A POLYNOMIAL BOUND ON THE COMPLEXITY
OF
THE DAVIS–PUTNAM ALGORITHM
APPLIED TO SYMMETRIZABLE PROPOSITIONS

Robert Vail Harris, Jr.
TR 72-131

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University for the Degree of
Doctor of Philosophy

August 1972

Department of Computer Science
Cornell University
Ithaca, New York 14850
Biographical Sketch

The author was born January 27, 1947, in Danbury, Connecticut. He was initiated into the Phi Eta Sigma freshman honorary fraternity in the spring of 1965. He received a B.S. in physics from the University of Wisconsin at Madison in June, 1968, and an M.S. in applied mathematics from Cornell University in June, 1971. He has had summer work experience in infrared spectroscopy and in the design and construction of miniature microwave transformers and ultrasonic transducers. An abstract of his thesis results has been published in Notices of the AMS, SIGACT, and SIGART.
Acknowledgments

to Professors

Stephen Cook, of the University of Toronto, whose work inspired me to attack this topic, and who read an early draft of this thesis;
Robert Constable, of Cornell University, my advisor, whose grant supported me and whose help has been invaluable; and
John Hopcroft, of Cornell University, who suggested Theorem 3 and made a remark which eliminated a tedious proof.
# Table of Contents

Title Page

Biographical Sketch

Acknowledgments

Table of Contents

Introduction

A survey of the problems

Summary of proof

Chapter 1

The Davis-Putnam algorithm: a method for determining satisfiability of propositional formulae in conjunctive normal form.

The subsumption rule

Chapter 2

DPDNF: An algebraization of the Davis-Putnam algorithm, applied to tautology detection in disjunctive normal form propositions

Chapter 3

Symmetrical formulae

Characterization in terms of "clause-types"

The effect of DPDNF on the clause-types

Chapter 4

A polynomial bound on the Davis-Putnam algorithm applied to symmetrical propositions

Chapter 5

Extension of the bound to symmetrizable formulae

Symmetry is p-equivalent to tautology
Glossary

Appendix

Some sidelights

Computer programs and examples

Bibliography
A POLYNOMIAL BOUND ON THE COMPLEXITY
OF
THE DAVIS-PUTNAM ALGORITHM
APPLIED TO SYMMETRIZABLE PROPOSITIONS
Robert Vail Harris, Jr., Ph.D
Cornell University, 1972

The problem considered is the relation of minimal computation space to minimal computation time; more specifically, it is desired to determine the time complexity of the tautology problem for propositional logic; the space complexity is known to be linear. This is of especial interest in view of Stephen Cook's p-reducibility theorem (May, 1971 ACM Symposium).

Algorithms considered are the analytic tableau method, which appears to be inalterably exponential in space and time, and the Davis-Putnam algorithm, part of a resolution procedure introduced in JACM in 1960. Although the algorithm as stated has been shown to be exponential, the known examples are disposed of by (1) locally optimizing the choice of elimination variable and (2) using the subsumption rule (absorption rule), which asserts that if any clause is identical to a subclause of another clause, the larger clause may be deleted.
An algebraic notation is developed which makes it clear that Davis and Putnam's Rules 1 and 2 are special cases of Rule 3 plus local optimization and subsumption. The algebraic notation consists of using vectors to represent clauses; the ith element of the vector is equal to +, -, 0, or 1, according as the ith variable occurs positively, negatively, or not at all. This notation is used in an APL computer program; by using 01, 10, 00, and 11 to represent +, -, 0, and 1, the program can be written entirely in terms of APL logical operators on arrays, with loops needed only because of size limitations, at a considerable savings in time and space compared to a list representation of the formula, for up to about twenty variables.

The proof in brief of the polynomial special-case bound: Call a DNF formula "reduced" if the subsumption rule will delete no clauses; call it "symmetric" if it is invariant (as a set of clauses) under interchange of variable names. For any clause R, let $R^+$ be the number of positive literals in R and $R^-$ the number of negated literals. Call $(R^+, R^-)$ the (clause-)type of R. A symmetric formula A then satisfies the property that if $(R^+, R^-)$ is the type of some clause in A, then every clause of type $(R^+, R^-)$ is also a clause of A.

An algebra of clause-types is developed; it is easily shown that if $r = (R^+, R^-)$ and $s = (S^+, S^-)$ are unequal types of a reduced symmetric DNF formula (RSDNF), then either
$R^+ \succ S^+$ and $R^- \prec S^-$, or $R^+ \prec S^+$ and $R^- \succ S^-$. The former pair is denoted "$r \oslash s$"; $\oslash$ is a strict ordering of clause-types, which is total for RSDNF's. If $r \oslash s$ and for no $p$ of $E$ is $r \oslash p \oslash s$, say $r \mathsf{adj} s$ (in $E$).

Denote by $\text{PROD}(A)$ the result of one iteration of the Davis-Putnam algorithm with subsumption. Let $E$ be an RSDNF in $V$ variables. Then $t$ is a type of $\text{PROD}(E)$ iff either (1) $t$ is a type of $E$ (with $t^+ + t^- \leq V-1$), and for no $x$ or $y$ is $(t^+ + 1, x)$ or $(y, t^- + 1)$ a type of $E$; or (2) $t = (r^+-1, s^- - 1)$, with $r \mathsf{adj} s$ in $E$. It is further shown that if $r \mathsf{adj} s$ in $\text{PROD}(E)$, then either $r^+ = s^+ + 1$, or $s^- = r^- + 1$, or $(r^+-1, s^- - 1)$ is not a type of $\text{PROD}(\text{PROD}(E))$. A simple combinatorial argument then shows that each type of $\text{PROD}(\text{PROD}(E))$ has fewer clauses than some type of $\text{PROD}(E)$. The polynomial bound is then immediate, since there are at most $V+1$ types in any RSDNF formula in $V$ variables, and the number of variables decreases with each iteration, and there can be at most $V$ iterations. The extension to symmetrizable is trivial.
Introduction

Computational Complexity
Algorithmic Efficiency
Outline
Time and Space
Tautology Problem
Polynomial Reducibility
Tautology Algorithms
Complexity of the Davis-Putnam Algorithm
Complexity of the Davis-Putnam Algorithm on Symmetrizable Formulae
Conclusion
Computational Complexity

The first two questions about any interesting computational process are, "How difficult is it?" and "Is there a better way?" The generalization and abstraction of these two questions has produced a large and active body of knowledge and research, known as the theory of computational complexity. While this theory has produced many interesting results, roughly paralleling those of the theory of degrees of unsolvability in recursive function theory (mathematicians' name for the theory of computation), there have been no significant practical theorems. For example, in the realm of theory, it is known that there are arbitrarily difficult problems for which no best computer program can exist; but in the practical realm, it is not known whether prime-number recognition can or cannot be carried out in any better way than by using the definition directly.

The practical, case-by-case approach to computation has produced viable algorithms and made possible the solution of many hitherto unapproachable problems, such as constructing a highly efficient, unified reservoir and aqueduct system to provide power and water to hundreds of towns. Yet this approach has not produced general theorems; there are isolated instances of exact time-bounds (for example in sorting and in polynomial evaluation), but these do not seem to fit in a larger pattern.
Algorithmic Efficiency

There must nonetheless be general theorems about algorithms. The search for these theorems, and the scant progress along either purely theoretical or strictly practical lines, has led to development in recent years of a recognizable intermediary field, which one might call "algorithmic efficiency". Algorithmic efficiency is concerned with general yet practical principles and methods applicable to studying specific computable problems. It is hoped that breakthroughs will be made in this field toward solving the fundamental problems about algorithms. This hope has recently been encouraged by some results of Stephen Cook. He has used the methods of automata and complexity theories to obtain valuable insight about the structure of classes of practical combinatorial problems.

This thesis is largely inspired by Cook's results, although it does not depend upon his work, nor employ the methods of automata or complexity theory.

Outline

In order to set my work in proper perspective, I will use the remainder of the introduction to survey some aspects of algorithmic efficiency. After defining notions of computation size and computation time, I will discuss the problem of recognizing tautologies in propositional logic. I will then present a result of Cook's which shows that any "reasonable" problem can be solved quickly if you have a quick way of recognizing tautologies in disjunctive
normal form propositions. Finally, I will outline my proof that the Davis-Putnam algorithm operates in polynomial time on any symmetrizable formula. It is my hope that this special case will shed light on the question of tautology recognition and of practical combinatorial problems in general.

**Time and Space**

One of the general problems of algorithmic efficiency is the relation between the physical size of a computation and the temporal length of a computation. This problem can be stated as follows: no computer program using \( n \) bits of storage can run for longer than \( 2^n \) machine cycles, or it must contain an endless loop (that is, the program never stops). Certainly one can write programs that use just this much time; for example, count by ones to \( 2^n \), using binary representation. However, can one exhibit a problem whose smallest-sized solution uses \( n \) bits of storage, but whose fastest solution uses at least \( 2^{f(n)} \) machine cycles, for some \( f \) such that \( \lim_{n \to \infty} \frac{\log(n)}{f(n)} = 0 \)?

Note that \( n \) must depend on the input to the problem as well as on the problem itself; a more precise formulation is as follows:

Let \( P \mid X + Y \), where \( X \) is an infinite set of inputs and \( Y \) is any set (finite or infinite) of outputs. Let \( P(x) = y \). We say that \( P \) is a problem about \( x \) whose answer is \( y \). Let \( M \) be a program which given input \( x \),
yields output \( y \) (that is, \( M \) is a program for \( P \), or, \( M \) computes \( P \)). Let \( S_M(x) \) be the number of bits (amount of space) used by \( M \) to compute \( P(x) \). Let \( t_M(x) \) = the number of machine cycles (amount of time) used by \( M \) to compute \( P(x) \). What we want to know is, given any \( M \) that computes \( P \), is there an \( M' \) that computes \( P \), and an \( f \), with

\[
\lim_{n \to \infty} \frac{\log(n)}{f(n)} = 0, \text{ such that } \\
t_M(x) \geq 2^{f(S_{M'}(x))} \quad (\text{for all } x).
\]

If \( P \) is such that for every \( M \), one can find such an \( M' \) and such an \( f \), then \( P \) is an "exponential problem"; otherwise \( P \) is a "polynomial problem". It is of considerable interest to discover whether there are any exponential problems.

It should be noted that there is a related problem whose answer is known: Namely that for every \( f \), one can find a \( P \) such that if \( M \) computes \( P \), then

\[
S_M(x) \geq f(lg(x)) \text{ and consequently } \\
t_M(x) \geq f(lg(x)), \text{ where } lg(x) \text{ means the number of bits in } x. \text{ This says nothing about the relation between } S_M \text{ and } t_M, \text{ but it does show that there are functions that are inherently difficult to compute. Another related problem concerns the functions } f: \{0,1\}^n + \{0,1\}. \text{ Given } n \text{ and } f, \text{ one can write a program for } f \text{ in at worst } 2^n \text{ bits of storage (really more like } n+2^n \text{ bits); but almost every } f \text{ needs this maximum! Of course, nobody has found much use for such "complicated" functions; or even exhibited one complicated function for } n \geq 5;
but this result, and similar results in recursive function
theory, suggest that the same situation (to wit, a prepon-
derance of "difficult" functions) may be expected for the
relation of computation time to computation space. In par-
ticular, the fact that the above result has been well
known for several years but never instantiated, indicates
that a conjecture about $t_M$ compared to $s_M$, based simply on
experience, is a very tenuous conjecture.

A classic example of an exponential task - though not
an exponential problem in the general sense - is the Tower
of Hanoi: a stack of discs is given, each smaller than the
one beneath. It is required to move the stack, in the fol-
lowing manner: only one disc may be lifted at a time;
there must never be more than three piles of discs; no
larger disc may be atop a smaller one. It is easy to show
by induction that to move a stack of $n$ discs takes $2^{n-1}$
disc movements (Basis: 1 disc, 1 move = 2 -1. Inductive
step: you must re-stack the first $n-1$ discs ($2^{n-1}-1$),
then move the bottom disc (1), then replace the $n-1$ others
($2^{n-1}-1$), for a total of $2^{n-1}+2^{n-1}-1 = 2^{n}-1$ moves). It
seems that some similar notion could be exploited to find
an exponential problem for computers - rather than for
piles of discs - but such a notion has not been found.
The big difference is that the constraints on "a com-
puter" are very loose. When the problem is restricted to
push-down stack automata, or further yet to finite auto-
mata, the relations of time and space become tractable,
although still very difficult.
Tautology Problem

In trying to formulate a general result about relations between computation time and computation space, it makes sense to consider problems which at any rate appear to be difficult (even though one may not have a proof that they are intrinsically difficult). It turns out that a large and interesting class of such problems is closely related to a fundamental problem of logic, the "satisfiability problem for CNF formulae in propositional logic". I will digress now to discuss this problem, and return later to its relation to other problems.

Propositional logic is a formalism for making statements about finite collections of objects. A statement in propositional logic is built up of variables and connectives. We shall denote variables by $x, x_1, x_2, x_3, \ldots$; their values are either true or false. Some possible connectives are and, or, not, implies, iff (meaning if and only if), xor (short for exclusive or), nor (short for not or), and nand (short for not and). Thus a typical statement in propositional logic is "$x_1$ implies $(x_1$ or (not $x_2))$". The meaning of the statement is that whenever $x_1$ is true, either $x_1$ is true or $x_2$ is false. Similarly "$x_1$ or ($x_1$ and $x_2$)" means that either $x_1$ is true or both $x_1$ and $x_2$ are true. Depending on the values assigned to $x_1$ and $x_2$, these statements will take on the values true or false. For example, the statement "not $x$" has value true if $x$ is false, and false if $x$ is true. The statement
"x₁ implies x₃" is false only when x₁ is true and x₃ is false. "x₁ or x₃" is false only when both x₁ and x₃ are false. Thus "x₁ implies (x₁ or (not x₂))" always has value true. A statement whose value is always true is called a "tautology" or "true". A statement such as "x₁ or (x₁ and x₂)" whose value is sometimes true (e.g. x₁=x₂=true) and sometimes false (e.g. x₁=false, x₂=true) is called "satisfiable". A statement such as "x₁ and (not x₁)", which never has value true, is called "unsatisfiable". It is clear that A is a tautology iff \( \bar{A} \) is unsatisfiable and vice versa.

