Machine Learning, 47, 257–295, 2002

Theory Revision with Queries: DNF Formulas

JUDY GOLDSMITH* goldsmit@cs.uky.edu
Department of Computer Science, University of Kentucky, 763 Anderson Hall, Lexington, KY 40506, USA

ROBERT H. SLOAN† sloan@uic.edu
Department of Computer Science, University of Illinois at Chicago, and National Science Foundation

GYÖRGY TURÁN** gyt@uic.edu
Math, Stat., & Computer Science Department, University of Illinois at Chicago, Research Group on AI of Hungarian Academy of Sciences at University of Szeged

Editor: Jyrki Kivinen

Abstract. The theory revision, or concept revision, problem is to correct a given, roughly correct concept. This problem is considered here in the model of learning with equivalence and membership queries. A revision algorithm is considered efficient if the number of queries it makes is polynomial in the revision distance between the initial theory and the target theory, and polylogarithmic in the number of variables and the size of the initial theory. The revision distance is the minimal number of syntactic revision operations, such as the deletion or addition of literals, needed to obtain the target theory from the initial theory. Efficient revision algorithms are given for three classes of disjunctive normal form expressions: monotone k-DNF, monotone m-term DNF and unate two-term DNF. A negative result shows that some monotone DNF formulas are hard to revise.

Keywords: theory revision, query learning, computational learning theory, knowledge revision, disjunctive normal form, Boolean function learning

1. Introduction

Consider the following situation. You hire a domain expert (whom we shall call "Mommy") and a knowledge engineer to develop an expert system for predicting what your picky preschooler will eat. The following "pretty close" initial theory is developed:

WillEat := *VeryBland* OR ((NOT *vegetables*) AND *bland* AND *Meat*).

Then, though you use the initial theory as a general guide, you happen to observe the preschooler consume a full pound of grilled lamb ribs in spicy black pepper-teriyaki marinade. You must *revise* or *edit* your initial theory, perhaps to either

WillEat := *VeryBland* OR((NOT *vegetables*) AND *Meat*),

*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.

or

WillEat := *VeryBland* OR ((NOT *vegetables*) AND *bland* AND *Meat*) OR *Lamb*.[1]

This is the problem known in machine learning as *theory revision* (or *knowledge-base revision*) (e.g., Towell & Shavlik, 1993; Ourston & Mooney, 1994; Richards & Mooney, 1995; Koppel, Feldman, & Segre, 1994). Note that the artificial intelligence term *theory*, as used in this paper, has the same meaning as what the computational learning theory community calls a *concept*.[2] Thus the machine learning problem of theory revision may be viewed as the problem of correcting a given, roughly correct concept or theory.

A typical application for theory revision is revising a theory such as a set of rules generated by an expert. A common assumption is that the theory comes with a set of labeled examples, which always includes some that disagree with the classification provided by the theory. Typically in the AI literature one is supposed to make a "small" revision to the theory so that it classifies all the given examples correctly.

In this paper we study the theory revision problem for several sorts of disjunctive normal form (DNF) formulas, in the context of Angluin's (Angluin, 1988) model of learning with membership and equivalence queries. Briefly, a membership query allows the learner to ask the classification of any instance, and an equivalence query allows the learner to ask whether its hypothesized theory is the correct one, and if the hypothesized theory is incorrect, then the learner is given a counterexample—one instance for which its hypothesized theory and the true theory give different classifications.

The use of equivalence queries appears to be reasonable in the scenario just discussed, as each equivalence query can be simulated by cycling through the given set of examples until we find a counterexample to our current version of the theory. If no counterexamples are found, then the learning process terminates. This simulation can never require more queries than the worst-case number in the formal model. In addition, when the initial theory is provided by an expert, it may be realistic to assume that the expert can correctly answer membership queries.

In a companion paper (Goldsmith et al., 2001), we give results for various other forms, including Horn sentences and read-once formulas.

1.1. Related work

Theory revision is one of several problems, also including *theory restructuring*, which is aimed at making a theory more efficient or transparent, that make up the area of *theory refinement* in machine learning (see, e.g., Wrobel, 1995). There are many systems for theory revision in propositional and predicate logic (e.g., Koppel, Feldman, & Segre, 1994; Ourston & Mooney, 1994; Richards & Mooney, 1995; Towell & Shavlik, 1993). For instance, the EITHER system (Ourston & Mooney, 1994) uses deductive, abductive, and inductive components to revise a propositional Horn theory from a given set of counterexamples, and the KBANN system (Towell & Shavlik, 1993, 1994) uses neural networks for the same task.

Formal work in theory revision includes (Koppel, Feldman, & Segre, 1994). They considered the problem of fixing a given theory for a given set of examples, and gave an algorithm

together with a proof of its convergence under certain assumptions. Argamon-Engelson and Koppel also considered a version of the revision problem called *theory patching* where there is a prespecified part of the initial theory that cannot be modified (Argamon-Engelson & Koppel, 1998).

Mooney (1995) formulated an approach to the study of theory revision in computational learning theory based on *syntactic distances*. 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. Mooney proposed considering the PAC-learnability of the class of concepts having a bounded syntactic distance from a given concept representation. This approach provides an interesting formalization of the availability of *prior knowledge* to the learner, even in propositional logic. Mooney observed that for a fixed number of revisions, the polynomial size of these concept classes implies that only *logarithmic* sample size is needed. Mooney's work was extended by Greiner. Greiner concentrates on Horn sentences, and we will discuss his results in our companion paper (Goldsmith et al., 2001).

The problem of theory revision appears in several other guises, including *program debugging* and *system diagnosis*. Indeed, one of the motivating examples in Angluin's seminal paper on learning with queries (Angluin, 1988) is Shapiro's interactive program debugging system (Shapiro, 1983), where the user is required to provide the answers to the queries. Reducing the query complexity of Shapiro's algorithm has been of some interest in machine learning (e.g., Alexin, Gyimóthy, & Boström, 1997).

When theories are expressed as propositional Boolean formulas, the theory revision problem is related to *fault analysis of circuits* in switching theory (see, e.g., Kohavi, 1978). There a major problem is simply fault *testing*, determining whether the given circuit works correctly. (Faulty circuits may simply be thrown away.)

Model-based diagnosis is a similar but slightly less closely related field. Davis and Hamscher (1988) give a good survey of the field through 1987 or so; there is also lots of more recent work (e.g., de Kleer, Mackworth, & Reiter, 1992; Marcotte et al., 1992). One of the key concerns of model-based diagnosis is handling a very broad set of *errors* that is typically unknowable in advance. However, for theory revision, while a great variety of errors are considered, the set of possible errors can be specified in advance. In the revision of theories in propositional logic it may be assumed that literals and clauses are *superfluous* and/or *missing*. We view these possibilities as operations available to edit the initial formula, and therefore refer to them as *revision operators*. Many model-based diagnosis systems rely implicitly or explicitly on solving some NP-complete or inherently exponential problem (e.g., enumerating all the minimal hitting sets of a set system (Reiter, 1987)).

Several areas of machine learning have at least some connection to theory revision. The study of *drifting concepts* in learning theory (Helmbold & Long, 1994) may perhaps be viewed as a "dynamic" version of theory revision. Much has been written about *learning from partial information*. A few examples include SOAR (Laird, Newell, & Rosenbloom, 1987) in systems-building machine learning (Case et al., 1997) and (Jain & Sharma, 1991) in inductive inference, and Rivest and Sloan (1994) in PAC learning. Of course, theory revision can be viewed as simply "learning from *a lot* of partial information." (And all

machine learning is then the special case of learning from partial information where the partial information is the empty set.) However, in practice, different techniques can be brought to bear when one knows a close approximation to the correct concept instead of just some partial information about it. In this work, we seek to formally model this phenomenon.

1.2. Overview of this paper

We extend Mooney's syntactic distance approach to query models, in particular to learning with equivalence and membership queries. We consider the complexity of concept revision for several subclasses of DNF formulas. We consider two sets of revision operators, or, equivalently, error models. Our first set of revision operators allows for the replacement of an *occurrence of a variable* by a constant 0 or 1. This corresponds to allowing the *deletion* of a variable or a clause in a DNF in theory revision (Argamon-Engelson & Koppel, 1998), and to the "stuck at" faults in circuit fault analysis. Hence we call this the *deletions-only* model of theory revision. In this model, a *revision algorithm* for a formula φ has to identify a target concept represented by a formula that can be obtained from φ by replacing certain variable occurrences by constants, using equivalence and membership queries. Our second set of revision operators also permits the addition of literals to terms (arbitrary literals in the case of unate or more general formulas, only positive literals in the case of monotone formulas); we call this model the *general* model.

Definition. A revision algorithm is *query efficient* if its query complexity depends polynomially on the syntactic distance between the initial formula and the target, but only polylogarithmically on both the total number of variables in the universe and the size of the concept representation.

We will normally write simply "efficient" when we mean "query efficient," because all the algorithms presented in this paper have *running time* that is not much larger than the query complexity.

It is an interesting feature of revision complexity that a single formula, such as a particular DNF formula, induces a concept class, rather than being a single concept in a concept class. Thus a revision query complexity is associated with each individual formula through the (standard) learning complexity of the concept class of its revisions, perhaps parameterized by a revision distance.

Moreover, technically the algorithms we give in this paper are all meta-algorithms, as they take any formula from a specified class of formulas (e.g., monotone DNF formulas) and then function as a revision algorithm for the corresponding concept class. Incidentally, the choice of revision operator(s) has a double role. First it defines the concept class. For instance, for a monotone k-DNF, the deletions-only revision operators give us a class of certain monotone k-DNFs, but the general revision operators give us a class of monotone DNFs with arbitrarily many variables per term. Second, the choice of revision operator(s) determines the revision distance, which gives us a performance metric.

The efficiency criterion is similar to the notion of efficiency for *attribute efficient* learning algorithms (Blum, Hellerstein, & Littlestone, 1995; Bshouty & Hellerstein, 1998). In fact,

there seems to be an interesting relationship between concept revision and attribute efficient learning. Informally, in attribute efficient learning, most variables are *irrelevant*, while in concept revision most variables *act in a known way*.

Let e be the syntactic distance between an initial formula and a target formula. In Section 3, we give a revision algorithm for k-DNF formulas over n variables using $O(ke \log n)$ queries in the deletions-only model. In Section 4, we give a revision algorithm for monotone DNF in the general model. Its query complexity for revising a monotone m-term DNF in a universe of n variables is $O(m^3 e \log n)$ This meets our definition of "efficient revision" when m is polylogarithmic in n—that is, if m is small or n is very large. Very large values of n will arise in practice when we have a universe with very many variables, but not so many of them occur in the target formula. The query complexity of learning a monotone DNF from scratch is $O(mn)$ (Angluin, 1988). Thus, for small e, revision is more efficient than learning from scratch whenever m is significantly smaller than n. In the deletions-only model, we show that the monotone DNF revision algorithm simplifies considerably and has query complexity $O(m^2 + em)$.

In Section 5, we give an efficient revision algorithm for two-term unate DNF using the general model of revision operators. Bshouty et al. (1994), building on earlier work (Angluin, Hellerstein, & Karpinski, 1993), give a general technique for converting membership and equivalence query algorithms for monotone classes to corresponding unate classes. However, we cannot use Bshouty et al.'s conversion from monotone to unate in our context of theory revision, because the conversion multiplies query complexity by a factor of n. That conversion gives a query complexity of $O(n^2)$ for learning any unate DNF with a constant number of terms; the revision algorithm we present for two-term unate DNF has query complexity $O(e^2 \log n)$, where e is the revision distance.

In general, as the concept classes considered are small, but very efficient learning algorithms are required, one needs somewhat different techniques than the previous ones (Angluin, 1988; Angluin, Hellerstein, & Karpinski, 1993) for these much studied concept classes.

We also give an example of a monotone DNF for which there is no efficient concept revision algorithm, in Section 6. In Section 7, we mention some further results and open problems.

2. Notation

We consider a variety of types of Boolean formulas here. We use standard notions from propositional logic, such as variable, literal, term (a conjunction of literals), disjunctive normal form (DNF), etc. A formula is

- *monotone* if no variable occurs negated;
- *unate* if no variable ever occurs in it both negated and unnegated;
- *k-DNF* if it is DNF and every term contains at most k literals.

For a given fixed universe of variables $\{v_1, \ldots, v_n\}$, we can represent a truth assignment or *instance* as a Boolean vector $x \in \{0, 1\}^n$. The all-zero (resp., all-one) vector is denoted by $\mathbf{0}$ (resp., $\mathbf{1}$).

A DNF (resp., monotone DNF) formula can be viewed as a collection of subsets of the set of literals (resp., 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. Moreover, in a mild abuse of notation, we then sometimes treat those elements of $\{0, 1\}^n$ as subsets of $\{1, \ldots, n\}$.

Definition. For bit vectors y and z, and by extension, for monotone terms, we write $y \leq z$ if y is below z in the Boolean lattice; that is, if z has a 1 in every position where y has a 1.

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

When all formulas are monotone, we define the operation of *turning off* a bit or position of an instance as setting that bit to 0. In the unate case, only for positions of known orientation, we again define *turning off* position i of instance x to mean setting $x[i]$ to 0 if $x[i]$ has positive orientation, and to 1 if $x[i]$ has negative orientation.

We will need to combine terms with instances in various ways.

A term and an instance can be combined to form a term. We define the operation $t \cap x$ for a term t and an instance x to be a *term* that is the conjunction of those literals in t that are satisfied by x. Thus, for example $v_1 \bar{v}_2 \bar{v}_5 \cap 11000 = v_1 \bar{v}_5$.

