ON EVALUATING BOOLEAN EXPRESSIONS

by

E. Gudes and A. Reiter

Technical Report #18

June 1972
ABSTRACT. An evaluation algorithm for Boolean expressions is efficient if it recognizes when particular conditions cannot affect the value of the result. Of special interest are efficient algorithms which do not expect the conditions to be evaluated in the order in which they appear in the expression. This is important for selective retrieval from a large data base, when the evaluation (retrieval) order depends on the data organization and not on the order in which the qualifiers appear in the query. Another aspect of data retrieval is that we may repeatedly change part (but not all) of the values of the variables, and wish to reevaluate part of the expression. Algorithms are given for representing and efficiently evaluating Boolean expressions both for the sequential and random cases; for the latter, an algorithm for partial reinitialization is also given.

Computing Review Categories: 3-74, 4-12

Key Words and Phrases: Boolean expressions, data base systems, information retrieval, optimization, query languages, searches, trees.
INTRODUCTION

We are concerned in this paper with good ways of evaluating Boolean expressions; i.e., a collection of conditions each of which is either TRUE or FALSE and which are connected into a well-formed combination by AND/OR operators*. The conditions themselves are denoted by \{X_i\}, and for brevity we shall denote the operators by * and +.

The basic problem is finding some convenient way of determining automatically, that the evaluation of a given \(X_i\) cannot affect the outcome of the result and can be skipped. The problem is not new, and various algorithms for optimizing such expressions have been used in compilers. The reader is, in particular, referred to Gries [1], Section 13.6. We are, however, interested in a variant of the problem, which comes about as follows.

In writing compilers, it is commonly assumed that all variables are located in the fast memory during execution, and, hence, that all are equally accessible. Without additional information as to the probabilities of the various conditions \(X_i\) being TRUE or FALSE, it is usual to evaluate the expression in the order in which it is written. In some cases, however, notably for the execution of queries over large data bases, evaluating a

* NOT operators cause no special problems and are not discussed here.
condition frequently means accessing a peripheral store, such as a rotating disc. Depending upon the particular data organization in the store (which need have no connection with the structure of the Boolean expression), at a given point during the evaluation of the expression the most convenient \( X_i \) to evaluate need not be the next one in the expression. In general, we are given some selection function \( f \) which (based on the data organization and on the query) determines the \( X_i \) to evaluate next. Such a function for a particular generalized data organization is, in effect, described in Reiter et al [2]. Our task here is to develop some tools which enable us to skip evaluation of the selected \( X_i \) in case its value cannot affect the result.

Existing systems for querying data are often based on an inverted file concept (see e.g. Fox and Edwards [3], Summit [4], Wong and Chiang [5].) In such data organizations, associated with each attribute value is a set of pointers to records containing the attribute value. Boolean operators in such systems are executed as set intersection and set union operations, and entirely different optimization problems arise. Here, however, we are concerned with systems which fetch all or parts of a record to verify directly a given qualifying condition.

We shall refer to the act of evaluating \( X_i \) as "lookup" emphasizing the fact that we are primarily concerned with avoiding unnecessary accesses to the peripheral store. At times, a fair amount of CPU work has to be
invested to avoid an access; however, since the CPU is several orders of magnitude faster than the disc, we shall assume that the investment of CPU time is worthwhile.

In practice, of course, not every $X_i$ would require an access; usually, a record read into core would contain the values of several relevant data items. We are not explicitly concerned with these considerations here, but assume that the selection function $f$ makes use of such facts.

In section 2, we introduce a table representation of a Boolean expression, prove some results about its properties, and describe an algorithm for sequential evaluation of the table. This algorithm is a variant of well-known techniques. In the following sections, we represent the expression as a conventional tree, and define the random evaluation algorithm.
2. THE TABLE $P_E$ FOR SEQUENTIAL EVALUATION

We shall assume that each $X_i$ appears only once, and that the $i$'s appear in ascending order. We do not require that the $i$'s be numbered consecutively, but when convenient, assume that they have so been renumbered. The slight restriction on the single appearance of each $X_i$ is discussed in section 3.5.

The following proposition is well-known:

(2.1) For any Boolean expression $E$ there is a unique tree $T_E$ (which we shall call the canonical tree of $E$) such that the terminal nodes consist of $X_1, \ldots X_n$, the non-terminal nodes of $+$ and $*$, and in the path from the root to any terminal node the $*$ and $+$ nodes strictly alternate.