It is of note that we need not use all the connectives to express every possible statement. For example "x₁ iff x₂" can be expressed by "(x₁ and x₂) or ((not x₁) and (not x₂))". Similarly "x₁ implies x₂", is expressed by "(not x₁) or x₂"; and "x₁ xor x₂", by "(x₁ or x₂) and (not (x₁ and x₂))". The connectives that we will use are and, or, and not.

Instead of using true and false we can use 1 and 0 respectively, as the two elements of the ring with 1+1 = 0. Then "+" corresponds to "exclusive or", "." to "and", and "x+1" to "not x". "x₁ or x₂" is then expressed as "(x₁ xor x₂) xor (x₁ and x₂)", and is written "(x₁+x₂)+(x₁•x₂)". The statement "x₁ implies (x₁ or (not x₂))" then becomes "((x₁+1)+(x₁+x₂+1+x₁•(x₂+1))+(x₁+1)•(x₁+(x₂+1)+x₁•(x₂+1)))".

This is rather clumsy, but it is amenable to straightforward simplification: using associativity, distributi-
vity, and commutativity, we have
\[ x_1 + 1 + (x_1 + (x_2 + 1)) + x_1 \cdot (x_2 + 1) + x_1 + (x_2 + 1) + x_1 \cdot (x_2 + 1) \]
\[ + (x_1 + (x_2 + 1)) + x_1 \cdot (x_2 + 1) \];

since the third and fifth terms of the sum are equal, their sum is 0, so
\[ = x_1 + 1 + x_1 \cdot (x_1 + (x_2 + 1)) \]
\[ = x_1 + 1 + x_1^2 + x_1 \cdot x_2 + x_1 \cdot x_2 + x_1^2 \]
\[ = x_1 + 1 + x_1 + x_1 \cdot x_2 + x_1 \cdot x_2 \cdot x_1 \]
\[ = (x_1 + x_1) + (x_1 + x_1) + (x_1 \cdot x_2 + x_1 \cdot x_2) + 1 \]
\[ = 0 + 0 + 0 + 1 \]
\[ = 1 \]

which shows that it is a tautology (always has value 1, corresponding to true). In fact, this is a perfectly general way to prove that an expression is a tautology: convert it to a sum of products of variables, and it is a tautology iff it can be reduced to 1. Similarly, if, as a product of sums, it reduces to 0, it is unsatisfiable. Why then is there any research done on recognizing tautologies and unsatisfiability? It is because the above process can be inordinately lengthy. To illustrate this, we re-introduce the connectives and, or, and not, with a different notation (basically the notation of Boolean algebra). "and" is denoted "\&" (called \texttt{inf} in Boolean algebra), "or" is denoted "\texttt{v}" (called \texttt{sup}), "not" is denoted "\texttt{x}" (called complement of \texttt{x}); and true and false are 1 and 0 respectively. Thus the statement "\( x_1 \) implies \((x_1 \texttt{ or } (\texttt{not } x_2))\)" is represented by the formula "\( \texttt{x} \texttt{v} (x_1 \texttt{ v} \texttt{x}_2) \)". This system has the same rules
as set theory, with & for intersection, v for union, \(-\) for complement, 0 for \(\emptyset\), and 1 for the universe. For example, if \(A, B, \) and \(C\) are any formulae, then
\[A \lor (B \land C) = (A \lor B) \land (A \lor C)\] (distributivity of \(\lor\) over \(\land\))
\[A \land (B \lor C) = (A \land B) \lor (A \land C)\] (distributivity of \(\land\) over \(\lor\))
\[A \lor (B \lor C) = (A \lor B) \lor C\] (so we may write \(A \lor B \lor C\); also \(A \land B \land C\))

(associativity)
\[A \land B = B \land A, \quad A \lor B = B \lor A\] (symmetry)
\[A \land \overline{A} = 0, \quad A \lor \overline{A} = 1,\]
\[0 \land A = 0, \quad 1 \land A = A, \quad 0 \lor A = A, \quad 1 \lor A = 1,\]
\[\overline{\overline{A}} = A\]
\[A \lor \overline{B} = \overline{A} \land B, \quad A \land \overline{B} = \overline{A} \lor B,\] (de Morgan's laws)
et cetera.

In this notation the statement "\(\overline{x_1} \lor (x_1 \lor \overline{x_2})\)" is immediately seen to be a tautology:
\[
\begin{align*}
\overline{x_1} & \lor (x_1 \lor \overline{x_2}) \\
& = (\overline{x_1} \lor x_1) \lor \overline{x_2} \\
& = x_1 \lor \overline{x_2} \\
& = 1
\end{align*}
\]
However, there are still formulae which are not amenable to this kind of analysis. For example,
\[((x_1 \lor x_2 \lor x_3) \land (x_4 \lor x_5 \lor x_6)) \lor (x_7 \land x_8 \land x_9)\]
First, to see if it is a tautology, we expand to \(\land\) of \(\lor\)'s:
using \((A \land V) \land C = (A \lor C) \land (B \lor C)\),
\[
\begin{align*}
& = (x_1 \lor x_2 \lor x_3 \lor (x_7 \land x_8 \land x_9)) \land (x_4 \lor x_5 \lor x_6 \lor (x_7 \land x_8 \land x_9)) \\
\text{using } A \lor (B \land C) = (A \lor B) \land (A \lor C) \text{ twice,}
\end{align*}
\]
\[(x_1 \lor x_7 \lor x_8 \lor x_9) \land (x_2 \lor x_7 \lor x_8 \lor x_9) \land (x_3 \lor x_7 \lor x_8 \lor x_9) \land (x_4 \lor x_7 \lor x_8 \lor x_9) \land (x_5 \lor x_7 \lor x_8 \lor x_9) \land (x_6 \lor x_7 \lor x_8 \lor x_9),\]
so it is not a tautology.

Expanding to \(v\) of \(\&\)'s, repeatedly using distributivity of \(\&\) over \(v\), we get

\[(x_1 \land x_4) \lor (x_1 \land x_5) \lor (x_1 \land x_6) \lor (x_2 \land x_4) \lor (x_2 \land x_5) \lor (x_2 \land x_6) \lor (x_3 \land x_4) \lor (x_3 \land x_5) \lor (x_3 \land x_6) \lor (x_7 \land x_8 \land x_9),\]
so it is satisfiable. This kind of example can be extended to formulae of length \(L\) in \(V\) variables which on expansion to a standard form will have length about \(2^{L/V}\).

There are two standard forms, called DNF and CNF (disjunctive and conjunctive normal forms). A formula is in DNF if it is a \(v\) of \(\&\)'s of (variables or complements of variables), i.e. the form in which an unsatisfiable formula reduces to 0. CNF means \(\&\) of \(v\)'s of (\(x\)'s or \(\bar{x}\)'s); a tautology reduces to 1 in CNF. By application of deMorgan's laws (\(\bar{A \land B} = \bar{A} \lor \bar{B}\), \(\bar{A \lor B} = \bar{A} \land \bar{B}\)), we see that if \(A\) is in CNF, \(\bar{A}\) is in DNF and vice versa.

Because \(\bar{0} = 1\), \(A\) is a tautology iff \(\bar{A}\) is unsatisfiable. Thus tautology recognition is essentially the same problem as satisfiability recognition. Another way to see this is that by evaluating \(A\) for one choice of true and false variables, we eliminate one of the possibilities: if \(A\) evaluates to true, it must be satisfiable, and if \(A\) evaluates to false, it must not be a tautology. Indeed, using the notion of Boolean duality (which is discussed
further in Chapter 2) we can state that for any formula $G$, its dual $G^*$ (which is formed from $G$ by interchanging certain connectives) is satisfiable iff $G$ is a tautology. Thus, we can speak interchangeably of "the tautology problem" or "the satisfiability problem".

The $\&$, $\lor$, $\neg$ formulation is of practical use in the design of computer components. Sometimes $\neg$ is used for $\&$, and $\lor$ for $\lor$ (to be distinguished from $+$ in the ring mentioned earlier). A problem of interest to the engineer is to discover the shortest formula equivalent to a given formula, as this corresponds to the simplest logic circuit to realize the function represented by that formula. This problem is a more general form of tautology/unsatisfiability recognition (because the simplest equivalent formula for these latter is simply "1" or "0"); some of the same techniques are applicable.

Polynomial Reducibility

A recent paper by Stephen Cook of the University of Toronto (ACM symposium, May, 1971) has given a renewed impetus to the study of the tautology problem. This paper provides a framework within which to search for relations between computation time and space, and shows that the relation between time and space for many practical combinatorial problems is essentially the same as the time-space relation for a special case of the tautology problem. Before further discussing the ramifications of Cook's result, I would like to carefully define the problems and principles involved and outline a proof of the result.
We will be discussing time and space requirements of computations; some special notation is helpful. If $A$ is a string of characters, then $\lg(A)$ (short for length of $A$) is the number of characters in $A$ (thus if $A$ is a word, $\lg(A)$ is the number of letters in $A$; if $A$ is a number in base $n$, then $\lg(A)$ is approximately $\log_n(A)$). If $F$ is an algorithm (that is, a sequence of steps, such as computer program or a linearly bounded automaton, which is well-defined and always stops after a finite number of finite calculations) then $F$ defines a function, which we also denote $F$, which given the input $x$ yields the output $F(x)$. Depending on the particular kinds of instructions constituting the algorithm, there will be a way to count calculations and measure the total time (or number of basic calculations) involved in obtaining $F(x)$ from $x$. This time we call $t(x)$; to indicate what algorithm takes that long, we sometimes write $t_F(x)$. Similarly we can measure the maximum space (blackboards, reams of paper, Turing tape squares, magnetic tape, etc.) ever in use at any one time in any intermediate step during the calculation; this we call $s(x)$ or $s_F(x)$. For our purposes the basic unit of space is the space occupied by one symbol (including blanks, if used). The basic unit of time is the time taken to (a) read a symbol or (b) write a symbol (including replacing one symbol by another, or erasing an unwanted symbol), or (c) compute and write down a third symbol on the basis of two other symbols.
For example, consider the usual grade-school algorithm (call it \( M \)) for multiplication (so if \( x = (y, z) \), \( M(x) = y \cdot z \)). We know from experience that \( M \) takes about \( (\lg(x) \cdot \lg(y)) \) space to compute \( x \cdot y \), so \( s_M(x) = (\lg(x))^2 \); more important, \( t_M(x) = (\lg(x))^2 \) also. Thus we say that \( M \) is a "polynomial" algorithm, meaning that \( t_M(x) \leq p(\lg(x)) \) for some polynomial \( p \); in this case \( p(n) = 4n^2 \). If it is possible to find a polynomial algorithm to solve a problem, the problem itself is called polynomial.

In contrast to obviously polynomial problems such as multiplication are the "combinatorial" problems. A problem is combinatorial if it is defined in terms of the totality of different combinations of characteristics of a situation. Some typical combinatorial problems: determining whether \( x \) is a prime number (in which you consider \( m \cdot n \) for every combination of \( m, n \) less than \( x \)); finding the shortest path through a directed graph (in which you consider every path, the number of possibilities being the ways you can combine successions of nodes for a valid path); finding the simplest propositional-logic formula equivalent to a given one (in which you consider every shorter formula, the number of formulae being the number of ways to combine variables and connectives for a meaningful formula); determining whether a formula of propositional logic is satisfiable/unsatisfiable/a tautology (in which you consider every combination of values true or false for variables). An algorithm which
follows the definition will of course take little space (assuming you erase your wrong guesses) but a lot of time; it is of interest to develop algorithms which take more than the minimum amount of space but far less time. It is of note in this context that the "minimum equivalent formula" problem is doubly combinatorial, in that each time you guess a shorter formula G to be equivalent to the original F, you have to test whether "F iff G" is a tautology.

Cook's contribution is to define a notion which seeks to characterize "combinatorial" in a quantitative way. This notion is called polynomial reducibility, abbreviated \( p \)-reducibility. A function \( f \) is \( p \)-reducible to a function \( g \) if there is an algorithm \( F \), within which \( g \) is allowed as a basic computational step (i.e. \( g(y) \) can be computed within \( F \) in time \( \lg(y) + \lg(g(y)) \), and a number \( n \) depending only on \( F \) and \( g \), such that for all \( x, F(x) = f(x) \) and \( t_F(x) \leq (\lg(x))^n \) (in automata theory language, \( F \) is an "oracle machine", with oracle for \( g \), which runs in polynomial time).

Before stating the basic theorem about \( p \)-reducibility, I need to state two definitions. (1) Restating the definition of CNF given earlier, a formula in propositional logic is in CNF if it is formed as follows: choose any distinct variables, negate any or none of them, and connect them with \( \lor \). Call this a clause. Connect any number of clauses with \( \land \); this is a CNF formula. The \text{CNF satisfiability problem} is, given a
formula in CNF, to determine whether it is satisfiable.  

(2) An algorithm is non-deterministic if there is some guesswork involved in following the instructions. For example, a non-deterministic algorithm which accepts precisely the non-prime numbers:

[1] Read x
[2] Let n be some number between 2 and the square root of x.
[4] If y is an integer, then accept x as non-prime.

The algorithm is said to accept x if some choice of n would cause x to be accepted. "A set S is accepted by an algorithm" means "x∈S iff x is accepted by the algorithm". It is not hard to show that any non-deterministic algorithm can be converted with at most polynomial increase in time to one in which all guesses are of the form "let the ith bit of variable v be either 0 or 1". It is also easy to show that any non-deterministic algorithm can be converted to a deterministic one, but with a possibly exponential increase in time (for example, if x is a prime number in binary, then the deterministic form of the algorithm given above would require about $2^{(1/2)\log(x)}$ different values for n). Non-deterministic algorithms are precisely intended to deal with combinatorial problems.

