Debugging a DAG Efficiently

Dana Angluin*, Yale University
YALEU/DCS/RR-591
December 1987

*Supported by the National Science Foundation, IRI-8404226

Debugging a DAG Efficiently

Dana Angluin *
Yale University

December 1987

Abstract

We formulate a debugging problem on directed acyclic graphs which is a special case of an incremental learning problem for propositional Horn sentences. There is a correct DAG G_*, and an initial DAG G_0, on the same (known) set of nodes V. G_0 must be modified to be transitively equivalent to G_* using information about G_* from two types of queries, equivalence queries and requests for hints. We demonstrate an algorithm that runs in time polynomial in $|V|$ and uses $O(\log |V|)$ queries per "incorrect" or "missing" edge in G_0. This query bound is shown to be optimal up to a multiplicative factor for graphs with $O(|V|^{2-\epsilon})$ edges.

1 Introduction

The theoretical aspects of learning have recently attracted a lot of attention; some of the current work is cited in the bibliography. In [3] we consider a problem of learning propositional Horn sentences using equivalence queries and requests for hints, and show that in the case of positive Horn sentences with two literals per clause, the problem can be cast as one of debugging "errors" in a directed graph. Here we give an efficient solution to the debugging problem in the case that the graphs are restricted to be acyclic. A directed acyclic graph is called a DAG. Our terminology and notation for directed graphs are fairly standard, see the references [12,14].

One special notation is the following. The assertion that there is a directed path from x to y of length at least zero in G is denoted $x \rightsquigarrow y$ *in* G. If this is not the case, we write $x \not\rightsquigarrow y$ *in* G.

If G_1 and G_2 are two directed graphs on the same set V of nodes, then G_1 is *transitively equivalent* to G_2 if and only if for all $x, y \in V$, $x \rightsquigarrow y$ in G_1 if and only if $x \rightsquigarrow y$ in G_2. That is, they are transitively equivalent if the same pairs of nodes are connected by directed paths in both graphs. The *size* of a directed graph G is the sum of the number of nodes and edges in G.

*Supported by NSF grant IRI-8404226

2 Formulation of the diagnosis problem for DAGs

Let V be a known set of nodes. There is an unknown DAG G_* on the nodes V. The input is another DAG G_0 on the nodes V, called the *initial graph*. The problem is to find a DAG on the nodes V that is transitively equivalent to G_* using the following two types of queries.

1. An equivalence query: propose a DAG G. If G is transitively equivalent to G_* the answer is "yes". Otherwise, the answer is "no", and a counterexample is provided, that is, an ordered pair of nodes (x, y) such that there is a path from x to y in G but not in G_* or vice versa.

2. A request for a hint: propose a pair of nodes (x, y). If there is no path from x to y in G_*, the answer is "no path". If (x, y) is an edge of G_*, the answer is "edge". If there is a path but no edge from x to y in G_*, the answer is a node z that is not equal to x or y but appears in some path from x to y in G_*.

Note that the choice of counterexample or hint is arbitrary; a debugging algorithm must work properly no matter which legal counterexample or hint is given in reply to a query.

One simple algorithm for this problem is to make a request for a hint with every pair (x, y) of nodes, record which ones are edges in G_*, and finally output a graph exactly equal to G_*. This approach necessarily uses $|V|(|V|-1)$ queries in the worst case, even if G_* is an extremely sparse graph. It also is not "incremental" in the sense that if the initial graph G_0 is "close" to G_*, that information is not used to reduce the work of finding a graph exactly equivalent to G_*.

3 A Whimsical Motivation

Suppose you are a member of a very traditional society of one million or so members. There is a high priest, who holds a position of great honor in the society, since he is the only individual who knows the genealogies of all the members of the society for the ten thousand generations that records have been kept. For various sacramental purposes, members of the society consult the high priest with questions of the form "Was Ansemarde an ancestor of Balacthon?". The answer, which costs dearly in the traditional currency of bat's ears, is either "no", or "parent", with the obvious interpretations, or is a third possibility. The third case is that "A." was an ancestor of "B.", and as a token of good faith, the high priest gives an intermediate ancestor, e.g., "Cullaforth", who was both a descendant of "A." and an ancestor of "B."

You aspire to usurp the high priest, and by a stroke of good luck and sound bribery, have acquired an early and somewhat incorrect draft of the entire genealogy graph. You proceed to set up a surreptitious cut-rate genealogy service using the old draft. Every so often, angry customers return to complain that some answer of yours differs from that of the high priest. You are compelled to return their fees and you lose some credibility with your other customers.

However, you have in effect a counterexample in answer to the query of whether your genealogy graph is equivalent to that of the high priest. Using the algorithm described below, some of your capital of bat's ears, and a network of people paid to ask questions of

the high priest, you proceed to remove one or more bugs in your copy of the genealogy. Note that his answers are precisely answers to requests for hints concerning the genealogy graph. See Littlestone's paper [23] for a formal treatment of the relationship between equivalence queries and errors of prediction.

4 The debugging algorithm

We give an algorithm that uses equivalence queries and request for hint queries to find incorrect and missing edges in the initial graph G_0 in $O(\log |V|)$ queries per edge found to be missing or incorrect. The algorithm is an optimized version of the incremental learning algorithm *IHL* in [3].

4.1 Incorrect and missing edges

Let G denote any DAG on the nodes V. An edge (x, y) is *incorrect* if there is no path from x to y in G_*. Let $i(G)$ denote the number of incorrect edges in G. The *correct part* of G, denoted $C(G)$, is the graph obtained from G by deleting all the incorrect edges. An edge (x, y) is defined to be *missing with respect to* G if $x \rightsquigarrow y$ in G_* but not in $C(G)$. Let $m(G)$ denote the number of edges in G_* that are missing with respect to G.