A term and an instance can be combined to form an instance. When we intersect an instance x with a term t to form a new instance, intuitively, we want to say that we will leave all bits of x that occur in t unchanged, and turn off all variables of x that do not occur in t. For monotone formulas this is straightforward, and we define $x \cap t$ in the monotone case as

$$(x \cap t)[i] = \begin{cases} x[i] & \text{if } v_i \in t \\ 0 & \text{otherwise.} \end{cases}$$

In the unate case, there may be many variables whose orientation is unknown, and we cannot reliably turn off those variables. Instead, 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 a set K of literals of known orientation. The literals in the initial theory φ are always in K, as are any literals in the current hypothesis h. Both operations leave variables of x that are mentioned in t unchanged, and both turn off variables of x that are not mentioned in t but that have known orientation. The operations differ in how they treat variables of unknown orientation not mentioned in t: *intersect down* (denoted $\underline{\cap}$) flips those variables of x, and *intersect up* (denoted $\overline{\cap}$) leaves those variables unchanged.

Formally, let K be the set of literals of known orientation, each oriented in its on direction. "Intersect down" is defined by

$$(x \underline{\cap} t)[i] = \begin{cases} x[i] & \text{if one of } v_i, \bar{v}_i \in t \\ 0 & \text{if } v_i \in K \backslash t \\ 1 & \text{if } \bar{v}_i \in K \backslash t \\ \bar{x}[i] & \text{otherwise,} \end{cases}$$

and "intersect up" is defined by

$$(x \,\overline{\cap}\, t)[i] = \begin{cases} x[i] & \text{if one of } v_i, \bar{v}_i \in t \\ 0 & \text{if } v_i \in K \setminus t \\ 1 & \text{if } \bar{v}_i \in K \setminus t \\ x[i] & \text{otherwise.} \end{cases}$$

(A small mnemonic for this notation is that if the term comes first, then a term is returned, as in $t \cap x$, and if the instance comes first an instance is returned, as in $x \,\overline{\cap}\, t$.)

The differences of terms and instances. For two instances $x, y \in \{0,1\}^n$, 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 also to indicate the symmetric difference of two terms, namely the set of literals that appears in exactly one of the two terms. In the cases where we use this notation, literals will not occur with different orientations in the different terms.

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

2.1. *Revision: Distance, operators*

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 model, we are also allowed to add literals to a DNF term. In the monotone case, of course, we can add only unnegated variables. In the unate case, we add the restriction that a literal that appears in the initial formula cannot appear negated in the target formula. This technical assumption is needed by our algorithm.

Note that for DNF formulas, our definitions allow us to 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. However, under our definitions, we cannot have a target with more terms than were in the initial DNF, as the additions are not allowed to start a new term.

Notice finally that the revision distance for simply deleting any one term of a DNF with no replacement is only 1, since this can be done merely by fixing any one variable of the term to be false.

2.2. *Learning model*

Angluin introduced two types of oracles for learning formulas: a membership oracle and an equivalence oracle (Angluin, 1988). These are the tools that we use here for revising formulas.

In an *equivalence query*, the learning algorithm proposes a *hypothesis*, a concept h from the concept class, and the answer depends on whether $h = \varphi^*$, where φ^* is the target concept.[3] If so, the answer is "Yes!" and the learning algorithm has succeeded in its goal of exact identification of the target concept. Otherwise, the answer is a *counterexample*: any instance x such that $h(x) \neq \varphi^*(x)$.

In a *membership query*, the learning algorithm gives an instance x, and the answer is either 1 or 0, depending on $\varphi^*(x)$; that is, $\mathrm{MQ}(x) = \varphi^*(x)$, where again φ^* is the target concept. In each algorithm and lower bound considered here, we allow the concepts TRUE and FALSE as equivalence queries. Note that membership queries are on *instances* and equivalence queries are on *formulas*.

3. Deletions-only revisions of monotone k-DNF expressions

In this section we present a revision algorithm for monotone DNF in the deletions-only model. The algorithm is efficient for k-DNF.

Lemma 1. *Let x be a satisfying truth assignment for an unknown monotone DNF ψ. Then one can identify a term of ψ satisfied by x, using $O(k \log n)$ membership queries to ψ, where k is the maximum number of variables in any term of ψ.*

Proof: We give an informal description of a procedure FINDMINTERM(ψ, x) that performs this task. An initial membership query MQ($\mathbf{0}$) decides if ψ is TRUE. Let us assume that this is not the case. For every term t in the unknown formula, let us consider the sequence of the indices of variables occurring in t in decreasing order. Let t^*, corresponding to the sequence $i_1 > \cdots > i_\ell$, $\ell \leq k$, be the lexicographically first term in ψ satisfied by x. (In this ordering, a set precedes another if at the first position where they differ it has a smaller number (or none). Thus, e.g., $\{10, 7\}$ precedes $\{10, 7, 3\}$, which precedes $\{10, 7, 4, 2\}$.) We claim that this term can be found using $O(k \log n)$ membership queries to ψ.

Let x_j, for $0 \leq j \leq n$, be the vector obtained from x by leaving it unchanged on the first j positions, and setting all its other bits to 0. Then i_1 is the smallest number j such that $\mathrm{MQ}(x_j) = 1$. It can be found using binary search on the sequence $\mathbf{0} = x_0 \leq x_1 \leq \cdots \leq x_n = x$, and it holds that $i_1 > 0$. Now ask the membership query $\mathrm{MQ}(e_{i_1})$, where e_{i_1} has a single bit that is 1, at the i_1th position. If the answer is 1, then $t^* = x_{i_1}$. Otherwise, let $x_j^{i_1}$, for $0 \leq j < i_1$ be the vector obtained from x by keeping it unchanged on the first j positions and on the i_1th position, and turning all its other bits off. Then i_2 is the smallest number i such that $\mathrm{MQ}(x_i^{i_1}) = 1$ and again it can be found by binary search. Now ask the membership query $\mathrm{MQ}(\mathbf{e}_{i_1,i_2})$, where $\mathbf{e}_{i_1,i_2}$ is 1 in positions i_1 and i_2, and it is 0 otherwise. If the answer is 1 then $t^* = x_{i_1}x_{i_2}$. Continuing this process, t^* is identified using $O(k \log n)$ membership queries.

We note that FINDMINTERM can be applied to any monotone DNF ψ; it is only the upper bound for its query complexity which depends on the size of the terms in ψ. □

The query algorithm REVISEkDNF is shown in Algorithm 1. It starts with initial formula φ as its initial hypothesis, and repeatedly makes equivalence queries with current hypothesis h.

Algorithm 1 REVISEkDNF(φ)

```
1: h = φ
2: while (x = EQ(h)) ≠ "Yes!" do
3:   if x is a negative counterexample then
4:     delete all terms satisfied by x from h
5:   else
6:     t* = FINDMINTERM(ψ, x)
7:     h = h ∨ t*
8:     Delete from h every proper implicant of t*
9:   end if
10: end while
```

First we give a small example, and then we will argue that the algorithm's complexity is reasonable.

As a small example, consider revising $wxy \vee wxz$ to $wx \vee xz$. Assume that the first counterexample is 1100. Then we use FINDMINTERM(ψ, 1100) to find the term wx, we add it to the original formula, and we delete the terms wxy and wxz. Thus, we are left with the hypothesis wx. Now assume that the next counterexample is 0111. Then FINDMINTERM(ψ, 0111) finds the term xz, and adding it to wx, it identifies the target concept. We note that terms may disappear and reappear in the course of the revision process.

Lemma 2. *Let φ be an arbitrary monotone DNF. Then* REVISEkDNF(φ) *is a correct revision algorithm for φ. The number of calls to* FINDMINTERM *is at most the revision distance between φ and the target.*

Proof: The algorithm terminates only when it identifies the target; therefore it is sufficient to prove the upper bound for the number of calls to FINDMINTERM.

Whenever we receive a negative counterexample x, we delete from our hypothesis those terms t that x satisfies. Note that if $t(x) = 1$, then deleting variables from t will not change that condition. Thus, if $t(x) = 1$ and x is a negative counterexample, the only way to rectify that is to delete t from the formula. Thus the deleted terms *must* be deleted in any editing to reach the target. So when we get negative counterexamples we "spend" only one query for at least a gain of one in revision distance. Also notice that the deleted terms must be original terms of φ, not terms added by FINDMINTERM. This holds because if a term added by FINDMINTERM is satisfied by some x, then x is a positive example. Thus terms added by FINDMINTERM will never be deleted.

Now let us assume that we receive a positive counterexample x. Then x satisfies a term in the target that is obtained from some term in φ by deleting at least one variable. We run the procedure FINDMINTERM to find such a term t^*, and then we add t^* as a new term of h.

Notice that right after setting h to $h \vee t^*$, our hypothesis h will generally not be as compact as possible and sometimes will not be of the form of an allowable revision of φ. (e.g., with $n = 3$, we might update h from $xy \vee xz$ to $xy \vee xz \vee x$, which has more terms than the original formula.) Therefore, we need to transform h to a semantically equivalent formula

that is a revision of φ, by removing all proper implicants of t^* from h. Below we argue that the output h is indeed a revision of φ.

We now finish our analysis of the number of calls to FINDMINTERM, begun with the discussion of negative counterexamples. Each call to FINDMINTERM on a positive counterexample finds a distinct new minterm of the target that was not contained in φ. At least one edit is required to obtain each such minterm. So the total number of calls to FINDMINTERM is at most the revision distance between φ and the target.

Now we must argue that there is a revision that edits φ to the h produced at the end of the main **while** loop.

First we introduce some terminology. Let φ be of the form $t_1 \vee \cdots \vee t_\ell$, and let ψ be a representation of the target. Assume that a monotone DNF g is of the form $t_1 \vee \cdots \vee t_m \vee t_1^* \vee \cdots \vee t_r^*$, for $m \leq \ell$ such that the following properties hold:

- every t_j^* is a minterm of ψ but not an implicant of φ
- every t_i for $m < i \leq \ell$ (i.e., the terms of φ missing from g) either is satisfied by a negative instance of ψ, or is a proper implicant of some t_j^*
- no t_i for $1 \leq i \leq m$ (i.e., the terms of φ present in g) is a proper implicant of any t_j^*.

We claim that under these conditions g is a revision of φ and we can even find a suitable revision in polynomial time (without using any queries). This follows by considering the bipartite graph that has the terms of φ as its left vertices and the terms of g as its right vertices, with edges corresponding to the implicant relation. We show that this bipartite graph has a matching that matches every right vertex. Finding such a matching then provides the required revision.

Every t_j^* is a term of ψ, and there are r distinct terms t_{i_j} in φ that are revised to these terms by the deletion of at least one variable from each term. The correspondence between old and new terms is not necessarily unique, but any such correspondence is suitable. By the third condition, $i_j > m$ for every j. Thus the edges (t_j^*, t_{i_j}) for $j = 1, \ldots, r$, together with the edges (t_i, t_i) for $i = 1, \ldots, m$ indeed form a matching.

To complete the proof, we show by induction on the iteration of the **while** loop of REVISEkDNF that h satisfies the above conditions for g. In the beginning, $h = \varphi$, so the claim holds vacuously.

If the last counterexample is negative, then the number of terms missing from h increases. The newly deleted terms are satisfied by a negative instance of ψ, showing that the second condition remains valid. The validity of the other two conditions is automatic.

If the last counterexample is positive, then the newly added term satisfies the first condition by the correctness of the procedure FINDMINTERM. All the proper implicants of the new term are deleted from h, and this implies the validity of the other two conditions. □

Theorem 3. *There is a revision algorithm for monotone DNF formulas in the deletions-only model, using $O(ke \log n)$ queries for monotone k-DNF formulas on n variables, where e is the revision distance between the initial and the target formulas.*

Proof: This theorem follows from first Lemma 2 and then to get the query complexity multiply e calls to REVISEkDNF by the $O(k \log n)$ queries per call from Lemma 1. □

4. 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 monotone two-term DNF formulas in Goldsmith and Sloan (2000).

4.1. Description of algorithm

The heart of the construction is the procedure MONOREVISEUPTOE, Algorithm 2. It takes as parameters an m-term monotone DNF formula φ over a universe of n variables, and e, the assumed revision distance from φ to the target. If e is in fact too small, MONOREVISEUPTOE(φ, e) fails (formally, it returns the value "Failure"), and e is doubled. The claim, discussed in the next subsection, is that whenever the revision distance is at most e, MONOREVISEUPTOE(φ, e) succeeds, and uses only a bounded number of each type of query.

At a very high level, MONOREVISEUPTOE works as follows. It initializes its hypothesis h to FALSE (the empty disjunction), and repeatedly asks for a counterexample to its current hypothesis. The algorithm is designed so that each such counterexample is always a positive counterexample. Each such counterexample is used either to create a new hypothesis term or to delete variables from existing hypothesis terms. When a new hypothesis term is created (which is how the first counterexample will be used assuming that the target formula is not FALSE), it is always a superset of a term in the target formula. In other words, all necessary additions of variables are made when the hypothesis is created.

Once a hypothesis term is created, we never delete it, and we never add any additional variables to it. Our general strategy is to use counterexamples to delete variables from hypothesis terms whenever possible; we create a new term only when that is not possible. In the pseudocode, the **if** at Line 3 determines which of those two cases we are in; Lines 4–7 handle deletions from current terms; Lines 8–35 handle creation of new terms.

