every } j \neq i\}$. Similarly, let L_{i*} denote the set $\{\ell_{ij} \mid \text{for every } j \neq i\}$.

A.1.1. An outline of Angluin's algorithm. We now give an outline of Angluin's algorithm [3] for learning k -term DNF formulas by k -term DNF formulas.

Let $f = t_1 \vee \dots \vee t_k$ be a reduced representation of the target formula. The original paper proves that f has a discriminant, so let ℓ_{ij} , $(i, j) \in I_k$ be a discriminant of f . Angluin's algorithm is based on the following lemma.

LEMMA A.1. *Let $f = t_1 \vee \dots \vee t_k$, $k \geq 1$, be a k -term DNF formula and let ℓ_{ij} for $(i, j) \in I_k$ be a discriminant for f . If a satisfies a term t_i in f then the literals of t_i are exactly those in the set $L_{i*} \cup \text{sensitive}(a_{L_{*i} \leftarrow 0})$.*

The following is an outline of Angluin's algorithm. Assume for now that a discriminant of the target formula f is known, and let ℓ_{ij} , $(i, j) \in I_k$ be this discriminant. Let h be identically false. Suppose we make an equivalence query with h and let a be the positive counterexample returned (unless f is identically false, in which case we are done). We know that a satisfies some term t_{i_0} in f , but we do not know which one. However, if for every i , we produce the term whose set of literals is exactly the set $L_{i*} \cup \text{sensitive}(a_{L_{*i} \leftarrow 0})$, then the above lemma A.1 guarantees that t_{i_0} is one of these k terms. We add these terms to h and execute the same process again, making an equivalence query. If the counterexample a' is positive then we add k terms to h one of which is in f and is satisfied by a' . If a' is a negative counterexample, then we know that there is some term in h that does not exist in f (in particular any term in h that is satisfied by a') so we drop it. Every positive counterexample adds a new

term to h that is in f , so there can be at most k positive counterexamples. Every positive counterexample adds at most k terms to h , so there can be at most k^2 terms in h . Since every negative counterexample drops at least one term from h , there can be at most $k^2 - k$ negative counterexamples. Summing up the number of negative counterexamples and the number of positive counterexamples, we see that the number of equivalence queries needed to learn f (assuming we know a discriminant for f) is at most k^2 . We need at most $k(n + 1)$ membership queries, because whenever we get a positive counterexample, we find the set of literals $L_{i*} \cup \text{sensitive}(a_{L_{*i} \leftarrow 0})$, which involves finding the sensitive set of $a_{L_{*i} \leftarrow 0}$ (as stated above, this requires $n + 1$ membership queries). Getting a negative counterexample, does not involve making membership queries. There are at most k positive counterexamples, so the total number of membership queries needed (assuming we know a discriminant) is at most $k(n + 1)$.

The problem now is to find a discriminant for f . Note that any suitable discriminant for f is a collection of $k(k - 1)$ literals. What Angluin's algorithm does, in order to find a discriminant, is to try all possible collections of literals of size $k(k - 1)$. Since there are $2n$ literals, the number of such collections is at most $\binom{2n}{k(k-1)} \leq (2n)^{k(k-1)} = O(n^{k^2})$. Then, the algorithm tries to learn f for every possible collection. We thus have the following lemma.

LEMMA A.2. *Angluin's algorithm [3] learns k -term DNF formulas in time $O(n^{k^2})$ using at most $n^{O(k^2)}$ membership queries and equivalence queries.*

A.1.2. Our refinement. Using lemma A.1 and the fact that for each i both L_{i*} and L_{*i} are of size less than k , we describe the following version of **produce-terms**.

Let \mathcal{L} be the set of all nonempty subsets of literals of size less than or equal to

$k - 1$. The size of \mathcal{L} is

$$\begin{aligned}
\binom{2n}{1} + \binom{2n}{2} + \cdots + \binom{2n}{k-1} &\leq \sum_{i=1}^{k-1} (2n)^i \\
&\leq \frac{(2n)^k - 1}{2n - 1} \\
&= O(n^k).
\end{aligned}$$

Note that \mathcal{L} contains both L_{i*} and L_{*i} for every $i = 1, \dots, k$.

Suppose now that **produce-terms** is given a positive example a of f , and suppose a satisfies t_i in f . **produce-terms** outputs, for each $L, L' \in \mathcal{L}$, the term whose literals are exactly $L' \cup \text{sensitive}(a_{L \leftarrow 0})$. In particular for $L' = L_{i*}$ and $L = L_{*i}$, thus **produce-terms** outputs t_i .

The number of membership queries that **produce-terms** uses and its running time are both $|\mathcal{L}|^2 n = O(n^{2k+1})$. The number of terms that **produce-terms** outputs is $|\mathcal{L}|^2 = O(n^{2k})$.

A.2. Summary

Using the version of **produce-terms** presented above with our techniques presented in 4.2, we obtain the results shown in table A.1. The complexities shown in the table are polynomial for k constant.

