Improved Algorithms for Theory Revision with Queries

Judy Goldsmith*
Dept. of Computer Science
University of Kentucky
763 Anderson Hall
Lexington, KY 40506
goldsmit@cs.uky.edu

Robert H. Sloan†
Dept. of EE & Comp. Sci.
U. Illinois at Chicago
851 S. Morgan St. Rm 1120
Chicago, IL 60607-7053
sloan@eecs.uic.edu

Balázs Szörényi
Dept. of Computer Science
University of Szeged
Hungary
sirnew@edge.stud.u-szeged.hu

György Turán‡
Math, Stat., & CS Dept.
U. Illinois at Chicago,
Research Group on AI
Hungarian Acad. of Sciences
gyt@uic.edu

Abstract

We give a revision algorithm for monotone DNF formulas in the general revision model (additions and deletions of variables) that uses $O(m^3 e \log n)$ queries, where m is the number of terms, e the revision distance to the target formula, and n the number of variables. We also give an algorithm for revising 2-term unate DNF formulas in the same model, with a similar query bound. Lastly, we show that the earlier query bound on revising read-once formulas in the deletions-only model can be improved from $O(e \log^2 n)$ to $O(e \log n)$.

1 INTRODUCTION

A doctor has a theory about the patient and makes recommendations. They don't work. The doctor must change the theory. Rather than start from scratch again, she runs diagnostics designed to lead to incremental changes in the theory. If she was nearly correct, this should be more efficient than beginning all over again.

The goal of concept learning, and indeed of all learning from examples, is to obtain a representation of a concept or function on some domain so that one can use it to predict the function's value on new instances from the domain. However, in using this function on some performance task, one may well learn that it is not exactly correct (e.g., in medical diagnosis if the patient does not recover). Hence one wants to *revise* this function. Intuitively, if one already has a roughly correct function, then altering it to be exactly correct should require much less training data than learning the function from scratch. This paper and previous work [6, 14] show that this is indeed the case.

Note that what the computational learning theory community calls a *concept* is often referred to as a *theory* in logic, and either a theory or a knowledge base elsewhere in artificial intelligence. We will henceforth refer to the problem of revising a concept by its most common name in machine learning: *theory revision*.

We frame this problem in the model of learning with membership and equivalence queries. We believe that the query model with both equivalence and membership queries is especially well suited to the theory revision problem for two reasons. First, in practice theory revision would be used for deployed AI systems that make mistakes, and typically a human expert would be the one to say that a system had made a mistake. So there is a human expert who is providing something like counterexamples to equivalence queries, and this human expert should be able to answer membership queries as well. Second, as we will discuss in more detail, there is evidence that it will be very difficult or impossible to make progress on theory revision using only equivalence queries (or only PAC-type sampling).

In this paper, we present three new results. We show how to revise m-term monotone DNF, and 2-term unate DNF, in time $O(\log n \cdot \text{poly}(m) \cdot \text{poly}(e))$, where e is the minimum number of revisions needed, and n is the total number of variables, allowing essentially arbitrary revisions to the initial theory; as long as $m = o(n)$, this is faster than relearning the theory from scratch. Each of these results improves over a previous result for 2-term monotone DNF [6]. Additionally, we reduce the query complexity for revising read-once formulas from $O(e \log^2 n)$ to $O(e \log n)$. This is very close to optimal; the lower bound on the number of queries is $\Omega(e \log(n/e))$ [15].

We next explain a bit more about the model of theory revision used here, put our results into context, and then compare the results in this paper with previous results.

1.1 MODEL OF THEORY REVISION

The key metric for theory revision is the *syntactic distance* between the initial theory and the target theory. The syntactic distance between a given concept representation and another concept is the minimal number of elementary operations (such as the addition or the deletion of a literal or a clause) needed to transform the given concept representation to a representation of the other concept. Our goal in theory revision is to find algorithms whose query complexity is polynomial in the syntactic difference (or *revision distance*) between the initial theory and the target theory, but only polylogarithmic in the total number of possible variables. Thus, this work has some similarities to the work on

*Partially supported by NSF grant CCR-9610348; work done while visiting the Dept. of EECS at the University of Illinois at Chicago and the Dept. of Computer Science at Boston University.

†Partially supported by NSF grant CCR-9800070.

‡Partially supported by NSF grant CCR-9800070, and OTKA T-25721.

attribute-efficient learning [3, 4].

A particular measure of revision distance is determined by fixing a specific set of elementary operations, which we will call *revision operators*. Following the spirit of much work in machine learning on theory revision, we consider two sets of revision operators, the deletions-only revision operator and the general revision operator. These are defined in Section 2.

1.2 RELATED WORK

Mooney [10] formulated an approach to theory revision based on *syntactic distances*, and was the first to look at theory revision in the context of computational learning theory. He considered PAC-learnability and gave a positive result for *sample complexity* (which is equivalent to a positive result on query complexity). Computational efficiency was left as an open problem.

Sloan and Turán [14] specified the precise model of theory revision that we use here. In the deletions-only model, they gave revision algorithms for 2-term unate DNF using $O(e)$ queries, unate k-DNF using $O(ke \log n)$ queries, and read-once formulas using $O(e \log^2 n)$ queries, where e is the revision distance between the initial and the target theories, and n is the number of variables. In the general model, they gave revision algorithms for threshold and parity functions. Goldsmith and Sloan [6] gave algorithms for 2-term monotone DNF in the general model and for propositional Horn sentences in the deletions-only model.

More generally, there is a wide AI literature on theory revision (e.g., [8, 12, 16]). Many systems for theory revision, such as EITHER [11], have been implemented.

The problem of correcting errors is pervasive, and error-correcting algorithms appear in a variety of contexts. Among them are fault analysis of circuits in switching theory (see, e.g., Kohavi [7]), program debugging (e.g., [13]), and model-based diagnosis (see, e.g., [5, 9]). See Sloan and Turán [14] for a somewhat longer discussion of these connections.

Sloan and Turán [14] present a family of DNF formulae on n variables with $O(n)$ terms for which any revision algorithm requires $\Omega(n)$ queries. Thus, the problem of theory revision as we have posed it is interesting only for DNF formulas with substantially fewer terms than possible variables (and only for Horn sentences with substantially fewer clauses than possible variables). Note that in the general model, this does not mean that the initial theory must contain many fewer terms or clauses than variables, but that the universe of *possible* variables considered in revising the initial theory must be much larger than the number of terms or clauses, in order for revision to be more efficient than from-scratch learning.

1.3 DISCUSSION OF OUR RESULTS

In the deletions-only model, revising unate formulas is almost identical to revising monotone formulas, since the orientation of every variable that can appear in the target is known from the start. This is *not* the case in the general model. That is why the first result on DNF formulas was for 2-term *unate* DNF in the deletions-only model [14], but it was then extended to 2-term *monotone* DNF in the general model [6].

In the deletions-only model, given a positive counterexample to one's current hypothesis (or to the initial theory), it is relatively straightforward to use membership queries to decide which term to revise (i.e., to solve the *credit assignment problem*.) Consider revising a 2-term unate DNF in the deletions-only model. If, say, instance x is positive for the target but satisfies neither term t_1 nor t_2 of the current hypothesis $t_1 \vee t_2$, then one or both of t_1 or t_2 has at least one extra variable to be deleted. Since neither t_i needs to have any literals added to it, then the following will tell us which one to revise: First, ask a membership query of x modified by turning "off" any literals not in t_1. If the response is yes, then x must satisfy a term of the target that is syntactically a subset of t_1, and we can delete from t_1 any variables not "on" in x. Next, ask a membership query of x modified by turning "off" any literals not in t_2. Again, if the response is yes, then we can delete from t_2 any variables not "on" in x. This was one of the key ideas used by our earlier algorithm for revising 2-term unate DNF [14].

The situation becomes more complicated when revisions can include the addition of variables. In particular, in the unate case, we do not know *a priori* how to turn off variables in an instance if those variables do not occur in any term of the initial theory. The problem is that if a literal involving x_i needs to be added, and x_i did not occur in the original theory, we do not know whether x_i or $\bar{x}_i$ occurs in the target.

