Improved Algorithms for Theory Revision with Queries

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

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

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

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

Abstract

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

1 INTRODUCTION

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

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

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

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

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

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

1.1 MODEL OF THEORY REVISION

The key metric for theory revision is the syntactic distance between the initial theory and the target theory. The syntactic distance between a given concept representation and another concept is the minimal number of elementary operations (such as the addition or the deletion of a literal or a clause) needed to transform the given concept representation to a representation of the other concept. Our goal in theory revision is to find algorithms whose query complexity is polynomial in the syntactic difference (or revision distance) between the initial theory and the target theory, but only polylogarithmic in the total number of possible variables. Thus, this work has some similarities to the work on
attribute-efficient learning [3, 4].

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

1.2 RELATED WORK

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

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

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

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

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

1.3 DISCUSSION OF OUR RESULTS

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

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

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

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

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

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

2 NOTATION

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

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

The symbol \( \subseteq \) will always denote strict subset.

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

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

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

“Intersect down” is defined by

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

and “intersect up” is defined by

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

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

If \( \varphi = T_1 \lor T_2 \), then \( T_\overline{T} \) denotes the term other than \( T_i \).

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

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

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

3 REVISING MONOTONE DNF

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

3.1 DESCRIPTION OF ALGORITHM

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

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

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

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

The heart of the construction is the procedure REVISE-UpToE. It takes as parameters an \( m \)-term monotone DNF formula \( \varphi \) and \( e \), the assumed revision distance from \( \varphi \) to the
Algorithm 1 \textsc{ReviseUpTo}$(\varphi, e)$. Revises \(\varphi\), a set of monotone terms, if possible using at most \(e\) revisions; otherwise returns "Failure." Note that if any subroutine either finds the correct hypothesis or returns "Failure", then this algorithm also terminates. Also, if the error limit \(e\) is ever exceeded, this algorithm terminates immediately and returns "Failure".

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

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

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

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

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

### 3.2 MONOTONE DNF CORRECTNESS AND QUERY COMPLEXITY

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

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

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

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

**Lemma 3** If \( x \) is used to add a new term to \( h \), the new term does not cover any target term already covered by \( h \). Thus, no two terms in \( h \) cover the same target term.
Proof sketch. Suppose $t'$ is added to $h$ because of counterexample $x$. In order for the algorithm to reach Line 8, it must be the case that for any $t$ already in $h$, $MQ(x \cap t) = 0$. Note that $t' \subseteq x$, by construction. Therefore, $t \cap t' \subseteq t \cap x$, so (by monotonicity) $MQ(t \cap t') = 0$.

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

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

**Proof.** Note that no term in $h$ covers either $T_1^*$, since $x$ was not used for deletions.

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

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

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

**Proof sketch.** Note that there can be at most $e$ additions to any $x \cap t$ for any $t \in \varphi$; if more additions are needed, the attempt to edit that term fails. Each search for an addition requires $\log n$ membership queries. However, even if the counterexample $x$ covers a unique target term, the algorithm may try each of the $m$ terms of $\varphi$ 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 $O(m^2e \log n)$ membership queries.

**Theorem 6** REVISEUPTOE($\varphi$, $e$) uses at most $O(m^2e \log n)$ membership queries and $O(e + m)$ equivalence queries, and succeeds if $\varphi$ has revision distance less than or equal to $e$. Therefore, an $m$-term monotone DNF formula with an edit distance $e$ from the target formula can be revised using $O(m^2e \log n)$ queries.

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

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

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

There are $O(e)$ deletions needed, and each deletion requires $O(m)$ membership queries, for a total of $O(emn)$, which is $O(m^2e \log n)$ queries.

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

4. REVIS E UNATE DNF

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

4.1 DESCRIPTION OF ALGORITHM

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

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

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

4.1.1 Binary search

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

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

1: while MQ(Y) == 0 and e > 0 do
2: \( S = Y, T = Z \).
3: while \( |T \otimes S| > 1 \) do //binary search for 1 addition
4: Divide positions where S and T disagree into approximately equal-size sets \( d_1 \) and \( d_2 \).
5: Put Mid = S with positions in \( d_1 \) replaced by T’s values
6: if MQ(Mid) == 0 then
7: \( S = \text{Mid} \)
8: else
9: \( T = \text{Mid} \)
10: end if
11: end while
12: Let \( l \) be the 1 position in \( T \otimes S \)
13: if MQ(Z with position \( l \) flipped) = 1 then
14: throw PivotException(Y, l, e, Z)
15: end if
16: \( Y = Y \cup \text{the value of Z for position } l \)
17: \( e = e - 1 \)
18: end while
19: if MQ(Y) == 1 then //Y covers term
20: return Y, e
21: else //\( e \leq 0 \), so all edits already used
22: return “Failure”
23: end if