	Running Time	Membership Queries	Equivalence Queries
<code>learn1-k-term-dnf</code>	$n^{O(k)}$	$n^{O(k)}$	$n^{O(k)}$
<code>learn2-k-term-dnf</code>	$n^{O(k^2)}$	$n^{O(k^2)}$	k or $k + 1$
<code>learn3-k-term-dnf</code>	$n^{O(k)}$	$n^{O(k)}$	k or $k + 1$

TABLE A.1. *Summary of results when using the version of **produce-terms** based on Angluin's algorithm above.*

Bibliography

- [1] Howard Aizenstein and Leonard Pitt. Exact learning of read-twice DNF formulas. In *32nd Annual Symposium on Foundations of Computer Science*, pages 170–179, October 1991.
- [2] Howard Aizenstein and Leonard Pitt. Exact learning of read- k disjoint DNF and not-so-disjoint DNF. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 71–76, July 1992.
- [3] Dana Angluin. Learning k -term DNF formulas using queries and counterexamples. Technical Report YALEU/DCS/RR-559, Yale University, August 1987.
- [4] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, November 1987.
- [5] Dana Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.
- [6] Dana Angluin. Negative results for equivalence queries. *Machine Learning*, 5:121–150, 1990.
- [7] Dana Angluin, Michael Frazier, and Leonard Pitt. Learning conjunctions of horn clauses. *Machine Learning*, 9:147–164, 1992.
- [8] Dana Angluin, Lisa Hellerstein, and Marek Karpinski. Learning read-once formulas with queries. *J. ACM*, Volume 40, Number 1, 185–210, January, 1993.
- [9] Dana Angluin and Michael Kharitonov. When won't membership queries help? In *Proceedings of the 23rd annual ACM Symposium on Theory of Computing*, pages 444–454, ACM, May 1991.
- [10] José Luis Balcázar, Josep Díaz, and Joaquim Gabarró. *Structural complexity I*. Springer-Verlag, Germany, 1988.
- [11] Ian Martynovich Barzdin and Rūsiņš Freivalds. On the prediction of general recursive functions. *Soviet Mathematics Doklady*, 13:1224–1228, 1972.
- [12] Avrim Blum and Steven Rudich. Fast learning of k -term DNF formulas with queries. In *Proceedings of the Twenty Fourth Annual ACM Symposium on Theory of Computing*, pages 382–389, May 1992.
- [13] John A. Bondy and U. S. R. Murty. *Graph theory with applications*. Macmillan, London, 1977, c1976.

- [14] Nader H. Bshouty. Exact learning. In *34th Annual Symposium on Foundations of Computer Science*, October 1993.
- [15] Nader H. Bshouty and Richard Cleve. On the exact learning of formulas in parallel. In *Proceedings of the 33rd Symposium on Foundations of Computer Science*, pages 513–522, October 1992.
- [16] Nader H. Bshouty, Thomas R. Hancock, and Lisa Hellerstein. Learning arithmetic read-once formulas. In *Proceedings of the Twenty Fourth Annual ACM Symposium on Theory of Computing*, pages 370–381, May 1992.
- [17] Nader H. Bshouty, Thomas R. Hancock, and Lisa Hellerstein. Learning Boolean read-once formulas with arbitrary symmetric and constant fan-in gates. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 1–15, August 1992.
- [18] Nader H. Bshouty, Thomas R. Hancock, Lisa Hellerstein, and Marek Karpinski. Read-once threshold formulas, justifying assignments, and transformations. Technical report, International Computer Science Institute TR-92-020, 1991.
- [19] Gérard D. Cohen and Gilles Zémor. Intersecting codes and independent families. Draft.
- [20] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. The MIT press, 1990.
- [21] Thomas R. Hancock and Lisa Hellerstein. Learning read-once formulas over fields and extended bases. In *Proceedings of the 1991 Workshop on Computational Learning Theory*, pages 326–336, Morgan Kaufmann, 1991.
- [22] David Haussler, Michael Kearns, Nick Littlestone, and Manfred K. Warmuth. Equivalence of models for polynomial learnability. In *Proceedings of the 1988 Workshop on Computational Learning Theory*, pages 42–55, Morgan Kaufmann, August 1988.
- [23] Michael Kearns, Ming Li, Leonard Pitt, and Leslie Valiant. On the learnability of boolean formulae. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 285–295, ACM, 1987.
- [24] Nick Littlestone. Learning when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.
- [25] Wolfgang Maass and György Turán. Lower bound methods and separation results for on-line learning models. *Machine Learning*, 9:107–145, 1992.
- [26] Leonard Pitt and Leslie Valiant. Computational limitations on learning from examples. *J. ACM*, 35:965–984, 1988.
- [27] Vijay Raghavan and Stephen R. Schach. Learning Switch Configurations. In *Proceedings of the 1990 Workshop on Computational Learning Theory*, pages 38–51, Morgan Kaufmann, August 1990.
- [28] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 411–420, May 1989.

- [29] Leslie Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.