It may be instructive to compare the strategy of MONOREVISEUPTOE with the strategy of Angluin's original query learning algorithm for monotone DNF (Angluin, 1988). Angluin's algorithm uses positive counterexamples from equivalence queries to initialize each hypothesis term to a term covering a target term, and uses membership queries to delete extra variables from hypothesis terms. MONOREVISEUPTOE also uses uses positive counterexamples from equivalence queries to initialize each hypothesis term. However, in order to obtain a query complexity that is only logarithmic in n, MONOREVISEUPTOE must also take advantage of the fact that the target term(s) satisfied by this positive counterexample must be "close" to one of the initial theory terms. To make deletions from hypothesis terms, MONOREVISEUPTOE uses a positive counterexample from an equivalence query followed by membership queries to see which terms, if any, deletions should be made from.

When MONOREVISEUPTOE uses a positive counterexample z to add a new term "close" to an initial theory term, it adds the variables of the initial theory that are 1 in both z and the initial theory term, plus any other necessary variables corresponding to 1's in z. A sort of binary search is used to find those necessary variables.

Algorithm 2 MONOREVISEUPTOE(φ, e). Revises m-term φ

```
 1: h = ∅ {the initial hypothesis}
 2: while (x = EQ(h)) ≠ "Yes!" and h ≤ m terms and e > 0 do
 3:   if MQ(x ∩ t) == 1 for some t ∈ h then {delete vars from hyp.}
 4:     for all t ∈ h for which MQ(x ∩ t) = 1 do
 5:       e = e − |t − (t ∩ x)|
 6:       t = t ∩ x
 7:     end for
 8:   else {find a new term to add to the hypothesis}
 9:     min = e
10:     FoundATerm = false
11:     for all t ∈ φ do
12:       new = t ∩ x
13:       numAddedLits = 0
14:       while MQ(new) == 0 and numAddedLits < e do
15:         l = BINARYSEARCH(new, x)
16:         new = new ∪ {l}
17:         numAddedLits = numAddedLits + 1
18:         if MQ(x − {l}) == 1 then
19:           x = x − {l} {still a positive counterexample}
20:           restart the for all t loop with this x by backing up to
              line 9 to reset other parameters
21:         end if
22:       end while
23:       if MQ(new) == 1 then
24:         x = new
25:         FoundATerm = true
26:         min = min(numAddedLits, min)
27:       end if
28:     end for
29:     if not FoundATerm then
30:       return "Failure"
31:     else
32:       h = h ∨ x {now treating x as monotone term}
33:       e = e − min
34:     end if
35:   end if
36: end while
37: if x == "Yes!" then
38:   return h
39: end if
40: return "Failure"
```

4.1.1. Binary search. For any positive instance *pos* and negative instance *neg* it is always true that *pos* satisfies a target term not satisfied by *neg*. When $neg \leq pos$, and the target is a monotone DNF, it is fairly intuitive that we can use binary search to find one variable in $pos \backslash neg$ that is part of a target term. The algorithm BINARYSEARCH(*neg*, *pos*) does this. In fact, it is written more generally, in terms of literals, because we will reuse it for unate DNFs in Section 5.

To be precise, we can find a set A of variables and a variable $l \in pos \backslash (neg \cup A)$ such that $\mathrm{MQ}(neg \cup A) = 0$ but $\mathrm{MQ}(neg \cup A \cup \{l\}) = 1$. More generally, in the unate case, we can find a set of literals A on which *pos* and *neg* disagree, and an additional literal $l \notin A$ where *pos* and *neg* disagree, with the property that if x agrees with *neg* on $(A \cup \{l\})$ and x agrees with *pos* on all other positions, then $\mathrm{MQ}(x) = 0$, but $\mathrm{MQ}(x$ with position l flipped to agree with $pos) = 1$.

This is done by the procedure BINARYSEARCH, Algorithm 3. We summarize the behavior of BINARYSEARCH in the following proposition.

Proposition 4. *Let φ^* be a unate DNF and let pos and neg be two instances such that $\varphi^*(pos) = 1$ and $\varphi^*(neg) = 0$. Binary search uses at most $\lceil \log n \rceil$ membership queries and returns a literal that is in φ^* and that is set to on in pos and set to off in neg.*

Notice that in the complete algorithm MONOREVISEUPTOE, we actually wish to repeat the binary search until the term in question covers a target term. This occurs in lines 14–22 of MONOREVISEUPTOE. A similar idea was used by Angluin, Hellerstein, and Karpinski (1993) and is explicitly discussed by Uehara, Tsuchida, and Wegener (1997).

4.1.2. Pivots. A complication can arise when we call BINARYSEARCH repeatedly to find all the missing variables in a term. We would like to consider l a necessary addition to *neg*, but, if *pos* satisfies several target terms, it may be necessary to add l to *neg* to satisfy one of those terms but not another. This could lead to our building up *neg* to cover more than one target term in an inefficient manner. We call such an l a *pivot*, because the choice of which target term will be covered pivots on whether l is added to *neg*. We can recognize a pivot

Algorithm 3 BINARYSEARCH(*neg*, *pos*).

Require: $\mathrm{MQ}(neg) == 0$ and $\mathrm{MQ}(pos) == 1$
1: **while** *neg* and *pos* differ in more than 1 position **do**
2: Divide $neg \otimes pos$ into approximately equal-size sets d_1 and d_2.
3: Put $mid = neg$ with positions in d_1 replaced by *pos*'s values
4: **if** $\mathrm{MQ}(mid) == 0$ **then**
5: $neg = mid$
6: **else**
7: $pos = mid$
8: **end if**
9: **end while**
10: **return** l, the literal of *pos* in unique position where $pos \neq neg$

because without it, *pos* still covers a target term, so $\mathrm{MQ}(pos \backslash \{l\}) = 1$. If a pivot is found in the course of the binary search, we throw it out and restart the search. Notice that $pos \backslash \{l\}$ satisfies fewer target terms than *pos*.

4.1.3. More detailed description of algorithm. As we show in Lemmas 5 and 6, given φ, MONOREVISEUPTOE(φ, e) constructs a hypothesis monotone DNF formula h so that, at each stage of the construction each term of h covers a (distinct) term of the target formula. Thus, at each stage of the construction, we get a positive counterexample, x, to h.

If x satisfies a target term already covered by a term of h, then x is used to delete variables from all hypothesis terms that cover a term satisfied by x. This is decided by asking $\mathrm{MQ}(x \cap t)$ in Line 4. Since x is a positive counterexample, for any $t \in h$, it must be that $t \cap x \subset t$, so this yields at least one deletion.

Otherwise, x is used to add a new term to the hypothesis. For each term t of the initial formula, 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 extension a candidate new term. (Note that in the **if** branch of the main **while** loop we consider terms in the current hypothesis, but in the **else** branch we consider terms of the initial theory.) That extension is then treated as the positive counterexample for each subsequent initial term.

Why do we alter the positive counterexample x for *every* initial theory term t where we find an unambiguous extension of $t \cap x$ with no more than e additions? Intuitively, we need to do this only for an initial theory term t that is closest to a target term satisfied by x. Of course, we do not know which initial theory term that is. So, as we go along, we give each initial theory term the opportunity to make deletions from x. There is no harm in the deletions contributed by the "wrong" initial theory terms, since x still remains a positive instance and therefore still covers a target term. A more precise understanding of what is going on is given in the proof of Lemma 7.

4.2. Monotone DNF correctness and query complexity

For all of the lemmas below, we assume that the initial formula φ is an m-term monotone DNF formula, and the target φ^* is an m'-term monotone DNF formula ($m' \leq m$) with revision distance at most e from φ. Recall that $m' \leq m$ because we are not allowed to create entirely new terms.

In this section, we will generally use lower-case t for a term of the initial hypothesis, and upper-case T^* for a term of the target formula.

The following lemma shows that any counterexample must be positive.

Lemma 5. *Algorithm* MONOREVISEUPTOE *maintains the invariant that each of its hypothesis terms covers some term of the target.*

Proof: The initial hypothesis is the empty disjunction FALSE, so the first counterexample is positive, and the statement about hypothesis terms is vacuously true.

Each positive counterexample x satisfies at least one target term T^*. If T^* is already covered by a term $t \in h$, then the **if** in Line 3 of MONOREVISEUPTOE must be true, and t

is replaced by $t \cap x$, which still covers T^*. (Note that x cannot satisfy any $t \in h$, since x is a positive counterexample to h. This forces $(t \cap x) \subset t$.)

If x does not satisfy any term covered by h, then, either MONOREVISEUPTOE returns "Failure", or a new term is added to h. The new term is the x from the final execution of Line 24, and in Line 23 it was determined that that value of x is a positive instance, so the new hypothesis term covers some target term covered by x. □

Our next lemma shows that no two terms in h cover the same target term.

Lemma 6. *If x is used to add a new term to h, then the new term does not cover any target term already covered by h.*

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

If both t and t' did cover some target term T^*, then $T^* \subseteq t \cap t'$, implying that $\mathrm{MQ}(t \cap t') = 1$. □

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

Next, we argue that we are building the hypothesis efficiently when we add a new term. That is, the new term will, over the course of its initial construction and later deletions to it, be charged for at most the minimum number of edits necessary.

Lemma 7. *Let x be a counterexample that is not used to make deletions from an existing term of h, such that every target term that x satisfies is within revision distance e from some initial theory term. Then the iteration of the outer loop of* MONOREVISEUPTOE *for this x does not fail, so it will add a new hypothesis term t_x to h.*

Furthermore, let T^* be a target term covered by t_x, let $t_0 \in \varphi$ be a term of the initial formula that minimizes the revision distance from t_0 to T^*, and let d be that revision distance. In the iteration of the outer loop of MONOREVISEUPTOE, *the sum of the number of additions charged to initializing the hypothesis term t_x from x (i.e., the value of min in Line* 33 *of* MONOREVISEUPTOE) *plus the number of deletions later made to this hypothesis term is at most d.*

Proof: First we must argue that the algorithm does not fail. In the beginning, pivots may be found some number of times, causing the algorithm to back up to Line 9 without changing h, and restart the **for all** $t \in \varphi$ loop that begins at Line 11. The algorithm cannot fail until it has completed that loop, so there is no possibility of failing until we start an iteration of the **for all** $t \in \varphi$ loop in which no pivots are found.

In such an iteration, we eventually must set t to be an initial theory term within revision distance e of a target term satisfied by x. The repeated calls to BINARYSEARCH to find additions to $t \cap x$ will find precisely the (at most e) variables in $U^* \setminus (t \cap x)$ for some target term U^*. (Otherwise, if BINARYSEARCH finds a variable not in U^*, then it would be a pivot.)

At that point, the algorithm finds a value of *new* with at most e additions to $t \cap x$, so it will not fail.

Now we discuss the number of revisions imputed to the term t_x. First we point out that it is slightly sloppy to speak of x, because x is a variable whose value changes during the execution of the algorithm's outer loop. The value of variable x is altered only by changing 1's to 0's. That is what happens when we restart after finding a pivot. Other than pivots, we update x only in those iterations of the **for all** $t \in \varphi$ loop that succeed in finding additions to $t \cap x$. In this case, intersecting x with t at Line 12 may have turned off some 1's, but any variables added back in Line 16 must correspond to some of the 1's turned to 0's by the intersection at Line 12.

By the assumptions of the lemma, the final value of x (which is t_x) satisfies T^*, so all earlier values of x must also satisfy T^*.

We will in fact prove something slightly stronger than the statement of the lemma. Let t_1 be an initial theory term needing a minimum number of additions in order to cover T^*; that is, t_1 is an initial theory term minimizing the quantity $|T^* \backslash t|$. We will show that the number of additions charged to x, that is, the value of *min* at Line 33, will be $|T^* \backslash t_1|$. On the other hand, we will also show that the number of deletions later made to this term, will be at most $|t_0 \backslash T^*|$. These two claims together imply the lemma.

After starting with the initial value of x, some number of pivots may be found. However, at some point we start Line 9 for the final time in this iteration of the main **while** loop of MONOREVISEUPTOE. We consider only the part of the execution of MONOREVISEUPTOE from this point (after any pivots have been found using x) onward.[4]

Consider the iteration of the **for all** $t \in \varphi$ loop in which the algorithm uses the initial theory term t_0. By the assumptions of the lemma, e is large enough that at the end of this iteration, we will set x to be $x \cap t_0$ plus those variables found by the repeated calls to BINARYSEARCH. The repeated calls to BINARYSEARCH will find precisely the variables in $T^* \backslash t_0$, because any variable we found that was not in T^* would be a pivot. So from this point on, the only necessary deletions from a hypothesis term created from x are variables in $t_0 \backslash T^*$.

Consider now the iteration of the **for all** $t \in \varphi$ loop in which the algorithm uses the initial theory term t_1. (Incidentally, we do not know whether this $t = t_1$ iteration will be before or after the iteration where $t = t_0$; we are making no assumptions about the order in which initial theory terms are considered.) Again, the repeated calls to BINARYSEARCH will find precisely the variables in $T^* \backslash t_1$. Hence the value of *numAddLits* from this iteration of the **for all** $t \in \varphi$ loop will be $|T^* \backslash t_1|$. From this point on, the value of *min* must be $\leq |T^* \backslash t_1|$. In fact, from this point on, $min = |T^* \backslash t_1|$, although the equality is not needed for this proof.

Now, by definition,

$$\begin{aligned} d &= |t_0 \backslash T^*| + |T^* \backslash t_0| \\ &\geq |t_0 \backslash T^*| + |T^* \backslash t_1|. \end{aligned}$$

□

Remark. In fact, the actual quantity may be even less than $|t_0 \backslash T^*| + |T^* \backslash t_1|$, because the number of deletions needed after the term is created could be strictly less than $|t_0 \backslash T^*|$, if some other initial theory term besides t_0 happened to effectively delete some of t_0's extra variables.

Lemma 8. *A single addition of a term to the hypothesis requires* $O(m^2 e \log n)$ *membership queries and one equivalence query.*

Proof: 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 $\lceil \log_2 n \rceil$ 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.