However, even for monotone DNF, there are additional complications in the general model. By taking a positive counterexample and turning off all the variables that are not in a given hypothesis term, we can tell whether to make deletions from that hypothesis term. However, if we know that additions are needed, it is difficult to tell which initial theory term needs the added variables. In our earlier algorithm for revising 2-term monotone DNF, we solved this problem by trying in parallel (or, equivalently, nondeterministically) to revise *both* terms from a given counterexample. Such a strategy is inherently exponential in the number of terms.

Thus, new ideas were needed in the general model of revisions to extend the algorithm for 2-term monotone DNF both to monotone DNF, and to 2-term unate DNF.

Notice that while the result on revising monotone DNF holds for any number of terms, it is most interesting for a number of terms that is $o(n)$, where n is the number of variables. Once the number of terms is $\Omega(n)$, then one cannot do substantially better than to throw away the initial theory and use Angluin's algorithm to learn the formula from scratch [1]. This is because Sloan and Turán [14] have exhibited a $\Theta(n)$-term monotone DNF that requires $\Omega(n)$ queries to revise a single error, and Angluin's algorithm requires only $O(mn)$ queries to learn an m-term monotone DNF.

Lastly, for read-once formulas in the deletions-only model, we lower the query complexity from $O(e \log^2 n)$ to $O(e \log n)$, where e is the revision distance between the initial read-once formula and the target and n is the number of variables. We conjecture that for the type of formulas we consider (monotone, unate, and "near-unate," such as Horn) if there is a revision algorithm at all, then there is a revision algorithm whose query complexity's dependence on the number of variables n is only multiplicative in $O(\log n)$, not polylog n. (Note, however, that there may *also* be a dependence on the number of terms or clauses in the formula being revised.)

2 NOTATION

We are using the standard model of membership and proper equivalence queries (with counterexamples), denoted by MQ and EQ [2]. In an equivalence query, the learning algorithm proposes a *hypothesis*, a concept h from the concept class, and the answer depends on whether $h = C$, where C is the target concept. If so, the answer is "correct", and the learning algorithm has succeeded in its goal of exact identification of the target concept. Otherwise, the answer is a *counterexample*: any instance x such that $C(x) \neq h(x)$. In a membership query, the learning algorithm gives an instance x, and the answer is either 1 or 0, depending on $C(x)$; that is, $\mathrm{MQ}(x) = C(x)$, where again C is the target concept. We assume throughout that the concepts TRUE and FALSE are allowed as equivalence queries.

We use standard notions from propositional logic, such as variable, term, disjunctive normal form (DNF), monotone, etc. A formula is *read-once* if no variable occurs in it more than once; a formula is *unate* if no variable ever occurs in it both negated and unnegated.

The symbol $\subset$ will always denote *strict* subset.

We will need to combine terms of the initial theory and various hypotheses of our revision algorithms with instances in various ways. We define the operation $t \cap x$ for a term t and an instance x to be a *term* that is the `and` of those literals in t that are satisfied by x.

On the other hand, we will need intersection-like operations that return *instances*. The meaning of the instance $x \cap t$ in the monotone case is to set to *off*, that is, 0, all bits of x that do not occur in the monotone term t. In the unate case, we no longer know which orientation of a variable not occurring in a term is off.

Thus we define two different operations: $x \underline{\cap} t$ and $x \overline{\cap} t$ for "intersecting" instance $x \in \{0,1\}^n$ with term t to get back a new instance. These intersections are always with respect to an initial theory φ.

"Intersect down" is defined by

$$(x \underline{\cap} t)[i] = \begin{cases} x[i] & \text{if one of } v_i, \bar{v}_i \in t \\ \overline{(\varphi \setminus t)[i]} & \text{if } v_i \text{ or } \overline{v}_i \in \varphi \setminus t \\ \bar{x}[i] & \text{otherwise,} \end{cases}$$

and "intersect up" is defined by

$$(x \overline{\cap} t)[i] = \begin{cases} x[i] & \text{if one of } v_i, \bar{v}_i \in t \\ \overline{(\varphi \setminus t)[i]} & \text{if } v_i \text{ or } \overline{v}_i \in \varphi \setminus t \\ x[i] & \text{otherwise.} \end{cases}$$

For two vectors $x, y \in \{0,1\}^n$, we will use $x \otimes y$ to denote the set of indices or variables on which x and y disagree; thus $|x \otimes y|$ is the number of variables on which x and y disagree. We overload this operator to also indicate the symmetric difference of two terms, namely the set of literals that appears in exactly one of the two terms.

If $\varphi = T_1 \vee T_2$, then $T_{\bar{\imath}}$ denotes the term other than T_i.

The *revision distance* between a formula φ and some target concept C is defined to be the minimum number of applications of a specified set of revision operations to φ needed to obtain a formula for C. In the *deletions-only* model, our specified set of revision operators is fixing an occurrence of a variable to the constant 0 or 1. This corresponds to allowing deletions of variables and terms in DNF theory revision.

In the *general* revision operator, we are also allowed to add variables to a DNF term, with the following limitations in the unate case. First, a literal that appears in the initial formula cannot appear negated in the target formula. Second, the target formula must also be unate: in other words, variables not used at all in the initial formula cannot be added into one term negated and into another term unnegated. Furthermore, in our construction, all intermediate hypotheses must also be unate.

Note that this model allows us to entirely replace one term of the initial theory by a new term with entirely distinct variables. The revision distance for this replacement is the sum of the length of the deleted term (all of whose variables must be fixed to true) and the length of the added term. On the other hand, the revision distance for simply deleting a term with no replacement is only 1, since this can be done merely by fixing any one variable of the term to be false.

3 REVISING MONOTONE DNF

In this section, we present an algorithm to revise a monotone m-term DNF formula in the general revision model. This extends the algorithm for revising 2-term DNF formulas in [6].

3.1 DESCRIPTION OF ALGORITHM

A monotone DNF formula can be viewed as a collection of subsets of the set of variables, with each term defining a subset. We say that one term *covers* another if it is a superset of the other. When convenient, we sometimes treat monotone terms as elements of $\{0,1\}^n$, where the bit vector has a 1 exactly in those positions where the term contains a variable. If a term t covers a term of the target formula, then $\mathrm{MQ}(t) = 1$ and any counterexample x to $\mathrm{EQ}(t)$ must satisfy the target and not t; namely, x must be positive.

If $Y \subseteq Z$ and $\mathrm{MQ}(Y) = 0$ and $\mathrm{MQ}(Z) = 1$, then Z covers a target term not covered by Y. We can use binary search to find a subset of Z containing Y that covers a target term. In fact, we can find a set A of variables and a variable $l \in Z \setminus (Y \cup A)$ such that $\mathrm{MQ}(Y \cup A) = 0$ but $\mathrm{MQ}(Y \cup A \cup \{l\}) = 1$.

We previously [6] used a sort of binary search for this purpose. In the present setting, however, one new complication arises. We would like to consider l a necessary addition to Y. However, if Z covers several target terms, it may be necessary to add l to Y to cover one of those terms but not another. This could lead to our building up Y to cover more than one target term in an inefficient manner. We call such an l a *pivot*, because the choice of which term to cover pivots on whether l is added to Y. We recognize a pivot because without it, Z still covers a target term, so $\mathrm{MQ}(Z \setminus \{l\}) = 1$. If a pivot is found in the course of the binary search, we throw it out and restart the search from Y to $Z \setminus \{l\}$.

With that in mind, we can now describe the algorithm.

The heart of the construction is the procedure REVISE-UPTOE. It takes as parameters an m-term monotone DNF formula φ and e, the assumed revision distance from φ to the

Algorithm 1 REVISEUPTOE(φ, e). Revises φ, a set of monotone terms, if possible using at most e revisions; otherwise returns "Failure." Note that if any subroutine either finds the correct hypothesis or returns "Failure", then this algorithm also terminates. Also, if the error limit e is ever exceeded, this algorithm terminates immediately and returns "Failure".

