A Satisfiability Tester for Non-Clausal Propositional Calculus

Allen Van Gelder
Computer Science Dept.
University of California
Santa Cruz, CA 95064, USA

Abstract

An algorithm for satisfiability testing in the propositional calculus with a worst case running time that grows at a rate less than $2^{(.25+\varepsilon)L}$ is described, where L can be either the length of the input expression or the number of occurrences of literals (i.e., leaves) in it. This represents a new upper bound on the complexity of non-clausal satisfiability testing. The performance is achieved by using lemmas concerning assignments and pruning that preserve satisfiability, together with choosing a "good" variable upon which to recur. For expressions in conjunctive normal form, it is shown that an upper bound is $2^{.128L}$.

This paper appears in *Information and Computation*, **79**, 1, October, 1988, pages 1–21.

Errata have not been corrected in the text. Figures are absent.

Known errata (we thank Yumi K. Tsuji for pointing out the first two):

The fourth line of Eq. 6.3 on p. 5 is incorrect, but it is never used.

Lemma 6.4 on p. 7 is incorrect as stated, but the analysis of Section 7 does not rely on Lemma 6.4. To correct the lemma, strengthen the hypothesis by adding that each operator node with y as a child also has x as a child and each operator node with $\tilde{y}$ as a child also has $\tilde{x}$ as a child.

References to Davis-Putnam [DaP60] should be to Davis, Logemann, and Loveland, *CACM* **5**, pp. 394–397, 1962.

1. Introduction

An algorithm for satisfiability testing in the propositional calculus with a worst case running time that grows at a rate less than $2^{(.25+\varepsilon)L}$ for any positive ε is described, where L can be either the length of the input expression or the number of occurrences of literals (i.e., leaves) in it. This represents a new upper bound on the complexity of non-clausal satisfiability testing. We adopt the expression length as the size measure, as usual in complexity theory [Coo71]; only counting variables is not a satisfactory measure because the time required for elementary operations on expressions is not bounded by any function of the variable count in standard models of computation, such as the Turing Machine and Random-Access Machine.

The input to the algorithm is a Boolean expression with connectives **and**, **or**, and **not**. The connectives **iff** and **xor** are not used. The expression need not be in conjunctive normal form or any normal form. The output is either "satisfiable" followed by a satisfying assignment, or "unsatisfiable". In the latter case, an enumeration proof is available, but it is very long and not very illuminating.

The algorithm works by a variation of Quine's method [Qui50] but for inputs in conjunctive normal form (CNF), can be viewed as an extended form of the Davis-Putnam method [DaP60]. The performance is achieved by using lemmas in Section 6 concerning assignments and pruning that preserve satisfiability, together with choosing a "good" variable upon which to recur.

To determine satisfiability for a given Boolean expression, the algorithm first makes simplifications and assignments that shorten the expression while preserving its satisfiability, then chooses a variable and recursively tests satisfiability after each assignment (true, false) to that variable throughout the expression.

A program implementing this algorithm was written in Franz Lisp. Non-clausal expressions ranging in size from 48 to 106 literals and 12 to 32 variables were solved. (Transforming to CNF would have yielded substantially larger sizes.) Test results are summarized in Section 8. The larger tests suggest a growth rate of $2^{\frac{L}{12}}$, but the data is no where near sufficient to support this as a firm conclusion.

Previous work on efficient satisfiability testing (see [Gol79] for survey) has been concerned primarily with expressions in clause form. Transformation of an arbitrary expression into CNF can be done in polynomial time only by introducing new variables [Tse68]. This can almost double the number of variables. In addition, the method given there (not necessarily optimal) may increase the length of the expression by a factor of seven. Without introduction of new variables, an exponential blowup is possible. A linear expansion of problem size, while not critical in complexity theory [Coo71] may have a substantial impact in practice. In fact, no non-trivial upper bounds have been shown for non-clausal expressions. Exponential lower bounds for several proof procedures have been shown [Tse68, Gal75, Gal77] but the bound is so low (below $2^{.001L}$) as to be of theoretical significance only. In addition, no non-trivial lower bound is known for *extended* resolution [Tse68, Gal77].

2. Succinct Representation of Boolean Expressions

Boolean expressions are modeled as n-ary trees with two types of nodes, *operation* and *leaf*. Each operation node is either an **and** or an **or**, and may have an any number of children. Each leaf is an occurrence of a literal. A literal is a "polarized" variable, i.e., either a variable (positive polarity) or a complemented variable (negative polarity). Literals can also be polarized. In addition, in interim expressions, an operation node may hold a truth value by containing **and** (for **true**) or **or** (for **false**), and no children.

The input expression may also contain **not** at interior nodes, but the program merely pushes these **not**s down to the leaves, using DeMorgan's rules. This one-time transformation affects neither the number of variables nor the number of leaves, and hence has a negligible effect upon running time. After the **not**s have been pushed down to the leaves, we call the resulting tree an AND-OR tree.

To analyze running time it is convenient to have a uniform and compact representation of Boolean expression trees, which motivates the following definitions.

Definition Let two children of a node both be leaves. If they contain a literal and the negation of that literal, they are *inconsistent*. If they both represent the same literal, one is *redundant*. These definitions do not apply to children that are not leaves.

Definition A *succinct AND-OR tree* is one in which:

(1) The tree contains a truth-value node (empty **and** or **or**) if and only if that is the only node of the tree.

(2) No **and** has an **and** for a child; no **or** has an **or** for a child.

(3) Every operation node has at least two children (unless (1) applies).

(4) No operation node has *inconsistent* or *redundant* children.

Definition A Boolean expression is *succinct* if it is in the form of a succinct AND-OR tree.