Our measure of how close any graph G on the nodes V is to the correct graph G_* is

$$d(G) = i(G) + m(G).$$

As an example, suppose $V = \{1, 2, 3, 4\}$, G_* has edges

$$\{(1,2), (1,3), (2,4), (3,4)\}$$

and G has edges

$$\{(1,2), (1,4), (2,3), (3,4)\}.$$

Since there is no path from 2 to 3 in G_*, the edge $(2,3)$ of G is incorrect. The remaining edges of G are correct, so $i(G) = 1$ and $C(G)$ has edges

$$\{(1,2), (1,4), (3,4)\}.$$

Since there is no path from 1 to 3 and no path from 2 to 4 in $C(G)$ (though there are such paths in G), the edges $(1,3)$ and $(2,4)$ of G_* are missing with respect to G. Thus $m(G) = 2$ and $d(G) = 3$.

Lemma 1 *G is transitively equivalent to G_* if and only if $d(G) = 0$.*

Suppose G is transitively equivalent to G_*. Then for every edge (x, y) in G, $x \rightsquigarrow y$ in G, so there are no incorrect edges in G and $i(G) = 0$. Thus $C(G)$ is equal to G. For every edge (x, y) in G_*, $x \rightsquigarrow y$ in G and therefore also in $C(G)$. Hence there are no edges of G_* missing with respect to G, so $m(G) = 0$ and $d(G) = 0$.

Conversely, suppose $d(G) = 0$, so $i(G) = 0$ and $m(G) = 0$. Since $i(G) = 0$, there are no incorrect edges in G, and for every edge (x, y) of G, $x \rightsquigarrow y$ in G_*. Since $m(G) = 0$, for every edge (x, y) of G_*, $x \rightsquigarrow y$ in $C(G)$ and hence in G. These imply that G and G_* are transitively equivalent. Q.E.D.

4.2 Description of subprocedures

The main procedure, *Debug*, is called with a DAG G_0 and uses equivalence queries and calls to subprocedures to find a DAG transitively equivalent to G_*. We first describe the subprocedures *Find-Path*, *Remove-Incorrect*, *Add-Edges*, and *Find-Missing*. There is a global variable G, called the *current graph*, which is initially equal to G_0 and is modified by the addition and deletion of edges to be transitively equivalent to G_*.

The procedure *Find-Path*(x, y)

The procedure *Find-Path* takes as input a pair of nodes x and y. If there is no path from x to y in the current graph G, the output is the special value *none*. Otherwise, the output is a directed path from x to y in G. A breadth-first or depth-first search of G starting from the node x returns a correct output in time linear in the number of edges of G.

The procedure *Remove-Incorrect*(x, y)

The procedure *Remove-Incorrect* takes as input a pair of nodes x and y such that $x \rightsquigarrow y$ in G and $x \not\rightsquigarrow y$ in G_*. It modifies G by removing one edge (x', y') of G that is incorrect.

The method is to call *Find-Path*(x, y), which returns a directed path $x_1, x_2, \ldots, x_n$ from $x = x_1$ to $y = x_n$ in G. At least one edge in this path is incorrect. If the path consists of one edge, this edge is removed from G. Otherwise, a binary search is used to locate an incorrect edge in the path, which is then removed.

In particular, let $m = \lceil n/2 \rceil$. Use a request for a hint to determine whether there is a path from x_1 to x_m in G_*. If not, recursively search the segment of the path from x_1 through x_m. If so, recursively search the segment of the path from x_m through x_n. This method runs in time linear in the size of G and makes at most $\lceil \log(n-1) \rceil$ queries, where $n \leq |V|$.

The procedure *Add-Edges*(E)

The procedure *Add-Edges* takes as input a set E of edges of G_*. It adds each edge in E to G after removing enough incorrect edges to be sure that no cycles are created in G.

1. For each edge (x, y) in E, do the following.
   (a) While $y \rightsquigarrow x$ in G, call *Remove-Incorrect*(y, x).
   (b) Add edge (x, y) to G if it is not already there.
2. When all edges in E have been processed, return.

Lemma 2 *If Add-Edges is called with a set E of edges of G_* and G is any DAG on the nodes V, then some incorrect edges may be removed from G and the edges in E will be in G when this procedure returns. The graph G will remain acyclic when this procedure returns. The procedure runs in time bounded by a polynomial in the size of G and $|E|$, and makes at most $O(\log |V|)$ queries per removed edge.*

If (x, y) is an edge of G_* then since G_* is acyclic, $y \not\rightsquigarrow x$ in G_*. Thus, *Remove-Incorrect* is called with correct input conditions, and will remove at least one incorrect edge from G. Since no incorrect edges can be added to G, step (1a) must eventually terminate.

No edge in E is incorrect, so no edge in E can be removed by *Remove-Incorrect*. Since every edge in E is added to G if it is not already present, G must contain all the edges in E when this procedure returns. If G is acyclic when this procedure is called, no cycles can be created by the added edges.

It is clear that the running time is bounded by a polynomial in the size of G and $|E|$. Since the only queries made are from the *Remove-Incorrect* routine, there is a bound of $O(\log |V|)$ queries per removed edge. Q.E.D.

The procedure *Find-Missing*(x, y)

The procedure *Find-Missing* takes as input a pair of nodes x and y such that $x \rightsquigarrow y$ in G_* but $x \not\rightsquigarrow y$ in G. It modifies G by deleting a set I of edges that are incorrect. It returns a set of edges M that are in G_* and that are missing with respect to G. The set M is non-empty. *Find-Missing* runs in time bounded by a polynomial in the size of G and makes $O((|M| + |I|) \log |V|)$ request for hint queries. The method is described and analyzed in the next section.

4.3 The main procedure

The procedure *Debug*(G_0)

The procedure *Debug* takes as input a DAG G_0 on the nodes V. It uses equivalence queries and calls to *Remove-Incorrect*, *Find-Missing*, and *Add-Edges* to find a graph that is transitively equivalent to G_*.