1: $h = \emptyset$ //the initial hypothesis
2: **while** EQ(h) gives a counterexample x and $e > 0$ **do**
3:   **if** MQ$(x \cap t) = 1$ for some $t \in h$ **then** //delete variables from hypothesis terms
4:     **for all** $t \in h$ for which MQ$(x \cap t) = 1$ **do**
5:       $t = t \cap x$
6:       $e = e - |t - (t \cap x)|$
7:     **end for**
8:   **else** //find a new term to add to the hypothesis
9:     $terms = \varphi$ //the set of terms to consider
10:     $min = e$
11:     $FoundATerm = false$
12:     **for all** $t \in terms$ **do**
13:       $new = t \cap x$
14:       $numAddedLits = 0$
15:       **while** MQ$(new) == 0$ **and** $numAddedLits < e$ **do**
16:         $d =$ BINARYSEARCH(new, x)
17:         $new = new \cup \{d\}$
18:         $numAddedLits = numAddedLits + 1$
19:         **if** MQ$(x - \{d\}) == 1$ **then**
20:           //$x - \{d\}$ is a positive counterexample that covers fewer goal terms
21:           $x = x - \{d\}$
22:           **restart for all** t loop with this x by backing up to line 9 to reset other parameters
23:         **end if**
24:       **end while**
25:       **if** MQ$(new) == 1$ **then**
26:         $x = new$
27:         $FoundATerm = true$
28:         $min = \min(numAddedLits, min)$
29:       **end if**
30:     **end for**
31:     **if not** *FoundATerm* **then**
32:       **return** Failure
33:     **else**
34:       $h = h \cup \{x\}$
35:       $e = e - min$ //minimum number of edits done on any $t \in \varphi$ which contributed to x
36:     **end if**
37:   **end if**
38: **end while**

target. If e is in fact too small, REVISEUPTOE(φ, e) fails, and e is doubled. The claim, discussed in the next subsection, is that whenever the revision distance is $\leq e$, REVISEUPTOE(φ, e) succeeds, and uses only a bounded number of each type of query.

Given φ, REVISEUPTOE(φ, e) constructs a hypothesis monotone DNF formula h so that, at each stage of the construction, each term of h covers a term of the target formula. At each stage of the construction, we get a positive counterexample, x to h. (The initial h is $\emptyset$, which is interpreted as the everywhere-false formula.)

If x covers a target term already covered by a term of h, then x is used to delete variables from any hypothesis terms that cover a term covered by x. Since x is a positive counterexample, for any $t \in h$, it must be that $x \cap t \subset t$, so this yields at least one deletion.

Otherwise, x is used to add a new term to the hypothesis. For each initial term t, if binary search finds an unambiguous extension of $t \cap x$ in x that covers a target term (has positive membership query) with no more than e additions, then we consider that a candidate new term. It is then treated as the positive counterexample for each subsequent initial term. In this way, if several initial target terms could be revised by x, we get a new term that is a "close" revision (no more than e additions) of each of them. In particular, if the revision distance is $\leq e$ and the most efficient revision to that target term is t_i, then the new term is a revision of t_i with no unnecessary additions or deletions.

3.2 MONOTONE DNF CORRECTNESS AND QUERY COMPLEXITY

For all of the lemmas below, we assume that φ is an m-term monotone DNF formula, and the target is an m' monotone DNF formula ($m' \leq m$) with revision distance at most e from φ.

Lemma 1 *Algorithm* REVISEUPTOE *(Algorithm 1) maintains the invariant that each term of its hypothesis covers some term of the target. Therefore, any counterexamples must be positive counterexamples.*

Proof sketch. The initial counterexample is positive. Each positive counterexample, x, covers at least one target term T^*. If T^* is already covered by a term $t \in h$, then the "If" in Line 3 of REVISEUPTOE must be true, and t is replaced by $t \cap x$, which still covers T^*. (Note that x cannot cover any $t \in h$, since it is a counterexample to h. This forces at least one deletion if MQ$(t \cap x) = 1$.) If x does not cover any term covered by h, then, unless REVISEUPTOE returns Failure, Lines 9–37 add a new term to h that covers some target term covered by the counterexample x. $\square$

Lemma 2 *Each counterexample that covers a target term also covered by at least one term in h is used to delete variables from any terms $t \in h$ such that $t \cap x$ still covers a term. Because each term in h must be queried, each deletion requires $O(m)$ queries.*

Lemma 3 *If x is used to add a new term to h, the new term does not cover any target term already covered by h. Thus, no two terms in h cover the same target term.*

Proof sketch. Suppose t' is added to h because of counterexample x. In order for the algorithm to reach Line 8, it must be the case that for any t already in h, MQ$(x \cap t) = 0$. Note that $t' \subseteq x$, by construction. Therefore, $t \cap t' \subseteq t \cap x$, so (by monotonicity) MQ$(t \cap t') = 0$. □

Thus, there are at most m terms in h at any time.

Note that each x adds at most one term to h; once a set of additions is found for some $t \in \varphi$, that new positive example replaces x, and the process is repeated for each t remaining in φ. Thus, the term produced is as near as possible to one of the original terms.

Lemma 4 *If counterexample x covers more than one target term, say T_1^* and T_2^* (and perhaps others), and is used to add a new term to h, then both T_1^* and T_2^* will be covered in the most efficient manner from terms in φ.*

Proof. Note that no term in h covers either T_i^*, since x was not used for deletions.

Suppose that, for both i, there are variables $v_i \in T_i^* \setminus (x \cap t)$. Then, the binary search from $(x \cap t)$ to x will eventually find one of the v_is. (We call these variables "pivots.") At that point, the code backs up to Line 9 with x replaced by the less ambiguous $x - v_i$. Once the last pivot is found, any additions to any $(x \cap t)$ must be variables that appear in the unique target term still covered, or in the intersection of all remaining covered target terms.

Suppose, however, that we add a term, new, to h that covers both T_1^* and T_2^*. We know that for any term $t \in \varphi$, if $t \cap x$ was edited to new, then this involved at most e additions, and all those additions were in $T_1^* \cap T_2^*$, since no pivots were found. Therefore, if t is the appropriate term to revise to T_i^* and e is the correct error, new is a necessary revision of t. Furthermore, if t' is the appropriate term to revise to $T_{\bar{i}}^*$, new is also a necessary revision of t'. Eventually, new will have one T_i^* deleted, and the appropriate t or t' will contribute another term to h—one that does not cover $T_{\bar{i}}^*$. □

Lemma 5 *A single addition of a term to the hypothesis requires $O(m^2 e \log n)$ membership queries.*

Proof sketch. Note that there can be at most e additions to any $x \cap t$ for any $t \in \varphi$; if more additions are needed, the attempt to edit that term fails. Each search for an addition requires $\log n$ membership queries. However, even if the counterexample x covers a unique target term, the algorithm may try each of the m terms of φ to find one that works. This means $O(me \log n)$ membership queries.

If x is ambiguous, then every time a pivot is found, the entire additions procedure is restarted. Since x can cover at most m target terms, this can happen $m - 1$ times, so the entire search for one unambiguous addition may require $O(m^2 e \log n)$ membership queries. □

Theorem 6 REVISEUPTOE(φ, e) *uses at most $O(m^3 e \log n)$ membership queries and $O(e + m)$ equivalence queries, and succeeds if φ has revision distance less than or equal to e. Therefore, an m-term monotone DNF formula with an edit distance e from the target formula can be revised using $O(m^3 e \log n)$ queries.*

Proof sketch. Note that it is possible to get an initial theory that is, in fact, correct. Because of this possibility, we begin by asking MQ(φ). If the answer is not "Correct!" we apply REVISEUPTOE(φ, e) for repeatedly doubled values of e until it produces a "Correct!"

We give the full analysis of REVISEUPTOE; the rest follows. Note that the addition of a term to the hypothesis may involve simply copying that term from the initial theory, or may also involve deleting some variables not in the counterexample that triggers the addition, or perhaps some additions of variables in x that were not in the initial theory term. The construction of the revised theory requires m' additions of terms to the hypothesis plus up to e additions of variables to initial terms.