It is clear that any AND-OR tree may be transformed into a logically equivalent succinct AND-OR tree with no increase in the number of leaves. Consequently we shall use the succinct AND-OR tree as the starting point for analysis, and use the number of leaves as the measure of problem size.

3. Notation

We shall frequently denote the set of variables in a Boolean expression by $v_1,...,v_n$ and the literals by $x_1,...,x_n$. Thus $\tilde{v}_i$ is v_i negated; x_i means either v_i or $\tilde{v}_i$, and $\tilde{x}_i$ is the negation of x_i. Here we say v_i is *associated* with x_i and $\tilde{x}_i$, and *vice versa*. In the context of a particular discussion, either $x_i=v_i$ or $x_i=\tilde{v}_i$ (and $\tilde{x}_i=v_i$), but for purposes of analysis we usually do not care which. In some cases we use w, y and z for literals and u for variables, as well.

We denote the number of leaves in an expression by L, and take this as the basic measure of the length of the expression. If we say a *literal* x occurs k times in an expression, we mean that there are k leaves with value x and we imply nothing about leaves with value $\tilde{x}$. If we say a *variable* v occurs k times, we mean that there are k leaves with values of either v or $\tilde{v}$. Similarly, if we say a literal x occurs in a sub-tree, we mean x and not $\tilde{x}$; if we say a variable v occurs in a sub-tree, we mean either v or $\tilde{v}$.

We shall use some standardized symbols in diagrams of trees, as illustrated in Fig. 3.1. Circles denote operation nodes; boxes denote leaves; triangles denote subtrees with at least one operation node; a triangle within a box denotes a subtree that may be a leaf. For edges, a single line denotes one edge, a dashed line an optional edge (i.e., the subtree below it is optional). A double dashed line means zero or more edges, and a combination solid and dashed line means one or more edges. The symbol under either type of double line actually represents a set of such symbols. Therefore, in the illustration, A is a possibly null forest, B is completely optional, and

Figure 3.1. Illustration of Symbols in Tree Diagrams.

C is a non-null forest in which every top level tree has some leaf containing x. Note that names of items are outside the symbols, and contents inside. Finally, the two horizontal dashed lines connecting n_1 and n_2 indicate that they may be the same node; otherwise nodes are taken to be distinct.

4. Overview of the Algorithm

The top level recursive part of the algorithm is described here. It begins after an initialization phase in which the user's input is put into the form of a succinct expression and the variables in the expression are identified. In general, the input to each recursive instance is a succinct expression, a list of the variables in that expression, and a (variable, assignment) pair called the *pending assignment*. For the top level instance, the input expression is the original expression (but in succinct form) and the pending assignment is nil. The output of each recursive instance is "satisfiable" or "unsatisfiable", together with supporting documentation. One recursive instance of the algorithm does the following steps.

1. Make the pending assignment and simplify the resulting expression, producing the *base expression*. The base expression becomes the initial value of the *current expression*.
2. Repeat until no progress is made (terms are defined in Section 6): (a) Find dominances and prune the current expression accordingly. (b) Apply elementary resolution, if possible. (c) Find satisfiability preserving assignments, substitute them into the current expression, and simplify it into succinct form (Section 2).
3. Determine satisfiability of the current expression: do the first one that applies of (3a), (3b), and (3c).

3a. If the current expression has no more than a predetermined number of variables (5 in our implementation), then determine its satisfiability by the method of truth tables.

3b. If the root of the current expression is an **or**, then for each subtree of the root: recursively call this algorithm with inputs consisting of the subtree and the null assignment pair. The current expression is satisfiable if any subtree is satisfiable.

3c. Otherwise, choose a variable to "branch" on, say v. Recursively call this algorithm with inputs consisting of the current expression and the assignment (v,**true**). If it returns "unsatisfiable", then try it with the current expression and (v,**false**). If it again returns "unsatisfiable", then the current expression is indeed unsatisfiable.

4. Return "satisfiable" or "unsatisfiable" as determined in (3), together with supporting documentation. The supporting documentation consists of that returned by recursive calls, with this instance's assignments and reasons prepended.

5. Asymptotic Analysis of Worst Case Running Time

An algorithm like this is usually called "divide and conquer", but actually a better name is "chip and conquer", because instead of truly dividing the problem up, it only chips off a piece of constant size as it produces each subproblem. When all subproblems have equal size, the analysis is well known [AHU74] This section describes the analysis when the subproblems have varying sizes.

Let $T(L)$ be a worst case upper bound on running time for expressions with L leaves, i.e., with length L. Let the cost per recursive step of splitting up the base problem into subproblems, including the cost of applying satisfiability preserving assignments and of simplifying to succinct form, be bounded by $P(L)$. It will be easy to implement the algorithm so that $P(L)$ is quadratic. Most analysis and simplification steps can be made linear without much difficulty, but a few present problems, so we take $P(L)$ to be quadratic for asymptotic purposes. We shall show later that, depending upon the structure of the base expression, one of several cases occurs. Each case can be characterized by a small set of positive integers. Say that $A=\{a_1,...,a_{c_A}\}$ represents one case, that $B=\{b_1,...,b_{c_B}\}$ represents another case, etc. The meaning of the set A is as follows: For case A the base problem is split into c_A subproblems. The lengths of the subproblems are at most $L-a_1, L-a_2, ..., L-a_{c_A}$. In the worst case, all subproblems are of maximum possible length

and all must be solved. Running through the cases, we have:

$$T_A(L) \le P(L) + T(L - a_1) + ... + T(L - a_{c_A}) \tag{5.1}$$

$$T_B(L) \le P(L) + T(L - b_1) + ... + T(L - b_{c_B})$$

and therefore

$$T(L) \le \max(T_A(L), T_B(L), ...) \tag{5.2}$$

Let us advance the inductive hypothesis that there exist K and γ independent of L, whose values will be detemined subsequently, such that for all $m < L$:

$$T(m) < K\gamma^m \tag{5.3}$$

then

$$T_A(L) \le P(L) + \sum_{j=1}^{c_A} K\gamma^{L-a_j} = P(L) + K\gamma^L \sum_{j=1}^{c_A} \gamma^{-a_j} \tag{5.4}$$