Thus, the canonical tree for the expression

(2.1.1) $(X_1 + X_2 * X_3 * \ldots) * (X_5 * X_6 + X_7)$

is

(2.1.2)

(2.2) We will define a table which is a concise representation of the canonical tree.
(2.2.1) **Definition.** For a given Boolean expression $E$ and canonical tree $T_E$, let $R_E$ and $S_E$ be functions defined on every (terminal and non-terminal) node $Y$ of $T_E$ as follows:

i) If $Y$ is the root node, then $R_E(Y) = S_E(Y) = 0$.

ii) If $Y$ is not the rightmost branch of its father, then $S_E(Y) = 0$ or 1 depending on whether the father node is * or +; and $R_E(Y) = i$, where $i$ is the index of the rightmost terminal $X_i$ in the arborescence of the father of $Y$.

iii) If $Y$ is the rightmost branch of its father, then $R_E(Y) = R_E(\text{father}(Y))$, $S_E(Y) = S_E(\text{father}(Y))$.

According to the definitions, each non-terminal node has the same $R_E$ and $S_E$ values as its rightmost son. We save these values only for the terminal nodes of $T_E$; it turns out (as we will show) that we have enough information to recover the tree from the R's and S's.

(2.2.2) **Definition.** Let $P_E = \{S_E, R_E\}$ be a table of $n$ entries, one for each terminal node $X_E$, such that $S_E(i) = S_E(X_i)$, $P_E(i) = P_E(X_i)$. 
Thus, the table corresponding to (2.1.2) is

<table>
<thead>
<tr>
<th>( R_E )</th>
<th>( S_E )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

(2.3) We want to first state some obvious facts about the table \( P_E \), which follow directly from the construction.

(2.3.1) There is at least one terminal node \( X_i \) such that \( i \) does not appear in the \( R \) column, i.e., such that \( X_i \) is not a rightmost son.

(2.3.2) If \( X_j \) is a rightmost node, the operation in the father of \( X_j \) is given by \( S_E(k) \), where \( k \) is the entry immediately preceding \( j \) in the table.

(2.3.3) If \( X_i, X_{i+1}, \ldots X_n, X_{n+1} \) are adjacent brother nodes with \( X_{n+1} \) appearing in the \( R_E \) column, then \( R_E(i) = R_E(i+1) = \ldots = R_E(n) = n+1; \)
\( S_E(i) = \ldots = S_E(n) \); and none of the indices \( i, i+1, \ldots n \) appear in the \( R_E \) column.
(2.4) **Theorem.**

The table $P_E$ uniquely corresponds to the tree $T_E$.

**Proof:** By definitions (2.2.1) and (2.2.2) a tree $T_E$ gives a unique table $P_E$.

We have to show the converse; that is, given a table $P_E$ known to have come from a tree $T_E$ we can reconstruct $T_E$, and that if some other tree $W$ has the same table $P_E$, then $W = T_E$. We do the construction by induction on the number $N$ of entries in the table.

For $N=1$, $T_E$ must be the single node $X_1$. Now assume that we can reconstruct any tree of $n \leq N$ terminal nodes, and let $P_E$ be a table with $N+1$ entries labeled $1, 2, \ldots, N+1$, known to have come from tree $T_E$. Let $k$ be the largest index which does not appear in $R_E$. We must have $R_E(k) = k+1$ (node $X_k$ precedes node $X_{k+1}$ in $T_E$) and also $S(k)$ is the operation of the common father of $X_k$ and $X_{k+1}$; for concreteness assume $S(k) = 0$. Collect all of the immediately preceding entries $i, i+1, \ldots, k$ such that $R_E(i) = k+1, S_E(i) = S_E(k)$; by (2.3.3), these are all of the brothers of the node $X_{k+1}$. Let $P_{E'}$ be the table obtained from $P_E$ by deleting the rows $i, \ldots, k$, and let $T_{E'}$ be the tree obtained from $T_E$ by replacing the subtree $\xymatrix{ X_i & \ast \\ X_k & X_{k+1} \ar[ur] }$ by the node $X_{k+1}$. It is easy to verify that $P_{E'}$ is the table obtained from $T_{E'}$, and by hypothesis only one such table exists. We get $T_E$ uniquely from $T_{E'}$ by replacing node $X_{k+1}$ by the branch $\xymatrix{ X_i & \ast \\ \ldots & X_{k+1} \ar[ur] }$. If $W$ is any other tree with table $P_E$, we can in the same way obtain the smaller $W'$ from $W$, and since by hypothesis $W' = T'$, also $W = T$. Q.E.D.
The table turns out to be both convenient to build and convenient to use for evaluating the expression.