Each addition of a term to the hypothesis requires $O(m^2 e \log n)$ queries, and there are $m' = O(m)$ terms, so this requires $O(m^3 e \log n)$ membership queries needed for additions.

There are $O(e)$ deletions needed, and each deletion requires $O(m)$ membership queries, for a total of $O(em)$, which is $O(m^3 e \log n)$ queries.

Finally, each addition of a term to the hypothesis and each revision may require an equivalence query, for a total of $O(e + m)$ queries. □

4 REVISING UNATE DNF

In this section, we present an algorithm that can revise a 2-term unate DNF in the general model of revisions. The only restriction we make is that we assume that no variable in the initial theory has the wrong orientation. That is, if x_i occurs in the initial theory, then x_i could be deleted, or moved to the other term if it occurs only in one term, but we cannot delete x_i and add $\bar{x}_i$.

4.1 DESCRIPTION OF ALGORITHM

Throughout this section, we will refer to a term t of a hypothesis DNF as *full with respect to target term T^** if t's variables are a superset of T^*'s variables. Generally it will be clear which target term we are referring to, so we will simply refer to t as *full*. Intuitively, if t is full, then any necessary additions have been found, and t requires only deletion edits.

We refer to the variables that do not occur in the initial theory as the *outside* variables, and those that do occur in the initial theory as the *inside* variables.

We begin by explaining how we must alter two subroutines, BINARYSEARCH used earlier in this paper, and REVISEDOWN, used in our earlier work, in order to make them work in the unate case.

4.1.1 Binary search

The BINARYSEARCH referred to in this section is different from that used previously, in that it does not deal with sets of variables, but with *settings* of variables. When we look at $T \setminus S$, we are really considering the variables corresponding to elements of $T \otimes S$. When we divide $T \otimes S$ into two roughly-equal size sets, what we do is "flip the bits" (change the signs) of the variables in one of those sets. Other than this minor definitional change, and a small change involving pivots, which we discuss next, BINARYSEARCH works the same. The code for binary search for the unate DNF case is broken out in the figure entitled Algorithm 2.

Algorithm 2 BINARYSEARCH(Y, Z, e). For unate revisions, finds necessary additions to Y from Z to cover a target term, if this can be done with $\leq e$ additions. We require that initially MQ$(Z) = 1$.

```
while MQ(Y) == 0 and e > 0 do
    S = Y, T = Z.
    while |T ⊗ S| > 1 do   //binary search for 1 addition
        Divide positions where S and T disagree into approximately equal-size sets d_1 and d_2.
        Put Mid = S with positions in d_1 replaced by T's values
        if MQ(Mid) == 0 then
            S = Mid
        else
            T = Mid
        end if
    end while
    Let l be the 1 position in T ⊗ S
    if MQ(Z with position l flipped) = 1 then
        throw PivotException(Y, l, e, Z)
    end if
    Y = Y ∪ the value of Z for position l
    e = e − 1
end while
if MQ(Y) == 1 then                    //Y covers term
    return Y, e
else                    //e ≤ 0, so all edits already used
    return "Failure"
end if
```

We found the overall algorithm for the unate case easiest to describe by thinking of BINARYSEARCH stopping execution and throwing an exception back to the main algorithm whenever it finds a pivot. The action of the main algorithm is much the same as for the monotone case—it backs up and restarts with a counterexample that is altered by turning the pivot variable to the off position.

4.1.2 Revise Down

The procedure REVISEDOWN was used introduced in our earlier work [14], where we called it REFINEDOWN. It is used to delete unnecessary variables from hypothesis terms. Given a hypothesis where each term covers a target term, we will get only positive counterexamples. Suppose we are given a hypothesis $T_j \vee T_k$ such that T_j and T_k cover distinct target terms, and positive counterexample x, which necessarily covers neither T_j nor T_k. Then for one T_r, $x \cap T_r$ covers a target term and can therefore replace T_r. (If x covers both target terms, it is used to delete variables from both hypothesis terms.) This process is repeated until EQ(h) = "Correct" or the number of deletions performed exceeds the error bound, or EQ(h) returns a negative counterexample.

For unate formulas with general revisions, however, the situation is not so straightforward. Either or both initial theory terms might require additions as well as deletions. However, the main algorithm that calls REVISEDOWN is designed so that for any two-term hypothesis passed to REVISEDOWN, at least one of the two terms is full.

So REVISEDOWN may have a two-term hypothesis for a two-term target unate DNF, but still receive a negative counterexample x to an equivalence query, because the hypothesis term that x satisfies is not full. Notice, however, that there is no ambiguity about which hypothesis term to revise when REVISEDOWN gets a negative counterexample, because the negative counterexample can satisfy only the one term of the hypothesis that is not full. We will discuss what should be done with this negative counterexample in Section 4.1.3.

Algorithm 3 REVISEDOWN(T_1, T_2, e, p_1, p_2) Edits the two-term hypothesis $T_1 \vee T_2$, but never makes more than e edits. The p_i are positive instances (or NULL) associated with the respective T_i. Terminates either in failure, or with correct target formula. Terminates in failure if call to BINARYSEARCH fails.

```
Put h = T_1 ∨ T_2.
while (x = EQ(h)) ≠ "Correct" do
    if e ≤ 0 then                    //Used up all allowable edits
        return Failure
    end if
    if h(x) = 0 then           //x is positive counterexample
        for all terms T_i do
            x' = x ∩̄ T_i
            Turn "off" in x' any outside variables in T_ī \ T_i
            if MQ(x') == 1 then
                T_i = T_i ∩ x
                Decrement e by # deletions to T_i
            end if
        end for
        if no term of h was revised by x then
            return "Failure"
        end if
    else                    //x is a negative counterexample
        if T_1(x) == T_2(x) == 1 then
            return Failure
        else
            Let T_i be unique term of h such that T_i(x) = 1
            if p_i == NULL then
                return Failure
            end if
            z = vector with inside variables of x and outside variables of p_i.
            if MQ(z) ≠ 1 then
                return Failure
            end if
            Perform binary search from x to z
            Add all variables found to T_i and decrement e accordingly
        end if
    end if
end while
```

The preceding discussion actually applies only to certain calls of REVISEDOWN. As we will discuss soon in Section 4.1.4, the main algorithm tries calling REVISEDOWN with several different parameters, intending to abandon all but one of the calls. For the calls that will be abandoned (intuitively, the ones where the algorithm has made the wrong "guesses" about the parameters), the two-term hypothesis given to REVISEDOWN could have neither term full. In this case, however, it is fine for REVISEDOWN to fail. In fact, when REVISEDOWN receives a negative counterexample, it checks, in Line 19, to see whether that negative counterexample satisfies both hypothesis terms. If so, then it must be that neither hypothesis term is full, so REVISEDOWN terminates with failure.

4.1.3 Negative counterexamples

We will sometimes have hypothesis terms that are not full. This situation can arise both with a one-term hypothesis in REVISEUPTOE, and as just discussed, for one of the two terms of REVISEDOWN's two-term hypothesis. In both cases, the algorithm is designed so that the hypothesis term always contains all the inside variables of the associated target term.

We keep associated with each hypothesis term T_i a positive instance p_i that is supposed to satisfy the target term associated with T_i^*. The special case of p_i being NULL indicates that T_i is supposed to be full, and any negative counterexample to T_i must indicate that an incorrect nondeterministic choice has been made.

When we receive a negative counterexample y that satisfies hypothesis term T_i, it must be that y has one or more outside variables of the associated target term set to off, since T_i has all its inside variables. Meanwhile, p_i, by definition, has all those outside variables set to on. So, if we do a binary search from y to a vector with the same inside variables as y and the same outside variables as p_i, we will discover some number of necessary additions to T_i, at a cost of $O(\log n)$ queries per addition for BINARYSEARCH. (We could equally well search from a vector with the same inside variables as p_i and the same outside variables as y to y.)

