EFFICIENT EVALUATION OF BOOLEAN EXPRESSIONS

by

Bryan S. Todd

Oxford University Computing Laboratory
Programming Research Group

EFFICIENT EVALUATION OF BOOLEAN EXPRESSIONS

by

Bryan S. Todd

Technical Report PRG-TR-3-97 Feb 1997

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building
Parks Road
Oxford OX1 3QD
England

Copyright © 1997 Bryan S. Todd

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building
Parks Road
Oxford OX1 3QD
England

Electronic mail: todd@comlab.ox.ac.uk (JANET)

Efficient Evaluation of Boolean Expressions

Bryan S. Todd

Summary

In expert systems research, and more widely in computer science, a recurring and important problem is that of determining whether a boolean expression has a satisfying truth assignment. The general case is well known to be NP-complete, although efficient algorithms exist for many restricted classes. Unfortunately, the known feasible classes tend to be insufficiently expressive for knowledge representation. In this paper we transfer ideas from bayesian network theory to present a general-purpose algorithm for evaluating boolean expressions. The algorithm is a special case of that due to Lauritzen and Spiegelhalter for propagating evidence through bayesian networks. The algorithm is efficient for a large class of boolean expressions of practical relevance and may possibly offer a useful alternative to other general-purpose tautology checkers.

Contents

1 Introduction

A commonly occurring problem in computer science is the need to determine quickly whether or not a boolean expression has a satisfying truth assignment. Unfortunately, the general problem is NP-complete, although efficient algorithms exist for some special cases. These include

1. expressions in which no variable appears more than once ('μ-expressions'),
2. monotone expressions,
3. CNF expressions in which no clause contains more than two variables ('2-CNF expressions'), and
4. CNF expressions in which no clause contains more than one negated variable (or alternatively, no clause contains more than one positive variable).

In the rest of this paper we develop a general-purpose algorithm for evaluating boolean expressions. The algorithm is shown to be efficient both for μ-expressions and monotone expressions, but it is also applicable to a larger class of expressions that are of practical relevance.

2 Expressions and Satisfiability

We start by introducing the notion of concealment of variables in boolean expressions. We then present a lemma that shows satisfiability testing reduces to the task of hiding all the variables in an expression.

A boolean expression $\mathcal{E}$ imposes logical constraints over a number of variables. A variable α can be *concealed* in $\mathcal{E}$ by relaxing all logical constraints over the truth value of α. Let $\mathcal{E}\backslash\alpha$ be the expression obtained by hiding α in this way. Thus for any such expression $\mathcal{E}$ and variable α,

$$\mathcal{E}\backslash\alpha \quad \widehat{=} \quad \mathcal{E}[\alpha\backslash\mathsf{F}\,] \;\vee\; \mathcal{E}[\alpha\backslash\mathsf{T}\,] \tag{1}$$

where 'F' and 'T' are the logical constants denoting the truth values 'false' and 'true', respectively.

For example,

$$\begin{aligned}((\alpha\wedge\beta)\vee\gamma)\backslash\alpha \;&\equiv\; ((\alpha\wedge\beta)\vee\gamma)[\alpha\backslash\mathsf{F}\,] \;\vee\; ((\alpha\wedge\beta)\vee\gamma)[\alpha\backslash\mathsf{T}\,]\\ &\equiv\; ((\mathsf{F}\wedge\beta)\vee\gamma) \;\vee\; ((\mathsf{T}\wedge\beta)\vee\gamma)\\ &\equiv\; (\mathsf{F}\vee\gamma) \;\vee\; (\beta\vee\gamma)\\ &\equiv\; \beta\vee\gamma\end{aligned}$$

where '$\equiv$' denotes logical equivalence of two expressions.

An expression $\mathcal{E}$ *constrains* a variable α precisely when $\mathcal{E}$ is altered logically by concealment of α. For any expression $\mathcal{E}$, let $\mathsf{vars}(\mathcal{E})$ be the set of variables constrained by $\mathcal{E}$. Thus,

$$\mathsf{vars}(\mathcal{E}) \quad \widehat{=} \quad \{\alpha : \mathrm{Var} \mid \mathcal{E}\backslash\alpha \not\equiv \mathcal{E}\} \tag{2}$$

where 'Var' is the set of all boolean variables.