The only equivalence query is the one made in Line 2 of MONOREVISEUPTOE to obtain the positive counterexample x. □

Next, we show a bound on the query complexity of any run of MONOREVISEUPTOE—whether it terminates by finding the target or by failing.

Lemma 9. *For any target monotone DNF, and any m-term* φ *and* e, MONOREVISEUPTOE (φ, e) *makes at most* $O(m^3 e \log n)$ *membership queries and at most* $\max(e + m, 1)$ *equivalence queries.*

Proof: By Lemma 8 each creation of a hypothesis term requires at most $O(m^2 e \log n)$ membership queries and one equivalence query. At most m hypothesis terms are created, so over the whole run of the algorithm, this consumes at most $O(m^3 e \log n)$ membership queries and at most m equivalence queries.

The other place where queries are consumed is in making deletions in the **if** branch of the main loop, Lines 3–7 of MONOREVISEUPTOE. Each execution of that branch decrements the number-of-edits-remaining parameter by at least one, so we execute that branch at most e times. Each such execution requires one equivalence query to obtain the positive counterexample x (at Line 2), and m membership queries; thus this section of the algorithm consumes at most e equivalence and me membership queries. □

Lemma 10. MONOREVISEUPTOE(φ, e) *succeeds in finding the target monotone DNF if* φ *has revision distance at most e from the target.*

Proof: The algorithm terminates only by failing or by finding the target. So, we need to show that "Failure" cannot be returned.

Let us assume that the target formula has its terms numbered to line up with an actual revision from the initial formula. That is, we assume that the target is

$$\varphi^* = T_1^* \vee \cdots \vee T_{m'}^*,$$

for some $m' \leq m$, where T_i^* can be obtained from initial formula term t_i with e_i revisions, and the total revision distance is $e_1 + \cdots + e_{m'} + (m - m')$. The quantity $(m - m')$ represents the one edit needed to fix to $\mathbf{0}$ one variable of each of the initial terms $t_{m'+1}, \ldots, t_m$—that is, to delete those terms altogether.

We argue by induction on the number of iterations of the outer **while** loop of MONOREVISEUPTOE. Assume that after a certain number of iterations we have hypothesis $h = s_1 \vee s_2 \vee \cdots \vee s_\ell$, where, for simplicity of notation, we assume the terms are ordered such that hypothesis term s_i covers target term T_i^*. Let the current value of variable e be e_0 (i.e., there are still e_0 edits remaining), and let $d_i = |s_i \setminus T_i^*|$. We claim that MONOREVISEUPTOE does not fail and that

$$e_0 - \sum_{i=1}^{\ell} d_i \geq \sum_{i=\ell+1}^{m'} e_i. \tag{1}$$

The proof is by induction on the number of iterations. The induction step follows directly from the algorithm for deletion edits, and from Lemma 7 for edits that create terms. □

Theorem 11. *There is a revision algorithm for monotone DNF formulas in the general model of revisions, using $O(m^3 e \log n)$ queries for monotone m-term DNF formulas in a universe of n variables, where e is the revision distance between the initial and the target formulas.*

Proof: Let φ be the initial formula. 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 "Yes!" we apply MONOREVISEUPTOE(φ, e) for repeatedly doubled values of e until it produces a "Yes!"

Now the result follows from Lemmas 9 and 10. □

Algorithm 4 MONODELETIONS(φ). Revises φ, a set of monotone terms using only deletion edits.

```
h = ∅ {the initial hypothesis}
while (x = EQ(h)) ≠ "Yes!" do
  if MQ(x ∩ t) = 1 for some t ∈ h then {delete vars from hyp. terms}
    for all t ∈ h for which MQ(x ∩ t) == 1 do
      t = t ∩ x
    end for
  else [find a new term to add to the hypothesis]
    for all t ∈ φ do
      new = t ∩ x
      if MQ(new) == 1 then
        x = new
      end if
    end for
    h = h ∨ x
  end if
end while
```

4.3. Deletions-only case

In this subsection we briefly sketch out the special case of the previous algorithm for the deletions-only model of revisions. Note that BINARYSEARCH is not needed in this model. Furthermore, we do not need to keep track of the number of edits assumed to be needed.

The pseudocode given for MONODELETIONS is a strict subset of the code for MONOREVISEUPTOE. The proof of correctness for this algorithm is similar to that for MONOREVISEUPTOE and thus is not included here.

Making one deletion from an existing hypothesis term requires membership queries related to each hypothesis term; each new term requires membership queries related to each hypothesis term and then to each initial theory term. Each deletion (or set of deletions) and each addition of a new hypothesis term requires one equivalence query.

Theorem 12. *There is a revision algorithm for monotone DNF formulas in the deletions-only model, using $O(em + m^2)$ membership queries and at most $(m + e)$ equivalence queries for monotone m-term DNF formulas over n variables, where e is the revision distance between the initial and target formulas.*

5. Revising unate DNF

In this section, we present an algorithm that can revise a two-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 added to the other term if it occurs in only one term, or both, but we cannot delete x_i and add $\bar{x}_i$.

5.1. Description of algorithm

As was the case for our monotone DNF algorithm in Section 4, the main routine for revising unate two-term DNF takes a maximum revision distance e as a parameter, and finds the target if it is within revision distance e of the initial formula. This main algorithm, UNATEREVISEUPTOE, should be repeatedly called with $e = 1, 2, 4, 8, 16, \ldots$ until it finds the target.

In Algorithm UNATEREVISEUPTOE (Algorithm 5), we begin the revisions with one positive example x_0, which must satisfy (at least) one of the two target terms, but we do not know which one. Therefore we consider the possibility of revising each initial theory term sequentially. We begin by using x_0 to edit the first term of the initial theory. If that beginning does not lead to a correct revision within e edits, then we restart the revision process, this time using x_0 to edit the second term of the initial theory.

In most cases, when we construct our initial one-term hypothesis using x_0, we immediately try to find all literals that must be added to the initial term to obtain the target term. Instance x_0 allows us to determine the orientation (positive or negative) of any literals that need to be added, given the assumption that x_0 satisfies the corresponding target term. Nevertheless, we can sometimes find only some of the literals that need to be added, and sometimes we simply cannot tell whether any variables need to be added. In such cases, the algorithm uses later negative counterexamples to find additional literals.

Algorithm 5 UNATEREVISEUPTOE(t_1, t_2, e). Note that an iteration of the main **for** loop immediately terminates if FILLTERM or ADDLITERAL returns "Failure" *except* when return value is explicitly tested at Line 24.

```
 1: Let x_0 be positive instance (from EQ(FALSE))
 2: for k = 1, 2
 3:   if MQ(x_0 ⨅ t_k) == 0 then
 4:     h = FILLTERM(t_k ∩ x_0, x_0 ⨅ t_k, x_0, e)
 5:     if h == PivotException(l) then
 6:       x_0 = x_0 with l turned off
 7:       Restart at Line 2 with this x_0
 8:     end if
 9:   else if MQ(x_0 ⨆ t_k) == 0 then
10:     x_0 = x_0 ⨅ t_k
11:     h = FILLTERM(t_k̄ ∩ x_0, x_0 ⨅ t_k̄, x_0, e)
12:     if h == PivotException(l) then
13:       x_0 = x_0 with l turned off
14:       Restart at Line 2 with this x_0
15:     end if
16:     Note: Line 11's h is term derived from t_k̄; so swap t_k, t_k̄ at Lines 24 and 30
17:   else
18:     h = t_k ∩ x_0
19:   end if
20:   {We now have a one term hypothesis}
21:   while (y = EQ(h)) ≠ "Yes!" and e > 0 do
22:     if h(y) == 0 then {y is a positive counterexample}
23:       e_save = e
24:       if FILLTERM(t_k̄ ∩ y, y ⨅ t_k̄, y, e) returns a term t and then
          EQ(REVISE2TERMS(h, t_k, t, t_k̄, x_0, e)) =="Yes!" then
25:         return correct theory found by REVISE2TERMS
26:       end if
27:       e = e_save − |h\(h ∩ y)| {Use y to make deletions}
28:       h = h ∩ y
29:     else [y is negative counterexample]
30:       h = h ∧ ADDLITERAL(x_0, y, t_k)
31:       e = e − 1
32:     end if
33:   end while
34: end for
```

While we have a one-term hypothesis, any negative counterexample must be used to add a literal to that hypothesis term. A positive counterexample y might indicate either that we need to delete literals from the first hypothesis term or that we need to initialize a second hypothesis term. When presented with such a choice, we first try to initialize a second term. If this leads to too many edits, we backtrack and use y to delete literals from the first term.

When we do initialize the second term, again we immediately look for all necessary additions of literals to the initial term to cover the corresponding target term. It turns out that for the second term we can always tell whether additional variables are needed, and if so find all of them.

Algorithm UNATEREVISEUPTOE uses several subroutines. The subroutine BINARYSEARCH is the same as in the monotone case, although now there may be negative literals. We also break out the process of repeatedly using BINARYSEARCH to add all the necessary literals to a hypothesis term. This subroutine is called FILLTERM.

The handling of a two-term hypothesis is passed on to the subroutine REVISE2TERMS. When the input is a hypothesis that requires at most e edits, including deletions to either term and additions to the first term, then REVISE2TERMS successfully completes the revisions. It uses negative counterexamples to add literals to the first term, and positive counterexamples to delete literals as needed.

The first half of the pseudocode for UNATEREVISEUPTOE (Lines 3–19) covers the construction of the initial one-term hypothesis. The next chunk of code is concerned with preparing the hypothesis for a call to REVISE2TERMS. If that call fails, then the assumption that a positive counterexample y should have created a second term was incorrect. The algorithm therefore returns to that checkpoint and uses that counterexample y to delete literals from the one-term hypothesis.

If the entire algorithm UNATEREVISEUPTOE(t_1, t_2, e) fails, then the edit distance between initial theory $t_1 \vee t_2$ and the target was greater than e. The higher level routine then doubles e and calls UNATEREVISEUPTOE(t_1, t_2, e) again until it succeeds.

In the remainder of this subsection, we discuss each of the subroutines and also the main algorithm. We first recall the algorithm BINARYSEARCH from the monotone case, and describe the subroutine FILLTERM that is used to repeatedly call BINARYSEARCH to find missing literals, as these are utility subroutines that are frequently used. We next discuss the main algorithm, since most of its code is concerned with initializing a one-term hypothesis and refining that one-term hypothesis. Finally, we discuss two subroutines used closer to the end of a run of UNATEREVISEUPTOE: ADDLITERAL and REVISE2TERMS.

The next subsection, Section 5.2, contains a detailed example. It may be helpful to skip back and forth between that example and the overview of the algorithm that precedes it. Finally, we prove the correctness and query complexity bound in Section 5.3.

We conclude this introductory material with some definitions we will use throughout the remainder of this section.

Definition. A term t of a hypothesis DNF is *full with respect to target term* T^* if t's literals are a superset of T^*'s literals.

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.

Definition. The variables that do not occur in the initial theory are the *outside variables*, and those that do occur in the initial theory are the *inside variables*.

5.1.1. BinarySearch and FillTerm. The subroutine BINARYSEARCH, Algorithm 3, described in Section 4 on revising monotone DNF, actually works for unate as well as for monotone DNFs. The precise formal guarantee about its behavior is given in Proposition 4.

Here is the general idea of how to use BINARYSEARCH to find outside literals of a new term in the unate case. Assume x_0 satisfies target term T^* that should be derived from editing initial theory term t, and that $\mathrm{MQ}(x_0 \underline{\cap} t) = 0$. Clearly there must be some literal(s) not in t that are in $x_0 \otimes (x_0 \underline{\cap} t)$ and that should be added to t. In the monotone case, repeated calls to BINARYSEARCH normally find all such literals.

In the unate case, we may find only some but not all of the literals that need to be added, because we no longer know how to ask a membership query on an instance with "literals in t set as in $x_0 \underline{\cap} t$, the necessary added literals we found set to on, and *all other literals set to off*." It will turn out that the first hypothesis term we construct will sometimes not be full, but that when UNATEREVISEUPTOE initializes the second term of the hypothesis, it will be full.

Another complication arises because BINARYSEARCH might return a literal that does not belong to T^* but only to the other target term. We can detect this situation: it occurs when BINARYSEARCH returns a literal l such that $\mathrm{MQ}(x_0$ with l set to off$) = 1$. We call such an l a *pivot*, and we restart the algorithm UNATEREVISEUPTOE with x_0 modified to have l set to off and with the orientation of l now known. Our test for a pivot will also say we have found a pivot whenever the initial instance x_0 satisfies both target terms and BINARYSEARCH finds an outside literal in only one target term; we also consider that to be a pivot.

In the unate case, the action of repeatedly calling BINARYSEARCH to find as many literals as possible, and checking each new literal that is found to see if it is a pivot, is broken out into the subroutine FILLTERM. (The analogous pseudocode for the monotone case consists of Lines 14–22 of MONOREVISEUPTOE.) In the pseudocode, FILLTERM returns the special value PivotException if a pivot is found. Also, FILLTERM keeps track of the number of additions of literals it is proposing; if that number gets too large, then FILLTERM returns the value "Failure".

We will show in Section 5.3 that in one run of UNATEREVISEUPTOE FILLTERM can return the value PivotException at most one time in calls made to initialize the first term (Lemma 14).

5.1.2. Details of main algorithm. The procedure UNATEREVISEUPTOE, specified in Algorithm 5, takes e and the initial formula $\varphi = t_1 \vee t_2$ as inputs, and begins with an equivalence query to FALSE. If that is not the target formula, then we get a positive counterexample, x_0, which is used to create a one-term hypothesis.