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

4.1.2 Revise Down

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

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

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

Algorithm 3 REVISEDOWN(T₁, T₂, e, p₁, p₂)

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

1: Put \( h = T_1 \vee T_2 \).
2: while \( (x = \text{EQ}(h)) \neq \text{“Correct”} \) do
3: if \( e \leq 0 \) then //Used up all allowable edits
4: return Failure
5: end if
6: if \( h(x) = 0 \) then //x is positive counterexample
7: for all terms \( T_j \) do
8: \( x' = x \cap T_j \).
9: Turn “off” in \( x' \) any outside variables in \( T_j \cap T_1 \)
10: if MQ(x) == 1 then
11: \( T_j = T_j \cap x \)
12: Decrement \( e \) by # deletions to \( T_j \)
13: end if
14: end for
15: if no term of \( h \) was revised by \( x \) then
16: return “Failure”
17: end if
18: else //\( x \) is a negative counterexample
19: if \( T_1(x) == T_2(x) == 1 \) then
20: return Failure
21: else
22: Let \( T_i \) be unique term of \( h \) such that \( T_i(x) = 1 \)
23: if \( p_i == \text{NULL} \) then
24: return Failure
25: end if
26: \( z = \text{vector with inside variables of } x \) and \( \text{outside variables of } p_i \).
27: if MQ(z) \neq 1 then
28: return Failure
29: end if
30: Perform binary search from \( x \) to \( z \)
31: Add all variables found to \( T_i \) and decrement \( e \) accordingly
32: end if
33: end if
34: end while

The preceding discussion actually applies only to certain calls of REVISEDOWN. As we will discuss soon in Section 4.1.4, the main algorithm tries calling REVISEDOWN with several different parameters, intending to abandon all but one of the calls. For the calls that will be abandoned (intuitively, the ones where the algorithm has made the wrong
ample satisfies both hypothesis terms. If so, then it must be that neither hypothesis term is full, so REVISEDOWN terminates with failure.

4.1.3 Negative counterexamples

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

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

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

4.1.4 Main Algorithm: REVISEUPTOE

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

Assume for now that both \( \text{MQ}(x \overline{t}_1(t_1 \cap t_2)) = 0 \) and \( \text{MQ}(x \overline{t}_2(t_1 \cap t_2)) = 0 \). We will describe how to handle the case where that is not true a bit later.

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

Here is how we proceed when we are assuming that \( t_1 \) should be edited to create a hypothesis term \( T \) such that \( T(x) = 1 \). We ask the two membership queries \( \text{MQ}(x \overline{t}_1) \) and \( \text{MQ}(x \overline{t}_2) \). If both return 1, then our initial one-term hypothesis is \( t_1 \cap x \). Intuitively, we are hoping that the responses to the membership queries indicated that \( x \) satisfies a target term that is contained in \( t_1 \) though, as we discuss in the proof of Lemma 8, this is not necessarily the case. We remember for later that \( t_1 \) is the term that we edited, and that

Algorithm 4 REVISEUPTOE(\( \varphi_0(= t_1 \lor t_2), \epsilon \))

Note that a branch of a “try both” fails if one of the subroutines it calls fails without being explicitly tested for failure.