Now consider the equation:

$$f_A(\gamma) \equiv \sum_{j=1}^{c_A} (\gamma^{-1})^{a_j} = 1 \tag{5.5}$$

It is clear that (for $c_A > 1$) this equation has precisely one real root in the range $0 < \gamma^{-1} < 1$, and no other positive roots. Call this root γ_A. Similarly we define γ_B,..., for all the other cases. Finally, define γ^* to be the maximum γ over all cases. The set of possible cases is a property of the *algorithm*, not the *problem*, as will be clarified later when the specifics of the algorithm are discussed. Consequently, γ^* also depends only on the algorithm. Because $P(L)$ is a polynomial, for any given $\varepsilon > 0$, we can choose a K_ε such that, for all L:

$$P(L) + K_\varepsilon(\gamma^* + \varepsilon)^L \; f_A(\gamma^* + \varepsilon) < K_\varepsilon(\gamma^* + \varepsilon)^L \tag{5.6}$$

A base case for Eq. 5.3 (enlarging K_ε if necessary) is clearly true, so for all L:

$$T(L) < K_\varepsilon(\gamma^* + \varepsilon)^L \tag{5.7}$$

□

For example, one naive algorithm is to substitute for variables that occur with only one polarity, if any, then pick a variable to "branch" on arbitrarily. There is only one case, and $A = \{2,2\}$ because there are two subproblems, each being at least 2 shorter than the base problem. Eq. 5.5 becomes $2\gamma^{-2} = 1$, and $\gamma^* = \sqrt{2}$.

In order to determine γ^* for the algorithm of this paper, it is necessary to delineate the cases and subproblems that arise. This is done in the following sections.

6. Satisfiability Preserving Assignments and Dominance Pruning

If we adopt the convention that **false** < **true** (cf. [End72] Sec. 1.5), then we may define functions over expressions as follows:

$$eval(E, A) = \text{truth value of } E \text{ with assignment } A \tag{6.1}$$

$$sat(E) = \max_A(eval(E, A)) \tag{6.2}$$

Assignment here means the assignment of a truth value to every variable in E. It is clear that $sat(E)$ is **true** precisely when E is satisfiable. We observe without proof that

$$eval(E_1 \textbf{ or } E_2, A) = \max(eval(E_1, A), eval(E_2, A)) \tag{6.3}$$

$$eval(E_1 \textbf{ and } E_2, A) = \min(eval(E_1, A), eval(E_2, A))$$

$$sat(E_1 \textbf{ or } E_2) = \max(sat(E_1), sat(E_2))$$

$$sat(E_1 \textbf{ and } E_2) = \min(sat(E_1), sat(E_2))$$

Now max and min are monotonically non-decreasing in each argument. Therefore, in an AND-OR tree, *eval* at each node is a monotonically non-decreasing function of *eval* at its children. The same is true of *sat*.

In the following discussion we shall abuse notation somewhat in order to avoid excessive verbiage by writing statements like:

$$eval(E, x = \textbf{true}) \geq eval(E, x = \textbf{false})$$

when E has other literals besides x. What we mean by this is that the inequality holds for any pair of assignments that only differ on x (and of course $\tilde{x}$).

The following lemma generalizes the pure literal rule of the Davis-Putnam procedure to non-clausal expressions.

Lemma 6.1 (Triviality Lemma) Let E be a succinct AND-OR tree in which the literal x occurs but $\tilde{x}$ does not occur. Then E is satisfiable if and only if it is satisfiable with an assignment that includes $x = \textbf{true}$.

Proof *If* is immediate. *Only if*: Proceed by induction from a leaf containing x to the root.

$$eval(x, x = \textbf{true}) \geq eval(x, x = \textbf{false})$$

For any subexpression S with subtrees S_i, suppose for all subtrees that:

$$eval(S_i, x = \textbf{true}) \geq eval(S_i, x = \textbf{false})$$

Then by Eq. 6.3 the same holds for S. That is, whenever E can be satisfied by an assignment that includes $x = \textbf{false}$, then the same assignment with $x = \textbf{true}$ also works. □

Definition For any leaf node n, let $family(n)$ be the set of leaves in (the fringe of) the subtree whose root is the parent of n.

Definition For any literal x, let $families(x)$ be the union of $family(n)$ over all leaves n that contain the literal x.

We note that any two subtrees either are disjoint or nested. Consequently, $families(x)$ can be represented as a disjoint union of $family(n_i)$ over some subset of leaves $\{n_i\}$ that contain x. Moreover, this subset is unique in succinct expressions. This leads to the following

Figure 6.1. Illustration of Dominance.

Definition For any literal x in a succinct expression E, the *defining subset* of x is the set of leaf nodes $\{n_i\}$ containing x such that the disjoint union of $family(n_i)$ is $families(x)$.

Definition (Dominance) Let E be a succinct AND-OR tree (see Fig. 6.1) with root node r and other operation node p, such that some child q of r is a leaf containing literal x, and some child n of p is a leaf containing x or $\tilde{x}$. Then we say q is a *dominant* node; q *dominates* n; and n is *dominated* by q.