For example,

$$\begin{array}{lcl}
\mathsf{vars}((\alpha \wedge \beta) \vee \gamma) & = & \{\alpha, \beta, \gamma\} \\
\mathsf{vars}((\alpha \wedge \beta) \vee \alpha) & = & \{\alpha\} \\
\mathsf{vars}(\mathsf{F}) & = & \{\} \\
\mathsf{vars}(\mathsf{T}) & = & \{\}
\end{array}$$

Notice that hiding is commutative. (The proofs of this and all subsequent lemmas are trivial exercises in propositional logic, and are therefore not included.)

Lemma 1

$$\vdash \quad (\mathcal{E}\backslash\alpha)\backslash\beta \quad \equiv \quad (\mathcal{E}\backslash\beta)\backslash\alpha$$

Therefore hiding can be promoted to that of sets of variables (for example, $\mathcal{E}\backslash\{\alpha, \beta, \gamma\}$), hidden in an arbitrary order.

Notice also that hiding distributes through disjunction.

Lemma 2

$$\vdash \quad (\mathcal{E} \vee \mathcal{F})\backslash\alpha \quad \equiv \quad (\mathcal{E}\backslash\alpha) \vee (\mathcal{F}\backslash\alpha)$$

In the general case hiding does not distribute through conjunction, but it does do so in the case that one or other of the two operands contains none of the variables that are being hidden. (This result will be useful later.)

Lemma 3

$$\alpha \notin \mathsf{vars}(\mathcal{E}) \quad \vdash \quad (\mathcal{E} \wedge \mathcal{F})\backslash\alpha \quad \equiv \quad \mathcal{E} \wedge (\mathcal{F}\backslash\alpha)$$

It follows that an expression $\mathcal{E}$ has a satisfying truth assignment precisely when concealment of all the variables in $\mathcal{E}$ reduces the expression to the logical constant 'true'.

Lemma 4

$$\vdash \quad \mathcal{E} \not\equiv \mathsf{F} \quad \Leftrightarrow \quad \mathcal{E}\backslash\mathsf{vars}(\mathcal{E}) \equiv \mathsf{T}$$

In the next section we use this lemma to obtain an algorithm for satisfiability testing.

3 Decomposition

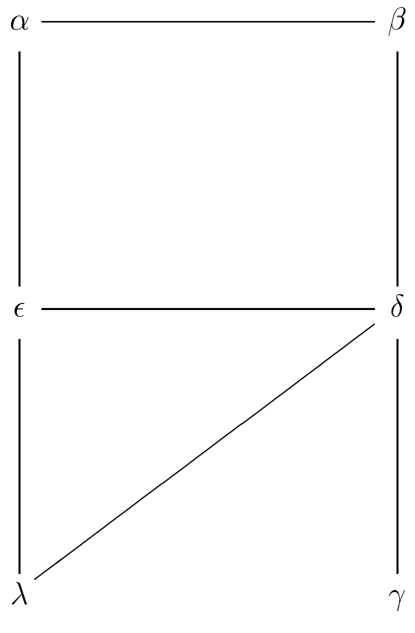
In most practical applications, a logical theory $\mathcal{E}$ tends to be expressed as a conjunction of smaller expressions $\mathcal{E} \equiv \mathcal{E}_1 \wedge \mathcal{E}_2 \wedge \ldots$, each containing relatively few variables. This suggests that for satisfiability testing we adapt the algorithm due to Lauritzen and Spiegelhalter for propagating evidence through bayesian networks [Lau88]. This is best illustrated by an example.

Suppose that

$$\mathcal{E} \equiv (\alpha \Leftrightarrow \beta) \wedge (\gamma \vee \delta) \wedge ((\delta \vee \lambda) \Rightarrow \epsilon) \wedge (\beta \vee \delta) \wedge (\alpha \Rightarrow \epsilon)$$

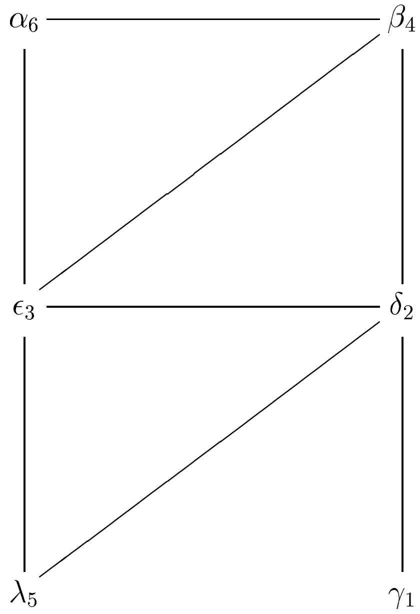
This theory has five component expressions. We represent graphically the logical dependencies imposed between the variables: an arc between two variables denotes a direct logical dependency arising from one or more of the component expressions. Each component expression thus induces a complete subgraph over the variables that it constrains. The entire dependency graph is formed by the superposition of all five complete subgraphs (Figure 1).