4.1.4 Main Algorithm: REVISEUPTOE

As before, we test whether the initial formula $\varphi = t_1 \vee t_2$ can be revised to the target formula, for $e = 1, 2, 4, \ldots$. The procedure, REVISEUPTOE, begins with an equivalence query to $\emptyset$. If that is not the target formula, then we get a positive counterexample, x, which is used to create a one-term hypothesis.

Assume for now that both MQ$(x\overline{\cap}(t_1 \cap t_2)) = 0$ and MQ$(x\underline{\cap}(t_1 \cap t_2)) = 0$. We will describe how to handle the case where that is not true a bit later.

Intuitively, we nondeterministically try both the assumptions that x satisfies a target term that should be derived by editing initial theory term t_1 (i.e., the revision distance is minimized by editing t_1 rather than t_2 to get this target term) and that x satisfies a target term that should be derived by editing initial theory term t_2. In practice, the "try both" construct tries first one and then the other alternative.

Here is how we proceed when we are assuming that t_1 should be edited to create a hypothesis term T such that $T(x) = 1$. We ask the two membership queries MQ$(x\overline{\cap}t_1)$ and MQ$(x\underline{\cap}t_1)$. If both return 1, then our initial one-term hypothesis is $t_1 \cap x$. Intuitively, we are hoping that the responses to the membership queries indicated that x satisfies a target term that is contained in t_1 though, as we discuss in the proof of Lemma 8, this is not necessarily the case. We remember for later that t_1 is the term that we edited, and that

Algorithm 4 REVISEUPTOE$(\varphi_0 (= t_1 \vee t_2), e)$
Note that a branch of a "try both" fails if one of the subroutines it calls fails *without* being explicitly tested for failure.

Let x be positive instance (from EQ(FALSE))
try both $b = 1, 2$:
Work with x as described in text to create a one-term hypothesis h assuming that x satisfies target term that can be derived from t_b to minimize total edits
Let t_i be term of φ_0 that h is derived from
Let p_i be positive instance associated with h
while $(y = \text{EQ}(h)) \neq$ "Correct" **and** $e > 0$ **do**
  if $h(y) = 1$ **then** //Negative counterexample
    if $p_i ==$ NULL **then**
      return "Failure"
    else //h needs more variables
      $z =$ vector with inside variables of y and outside variables of p_i.
      if MQ$(z) \neq 1$ **then**
        return Failure
      end if
      Perform binary search from x to z
      Add all variables found to h; decrement e accordingly
    end if
  else //y is a positive counter example
    $h_0 = h$; $e_0 = e$
    $y' = y\underline{\cap}t_{\bar{\imath}}$
    if $t_{\bar{\imath}} \cap y \subset h$ **and then** $MQ(y') == 1$ **then**
      $h = h \cap y'$
    else if BINARYSEARCH(y', y, e) returns (z, e) (rather than "Failure") **then**
      $h = h_0 \vee ((t_{\bar{\imath}} \cap y)$ plus literals of z in $z \otimes y')$
      //REVISEDOWN may find target
      if REVISEDOWN(h, e) returns "Failure" **then**
        $h = h_0 \cap y$
        $e = e_0 - |h_0 \setminus h|$
      end if
    else //BINARYSEARCH(y', y, e) returns "Failure"
      $h = h_0 \cap y$
      $e = e_0 - |h_0 \setminus h|$
    end if
  end if
end while
end try both

x is an instance that we are assuming satisfies the associated term of the target formula. The fact that we remember x, instead of NULL, indicates that we could later legitimately receive a negative counterexample satisfying this term; that is, that this term might not be full.

If, instead, $\mathrm{MQ}(x \underline{\sqcap} t_1) = 0$, then we perform a binary search from $x \underline{\sqcap} t_1$ to x, and our initial one-term hypothesis is $t_1 \cap x$ plus whatever additional variables were found by the binary search. In this case, we should never see a negative counterexample to this term, so we make the associated positive instance NULL. (If we do receive a negative counterexample, it indicates that we are in the wrong branch of the "try both.") Again, t_i is the edited initial theory term.

The final possibility is that $\mathrm{MQ}(x \overline{\sqcap} t_1) = 0$, but $\mathrm{MQ}(x \underline{\sqcap} t_1) = 1$. If $T_1^*(x) = 1$, then it must be that T_1^* contains some variables from $t_2 \setminus t_1$, since $\mathrm{MQ}(x \overline{\sqcap} t_1) = 0$. Thus, we can be certain that $x' = x \underline{\sqcap} t_1$ satisfies only the *other* target term. So, if $\mathrm{MQ}(x' \underline{\sqcap} t_2) = 1$, then we use $(t_2 \cap x')$ as our initial one-term hypothesis. If not, then we do a binary search from $(x' \underline{\sqcap} t_2)$ to x' to find which variables we need to add to $t_2 \cap x'$ to create our initial hypothesis. Either way, we indicate that our hypothesis term has actually been derived from t_2, and that it is full, so we should never receive any negative counterexamples to it.

Now let us explain what we do if one or both of $\mathrm{MQ}(x \overline{\sqcap} (t_1 \cap t_2)) = 1$ or $\mathrm{MQ}(x \underline{\sqcap} (t_1 \cap t_2)) = 1$. In this case, $t = t_1 \cap t_2$ plays a very similar role to t_i above in the case where we assumed that both these membership queries returned 0. If both membership queries return 1, then we initialize our one-term hypothesis to be $t \cap x$, and $x \overline{\sqcap} t$ is the associated positive instance, and we must "try both" associations of the derivation of $t \cap x$ with t_1 and t_2. (I.e., when a second term is added, it should be derived from the *other* initial theory term).

If exactly one of $\mathrm{MQ}(x \overline{\sqcap} t)$ and $\mathrm{MQ}(x \underline{\sqcap} t)$ is 1, then we can do a binary search between $x \underline{\sqcap} t$ and $x \overline{\sqcap} t$ or vice versa and derive a hypothesis term that is full. In this case the associated positive instance is NULL, but we still have to "try both" possibilities (t_1 and t_2) for the derivation of the new term.

Unlike the monotone case, once we have a one-term hypothesis, we are not always sure whether subsequent positive examples should be used to add variables to an initial theory clause in order to generate a new hypothesis clause, or to delete variables from an existing hypothesis clause. Our algorithm is designed so that an incorrect guess will only propagate down twice: if we make two false assumptions, the algorithm will backtrack. The places where assumptions are made are in with the initial counterexample, which may be used to edit one or the other initial term, and then, given a one-term hypothesis, whether to use a positive counterexample to edit the existing term or to create a new term.

4.2 CORRECTNESS AND QUERY COMPLEXITY

We first make an observation about REVISEDOWN that follows immediately from an examination of its code.

Lemma 7 *When* REVISEDOWN *is called with its maximum number of edits parameter e set to d, then it makes $O(d \log n)$ queries.*

The following lemma is the heart of the correctness argument.

Lemma 8 *Let $\varphi_0 = t_1 \vee t_2$ be an initial theory, and let $\Phi^* = T_1^* \vee T_2^*$ be a target theory, with the T_i^* labeled so that $e = |t_1 \otimes T_1^*| + |t_2 \otimes T_2^*| \leq |t_1 \otimes T_2^*| + |t_2 \otimes T_1^*|$. Consider a run of* REVISEUPTOE(φ_0, e). *If the positive instance x used in Line 1 satisfies only the one target term T_j^*, and $\mathrm{MQ}(x \overline{\sqcap} (t_1 \cap t_2)) = \mathrm{MQ}(x \underline{\sqcap} (t_1 \cap t_2)) = 0$, then the branch of the "Try both" where $i = j$ finds the target theory using at most $O(e^2 \log n)$ queries.*

Proof. First, we point out that the pivot exception in binary search will not occur because x covers only one of the two target terms.

We proceed by cases.

Case I: Both $\mathrm{MQ}(x \underline{\sqcap} t_i) = 1$ and $\mathrm{MQ}(x \overline{\sqcap} t_i) = 1$.