Lemma 6.2 (Dominance Lemma) Let E, r, q, p, n, x be as in the preceding definition of dominance. (See Fig. 6.1.) Let E_1 be E with node n removed. Let E_2 be E with subtree p removed. Then for all assignments A:

(a) If r and p have the same operation and n contains x,

$$eval(E,A)=eval(E_1,A).$$

(b) If r and p have the same operation and n contains $\tilde{x}$,

$$eval(E,A)=eval(E_2,A).$$

(c) If r and p have different operations and n contains x,

$$eval(E,A)=eval(E_2,A).$$

(c) If r and p have different operations and n contains $\tilde{x}$,

$$eval(E,A)=eval(E_1,A).$$

Proof (a) Suppose r and p contain **and**. Any assignment with x=**false** makes both E and E_1 **false**, so assume x=**true**. Then $eval(p)$ is the same in both E and E_1, and the expressions are identical elsewhere. If r and p contain **or**, then the same argument applies with the roles of **true** and **false** interchanged.

(b) Suppose r and p contain **and**. Any assignment with x=**false** makes both E and E_2 **false**, so assume x=**true**. Then $parent(p)$ (a descendant of r) contains an **or** and $eval(p)$=**false**. Therefore, $eval(parent(p))$ is the same in both E and E_2, and the expressions are identical elsewhere. If r and p contain **or**, then $parent(p)$ contains **and**, and the same argument applies with the roles of **true** and **false** interchanged.

(c) The argument is similar to (b). (d) The argument is similar to (a). □

The unit clause rule of the Davis-Putnam procedure is a special case of the above Dominance Lemma.

The next lemma allows us to identify situations in which two variables may be collapsed into one. For expressions in clause form this lemma is not needed: whenever its conditions apply, the Dominance Lemma also applies and is more powerful. Consider this simple example:

> Let E be a succinct AND-OR tree in which the only occurrences of $x, \tilde{x}, y, \tilde{y}$ are as follows: some subtree contains x **and** y **and** ... and some other subtree contains $\tilde{x}$ **or** $\tilde{y}$ **or** ....

Here x and y could be represented by a single literal with $z=x$ **and** y, $\tilde{z}=\tilde{x}$ **or** $\tilde{y}$. Looking more closely, we see that we can simply assign y=**true**, identify the new literal z with x, and preserve satisfiability.

Definition A node is said to be *under* an **and** or **or**, if the parent of that node contains **and** or **or**, respectively.

Definition A literal x in a succinct expression E is said to be *symmetric* (in E) if the following conditions hold:

(a) Let $\{n_i, i=1,...,k\}$ be the defining subset of x. Then $parent(n_i)$ contains **and** for $i=1,...,k$.

(b) Let $\{m_i, i=1,...,p\}$ be the defining subset of $\tilde{x}$. Then $parent(m_i)$ contains **or** for $i=1,...,p$.

Definition Let variable v be associated with literals x and $\tilde{x}$ in succinct expression E; i.e., either $x=v$ or $\tilde{x}=v$. Then v is said to be symmetric (in E) if either x or $\tilde{x}$ is symmetric in E.

Definition A variable is said to be *mixed* if it is not symmetric.

The qualification "in E" will be omitted where the meaning is clear without it. The motiviation for the term "symmetric" lies in the effect of assignments to x on the main subexpressions containing x and $\tilde{x}$, i.e., those rooted at the parents of their defining subsets. If x=**true**, none of these subexpressions necessarily become resolved (i.e., evaluate to **true** or **false**); if x=**false**, the defining parents of x resolve to **false**, while the defining parents of $\tilde{x}$ resolve to **true**. Thus the resolving effect of each assignment is symmetric between x and $\tilde{x}$.

Lemma 6.3 (Symmetric Collapsibility Lemma) Let x, y be symmetric literals in succinct expression E, and let $families(x)=families(y)$ and $families(\tilde{x})=families(\tilde{y})$. Then E is satisfiable if and only if it is satisfiable with an assignment containing y=**true**.

Proof *If* is immediate. *Only if*: Let $\{n_i, i=1,...,k\}$ be the defining subset of x, and let $\{m_j, j=1,...,p\}$ be the defining subset of $\tilde{x}$. Let S_i be the subexpression rooted at $parent(n_i)$, and let R_j be the subexpression rooted at $parent(m_j)$. Then S_i all contain **and** at their roots, and R_j all contain **or** at their roots. Suppose some assignment including y=**false** satisfies E. Since x and y have the same *families* and the same is true for $\tilde{x}$ and $\tilde{y}$, it follows for all applicable i, j that:

$$eval(S_i, x=\textbf{false and } y=\textbf{true})=\textbf{false}=eval(S_i, y=\textbf{false}),$$

$$eval(R_j, x=\textbf{false and } y=\textbf{true})=\textbf{true}=eval(R_j, y=\textbf{false}).$$

But $x, \tilde{x}, y, \tilde{y}$ do not effect $E-\cup S_i-\cup R_j$, so:

$$eval(E, x=\textbf{false and } y=\textbf{true})=eval(E, y=\textbf{false}).$$

Consequently, E is also satisfiable by an assignment containing y=**true**. □

The next two lemmas are important because, unlike the preceding one, they can apply to expressions in conjunctive normal form (CNF).

Lemma 6.4 (Mixed Collapsibility Lemma) Let x, y be mixed literals in succinct expression E, and let $families(y) \subseteq families(x)$ and $families(\tilde{y}) \subseteq families(\tilde{x})$.

(a) If all nodes in the defining subsets of y and $\tilde{y}$ are under **or**s (see Fig. 6.2), then E is satisfiable if and only if it is satisfiable with an assignment in which $y=\tilde{x}$. In this case, all those **or**s may be replaced by the constant **true**.