Figure 1: Dependency graph.



Next, the graph is triangulated by the inclusion of an arc between β and ϵ to bridge the cycle $<\alpha, \beta, \delta, \epsilon, \alpha>$. (Efficient algorithms exist for checking triangulation of undirected graphs and for computing a minimal fill-in.) Then the nodes are labelled by maximum cardinality search (Figure 2).

Figure 2: Triangulated graph indexed by maximum cardinality search.



Finally, the cliques of the triangulated graph are enumerated in an order that reflects the largest index in each clique. (An efficient algorithm exists for enumerating the cliques of a triangulated graph.) There are four cliques in our example, $C_1 \ldots C_4$.

$C_1 = \{\gamma, \delta\}$	Variable with highest index: δ	(index = 2)
$C_2 = \{\beta, \delta, \epsilon\}$	Variable with highest index: β	(index = 4)
$C_3 = \{\delta, \epsilon, \lambda\}$	Variable with highest index: λ	(index = 5)
$C_4 = \{\alpha, \beta, \epsilon\}$	Variable with highest index: α	(index = 6)

Processing the graph in this way guarantees that the four cliques have the 'running intersection property'. Each clique is partitioned into 'separator' variables and 'residual' variables. For each clique C_i, the *separator* variables S_i are those that appear in any clique of lower index.

$$S_i \quad \widehat{=} \quad (C_1 \cup C_2 \cup \ldots \cup C_{i-1}) \;\cap\; C_i \tag{3}$$

Notice that $S_1 = \{\}$.

The *residual* variables R_i are the remainder.

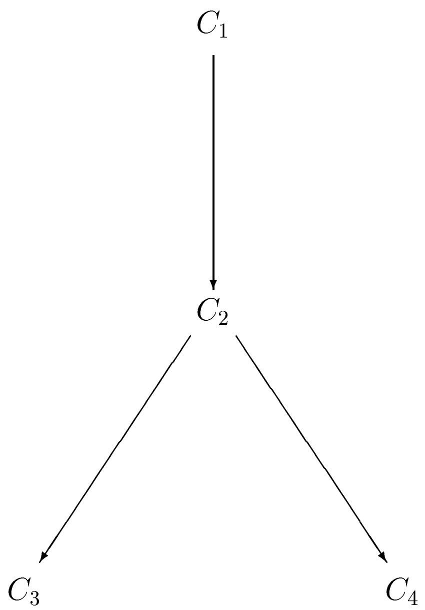
$$R_i \quad \widehat{=} \quad C_i - S_i \tag{4}$$

Thus, in our example,

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl} C_1 &=& \{\gamma,\delta\} & S_1 &=& \{\} & R_1 &=& \{\gamma,\delta\} \\ C_2 &=& \{\beta,\delta,\epsilon\} & S_2 &=& \{\delta\} & R_2 &=& \{\beta,\epsilon\} \\ C_3 &=& \{\delta,\epsilon,\lambda\} & S_3 &=& \{\delta,\epsilon\} & R_3 &=& \{\lambda\} \\ C_4 &=& \{\alpha,\beta,\epsilon\} & S_4 &=& \{\beta,\epsilon\} & R_4 &=& \{\alpha\} \end{array}$$

A sequence of sets of variables is said to have the *running intersection property* precisely when for each $i > 1$ there exists a $j < i$ such that $S_i \subseteq C_j$. If so, for each i we choose any such j and define C_j to be the *parent* of C_i. This generates a tree of dependency between the sets. The tree generated for our example is shown in Figure 3.

Figure 3: Clique tree.