1. Initialize $G = G_0$.
2. Make an equivalence query with G. If the reply is "yes", output G and halt.
3. Otherwise, the reply contains a counterexample (x, y).
4. If $x \rightsquigarrow y$ in G, then call *Remove-Incorrect*(x, y), and go to step 2.
5. If $x \not\rightsquigarrow y$ in G, then let E be the set of edges returned by *Find-Missing*(x, y), call *Add-Edges*(E), and go to step 2.

We now analyze *Debug*, deferring discussion of the *Find-Missing* procedure to the next section.

Theorem 3 *If the procedure Debug is called with the DAG G_0 as input, its output is a DAG that is transitively equivalent to G_*. Moreover, it runs in time polynomial in the sizes of G_0 and G_* and makes at most $d(G_0) + 1$ equivalence queries and $O(d(G_0) \log |V|)$ requests for hints.*

G is initially the DAG G_0 on the nodes V. If G is a DAG when step (2) is executed, then the equivalence query returns a correct answer. Thus, if the reply is "yes", G is a DAG that is transitively equivalent to G_*, and it is correct to output G and halt.

Otherwise, the reply is a correct counterexample (x, y), and control continues with step (3). If $x \rightsquigarrow y$ in G then it must be that $x \not\rightsquigarrow y$ in G_*, so the call to *Remove-Incorrect* in step (4) has proper input conditions, so an incorrect edge will be removed from G. Then G will remain a DAG when control returns to step (2) from step (4).

If $x \not\rightsquigarrow y$ in G, then it must be that $x \rightsquigarrow y$ in G_*, so the call to *Find-Missing* in step (5) has proper input conditions. Then *Find-Missing* may delete some incorrect edges from G, and returns a nonempty set E of edges of G_* missing with respect to G. Thus G will remain a DAG after *Find-Missing* returns, and the call to *Add-Edges* has proper input conditions. Then *Add-Edges* may delete some incorrect edges from G, and adds the edges in E to G. When *Add-Edges* returns, G is guaranteed to be a DAG, so it remains a DAG when control returns to step (2).

Thus G is a DAG throughout the execution of the *Debug* procedure. Edges are only removed from G by the procedure *Remove-Incorrect*, and they must be incorrect edges. Edges are only added to G by the procedure *Add-Edges*, and they must be edges of G_* that are missing with respect to the current value of G. Such an edge is not incorrect and cannot be subsequently deleted from G.

Thus $i(G)$ is decreased by one for each edge deleted from G, and it is never increased. Since the correct part of G, $C(G)$, can only have edges added to it, an edge missing with respect to the current value of G must be missing with respect to G_0. When the edges in E are added to G by *Add-Edges*, $m(G)$ is decreased by at least $|E|$, and it is never increased. Recall that $|E|$ is at least one.

Every equivalence query in step (2) that is answered "no" causes either *Remove-Incorrect* or *Find-Missing* to be called. *Remove-Incorrect* decreases $i(G)$ by one and *Find-Missing* decreases $m(G)$ by at least one. Since G is transitively equivalent to G_* when $i(G)+m(G) = 0$, this can happen at most $d(G_0) = i(G_0) + m(G_0)$ times. Thus, there are a total of at most $d(G_0) + 1$ equivalence queries made before *Debug* halts with a correct answer.

There are at most $O(\log |V|)$ request for hint queries made for every edge deleted from or added to G, so there are a total of at most $O(d(G_0) \log |V|)$ request for hint queries made before *Debug* halts with a correct answer.

A straightforward implementation of *Debug* clearly runs in time polynomial in the sizes of G and G_*. Q.E.D.

5 The *Find-Missing* procedure

To complete the analysis of the *Debug* procedure, we must give a detailed description and analysis of the *Find-Missing procedure.*

5.1 Some motivation

Since the *Find-Missing* procedure is somewhat complicated, we motivate its design by first considering the following straightforward but inefficient procedure.

Given the pair (x, y) such that $x \rightsquigarrow y$ in G_* but not in G, make a request for a hint with (x, y). If the reply is "edge", we have found an edge of G_* missing with respect to G. Otherwise, the reply will be a hint z. If $x \not\rightsquigarrow z$ in G then we iterate with (x, z), otherwise, we iterate with (z, y). Because G_* is acyclic, this process must find an edge of G_* that is missing with respect to G after at most $|V| - 1$ requests for hints.

One case in which this simple procedure is needlessly inefficient is the following. Suppose V is the set of nodes numbered 1 to $n+1$, and G is the graph with an edge from i to $i+1$ for $i = 1, \ldots, n-1$. Suppose G_* is equal to G with one additional edge, from some $i \leq n$ to the node $n+1$. An equivalence query with G might return the pair $(1, n+1)$. Successive requests for hints might return the intermediate nodes $2, \ldots, n$, until finally a query with $(n, n+1)$ yields the reply "edge". Here n queries have been used to find just one missing edge.

One obvious improvement in this particular case is to do a binary search. Let $m = \lceil n/2 \rceil$ and make a request for a hint with $(m, n+1)$. If the answer is that there is a path, then we restrict further queries to nodes m through n. If there is no path, then we restrict further queries to nodes 1 through $m-1$. Continuing recursively, an edge of G_* that is missing from G will be discovered after $O(\log n)$ queries.

We can generalize this idea to do a kind of binary search in single-source or single-sink DAGs. However, there are two ways that this approach can break down. One of them is there may be no node to divide the graph nearly in half. In this case, it turns out that the value of the hint allows us to discard about half the remaining nodes. Another is that incorrect edges in the current graph may cause us erroneously to discard nodes from consideration. In this case, the *Remove-Incorrect* procedure is invoked to delete an incorrect edge from G, and the process is restarted.