(2.5) **Construction of the table from the expression.**

Let us assume the following grammar (or a suitable variant) for Boolean expressions of the type under consideration:

0. \( Z ::= E \)
1. \( E ::= T \)
2. \( E ::= T + E \)
3. \( T ::= F \)
4. \( T ::= F \times T \)
5. \( F ::= (E) \)
6. \( F ::= X_1 \)

and take for granted some bottom-up left-canonical parsing algorithm (see for example Gries [1] Ch. 5 and 6). Also let \( Z \) denote the distinguished symbol of the grammar. The parsing algorithm itself does not interest us, except in so far as whenever rules 2, 4, or 6 are invoked, the corresponding semantic routine is called. No semantic routine is called for rules 0, 1, 3, and 5.

We make use of a semantic stack \( SS \), which contains pointers to the entries of \( PE \) corresponding to the symbols currently in the syntax stack. The pointer to the top of \( SS \) we denote by \( k \). The following three semantic routines build \( PE \).
(2.5.1) Algorithm B.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Semantic Action</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F::=X&lt;sub&gt;i&lt;/sub&gt;</td>
<td>K:=k+1, i:=SS(k)</td>
<td>Push i onto the stack.</td>
</tr>
<tr>
<td></td>
<td>Set R(i)=S(i)=0</td>
<td>Initialize the table entry for i.</td>
</tr>
</tbody>
</table>
| T:=F*T | S(SS(k-i))=0 | For the F node, set the operation S to *.
| α:    | R(SS(K-1)) ⊕ SS(K) | Set the pointer R to the right brother. |
|       | SS(K-1) + SS(K) | Delete the left brother from the stack. |
| K: = K-1 |                |            |
| E:=T+E | S(SS(K-1)) + 1 | For the T node, set the operation S to +. |
|       | GO TO α         |            |

(2.5.2) Proof that the Algorithm Builds the Table.

We prove the correctness of the construction by induction on the number n of X<sub>i</sub>'s in E (i.e., on the number of terminal nodes in T<sub>E</sub>). For n=1 the only applicable rule is F::=X<sub>i</sub> which yields the desired final result R(1)=S(1)=0. Now assume that for all expressions with not more than n terminals, the algorithm gives the right table. Let E be an expression with n+1 variables, and for simplicity assume no redundant parentheses. Let Z=P<sub>1</sub>=P<sub>2</sub>...P<sub>m</sub>=E be a left canonical parse of E. Somewhere during the parse we must deal with
a construct $X_j \ OP \ X_{j+1}$, where OP is $+$ or $*$; let $i$ be the first such $j$, and for concreteness assume the operation is $*$. Since the parse is assumed to be left canonical, we must have

\[ (2.5.2.1) \quad Z \rightarrow^+ a \ T \ b \rightarrow a \ F^* \ T \ b \rightarrow a \ F^* F \ b \rightarrow a \ F^* X_{i+1} \ b \rightarrow a \ X_i^* X_{i+1} \ b \rightarrow E \]

where $a$ and $b$ are some strings and $b$ contains only terminals. Now consider an expression $E'$ which we get from $E$ by replacing $X_i^* X_{i+1}$ by $X_{i+1}$. The canonical parse for $E'$ must be

\[ (2.5.2.2) \quad Z \rightarrow^+ a \ T \ b \rightarrow a \ F^* b \rightarrow a \ X_{i+1} \ b \rightarrow E' \]

where $a$ and $b$ are the same strings as before. $E'$ contains only $n$ terminals, and by hypothesis the algorithm builds the correct table for $E'$. How does the table for $E$ differ from the table for $E'$? The parsing algorithm goes through exactly the same steps for $E'$ and $E$ except that at a certain point we have for $E'$ the productions

\[ a \ T \ b \rightarrow a \ F^* b \rightarrow a \ X_{i+1} \ b \]

while for $E$ we have at the same point

\[ a \ T \ b \rightarrow a \ F^* T \ b \rightarrow a \ F^* F \ b \rightarrow a \ F^* X_{i+1} \ b \rightarrow a \ X_i^* X_{i+1} \ b \]