Furthermore, since triangulation has not removed any arcs from the original graph, the set of variables in each of the five component expressions is a subset of at least one of these cliques. We can therefore rearrange $\mathcal{E}$ so that it is a conjunction of four expressions ($\mathcal{E} \equiv \mathcal{E}_1 \wedge \mathcal{E}_2 \wedge \mathcal{E}_3 \wedge \mathcal{E}_4$) such that $\mathsf{vars}(\mathcal{E}_i) \subseteq C_i$ for $1 \leq i \leq 4$.

$$\begin{array}{rcl} \mathcal{E}_1 &\equiv& \gamma \vee \delta \\ \mathcal{E}_2 &\equiv& \beta \vee \delta \\ \mathcal{E}_3 &\equiv& (\delta \wedge \lambda) \Rightarrow \epsilon \\ \mathcal{E}_4 &\equiv& (\alpha \Leftrightarrow \beta) \wedge (\alpha \Rightarrow \epsilon) \end{array}$$

Now, since

$$\mathsf{vars}(\mathcal{E}) = R_1 \cup R_2 \cup R_3 \cup R_4$$

the task of computing $\mathcal{E}\backslash\mathsf{vars}(\mathcal{E})$ reduces to one of successively hiding each of the R_i. This is easily accomplished by the following algorithm.

In general let n be the number of cliques, and let π map each clique to its parent by index: thus, for $i > 1$, $C_{\pi(i)}$ is the parent of C_i.

<u>Algorithm</u>

$$\begin{array}{l}
\mathsf{for}\ \ i := n\ \ \mathsf{downto}\ \ 2\ \ \mathsf{do} \\
\qquad \mathcal{E}_i := \mathcal{E}_i\backslash R_i \\
\qquad \mathcal{E}_{\pi(i)} := \mathcal{E}_{\pi(i)} \wedge \mathcal{E}_i \\
\mathsf{if}\ \ \mathcal{E}_1\backslash R_1 \equiv \mathsf{T}\ \ \mathsf{then} \\
\qquad \mathcal{E} \text{ is satisfiable} \\
\mathsf{else} \\
\qquad \mathcal{E} \text{ is unsatisfiable}
\end{array}$$

<u>Loop Invariant</u>

$$\bigwedge_{j \leq i} \mathcal{E}_j \quad \equiv \quad \mathcal{E}\backslash \bigcup_{i<j} R_j$$

Let us now see how this algorithm works on our example. We start by choosing the clique with highest index, C_4. Since by definition,

$$R_4 \cap (C_1 \cup C_2 \cup C_3) = \{\}$$

we have that

$$R_4 \cap \mathsf{vars}(\mathcal{E}_1 \wedge \mathcal{E}_2 \wedge E_3) = \{\}$$

Therefore the concealment of R_4 distributes through conjunction (Lemma 3), and we have that

$$\mathcal{E}\backslash R_4 \quad \equiv \quad \mathcal{E}_1 \wedge \mathcal{E}_2 \wedge \mathcal{E}_3 \wedge (\mathcal{E}_4\backslash R_4)$$

We now conjoin the expression $\mathcal{E}_4\backslash R_4$ with that of C_4's parent, $\mathcal{E}_2$. Thus,

$$\mathcal{E}\backslash R_4 \quad \equiv \quad \mathcal{E}_1 \wedge \mathcal{E}_2' \wedge \mathcal{E}_3$$

where

$$\begin{aligned}
\mathcal{E}_2' &\equiv \mathcal{E}_2 \wedge (\mathcal{E}_4\backslash R_4) \\
&\equiv (\beta \vee \delta) \wedge (((\alpha \Leftrightarrow \beta) \wedge (\alpha \Rightarrow \epsilon))\backslash\{\alpha\}) \\
&\equiv (\beta \vee \delta) \wedge (\beta \Rightarrow \epsilon)
\end{aligned}$$

We now repeat the process for the preceding clique, C_3. Its parent is also C_2, so

$$\mathcal{E}\backslash R_4\backslash R_3 \quad \equiv \quad \mathcal{E}_1 \wedge \mathcal{E}_2''$$

where

$$\begin{aligned}\mathcal{E}_2'' &\equiv \mathcal{E}_2' \wedge (\mathcal{E}_3 \backslash R_3)\\ &\equiv (\beta \vee \delta) \wedge (\beta \Rightarrow \epsilon) \wedge (((\delta \vee \lambda) \Rightarrow \epsilon) \backslash \{\lambda\})\\ &\equiv (\beta \vee \delta) \wedge (\beta \Rightarrow \epsilon) \wedge (\delta \Rightarrow \epsilon)\\ &\equiv (\beta \vee \delta) \wedge \epsilon\end{aligned}$$