5.2 Critical nodes

We need a little graph-theoretic machinery at this point. Let G be a DAG on a subset of the nodes V. If x is any node, let $R^1_G(x)$ denote the set of nodes y such that (x, y) is an edge of G. Let $R^*_G(x)$ denote the set of nodes y such that $x \rightsquigarrow y$ in G. Similarly, let $P^1_G(x)$ denote the set of nodes such that (y, x) is an edge of G, and let $P^*_G(x)$ denote the set of nodes y such $y \rightsquigarrow x$ in G.

Define the *R-weight* of the node x in G, denoted $w^R_G(x)$, to be $|R^*_G(x)|$, the cardinality of the set of nodes reachable from x in G. Define the *P-weight* of the nodes x, denoted $w^P_G(x)$, to be $|P^*_G(x)|$.

Suppose now that the DAG G has a single source node, x_0. Let n denote the R-weight of x_0 in G, that is, the number of nodes in G. We define a node x of G to be *R-critical in* G if and only if $w^R_G(x) \geq n/2$, and for each $y \in R^1_G(x)$, $w^R_G(y) < n/2$. That is, the R-weight of x is at least half of that of x_0 but this is not true for any of the immediate successors of x in G. The definition of P-critical in a DAG with a single sink is analogous.

Lemma 4 *If G is any DAG with a single source node then there exists an R-critical node in G. Moreover, some R-critical node in G can be found in time linear in the size of G.*

Let the single source of G be x_0 and let $n = w^R_G(x_0)$. Imagine the nodes in G each labelled by its R-weight. From x_0 construct a directed path in G by successively selecting

any node y with R-weight at least $n/2$. This path will terminate in a node x of weight at least $n/2$ such that for every $y \in R^1(x)$, the R-weight of y is less than $n/2$. (Note this is vacuously true if x has no out edges.) Hence x is an R-critical node.

It is clear that we can compute the R-weights of all the nodes in time linear in the size of G by processing them in reverse topological order. Then an $O(|V|)$ search suffices to find an appropriate x. Q.E.D.

The analogous lemma for the predecessor relation is proved similarly.

Lemma 5 *If G is any DAG with a single sink node then there exists a P-critical node in G. Moreover, some P-critical node in G can be found in time linear in the size of G.*

5.3 A sketch of *Find-Missing*

Given nodes x and y such that $x \rightsquigarrow y$ in G_* but not in G, we attempt to find an R-critical node x' in the subgraph of G reachable from x and an P-critical node y' in the subgraph of G from which y is reachable such that $x' \rightsquigarrow y'$ in G_*. If we succeed in finding such a pair and make a request for a hint with x' and y', the answer may be "edge", in which case we have found a missing edge (x', y') which can be returned.

Otherwise, the answer is a hint z. If $x' \not\rightsquigarrow z$ and $z \not\rightsquigarrow y'$ in G, then we can recursively call *Find-Missing* on the pairs (x', z) and (z, y'), and be guaranteed that at least two distinct missing edges will be returned. If, however, $x' \rightsquigarrow z$ in G, then we iterate, replacing x by z. Since x' is a R-critical node, we have eliminated at least half the nodes that were reachable from x from consideration. Similarly if $z \rightsquigarrow y'$ in G.

To find the pair x' and y', we first find an R-critical node x''. A request for a hint is used to test whether $x'' \rightsquigarrow y$ in G_*. If so, we let x' be x'' and begin the search for y'. Otherwise, we remove the nodes reachable from x'' in G from consideration, and continue. Since x'' is R-critical, this is at least half the remaining node reachable from x. Once x' is found, the search for y' is similar.

One further complication is that incorrect edges in G may disrupt the progress of the algorithm described above, so checks must be included to detect this situation, and *Remove-Incorrect* must be called to get rid of the edges causing problems.

5.4 Detailed description of *Find-Missing*

The input nodes are x and y, and G is the current graph. X and Y are subsets of the nodes, and x', y', and z are individual nodes. G, X, Y, x', y', and z are modified as the algorithm progresses, but x and y always denote the input nodes.

As the algorithm progresses, nodes may be removed from X or Y. When a node v is removed from X or Y in step (3) or (5), we associate with it a node, $r(v)$, which is the node "responsible" for the removal of v. The initial value of $r(v)$ is $\perp$.

We use the notation G/X to denote the subgraph of G induced by the set of nodes X, and similarly for G/Y. It will be shown that the graph G/X has a single source, denoted $source(G/X)$, and G/Y has a single sink, denoted $sink(G/Y)$.

The procedure *Find-Missing*(x, y)

1. Let $X = R^*_G(x)$ and $Y = P^*_G(y)$. Also, let $r(v) = \perp$ for all $v \in V$.
2. Let x' be an R-critical node in G/X. Make a request for a hint with $(source(G/X), x')$. If the reply is "no path", call *Remove-Incorrect*$(source(G/X), x')$, and go to step 1.
3. Make a request for a hint with $(x', sink(G/Y))$. If the reply is "no path", set $r(v) = x'$ for each $v \in R^*_{G/X}(x')$, set $X = X - R^*_{G/X}(x')$, and go to step 2.
4. Let y' be a P-critical node in G/Y. Make a request for a hint with $(y', sink(G/Y))$. If the reply is "no path", call *Remove-Incorrect*$(y', sink(G/Y))$, and go to step 1.
5. Make a request for a hint with (x', y').
   (a) If the reply is "no path", set $r(v) = y'$ for each $v \in P^*_{G/Y}(y')$, set $Y = Y - P^*_{G/Y}(y')$, and go to step 4.
   (b) If the reply is "edge", then return $\{(x', y')\}$.
   (c) Otherwise, the reply is a hint z.
6. If $x' \rightsquigarrow z$ in G/X, then let $X = R^*_{G/X}(z)$ and go to step 2. If $z \rightsquigarrow y'$ in G/Y, then let $Y = P^*_{G/Y}(z)$ and go to step 2.
7. If $x' \rightsquigarrow z$ in G but $x' \not\rightsquigarrow z$ in G/X, then call *Remove-Incorrect*$(r(z), z)$ and go to step 1. If $z \rightsquigarrow y'$ in G but $z \not\rightsquigarrow y'$ in G/Y, then call *Remove-Incorrect*$(z, r(z))$ and go to step 1.
8. Return the union of the sets returned by the recursive calls *Find-Missing*(x', z) and *Find-Missing*(z, y').