Cook's p-reducibility theorem: Any set acceptable, or whose complement is acceptable, in polynomial time by a
non-deterministic algorithm is \( p \)-reducible to the CNF satisfiability problem [provided an explicit polynomial time-bound on the algorithm is known].

This result is somewhat surprising, because it provides a way to convert satisfiability of an arbitrary formula to satisfiability of a CNF formula in polynomial time, while as indicated earlier, direct expansion of an arbitrary formula to CNF can require exponential space and time.

The proof of the theorem is direct: the action of the algorithm is described by a CNF formula; the formula is satisfiable iff some sequence of guesses causes the algorithm to accept the input. Thus the set accepted by the algorithm is precisely the set for which the formula is satisfiable, and the complement of that set is precisely the set for which the formula is unsatisfiable. Since the description of "algorithm" is informal, the proof of the theorem will be informal. For a formal proof with a few minor flaws, see Cook's paper (ACM conference, May, 1971), or for a corrected (I hope) version, see the appendix to this thesis.

There are three essential ideas:

(1) The values for variables are bounded and can be described by a CNF formula whose length is polynomial in the length of the input plus the (fixed) size of the algorithm.

(2) The progression from one step of the algorithm to
the next can be described by a CNF formula: if \( A \) is of
the form \( x_1 \& x_2 \& x_3 \& \ldots \& x_k \) (as in the description of the
variables at one step) and \( B \) is \( x_{k+1} v x_{k+2} v \ldots v x_n \) (as
in the description of the possible values of the variables
at the next step) then \( A \) implies \( B \) is in the form of a
clause of a CNF formula \( (A \text{ implies } B = \bar{A} v B =
\bar{x}_1 v \bar{x}_2 v \ldots v \bar{x}_k v x_{k+1} v \ldots v x_n) \).

(3) If \( A \) and \( B \) are CNF formulae, then so is \( A \& B \).

Note that because of our notation for algorithms
and our conventions for units of time and space, each
instruction takes many time units to execute. Hence
some device is needed to indicate when an instruction has
been completely executed. This will be reflected in the
"more formal" description following the preliminary infor-
mal one. For simplicity all variables will be in binary
form: \( v[i] \) is the bit in the \( 2^{i-1} \) place, so \( v \) represents
\( \Sigma v[i] \cdot 2^{i-1} \).

\( V(v,i,t) \) means (i.e. has value true iff) "the \( i \)th bit of
variable \( v \) is 1 at time \( t \)."

\( S(n,t) \) means "the \( n \)th instruction is being executed at
time \( t \)."

The operation of the algorithm on input \( x \) (also in binary
form; characters coded as numbers if it is a character
algorithm) is described as follows:

In words:

Let \( v_1 \) be the input, so
At time 1, \( v_1 = s \).
At each time, each variable has a unique value.
At each time, a unique instruction is being executed.
If instruction $s$ is being executed at time $t$,
   but is not yet finished, then instruction $s$ is being
   executed at time $t+1$.
If instruction $s$ is finished being executed a time $t$, and
   variable $v$ has value $z$ at time $t$, then at time $t+1$
   instruction $s'$, as dictated by instruction $s$ under
   those conditions, is being executed.
If instruction $s$ is finished being executed at time $t$, and
   variable $v$ has value $z$ at time $t$, then at time $t+1$
   variable $v$ has one of the possible values accorded to
   it by instruction $s$.
If variable $v$ is not involved in the instruction being
   executed at time $t$, then at time $t+1$ $v$ has the same
   value as at time $t$.
At some time an instruction is executed which says "the
   input is accepted".

Note that in the formal description we have to speak
about the individual bits of the variables, because
whereas each bit can have only two values, the variable as
a whole can have $2^{(\text{number of bits})}$ values, the description
of which would require an exponential number of clauses
in the formula.

More formally:
Let each variable have $I$ bits, let the algorithm have $N$
instructions, let $D(t)$ mean that the current instruction
has been executed (and a flag set to so indicate), let T be the time bound which we have calculated on the basis of the input x, and let A(t) mean x is accepted at time t. Since not all the time T will necessarily be used, we make the convention that the Stop instruction is executed repeatedly once encountered.

The description is then:
V(1,i,l) = true if x[i] = 1, for each bit of x.
V(1,i,l) = false for 1 < i ≤ I.

(There is nothing corresponding to "each variable has
a unique value", because the notation permits only
one value -- 1 (true) or 0 (false) -- for each bit
at each time.)

S(1,t) ∨ S(2,t) ∨ S(3,t) ∨ ... ∨ S(N,t) for 1 ≤ t ≤ T

(A unique instruction at each time)

S(s,t) & D(t) implies S(s,t+1) (Continue executing
statement s if it is not completed.)

S(s,t) & D(t) & V(v,i,t) implies S(s',t+1) for the appro-
priate v, i, s' as dictated by statement s.

S(s,t) & D(t) & V(v,i,t) implies S(s',t+1) for the appropriate
v, i, s'.

S(s,t) & V(v,i,t) implies V(v',i',t+1) for appropriate
v, i, v', i', such that no guess is made. (A
guess is indicated by not specifying whether the
bit should be 0 or 1.)

S(s,t) & V(v,i,t) implies V(v',i',t+1) for appropriate
v,i,v',i', such that no guess is made.

S(s,t) & V(v,i,t) implies V(v',i',t+1) for appropriate
v, i, v', i', such that no guess is made.

S(s,t) & V(v,i,t) implies V(v',i',t+1) for appropriate
v, i, v', i', such that no guess is made.

S(1,t) & S(2,t) & ... & S(m,t) & V(v,i,t) implies V(v,i,t+1)

for all v, i, and all s_1 ... s_m involving the i^{th} bit
of variable $v$.

$\overline{s}(s_1,t) \& \overline{s}(s_2,t) \& \ldots \& \overline{s}(s_m,t) \& \overline{V}(v,i,t)$ implies $\overline{V}(v,i,t+1)$

for all $v$, $i$, and all $s_1 \ldots s_m$ involving the $i^{th}$

bit of variable $v$.

$A(1) \lor A(2) \lor A(3) \lor \ldots \lor A(T)$.

Finally, in CNF we have

& $V(1,i,1)$

i such that

x[i]=1

& & $\overline{V}(1,i,1)$

i such that $x[i]=0$

or $i>\lg(x)$

& & ( $v \overline{s}(s,t)$ )

$t \leq t \leq T$ 1 \leq s \leq N

& & ( $\overline{s}(s,t) \lor d(t) \lor s(s,t+1)$ )

all $s$

$t \leq t \leq T-1$

& & ( $\overline{s}(s,t) \lor d(t) \lor \overline{V}(v,i,t) \lor s(s',t+1)\lor $)

all $s$, $t \leq t \leq T-1$

appropriate $v,i,s'$

& & ( $\overline{s}(s,t) \lor d(t) \lor \overline{V}(v,i,t) \lor s(s',t+1)\lor $)

all $s$, $t \leq t \leq T-1$

appropriate $v,i,s'$

& & ( $\overline{s}(s,t) \lor \overline{V}(v,i,t) \lor \overline{V}(v',i',t+1)\lor $)

all $s$, $t \leq t \leq T-1$

appropriate $v,i,v',i'$

(no guess made)

& & ( $\overline{s}(s,t) \lor \overline{V}(v,i,t) \lor \overline{V}(v',i',t+1)\lor $)

all $s$, $t \leq t \leq T-1$

appropriate $v,i,v',i'$

(no guess made)

& & ( $\overline{s}(s,t) \lor \overline{V}(v,i,t) \lor \overline{V}(v',i',t+1)\lor $)

all $s$, $t \leq t \leq T-1$

appropriate $v,i,v',i'$

(no guess made)
\( \forall s, t \forall v, i, v', i' \forall T \forall \ldots \)
\( \forall v, i, v', i' \forall T \forall \ldots \)
\( \forall v, i, v', i' \forall T \forall \ldots \)

Again, this has been an informal exposition; the purpose is to illustrate the method of proof.

Having introduced the concept of p-reducibility and proved a basic characterization of it, we ask, "What advantage is there to this concept?" I said earlier that it seeks to provide some kind of quantitative meaning for the term "combinatorial". Another meaning which can be given is that any problem which can be done in polynomial time (that is, is p-reducible to the identity function) is practicable. How well as these goals met? The first one is met quite well. Algorithms that operate using the definitions of combinatorial problems will use up time on the order of factorials of the space needed to pose and solve the problem. By Stirling's approximation (\( n! \approx n^n e^{-n} \sqrt{2\pi n} \)), this will be exponential, hence greater than any polynomial. Thus combinatorial problems for which no "clever" algorithms have been found are those which are not known to be p-reducible to the identity.
function.

How about practicability? By the definition of p-reducibility, we don't distinguish between $x$, $x^{1/5}$, $x^2$, and $x^{107}$. Certainly if an algorithm takes $(\lg(x))^{107}$ time to process $x$, it is not practical. This is an oversimplification inherent in the concept of p-reducibility. However, every known algorithm either requires exponential time, or appears to take no more (even using very crude bounds) than about $(\lg(x))^6$ time. As far as I am aware there is no algorithm that is the fastest known for its problem and whose time complexity is known to lie below the exponential but above $(\lg(x))^2$. Thus the oversimplification does not seem to be of importance at present.

Another reason for considering p-reducibility is that it does pose some tractable problems in what has been an almost intractable area. If anyone can show that any of the combinatorial problems currently considered -- including many graph-theoretic problems -- is definitely not p-reducible to the identity function; or if it is shown that the CNF satisfiability problem can be done in polynomial time; then it will be time to renew a vigorous consideration of lower time-bounds on the order of $(\lg(x))^2$, $(\lg(x))^3$, etc.

What are some problems that the study of p-reducibility leads us to consider? Many combinatorial problems
are clearly solvable in polynomial time by a non-deterministic Turing machine, and hence are p-reducible to the CNF satisfiability problem: primes recognition; shortest path through a graph; general satisfiability and tautology; graph isomorphism. Another combinatorial problem which is known to be p-reducible to the satisfiability problem is the traveling salesman problem (the shortest path through a graph passing through all nodes). A problem which is not known to be p-reducible to the satisfiability problem is the minimal Boolean form problem. Of somewhat more interest, perhaps, than finding what problems are p-reducible to the satisfiability problem, is learning to what other problems is satisfiability p-reducible; that is, what are other general problems in the sense of p-reducibility. Cook showed in his May, 1971, ACM paper that sub-graph isomorphism is a general problem; and Karp and Tarjan recently showed that the traveling salesman problem is general. Minimal Boolean forms is obviously general. It is not known whether primes and graph isomorphism are general. Another general problem is to recognize "$D_3$-tautologies", that is, DNF tautologies with at most 3 literals in each clause (proved in Cook's paper; I give a brief proof in Chapter 5). It is unlikely that $D_2$ is general, since the Davis-Putnam algorithm is obviously polynomial on $D_2$. 
Tautology Algorithms

Let us now pick one general problem, the DNF tautology problem, and see what is known about solving it. Aside from exhaustive testing, there are two important methods: tableau methods and Quine methods. The Davis-Putnam algorithm is an example of the latter, and will be thoroughly explored in Chapters 1 and 2. The tableau methods have long been popular with logicians; for a detailed treatment see, for example, Smullyan, First-Order Logic; I will give here a brief description.

Tableau methods are not specific to DNF, but it is particularly simple to describe at least one tableau method, which Smullyan calls "analytic tableau", as it applies to DNF formulae. Usually a diagram is drawn, but I shall first state the algorithm in terms of the usual formal-logic notation.

Given G

Pick a clause B and a literal x in that clause.

Let G = B\lor C

B = x\land A

Write G as (x\land A)\lor C

Rewrite G as (x\lor C)\land (A\lor C)

Now G is falsifiable (i.e. not a tautology) iff either x\lor C or A\lor C is falsifiable.

Re-apply this process to the DNF formulae x\lor C and A\lor C.
From time to time C will be a single clause.
If C does not contain \overline{x}, than x\lor C is falsifiable,
and hence G is falsifiable, since G = (xvC)&H, for some formula H.

If C does contain $\overline{x}$, then xvC = true and is deleted.
Eventually either G reduces to 1 or G is seen to be falsifiable.

The diagram associated with this process is a tree. Initially G is the root. Scratch some clause from G and put one literal of that clause as one successor to G, and the rest of the clause as the other successor. Repeat this, adjoining the new successors to each node produced at the last step. Eventually either some branch will appear such that for no pair of nodes does x occur in a clause at one node and $\overline{x}$ at a succeeding node; or else every branch will contain one node labeled x and another labeled $\overline{x}$ (different x's for different branches). The latter case is a tautology.

It is straightforward to show that the tableau method is exponential. One example is the tautology consisting of disjunction of every clause of length n in n variables; this is a formula of length $n \cdot 2^n$, and the tableau has $2^n$ nodes, so

$$S_M(x) = 2 \frac{\log(x)}{n}$$
Another example, this one with at most three literals per clause, is the formula (for any even n)

\[(x_1 \& x_2) \lor \lor (\overline{x}_i \& x_{i+2} \& x_{i+3}) \lor \lor (\overline{x}_i \& x_{i+1} \& x_{i+2}) \]

\[1 \leq i \leq n-3 \quad \text{i odd} \]

\[i=0,1 \]

\[\lor \lor (\overline{x}_{n-i} \& \overline{x}_{n-j}) \]

\[i=2,3 \]

\[j=0,1 \]

For example with \(n=8\), this is

\[(x_1 \& x_2) \lor (\overline{x}_1 \& x_3 \& x_4) \lor (\overline{x}_2 \& x_3 \& x_4) \lor (\overline{x}_3 \& x_5 \& x_6) \lor (\overline{x}_4 \& x_5 \& x_6) \]

\[\lor (\overline{x}_5 \& x_7 \& x_8) \lor (\overline{x}_6 \& x_7 \& x_8) \lor (\overline{x}_5 \& \overline{x}_7) \lor (\overline{x}_5 \& \overline{x}_8) \lor (\overline{x}_6 \& \overline{x}_7) \lor (\overline{x}_6 \& \overline{x}_8) \].