(b) If all nodes in the defining subsets of y and $\tilde{y}$ are under **and**s, then E is satisfiable if and only if it is satisfiable with an assignment in which $y=x$. In this case, all occurrences of y and $\tilde{y}$ may be pruned from E.

Figure 6.2. Illustration of Mixed Collapsibility.

Proof *If* is immediate. *Only if*: Let $\{n_i, i=1,...,k\}$ be the defining subset of y, and let $\{m_j, j=1,...,p\}$ be the defining subset of $\tilde{y}$. Let S_i be the subexpression rooted at $parent(n_i)$, and let R_j be the subexpression rooted at $parent(m_j)$.

Case (a): S_i and R_j all contain **or** at their roots. It follows for all applicable i, j that:

$$eval(S_i, y=\tilde{x}) = \textbf{true} \geq eval(S_i, y=x),$$

$$eval(R_j, y=\tilde{x}) = \textbf{true} \geq eval(R_j, y=x).$$

But $y, \tilde{y}$ do not effect $E - \cup S_i - \cup R_j$, so:

$$eval(E, y=\tilde{x}) \geq eval(E, y=x).$$

Consequently, if E is satisfiable, then it is satisfiable by an assignment in which $y=\tilde{x}$.

Case (b): S_i and R_j all contain **and** at their roots. It follows for all applicable i, j that:

$$eval(S_i, y=x) \geq \textbf{false} = eval(S_i, y=\tilde{x}),$$

$$eval(R_j, y=x) \geq \textbf{false} = eval(R_j, y=\tilde{x}).$$

But $y, \tilde{y}$ do not effect $E - \cup S_i - \cup R_j$, so:

$$eval(E, y=x) \geq eval(E, y=\tilde{x}).$$

Consequently, if E is satisfiable, then it is satisfiable by an assignment in which $y=x$. With this substitution, the nodes that used to contain y and $\tilde{y}$ become redundant, so may be pruned. □

The next lemma is useful mainly on expressions in CNF, although it is stated in more generality. When applicable, it shortens the expression by two literals and removes one variable. Its repeated application to a CNF expression eliminates all nontrivial variables with only two occurrences.

Lemma 6.5 (Elementary Resolution Lemma) Let the only occurrences of literals x and $\tilde{x}$ in succinct expression E be in the subexpression $(x \textbf{ or } A) \textbf{ and } (\tilde{x} \textbf{ or } B)$, where A and B are any subexpressions. Construct E_1 by replacing that subexpression by $(A \textbf{ or } B)$. Then E is satisfiable if and only if E_1 is.

Proof The original subexpression is satisfiable if and only if at least one of A and B is. □

By checking for satisfiability preserving rules, we give ourselves the opportunity to reduce the problem size while creating just *one* subproblem. Such rules do not contribute to exponential growth. However, the most important feature of the satisfiability preserving rules embodied in these lemmas is that they enable the exponential growth rate to be *reduced* by attacking some of the worst cases. This idea is developed in the next section.

7. Branching Rules of the Algorithm

In this section we consider a succinct expression in which none of the lemmas of the previous section apply, and investigate how to choose a variable to "branch" on in such a way that the exponential growth rate is low. The basic approach is to try to make the reduction in problem size high. The analysis involves a fairly tedious enumeration of cases. However, the end result is that a worst case growth rate of less than $2^{(.25+\varepsilon)L}$ can be achieved.

We note that whenever the operation at the root of the expression is an **or**, then it suffices to solve each subtree of the root independently. Although this possibility must be programmed, it cannot contribute to exponential growth, so for this analysis we assume that the operator at the root is an **and**.

The "branch" variable is of course the variable that is assigned **true** in one subproblem and **false** in the other. In general, each assignment reduces its subproblem's size by some constant, and both subproblems need to be solved to get the solution to the base problem. We choose the branch variable by means of a prioritized list of rules, A, B, C, and D, where A has highest priority. The rule of highest priority whose requirements are met is the one to be applied. With each

rule we associate a case of the same name, i.e., case A, case B, etc. Within cases there may be subcases that depend on further particulars of the expression. We will use the fact that case B only arises when case A does not, and so on, without mentioning it again. For each case (or subcase) we seek the list of positive integers that characterizes the worst case reductions in subproblem size for that case. The worst case is that combination that results in the highest value of γ in the solution of Eq. 5.5. In our analysis, all cases produce two subproblems, so are characterized by by two such positive integers, one corresponding to v=**true** and the other to v=**false**. In general these two integers are not equal. A case would involve more than two subproblems were the analysis to carry two recursive levels down, in order to derive a tighter bound.

Now we summarize the rules used by the algorithm, and what "branch" variable is chosen in each rule. In this summary, x and $\tilde{x}$ will always represent the literals associated with variable v. Recall the definitions of *mixed* and *symmetric* preceding Lemma 6.3.

Rule A Some variable occurs more than 3 times. For each variable v, let $score(v)$ equal the product of the number of occurrences of x and the number of occurrences of $\tilde{x}$. Choose a variable v with maximum *score* as the "branch" variable. If more than one such variable exists, prefer mixed to symmetric.

Rule B Some variable occurs 3 times. Choose one such variable v as the "branch" variable. If more than one such variable exists, prefer mixed to symmetric.

The remaining cases concern expressions in which every variable occurs exactly twice.

Rule C Some variable is mixed. Choose one such variable v as the "branch" variable.

Rule D All variables are symmetric. Choose any variable v as the "branch" variable.

In the following theorem, we analyze the worst case reductions in size that result from using the preceding prioritized list of rules.

Before proceding to the theorem, we present two lemmas. The first sometimes allows us to select the worst case among alternatives without solving Eq. 5.5 for each alternative. The second establishes a simple lower bound on the reduction in length.