5.5 Analysis of *Find-Missing*

This section is devoted to a proof of the following lemma.

Lemma 6 *Suppose that when Find-Missing is called the current graph G is a DAG on the nodes V and the input nodes x and y are such that $x \rightsquigarrow y$ in G_* and $x \not\rightsquigarrow y$ in G. Let G_1 denote the value of G when Find-Missing is called. Find-Missing may modify G by deleting a set I of incorrect edges. It returns a set M of edges of G_* missing with respect to G_1. Each edge in M is on some path from x to y in G_*. It makes at most $O((|I| + |M|) \log |V|)$ requests for hints, and can be implemented to run in time bounded by a polynomial in the sizes of G_1 and G_*.*

The following correctness conditions will be useful.

Correctness conditions

1. The graph G is obtained from G_1 by deleting some (possibly empty) set of incorrect edges.

2. X and Y are disjoint subsets of V such that G/X has a single source, denoted $source(G/X)$, and G/Y has a single sink, denoted $sink(G/Y)$.

3. $x \rightsquigarrow source(G/X)$ and $sink(G/Y) \rightsquigarrow y$ in G and in G_*.

4. $source(G/X) \rightsquigarrow sink(G/Y)$ in G_* but $source(G/X) \not\rightsquigarrow sink(G/Y)$ in G.

5. If $x_1, \ldots, x_s$ is the sequence of distinct values taken on by $source(X)$ and $y_1, \ldots, y_t$ is the sequence of distinct values taken on by $sink(G/Y)$ since step (1) was last executed, then both in G and in G_*, $x_i \rightsquigarrow x_{i+1}$ for $i = 1, \ldots, s-1$ and $y_{j+1} \rightsquigarrow y_j$ for $j = 1, \ldots, t-1$.

6. If for some v and w, $source(G/X) \rightsquigarrow w$ in G/X and $w \rightsquigarrow v$ in G but not in G/X then $v \notin X$, $r(v) \neq \perp$, and $r(v) \rightsquigarrow v$ in G. Similarly, if for some v and w, $w \rightsquigarrow sink(G/Y)$ in G/Y and $v \rightsquigarrow w$ in in G but not in G/Y then $v \notin Y$, $r(v) \neq \perp$, and $v \rightsquigarrow r(v)$ in G.

7. The node x' is an R-critical node in G/X such that $source(G/X) \rightsquigarrow x'$ in G_*.

8. $x' \rightsquigarrow sink(G/Y)$ in G_*.

9. The node y' is a P-critical node in G/Y such that $y' \rightsquigarrow sink(G/Y)$ in G_*.

10. $x' \rightsquigarrow z$ and $z \rightsquigarrow y'$ in G_*.

11. $x' \not\rightsquigarrow z$ in G/X and $z \not\rightsquigarrow y'$ in G/Y.

12. $x' \not\rightsquigarrow z$ and $z \not\rightsquigarrow y'$ in G.

We begin by proving the following lemma about these correctness conditions.

Lemma 7 *Suppose the graph G_1 is a DAG when Find-Missing is called, and the inputs are nodes x and y such that $x \rightsquigarrow y$ in G_* but not in G. If we consider the execution of Find-Missing until either a return statement or a recursive call is executed, the following correctness conditions are true.*

1. *At the start of step (1): condition (1) holds,*
2. *at the start of step (2): conditions (1) – (6) hold,*
3. *at the start of step (3): conditions (1) – (7) hold,*
4. *at the start of step (4): conditions (1) – (8) hold,*
5. *at the start of step (5): conditions (1) – (9) hold,*
6. *at the start of step (6): conditions (1) – (10) hold,*
7. *at the start of step (7): conditions (1) – (11) hold,*
8. *at the start of step (8): conditions (1) – (12) hold.*

The proof is tedious; we divide it into one lemma per step.

Lemma 8 *If condition (1) holds before the execution of step (1), then conditions (1) – (6) hold after it is executed.*

Suppose condition (1) holds before step (1) is executed. Since G is unchanged by step (1), condition (1) must still hold.

To see that condition (2) holds, note that X is the set of nodes reachable from x in G, and Y is the set of nodes reaching y in G. Since G is a subgraph of G_1, $x \not\rightsquigarrow y$ in G, so X and Y are disjoint, G/X has the single source x and G/Y has the single sink y.

To verify condition (3), we note that $x \rightsquigarrow x$ and $y \rightsquigarrow y$ in G and G_*. By the input conditions, $x \rightsquigarrow y$ in G_*, and since $x \not\rightsquigarrow y$ in G, condition (4) is verified.

Condition (5) is vacuously satisfied, since step (1) just terminated, and $source(G/X)$ and $sink(G/Y)$ have taken on only one value. Condition (6) is also satisfied, since for every v and w such that $source(G/X) \rightsquigarrow w$ in G/X and $w \rightsquigarrow v$ in G, we have $w \rightsquigarrow v$ in G/X, and similarly for G/Y. Q.E.D.

Lemma 9 *If conditions (1) – (6) hold before step (2) is executed, then either control is transferred to step (1) and condition (1) is true, or control is transferred to step (3) and conditions (1) – (7) are true.*

Suppose conditions (1) – (6) hold when step (2) is executed. Then G/X has a single source, so there must be an R-critical node x' in G/X. Clearly $source(G/X) \rightsquigarrow x'$ in G.

If the query discloses that $source(G/X) \not\rightsquigarrow x'$ in G_*, the call to *Remove-Incorrect* has its input conditions satisfied, so an incorrect edge in G will be removed. Condition (1) is preserved by this change to G, and control is then transferred to step (1).