A tableau for this (using the indices to stand for the variables) is (after all clauses are scratched off)
Although I do not have a rigorous proof, this appears to require a minimum of $2^{(n/2-1)}$ nodes in the tableau method, no matter what root you choose.

By contrast, the Davis-Putnam algorithm is linear on both of these types of example (see computer results in appendix).

Of course, for a non-tautology, all you need is one branch that indicates a falsifying choice of variables; if one knew that for any non-tautology such a branch would show up before too long, then one would never have to complete the tableau: one could say, "this must be a tautology, because no falsifying branch has been found yet." Thus for the tableau method, although it is hard to say much about the analysis of non-tautologies, it is easy to prove that tautologies can generate exponentially large tableaux; I can see no way to circumvent this.

It would be very interesting to discover some similar fact about the Davis-Putnam algorithm. Toward this end, I will give here a brief geometrical description of Quine-type methods, and the Davis-Putnam algorithm in particular. The reader who is not familiar with the Davis-Putnam algorithm should be able to verify the accuracy of this description after reading chapter 1.

We consider any formula in n variables as denoting a subset of the vertices of the n-dimensional hypercube. The set of all those vertices is often denoted $\{0,1\}^n$. 
or for short, $2^n$. There are indeed $2^n$ vertices in $2^n$, so the notation is not arithmetically misleading. An element of $2^n$ is denoted by an $n$-tuple of 0's and 1's; and the geometric interpretation of each $n$-tuple is as though the $n$-tuple were a vector in $n$-dimensional space. Thus in 3-space, the point $(0,1,0)$ corresponds to $x=0$, $y=1$, $z=0$ (the point 1 on the $y$-axis); $(1,1,1)$ is the corner farthest from the origin.

How do we associate a subset of $2^n$ with a formula $G$ in $n$ variables? We associate the $i^{th}$ variable with the $i^{th}$ coordinate, and the values false and true with 0 and 1 respectively. Then $G$ induces a map from $2^n$ into \{true, false\}, with $G(\vec{x}) = \text{true}$ iff making $x_i$ true for the $i^{th}$ coordinate of $\vec{x}$ equal to 1, and false otherwise, and then evaluating $G$ with these choices, produces a value of true. The subset of $2^n$ denoted by $G$ is those vertices which are mapped to true.

Thus we see that "tautology" and "unsatisfiable" correspond to the set-theoretic notions of "universe" and "empty set" respectively. It will be most convenient to consider the Davis-Putnam algorithm as determining satisfiability of a CNF formula (rather than tautology for a DNF formula). Thus we wish to begin by seeing how a CNF formula "looks" geometrically.

It is not hard to verify that "and" and "or" correspond to "intersection" and "union" respectively. Thus a
CNF formula represents an intersection of unions of subsets of $2^n$, these last subsets being those represented by single literals. What subset of $2^n$ does a single literal represent? Well, consider the literal $x_1$. This is a formula $G$ in $n$ variables, and thus induces a map from $2^n$ into {true, false}. What map? Well, if $x_1$ is true, then $G=x_1=true$; otherwise $G$ is false. Thus $x_1$ denotes all vertices whose first coordinates are 1: namely, the vertices of the $(n-1)$-dimensional hyperplane parallel to the $x_2-x_3-x_4..._n$ hyperplane. For example, if $n=3$, then $x_1$ is the side of the box opposite to the $y-z$ plane.

Thus each clause represents a union of differently-oriented copies of $2^{n-1}$, and a CNF formula represents an intersection of a bunch of these things. For instance, the unsatisfiable formula $x_1 \land \bar{x}_1 \land (x_2 \lor x_3)$ represents the intersection of whatever $(x_2 \lor x_3)$ represents with (the intersection of (all vertices not in the $y-z$ plane) with (all vertices in the $y-z$ plane)); this is (some set) intersected with (the empty set), which is the empty set. Incidentally, $x_2 \lor x_3$ represents the top and right-hand side of the box, if $z$ is the upward coordinate and $x$ points toward the viewer.

What does the Davis-Putnam algorithm do? Suppose $x_1$ is to be eliminated from $G$. All clauses of $G$ not involving $x_1$ are kept intact; these represent subsets that are the same in the $x_1=0$ hyperplane as in the $x_1=1$ hyper-
plane. If \( A=\overline{x}_1 v B \) and \( C=\overline{x}_1 v D \) are clauses of \( G \), then after elimination we have \( BvD \) as a clause of the new formula. A geometric interpretation is, that the part of \( A \) in the \( x_1=0 \) hyperplane looks like \( B \), and the part of \( C \) in the \( x_1=0 \) hyperplane is the whole \( x_1=0 \) hyperplane. Thus \( A&C \) (which is part of the formula \( G \)), viewed in the \( x_1=0 \) hyperplane, looks just like \( B \). Similarly the part of \( A&C \) in the \( x_1=1 \) hyperplane looks like \( D \). Thus \( BvD \) looks like (the part of \( A&C \) in the \( x_1=0 \) hyperplane) union (the part of \( A&C \) in the \( x_1=1 \) hyperplane). The usual geometric name for this is the "projection" of \( A&C \) onto the \( x_1=0 \) hyperplane. For example, with \( n=3 \), if \( G = (\overline{x}_1 v x_3) & (x_2 v x_3) & (\overline{x}_1 v x_2) & (x_1 v \overline{x}_2 v x_3) \), then the vertices represented by \( G \) are \((0,0,1), (0,1,0), \) and \((1,1,1) \). Eliminating \( x_1 \) we get \((x_2 v x_3) \) as the only clause, representing the vertices \((0,1), (1,0), \) and \((1,1) \). It is easy to see from a picture of the cube and square that this is indeed the projection:
Thus the effect of using Davis-Putnam elimination is to form the projection, parallel to the axis of elimination, of the original set. It is clear that projecting the empty set yields the empty set; projecting any other set eventually produces a set which is obviously non-empty. Thus unsatisfiability (i.e. emptiness) is preserved, and the procedure works. The geometric interpretation of the Davis-Putnam algorithm for DNF formulae is complement (projection (complement (G))).

For a third tautology algorithm, see Cook's Method I in the appendix.
Complexity of the Davis-Putnam Algorithm

We come now to the specific task of analyzing the time complexity of the Davis-Putnam algorithm. How are we to proceed? Perhaps the best approach would be to show somehow that the different tautology algorithms can be mapped onto each other by some polynomial conversion; if this were so it would certainly prove that the problem is exponential. I know of no results at all along this line.

Another possible approach is to find a statistical time-bound; that is, what is the average time of analysis over all formulae? This has been done for algorithms in the area of operations research, and might prove fruitful here. I attempted such an analysis, but was not well rewarded; this is probably more a reflection on my ignorance of statistics than a characteristic of the problem.

If we limit ourselves to one particular algorithm, it seems reasonable to find the worst case and see how long the algorithm takes for formulae of that variety; the catch is, how to find the worst case? For the tableau method it was not too difficult; it seems that it should be feasible for the Davis-Putnam method.

One approach to finding a worst case is through the geometry of the n-hypercube. This has the advantage of presenting one picture instead of a thousand clauses; however, it also presents n dimensions instead of one, and it is hard to develop much intuition in more than 4 or 5 dimensions; moreover it presents $2^n$ vertices, so it becomes
very hard even to draw the hypercube. Nonetheless, the notion of projection is fairly comprehensible; we just imagine the n-cube as two parallel blobs representing (n-1)-cubes, and try to picture what could make the description of the projection more complicated than the original formula.

This matter of "descriptions" is basic to the tautology problem, and indeed to all of computational complexity. What we are dealing with is not really the projection, for example, of a set in $2^n$; these sets have huge cardinality, yet our descriptions -- at least, those that we are interested in -- are of quite manageable size; rather, when we use the Davis-Putnam or other Quine-type algorithms, we are dealing with a transfer of information about a subset of the hypercube. Just as in the Tower of Hanoi problem we have constraints on the configurations of of the discs, so in the tautology problem do we have constraints on the appearance of the formulae. If we understood these constraints, we would probably have the answer to our problem.

Specifically, I would conjecture that any algorithm which accepts as input an arbitrary DNF, and produces output "tautology" or "not tautology", must "carry along" at least as much information as the Davis-Putnam algorithm; for example, it seems unlikely that some kind of "limited consensus" operation, in which clauses of more than a certain length are discarded, will be a general tautology
algorithm; I would expect it to be unable to distinguish, say, "long skinny" sets from the entire cube. I do not mean this to be any sort of a conjecture about the complexity of the tautology problem; I merely claim that the constraints on the problem must arise from within the problem, not simply be imposed upon an algorithm.

In this same vein, I think it is appropriate to find out just how complex the Quine algorithm is: this means precisely, in geometric terms: if a subset of $2^n$ can be described by $K$ boxes, none containing any other, how many more such boxes can that set contain? This seems like a tractable problem; it is certainly an interesting problem; there are results which bear obliquely upon it; and yet, to my knowledge, it is an unsolved question. I spent several weeks investigating it and came away with a better understanding of the 6-cube, but little else.

The class of "descriptive" formulae is an interesting one, even though it is not preserved by projection, because we can generate complicated formulae and know ahead of time whether they are tautologies or not. For instance it takes over 100 clauses in 30 variables to express the formula "x is a divisor of 15" in CNF; we know that this is satisfiable, we know that "x is a divisor of 13" is not. We can compare the actual operation of the Davis-Putnam algorithm on examples like these, and hope to draw some general conclusions. My computer program is in APL, and requires iteration in the subsumption rule, so it runs
relatively slowly. I think that a good compiled or even microprogrammed Davis-Putnam algorithm would provide a useful tool for investigating the complexity of the tautology problem.

There is also merit in finding the most special general cases; for example Cook's 3-literal-per-clause formulae, or the set of formulae describing Turing machines. However, application of the Davis-Putnam procedure does not preserve the special characteristics of either of these sets; indeed in $D_3$, if we choose the variable which minimizes (number of positive occurrences) x (number of negative occurrences), in analyzing a $D_3$ formula obtained from an arbitrary formula, we will usually wind up with the original formula, after the "dummy" variables have been eliminated. One would certainly like to find classes of formulae which are preserved by projection; one of these is the symmetrizable formulae. These have the further nice property that order of elimination is immaterial. As Theorem 1 shows, they have the further nice property that the Davis-Putnam algorithm (with subsumption rule) analyzes them in polynomial time, and hence we lose them as a potential source of "difficult" formulae. Let us proceed now to the proof of the theorem.

Complexity of the Davis-Putnam Algorithm on Symmetrizable Formulae

The rest of this paper is divided into five chapters and an appendix. The first two chapters deal with the
Davis-Putnam algorithm in general; the first is motivational and the second, notational. The third and fourth chapters contain a proof of the polynomial bound, and the fifth, extensions and corollaries. A glossary is given at the end of Chapter 5. The appendix consists mostly of an APL computer program for the Davis-Putnam algorithm and some sample runs.

The proof (Chapters 3 and 4) is essentially in three stages. First, we show that the Davis-Putnam algorithm preserves symmetry, and characterize "reduced" symmetric formulae in terms of the sets of clauses they can Second, we show exactly what happens to these sets of clauses after an iteration of the Davis-Putnam algorithm; third, we see that this implies at most a polynomial growth in the size of the formula, from the first iteration to any subsequent iteration.

Chapter 1 Let A be a propositional formula in DNF (disjunctive normal form), e.g. \( A = (x_1 \land x_2 \land \bar{x}_3) \lor (x_2 \land x_3) \lor (x_1 \land \bar{x}_2) \). Call A reduced if no clause of A is a subclause of any other clause of A, e.g. \((x_1 \land x_2) \lor (x_1 \land x_2 \land \bar{x}_3)\) is not reduced, but \((x_1 \land x_2) \lor (x_1 \land \bar{x}_2 \land \bar{x}_3)\) is reduced. Let the reduced form of A be formed from A by deleting all but one copy of any repeated clause, and deleting all clauses that contain another clause. Note that A is a tautology iff the reduced form of A is a tautology (Rule(4); Lemma 2) (reduction is also known as the "subsumption rule": any clause which subsumes another is deleted).
Chapter 2  It can be shown that applying one iteration of the (dual) Davis-Putnam algorithm with subsumption rule to any DNF formula \( A \) is equivalent to forming "PROD(\( A \))", as follows: define \( \oplus \) as the following (asymmetric) binary operation on clauses: \( C = A \oplus B \) iff either

1. \( C = A = B \) and neither \( x_1 \) nor \( \overline{x_1} \) occurs in \( A \)
   (or \( B \) or \( C \));

or

2. \( C = A' \& B' \) where \( x_1 \) occurs in \( A \) and \( \overline{x_1} \) occurs in \( B \) and \( X' \) means \( X \) with \( x_1 \) or \( \overline{x_1} \) deleted;

or

3. \( C \) is the empty clause if (1) and (2) do not apply.

(Note that under (2), \( C \) will be the empty clause if for some \( i \neq 1 \), \( x_i \) occurs in \( A \) and \( \overline{x_i} \) in \( B \), or vice versa.) Define PROD(\( A \)) as the reduced form of the DNF formula formed by the disjunction of all \( C \oplus D \) such that \( C, D \) are clauses of \( A \). PROD(\( A \)) is the same formula as would be obtained by applying Davis-Putnam elimination to variable \( x_1 \), then applying the subsumption rule. (If some other variable is to be eliminated, simply switch indices; this will not affect whether the formula is a tautology).