We analyze the step by step effect on the semantic stack $S_S$ to show that the only difference in the results is that for $E$ we build the entry $i$ and build it properly. Let $<y>$ denote the prior contents of $S_S$; $<y>$ does not contain $i$. Remember that the parse is bottom-up, so that we execute the semantics in reverse order from the productions.
\[
\begin{array}{lll}
\text{E} & \text{Rule} & \text{Action} \\
F ::= X_i & R(i) = S(i) = 0 & i, <y> \\
F ::= X_{i+1} & R(i+1) = S(i+1) = 0 & i+1, i, <y> \\
T ::= F & \text{none} & i+1, i, <y> \\
F ::= F \ast T & S(i) = 0, R(i) = i+1 & i+1, <y> \\
\end{array}
\]

\[
\begin{array}{lll}
\text{E}' & \text{Rule} & \text{Action} \\
F ::= X_{i+1} & R(i+1) = S(i+1) = 0 & i+1, <y> \\
T ::= F & \text{none} & i+1, <y> \\
\end{array}
\]

Since the semantic stack and the values of \(R(i+1), S(i+1)\) are not changed by the introduction of node \(i\), and since \(i\) can never reappear in the semantic stack and hence the \(i^{th}\) entry can never change, we conclude that the table built for E will be the same as that for E', except for node \(i\) which has the value \(R(i)=i+1, S(i)=0\). Let \(T_E\) be the tree for E'. The tree \(T_E\) for E can be obtained from \(T_E\) by either replacing the terminal node \(X_{i+1}\) by \(X_i \ast X_{i+1}\) (if the parent of \(X_{i+1}\) in \(T_E\) is \(+\)) or by inserting \(X_i\) just to the left of \(X_{i+1}\) (if the parent in \(T_E\) is \(*\)).

It is trivial to verify that the table constructed is indeed the table for \(T_E\).
(2.6) Evaluating the table in sequence.

We can find the value of a Boolean expression by a traversal algorithm on its canonical tree. That is, we define the value of any node recursively as follows.

(2.6.1) Evaluating the tree \( T \):

1) The value of a terminal node \( X_i \) is obtained by "look-up".

2) The value of a "+" node is 1 if any of its sons has the value 1, and 0 otherwise.

3) The value of a "*" node is 0 if any of its sons has the value 0, and 1 otherwise.

It follows that the value of the expression is the same as the value of the root node of the tree.

(2.6.2) Evaluating the table \( P \).

Based on the above, we can define a traversal algorithm convenient for evaluating the table. We start evaluating the sons of a node; the moment the result matches the operation, we associate the result with the rightmost son and hence with the node itself. If the result doesn't match the operation, we continue until the rightmost son has been evaluated; this result is then the value of the father node.
(2.6.3) **Algorithm Ev** (sequential evaluation).

1) Set $K=1$.

2) "Lookup" $X_K$, set $V=$result.

3) If $K=N$, we are finished (the result is $V$); else compare the result to $S(K)$. If they agree, go to step 4, else go to step 5.

4) Set $K=R(K)$, repeat step 3.

5) Set $K=K+1$, repeat step 2.

The proof of the correctness of the algorithm follows from the definitions of $R(i)$, $S(i)$ and (2.6.1) and is skipped here.

The algorithm Ev (similar to the one given in Gries [1] Section 14.6.2) does not need any "memory"; by the virtue of being at a node, we know that the node needs evaluation. Thus, the table $P_E$ has sufficient structure to allow efficient evaluation of the Boolean expression.
3. **EVALUATING THE TABLE OUT OF SEQUENCE.**

We have come to the heart of the matter. We are really interested in knowing when the expression has been completely evaluated, given a random sequence of "look-ups". That is, we would like to modify algorithm Ev so that in steps 1 and 5 we can pick the most convenient node, not necessarily the next one in sequence. In Section 4 we will also be concerned with the problem of partial re-initialization.

To do the evaluation efficiently, we must keep track of intermediate results associated with each node of the canonical tree; the table PE does not contain sufficient structure to allow us to do this. Of course, we can (by 2.4) always reconstruct the entire tree from the table; to do so each time, however, is awfully wasteful, and we might as well originally build a tree to represent the expression and to keep track of the dynamic information.