If the query indicates that $source(G/X) \rightsquigarrow x'$ in G_* then G, X, Y, and $r(v)$ for all v are unchanged by this step, and the conditions (1) – (6) must still hold after it is executed. Moreover, condition (7) is now true, since x' is an R-critical node in G/X such that $source(G/X) \rightsquigarrow x'$ in G_*. In this case, control is transferred to step (3). Q.E.D.

Lemma 10 *If conditions (1) – (7) hold before step (3) is executed, then either control is transferred to step (2) and conditions (1) – (6) are true, or control is transferred to step (4) and conditions (1) – (8) are true.*

Suppose conditions (1) – (7) hold when step (3) is executed.

If the query in step (3) indicates that there is a path from x' to $sink(G/Y)$ in G_*, then G, X, Y, x', and $r(v)$ for all $v \in V$ are unchanged, so conditions (1) – (7) are preserved in this case. Moreover, condition (8) is now true, since $x' \rightsquigarrow sink(G/Y)$ in G_*. In this case, control is transferred to step (4).

If the query in step (3) indicates that $x' \not\rightsquigarrow sink(G/Y)$ in G_*, then X is modified by removing all the nodes reachable from x' in G/X. Let X_1 denote the value of X before these nodes are removed, and let X_2 denote the value of X after these nodes are removed. Since by condition (4), $source(G/X_1) \rightsquigarrow sink(G/Y)$ in G_*, x' is some node of G/X_1 other than the source. Thus, G/X_2 has the same single source as G/X_1 and G and Y are unchanged, so conditions (1) – (5) hold in this case.

To see that condition (6) holds, consider any nodes v and w such that $source(G/X_2) \rightsquigarrow w$ in G/X_2 and $w \rightsquigarrow v$ in G but not in G/X_2. Then $source(G/X_1) \rightsquigarrow w$ in G/X_1.

If $w \not\rightsquigarrow v$ in G/X_1, then at the start of this step, since condition (6) holds, v is not in X_1, $r(v) \neq \perp$, and $r(v) \rightsquigarrow v$ in G. Since $v \notin X_1$, the value of $r(v)$ is unchanged by this step. Since G is not modified and X_2 is a subset of X_1, then after this step is executed, $v \notin X_2$, $r(v) \neq \perp$, and $r(v) \rightsquigarrow v$ in G. Thus in this case, condition (6) holds at the end of this step.

If, however, $w \rightsquigarrow v$ in G/X_1, then some node on a path from w to v in G/X_1 must be removed in computing X_2. This node must be reachable from x' in G/X_1, so v is reachable from x' in G/X_1. Thus, v is one of the nodes removed at this step, so $v \notin X_2$, and $r(v) = x' \neq \perp$. Since G is unchanged, $r(v) \rightsquigarrow v$ in G. Thus in this case also, condition (6) is preserved. Hence when control is transferred to step (2), conditions (1) – (6) hold. Q.E.D.

Lemma 11 *If conditions (1) – (8) are true before step (4) is executed, then either control is transferred to step (1) and condition (1) is true, or control is transferred to step (5) and conditions (1) – (9) are true.*

Suppose conditions (1) – (8) hold when step (4) is executed. Then G/Y has a single sink, so there is a P-critical node y'. Clearly $y' \rightsquigarrow sink(G/Y)$ in G.

Hence if the query in step (4) indicates that $y' \not\rightsquigarrow sink(G/Y)$ in G_*, the call to *Remove-Incorrect* has its input conditions satisfied. In this case, an incorrect edge will be removed from G, which preserves condition (1), and control will be transferred to step (1).

If the query indicates that $y' \rightsquigarrow sink(G/Y)$ in G_*, then G, X, Y, x', and $r(v)$ for all $v \in V$ are unchanged, so conditions (1) – (8) are preserved. Moreover, condition (9) is now true, since y' is a P-critical node in G/Y such that $y' \rightsquigarrow sink(G/Y)$ in G_*. Thus, conditions (1) – (9) hold when control is transferred to step (5). Q.E.D.

Lemma 12 *If conditions (1) – (9) hold when step (5) is executed, then either control is transferred to step (4) and conditions (1) – (8) are true, or a return statement is executed, or control is transferred to step (6) and conditions (1) – (10) are true.*

Suppose conditions (1) – (9) hold when step (5) is executed.

If the query with (x', y') indicates that $x' \not\rightsquigarrow y'$ in G_*, then by condition (8), $x' \rightsquigarrow sink(G/Y)$, so y' must be distinct from $sink(G/Y)$. In this case, the nodes reaching y' in G/Y are removed from Y, so G/Y still has the same single sink. The argument that condition (6) is preserved by this change is analogous to that in Lemma 10. Since G, X, $sink(G/Y)$, and x' are unchanged, conditions (1) – (8) are preserved, and control is transferred to step (4).

If the query indicates that (x', y') is an edge of G_*, then a return statement is executed.

Otherwise, the reply to the query must be a hint z such that $x' \rightsquigarrow z$ in G_* and $z \rightsquigarrow y'$ in G_*. Since G, X, Y, x', y', and $r(v)$ for all $v \in V$ are unchanged, conditions (1) – (9) are preserved. Moreover, condition (10) is also true, since $x' \rightsquigarrow z$ and $z \rightsquigarrow y'$ in G_*. Thus, conditions (1) – (10) are true when control is transferred to step (6). Q.E.D.

Lemma 13 *If conditions (1) – (10) are true when step (6) is executed then either control is transferred to step (2) and conditions (1) – (6) are true, or control is transferred to step (7) and conditions (1) – (11) are true.*

Assume conditions (1) – (10) hold when step (6) is executed. If neither $x' \rightsquigarrow z$ in G/X nor $z \rightsquigarrow y'$ in G/Y, then step (6) leaves G, X, Y, x', y', z, and $r(v)$ for each $v \in V$ unchanged, so conditions (1) – (10) are preserved. Moreover, condition (11) is now true, since $x' \not\rightsquigarrow z$ in G/X and $z \not\rightsquigarrow y'$ in G/Y. Thus, in this case control is transferred to step (7) with conditions (1) – (11) true.