The initial one-term hypothesis created in Line 2 of REVISEUPTOE is $t_i \cap x$. Call this term of the hypothesis T. Notice that T cannot cover $T_{\bar{\imath}}^*$. This is true if $T_{\bar{\imath}}^*$ contains any variables not in t_i. Even if all $T_{\bar{\imath}}^*$'s variables are in t_i, however, T still cannot cover $T_{\bar{\imath}}^*$, since the hypothesis of the lemma is that x does not satisfy $T_{\bar{\imath}}^*$, and $T = t_i \cap x$.

i. $T_i^* \subseteq t_i$.

Since $T_i^*(x) = 1$, it must be that T covers T_i^*. Thus, any counterexample to $\mathrm{EQ}(h)$ where h includes T must be positive, or it contradicts that $T_i^* \subseteq T$. Note that so far we have use only a constant number of queries.

For each subsequent positive counterexample y, we first assume that y does not satisfy T_i^*, so y should be used to create a second hypothesis term derived from $t_{\bar{\imath}}$. Notice that $y' = y \underline{\sqcap} t_{\bar{\imath}}$ also cannot satisfy T_i^*, since T_i^* has no outside variables. So, if $\mathrm{MQ}(y') = 1$, then we can initialize the second hypothesis term to $t_{\bar{\imath}} \cap y$, and if not we can do one binary search from y' to y to decide which additional outside variables should be added to $t_{\bar{\imath}} \cap y$. Thus initializing the second term requires $O(e \log n)$ queries.

One special case can arise. If $\mathrm{MQ}(y') = 1$ and $t_{\bar{\imath}} \cap y \subseteq T$ (and, in fact, since y is a counterexample, it must be that $t_{\bar{\imath}} \cap y \subset T$), then intuitively we certainly do not want to add $t_{\bar{\imath}} \cap y$ as a second hypothesis term, because then the first term would be redundant. Formally, we can argue as follows. If both y and $y' = y \underline{\sqcap} t_{\bar{\imath}}$ satisfy $T_{\bar{\imath}}^*$, then $T_{\bar{\imath}}^*$ has all its variables in $t_{\bar{\imath}}$, and indeed in $t_{\bar{\imath}} \cap y$. However, since $t_{\bar{\imath}} \cap y \subset T$, that would mean that T covers $T_{\bar{\imath}}^*$, which is false. So, at least one of y or y' satisfies T_i^*. In this case (checked for in Lines 18–21 of REVISEUPTOE), we can safely use the inside variables of y (which are the same as the inside variables of y') to make deletions from T.

When instead we are trying to use y to start a second hypothesis term, we make one binary search, using $O(e \log n)$ membership queries, to initialize a second term of our hypothesis. After that, we are in the subroutine REVISEDOWN, which performs only deletions to our two-term hypothesis, using only a constant number of queries per deletion. Thus, if y actually satisfies $T_{\bar{\imath}}^*$, we obtain the target using at most $O(e)$ equivalence and $O(e \log n)$ membership queries. If y does *not* satisfy $T_{\bar{\imath}}^*$, we backtrack after $O(e \log n)$ queries, and use y to perform at least one needed deletion from term T. Thus the total number of queries is at most $O(e)$ equivalence and $O(e \log n)$ membership queries per deletion.

ii. $T_i^* \not\subseteq t_i$.

Notice that in this case T is not full. If we receive a negative counterexample to $\mathrm{EQ}(T)$, then we can use it to make T full, at a cost of $O(e \log n)$ queries. After this, we are in the same situation as Case I.i.

The other possibility is that we receive a positive counterexample. We now digress a bit to describe some properties that our one-term hypothesis must have, and then return to describing how the positive counterexample is handled.

Let x be the positive instance that was used to create T. We claim that the following must hold:

1. T includes all inside variables of T_i^*, but no outside variables.

2. T_i^* contains at least one outside variable, and the sets of outside variables of T_1^* and T_2^* are disjoint.

Recall that $\mathrm{MQ}(x \overline{\sqcap} t_i) = \mathrm{MQ}(x \underline{\sqcap} t_i) = 1$, but by assumption, t_i does not cover T_i^*. Now also by assumption, $T_i^*(x) = 1$, so it must be that $x \overline{\sqcap} t_i$ satisfies T_i^*. Since T_i^* contains variables not in t_i and x and $x \underline{\sqcap} t_i$ differ on those variables, $x \underline{\sqcap} t_i$ must satisfy $T_{\bar{\imath}}^*$. Since $x \overline{\sqcap} t_i$ and $x \underline{\sqcap} t_i$ satisfy two different target terms, and both those instances have all variables in $t_{\bar{\imath}} \setminus t_i$ set to off, neither target term can have any variables from $t_{\bar{\imath}} \setminus t_i$. Now Item 1 follows because T was initialized to $t_i \cap x$, and all T_i^*'s inside variables are from t_i. Next, T_i^* must contain outside variables, because otherwise t_i would cover T_i^*. Target term $T_{\bar{\imath}}^*$ cannot contain any of T_i^*'s outside variables, because those variables are off in $x \underline{\sqcap} t_i$, which satisfies $T_{\bar{\imath}}^*$. This concludes the argument that Item 2 holds.

Now consider a positive counterexample y to $\mathrm{EQ}(T)$. there are three possibilities. One is that we execute Lines 19–22 of REVISEUPTOE because $t_{\bar{\imath}} \cap y \subset T$ and $\mathrm{MQ}(y') = 1$, where $y' = y \underline{\sqcap} t_{\bar{\imath}}$. In this case, we can argue exactly as we did for this situation in Case I.i that at least one of y or y' must satisfy T_i^*, and it is fine to edit T to $y \cap T$.

The second possibility is that y covers all the inside variables of T_i^*. In this case, editing T to become $T \cap y$ is performing necessary deletions. Before doing that we will have called REVISEDOWN, but it must always terminate within $O(e \log n)$ queries, so that can do no harm.

Otherwise, y is missing some inside variables of T_i^*, so so is $y' = y \underline{\sqcap} t_{\bar{\imath}}$. Thus a binary search from y' to y is guaranteed to find us a second hypothesis term that is full with respect to $T_{\bar{\imath}}^*$, so REVISEDOWN will return the target formula in $O(e \log n)$ queries.

Case II: $\mathrm{MQ}(x \underline{\sqcap} t_i) = 0$.

This implies that $T_i^* \not\subseteq t_i$. In this case, we perform a binary search from $x \underline{\sqcap} t_i$ up to x. That binary search will find the necessary additions to $t_i \cap x$ using $O(e \log n)$ membership queries, and after that the analysis is just as in Case 1. The total asymptotic query complexity is the same.

Case III:

$\mathrm{MQ}(x \underline{\sqcap} t_i) = 1$, and $\mathrm{MQ}(x \overline{\sqcap} t_i) = 0$.

Notice that in this case T_i^* must contain variables not in t_i, specifically some variables from $t_{\bar{\imath}} - t_i$. (If the necessary additions to $x \cap t$ were all outside of $t_{\bar{\imath}}$ then $\mathrm{MQ}(x \overline{\sqcap} t_i)$ would be 1.) Furthermore, since those variables are *off* in $x \underline{\sqcap} t_i$, it must be that $x \underline{\sqcap} t_i$ satisfies $T_{\bar{\imath}}^*$.

So $x \underline{\sqcap} t_i$ is a positive example that definitely satisfies term T_2^* and not term T_1^*. Now one of the previous cases applies, with the roles of i and $\bar{\imath}$ switched and $x \underline{\sqcap} t_i$ replacing x. Thus, this is equivalent to Case I in the second branch of the "Try both," and the complexity analysis is subsumed by that of Case I. □

Remark: The only place we used the restriction on x that $\mathrm{MQ}(x \overline{\sqcap} (t_1 \cap t_2)) = \mathrm{MQ}(x \underline{\sqcap} (t_1 \cap t_2)) = 0$, was to restrict the number of cases in the proof. The cases correspond to the different ways in which the initial one-term hypothesis is created.

The arguments for the case where we instead work with $t_1 \cap t_2$ to create the initial one-term hypothesis are broadly similar, and will be included in the full paper.