Chapter 3  Call \( A \) symmetrical iff \( A \) is invariant under permutation of variable indices. Note that \( A \) is symmetric iff every permutation of any clause of \( A \) is likewise a clause of \( A \) (Lemma 4). Henceforth \( E \) denotes a reduced symmetric DNF formula (RSDNF). If \( A \) is symmetrical, then any two clauses are either obtained from each other by permutation of variable indices, or else they differ in the respective numbers of positive and negative occurrences
of variables. If a clause contains \( m \) variables positively and \( n \) negatively, then the clause-type of that clause is \((m,n)\). Any symmetrical DNF formula \( A \) is completely specified by (a) the number of distinct variables used in \( A \), and (b) the set of all clause-types in \( A \). For any RSDNF \( E \), if \( p \) and \( q \) are both clause-types of \( E \), then either \( p \odot q \) or \( q \odot p \), where \((a,b) \odot (c,d)\) means \( a > c \) and \( b < d \) (Lemma 7).

Some consideration will show that the clause-types of \( \text{PROD}(E) \) are highly restricted: for any RSDNF \( E \), \((a,b)\) is a clause-type of \( \text{PROD}(E) \) only if \((a,b)\) is a clause-type of \( E \) or there are \( x \) and \( y \) such that \((a+1,x)\) and \((y,b+1)\) are clause-types of \( E \) with \((a+1,x) \odot (y,b+1)\), \( E \) having no clause-type \((m,n)\) "between" the two (in the sense of \( \odot \)) (Lemma 11). This may be strengthened to "if and only if" with the additional provisions (1) if \((a,b)\) is a clause-type of \( E \), then \((a+1,b)\) and \((a,b+1)\) are not; (2) \( a+b \leq (\text{the number of variables in } E)-1 \) [= the number of variables in \( \text{PROD}(E) \)] (Lemma 12).

Chapter 4 This is then used to show that if \((a,b)\) is a clause-type of \( \text{PROD}^n(E) \) (the result of \( n \) iterations of the Davis-Putnam algorithm) then either \((a,b)\), or \((a+1,b)\), or \((x,b+1)\) is a clause-type of \( \text{PROD}^{n-1}(E) \), provided \( n \geq 2 \) (Lemma 14). Applying a simple combinatorial argument to this shows that for \( n \geq 2 \), the number of clauses of any one given clause-type in \( \text{PROD}^n(E) \) is no greater than the maximum number of clauses of any given clause-type in \( \text{PROD}^{n-1}(E) \) (and hence in \( \text{PROD}^1(E) \), which equals \( \text{PROD}(E) \))
(Lemma 14).

Finally, observe that there is at most one more clause-type in $E$ than the number of variables in $E$ (Lemma 13), so that the number of clauses in $\text{PROD}^n(E)$ for $n \geq 1$ is no more than (number of variables in $E$) times (maximum number of clauses of any one clause-type in $\text{PROD}(E)$) (Theorem 1); the second factor is at most (the number of clauses in $\text{PROD}(E)$), which from the definition of $\text{PROD}$ is at most (number of clauses in $E$)$^2$. Thus the maximum length of any formula produced by applying $\text{PROD}$ to an $\text{RSDNF}$ $E$ (and hence the maximum length formula generated by the Davis-Putnam procedure with subsumption, given an RSDNF $E$) is at most $V \cdot |E|^2$, where $V$ is the number of variables in $E$ and $|E|$ is the number of clauses in $E$ (whence $V \cdot |E|$ is the maximum possible length of $E$). Since each iteration of the procedure is polynomially bounded, this yields a polynomial time bound on the entire analysis (Theorem 2).

Conclusion

It had been my hope to "crack" the problem of the complexity of tautology recognition, or at least of the Davis-Putnam algorithm. Failing this, what have I accomplished? Most important, I think, is that I have spared other researchers a fruitless search, among the symmetrizable formulae, for formulae on which the Davis-Putnam algorithm is exponential. Second, I think it is of interest that a general-purpose algorithm such as the Davis-Putnam algorithm should intrinsically recognize
the simplicity of the special class of symmetrizable formulae. This is the kind of "general insight into specific algorithms" that the field of algorithmic efficiency should produce.

Another interesting product of my research is the algebra of clause-types. In view of Cook's $D_3$ result, and other factors, the notion of clause-types should be applicable to non-symmetrizable formulae; for instance if one could show that the Davis-Putnam algorithm does not increase the size of clause-types very much, then the $D_3$ result would show that the Davis-Putnam algorithm is polynomial.

Finally, I hope that the solution of one special case will inspire the solution of other special cases, which may eventually give us the intuition and perspective necessary for a definitive resolution of the tautology problem.
CHAPTER 1

The Davis-Putnam Algorithm: a method for determining satisfiability of propositional formulae in conjunctive normal form.

Definitions:

CNF, conjunctive normal form
literal
clause
disjunction
conjunction
subclause
satisfiable
unsatisfiable
implication, implies, $ightarrow$
Davis-Putnam algorithm
Rules (1), (2), (3), (4)
annihilation, REDUCTION
product rule, PRODUCT, PROD
reduced

Proofs:

Validity of Davis-Putnam algorithm for satisfiability of CNF formulae

Validity of annihilation rule (subsumption rule, reduction)

Product Rule

43
In an article entitled "A Computing Procedure for Quantification Theory" (Journal of the Association for Computing Machinery, Vol. 7 (1960) pp. 201-215), Martin Davis and Hilary Putnam exhibited an algorithm for determining whether or not a formula in the propositional calculus, in conjunctive normal form (CNF) is unsatisfiable, by eliminating one variable after another from certain derived formulae. This algorithm is hereinafter referred to as "the Davis-Putnam algorithm", abbreviated "DP".

The number of iterations required is no larger than the number of variables, but there is no obvious subexponential bound on the time and space taken for each iteration, as a function of the initial space used in a standard propositional representation of the formula.

In a talk entitled "The Complexity of Theorem-Proving Procedures", at the Association for Computing Machinery's Conference on Theory of Computation in May, 1971, Stephen Cook showed that a polynomial bound on this would yield polynomial bounds on some other interesting problems, including the testing of an arbitrary propositional formula for truth, and in general any problem solvable in polynomial time by a nondeterministic Turing machine. In response to an inquiry last summer, Mr. Cook sent me two examples bearing on this question (see appendix). Example 1 shows that if variables are eliminated in an unfavorable order, the space required by the Davis and Putnam algorithm will be
exponential, regardless of any evident modifications that could be made to the procedure. Example 2 is a set of formulae in which, regardless of the order of elimination of variables, the algorithm, if followed literally (as by a computer) would require an exponential space. The former example can be set aside for the moment by agreeing to eliminate the variables in what appears, as one proceeds, to be a favorable order. The latter example is the one I am concerned with here. What I have shown is that following Cook's suggestion in Example 2, namely, at each iteration, to eliminate redundant clauses, yields a polynomial bound on a class of formulae that includes the one he exhibited. (Mr. Simon [5] independently discovered the same class of formulae, but did not suggest a way to simplify their analysis.)

I will first give a brief description of the Davis-Putnam algorithm for satisfiability of CNF formulae. The reader is referred to their JACM article for fuller description and further examples.

An expression is a formula of formal logic (specifically, of the propositional calculus) if it is made up of variables (indexed by the natural numbers 1, 2, 3, ...) and using and, or, not, and possibly other connectives. The variables take on the values true or false. A formula in CNF is of the type \((x_1 \text{ or } \neg x_2) \text{ or } x_3 \text{ or } \ldots\) and \((\text{not } x_1) \text{ or } x_4 \text{ or } \ldots\) and... Specifically, a formula is in CNF
(conjunctive normal form) if it is a "conjunction of disjunctions of literals"; that is, it is built up as follows: a variable or the negation of a variable is called a literal; for example \( x, x_3, \overline{x} \) (which means (not \( x \))). A clause is a string of literals separated by "or", written "v"; "\( x_1 \vee x_2 \)" is the disjunction of \( x_1 \) and \( x_2 \), so this is a disjunction of literals. A CNF formula is a string of (parenthesized) clauses separated by "and", written "&" and called conjunction, whence a conjunction of disjunctions of literals. One of the members of a conjunction may be called a "conjunct", so that clauses in a CNF formula are sometimes called conjuncts.

In formal language notation, we have

\[
\begin{align*}
\text{<subscript>} & \quad \text{+ positive integer} \\
\text{<variable>} & \quad x|x_{<subscript>} \\
\text{<literal>} & \quad <\text{variable}> \mid <\text{variable}> \\
\text{<clause>} & \quad <\text{literal}> \mid <\text{clause}> \vee <\text{literal}> \\
\text{<CNF>} & \quad (<\text{clause}>) \mid <\text{CNF}> \& <\text{CNF}>
\end{align*}
\]

Note: I will use the associativity and symmetry of \& and v without further comment (i.e. \((A \& B) \& C = A \& B \& C = A \& (B \& C) = B \& (A \& C)\), etc.). Also, note that if \( x \) and \( \overline{x} \) both occur in a clause \( A \), then \( A \) is always true (\( G \vee \overline{G} \vee H = \text{true} \vee H = \text{true} \) for any \( G \) and \( H \)), so unless \( A \) is the only clause in the formula, \( A \) may be deleted (\( G \& \text{true} = G \) for any non-empty \( G \)).
Example of a CNF formula, with commentary

\[ A = (x_1 \lor \overline{x}_3) \land (x_2 \lor x_3 \lor \overline{x}_4) \land (\overline{x}_1 \lor x_1 \lor x_2 \lor \overline{x}_5) \]

\[ \land (\overline{x}_2 \lor x_4) \land (\overline{x}_1 \lor x_2 \lor \overline{x}_3 \lor x_4) \]

\[ = (x_1 \lor \overline{x}_3) \land (x_2 \lor x_3 \lor \overline{x}_4) \land (\overline{x}_2 \lor x_4) \land (\overline{x}_1 \lor x_2 \lor \overline{x}_3 \lor x_4), \] since the third clause contained both \( x_1 \) and \( \overline{x}_1 \). Note that the disjunction of two clauses is a clause; for example if \( B = (\overline{x}_1 \lor x_3) \) and \( C = (x_2 \lor x_4) \) then \( B \lor C \) is the last clause of \( A \) above. \( B \) and \( C \) are called subclauses of the clause \( (B \lor C) \). Likewise the conjunction of two CNF formulae is a CNF formula; for example,

If \( D = (x_1 \lor x_3) \land (x_2 \lor x_3 \lor \overline{x}_4) \) and

\[ E = (\overline{x}_2 \lor x_4) \land (B \lor C), \] then \( D \land E = A \).

The Davis-Putnam algorithm may be viewed as a variation of Quine's Algorithm (W.V. Quine, "A Way to Simplify Truth Functions", American Mathematical Monthly, v. 62, 1955, pp. 627-631), also known as the method of consensus or iterative consensus, for finding the prime implicants of a formula. A more recent name for similar techniques is "resolution". All these methods are based on the following consideration: consider the formulae \( A \) and \( B \), neither containing the variable \( x \). Consider the formula \( (x \lor A) \land (\overline{x} \lor B) \). We claim that this formula is satisfiable iff \( A \lor B \) is satisfiable. Recall that "a formula \( G \) is satisfiable" means that we can pick values true or false for the variables in \( G \) such that evaluating the expression (by the truth table definitions of the connectives) results in a
value of true. For example, \( x_1 \land (\overline{x}_1 \lor x_2) \) is satisfied by
\( x_1 = \text{true}, x_2 = \text{true} \) (true \& (false \lor \text{true}) = true \& true =
true), whereas \((x_1 \lor x_2) \land (x_1 \lor \overline{x}_2) \land (\overline{x}_1 \lor x_2) \land (\overline{x}_1 \lor \overline{x}_2)\) is not satisfiable (trying every combination of true or false
for \( x_1 \) and \( x_2 \), we have
\((\text{true} \lor \text{true}) \land (\text{true} \lor \text{false}) \land (\text{false} \lor \text{true}) \land (\text{false} \lor \text{false})\)
= true \& true \& true \& false = false;
\((\text{true} \lor \text{false}) \land (\text{true} \lor \text{true}) \land (\text{false} \lor \text{false}) \land (\text{false} \lor \text{true})\)
= true \& true \& false \& true = false;
\((\text{false} \lor \text{true}) \land (\text{false} \lor \text{false}) \land (\text{true} \lor \text{true}) \land (\text{true} \lor \text{false})\)
= true \& false \& true \& true = false;
\((\text{false} \lor \text{false}) \land (\text{false} \lor \text{true}) \land (\text{true} \lor \text{false}) \land (\text{true} \lor \text{true})\)
= false \& true \& true \& true = false). In the proof
below, "satisfied" means "satisfied by picking the vari-
ables to be true or false in some (possibly unspecified)
way."

We prove that that \((x \lor A) \land (\overline{x} \lor B)\) is satisfiable
iff \((A \lor B)\) is satisfiable.

(1) \( A \lor B \) satisfiable implies \((x \lor A) \land (\overline{x} \lor B)\)
satisfiable.

If \( A \lor B \) is satisfied, then either \( A \) is satisfied, or
\( B \) is satisfied. If \( A \) is satisfied, we satisfy \((x \lor A) \land
(\overline{x} \lor B)\) by choosing \( x \) to be false ((false \lor \text{true}) \land (true
\lor \text{anything}) = true \& true = true); if \( B \) is satisfied,
choose \( x \) true. In either case, we can choose \( x \) freely
because neither \( A \) nor \( B \) contains \( x \).
(2) \((x \lor A) \land (\bar{x} \lor B)\) satisfiable implies \((A \lor B)\) satisfiable.

This can be seen intuitively as follows:

Recall that implication is defined by "\((G \rightarrow H)\) means \((\bar{G} \lor H)\)."

Thus \((x \lor A) \land (\bar{x} \lor B)\) can be rewritten

\((\bar{x} \rightarrow A) \land (x \rightarrow B)\). Now suppose this is satisfied;
then in words, "(not x) implies A, and x implies B."