We begin with the assumption that x_0 satisfies a target term that "should" be derived by editing initial theory term t_1. That is, the revision distance is minimized by editing t_1 rather than t_2 to get this target term, or more precisely, the revision distance

$$e = |t_1 \otimes T_1^*| + |t_2 \otimes T_2^*| \leq |t_1 \otimes T_2^*| + |t_2 \otimes T_1^*|. \tag{2}$$

If this leads to a failure to revise the initial theory within e revisions, then the algorithm restarts using the same x_0 with the assumption that x_0 satisfies a target term that should

Algorithm 6 FILLTERM(t, v, w, e). Finds necessary additions to hypothesis term t from $v \otimes w$ to cover a target term, if this can be done with at most e additions. Returns either PivotException, "Failure", or a term. (Note that e is passed by *reference*; the caller *does* see changes made to e.)

Require: $t(v) = t(w) = 1$ and $\mathrm{MQ}(w) = 1$
1: $e_0 = e$
2: **while** $\mathrm{MQ}(v) == 0$ **and** $e > 0$ **do**
3: $l = \mathrm{BINARYSEARCH}(v, w)$
4: **if** $\mathrm{MQ}(w$ with position l flipped$) == 1$ **then**
5: $e = e_0$
6: Add l to the set of known literal orientations
7: **return** PivotException(l)
8: **end if**
9: $v = v$ with l flipped to agree with w
10: $t = t \wedge l$
11: $e = e - 1$
12: **end while**
13: **if** $\mathrm{MQ}(v) == 1$ **then** {v covers term}
14: **return** t
15: **else** {$e \leq 0$, so all edits already used}
16: $e = e_0$
17: **return** "Failure"
18: **end if**

be derived by editing initial theory term t_2. This is the purpose of the **for** $k = 1, 2$ control structure in the pseudocode at Line 2.

In the entire remainder of Subsection 5.1, our discussion will assume that x_0 satisfies T_1^*, where the target formula is $T_1^* \vee T_2^*$, with the terms labeled to minimize the revision distance, as per (2).

The algorithm starts by asking two membership queries, $\mathrm{MQ}(x_0 \overline{\cap} t_1)$ and $\mathrm{MQ}(x_0 \underline{\cap} t_1)$, to try to get some idea of whether the target term T_1^* contains any literals not in t_1. Given that x_0 satisfies target term T_1^*, we know that T_1^* can contain only those literals of t_1 that are set to on in x_0 and/or literals not in t_1.

There are three main cases depending on the answers to the two membership queries. We cover them here in (what seems to us to be) increasing order of complexity, not in the order of the pseudocode.

One case is that both membership queries return 1, in which case Line 18 is executed. (This corresponds to Case I in the proof of Lemma 23.) Our initial one-term hypothesis is $h = t_1 \cap x_0$. In the proof of correctness (Lemma 23) there are two subcases. In the first case, h is full; in the second case, h is not full. We do not at this point know which case we are in.

If, instead, $\mathrm{MQ}(x_0 \underline{\cap} t_1) = 0$, then we use the subroutine FILLTERM to repeatedly perform a binary search from $x_0 \underline{\cap} t_1$ to x_0, and our initial one-term hypothesis is $h = t_1 \cap x_0$ plus those additional variables found by FILLTERM

We point out here two unusual ways that the call to FILLTERM might end: by returning "Failure" or PivotException. FILLTERM will return "Failure" if it needs to find more than e literals to be added. This indicates that either our assumption that x_0 satisfies T_1^* is wrong, or that our assumption that the revision distance between the initial and target formulas is at most e is wrong. If FILLTERM fails, then the **for** $k = 1$ iteration is terminated.

PivotException(l), where l must be some literal that is set to on in x_0, indicates one of two things. One possibility is that x_0 satisfied both target terms, and l is contained in exactly one of those two terms. The other possibility is that l is contained in T_2^* and not in T_1^*. Furthermore, at some intermediate point in the binary search from $x_0 \underline{\cap} t_1$ to x_0, an instance was considered that satisfied T_2^*.

In either case, the whole UNATEREVISEUPTOE algorithm is restarted with x_0 modified by setting l to off. Since l's orientation is now known, a PivotException cannot recur, as x_0, $x_0 \underline{\cap} t_1$, and $x_0 \underline{\cap} t_2$ will all have l set to off.

The third and final starting case is that $\mathrm{MQ}(x_0 \overline{\cap} t_1) = 0$ and $\mathrm{MQ}(x_0 \underline{\cap} t_1) = 1$. By assumption x_0 satisfies T_1^*. The only difference between instances x_0 and $x_0 \overline{\cap} t_1$ is on the variables in $t_1 \backslash t_2$. It must be that T_1^* contains some variables from $t_2 \backslash t_1$, since $\mathrm{MQ}(x_0 \overline{\cap} t_1) = 0$. Thus, we can be certain that $x' = x_0 \underline{\cap} t_1$ satisfies the *other* target term, T_2^*.

We start our hypothesis term with $t_2 \cap x'$, and use FILLTERM to find additional variables outside of t_2 that we may need to add, starting at $(x' \underline{\cap} t_2)$ and going up to the known positive instance x'. We must remember that our hypothesis term has actually been derived from t_2.

This completes the discussion of Lines 3–19 of UNATEREVISEUPTOE. In summary, if we are in the proper iteration of the **for** loop, say k , then we now have a one-term hypothesis that includes all literals of T_k^* that are in the corresponding initial theory term t_k. Any literals in the hypothesis term that are not in t_k *are* in T_k^* and will never need to be deleted. We may still need to delete from the hypothesis some literals of t_k and/or add some more literals not in t_k.

We next describe how to handle positive counterexamples. Unlike the monotone case, once we have a one-term hypothesis, we often do not know whether subsequent positive counterexamples should be used to generate a new hypothesis term, or to delete variables from the existing hypothesis term. So, for each new positive counterexample y, we first assume that y cannot be used to edit the existing term, and therefore that y should start a new term. If subsequent editing does not produce a correct hypothesis within e edits, then our assumption must have been wrong, and we use y to make deletions from the first term. Our complexity analysis will show that this is not dreadful; in fact this will add at most $O(e \log n)$ queries for making one deletion from the first hypothesis term.

We first try initializing a second hypothesis term to $t_2 \cap y$ plus any literals found by FILLTERM. We will show in Lemma 23 that if y cannot be used to make deletion edits to the existing first hypothesis term, then this second hypothesis term is full when it is initialized.

If y can, in fact, be used to edit the first hypothesis term h, then after at most e edits after using y to create a second term, (either we have succeeded in finding the target, or) REVISE2TERMS will fail. If it fails, UNATEREVISEUPTOE uses y to edit the existing hypothesis term h to $h \cap y$. This deleting literals from the single term h will in this case be the only result of this iteration of the **while** loop, Lines 21–33.

We have now described how the algorithm UNATEREVISEUPTOE creates a two-term hypothesis, except for describing how negative counterexamples to a one-term hypothesis are processed. Negative counterexamples are handled by ADDLITERAL; we describe it next.

Once a two-term hypothesis has been created, any negative counterexamples are used to add variables to the first term, and positive counterexamples are used to delete variables from terms in a manner similar to the deletions-only case. Formally, that work is handled by subroutine REVISE2TERMS, which we will describe after describing ADDLITERAL.

5.1.3. Negative counterexamples and algorithm AddLiteral. Now we describe what to do with a negative counterexample to a one-term hypothesis. Negative counterexamples to a one-term hypothesis can occur either because FILLTERM found only some but not all of the literals in $T_1^* \backslash t_1$, or because we are in the case where $\mathrm{MQ}(x_0 \underline{\cap} t_1) = \mathrm{MQ}(x_0 \overline{\cap} t_1) = 1$.

Given x_0 and such a negative counterexample y, we do a binary search on the variables of $(y \otimes x_0) \backslash t_1$. Note that this may find only one necessary addition, not all of them; one variable set to off is sufficient to make y a negative instance. There may be other necessary additions to the hypothesis term that are set to on in both y and x_0, and these will not be found at this point. In fact, for simplicity of exposition and analysis, we will find only one necessary addition using y and then stop. As this work of finding one variable may also need to be done inside subroutine REVISE2TERMS, we break it out into its own subroutine, ADDLITERAL, Algorithm 7.

Notice that ADDLITERAL's search for a literal to add is restricted to literals outside of initial theory term t_1. By assumption, the one-term hypothesis was initialized containing all literals in $t_1 \cap T_1^*$, and no literals in T_1^* are ever deleted, so no literals in t_1 ever need to be added.

In general ADDLITERAL might find a literal from the "wrong" target term or a pivot. This issue does not arise in our algorithm, because of the special nature of the inputs we give ADDLITERAL. (The necessary conditions are given in Lemma 18, and in the proof of Lemma 23 we argue that the inputs to ADDLITERAL meet the necessary conditions.)

5.1.4. Algorithm Revise2Terms. As its name suggests, the procedure REVISE2TERMS takes a two-term hypothesis $T_1 \vee T_2$ as input. It is loosely related to a procedure called REFINEDOWN that was was introduced in our earlier work (Sloan & Turán, 1999). Like the

---

Algorithm 7 ADDLITERAL(y, x_0, t). Inputs: negative instance y, positive instance x_0, and initial theory term t. Finds a literal l not in t that agrees with x_0 and disagrees with y or returns "Failure." (Inputs to this routine use call-by-value; the caller does not see changes to the input.)

---

for all variables v such that either $v, \bar{v} \in t$ and $y[v] \neq x_0[v]$ **do**
  Turn off $y[v]$ and $x_0[v]$ {Make y, x_0 agree on vars of t}
end for
if $\mathrm{MQ}(x_0) == 0$ **then**
  return "Failure"
end if
return BINARYSEARCH(y, x_0)

Algorithm 8 REVISE2TERMS($T_1, t_1, T_2, t_2, x_0, e$). Edits the two-term hypothesis $T_1 \vee T_2$, but never makes more than e edits. The positive instance x_0 was used to create T_1 from t_1. Returns "Failure" if any call to ADDLITERAL returns "Failure." Terminates either by returning "Failure", or with correct target formula. (Uses call-by-value)

```
1: h = T1 ∨ T2.
2: while (x = EQ(h)) ≠ "Yes!" do
3:   if e ≤ 0 then {Used up all allowable edits}
4:     return "Failure"
5:   end if
6:   if h(x) == 0 then {x is positive counterexample}
7:     if MQ(x ∩̄ T2) == 1 then
8:       e = e − |T2\(T2 ∩ x)|
9:       T2 = T2 ∩ x
10:    else
11:      e = e − |T1\(T1 ∩ x)|
12:      T1 = T1 ∩ x
13:    end if
14:  else {x is a negative counterexample}
15:    if x satisfies T2 then
16:      return "Failure"
17:    end if
18:    T1 = T1 ∧ ADDLITERAL(x, x0)
19:    e = e − 1
20:  end if
21: end while
```

procedure from our earlier work, REVISE2TERMS is used to delete unnecessary variables from hypothesis terms, but unlike our earlier work, REVISE2TERMS may also add literals to its first hypothesis term.

As we will discuss in Section 5.3, the second hypothesis term T_2 should be full. (By "should" we mean that if T_2 is not full, then either the first term T_1 of the hypothesis was derived from the wrong branch of the main **for** loop of UNATEREVISEUPTOE or the second term T_2 was derived from a counterexample that could have been used to make deletion edits to term T_1 instead of starting a second term.) If T_2 is full, then any instance satisfying T_2 must be positive. Therefore, REVISE2TERMS returns "Failure" if it receives a negative counterexample satisfying T_2.

Furthermore, the reason that the second term "should" be created full (the nature of the counterexample used to create it) also guarantees that there is a literal l that is in the first but not the second target term such that l is in hypothesis term T_1 but not in hypothesis term T_2. It turns out that this allows REVISE2TERMS to handle positive counterexamples. For each positive counterexample, REVISE2TERMS may safely first check whether it can be used to make a deletion from T_2; if not, it is used to make deletions from T_1. This is explained in detail in the proof of Lemma 20.

5.2. *An example run*

We provide here an example of Algorithm 5, UNATEREVISEUPTOE. Suppose that the initial theory is

$$\varphi = t_1 \vee t_2 = (x_1 x_4 x_6) \vee (x_2 x_3 x_6),$$

and the correct target formula is

$$\varphi^* = (x_2 x_4 \bar{x}_5) \vee (x_1 x_2 x_3).$$

Consider a call of UNATEREVISEUPTOE(t_1, t_2, e_{start}), for any $e_{\text{start}} \geq 6$, which is the revision distance from φ to φ^*.

Let $x_0 = 111100$ be the initial positive counterexample. Consider the $k = 1$ iteration of the main **for** loop, which starts at Line 2. Since MQ($x_0 \cap t_1$) = MQ($111100 \cap x_1 x_4 x_6$) = MQ(100110) = 0, the **if** at Line 3 directs the algorithm into Lines 4–8, and we call FILLTERM to see what additional literals must be added to the literals $x_1 x_4$ that are in t_1 and set to on in x_0. That is, we call FILLTERM($t_1 \cap x_0$, $x_0 \cap t_1$, x_0, e_{start}), or, evaluating the parameters, FILLTERM($x_1 x_4$, 100110, 111100, e_{start}).

Now FILLTERM calls BINARYSEARCH(100110, 111100) to find a literal from $100110 \otimes 111100$ that is in the target formula. Say BINARYSEARCH returns x_3. This could happen as follows. In the first iteration of the **while** loop that starts at Line 3 of BINARYSEARCH, *mid* could be set to 111110 (i.e., the choice of d_1 is $\{x_2, x_3\}$), and when MQ(*mid*) = 1, *pos* is replaced by that value of *mid*, and the second iteration of the **while** loop sets *mid* to 110110. Now MQ(*mid*) = 0, so in this iteration *neg* is replaced by *mid*. Now $neg \otimes pos = x_3$, and so BINARYSEARCH returns x_3.