Theorem 9 *We can revise two term unate DNF in $O(e^2 \log n)$ queries, where e is the revision distance between the initial and target theories.*

Proof sketch. We make repeated calls to REVISEUPTOE with the error parameter set to $1, 2, 4, 8, \ldots$ until REVISEUPTOE returns success. We claim that this happens by the time the error parameter reaches or first surpasses e.

Consider first the case where the initial positive example x covers only one term and the other condition of Lemma 8 is met. Lemma 8 guarantees that the branch of the "Try both" that has the "right" value of i halts after $O(e^2 \log n)$ queries with the target theory. Furthermore, the "wrong" branch of the "Try both" also keeps track of how many revisions it has made as it goes along, so it must halt after making at most $O(e \log n)$ queries as well.

The case where the conditions of Lemma 8 do not hold because there is a target term all of whose inside variables are in the intersection of the two initial theory terms has a broadly similar argument.

Consider next the case where the initial positive example covers both terms, and at least one branch of the "Try both" catches a pivot exception thrown by binary search. The branch that throws the exception can have made at most $O(e^2 \log n)$ queries before throwing the exception. After the exception we restart the program with a new counterexample that is guaranteed to satisfy the conditions of Lemma 8.

Finally, we have the case where the initial positive example x satisfies both terms of the target, and neither branch of the "Try both" finds a pivot. This means that all of the additions done are necessary to both possible revisions (the current term to T_i^* or to $T_{\bar{\imath}}^*$). As in the discussion for the monotone case, if both initial terms are revised, in their parallel branches, to the same target term, then one of those revisions is the correct one. If they are revised to different target terms, then that revision is at least as efficient as revising them to the opposite target terms. □

5 REVISING READ-ONCE FORMULAS

In this section we outline the improved deletion-only revision algorithm for read-once formulas.

An $\Omega(e \log(n/e))$ lower bound to the number of queries is proved in [15]. It is also shown in [15] that using only one type of query, one needs a number of queries that is linear in n.

Theorem 10 *Every n-variable read-once formula φ has a revision algorithm that uses $O(e \log n)$ queries, where e is the revision distance between φ and the target concept.*

Proof outline. Let us review a bit of terminology from [14]. We assume *w.l.o.g.* that φ is monotone. If φ' is a subformula of φ, then every truth assignment $\mathbf{x}$ can be written as $(\mathbf{x_1}, \mathbf{x_2})$, called the *$\varphi'$-partition* of $\mathbf{x}$. Here $\mathbf{x_1}$ contains all the variables in φ', and $\mathbf{x_2}$ contains all the variables not in φ'.

Let φ' be a subformula of φ and let P be the path leading from the root of φ to the root of φ' in the binary tree representing φ. Then, using the commutativity of AND and OR, φ can be written as

$$((\cdots(\varphi' \circ_r \varphi_r) \circ_{r-1} \cdots \circ_3 \varphi_3) \circ_2 \varphi_2) \circ_1 \varphi_1, \qquad (1)$$

where $\varphi_1, \ldots, \varphi_r$ are the subformulas corresponding to the siblings of the nodes of P, and $\circ_1, \ldots, \circ_r$ are either $\wedge$ or $\vee$. Let the sets of variables occurring in φ_i be X_i, and the set of variables occurring in φ' be Y. These sets form a partition of $\{x_1, \ldots, x_n\}$. Now let α be the partial truth assignment that assigns 1 (resp., 0) to every variable in X_i if $\circ_i$ is AND (resp., OR), for every $i = 1, \ldots, r$. Then α is called the *partial truth assignment sensitizing φ'*.

Also, given a substitution σ, let $\varphi\sigma$ be the formula obtained by replacing each variable in φ by the corresponding constant from σ. A subformula is *constant* if it computes a constant function. Maximal constant subformulas must be pairwise disjoint. Two substitutions σ_1 and σ_2 are *equivalent* if $\varphi\sigma_1$ and $\varphi\sigma_2$ compute the same Boolean function. Then it holds that substitutions σ_1 and σ_2 are equivalent if and only if their sets of maximal constant subformulas are identical.

The learning algorithm is based on the recursive procedure FINDCONSTANT of Figure 1. This procedure differs from the corresponding procedure in [14] at one point only. The procedure FINDFORMULA is replaced by the procedure FINDNEWFORM, described below. FINDCONSTANT takes a formula φ and a counterexample $\mathbf{x}$ and returns a substitution σ, which fixes a subformula to a constant c, such that this subformula must compute constant c in any representation of the target concept. Furthermore, this subformula is a maximal constant subformula in any representation of the target concept.

In the previous version of FINDCONSTANT, at each iteration, the current formula was split by finding an approximately half-size subformula of φ, i.e., a subformula containing between 1/3 and 2/3 of the original variables (which always exists). The algorithm was recursive, so there could be a total of $O(\log n)$ levels before obtaining a constant-size subformula. For each iteration, there were three cases. In one, we used $O(\log n)$ queries and did not need to recurse. In another, we used only $O(1)$ queries to recurse. These cases are unchanged. In the third case, we needed to use a procedure called FINDFORMULA that could use $\Theta(\log n)$ queries. This is where the $O(\log^2 n)$ factor in the query complexity comes from.

The modified version of FINDCONSTANT works as follows. It either succeeds in finding a subformula (which may be φ itself) that is a maximal constant subformula in any representation of the target C, and the value of the constant, or it reduces φ to a subformula that evaluates $\mathbf{x}$ differently in φ and in any representation of C. The number of queries used in the first case is logarithmic in the number of variables of φ.

In the second case, we use k queries for some k and we obtain a subformula such that the number of its variables decreases by a factor $O(1/2^k)$. The procedure FINDCONSTANT then continues recursively. This guarantees that after $O(\log n)$ membership queries the procedure finds a subformula that is a maximal constant subformula in any representation of C, and the value of the constant. One can then find a substitution with a minimal number of variables that forces the given constant value of the subformula by a standard recursive computation that does not involve making queries.

Let us consider the version of FINDCONSTANT in Figure 1. At the bottom of the recursion no queries have to be asked: if $\mathbf{x}$ is a counterexample to a formula consisting of a single variable, then the revision must be fixing this variable to the constant different from $\mathbf{x}$.

If the input formula has more than one variable, then FINDCONSTANT starts by making sure that $\text{MQ}(\mathbf{0}) = 0$ and $\text{MQ}(\mathbf{1}) = 1$. Otherwise, the whole subformula is identically true or false. Now we pick an approximately half-size subformula φ' of φ. Then FINDCONSTANT asks the membership queries $\text{MQ}(\mathbf{0}, \alpha)$ and $\text{MQ}(\mathbf{1}, \alpha)$, where α is the partial truth assignment sensitizing φ'. Depending on the outcome of these queries, we distinguish two cases.

Case I: $\text{MQ}(\mathbf{0}, \alpha) = \text{MQ}(\mathbf{1}, \alpha) = c$ for $c = 0$ or 1.

This case remains the same as in [14], and so its discussion is omitted.

Case II: otherwise, it must be the case that $\text{MQ}(\mathbf{0}, \alpha) = 0$ and $\text{MQ}(\mathbf{1}, \alpha) = 1$. Then for every truth assignment $\mathbf{y}$ to the variables of φ' it holds that

$$\text{MQ}(\mathbf{y}, \alpha) = \psi'(\mathbf{y}), \qquad (2)$$

where ψ' is the subformula corresponding to φ' in any representation of the target concept. Now we start considering the counterexample $\mathbf{x}$, which we write as $(\mathbf{x}_1, \mathbf{x}_2)$, corresponding to its φ'-partition. By Equation 2, we can compare the known value of $\varphi'(\mathbf{x}_1)$ to $\psi'(\mathbf{x}_1)$ by asking the membership query $\text{MQ}(\mathbf{x}_1, \alpha)$. There are two possibilities, and only one of them is different from [14].

Case II.1: $\text{MQ}(\mathbf{x}_1, \alpha) = \psi'(\mathbf{x}_1) \neq \varphi'(\mathbf{x}_1)$. Then $\mathbf{x}_1$ is a counterexample to the hypothesis φ' for the target concept ψ'. Thus we can continue recursively, to find a constant substitution in a problem which has at most two-thirds of the original variables. Note that by Equation 2 we can use the *original* membership queries to simulate membership queries to the new target concept.