Lemma 7.1 Let integers a,b,c,d be such that $a+b=c+d$ and $0<a<c\le d<b$. Let $\gamma(m,n)$ be the positive real solution of

$$f_{mn}(\gamma)\equiv\gamma^{-m}+\gamma^{-n}=1.$$

Then $\gamma(a,b)>\gamma(c,d)$. In other words, between two cases that achieve the same total reduction in subproblem lengths, the more extreme case is the worst.

Proof We restrict our attention to the region of interest, i.e., everything positive. Here f_{mn} is decreasing in γ. Let $e=(m-n)/2$. Then

$$f_{mn}(\gamma)=\gamma^{\frac{m+n}{2}}(\gamma^{e}+\gamma^{-e})$$

For $m+n$ and γ held constant this expression has a unique minimum at $e=0$. Therefore (with $m+n$ held constant), as e increases in absolute value, γ must also increase to maintain the value of f_{mn} at 1. □

Lemma 7.2 If variable v occurs k times in a succinct AND-OR tree with no dominant nodes, and it is chosen as the branch variable, then the sum of the reductions in the two subproblems is at least $3k$.

Proof Each node n in which v or $\tilde{v}$ occurs is in the appropriate defining subset. There must be some other leaf in $family(n)$; it contains neither v nor $\tilde{v}$ or n would be dominant. One assignment to v eliminates n and the other eliminates $family(n)$, for a total of at least 3 among both assignments. □

Theorem 7.3 Let the algorithm outlined in Section 4 choose the "branch" variable according to rules A, B, C, and D stated above. Then the resulting cases can be characterized by the following sets of integers.

$A=\{4,8\}$

$B=\{3,6\}$
$C=\{4,4\}$
$D=\{2,8\}$

Proof In this proof, x and $\tilde{x}$ will always represent the literals associated with variable v.

Case A: Some variable occurs more than 3 times. Rule A is applied to choose variable v. Each assignment to v reduces the expression length by at least 4. By Lemma 7.2, the sum of the reductions for both assignments is at least 12. By Lemma 7.1, the worst possibility is that one assignment reduces by 4 and the other by 8. Therefore $A=\{4,8\}$.

Case B: Some variable occurs exactly 3 times. Rule B is applied to choose one such variable v. Without loss of generality, assume x occurs twice and $\tilde{x}$ once. There are subcases depending on what operations x and $\tilde{x}$ are under. Fig. 7.1 illustrates a typical subcase. In subcases B3 and B6 v is symmetric; in the others it is mixed.

Subcase B1: Both x are under **and**, and $\tilde{x}$ is under an **and**. Each assignment to x reduces by at least 4 because either $families(x)$ or $families(\tilde{x})$ disappears, as well as the leaves containing x and $\tilde{x}$. But by Lemma 7.2 the sum of reductions is at least 9. Therefore, $B1=\{4,5\}$.

Subcase B2: One instance of x, say n_1, is under an **and**, the other, n_2, is under an **or**, and $\tilde{x}$ is under an **and**. Each assignment to x reduces by at least 4 because either $family(n_1)$ or $family(n_2)$ disappears, as well as the leaves containing x and $\tilde{x}$. But by Lemma 7.2 the sum of reductions is at least 9. Therefore, $B2=\{4,5\}$.

Subcase B3: Both x are under **and**, and $\tilde{x}$ is under an **or**. Consider the assignment x=**true**. Both instances of x disappear, as well as $\tilde{x}$, for a total reduction of at least 3 leaves. Now consider x=**false**. All of $families(x)$ disappears, plus $families(\tilde{x})$, for a total reduction of at least 6 nodes. That is, $B3=\{3,6\}$.

B4 Both x are under **or**, and $\tilde{x}$ is under an **or**. This is similar to B1.

B5 One instance of x is under an **and**, the other is under an **or**, and $\tilde{x}$ is under an **or**. This is similar to B2.

B6 Both x are under **or**, and $\tilde{x}$ is under an **and**. This is similar to B3 with **true** and **false** interchanged.

Therefore, applying Lemma 7.1, we conclude that $B=\{3,6\}$.

The remaining cases concern expressions in which every variable occurs exactly twice, and hence every *literal* occurs exactly once with each polarity.

Case C: Some variable is mixed. Apply Rule C to choose one such variable v as the "branch" variable. The parents of x and $\tilde{x}$ must both contain the same operation.

Subcase C1: Both x and $\tilde{x}$ are under an **or**. Consider the assignment x=**true**. Then $families(x)$ and $\tilde{x}$ disappear for a total of at least 3 nodes. But since every variable occurs exactly twice, if an odd number of nodes disappear, some remaining variable occurs once and can be eliminated using the Triviality Lemma. Therefore, the reduction is at least 4 nodes. A similar argument applies to the assignment x=**false**.

Subcase C2: Both x and $\tilde{x}$ are under an **and**. This is similar to C1 with **true** and **false** interchanged.

Therefore, we conclude that $C=\{4,4\}$.

Case D: All variables are symmetric. Apply Rule D to choose any variable v as the "branch" variable. Without loss of generality, assume that all literals are named so that x_i is under an **and** and $\tilde{x}_i$ is under an **or**.