Now FILLTERM determines that MQ(111100 with position x_3 flipped) = 1 (at Line 4), and FILLTERM returns PivotException(x_3). This causes UNATEREVISEUPTOE to restart at Line 2, with x_0 being modified by turning off x_3, so in the restarted algorithm, $x_0 = 110100$. Incidentally, turning off the pivot x_3 has given us an x_0 that satisfies only one term of the target formula. Now the $k = 1$ iteration of the **for** loop beginning at Line 2 finds MQ($x_0 \cap t_1$) = MQ($110100 \cap x_1 x_4 x_6$) = MQ(100110) = 0, as before, so we again go to call FILLTERM from Line 4 of UNATEREVISEUPTOE. This time the x_0 parameter is different; the call is to FILLTERM($x_1 x_4$, 100110, 110100, e_{start}). In the first iteration of its main **while** loop, Algorithm FILLTERM calls BINARYSEARCH(100110, 110100). Say BINARYSEARCH returns $\bar{x}_5$. Now FILLTERM makes the query MQ(100100) = 0, so FILLTERM calls BINARYSEARCH again, which this time would return x_2. Now MQ(110100) = 1, so FILLTERM returns $x_1 x_2 x_4 \bar{x}_5$. Note that FILLTERM also decremented the "number of edits" parameter e, so back in UNATEREVISEUPTOE, $e = e_{\text{start}} - 2$.

UNATEREVISEUPTOE's hypothesis now is $h = x_1 x_2 x_4 \bar{x}_5$, and execution moves to the **while** loop at Line 22. Say the counterexample returned from the equivalence query is $y = 011101$. We first try to use y to initialize a second hypothesis term derived from t_2 by calling FILLTERM($t_2 \cap y$, $y \cap t_2$, y, $e_{\text{start}} - 2$) = FILLTERM($x_2 x_3 x_6 \cap 011101$, $011101 \cap x_2 x_3 x_6$, 011101, $e_{\text{start}} - 2$) = FILLTERM($x_2 x_3 x_6$, 011011, 011101, $e_{\text{start}} - 2$). Now, inside FILLTERM, we first find that MQ(011011) = 0, so we begin making calls to BINARYSEARCH looking for literals from $011011 \otimes 011101 = \{x_4, \bar{x}_5\}$ to add to $x_2 x_3 x_6$. In fact, it will turn

out that both x_4 and $\bar{x}_5$ need to be added and neither is a pivot, so FILLTERM will return the hypothesis terms $x_2x_3x_4\bar{x}_5x_6$, and will also decrement the number-of-edits-remaining parameter by another 2, so now back in UNATEREVISEUPTOE the number-of-edits-remaining parameter will have the value $e_{\text{start}} - 4$, and the hypothesis will contain the two (ordered) terms $x_1x_2x_4\bar{x}_5$ and $x_2x_3x_4\bar{x}_5x_6$. Notice that intuitively our attempt to start a second term using y is not working out—we have two hypothesis terms that both cover the same one target term. As we will see as we continue to trace through the algorithm, we will eventually realize that y should be used to edit the first term, not to start a second term.

Right now, however, UNATEREVISEUPTOE is at Line 24 and FILLTERM returned a term. So we call REVISE2TERMS with parameters $T_1 = x_1x_2x_4\bar{x}_5$ and $T_2 = x_2x_3x_4\bar{x}_5x_6$; t_1 and t_2 coming from the initial formula, x_0, and $e = e_{\text{start}} - 4$. REVISE2TERMS begins by obtaining a counterexample x to $\text{EQ}(T_1 \vee T_2)$. Say $x = 111010$. Now $\text{MQ}(x \,\overline{\cap}\, T_2) = \text{MQ}(111010 \,\overline{\cap}\, x_2x_3x_4\bar{x}_5x_6) = \text{MQ}(011010) = 0$. Therefore, we set $T_1 = T_1 \cap x = x_1x_2x_4\bar{x}_5 \cap 111010 = x_1x_2$, and decrement e by 2, so now $e = e_{\text{start}} - 6$. Now REVISE2TERMS again asks an equivalence query; this time the query $x = \text{EQ}(x_1x_2 \vee x_2x_3x_4\bar{x}_5x_6)$. Say $x = 110010$.

Now if the initial value of $e_{\text{start}} = 6$, the test for $e \leq 0$ at Line 3 of REVISE2TERMS causes it to return "Failure" immediately. Intuitively, REVISE2TERMS has found that there is no revision of the initial hypothesis passed to it that required only 2 revisions (beyond the ones made before REVISE2TERMS was called).

Suppose instead that the initial value of e_{start} was 8, so that at Line 3 we find $e = 8 - 6 = 2 \neq 0$. In this case, we find that x is a negative counterexample that satisfies the first term of the current hypothesis, so we look for an additional literal to add to the first hypothesis term. We call $\text{ADDLITERAL}(x, x_0, t_1) = \text{ADDLITERAL}(110010, 111100, x_1x_4x_6)$.

ADDLITERAL will try to find some literal not in t_1 on which x_0 and the negative counterexample 110010 disagree. Intuitively, this literal is on in x_0 and is (at least one of) the missing literal(s) in the hypothesis term satisfied by the negative counterexample. To ensure that a binary search does not find a literal in t_1, ADDLITERAL turns to off any such disagreeing literals before calling BINARYSEARCH. In this case it thus modifies x_0 to be 110000, and since $\text{MQ}(110000) = 0$, ADDLITERAL returns "Failure." Then REVISE2TERMS returns "Failure" when ADDLITERAL RETURNS "Failure".

Once REVISE2TERMS returns "Failure", Algorithm UNATEREVISEUPTOE uses its positive counterexample to perform deletions from the single term hypothesis h. In this case, the update $h = h \cap y$ is $h = x_1x_2x_4\bar{x}_5 \cap 011101 = x_2x_4\bar{x}_5$. The number-of-edits-remaining parameter e is updated to be $e_{\text{start}} - 3$; this value is the previously saved value of $e_{\text{start}} - 2$, less the the number of literals deleted from h when h is set to $h \cap y$, which in this case is 1.

We begin the **while** loop at Line 21 again by obtaining a counterexample y from the query $\text{EQ}(h) = \text{EQ}(x_2x_4\bar{x}_5)$. Suppose that $y = 111111$. Now $\text{FILLTERM}(t_2 \cap y, y \,\underline{\cap}\, t_2, y, e)$ is called at Line 24 to try to initialize a second hypothesis term. The parameters evaluate to $\text{FILLTERM}(x_2x_3x_6 \cap 111111, 111111 \,\underline{\cap}\, x_2x_3x_6, 111111, e_{\text{start}} - 3) = \text{FILLTERM}(x_2x_3x_6, 011011, 111111, e_{\text{start}} - 3)$. In this case, FILLTERM will make one call to BINARYSEARCH and find the literal x_1 to add, and thus return the term $x_1x_2x_3x_6$; also, FILLTERM decrements the number-of-edits-remaining parameter e by 1, so now $e = e_{\text{start}} - 4$.

Now at Line 24 $\text{REVISE2TERMS}(x_2x_4\bar{x}_5, t_1, x_1x_2x_3x_6, t_2, x_0, e)$ is called, where t_1 and t_2 are the two initial theory terms. This time, since $x_2x_4\bar{x}_5$ is one term of the target, and

the second term of the target is a subset of the literals $x_1x_2x_3x_6$, REVISE2TERMS must get counterexamples that allow the second hypothesis term to be edited to the correct target term. For instance, the counterexample x to the equivalence query $\mathrm{EQ}(x_2x_4\bar{x}_5 \vee x_1x_2x_3x_6)$ made in Line 2 of REVISE2TERMS might be $x = 111000$. Then at Line 7 $\mathrm{MQ}(x \cap x_1x_2x_3x_6) = \mathrm{MQ}(111010) = 1$, so the second hypothesis term is updated to $x_1x_2x_3x_6 \cap 111000 = x_1x_2x_3$, and the number-of-edits-remaining parameter is decremented by 1, the number of deletions just made, so it now is $e_{\text{start}} - 6$.

Next, finally, the equivalence query $\mathrm{EQ}(x_2x_4\bar{x}_5 \vee x_1x_2x_3)$ returns "Yes!", because we have found the target formula.

5.3. Correctness and query complexity

We will prove the correctness and query complexity with a series of technical lemmas. The lemmas are presented in roughly the same order as subroutines were presented in Section 5.1.

Our overall strategy is to show that if e is at least as large as the revision distance between $t_1 \vee t_2$ and the target formula, then UNATEREVISEUPTOE(t_1, t_2, e) succeeds in finding the target formula in time polynomial in e and $\log n$. In particular, we will show that when constructing the first term, the subroutine FILLTERM can return the value PivotException at most once, and that if PivotException is *not* returned by FILLTERM then at least one of the two iterations of the main **for** k loop of UNATEREVISEUPTOE succeeds. This will lead to a worst-case query complexity of $3 \cdot$ (the query complexity of the main **for** loop) plus the query complexity of FILLTERM, since in the worst case we have an unsuccessful iteration of the main **for** loop followed by an iteration that consists only of FILLTERM returning PivotException, followed by two more iterations of the main **for** loop, the first failing to find the target and the second succeeding.

We begin with a series of propositions and lemmas related to FILLTERM. First we analyze the query complexity of FILLTERM (Proposition 13), and next we show why PivotException can be returned only once in creating the initial term (Lemma 14).

Proposition 13. *A call to* FILLTERM$(\cdot, \cdot, \cdot, e)$ *uses* $O(e \log n)$ *membership queries and no equivalence queries.*

Proof: FILLTERM makes at most e calls to BINARYSEARCH, which uses $O(\log n)$ membership queries per call. Additionally FILLTERM may make up to $2e + 1$ other membership queries. □

Lemma 14. *In a run of* UNATEREVISEUPTOE *the value PivotException is returned at most once from calls to* FILLTERM *from lines* 4 *and* 11 *of* UNATEREVISEUPTOE.

Proof: Consider the value of the variable x_0 immediately before the call to FILLTERM that returns the value PivotException(l). Let the target formula be $T_1^* \vee T_2^*$. It must be that x_0 satisfies (at least) one target term, say T_1^*, but l is in only the other target term, T_2^*. This is so because we know by Proposition 4 that BINARYSEARCH, which returned l, finds only literals in the target, and we also know that $\mathrm{MQ}(x_0$ with l set to off$) = 1$.

After PivotException(l) is returned, x_0 will always have l set to off, and thus x_0 will satisfy only T_1^*. For any t', instance $x_0 \underline{\cap} t'$ must also have l set to off, since the orientation of l is now known. Now for all subsequent calls to BINARYSEARCH by FILLTERM($t' \cap x_0$, $x_0 \underline{\cap} t'$, x_0, e), literal l will always be turned off, because for all variables on which FILLTERM's two input instances agree, all instances for which BINARYSEARCH asks membership queries also have that same agreed upon value.

Therefore, no call to BINARYSEARCH by FILLTERM(t', $x_0 \underline{\cap} t'$, x_0, e) can return any literal not in T_1^*, and thus no further pivots will be found. □

Next, Lemma 15 says that FILLTERM does not add extraneous literals to t. Immediately after Lemma 15, we give a lemma concerning the specific case of using FILLTERM to initialize the second term of the hypothesis at Line 24 of UNATEREVISEUPTOE.

Lemma 15. *Let y be a positive instance that satisfies term T^* of the target formula. Consider a call to* FILLTERM(t, x, y, e) *such that instances x and y both satisfy term t, and $|T^* \backslash t| \leq e$. If* FILLTERM *does not return PivotException, then it returns the term ($t \wedge$some or all literals of $T^* \backslash t$). This uses $O(\log n) \cdot$ (number of literals it added) membership queries.*

Proof: First of all, if $T^* \subseteq t$, then since x satisfies t, we will have MQ(x) = 1 in Line 1, and correctly return the unmodified term t.

If y satisfies T^* and PivotException is not returned, then each call to BINARYSEARCH finds a necessary addition to the term t using at most $O(\log n)$ membership queries. □

We will show in the main technical lemma, Lemma 23, that the conditions of Lemma 16 will hold when FILLTERM is used to initialize a second hypothesis term, when the first hypothesis was not derived from initial theory term t.

Lemma 16. *Let y be a positive instance of the unate two-term target DNF $T_a^* \vee T_b^*$. If there is a literal l of known orientation that is in T_a^* but is set to off in y, then, for any nonempty term t, if $e > |T_b^* - (t \cap y)|$, then* FILLTERM($t \cap y$, $y \underline{\cap} t$, y, e) *returns the term $(t \cap y) \wedge (T_b^* \backslash (t \cap y))$, which is full with respect to T_b^*, and does not contain l.*

The total number of queries used to initialize the term is at most $O(\log n)$ times the number of literals in the new hypothesis term that are not in t. If the new hypothesis term is a subset of t, then the total number of queries is $O(1)$.

Proof: Notice that y must satisfy T_b^* and must not satisfy T_a^*. Since l's orientation is known, l will be off in both $y \underline{\cap} t$ and y, and therefore in all intermediate instances for which BINARYSEARCH asks membership queries. Therefore, a PivotException cannot occur, and FILLTERM cannot terminate before it has found all literals in T_b^*.

Since l is off in y, it cannot be in the returned hypothesis.

The query complexity follows from Lemma 15. □

The next two lemmas discuss the complexity and correctness of ADDLITERAL.