1: Let \( x \) be positive instance (from EQ(FALSE))
2: \textbf{try both} \( b = 1, 2 \):
3: Work with \( x \) as described in text to create a one-term hypothesis \( h \) assuming that \( x \) satisfies target term that can be derived from \( t_b \) to minimize total edits
4: Let \( t_b \) be term of \( \varphi_0 \) that \( h \) is derived from
5: Let \( p_i \) be positive instance associated with \( h \)
6: \textbf{while} \( (y = \text{EQ}(h)) \neq \text{"Correct"} \) and \( e > 0 \) \textbf{do}
7: \textbf{if} \( h(y) = 1 \) then //Negative counterexample
8: \textbf{if} \( p_i = \text{NULL} \) then
9: \textbf{return} “Failure”
10: \textbf{else} //\( h \) needs more variables
11: \( z = \) vector with inside variables of \( y \) and outside variables of \( p_i \)
12: \textbf{if} \( \text{MQ}(z) \neq 1 \) then
13: \textbf{return} Failure
14: \textbf{end if}
15: Perform binary search from \( x \) to \( z \)
16: Add all variables found to \( h \); decrement \( e \) accordingly
17: \textbf{end if}
18: \textbf{else} //\( y \) is a positive counter example
19: \( h_0 = h; \ e_0 = e \)
20: \( y' = y \lor t_r \)
21: \textbf{if} \( t_r \cap y \subseteq h \) and then \( \text{MQ}(y') = = 1 \) then
22: \( h = h \lor y' \)
23: \textbf{else if} \( \text{BINARYSEARCH}(y', y, \epsilon) \) returns \( (z, e) \) (rather than “Failure”) then
24: \( h = h_0 \lor (t_r \cap y) \) plus literals of \( z \) in \( z \lor y' \)
25: //REVISEDOWN may find target
26: \textbf{if} REVISEDOWN(h, e) returns “Failure” then
27: \( h = h_0 \lor y \\
28: e = e_0 - |h_0 \setminus h| \)
29: \textbf{end if}
30: \textbf{else} //\( \text{BINARYSEARCH}(y', y, \epsilon) \) returns “Failure”
31: \( h = h_0 \lor y \\
32: e = e_0 - |h_0 \setminus h| \)
33: \textbf{end if}
34: \textbf{end if}
35: \textbf{end while}
36: \textbf{end try both}

“guesses” about the parameters), the two-term hypothesis given to REVISEDOWN could have neither term full. In this case, however, it is fine for REVISEDOWN to fail. In fact, when REVISEDOWN receives a negative counterexample, it checks, in Line 19, to see whether that negative counterexample satisfies both hypothesis terms. If so, then it must be that neither hypothesis term is full, so REVISEDOWN terminates with failure.
Lemma 7 When ReviseDown is called with its maximum number of edits parameter \(e\) set to \(d\), then it makes \(O(d \log n)\) queries.

The following lemma is the heart of the correctness argument.

Lemma 8 Let \(\varphi_0 = t_1 \lor t_2\) be an initial theory, and let \(\Phi^+ = T^*_1 \lor T^*_2\) be a target theory, with the \(T^*_i\) labeled so that \(e = |t_1 \otimes T^*_1| + |t_2 \otimes T^*_2| \leq |t_1 \otimes T^*_2| + |t_2 \otimes T^*_1|\). Consider a run of ReviseUpToE(\(\varphi_0, e\)). If the positive instance \(x\) used in Line 1 satisfies only the one target term \(T^*_j\), and MQ(\(x\)) \(= MQ(x_1 \lor x_2) = MQ(x_3 \lor x_4) = 0\), then the branch of the “Try both” where \(i = j\) finds the target theory using at most \(O(e^2 \log n)\) queries.

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

We proceed by cases.

Case 1: Both MQ(\(x_1 \lor x_2\)) = 1 and MQ(\(x_3 \lor x_4\)) = 1.

The initial one-term hypothesis created in Line 2 of ReviseUpToE is \(t_1 \lor x\). Call this term of the hypothesis \(T\). Notice that \(T\) cannot cover \(T^*_1\). This is true if \(T^*_1\) contains any variables not in \(t_1\). Even if all \(T^*_1\)’s variables are in \(t_1\), however, \(T\) still cannot cover \(T^*_2\), since the hypothesis of the lemma is that \(x\) does not satisfy \(T^*_2\), and \(T = t_1 \lor x\).

Thus, the second term requires \(O(e \log n)\) queries.

One special case can arise. If MQ(\(y\)) = 1 and \(t_1 \cap y \subseteq T\) (and, in fact, since \(y\) is a counterexample, it must be that \(t_1 \cap y \subseteq T\)), then intuitively we certainly do not want to add \(t_1 \cap y\) as a second hypothesis term, because then the first term would be redundant. Formally, we can argue as follows. If both \(y\) and \(y' = y_1 \otimes t_1\) satisfy \(T^*_1\), then \(T^*_1\) has all its variables in \(t_1\), and indeed in \(t_2 \cap y\). However, since \(t_1 \cap y \subseteq T\), that would mean that \(T\) covers \(T^*_1\), which is false. So, at least one of \(y\) or \(y'\) satisfies \(T^*_1\). In this case (checked for in Lines 18–21 of ReviseUpToE), we can safely use the inside variables of \(y\) (which are the same as the inside variables of \(y'\)) to make deletions from \(T\).