Suppose $x' \rightsquigarrow z$ in G/X. In this case we must show that conditions (1) – (6) are preserved when control is transferred to step (2). Let X_1 denote the value of X at the start of this step, and let $X_2 = R^*_{G/X_1}(z)$, the nodes reachable from z in G/X_1. Clearly X_2 is a subset of X_1. We must show that conditions (1) – (6) are true when we change the value of X from X_1 to X_2.

Clearly G is unchanged by this action, so condition (1) is preserved. X_2 is a subset of X_1 and so is disjoint from Y. The graph G/X_2 has a single source, z. Thus setting X to X_2 preserves condition (2).

Since by condition (3), $x \rightsquigarrow \mathit{source}(G/X_1)$ in G and in G_*, and by condition (7), $\mathit{source}(G/X_1) \rightsquigarrow x'$ in G and in G_*, and $x' \rightsquigarrow z$ in G and in G_* by the assumptions of this case, $x \rightsquigarrow z$ in G and in G_* in this case, so condition (3) is preserved if X is set to X_2.

Since $z \rightsquigarrow y'$ in G_* and by condition (9) $y' \rightsquigarrow \mathit{sink}(G/Y)$ in G_*, $z \rightsquigarrow \mathit{sink}(G/Y)$ in G_*. Moreover, since $x \rightsquigarrow z$ in G and $\mathit{sink}(G/Y) \rightsquigarrow y$ in G, $z \not\rightsquigarrow \mathit{sink}(G/Y)$ in G, for otherwise $x \rightsquigarrow y$ in G, violating the input conditions and condition (1). Thus, condition (4) is preserved if X is set to X_2.

When X is set to X_2, z will become a new distinct value x_{i+1}, in the sequence of values of $\mathit{source}(G/X)$ since step (1) was last executed. The previous value, x_i, is just $\mathit{source}(G/X_1)$, and we have shown above that $\mathit{source}(G/X_1) \rightsquigarrow z$ in G and in G_*. Thus condition (5) is preserved when X is set to X_2.

Finally, to see that condition (6) is preserved, suppose that v and w are nodes such that $\mathit{source}(G/X_2) \rightsquigarrow w$ in G/X_2 and $w \rightsquigarrow v$ in G but not in G/X_2. Then since $\mathit{source}(G/X_1) \rightsquigarrow \mathit{source}(G/X_2)$ in G/X_1, $\mathit{source}(G/X_1) \rightsquigarrow w$ in G/X_1.

If $w \not\rightsquigarrow v$ in G/X_1, then because condition (6) holds at the start of this step, and G and the value of $r(v)$ are unchanged, $v \notin X_1$, $r(v) \neq \perp$, and $r(v) \rightsquigarrow v$ in G. Thus, $v \notin X_2$, and condition (6) is preserved in this case when X is set to X_2.

However, suppose $w \rightsquigarrow v$ in G/X_1. Since $\mathit{source}(G/X_2) \rightsquigarrow w$ in G/X_2 and therefore also in G/X_1, and $\mathit{source}(G/X_2) = z$, $z \rightsquigarrow w$ in G/X_1. Since w is reachable from z in G/X_1, every node along the path in G/X_1 from w to v is likewise reachable from z in G/X_1, so this path must also be in G/X_2. Thus, $w \rightsquigarrow v$ in G/X_2, contradicting the hypotheses of this case.

Thus, if $x' \rightsquigarrow z$ in G/X, conditions (1) – (6) are preserved when X is replaced by $R^*_{G/X}(z)$ and control is transferred to step (2).

The argument is exactly analogous to show that conditions (1) – (6) are preserved if $z \rightsquigarrow y'$ in G/Y, Y is set to $P^*_{G/Y}(z)$, and control is transferred to step (2). Q.E.D.

6.2 The case of any $\epsilon \in (0,1)$

Let $0 < \epsilon < 1$. The generalization for graphs of $O(n^{2-\epsilon})$ edges is now sketched. Let n be any positive integer. Let $L = \lceil n^{\epsilon} \rceil$, and $N = \lceil n^{1-\epsilon} \rceil$. The base graph $G_{n,\epsilon}$ consists of $2LN$ nodes. Note that $2LN$ is certainly bounded by $6n + 2$.

The first LN nodes are organized into N groups of L consecutive nodes each. The r^{th} group is V_r and consists of the nodes $(r-1)L+1$ through rL. Within each group, the nodes are connected sequentially, that is, there is an edge from node i to nodes $i+1$. However there are no edges between nodes in different groups. Clearly this graph has $N(L-1)$ edges.

From this base graph we obtain a family of graphs $\mathcal{C}_{n,\epsilon}$ as follows. For $r = 1, \ldots, N$ and $s = 1, \ldots, NL$ let $i_{r,s}$ be a node in the group V_r. Then the graph $G_{i_{r,s}}$ is obtained from $G_{n,\epsilon}$ by adding for each $r = 1, \ldots, N$, and for each $s = 1, \ldots, NL$, the edge from node $i_{r,s}$ to node $NL + s$. The resulting graph has $N(L-1) + N^2L$ edges.

An argument similar to the one given above shows that any algorithm using equivalence queries and requests for hints must essentially make a separate binary search in a group of L nodes to discover each of N^2L missing edges. Thus $N^2L\lceil \log L \rceil$ queries will be required in the worst case. Since the total number of missing edges is N^2L, the number of queries required per missing edge is at least $\lceil \log L \rceil$, which is at least $\lceil \epsilon \log n \rceil$ queries. In terms of the number of nodes, this is at least $\epsilon \log |V| - O(1)$ queries per missing edge in the worst case.