Proposition 17. ADDLITERAL *uses no equivalence queries and at most* $O(\log n)$ *membership queries.*

Proof: ADDLITERAL makes one membership query before calling BINARYSEARCH, and BINARYSEARCH makes no equivalence queries and $O(\log n)$ membership queries. □

We will later show in the main lemma, Lemma 23, that the x_0 argument to ADDLITERAL does satisfy the technical condition stated at the end of Lemma 18. This condition is what allows us to prove that ADDLITERAL always finds a literal from the "correct" one of the two target terms.

Lemma 18. *Let* T_1^* *be a term of the target formula that is satisfied by positive instance* x_0*. Let* h *be a (hypothesis) term that contains all literals of* $T_1^* \cap t$*, for initial theory term* t*. Let* y *be a negative instance of the target that satisfies* h*.*

If x_0 *satisfies* h*, and if either the target formula consists of the single term* T_1^**, or the target formula has a second term* T_2^* *and* x_0 *has the property that for every literal* l *in* $T_2^* \setminus (t \cup h)$*, instance* x_0 *has* l *set to off, then Algorithm* ADDLITERAL(y, x_0, t) *finds a literal in* $T_1^* \setminus (t \cup h)$ *on which* x_0 *and* y *disagree.*

Proof: If the conditions of the lemma are met, then both y and x_0 must have all literals in $T_1^* \cap t$ set to on, since y and x_0 both satisfy h. Thus, x_0 and y should still have all the literals of $T_1^* \cap t$ set to on after x_0 and y are altered in Line 2. Therefore, x_0 should still satisfy T_1^*, so if the membership query in Line 4 returns 0, then the conditions of the lemma must be violated.

The altered y must still be a negative instance, since it was changed only by changing variables of known orientation from "on" to "off." Since y is a negative instance and x_0 is a positive instance, BINARYSEARCH will return a literal that is in the target formula with the orientation of x_0. Since the altered x_0 and y agree on $t \cup h$, by the condition on x_0, this returned literal must be in $T_1^* \setminus (t \cup h)$. □

Next we consider the procedure REVISE2TERMS.

Proposition 19. REVISE2TERMS *uses* $O(e \log n)$ *membership queries, and* $O(e)$ *equivalence queries, and always returns either the target formula or "Failure."*

Proof: The only way the algorithm can terminate is either by having an equivalence query return "Yes!" or by returning "Failure."

For each positive counterexample x, we will have made one equivalence query (to obtain x) and we make one membership query, then we decrement the parameter e by at least 1. For each negative counterexample x, we will have made one equivalence query (to obtain x) and we make one call to ADDLITERAL, which uses $O(\log n)$ membership queries, and we decrement the parameter e by 1. □

Lemma 20. *Let the target formula be* $T_1^* \vee T_2^*$ *and the initial formula be* $t_1 \vee t_2$*. Consider a call to* REVISE2TERMS$(T_1, t_1, T_2, t_2, x_0, e)$ *with a two term hypothesis* $T_1 \vee T_2$ *such that*

hypothesis term T_2 covers target term T_2^* and there is a literal l that is in both T_1 and T_1^* that is not in T_2. Let $d = |T_1 \triangle T_1^*| + |T_2 \triangle T_2^*|$.

1. *If T_1 covers T_1^* and $e \geq d$, then* REVISE2TERMS *finds the target using $O(d)$ queries.*
2. *Assume x_0 is such that x_0 satisfies T_1^*, and furthermore, that for each literal (if any) in $T_2^* \setminus (t_1 \cup T_1)$, that literal is set to off in x_0. If T_1 contains every variable in $T_1^* \cap t_1$ (but does not necessarily cover T_1^*) and if every variable in $T_1 \setminus t_1$ is in T_1^*, and if $e \geq d$, then* REVISE2TERMS *finds the target using $O(d \log n)$ queries.*

Proof: Notice that T_2 cannot cover T_1^*, because of l. We will maintain the invariant that T_2 always covers T_2^*, and if T_1 covers T_1^* at the start, then that will also be maintained as an invariant.

For any positive counterexample x, the instance $x' = x \mathbin{\overline{\cap}} T_2$ cannot satisfy T_1^*, because it has l set to off. Thus, if $\mathrm{MQ}(x') = 1$, then x' and x must both satisfy T_2^*, so we can set $T_2 = T_2 \cap x$, and we have used one query to make at least one deletion edit to T_2, and we have maintained the invariant that T_2 covers T_2^*.

If $\mathrm{MQ}(x') = 0$, then x cannot satisfy T_2^*. Thus x must satisfy T_1^*, so we can set $T_1 = T_1 \cap x$, and we have used one query to make at least one deletion edit to T_1. Furthermore, if T_1 covered T_1^* before this edit, then it still does after this edit.

In the case where T_1 covers T_1^*, then all counterexamples must be positive counterexamples, and we are done. If not, then the previous argument still applies to positive counterexamples.

Consider now a negative counterexample x. If T_2 is full, then x cannot be negative and satisfy T_2, so x must satisfy T_1. Lemma 18 guarantees that ADDLITERAL(x, x_0, t_1) will find one necessary addition using at most $O(\log n)$ queries.

Thus the query complexity for Condition 2 of the lemma is $O(|T_2 \setminus T_2^*| + |T_1 \setminus T_1^*|) + O(|T_1^* \setminus T_1| \cdot \log n) = O(d \log n)$, since $d = O(|T_2 \setminus T_2^*| + |T_1 \setminus T_1^*|) + O(|T_1^* \setminus T_1|)$. □

Remark. It can happen after some number of deletions to T_2 that $T_2 \subset T_1$. In this case, for any positive counterexample x, $\mathrm{MQ}(x \mathbin{\overline{\cap}} T_2) = 0$, and at some point setting $T_1 = T_1 \cap x$ will shrink T_1 so that it no longer contains T_2.

We next make an observation about REVISE2TERMS that follows immediately from an examination of its code.

Observation 21. When REVISE2TERMS is called with its maximum number of edits parameter e set to d, then it makes $O(d \log n)$ queries.

The next pair of lemmas are the heart of the correctness argument. The first, Lemma 22, gives a limit on the number of queries consumed by UNATEREVISEUPTOE when e is too small, and also during "wrong" choices of the branch of the main **for** loop. The second, Lemma 23, argues that when UNATEREVISEUPTOE makes the "correct" choices, it finds the target.

Lemma 22. *Algorithm* UNATEREVISEUPTOE(t_1, t_2, e) *makes at most $O(e^2 \log n)$ queries.*

Proof: The algorithm can be restarted at most once at Line 7 or 14 because of a PivotException (by Lemma 14). Therefore, it suffices to bound the number of queries in one iteration of the main **for** $k = 1, 2$ loop.

By Proposition 13, creating the initial one-term hypothesis in Lines 3–19 uses at most $O(e \log n)$ queries. In each iteration of the **while** loop, Lines 21–33, the parameter e decreases by at least 1, so that loop is executed at most e times. Inside that **while** loop, all queries are made inside subroutines. By Propositions 13, 19, and 17, at most $O(e \log n)$ queries are made in the combined subroutine calls of any one iteration of the **while** loop. As claimed, this gives a worst-case query complexity of $O(e^2 \log n)$. □

Lemma 23. *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 $e \geq |t_1 \otimes T_1^*| + |t_2 \otimes T_2^*| \leq |t_1 \otimes T_2^*| + |t_2 \otimes T_1^*|$. Algorithm* UNATEREVISEUPTOE(t_1, t_2, e) *finds the target theory using at most $O(e^2 \log n)$ queries.*

Proof: Assume that the target concept is not FALSE, and let x_0 be the positive counterexample from the initial equivalence query.

By Lemma 14, the value PivotException is returned at most once by a call to FILLTERM from either Line 4 or 11, and by Lemma 22, at most $O(e^2 \log n)$ queries will be used through the time the algorithm restarts after such a PivotException. So, from now on, we assume that PivotException will not be returned from calls to FILLTERM at either Line 4 or 11 (either because that never happens for this particular x_0, or because it has already happened once and the algorithm has restarted with a modified x_0).

We will argue that if x_0 satisfies T_1^*, then the first iteration of the main **for** loop must find the target formula in the specified number of queries. Now, if x_0 does not satisfy T_1^*, it must satisfy T_2^*. If x_0 satisfies T_2^* but not T_1^*, then the first iteration of the main **for** loop may fail after $O(e^2 \log n)$ queries, but in that case, the **for** $k = 2$ iteration will find the target in $O(e^2 \log n)$ queries, and the argument is exactly analogous to the case where x_0 satisfies only T_1^*. Thus it suffices to prove that if x_0 satisfies T_1^*, the $k = 1$ iteration of the **for** loop succeeds.

From here forward, we assume x_0 satisfies T_1^*. We proceed by cases to discuss the creation of the first term of the hypothesis. The goal of this discussion is for each case to get to a point where we can finish the proof by using Lemmas 16 and 20.

Case I: Both $\mathrm{MQ}(x_0 \underline{\cap} t_1) = 1$ and $\mathrm{MQ}(x_0 \overline{\cap} t_1) = 1$.

The initial one-term hypothesis h created in Line 18 of UNATEREVISEUPTOE is $t_1 \cap x_0$.

I.i. $T_1^* \subseteq t_1$.

Since $T_1^*(x_0) = 1$, it must be that $h = t_1 \cap x_0$ covers T_1^*. Creating the term h used only a constant number of queries. In Case I.i, we will maintain the hypothesis term h's covering T_1^* as an invariant. Thus, any counterexample to $\mathrm{EQ}(h)$, where h is one term, must be a positive counterexample.

For each positive counterexample y to h, if y satisfies T_1^*, then, as we discuss below, we will ultimately use y to make deletions to h, by setting $h = h \cap y$. If positive counterexample y does not satisfy T_1^*, then since h covers T_1^*, it holds that

$$\text{at least 1 literal } l \text{ of } T_1^* \text{ is both in } h \text{ and set to off in } y. \tag{3}$$

I.ii. $T_1^* \not\subseteq t_1$.

That is, T_1^* contains variables not in t_1. Since the operation $x_0 \underline{\cap} t_1$ flips the setting of all variables of x_0 not in t_1, instance $x_0 \underline{\cap} t_1$ cannot satisfy T_1^*, so

$$x_0 \underline{\cap} t_1 \text{ must satisfy } T_2^*. \tag{4}$$

Since $x_0 \underline{\cap} t_1$ has all variables in $t_2 \backslash t_1$ set to off,

$$T_2^* \text{ has no literals from } t_2 \backslash t_1. \tag{5}$$

In order to apply Lemma 18 for adding literals when we receive negative counterexamples, we wish to show that h contains all literals of $T_1^* \cap t_1$ and that x_0 has all literals of $T_2^* \backslash (t_1 \cup h)$ set to off.

Since h is initialized to $t_1 \cap x_0$ and x_0 satisfies T_1^*, it must be that h does contain all literals of $T_1^* \cap t_1$.

We can split the literals of T_2^* into three groups: outside literals, literals in $t_2 \backslash t_1$, and literals in t_1. By Fact (4), x_0 must have all T_2^*'s outside literals off, since x_0 and $x_0 \underline{\cap} t_1$ disagree on all those literals. Fact (5) says that T_2^* contains no variables in $t_2 \backslash t_1$. Therefore, x_0 has all literals in $T_2^* \backslash t_1$ set to off.

Thus, Lemma 18 applies when we use a negative counterexample y to add a literal to h using subroutine ADDLITERAL(y, x_0, t_1). Lemma 18 says that we add a necessary literal to h at the cost of $O(\log n)$ queries.

Now consider a positive counterexample y to EQ(h). If y covers all the inside variables of T_1^*, then we will ultimately edit h to become $h \cap y$. If not, then Fact (3) once again holds.

Case II: MQ$(x_0 \underline{\cap} t_1) = 0$.

This implies that $T_1^* \not\subseteq t_1$, since x_0 satisfies T_1^*. In this case, we call FILLTERM$(t_1 \cap x_0, x_0 \underline{\cap} t_1, x_0, e)$.

If PivotException is not returned, then Lemma 15 says that FILLTERM will find some necessary additions to $t_1 \cap x_0$ using $O(a \log n)$ queries, where $a = |T_1^* \backslash (t_1 \cap x_0)|$ is the number of additions found.

If FILLTERM found all the outside literals, then there will never be any negative counterexamples. If it did not, then we need to show that Lemma 18 can be applied. In particular, we will need to show that all the literals of T_2^* not in $t_1 \cup h$ are set to off in x_0.

If the a outside literals that FILLTERM found were not all the outside literals of T_1^*, then let $x' = (x_0 \underline{\cap} t_1)$ with those a literals flipped to on. When FILLTERM halted, it must have been because MQ$(x') = 1$. Furthermore, x' does not satisfy T_1^*, by the assumption that FILLTERM did not find all the outside variables of T_1^*. Therefore, x' satisfies T_2^*.

Now x' agrees with $x_0 \underline{\cap} t_1$ on all variables outside $t_1 \cup h$, where h is the hypothesis immediately after FILLTERM is returned. Thus, for all outside literals of T_2^* not in $t_1 \cup h$, x_0 has them off (since $x_0 \underline{\cap} t_1$ has them on). Any inside literal of T_2^* in $t_2 \backslash t_1$ must be in h, because all those literals were off in $x_0 \underline{\cap} t_1$, but x' satisfies T_2^*, and h contains all literals on which $x_0 \underline{\cap} t_1$ and x' differ.

For positive counterexamples, we have the same argument as in Case I.

Case III: MQ$(x_0 \underline{\cap} t_1) = 1$, and MQ$(x_0 \overline{\cap} t_1) = 0$.