When instead we are trying to use \(y\) to start a second hypothesis term, we make one binary search, using \(O(e \log n)\) membership queries, to initialize a second term of our hypothesis. After that, we are in subroutine ReviseDown, which performs only deletions to our two-term hypothesis, using only a constant number of queries per deletion. Thus, if \(y\) actually satisfies \(T^*_2\), we obtain the target using at most \(O(e)\) equivalence and \(O(e \log n)\) membership queries. If \(y\) does not satisfy \(T^*_2\), we backtrack after \(O(e \log n)\) queries, and use \(y\) to perform at least one needed deletion from term \(T\). Thus the total number of queries is at most \(O(e)\) equivalence and \(O(e \log n)\) membership queries per deletion.

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

If, instead, MQ(\(x_1 \lor t_1\)) = 0, then we perform a binary search from \(x_1 \lor t_1\) to \(x\), and our initial one-term hypothesis is \(t_1 \lor x\) plus whatever additional variables were found by the binary search. In this case, we should never see a negative counterexample to this term, so we make the associated positive instance NULL. (If we do receive a negative counterexample, it indicates that we are in the wrong branch of the “try both.”) Again, \(t_1\) is the edited initial theory term.

The final possibility is that MQ(\(x_1 \lor t_1\)) = 0, but MQ(\(x_2 \lor t_1\)) = 1. If \(T^*_1(x) = 1\), then it must be that \(T^*_1\) contains some variables from \(t_2 \setminus t_1\), since MQ(\(x_2 \lor t_1\)) = 0. Thus, we can be certain that \(x'^* = x_1 \lor t_1\) satisfies only the other target term. So, if MQ(\(x_2 \lor t_2\)) = 1, then we do a binary search from \(x'_2 \lor t_2\) to \(x'\) to find which variables we need to add to \(t_2 \lor x'\) to create our initial hypothesis. Either way, we indicate that our hypothesis term has actually been derived from \(t_2\), and that it is full, so we should never receive any negative counterexamples to it.

Now let us explain what we do if one or both of MQ(\(x_1 \lor t_1\)) = 1 or MQ(\(x_2 \lor t_1\)) = 1. In this case, \(t_1 \lor t_2\) plays a very similar role to \(t_1\) above in the case where we assumed that both these membership queries returned 0. If both membership queries return 1, then we initialize our one-term hypothesis to be \(t \lor x\), and \(x_1 \lor t_1\) is the associated positive instance, and we must “try both” associations of the derivation of \(t \lor x\) with \(t_1\) and \(t_2\). (I.e., when a second term is added, it should be derived from the other initial theory term).

If exactly one of MQ(\(x_1 \lor t_1\)) and MQ(\(x_2 \lor t_1\)) is 1, then we can do a binary search between \(x_2 \lor t_1\) and \(x_1 \lor t_1\) or vice versa and derive a hypothesis term that is full. In this case the associated positive instance is NULL, but we still have to “try both” possibilities (\(t_1\) and \(t_2\)) for the derivation of the new term.

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

4.2 Correctness and Query Complexity

We first make an observation about ReviseDown that follows immediately from an examination of its code.
ii. \( T^*_t \not\subseteq t_t \).

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

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

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

1. \( T \) includes all inside variables of \( T^*_t \), but no outside variables.

2. \( T^*_t \) contains at least one outside variable, and the sets of outside variables of \( T^*_t \) and \( T^*_w \) are disjoint.

Recall that \( MQ(x \cap t_t) = MQ(x \cap t_t) = 1 \), but by assumption, \( t_t \) does not cover \( T^*_t \). Also, by assumption, \( T^*_t(x) = 1 \), so it must be that \( x \cap t_t \) satisfies \( T^*_t \). Since \( T^*_t \) contains variables not in \( t_t \) and \( x \), and \( x \cap t_t \) differ on those variables, \( x \cap t_t \) must satisfy \( T^*_t \). Since \( x \cap t_t \) and \( x \cap t_t \) satisfy two different target terms, and both those instances have all the necessary additions to both possible revisions (the \( T^*_t \) made to the opposite target terms).

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

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

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

**Proof sketch.** We make repeated calls to **Revise**UpToE with the error parameter set to 1, 2, 4, 8, … until **Revise**UpToE returns success. We claim that this happens by the time the error parameter reaches or first surpasses \( e \).