(3.1) For each node we want to remember whether the node has been evaluated and (if so) what its value \( V \) was. A terminal node is evaluated by "look-up"; all other nodes are evaluated by evaluating as few sons as are required to determine the value. To help us do this, we associate with each non-terminal node a count \( C > 1 \) of all sons of the node; for terminal nodes, \( C = 1 \). For all nodes, we use a cell \( V \) to contain its current value (\(-1 = \"unevaluated\")). \( R \) will now denote the pointer to the father node, (0 for the root node) and \( S \) will denote the operation in the node itself; for terminal nodes \( S \) is \(-1\).
(3.2) **Definition.** Let \( P'_E \) be a table with a row entry for each node of the canonical tree. Each row \( i \) has four entries \( R(i), S(i), C(i), \) and \( V(i) \), where \( R, S, C, \) and \( V \) are as described in 3.1. \( P'_E \) is simply a particular representation of the tree \( T_E \).

(3.3) **Building the table \( P'_E \).** The table \( P'_E \) can be constructed directly from \( E \) in a manner similar to \( P_E \). We adopt the notation of 2.5.

**Algorithm B' (constructing \( P'_E \)).** We make use of a procedure `ALLOCATE (j)` which allocates a table entry with index \( j \), and initializes the values of \( C \) to 1, \( S \) and \( V \) to -1, and \( R \) to 0.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Semantic Action</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F := X_i )</td>
<td><code>ALLOCATE(j)</code></td>
<td>Allocate and initialize new entry, ( K + k+1, SS(K) + j ) put index onto semantic stack.</td>
</tr>
<tr>
<td>( T := F \ast T )</td>
<td>Set OP to 0</td>
<td>OP is the only difference between ( \ast ) and ( + ).</td>
</tr>
<tr>
<td>( E := T + E )</td>
<td>Set OP to 1</td>
<td></td>
</tr>
<tr>
<td>Common:</td>
<td>( S(SS(K)) = OP? )</td>
<td>Do we already have brothers of this type?</td>
</tr>
<tr>
<td>YES:</td>
<td>( R(SS(K-1)) \leftarrow SS(K) )</td>
<td>Set pointer in new son to father</td>
</tr>
<tr>
<td></td>
<td>( C(SS(K)) + 1 + C(SS(K)) )</td>
<td>Step count of sons.</td>
</tr>
<tr>
<td></td>
<td>( SS(K-1) + SS(K), K+K-1 )</td>
<td>Delete son from stack.</td>
</tr>
</tbody>
</table>
- 17 -

NO: ALLOCATE(j)  Allocate new non-terminal node.
      R(SS(K-1))+R(SS(K)+j Set pointers in each son to father.
      C(j)+2, S(j)+OP   Set count of sons, operation, in father.
      K+K-1, SS(K)+j   Delete sons from stack, enter index
                        of new node.

The correctness of this construction can be easily proven by the methods of
2.5. Thus, for (2.2.1) the table $P'_E$ is:

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>C</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

This is nothing but an enumeration of the nodes of the canonical tree in
endorder (see Knuth [5], p. 316).
(3.4) **Evaluating the table** $P_E'$ **randomly.** We are given a selection function $f$ which for each $m$ in sequence $(1 \leq m \leq N)$ produces an $i = f(m)$ which is the index in $P_E'$ of the next condition $X$ (terminal node $i$) to evaluate. The algorithm must decide first whether the value of $X_i$ can affect the result; and (if so) evaluate $X_i$, propagate its value as high as possible, marking the appropriate $V$'s, and decide whether the entire expression has already been evaluated.

If the value of a node (terminal or otherwise) can affect the result, we will call the node **significant**. The root node is always significant. The following proposition is self-evident.

(3.4.1) At any point during the evaluation, for any node $j$:

If $V(R(j)) \neq -1$, $j$ cannot affect the result; else, $j$ is significant if and only if $R(j)$ is significant.

(3.4.2) **Algorithm Rev** (random evaluation).

1) Set $k=1$.

2) Let $m = f(k)$ (m is the pointer to the node for the selected terminal condition).

3) Is $m$ significant?

3.1 $j \leftarrow m$;

3.2 If $V(R(j)) \neq -1$ then (no) goto 5 else if $R(j) = 0$ then (yes) goto 4 else $j \leftarrow R(j)$, repeat 3.2.
4) **Significant node:**

4.1 "Lookup" \( X_m \), set \( V(m) \) to value of the lookup, and \( j \rightarrow m \).