Notice that in this case T_1^* must contain variables not in t_1, specifically some variables from $t_2 \backslash t_1$. (If the necessary additions to $t_1 \cap x_0$ were all outside of t_2 then $\mathrm{MQ}(x_0 \cap \overline{t_1})$ would be 1.) Furthermore, since those variables are *off* in $x_0 \cup t_1$, it must be that $x_0 \cup t_1$ satisfies T_2^*.

So $x_0 \cup t_1$ is a positive example that definitely satisfies term T_2^* (and not term T_1^*).

Furthermore, T_2^* contains no literals in $t_2 \backslash t_1$, but T_2^* must contain at least one outside literal. Therefore, it must be that $\mathrm{MQ}((x_0 \cup t_1) \cup t_2) = 0$, since that instance has the outside literals satisfied by $x_0 \cup t_1$ set to off.

Thus Case II applies, with the roles of t_1, T_1^* and t_2, T_2^* switched and $x_0 \cup t_1$ replacing x_0.

This concludes the discussion of initializing the first term. Now we know we have a one-term hypothesis, and that we can handle negative counterexamples.

If we receive a positive counterexample y such that Fact (3) holds, then Lemma 16 applies, so we are guaranteed that FILLTERM$(t_2 \cap y, y \cup t_2, y, \cdot)$ returns a term that covers T_2^*. By Lemma 15, the number of queries consumed by that call to FILLTERM will be $O(\log n)$ times $|T_2^* - (t_2 \cap y)|$, and since y satisfies T_2^*, we have that $T_2^* - (t_2 \cap y) = T_2^* - t_2$. In other words, the number of queries consumed is $O(\log n)$ times the number of outside variables added to the second hypothesis term.

If instead Fact (3) does *not* hold for positive counterexample y, then y agrees with every literal that is in both h and T_1^*, then setting $h = h \cap y$ performs necessary deletions from h. In the worst case, we may have used $O(e \log n)$ queries inside FILLTERM before FILLTERM failed and returned, and y was then instead used to make one necessary deletion from the one term h.

Once we do have a call to FILLTERM that satisfies (3), we are in the subroutine REVISE2TERMS. FILLTERM cannot have added the literal l to the second term, so the conditions of Lemma 20 are met. Thus REVISE2TERMS performs only deletions to our two-term hypothesis, using only a constant number of queries per deletion.

Note that in the cases where the first hypothesis term is not yet full, the conditions we proved on x_0 that guaranteed via Lemma 18 that ADDLITERAL could successfully process negative counterexamples to a one-term hypothesis still hold. That is why REVISE2TERMS can successfully use ADDLITERAL to process negative counterexamples to the first hypothesis term.

The dominant factor in the query complexity is the $O(e \log n)$ queries we might use per deletion from the first term before the second term is initialized, giving an overall query bound of $O(e^2 \log n)$. □

Theorem 24. *There is a revision algorithm for two-term unate DNF in the general model of revisions, using $O(e^2 \log n)$ queries, where n is the number of variables in the universe and e is the revision distance between the initial and target formulas.*

Proof: We make repeated calls to UNATEREVISEUPTOE with the error parameter set to $1, 2, 4, 8, \ldots$ until UNATEREVISEUPTOE returns success. By Lemma 23, this happens by the time the number-of-edits-remaining parameter reaches or first surpasses e when the target contains two terms. A similar, but simpler argument shows that the same is true if the target contains only one term.

The only thing left to argue is that the calls to UNATEREVISEUPTOE with too-small values of e cumulatively used only the allowed number of membership queries. A careful analysis of the proof of Lemma 23 shows that all these failing calls to UNATEREVISEUPTOE also consume only $O(e^2 \log n)$ queries. □

6. A hard monotone DNF

The results of Sections 3 and 4.3 show that for deletion-type revisions, monotone DNF can be efficiently revised if either the terms are small or there are only a few terms. A third type of restriction is to assume that the terms have small overlap. In the extreme case this means that the terms are disjoint, that is, we are dealing with read-once DNF. We showed (Goldsmith et al., 2000; Sloan & Turán, 1999; Szörényi, 2000) that efficient revision is possible in this case as well. In this section, we show that at least one of these restrictions is necessary, as there are monotone DNF expressions with many large terms having large overlap that cannot be revised efficiently. A more precise understanding of the relationship between the revision complexity of monotone DNF and the different combinatorial parameters appears to be an interesting open problem.

Consider the variables $x_1, \ldots, x_n, y_1, \ldots, y_n$ and let $\varphi_n = t_1 \vee \cdots \vee t_n$, where, for $i = 1, \ldots, n$,

$$t_i = x_1 \wedge \cdots \wedge x_{i-1} \wedge x_{i+1} \wedge \cdots \wedge x_n \wedge y_i.$$

Theorem 25. *The formula φ_n requires at least $n-1$ membership and equivalence queries to be revised, even if it is known that exactly one literal y_i is deleted.*

Proof: We describe an adversary strategy for answering the queries of any learning algorithm. Let ψ_i be the formula obtained from φ_n by deleting the single occurrence of y_i. Thus, initially the set of possible target concepts is $\Psi = \{\psi_1, \ldots, \psi_n\}$. Using the adversary strategy described below, each query eliminates at most one concept from Ψ, and this implies the claimed lower bound.

Consider a truth assignment (a, b), where a is a truth assignment to the x_i's and b is a truth assignment to the y_i's. Then the membership query $\mathrm{MQ}(a, b)$ is answered as follows. If a has at most $n-2$ bits that are 1, then $\mathrm{MQ}(a, b) = 0$. This does not eliminate any concepts from Ψ. If a has $n-1$ bits that are 1 and $a_i = 0$, then $\mathrm{MQ}(a, b) = b_i$. If $b_i = 1$ then this does not eliminate any concept from Ψ. If $b_i = 0$ then ψ_i is eliminated from Ψ. If $a = \mathbf{1}$ then $\mathrm{MQ}(a, b) = 1$. This does not eliminate any concept from Ψ.

Now consider an equivalence query $\mathrm{EQ}(\theta)$, where θ is a revised version of φ_n. Thus θ may be obtained from φ_n by deleting any number of literals or terms. If θ contains a term t with at most $n-2$ of the x_i's, then return the negative counterexample $(t, \mathbf{1})$. This does not eliminate any concept from Ψ. If θ contains a term t such that t contains all the x variables except x_i and it does not contain y_i, then return the negative counterexample (t, f_i), where f_i has all 1's except in the ith position. This eliminates only ψ_i from Ψ. Finally, if θ is φ_n itself, then return the positive counterexample $(\mathbf{1}, \mathbf{0})$. This does not eliminate any concept from Ψ. □

7. Further results and open problems

Besides the classes considered in this paper, efficient theory revision algorithms for read-once formulas and propositional Horn formulas are given in Goldsmith and Sloan (2000), Goldsmith et al. (2000, 2001), Sloan and Turán (1999), Szörényi (2000). There remain many open problems. It would be interesting to extend the revision algorithm for unate two-term DNF to the case of several terms, and to eliminate the restriction on changing the orientation of a literal. Although the last section contains a negative result for revising general DNF, there are no lower bounds known for the subclasses discussed in this paper. Revising propositional Horn sentences is an important problem from the point of view of practical revision algorithms in machine learning. Some open problems in this direction are discussed in Goldsmith et al. (2001).

From a broader perspective, theory revision models learning situations where the learning process starts with a roughly correct theory. It can be argued that for many complex learning tasks having such an initial theory is indispensable for successful learning. Understanding the possibilities and limitations of efficient revision algorithms in this general context appears to be an important task for computational learning theory.

Acknowledgments

Various parts of this paper are based on preliminary results that appeared in three conference papers in COLT 99 (Sloan & Turán, 1999), STOC 2000 (Goldsmith & Sloan, 2000), and COLT 2000 (Goldsmith et al., 2000).

Judy Goldsmith and Robert Sloan are grateful to Martin Mundhenk for helpful discussions on the algorithm for revising monotone DNF in Section 4. All authors thank both Balázs Szörényi and one of the anonymous referees for very detailed, helpful feedback that has hopefully made this article much more readable.

Notes

1. This theory seems to work well for the second author's five-year old.
2. The AI notion of a theory encompasses both the notion of a single concept as well as richer constructs, such as multiple concepts being learned at the same time. Certainly the revision of propositional logic concepts falls within the AI notion of theory revision, and in this paper we will use the terms theory revision and concept revision interchangeably.
3. In the literature, the restriction that h come from the concept class is called *proper* equivalence queries.
4. Incidentally, at this point x may still satisfy more than one target term, but our argument applies to any one of those.

References

Alexin, Z., Gyimóthy, T., & Boström, H. (1997). IMPUT: An interactive learning tool based on program specialization. *Intelligent Data Analysis, 1:4*. ⟨http://www-east.elsevier.com/ida/⟩.

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

Angluin, D., Hellerstein, L., & Karpinski, M. (1993). Learning read-once formulas with queries. *J. ACM, 40*, 185–210.

Argamon-Engelson, S., & Koppel, M. (1998). Tractability of theory patching. *Journal of Artificial Intelligence Research, 8*, 39–65.

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

Bshouty, N., Hancock, T., Hellerstein, L., & Karpinski, M. (1994). An algorithm to learn read-once threshold formulas, and transformations between learning models. *Computational Complexity, 4*, 37–61.

Bshouty, N., & Hellerstein, L. (1998). Attribute-efficient learning in query and mistake-bound models. *J. Comput. Syst. Sci.*, 56, 310–319.

Case, J., Kaufmann, S., Kinber, E., & Kummer, M. (1997). Learning recursive functions from approximations. *J. Comput. Syst. Sci., 55*, 183–196.

Davis, R., & Hamscher, W. (1988). Model-based reasoning: Troubleshooting. In H. E. Shrobe, & the American Association for Artificial Intelligence (Eds.), *Exploring artificial intelligence: Survey talks from the national conferences on artificial intelligence* (Ch. 8, pp. 297–346). San Mateo, CA: Morgan Kaufmann.

de Kleer, J., Mackworth, A. K., & Reiter, R. (1992). Characterizing diagnoses and systems. *Artificial Intelligence, 56*, 197–222.

Goldsmith, J., & Sloan, R. H. (2000). More theory revision with queries. In *Proc. 32nd Annu. ACM Sympos. Theory Comput.* (pp. 441–448).

Goldsmith, J., Sloan, R. H., Szörényi, B., & Turán, G. (2000). Improved algorithms for theory revision with queries. In *Proc. 13th Annu. Conference on Comput. Learning Theory* (pp. 236–247). San Francisco: Morgan Kaufmann.

Goldsmith, J., Sloan, R. H., Szörényi, B., & Turán, G. (2001). Theory revision with queries: Horn and related formulas. In preparation.

Helmbold, D. P., & Long, P. M. (1994). Tracking drifting concepts by minimizing disagreements. *Machine Learning, 14:1*, 27–45.

Jain, S., & Sharma, A. (1991). Learning in the presence of partial explanations. *Inform. Comput., 95:2*, 162–191.

Kohavi, Z. (1978). *Switching and finite automata theory*. 2nd edn. New York, NY: McGraw-Hill.

Koppel, M., Feldman, R., & Segre, A. M. (1994). Bias-driven revision of logical domain theories. *Journal of Artificial Intelligence Research, 1*, 159–208.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence, 33:1*, 1–64.

Marcotte, R. A., Neiberg, M. J., Piazza, R. L., & Holtzblatt, L. J. (1992). Model-based diagnostic reasoning using VHDL. In J. M. Schoen (Ed.), *Performance and fault modeling with VHDL* (Ch. 6, pp. 304–399). Englewood Cliffs, NJ: Prentice Hall.

Mooney, R. J. (1995). A preliminary PAC analysis of theory revision. In *Computational learning theory and natural learning systems*, *Vol. III: Selecting Good Models* (Ch. 3, pp. 43–53). MIT Press.

Ourston, D., & Mooney, R. J. (1994). Theory refinement combining analytical and empirical methods. *Artificial Intelligence, 66*, 273–309.

Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence, 32*, 57–95.

Richards, B. L., & Mooney, R. J. (1995) Automated refinement of first-order Horn-clause domain theories. *Machine Learning, 19*, 95–131.

Rivest, R. L., & Sloan, R. (1994). A formal model of hierarchical concept learning. *Inform. Comput., 114*, 88–114.

Shapiro, E. Y. (1983). *Algorithmic program debugging*. Cambridge, MA: MIT Press.

Sloan, R. H., & Turán, G. (1999). On theory revision with queries. In *Proc. 12th Annu. Conf. on Comput. Learning Theory* (pp. 41–52). New York, NY: ACM Press.

Szörényi, B. (2000). Revision algorithms in computational learning theory. Master's thesis, Dept. of Computer Science, University of Szeged. (In Hungarian.).

Towell, G. G., & Shavlik, J. W. (1993). Extracting refined rules from knowledge-based neural networks. *Machine Learning, 13*, 71–101.

Towell, G. G., & Shavlik, J. W. (1994). Knowledge-based artificial neural networks. *Artificial Intelligence, 70:1/2*, 119–165.

Uehara, R., Tsuchida, K., & Wegener, I. (1997). Optimal attribute-efficient learning of disjunction, parity, and threshold functions. In *Computational learning theory: EuroColt '97*, Berlin: Springer-Verlag (pp. 171–184).

Wrobel, S. (1995). First order theory refinement. In L. De Raedt (Ed.), *Advances in ILP* (pp. 14–33). Amsterdam: IOS Press.

Received December 4, 2000
Revised June 14, 2001
Accepted June 20, 2001
Final manuscript July 2, 2001