7 Comments

The case of debugging a non-acyclic directed graph is open. For some of the pitfalls, see the examples given in [3].

The *Find-Missing* procedure arrived at its current form through several iterations of simpler but incorrect versions. The procedure is not terribly complex, but its detailed proof of correctness is. I cannot help thinking there must be a cleaner and more simply analyzed version of *Find-Missing*, but I have been unable to find it.

8 Acknowledgements

The support of the National Science Foundation, grant IRI-8404226, is gratefully acknowledged.

References

[1] D. Angluin. *Learning k-bounded context-free grammars.* Technical Report, Yale University Computer Science Dept., TR-557, 1987.

[2] D. Angluin. *Learning k-term DNF formulas using queries and counterexamples.* Technical Report, Yale University Computer Science Dept., TR-559, 1987.

[3] D. Angluin. *Learning propositional Horn sentences with hints.* Technical Report, Yale University Computer Science Dept., TR-590, 1987.

[4] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987. Preliminary version appeared as YALEU/DCS/RR-464.

[5] D. Angluin. *Types of queries for concept learning.* Technical Report, Yale University Computer Science Dept., TR-479, 1986. Revised version, *Queries and concept learning*, submitted for publication.

[6] D. Angluin, W. Gasarch, and C. Smith. *Training sequences.* Technical Report, University of Maryland, CS-TR-1894, UMIACS-TR-87-37, 1987. Submitted for publication.

[7] D. Angluin and P. Laird. *Identifying k-CNF formulas from noisy examples.* Technical Report, Yale University Computer Science Dept., TR-478, 1986. Revised version, *Learning from noisy examples*, to appear in *Machine Learning.*

[8] P. Berman and R. Roos. Learning one-counter languages in polynomial time. In *Proc. 28th IEEE Symposium on Foundations of Computer Science*, pages 61–67, IEEE, 1987.

[9] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Classifying learnable geometric concepts with the Vapnik-Chervonenkis dimension. In *Proc. 18th ACM Symposium on Theory of Computing*, pages 273–282, ACM, 1986.

[10] J. Cherniavsky and C. Smith. *Using telltales in developing program test sets.* Technical Report, Georgetown University, Dept. of Computer Science, TR-4, 1986.

[11] J. Cherniavsky and R. Statman. Testing and inductive inference: abstract approaches. 1987. Preprint, Georgetown University.

[12] S. Even. *Graph Algorithms.* Computer Science Press, 1979.

[13] U. Feige and A. Shamir. Learning in permutation groups (extended abstract). 1987. Preprint, Applied Mathematics Dept., The Weizmann Institute of Science.

[14] F. Harary. *Graph Theory.* Addison-Wesley Publishing Company, 1969.

[15] D. Haussler. *Learning conjunctive concepts in structural domains.* Technical Report, University of California at Santa Cruz, Dept. of Computer Science, UCSC-CRL-87-1, 1987.

[16] D. Haussler. *Quantifying inductive bias in concept learning.* Technical Report, University of California at Santa Cruz, Dept. of Computer Science, UCSC-CRL-86-25, 1986.

[17] D. Haussler, N. Littlestone, and M. Warmuth. Expected mistake bounds for on-line learning algorithms. Preprint, University of California at Santa Cruz, April 1987.

[18] M. Kearns and M. Li. *Learning in the presence of malicious errors.* Technical Report, Harvard University, Center for Research in Computing Technology, TR-03-87, 1987.

[19] M. Kearns, M. Li, L. Pitt, and L. Valiant. On the learnability of boolean formulae. In *Proc. 19th ACM Symposium on Theory of Computing*, pages 285–295, ACM, 1987.

[20] K. Kelly and C. Glymour. *On convergence to the truth and nothing but the truth.* Technical Report, Carnegie Mellon University, Laboratory for Computational Linguistics, CMU-LCL-87-4, 1987.

[21] P. Laird. Inductive inference by refinement. In *Proc. of AAAI-86*, pages 472–476, AAAI, 1986.

[22] P. Laird. *Learning From Good Data and Bad.* PhD thesis, Yale University, 1987. Computer Science Dept. TR-551.

[23] N. Littlestone. Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm. In *Proc. 28th IEEE Symposium on Foundations of Computer Science*, pages 68–77, IEEE, 1987.

[24] B. K. Natarajan. On learning boolean functions. In *Proc. 19th ACM Symposium on Theory of Computing*, pages 296–304, ACM, 1987.

[25] L. Pitt and L. Valiant. *Computational limitations on learning from examples.* Technical Report, Harvard University, Center for Research in Computing Technology, TR-05-86, 1986.

[26] L. Pitt and M. Warmuth. Reductions among prediction problems: on the difficulty of predicting automata (extended abstract). 1987. Preprint.

[27] R. Rivest and R. Schapire. Diversity-based inference of finite automata. In *Proc. 28th IEEE Symposium on Foundations of Computer Science*, pages 78–87, IEEE, 1987.

[28] R. Rivest and R. Schapire. Inference of visible simple assignment automata with planned experiments. 1987. Preprint, MIT Laboratory for Computer Science.

[29] R. Rivest and R. Schapire. A new approach to unsupervised learning in deterministic environments. In *Proc. of the 4th International Workshop on Machine Learning*, pages 364–375, Morgan Kaufmann Publishers, Inc., 1987.

[30] S. Rudich. Inferring the structure of a Markov chain from its output. In *Proc. 26th IEEE Symposium on Foundations of Computer Science*, pages 321–326, IEEE, 1985.

[31] C. Sammut and R. Banerji. Learning concepts by asking questions. In *Machine Learning, Vol. II*, pages 167–191, Morgan Kaufmann Publishers, Inc., 1986.

[32] L. Valiant. Learning disjunctions of conjunctions. In *Proc. 9th IJCAI*, pages 560–566, IJCAI, 1985.

[33] L. G. Valiant. A theory of the learnable. *C. ACM*, 27:1134–1142, 1984.