Subcase D1: At least one of $families(x)$ and $families(\tilde{x})$ contains more than two leaf nodes. First let $families(x)$ contain 3 or more nodes. With the assignment x=**true**, we can only count on x and $\tilde{x}$ disappearing for a reduction of 2 nodes. Consider the assignment x=**false**. Now both $families(x)$ and $families(\tilde{x})$ disappear, but in addition there is a

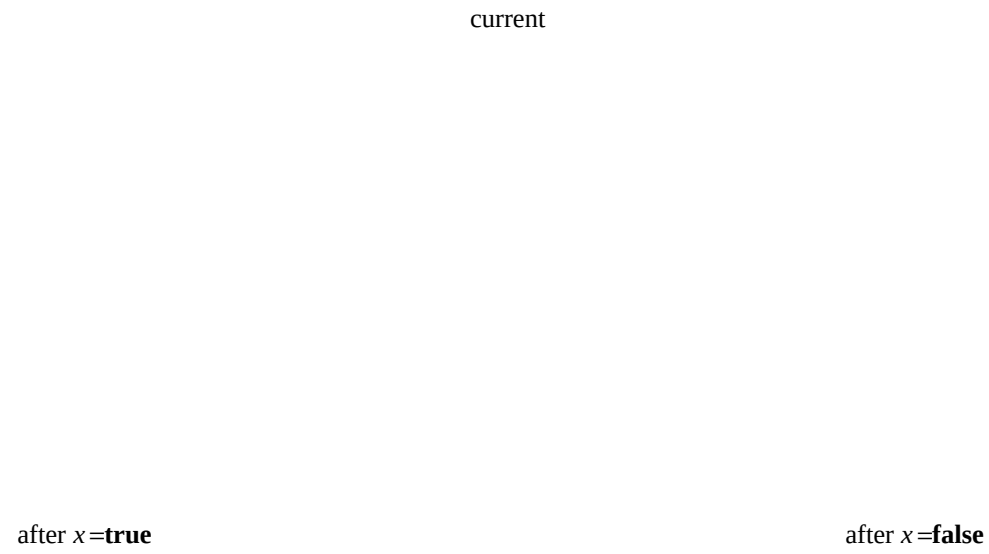

Figure 7.1. Case B2

domino effect. If $families(x)$ contains 5 nodes, then 7 disappear plus an unpaired one that can be eliminated by the Triviality Lemma, for a total of 8; so consider smaller cases. Let $families(\tilde{x})=\{\tilde{x},\tilde{w}\}$. (When it is larger, a reduction of 8 is easy to show using a similar argument.) Now examination of all possibilities in which $families(x)$ has 3 or 4 nodes, (keeping in mind the restriction to all symmetric variables, no dominance, and no collapsibility) reveals that there must be at least two *additional* literals (not complements of $\tilde{w}$ or each other) that occur in $families(x)$. (See Fig. 7.2.) When one literal associated with a variable disappears, the other can be eliminated by the Triviality Lemma, so in all, at least 4 variables, hence 8 literals disappear. Thus $D1=\{2,8\}$.

Subcase D2: Both $families(x)$ and $families(\tilde{x})$ contain exactly two leaf nodes. As in D1, the assignment v=**true** produces a reduction of 2 nodes. Consider the assignment v=**false**. Let w be the other literal in $families(x)$, and let $\tilde{y}$ be the other literal in $families(\tilde{x})$. (See Fig. 7.3.) (w and y cannot be the same literal or they would be collapsible into x.) Both w and $\tilde{y}$ disappear, so $\tilde{w}$ and y become trivial, and may be assigned **true**. But $\tilde{w}$ is under an **or**, so assigning **true** to it causes $families(\tilde{w})$ to disappear. Also, y disappears. If $families(\tilde{w})$ contains 3 or more nodes, this gives an immediate reduction of 7 or more, but exactly 7 would allow elimination of the unpaired node also, for a total reduction of 8. If $families(\tilde{w})$ contains 2 nodes, let $\tilde{z}$ be the other node, as shown in the diagram. It disappears, so z becomes trivial and disappears. This again brings the reduction to 8. Thus D2 achieves the same reductions as D1.

Therefore, we conclude that $D=\{2,8\}$, concluding the proof. □

Figure 7.2. Possibilities for $families(x)$ in Subcase D1.

Figure 7.3. Illustration of Subcase D2.

We should emphasize that this theorem provides an upper bound only, which is not necessarily "tight". A tighter bound might be possible by exploring the worst cases above more carefully, including looking at the next level of recursion.

We now turn to the evaluation of γ^*, the exponential growth rate, as defined following Eq. 5.5.

Corollary 7.4 The algorithm's running time is $O(2^{(.25+\varepsilon)L})$.

Proof By Lemma 7.1, $\gamma_A < \gamma_B$, and obviously $\gamma_C = 2^{.25}$, so it remains to compare B and D to C. The defining equations are:

$$\gamma_B^{-6} + \gamma_B^{-3} = 1 \tag{7.1}$$

$$\gamma_D^{-8} + \gamma_D^{-2} = 1 \tag{7.2}$$

The solutions to five decimal places are $\gamma_B = 2^{.23141}$ and $\gamma_D = 2^{.23248}$. So $\gamma^* = \gamma_C$. □

Corollary 7.5 The Davis-Putnam procedure has a worst case upper bound of $O(2^{(.25+\varepsilon)L})$ when the input is presented in clause form.

Proof Lemmas 6.4 and 6.5 were not used to prove Theorem 7.3. Lemma 6.1 applied to expressions in clause form is equivalent to the pure literal rule. Lemmas 6.2 and 6.3 applied to expressions in clause form are equivalent to the unit clause rule. Without Lemmas 6.4 and 6.5, the algorithm here becomes equivalent to the Davis-Putnam procedure. □

What happens when the algorithm is used *with* Lemmas 6.4 and 6.5 on CNF expressions? A substantial improvement in growth rate can be guaranteed.

Theorem 7.6 Let the expression to be solved be in conjunctive normal form. Let the algorithm outlined in Section 4 choose the "branch" variable according to rules A, B, C, and D stated above. Then cases C and D will not occur, and cases A and B can be characterized by the following sets of integers.