Thus if x is true, B is true, and if x is false, A is true. But x must be either true or false, so either A or B must be true, i.e. \((A \lor B)\) is satisfied, Q.E.D. This can be proven more formally, of course, as is done in the Davis-Putnam paper; but the argument is slightly involuted.

The method of the Davis-Putnam algorithm is, given a formula G in CNF containing the variable x, to derive a formula H which contains all variables of G except x, and which is satisfiable iff G is. This process is repeated with one variable after another until a formula is derived which is obviously satisfiable or unsatisfiable. As described in their JACM paper, there are three rules for elimination of variables:

(1) If a variable x occurs in G only as x or only as \(\bar{x}\), then delete all clauses containing x (\(\bar{x}\) respectively).

(2) If x or \(\bar{x}\) occurs as an entire one-literal clause, then delete all clauses containing x (\(\bar{x}\) respectively) and delete \(\bar{x}\) (x respectively) from the clauses in which it occurs.
(3) Otherwise rewrite the formula as \((x \lor A) \land (\overline{x} \lor B)\) 
& C, where A, B, and C do not contain \(x\), and replace G by 
\((A \lor B) \land C\).

For reasons that will appear shortly, it is desirable 
to add a fourth rule to the original three. Rule (4) does 
not eliminate a variable, but it does often simplify the 
formula.

(4) If any clause is a subclause of another clause, 
delete the larger clause.

This is often called the "subsumption rule", but 
because usage of "subsume" is not consistent, we will in-
stead refer to it as **annihilation**: the smaller clause 
annihilates the larger one.

**Justifications of Rules 1-4**

Rule (1) preserves satisfiability:
G = A & B, where A contains precisely the clauses in which 
x (\(\overline{x}\) respectively) occurs, and B contains all others. For 
example if G = \((x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x}_2) \land (x_2 \lor \overline{x}_3)\), and 
x = x_1, then A = \((x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x}_2)\), B = \((x_2 \lor \overline{x}_3)\).
Now because \(\lor\) distributes over \(\land\), A may be rewritten as 
x \(\lor\) D (\(\overline{x} \lor\) D respectively), where D is A with x (\(\overline{x}\) re-
spectively) deleted. For example, in the above, D = 
\((x_2 \lor x_3) \land \overline{x}_2\), so
\[
x \lor D = x_1 \lor ((x_2 \lor x_3) \land \overline{x}_2)
= (x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x}_2) \text{ by distributivity}
\text{ of } \lor \text{ over } \land
= A.
\]
Thus we have $G$ equivalent to $(x \lor D) \& B$, which we now claim is satisfiable iff $B$ is satisfiable. But if $(x \lor D) \& B$ is satisfiable, then each must be satisfiable separately ($X \& Y$ satisfiable implies $X$ satisfiable and $Y$ satisfiable); conversely if $B$ is satisfiable, then $(x \lor D) \& B$ is satisfied by making $x$ true (respectively, $(\overline{x} \lor D) \& B$ is satisfied by making $x$ false). We are free to choose $x$ true or false because $x$ does not occur in $B$.

This proves the validity of rule (1).

Rule (2) is really a special case of rule (3), augmented by Rule (4). We restate and prove rule (4) as the following:

**Annihilation rule:**

For any $A$, $B$, $((A \lor B) \& B)$ is equivalent to $B$. This is an elementary theorem of Boolean algebra; it is easily proved in this context by observing that if $B$ is satisfied, then $A \lor B$ is satisfied no matter what $A$ is, so $(A \lor B) \& B$ is satisfied; if $B$ is false, then $C \& B$ is false, no matter what $C$ is; in particular if $C = A \lor B$.

Q.E.D.

The use we make of the annihilation rule is that it allows us to delete all clauses containing $x$ ($\overline{x}$ respectively), because they are all annihilated by the clause $x$ ($\overline{x}$ respectively), and then apply rule (3). How this produces the effect of rule (2) will be explained after the proof of rule (3).
Rule (3) preserves satisfiability:

The formulae A, B, and C are determined as follows:
C is the conjunction of all clauses in G containing neither x nor \( \overline{x} \). B is the conjunction of all clauses in G containing \( \overline{x} \), but with \( \overline{x} \) deleted from each. A is the conjunction of all clauses containing x, but with x deleted. For example, if \( x = x_1 \) and \( G = (x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x}_2) \land \overline{x}_1 \lor x_3 \lor x_4) \land (x_1 \lor \overline{x}_4) \land (x_2 \lor \overline{x}_3) \land (x_3 \lor \overline{x}_4) \), we have

\[
A = (x_2 \lor x_3) \land \overline{x}_2 \\
B = (x_3 \lor x_4) \land \overline{x}_4 \\
C = (x_2 \lor \overline{x}_3) \land (x_3 \lor \overline{x}_4)
\]

By distributivity of \( \lor \) over \( \land \), \( (x \lor A) \land (\overline{x} \lor B) \) & C is equivalent to G.

By the argument given previously, \( (x \lor A) \land (\overline{x} \lor B) \) is satisfiable iff A \( \lor \) B is satisfiable. Moreover since they are satisfied by the same choices of true and false for the variables other than x, and C does not contain x, \( (x \lor A) \land (\overline{x} \lor B) \) & C is satisfiable iff \( (A \lor B) \) & C is satisfiable.

This concludes the justification of rule (3).

In respect to rule (2), WLOG taking the case that it is x which is a one-literal clause, we see that by rule (4) we delete all clauses containing x, and then we have \( G = x \land (\overline{x} \lor B) \land C \), which by application of rule (3) becomes \( (B) \land C \), A being the empty clause. This is what rule (2) says you get.
There remains a problem, however, after rule (3) has been used: \((A \lor B) \land C\) is not in CNF. This is remedied easily but expensively: use distributivity of \(\lor\) over \(\land\), first to the right and then to the left. For instance in the above example,

\[
A \lor B = ((x_2 \lor x_3) \land \overline{x_2}) \lor ((x_3 \lor x_4) \land \overline{x_4})
\]

\[
= (((x_2 \lor x_3) \land \overline{x_2}) \lor (x_3 \lor x_4))
\]

\[
\land (((x_2 \lor x_3) \land \overline{x_2}) \lor \overline{x_4})
\]

\[
= (x_2 \lor x_3 \lor x_3 \lor x_4) \land (\overline{x_2} \lor x_3 \lor x_4)
\]

\[
\land (x_2 \lor x_3 \lor \overline{x_4}) \land (\overline{x_2} \lor \overline{x_4})
\]

\[
= (x_2 \lor x_3 \lor x_4) \land (\overline{x_2} \lor x_3 \lor x_4)
\]

\[
\land (x_2 \lor x_3 \lor \overline{x_4}) \land (\overline{x_2} \lor \overline{x_4})
\].

It should be clear that this "expansion" of \(A \lor B\) may produce as many clauses as (the number of clauses in \(A\)) times (the number of clauses in \(B\)). This is the crucial step which makes it non-trivial to estimate the space used by the algorithm.

Thus the result of one iteration of the Davis-Putnam algorithm without rule (4) applied to \(G\) is \((x_2 \lor x_3 \lor x_4) \land (\overline{x_2} \lor x_3 \lor x_4) \land (x_2 \lor x_3 \lor \overline{x_4}) \land (\overline{x_2} \lor \overline{x_4})\). The third clause contains within it the last clause, so if the annihilation rule is used, this simplifies to \((x_2 \lor x_3 \lor x_4) \land (\overline{x_2} \lor x_3 \lor x_4) \land (x_2 \lor x_3 \lor \overline{x_4}) \land (\overline{x_2} \lor \overline{x_4})\).

A word is in order about the halting condition: if there is a clause consisting only of \(x\), and another clause consisting of only \(\overline{x}\), for that same \(x\), then the formula is
(clearly) unsatisfiable. If at some time there are no clauses left, then the formula is (vacuously) satisfiable.

This concludes my exposition of the Davis-Putnam algorithm as it appeared in that paper.

Because the expansion for rule three must occur every time that rule is used, it seems reasonable to combine the expansion process with rule (3). This gives us the following, which because of its multiplicative nature I call the

**Product Rule:**

To eliminate $x$ from $G$, replace $G$ by $C \land D$, where $C$ is the conjunction of all clauses in $G$ not containing $x$, and $D$ is the conjunction of all clauses $A \lor B$, for every $A$ and $B$ such that $(x \lor A)$ and $(\overline{x} \lor B)$ are clauses of $G$.

It is noteworthy that if the annihilation rule is applied to $G$ and then followed by the product rule, the result is the same as applying rules (1), (2), and (3), except that some redundant (i.e. annihilated) clauses may be deleted. Rule (3) followed by expansion is of course equivalent to the product rule; rule (2) is effected by the annihilation and product rules together, and rule (1) is effected by a literal understanding of how $D$ is formed in the product rule: namely if there is no clause $B$ (or $A$ respectively) then there is no clause $D$.

In order to refer easily to "an iteration of the Davis-Putnam algorithm" I will use the function PRODUCT
to denote application of the "Product Rule", REDUCTION to denote application of the annihilation rule (because the annihilation rule produces a "reduced" form of a formula), and PROD to denote application of PRODUCT followed by REDUCTION. The reduced form of a CNF formula means the result of applying REDUCTION to that formula; to "reduce" a formula is to replace it by its reduced form. Thus PROD(A), for a CNF formula A, is the result of applying one iteration of the Davis-Putnam algorithm to A, and then applying the annihilation rule to that: PROD(A) = REDUCTION (PRODUCT(A)). Note that use of the subsumption rule is mandatory in order to obtain the special-case polynomial bound, as Cook's Example 2 (see appendix) shows. Also note that since A may not be reduced, REDUCTION should be used before PRODUCT to ensure that rule (2) will be realized. In subsequent iterations, of course, the formula is already reduced.
CHAPTER 2

A reformulation of the Davis-Putnam algorithm: an "algebraic" algorithm for tautology detection in disjunctive normal form propositions.

Definitions:

- DNF, disjunctive normal form
- dual, G*
- literal
- tautology
- deMorgan laws
- +, −, 0, 1
- clause (as a vector)
- DNF (as a set of vectors)
- lattice
- W'S (means "occurrences of W")
- v, &, ≤
- complement
- literal (as a lattice element)
- A[i]
- VARS (A)
- annihilation rule for DNF's
- valuation, u, U
- true (for a valuation of a DNF)
- tautology (for a DNF)
- equivalent
Definitions, Continued:

annihilate, <

REDUCTION, reduced form

permutation

∇

∏

PRODUCT

PROD

F(A), FA, F A

locally optimal form

DPDNF

Proofs:

G unsatisfiable iff G* tautology

L 1  REDUCTION well-defined

L 2  REDUCTION(A) equivalent to A

L 3  PROD preserves truth
While the Davis-Putnam algorithm was intended to determine satisfiability of formulae in CNF, the concept of duality in Boolean algebra enables us to define an algorithm which is identical to the Davis-Putnam algorithm except that it determines whether a formula in "DNF" (disjunctive normal form) is a tautology. To form the dual of an expression in formal logic that uses only "and", "or", and "not" as connectives, simply interchange "and" with "or" throughout; any DNF formula is then the dual of some CNF formula. DNF formulae are of the form AvBv...vC, where A might be for example \( x_1 \land \bar{x}_{13} \land x_{25} \), B might be \( x_2 \land x_3 \land x_4 \land \bar{x}_4 \), etc. More explicitly,

**Definition:** Pick \( N \geq 0 \). Let "literal" mean \( x_i \) or \( \bar{x}_i \) for \( 1 \leq i \leq N \). Let "clause" mean "conjunction of literals" (i.e. a string of literals separated by "and", written "\&"). Then a "formula in DNF" is any (finite) disjunction of clauses (i.e. string of clauses separated by "or", written "\lor"). In terms of formal languages, DNF formulae are generated by the rules

\[
\begin{align*}
\text{<subscript>} & + \text{ positive integer} \\
\text{<variable>} & + \text{ } x | x_{\text{<subscript>}} \\
\text{<literal>} & + \text{ <variable> | <variable>} \\
\text{<clause>} & + \text{ <literal> | <clause> \& <literal>} \\
\text{<DNF>} & + \text{ (<clause>) | <DNF> \lor <DNF>} \\
\end{align*}
\]

**Definition:** A formula \( G \) is a tautology iff it is true under every choice of values for its variables.
Remark: A CNF formula $G$ is unsatisfiable iff its dual $G^*$ is a tautology, i.e. unfalsifiable.

Proof: Certainly $G$ is unsatisfiable iff $\overline{G}$ (= "not $G"\) is a tautology. But using the de Morgan laws ($A \lor B = \overline{A} \land \overline{B}$, $\overline{C \land D} = \overline{C} \lor \overline{D}$), we find that $\overline{G}$ is a DNF formula, just like $G^*$ except that every literal has been replaced by its negative.

For example

If $G = (x_1 \lor \overline{x_2}) \land (x_1 \lor x_2 \lor \overline{x_3}) \land (x_1 \lor x_3)\)

Then $\overline{G} = (x_1 \lor \overline{x_2}) \land (x_1 \lor x_2 \lor \overline{x_3}) \land (x_1 \lor x_3)$

$$= (x_1 \lor \overline{x_2}) \lor (x_1 \lor x_2 \lor \overline{x_3}) \lor (x_1 \lor x_3)$$

$$= (\overline{x_1} \land \overline{x_2}) \lor (\overline{x_1} \land \overline{x_2} \land \overline{x_3}) \lor (\overline{x_1} \land \overline{x_3})$$

Whereas $G^* = (x_1 \land \overline{x_2}) \lor (x_1 \land x_2 \land \overline{x_3}) \lor (x_1 \land x_3)$

Now how will the choices of true and false for $x_i$ affect the truth or falsity of $G^*$? To indicate a choice for $x_i$ we will say that $x_i$ is true if $i \in I$ and false if $i \in J$. Then if $G$ is false under that choice, $\overline{G}$ will be true; we can make $G^*$ true by reversing $I$ and $J$, i.e. let $x_i$ be true if $i \in J$ and false if $i \in I$.