4.2 If \( R(j) = 0 \), we are done with the evaluation (the result is \( V(m) \)); else set \( j \rightarrow R(j) \), and \( C(j) \rightarrow C(j)-1 \) (decrement count of sons left to evaluate);

4.3 If \( C(j) = 0 \) or \( V(m) = S(j) \) then (node \( j \) is evaluated)

Set \( V(j) \rightarrow V(m) \), repeat 4.2; else goto step 5.

5) **Next node:** \( k \rightarrow k+1 \), go to step 2.

The correctness of the algorithm can be demonstrated by induction on the number \( N \) of the terminal nodes, using the above methods, and is left to the reader as an exercise. (For the induction step, consider the first occurrence in the canonical parse of \( X_i \) OP \( X_{i+1} \)).

(3.5) With this technique, we can now handle efficiently multiple occurrences of the same condition. We still identify each occurrence by a different index, but "remember" that \( X_{i_1} \equiv X_{i_2} \ldots = X_{i_N} \). The first time we evaluate one of them, we also enter the result for all of the others which are significant.

(The reason for entering only the significant values will become clear in the next section, when we discuss partial re-initialization.)
4. PARTIAL REINITIALIZATION

In queries over data bases, one often wishes to change the values of some (but not all) of the variables and to re-evaluate the expression. For example, some of the conditions may depend on elements in a repeating group, and we may want to look at another instance of the occurrence of the group. We want to determine automatically whether and how the change affects the value of the expression, without necessarily re-evaluating the entire expression from the beginning.

The evaluation algorithm Rev of 3.4.2 has the nice property that not only is the minimum amount of work done, but also that the process can be reversed when required.

(4.1) Let \( g(k) \) (\( k=1, \ldots, L \)) be a random re-initialization function; i.e., for each \( k \) it produces an \( m = g(k) \) which is the index of a terminal node whose value is no longer defined. For each \( k=1, \ldots, L \) we want to re-initialize the table, so that we can subsequently determine (via algorithm Rev) whether and how the new values affect the value of the expression.

(4.2) Algorithm Rein (Partial re-initialization of the table).

1. Set \( k=1 \)
2. Let \( m = g(k) \)
3. **Was** \( m \) **evaluated?** If \( V(m) = -1 \), (no) nothing to do: go to 6. else (yes) set Prior \( \rightarrow V(m) \), and \( j \rightarrow m \).
4. Reset $V(j)$. $V(j) \leftarrow -1$. If $R(j) = 0$ then (we are at the root node) go to 6; else $C(R(j)) + C(R(j)) + 1$ (increment count of defined sons of father).

5. Is father affected? If $S(R(j)) = \text{Prior}$, then (yes) $j \leftarrow R(j)$, repeat 4; else go to 6.

6. Reinitialize next node: $k = k+1$; if $k \leq L$, go to 2.

(4.3) Proof of the correctness of Rein.
Since the algorithms Rev and Rein do not depend on the order in which terminal nodes $X_i$ are presented, it will suffice to show that for any $X_k$, execution of steps 2-4 of Rev followed by the execution of steps 2-5 of Rein leaves the table $P'E$ unchanged. If $k$ was not significant, neither Rev nor Rein change the table. If $k$ was significant, all of its ancestors had before the evaluation $V = -1$, and after the evaluation $p$ of the ancestors received the value $V(k)$; the fathers of each such node had their counts decremented by 1. Rein can be seen to increment their counts by 1 and reset their $V$'s to -1.

(4.4) If the re-initialization always involves the most-recently-evaluated collection of nodes, the algorithm Rev can restart from the changed nodes. In particular, if $g(k) = f(n+k)$ for all $k$ for which $g$ is defined, there is no need to redo Rev for $f(1), \ldots f(n)$, since we either evaluated those (and have not changed them) or they weren't (and still aren't) significant. Thus, in the important case where both $f$ and $g$ follow the organization of the data base, we can see effects of the local changes without having to reevaluate the entire expression.
5. **SUMMARY.**

We have described a technique for automatically dealing with Boolean expressions of arbitrary complexity, when the verification of the conditions involves accessing a database in peripheral storage. In retrospect, the results are not particularly deep or surprising. Nevertheless, having spent many hours in grappling with a solution in the context of attempting to implement a working system (see Gudes [7] and Borkowski [8]) the authors are presenting it here in the hope of saving others similar travail.

We are grateful to Shmuel Winograd for suggesting the basic idea in the proof of theorem 2.4.
REFERENCES