Consider first the case where the initial positive example \( x \) covers only one term and the other condition of Lemma 8 is met. Lemma 8 guarantees that the branch of the “Try both” catches a pivot exception thrown by binary search. The branch that throws the exception can have made at most \( O(e \log n) \) queries before throwing the exception. After the exception we restart the program with a new counterexample that is guaranteed to satisfy the conditions of Lemma 8.

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

# 5 REVISIGN READ-ONCE FORMULAS

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

An \( \Omega(e \log(n/e)) \) lower bound to the number of queries is proved in [15]. It is also shown in [15] that using only one type of query, one needs a number of queries that is linear in \( n \).
Theorem 10 Every $n$-variable read-once formula $\varphi$ has a revision algorithm that uses $O(e \log n)$ queries, where $e$ is the revision distance between $\varphi$ and the target concept.

Proof outline. Let us review a bit of terminology from [14]. We assume w.l.o.g. that $\varphi$ is monotone. If $\varphi'$ is a subformula of $\varphi$, then every truth assignment $\alpha$ can be written as $(x_1, x_2)$, called the $\varphi'$-partition of $x$. Here $x_1$ contains all the variables in $\varphi'$, and $x_2$ contains all the variables not in $\varphi'$. Let $\varphi'$ be a subformula of $\varphi$ and let $P$ be the path leading from the root of $\varphi$ to the root of $\varphi'$ in the binary tree representing $\varphi$. Then, using the commutativity of AND and OR, $\varphi$ can be written as

$$(\cdots (\varphi_{c_1} \varphi_{c_2} \cdots \varphi_{c_{2n}}) \varphi_{c_{2n-1}} \cdots \varphi_{c_2} \varphi_{c_1}) \alpha_1 \varphi_1,$$ (1)

where $\varphi_{c_1}, \ldots, \varphi_{c_{2n}}$ are the subformulas corresponding to the siblings of the nodes of $P$, and $\alpha_1, \ldots, \alpha_{2n}$ are either AND or OR. Let the sets of variables occurring in $\varphi_1$, $X_1$, and the set of variables occurring in $\varphi'$ be $Y$. These sets form a partition of $\{x_1, \ldots, x_n\}$. Now let $\alpha$ be the partial truth assignment that assigns 1 (resp., 0) to every variable in $X_1$, for every variable $X_i$ is AND (resp., OR), for every $i = 1, \ldots, r$. Then $\alpha$ is the partial truth assignment sensitizing $\varphi'$. Also, given a substitution $\sigma$, let $\varphi(\sigma)$ be the formula obtained by replacing each variable in $\varphi$ by the corresponding constant from $\sigma$. A subformula is constant if it computes a constant function. Maximal constant subformulas must be pairwise disjoint. Two substitutions $\sigma_1$ and $\sigma_2$ are equivalent if $\varphi(\sigma_1)$ and $\varphi(\sigma_2)$ compute the same Boolean function. Then it holds that substitutions $\sigma_1$ and $\sigma_2$ are equivalent if and only if their sets of maximal constant subformulas are identical.

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

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

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

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

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

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

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

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

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

$$MQ(y, \alpha) = \psi'(y),$$ (2)

where $\psi'$ is the subformula corresponding to $\varphi'$ in any representation of the target concept. Now we start considering the counterexample $x$, which we write as $(x_1, x_2)$, corresponding to its $\varphi'$-partition. By Equation 2, we can compare the known value of $\varphi'(x_1)$ to $\psi'(x_1)$ by asking the membership query $MQ(x_1, \alpha)$. There are two possibilities, and only one of them is different from [14].

Case II.1: MQ($x_1, \alpha) = \psi'(x_1) \neq \varphi'(x_1)$. Then $x_1$ is a counterexample to the hypothesis $\varphi'$ for the target concept $\psi'$. Thus we can continue recursively, to find a constant substitution in a problem which has at most two-thirds of the original variables. Note that by Equation 2 we can use the original membership queries to simulate membership queries to the new target concept.

Case II.2: MQ($x_1, \alpha) = \psi'(x_1) = \varphi'(x_1) = d$.