Continuing the above example, let $I = \{2\}$ and $J = \{1, 3\}$, so $x_2 = true$, $x_1 = x_3 = false$. Then,

$$G = (false \lor false) \land (false \lor true \lor true)$$

$$\land (false \lor false)$$

$$= (false) \land (true) \land (false)$$
\[ G = \text{false} \]

Using the DNF form of \( G \),

\[ G = (\text{true} \& \text{true}) \lor (\text{true} \& \text{false} \& \text{false}) \lor (\text{true} \& \text{true}) \]

\[ = \text{true} \lor \text{false} \lor \text{true} \]

\[ = \text{true}. \]

Reversing \( I \) and \( J \), we have \( x_1 = x_3 = \text{true}, \ x_2 = \text{false} \), and

\[ G^* = (\text{true} \& \text{true}) \lor (\text{true} \& \text{false} \& \text{false}) \lor (\text{true} \& \text{true}) \]

\[ = \text{true} \lor \text{false} \lor \text{true} \]

\[ = \text{true}, \]

as we expected.

Note that under the original choices \( x_2 = \text{true}, \ x_1 = x_3 = \text{false} \), we have,

\[ G^* = (\text{false} \& \text{false}) \lor (\text{false} \& \text{true} \& \text{true}) \lor (\text{false} \& \text{false}) \]

\[ = \text{false} \lor \text{false} \lor \text{false} \]

\[ = \text{false}. \]

One can make no general statement about the respective values of \( G \) and \( G^* \) under the same choices for variable values.

How do we show now that \( G^* \) is a tautology iff \( G \) is unsatisfiable? Simply that "\( G \) is unsatisfiable" means that every choice of values for variables makes \( G \) false, hence interchanging the true and false variables makes no
difference, so every choice of values makes \( G^* \) true; i.e. \( G^* \) is a tautology. The converse follows because \( (G^*)^* = G \).

Q.E.D.

Since the tautology problem seems to have enjoyed a wider popularity than the satisfiability problem, I will now "dualize" the preceding; it should be understood that no further proofs are needed, and that at each step in the DNF tautology algorithm the dual of the corresponding step in the CNF unsatisfiability algorithm is used; however I am supplying proofs of a somewhat different flavor as an example of the use of my "algebraic" notation which is described below.

In approaching the problem of computational complexity of the Davis-Putnam algorithm, I felt that a more algebraic notation would be helpful. First, because we are concerned solely with formulae in DNF (or CNF; I shall omit reminding the reader of duality henceforth), one can omit mention of "and" and "or", and make the convention that \( \{x_1, \overline{x}_3, x_4\}, \{x_2, \overline{x}_4\}, \{\overline{x}_1, x_2, x_4\} \) represents \( (x_1 \& \overline{x}_3 \& x_4) \lor (x_2 \& \overline{x}_4) \lor (\overline{x}_1 \& x_2 \& x_4) \). Of course, the symbol \( x \) is not really necessary either; we could equally well replace the above sets by, say, \( \{1, -3, 4\}, \{2, -4\}, \{-1, 2, 4\} \). However, this notation suggests arithmetic possibilities (such as \( 1+2=3 \)) which are not intended. Also, I wished to use interactive computation for analysis, and sets, strings, and lists are not convenient for a simple interactive language.
such as APL. My solution to this was to omit mention of the indices; in place of each clause, a vector is used whose $i$th component indicates the status of $a_i$ in the clause. Thus the previous example becomes \{(+,1,−,+), (1,+,-), (-,+,+))\}. The reason that 1 is used for "no occurrence" will become apparent shortly.

Since the central part of this paper is independent of the notation used for formulae, I will refer to these vectors "par abus de langage" as clauses; a clause means either a clause in formal logic notation or a clause in the special DNF notation.

The name that I give to the algebraic representation of DNF formulae is simply "a DNF". The symbols of which it is composed are the elements of a Boolean lattice, \{0,1,+,−\}. The diagram for this lattice is

```
1
 / \      
/  \     /
+/   −   /
\  \
  0
```

In terms of the lattice operations $\vee$ (sup), $\&$ (inf), we have

$+\vee− = 1$
$+\&− = 0$

As in any lattice, for every $x,y (= 0,1,+,or −)$, we have

$x \vee y = y \vee x$
$x \& y = y \& x$
$x \vee 1 = 1$
$x \& 1 = x$
$x \vee 0 = x$
$x \& 0 = 0$
The relation "\( \leq \)" is defined by \( a \leq b \) iff \( a \lor b = b \).

Geometrically, \( x \leq y \) iff \( x \) is below \( y \) in the diagram. Thus for all \( x, 0 \leq x \leq 1 \). However

\[ + \downarrow - \text{ and } - \uparrow + . \]

\( + \) is called the "complement" of \(-\) (and vice versa), and \(0\) is the complement of \(1\).

Convention: for any symbol \(W\), "\(W's\" means "occurrences of \(W\)".

Since my purpose in introducing this system is to represent logical formulae, I will call the elements of the lattice "literals". A clause is then a vector of literals, with the lattice operations extended elementwise.

For example,

\[
(+1+-0+1) \\
& (l-l+1-l) \\
= (+-+0001) ,
\]

\[
(+1+-0+1) \\
v (l-l+1-l) \\
= (1111111) ;
\]

\[
(+0+1+++) \\
\leq (+1++1l++) .
\]

If \(A\) is a clause, then \(A[i]\) is the \(i^{th}\) literal of \(A\) (and represents \(x_i\) in the corresponding DNF formula). Thus \(A \leq B\) iff for all \(i\), \(A[i] \leq B[i]\). I shall avoid the use of "\(A \leq B\)" for clauses \(A\) and \(B\) because of ambiguity as to whether we mean "\(A \leq B\) and \(A \neq B\)" or "for all \(i\), \(A[i] \leq \)
B[i]". Parentheses will be used only for readability.

Any formula contains a certain finite number of distinct variables; thus each clause in a DNF A has the same length, denoted VARS(A) or, where the reference is clear, simply V. It is convenient to write the clauses one above the other; thus \((x_1 \& \overline{x}_3 \& x_4) \vee (x_2 \& \overline{x}_4) \vee (\overline{x}_1 \& x_2 \& x_4)\) becomes

\[
\begin{align*}
1+ & \quad \text{+1+} \nonumber \\
1- & \quad \text{+1-} \\
-1 & \quad \text{--} \\
\end{align*}
\]

This is then an array of width VARS(A).

Formally: A **DNF** is a set of vectors, all of the same length, over \(\{0, 1, +, -\}\) (Note that since a DNF is a set, the number of clauses in A means the number of unequal clauses in A). If G is a formula, then A is the "DNF for G" iff the clauses of A represent precisely the clauses of G; a DNF clause B represents a formal-logic clause H iff the following table is satisfied: for each \(i, 1 \leq i \leq \text{VARS(B)},\)

\[
\begin{align*}
H & \text{ contains } G[i] = \\
x_i & + \\
\overline{x}_i & - \\
x_i \& \overline{x}_i & 0 \\
\text{neither } x_i \text{ nor } \overline{x}_i & 1
\end{align*}
\]

From the following examples, it is easy to see how the DNF algebra reflects the structure of the propositional logic.
\[(x_1 \& x_2) \& (\overline{x}_3 \& x_4) = (x_1 \& x_2 \& \overline{x}_3 \& x_4)\]

\[= (++11) \& (11-+)\]

\[= (+++-) \; ;\]

\[(x_1 \& x_3) \& (x_2 \& \overline{x}_3 \& x_4) = (x_1 \& x_2 \& x_3 \& \overline{x}_3 \& x_4)\]

\[= \text{false}\]

\[= (++0+), \text{ so a clause}\]

containing a 0 is false.

\[(x_1 \& x_2 \& x_4) \lor (x_1 \& x_2) = (x_1 \& x_2) \quad \text{(annihilation rule for DNF formulae)}\]

\[= (++1+1) \lor (1+-+)\]

\[= (++11) \quad \text{(annihilation rule for DNF's)}\]

Note that even though we may write a DNF A as an array, the order of the clauses (or "rows", in the usual array terminology) is immaterial. Also, the order of the columns merely indicates the names of the variables, so interchanging columns of A does not affect whether A represents a tautology. Note also that to represent CNF formulae and solve the satisfiability problem, we just re-interpret \& as "or", \lor as "and", 0 as "true", and 1 as "false" (in short, use duality).

Note that since a single clause is a DNF, any statement about DNF's applies also to clauses.
It is interesting from both a theoretical and computational point of view that the lattice \{+-01\} can itself be expressed as a vector lattice, this time over the lattice \{01\} (just as a DNF is a set of vectors over the lattice \{+-01\}). Namely, let + = 01, \(-\) = 10 (the complement of +), 0 = 00, 1 = 11 (the complement of 0). Then all the relations defined for \{+-01\} hold for \{01 10 00 11\}, and a DNF is readily expressed as a \(2 \times \text{number of clauses}\) \(\times\) (number of variables) binary array. Only the standard binary functions are needed to do all the DNF operations, which makes handling large formulae quite feasible.

(For instance a formula of 121 clauses in 30 variables required 8 minutes in APL/360).

Convention: Throughout, A and B are arbitrary DNF's.

A choice of true or false for each variable of a formula G determines a "valuation" of G (as either true or false); for instance the valuation of \(G = (x_1 \& \overline{x_2}) \lor (\overline{x_1})\) under \(x_1 = \text{true}, x_2 = \text{false}\) is \((\text{true} \& \text{true}) \lor \text{false} = \text{true} \lor \text{false} = \text{true}\). We wish to retain the same concept for DNF's; this means that if \(x_i\) is to be replaced by true, then for any clause \(R\), \(R[i] = +\) should be replaced by 1, and \(R[i] = -\) should be replaced by 0; conversely if \(x_j\) is chosen false, \(R[j] = +\) should be replaced by 0, and \(R[j] = -\) should be replaced by 1. Formally,

**Definition:** \(v\) is a valuation function (on a DNF A) if

\[ v : \{1, \ldots, \text{VARS}(A)\} \rightarrow \{0, 1\}. \]
Definition: B is a valuation of A under v if each clause S of B is obtained as follows:

Let R be a clause of A; then

for each i we have:

\[
\begin{array}{c|c|c}
R[i] & S[i] \\
\hline
0 & 0 & 0 \\
1 & 0 & 1 \\
+ & 1 & 0 \\
- & 0 & 1 \\
\end{array}
\]

Thus for instance if A = +1- -1 and v is given by

\[
\begin{array}{cccccccc}
i & 1 & 2 & 3 & 4 & 5 & 6 \\
v(i) & 1 & 1 & 0 & 0 & 1 & 0 \\
\end{array}
\]

then the valuation of A under v is 111100.

If B = +1+1

+11-

+1++

and u is given by

\[
\begin{array}{cccc}
i & 1 & 2 & 3 \\
u(i) & 1 & 0 & 0 \\
\end{array}
\]

then the valuation of B under u is 1101

1111

1100.

Definition: If u is a valuation function, then U, also called a valuation function, is the function such that U(R) = the valuation of R under u. For instance in the previous example, U(B) = 1101

1111

1100.

Definition: B is a valuation iff it is a DNF whose only
literals are 0 and 1.

**Definition:** 1 (an underlined one) is a clause of all 1's of the appropriate length (and 0 a clause of all 0's). Thus 1 is the only clause that has the value true, since false & R = false for any R. For example, the clause 110 represents the formal logic clause true & true & false = false.

**Definition:** A valuation B is **true** iff 1 is a clause of B (B is **false** otherwise). This is because true v G = true for any G, whereas false v false = false. For example, 10 represents (true & false) v (false & true) = false v false = false, whereas 01 represents (true & false) v 10 (false & true) v (true & true) = false v false v true = true.

**Definition:** A is a **tautology** (or **true**) iff every valuation of A is true.

**Definition:** A is **equivalent** to B iff VARS(A) = VARS(B) and for every valuation V, V(A) is true iff V(B) is true (thus in particular, if A is equivalent to B, then A is a tautology iff B is a tautology).

**Remark:** If the symmetrical set difference of A and B (i.e. \((B \cap A^C) \cup (A \cap B^C)\)) contains only clauses that contain 0 as some element, then A and B are equivalent.

**Proof:** Valuations of said clauses can never contain all 1's. Q.E.D.

(Thus one may freely insert or delete clauses containing
0's.)

The notion of "no redundant clauses" is specified by the term "reduced". Intuitively, a formula in DNF is reduced if you are never wasting your time (for the purpose of finding a tautology) looking at some clause; two ways you know for sure you are wasting your time looking at a clause are (1) The clause is never true, i.e. contains $x$ and $\overline{x}$; (2) If the clause in question is true only when some other clause is true. In DNF representation, (1) means the clause contains 0. (2) is the annihilation rule: let $R$ be a subclause of a clause $S$. Any valuation that makes $S$ true must give every literal of $S$ the value true; in particular every literal in $R$ is true. Hence $R$ is true whenever $S$ is true; in other words $S$ is true only when $R$ is true. In the DNF representation, $R$ is a subclause of $S$ iff $S \preceq R$ (i.e. $R$ has more 1's than $S$), and so we can read $\preceq$ as meaning "is less true than". For example $(x_1 \& \overline{x}_3)$ is a subclause of $(x_1 \& x_2 \& \overline{x}_3 \& \overline{x}_4)$; in DNF representation, $(++--) \preceq (+1-1)$. Likewise $(x_1 \& x_2)$ is a subclause of $(x_1 \& x_2 \& \overline{x}_2 \& x_3)$, and $(+0+) \preceq (++1)$.  

**Definition:** $R$ annihilates $S$ iff $S \preceq R$ (i.e. $S \lor R = R$). By the above discussion, annihilation means the same thing for DNF clauses as for formal-logic clauses.