Case II.2: $\text{MQ}(\mathbf{x}_1, \alpha) = \psi'(\mathbf{x}_1) = \varphi'(\mathbf{x}_1) = d$.

It is in this case that we have to modify the original algorithm in [14].

Let us write φ as in Equation 1. Put $\mathbf{x}_2 = (\mathbf{x}_{2,1}, \ldots, \mathbf{x}_{2,r})$, where $\mathbf{x}_{2,i}$ corresponds to the variables in X_i. Let ψ_i be the subformula corresponding to φ_i in some representation ψ of the target. Let y_i (resp., z_i) be the value computed at $\circ_i$ in φ (resp., ψ) on input $\mathbf{x}$, for $i = 1, \ldots, r$, and let $y_{r+1} = z_{r+1} = d$. Then by definition $y_i = y_{i+1} \circ_i \varphi_i(\mathbf{x}_{2,i})$ and $z_i = z_{i+1} \circ_i \psi_i(\mathbf{x}_{2,i})$ for $i = 1, \ldots, r$. Also, $y_1 = \varphi(\mathbf{x}) \neq \psi(\mathbf{x}) = z_1$. Let β_i be the partial truth assignment that assigns $\mathbf{x}_{2,j}$ to X_j

for $j = i, \ldots, r$ and is otherwise identical to α. Then $z_i = \mathrm{MQ}(\mathbf{x}_1, \beta_i)$.

As noted, $y_{r+1} = z_{r+1}$ and $y_1 \neq z_1$. Just as the procedure FINDFORMULA, the procedure FINDNEWFORM finds an i $(1 \leq i \leq r)$ such that $y_{i+1} = z_{i+1}$ and $y_i \neq z_i$, and we return i. For this i $\varphi(\mathbf{x}_{2,i}) \neq \psi(\mathbf{x}_{2,i})$. Thus we can continue by a recursive call on φ_i using the counterexample $\mathbf{x}_{2,i}$. For a given i, one can evaluate y_i without any membership queries from φ, and one can use the remark at the end of the previous paragraph to evaluate z_i with a single membership query.

FINDFORMNEW finds the required i by performing a weighted binary search. Let $|\varphi_j|$ denote the number of variables in the subformula φ_j. Let the weights w_j be defined by $w_j = |\varphi_{j-1}| + |\varphi_j|$ for $j = 2, \ldots, r$. The binary search proceeds by updating an interval $I = [a, b]$. Initially $a = 2$ and $b = r$. Let $s = \sum_{j \in I} w_j$. Note that for the initial value of s, $s \leq (4/3)n$. Query the value ℓ such that $\sum_{j=a}^{\ell} w_j \geq s/2$ and $\sum_{j=a}^{\ell-1} w_j < s/2$. If $y_\ell \neq z_\ell$ (resp., $y_\ell = z_\ell$) then update I to $[\ell + 1, b]$ (resp., to $[a, \ell - 1]$). If I is nonempty, then update s accordingly, and continue the search. Otherwise, the search is over, and we return $i = \ell$ (resp., $i = \ell - 1$) if $y_\ell \neq z_\ell$ (resp., $y_\ell = z_\ell$). In both cases $s \geq w_\ell \geq |\varphi_i|$.

If the search is completed after k queries then the last value of s is at most $1/2^{k-1}$ times its original value. Hence for the value i returned $|\varphi_i| \leq \frac{4}{3 \cdot 2^{k-1}} n$. The bound above implies that the recursive call is made on a formula of size $O(n/2^k)$.

We also note that in order to simulate the membership queries in the recursive call by membership queries to the original target, one uses the following fact. Let γ_i be the partial truth assignment that assigns 1 (resp., 0) to Y and to all X_j with $j > i$ if $\circ_i$ is AND (resp., OR) and is identical to α on X_j for $1 \leq j < i$. Then for every truth assignment $\mathbf{w}$ to X_i, it holds that

$$\mathrm{MQ}(\mathbf{w}, \gamma_i) = \psi_i(\mathbf{w}).$$

We claim that FINDCONSTANT uses $O(\log n)$ membership queries. There are three cases to consider. The procedure GROWFORMULA uses $O(\log n)$ queries and does not make any recursive calls. If FINDCONSTANT gets into the **else** branch and it continues by looking at φ' then it uses a constant number of queries, and continues with a recursive call to an input that is at most two-thirds of the original size. Finally, if it uses the procedure FINDFORMNEW, then it makes k membership queries for some k, and it continues with a recursive call to an input that is at most $O(1/2^k)$ times the original size. Hence the upper bound follows by induction. The rest of the description and analysis of the algorithm is again identical to [14] and so it is omitted.

References

[1] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, Apr. 1988.

[2] D. Angluin, M. Frazier, and L. Pitt. Learning conjunctions of Horn clauses. *Machine Learning*, 9:147–164, 1992.

[3] A. Blum, L. Hellerstein, and N. Littlestone. Learning in the presence of finitely or infinitely many irrelevant attributes. *J. of Comput. Syst. Sci.*, 50(1):32–40, 1995. Earlier version in 4th COLT, 1991.

[4] N. Bshouty and L. Hellerstein. Attribute-efficient learning in query and mistake-bound models. *J. Comput. Syst. Sci.*, 56:310–319, 1998.

[5] R. Davis and W. Hamscher. Model-based reasoning: Troubleshooting. In H. E. Shrobe and the American Association for Artificial Intelligence, editors, *Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence*, chapter 8, pages 297–346. Morgan Kaufmann, San Mateo, CA, 1988.

[6] J. Goldsmith and R. H. Sloan. More theory revision with queries. In *Proc. 32nd Annu. ACM Sympos. Theory Comput.*, 2000. To appear.

[7] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, New York, NY, second edition, 1978.

[8] M. Koppel, R. Feldman, and A. M. Segre. Bias-driven revision of logical domain theories. *Journal of Artificial Intelligence Research*, 1:159–208, 1994.

[9] J. de Kleer, A. K. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56:197–222, 1992.

[10] R. J. Mooney. *A preliminary PAC analysis of theory revision*, volume III: Selecting Good Models, chapter 3, pages 43–53. MIT Press, 1995.

[11] D. Ourston and R. J. Mooney. Theory refinement combining analytical and empirical methods. *Artificial Intelligence*, 66:273–309, 1994.

[12] B. L. Richards and R. J. Mooney. Automated refinement of first-order Horn-clause domain theories. *Machine Learning*, 19:95–131, 1995.

[13] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.

[14] R. H. Sloan and G. Turán. On theory revision with queries. In *Proc. 12th Annu. Conf. on Comput. Learning Theory*, pages 41–52. ACM Press, New York, NY, 1999.

[15] B. Szörényi. Revision algorithms in computational learning theory. Scientific Student Competition, University of Szeged, 31 pages, 2000. (In Hungarian.).

[16] G. G. Towell and J. W. Shavlik. Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13:71–101, 1993.

ACKNOWLEDGMENTS

We want to thank Martin Mundhenk for conversations that forced us to focus on these constructions, and Andy Klapper and Maury Neiberg for putting up with us through the deadline crunch.

```
FINDCONSTANT(φ, x)
    if φ has one variable
        return  substitution σ fixing it to constant 1 − x
    if MQ(0) == 1 or MQ(1) == 0
        return  substitution σ fixing φ to the appropriate constant
    φ′ = an approximately half-size formula of φ
    α = the partial truth assignment sensitizing φ′
    if (MQ(0, α) == MQ(1, α) == c)
        then return GROWFORMULA(φ, φ′, c)
    else
        (x_1, x_2) = the φ′-partition of x
        if MQ(x_1, α) ≠ φ′(x_1)
            then FINDCONSTANT(φ(·, α), x_1)            // look in φ′
            else
                i = FINDNEWFORM(φ, φ′, x)
                FINDCONSTANT(φ_i, x_{2,i})             // look in φ_i
```

Figure 1: The procedure FINDCONSTANT.