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

Let us write $\varphi$ as in Equation 1. Put $x_2 = (x_2, x_2, \ldots, x_2)$, where $x_2$ corresponds to the variables in $X_j$. Let $\psi$ be the subformula corresponding to $\varphi_1$ in some representation $\psi$ of the target. Let $y_i$ (resp., $z_i$) be the value computed at $x_i$ in $\varphi$ (resp., $\psi$) on input $x$, for $i = 1, \ldots, r$, and let $y_{r+1} = z_{r+1} = d$. Then by definition $y_1 = y_{i+1} z_1 \varphi'(x_2, i)$ and $z_1 = z_{i+1} \alpha_1 \psi'(x_2, i)$ for $i = 1, \ldots, r$. Also, $y_1 = \varphi(x) \neq \psi(x) = z_1$. Let $\beta_i$ be the partial truth assignment that assigns $x_{2,j}$ to $X_j$.
for \( j = i, \ldots, r \) and is otherwise identical to \( \alpha \). Then 
\[
z_i = \text{MQ}(x_i) \beta_i.
\]
As noted, \( y_{i+1} = z_{r+1} \) and \( y_i \neq z_i \). Just as the procedure \text{FindFormula}, the procedure \text{FindNewFormula} finds an index \( i \) (1 \( \leq i \leq r \)) such that \( y_{i+1} = z_{i+1} \) and \( y_i \neq z_i \), and we return \( i \). For this index \( i \), \( \varphi(x_i) \neq \psi(x_i) \). Thus we can continue by a recursive call on \( \varphi \) using the counterexample \( x_i \).

For a given \( i \), one can evaluate \( y_i \) without any membership queries from \( \varphi \), and one can use the remark at the end of the previous paragraph to evaluate \( z_i \) with a single membership query.

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

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

We also note that in order to simulate the membership queries in the recursive call by membership queries to the original target, one uses the following fact. Let \( \gamma_i \) be the partial truth assignment that assigns 1 (resp., 0) to \( Y_i \) and to all \( X_j \) with \( j > i \) if \( \alpha_i \) is AND (resp., OR) and is identical to \( \alpha \) on \( X_i \) for \( 1 \leq j < i \). Then for every truth assignment \( \text{w} \) to \( X_i \), it holds that
\[
\text{MQ}(\text{w}, \gamma_i) = \psi_i(\text{w}).
\]

We claim that \text{FindConstant} uses \( O(\log n) \) membership queries. There are three cases to consider. The procedure \text{GrowFormula} uses \( O(\log n) \) queries and does not make any recursive calls. If \text{FindConstant} gets into the \text{else} branch and it continues by looking at \( \varphi' \) then it uses a constant number of queries, and continues with a recursive call to an input that is at most \( O(1/2^n) \) times the original size. Hence the upper bound follows by induction. The rest of the description and analysis of the algorithm is again identical to [14] and so it is omitted.

**References**


**ACKNOWLEDGMENTS**

We want to thank Martin Mundhenk for conversations that forced us to focus on these constructions, and Andy Klapper and Maury Neiberg for putting up with us through the deadline crunch.
<table>
<thead>
<tr>
<th>FindConstant$(\varphi, x)$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>if $\varphi$ has one variable</td>
<td>return substitution $\sigma$ fixing it to constant $1 - x$</td>
</tr>
<tr>
<td>if $\text{MQ}(0) \equiv 1$ or $\text{MQ}(1) \equiv 0$</td>
<td>return substitution $\sigma$ fixing $\varphi$ to the appropriate constant</td>
</tr>
<tr>
<td>$\varphi' \equiv$ an approximately half-size formula of $\varphi$</td>
<td></td>
</tr>
<tr>
<td>$\alpha$ = the partial truth assignment sensitizing $\varphi'$</td>
<td></td>
</tr>
<tr>
<td>if $(\text{MQ}(0, \alpha) \equiv \text{MQ}(1, \alpha) \equiv c)$</td>
<td>then return GrowFormula$(\varphi, \varphi', c)$</td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>$(x_1, x_2) =$ the $\varphi'$-partition of $x$</td>
<td></td>
</tr>
<tr>
<td>if $\text{MQ}(x_1, \alpha) \equiv \varphi'(x_1)$</td>
<td>then FindConstant$(\varphi(-, \alpha), x_1)$ // look in $\varphi'$</td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>$i = \text{FindNewForm}(\varphi, \varphi', x)$</td>
<td></td>
</tr>
<tr>
<td>FindConstant$(\varphi_i, x_{2,i})$ // look in $\varphi_i$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: The procedure FindConstant.