$A = \{6,10\}$
$B = \{7,9\}$

Proof Lemmas 6.1 and 6.5 guarantee that branching occurs only on expressions in which every variable occurs at least three times, so cases C and D do not occur. If the "branch" variable v (whose literals are denoted by x and $\tilde{x}$, as in Theorem 7.3) occurs in any two-literal clause, say (x **or** y), then y will become a unit clause in one of the two branches, and allow further reductions in that branch by Lemma 6.2. Case by case examination shows that these reductions are always more favorable than when v is not in any two-literal clauses. Note that Lemma 6.4 rules out the case in which the only occurrences of x, y, and their complements are (x **or** y), (x **or** y) (*sic*), and ($\tilde{x}$ **or** $\tilde{y}$). Another critical case is (x **or** y), (x **or** $\tilde{y}$), and ($\tilde{x}$ **or** y). In this case v=**false** produces an expression with contradictory unit clauses, y and $\tilde{y}$, which is solved immediately, so for practical purposes there was only one branch for v. Details of other cases for two-literal clauses are omitted. Now we assume that v occurs only in clauses of three or more literals.

Case A: When v has two positive and two negative occurrences, reductions of 8 for each assignment are straightforward. When v has three occurrences of one polarity, say positive, and one of the complement, then reductions of 10 for v=**true** and 6 for v=**false** are immediate. Thus by Lemma 7.1 A={6,10}.

Case B: Since all variables have exactly three occurrences, those that occur in the same clauses as x are reduced to two occurrences when v=**false** and those that occur in the same clauses as $\tilde{x}$ are reduced to two occurrences when v=**true**. Such variables are eliminated without branching by Lemma 6.1 or 6.5. Including these reductions, B={7,9}. □

Corollary 7.7 The algorithm's running time is $O(2^{.128L})$ when the expression is in conjunctive normal form.

Proof By Lemma 7.1, $\gamma_A > \gamma_B$, and its defining equation is

$$\gamma_A^{-6} + \gamma_B^{-10} = 1 \tag{7.3}$$

Its solution to five decimal places is $\gamma_A = 2^{.12782}$, hence $\gamma^* < 2^{.128}$. □

8. Empirical Results

A program implementing this algorithm was written in Franz Lisp. Non-clausal expressions ranging in size from 48 to 106 literals and 12 to 32 variables were solved and timed on a Digital Equipment Vax 11/780. (Transforming to clause form would have yielded substantially larger sizes.) We tried to make these difficult cases for the algorithm by limiting initial opportunities for it to apply the important lemmas, but they certainly do not represent worst cases. (Other tests, made up without inspecting the algorithm, were an order of magnitude faster, and are not reported.)

For comparison, most expressions were also solved by Quine's method (the same program with the Dominance and Collapsibility Lemmas (6.2, 6.3 and 6.4) disabled, but the Triviality Lemma (6.1) in force). Tests results are summarized in the table below.

Expression Size		Main Algorithm			Quine Method		
Literals	Variables	Assignments		CPU Sec	Assignments		CPU Sec
48	12	18	($2^{4.2}$)	12	31	($2^{5.0}$)	11
61	20	28	($2^{4.8}$)	32	242	($2^{7.9}$)	98
72	24	44	($2^{5.5}$)	38	2370	($2^{11.2}$)	724
106	32	310	($2^{8.3}$)	314		?	

Table 8.1. Results of Tests.

The program discontinues branching when expressions reduce to five or fewer variables, using a direct truth table instead; therefore the branching figures should be interpreted relatively, not absolutely. The larger tests suggest a growth rate of $2^{\frac{L}{12}}$, but the data is no where near sufficient to support this as a firm conclusion.

9. Conclusion and Acknowledgement

The investigation of worst cases in a naive enumeration algorithm has led to the discovery of additional rules that improve performance. The rules embodied in Lemmas 6.4 and 6.5 can be added to resolution and Davis-Putnam procedures, producing proof procedures that have not been shown to be exponential. The connection between this rule and the extension rule of extended resolution [Tse68, Gal77] should be examined.

This research was supported in part by AFOSR grant 80-0212 while the author was at Stanford University. I wish to thank Tom Spencer for interesting test cases, Joe Weening for implementation advice, and Vaughn Pratt for helpful discussions.

10. References

AHU74.
Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The design and analysis of computer algorithms,* Addison-Wesley, Reading, MA (1974).

Coo71.
Cook, S.A., ‘‘The complexity of theorem-proving procedures,’’ in *Proc. Third Annual ACM Symposium on Theory of Computing*, (1971).

DaP60.
Davis, M. and Putnam, H., ‘‘A computing procedure for quantification theory,’’ *JACM* **7**(2?) pp. 201-215 (1960).

End72.
Enderton, H.B., *A Mathematical Introduction to Logic*, Academic Press, New York, NY (1972).

Gal75.
Galil, Z., ‘‘The complexity of resolution procedures for theorem proving in the propositonal calculus,’’ TR 75-239, Department of Computer Science, Cornell University (1975).

Gal77.
Galil, Z., ‘‘On the complexity of regular resolution and the Davis-Putnam procedure,’’ *Theoretical Computer Science* **4** pp. 23-46 (1977).

Gol79.
Goldberg, A.T., ‘‘On the complexity of the satisfiability problem,’’ NSO-16, Courant Institute of Mathematical Sciences, New York University (1979).

Qui50.
Quine, W.V., *Methods of logic,* Henry Holt (1950).

Tse68.
Tseitin, G.S., ‘‘On the complexity of derivations in the propositional calculus,’’ pp. 115-125 in *Studies in Constructive Mathematics and Mathematical Logic, Part II*, ed. Slisenko, A.O., (1968).

A Satisfiability Tester for Non-Clausal Propositional Calculus

Table of Contents