We now repeat the process for the preceding clique, C_2. Its parent is C_1, so

$$\mathcal{E} \backslash R_4 \backslash R_3 \backslash R_2 \equiv \mathcal{E}_1'$$

where

$$\begin{aligned}\mathcal{E}_1' &\equiv \mathcal{E}_1 \wedge (\mathcal{E}_2'' \backslash R_2)\\ &\equiv (\gamma \vee \delta) \wedge (((\beta \vee \delta) \wedge \epsilon) \backslash \{\beta, \epsilon\})\\ &\equiv \gamma \vee \delta\end{aligned}$$

Finally, therefore,

$$\begin{aligned}\mathcal{E} \backslash \mathsf{vars}(\mathcal{E}) &\equiv \mathcal{E} \backslash R_4 \backslash R_3 \backslash R_2 \backslash R_1\\ &\equiv \mathcal{E}_1' \backslash R_1\\ &\equiv (\gamma \vee \delta) \backslash \{\gamma, \delta\}\\ &\equiv \mathsf{T}\end{aligned}$$

Therefore $\mathcal{E}$ has a satisfying truth assignment (Lemma 4).

Once the clique tree is constructed (and efficient algorithms are available for that purpose) the algorithm is linear in the number of cliques (never more than the number of variables constrained by the original expression). The time limiting step is therefore that of computing the result of hiding variables from the expression associated with a clique. In the worst case this is exponential with respect to the size of the clique: the expression can always be represented as an explicit truth table, and hiding operations can be performed in linear time on truth tables.

4 Special Classes

Restricted classes of expression are amenable to efficient satisfiability testing by special purpose algorithms. It is worth considering how our algorithm applies to these classes of expression because it helps us see in a common framework why satisfiability testing is feasible for those classes.

4.1 μ-Expressions

Every μ-expression can be converted in linear time to a 3-CNF expression by introducing new variables for every subexpression. The 3-CNF expression induces a dependency graph that is a tree of triangles. The dependency graph is therefore already triangulated, and the largest clique contains no more than three variables.

4.2 Monotone Expressions

A variable α is said to be *positive* in an expression $\mathcal{E}$ precisely when $\mathcal{E}[\alpha\backslash\mathsf{F}] \Rightarrow \mathcal{E}[\alpha\backslash\mathsf{T}]$. Correspondingly, α is said to be *negative* in $\mathcal{E}$ precisely when $\mathcal{E}[\alpha\backslash\mathsf{T}] \Rightarrow \mathcal{E}[\alpha\backslash\mathsf{F}]$. Expression $\mathcal{E}$ is *monotone* precisely when every variable is either positive or negative in $\mathcal{E}$.

If every variable α is positive in the expression $\mathcal{E}_i$ associated with each clique then application of the algorithm is straightforward since

$$\begin{aligned} \mathcal{E}_i\backslash\alpha &\equiv \mathcal{E}[\alpha\backslash\mathsf{F}] \vee \mathcal{E}[\alpha\backslash\mathsf{T}] \\ &\equiv \mathcal{E}[\alpha\backslash\mathsf{T}] \end{aligned}$$

Similarly, if α is negative in $\mathcal{E}_i$,

$$\mathcal{E}_i\backslash\alpha \equiv \mathcal{E}[\alpha\backslash\mathsf{F}]$$

Therefore no growth in size of the expression occurs when variables are hidden: although conjunction of component expressions takes place at each iteration of the algorithm, no component expression exceeds the size of the original expression $\mathcal{E}$.

5 Comment

The dependency graphs induced in practical applications are likely to be sparse and should therefore produce small cliques. This has been the experience with bayesian networks for modelling complex probabilistic systems: since logical models are a special case of probabilistic ones it seems reasonable to suppose that the same holds. The algorithm should therefore have wide application and may possibly offer a useful alternative to other general-purpose tautology checkers (e.g. that due to Stålmarck [Sta89]). This would perhaps be best determined by some realistic comparative case studies.

Acknowledgement

I am grateful to Tony Hoare for his helpful comments on an earlier draft of this paper.

References

[Lau88] **Lauritzen SL, Spiegelhalter DJ.** Local computation with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society, Series B* (1988) **50** 157–224.

[Sta89] **Stålmarck G.** A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. *U.S. patent 5 276 897* (1989, approved 1994).