**Definition:** A DNF $A$ is **reduced** iff

1. No literal of $A$ equals 0
2. For no $R \neq S$ is $S \preceq R$
We now wish to produce from any DNF $A$ a reduced DNF equivalent to $A$. This is necessary not merely to mimic rule (2) of the Davis-Putnam algorithm, but also to ensure that the modified Davis-Putnam algorithm analyzes symmetrical expressions in polynomial time; cf. Stephen Cook's Example 2 in the Appendix. 

Definition: The reduced form of a DNF $A$, denoted \text{REDUCTION}(A), is a DNF $B$ such that

1. $B$ is reduced;
2. $S$ is a clause of $B$ only if $S$ is a clause of $A$;
3. if $S$ is a clause of $A$, then either $S$ is in $B$, or $S$ is annihilated by some clause of $B$, or $S$ contains a 0 (in brief, either $S$ contains 0 or for some $R$ in $B$, $S \leq R$).

"To reduce" $A$ means to replace $A$ by a reduced form of $A$, i.e. to delete all redundant clauses by using the annihilation rule. Note that since $A$ is a set of clauses, "$R \in A$" means "$R$ is a clause of $A$".

Lemma 1: The reduced form of $A$ is unique.

Proof: Let $B$, $C$ be reduced forms of $A$.

Let $R$ be a clause of $B$.

To show $R$ is a clause of $C$ (hence $B \subseteq C$, but since $B$ and $C$ are arbitrary, $C \subseteq B$ and $B = C$).

If $R$ is a clause of $B$, then by the definition of "reduced form of", $R$ is a clause of $A$. Therefore since $C$
is a reduced form of $A$, there is a clause $S$ in $C$ with $R \leq S$. But if $S$ is a clause of $C$, $S$ is a clause of $A$, so $B$'s being a reduced form of $A$ implies that there is a $T$ in $B$ with $S \leq T$. Thus $R \leq S \leq T$; in particular $R \leq T$.

Since $B$ is reduced, $R = T$. But $T \in C$, so $R \in C$, and so $B \subseteq C$; as indicated above, this shows $B = C$.

Q.E.D.

**Notation:** For any function $F$, if $S = F(R)$ then $(F(R))[i]$ denotes $S[i]$.

**Lemma 2:** Reduction preserves truth, i.e. $A$ is a tautology iff $\text{REDUCTION}(A)$ is a tautology.

**Proof:** We will show in fact that if $B$ is the reduced form of $A$, and $v$ is a valuation function, then the valuation of $A$ under $v$ is true iff the valuation of $B$ under $v$ is true. Recall that for any valuation function $v$, $V(A)$ denotes the valuation of $A$ under $v$.

$(\Rightarrow)$ Given $V(A)$ is true, to show $V(B)$ is true.

Suppose $A$ is true, then $A$ contains at least one true clause.

Let $R$ be a clause of $A$, with $V(R) = 1$. Then $R$ contains no 0 (since $V(0) = 0$ for any $V$), so by the definition of "reduced form of", there is a clause $S$ of $B$ with $R \leq S$. For each $i$, if $R[i] \neq 0$ and $R[i] \leq S[i]$ then either $R[i] = S[i]$ or $S[i] = 1$. Hence for each $i$ $(V(R))[i] = 1$ implies $(V(S))[i] = 1$, so $V(R) = 1$.

$\Rightarrow V(S) = 1$. Since $S$ is a clause of $B$, $V(B)$ is true,
as desired.

(\leq) Given \( V(B) \) is true, to show \( V(A) \) is true. \( B \subseteq A \), so

if \( R \) is a clause of \( B \), \( R \) is a clause of \( A \); in particular let \( R \) be a row of \( B \) with \( V(R) = 1 \); then \( V(A) \) contains 1, i.e. \( V(A) \) is true, as desired.

Q.E.D.

An algorithm for forming REDUCTION(A) is: (1) if \( R \) is a clause of \( A \) containing 0, delete \( R \); (2) for each pair of clauses \( R \neq S \) in \( A \), test whether \( R \leq S \). If so delete \( R \). Because \( \leq \) is transitive, the order in which we do this is immaterial; the only pitfall to avoid is, that if the form in which \( A \) is expressed contains the clause \( R \) two or more times, we will delete \( R \); thus we must either check equality separately (and before "proper" annihilation), or so order the search that, say, the last instance of \( R \) will always be saved. The latter approach is the one I use in my APL implementation of REDUCTION. In any event, the process takes at worst on the order of \((VARS(A)) \times (number \ of \ clauses \ in \ A)^2\) time. It would be very nice to find a significantly faster algorithm.

As I have mentioned before, one of my aims is to make the Davis-Putnam procedure simpler and more algebraic, since it should be easier to prove something about an algebraic operation than about an algorithm. One simplification is, the variable of elimination shall always be \( x \). If you want to eliminate \( x_i \), simply interchange indices 1 and \( i \); this
changes the formula, but does not affect whether it is a tautology. In the DNF representation, this means you interchange columns 1 and i. An interchange is a special case of a permutation, and we shall have use for the latter, so I will formalize this.

Definition: Let $\sigma$ be any permutation on \{1, \ldots, VARS(A)\}. Then $\sigma A$ is defined to be that DNF B such that R is a clause of B iff there is a clause S of A with $R[i] = S[\sigma i]$ for each i (in other words, $\sigma$ replaces the i\textsuperscript{th} column by the (\sigma i)\textsuperscript{th} column). For example if

$$A = +1+1$$
$$+11-$$
$$+1++$$

and $\sigma$ is given by

$$i \quad 1234$$
$$\sigma(i) \quad 2314$$

then

$$A = l++1$$
$$l1+-$$
$$l+++$$

Thus if we wish to eliminate, say, $x_4$ from A, we let $\sigma(1) = 4$, $\sigma(4) = 1$, $\sigma(i) = i$ otherwise, and eliminate $x_1$ from A. After elimination, we decrement all indices by one.

By way of algebraization of the Davis-Putnam algorithm, I will replace rule (3), which requires looking through the clauses R of A to see what R[1] is (remember we always eliminate $x_1$), by a function called "PRODUCT", which essentially realizes the DNF analog of the "Product rule" for CNF for-
mulae. In place of searching through the clauses, we will do an algebraic operation \( \oplus \) on every pair of clauses in \( A \) (including \( R \oplus R \) for each \( R \)), which contains a "selector" operator called \( \ominus \), which selects the appropriate pairs of clauses for rule (3): \( R, S \) is appropriate if \( x_1 \) is in \( R \) and \( \overline{x}_1 \) is in \( S \), or \( R = S \); the latter requirement will have to be loosened a little. The effect of \( \text{PRODUCT} \) when combined with \( \text{REDUCTION} \), is to produce exactly the clauses \( R \& S \) such that \( (x_1 \& R) \), \( (\overline{x}_1 \& S) \) are clauses of \( A \), as well as all \( T \) such that \( x_1 \) does not occur in \( T \) and \( T \) is a clause of \( A \). It will be evident that \( \text{PRODUCT} \) is not an efficient algorithm; however, it is a very simple algebraic operation. Recall that \( R[1] = + \) in DNF representation iff \( x_1 \) is in \( R \), and \( R[1] = - \) if \( \overline{x}_1 \) is in \( R \).

Also recall that rule (1) is taken care of by an appropriate interpretation of \( \text{PRODUCT} \) (this interpretation is unavoidable in the DNF representation), and rule (2) is equivalent to \( \text{REDUCTION} \) followed by \( \text{PRODUCT} \).

**Definition:**  \( \ominus \) is an operation between literals defined by \( 1 \ominus - = 1 \), \( 1 \ominus 1 = 1 \), \( a \ominus b = 0 \) otherwise.

**Definition:** \( \oplus \) is an operation between clauses of equal length defined by \( (R \oplus S)[i] = (R[1] \ominus S[1]) \& (R[i+1] \& S[i+1]), 1 \leq i \leq \text{VARS}(R) - 1, \) and \( \text{VARS}(R \oplus S) = \text{VARS}(R) - 1 \).

For example if \( R = 1l--+ \) and \( S = 1--l+ \) then \( R \oplus S = S \oplus R = --++ \). If \( U = +l--+ \) and \( V = --l+- \) then \( U \oplus V = +++- \) but \( V \oplus U = 0000 \). \( \oplus \) is asymmetrical because \( \ominus \) is; we don't
need both A & B and B & A in the product rule.)

**Definition:** PRODUCT is a function on DNF's defined by

\[
\text{PRODUCT}(E) = \{ R \oplus S | R, S \text{ are clauses of } E \} \text{ (including } \ R \oplus R \text{ for each } R, \text{ of course).}
\]

I will indicate why this is equivalent to Davis-Putnam rule (3): namely, if \( x_1 \) is the variable being eliminated, then (1) if \( R \) represents a clause \( R' \) not containing \( x_1 \), then \( R \oplus R \) (which still represents \( R' \) since \( R[i] = 1 \) and \( 1 \lor 1 = 1 \)) will occur in \( \text{PRODUCT}(A) \); (2) if \( R' \) contains \( x_1 \) and \( S' \) contains \( \overline{x}_1 \), then \( R \oplus S \) occurs, which represents \( (R' \text{ with } x_1 \text{ deleted}) \land (S' \text{ with } \overline{x}_1 \text{ deleted}) \). There are also some extra clauses inserted (other than unsatisfiable ones), but they all represent \( R' \land S' \) where \( R' \) and \( S' \) are both clauses not containing \( x_1 \), and hence \( R' \) will also occur, and \( R' \land S' \) will be annihilated by \( R' \) in the reduced form of \( \text{PRODUCT}(A) \). I will shortly give a rigorous proof that \( \text{PRODUCT} \) does preserve tautologies.

**Definition:** \( \text{PROD}(A) = \text{REDUCTION}(\text{PRODUCT}(A)) \)

We will now perform \( \text{PRODUCT} \) on the DNF for \( G^* \), where \( G \) is the formula considered in the section on rule (3),

\[
G = (x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x}_2) \land (\overline{x}_1 \lor x_3 \lor x_4) \land (\overline{x}_1 \lor \overline{x}_4) \land (x_2 \lor \overline{x}_3) \land (x_3 \lor \overline{x}_4)
\]

\[
G^* = (x_1 \land x_2 \land x_3) \lor (x_1 \land \overline{x}_2) \lor (\overline{x}_1 \land x_3 \land x_4) \lor (\overline{x}_1 \land \overline{x}_4) \lor (x_2 \land \overline{x}_3) \lor (x_3 \land \overline{x}_4).
\]

I will show rule (3) and \( \text{PROD} \) side-by-side for illustration; to avoid confusion, I will use the Boolean dual of
rule (3). Let $A$ be the DNF for $G^*$.

$$G^* = (x_1 \& x_2 \& x_3) \vee (x_1 \& \overline{x_2}) \vee (\overline{x_1} \& x_3 \& x_4) \vee (\overline{x_1} \& \overline{x_4}) \vee (x_2 \& \overline{x_3}) \vee (x_3 \& \overline{x_4})$$

The result of dual-rule (3) is $(A \& B) \vee C$, where

$$G^* = (x_1 \& A) \vee (\overline{x_1} \& B) \vee C$$

Thus $G^* = (x_1 \& ((x_2 \& x_3) \vee \overline{x_2})) \vee (\overline{x_1} \& ((x_3 \& x_4) \vee \overline{x_4})) \vee ((x_2 \& \overline{x_3}) \vee (x_3 \& \overline{x_4}))$.

And $A = (x_2 \& x_3) \vee (\overline{x_2})$

$B = (x_3 \& x_4) \vee (\overline{x_4})$

$C = (x_2 \& \overline{x_3}) \vee (x_3 \& \overline{x_4})$

Using distributivity,

$$(A \& B) = (x_2 \& x_3 \& x_3 \& x_4) \vee (x_2 \& x_3 \& \overline{x_4}) \vee (\overline{x_2} \& x_3 \& x_4) \vee (\overline{x_2} \& \overline{x_4})$$

For readability I will list the products of the $i^{th}$ by the $j^{th}$ clause in this order (let $i$ stand for the $i^{th}$ clause)

1 [F] 1 2 [F] 1 ... 6 [F] 1
1 [F] 2
.
.
.
1 [F] 6 . . . . . . . . . . 6 [F] 6

PRODUCT (A) =

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | + | + | + | + |
| + | + | + | + | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| + | - | - | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | + | + | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | + | + | 0 | 0 | 0 | 0 | 0 | 0 |

Deleting all clauses containing 0, we have

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| + | + | + | + |
| + | + | + | + |
| + | + | + | + |
| + | + | + | + |
| + | + | + | + |
| + | + | + | + |


so \((A&B)vC = (x_2 \land x_3 \land x_4)
\v (x_2 \land x_3 \land \bar{x}_4)
\v (\bar{x}_2 \land x_3 \land x_4)
\v (\bar{x}_2 \land \bar{x}_4)
\v (x_2 \land \bar{x}_3)
\v (x_3 \land \bar{x}_4)\)

We see that \((x_3 \land \bar{x}_4)\) is a subclause of \((x_2 \land x_3 \land \bar{x}_4)\), so the latter is annihilated, and we have for a final result,

\((x_2 \land x_3 \land x_4)
\v (\bar{x}_2 \land x_3 \land x_4)
\v (\bar{x}_2 \land \bar{x}_4)
\v (x_2 \land \bar{x}_3)
\v (x_3 \land \bar{x}_4)\)

(\text{Note that it is merely fortuitous that } 5 \oplus 6 \text{ and } 6 \oplus 5 \text{ each had a 0, but that in any case } (5 \oplus 6) \leq (5 \oplus 5),
(5 \oplus 6) \leq (6 \oplus 6), (6 \oplus 5) \leq (5 \oplus 5), (6 \oplus 5) \leq (6 \oplus 6).)

We see that \((++-) \leq (1-+)\), so,

\[
\text{PROD}(A) = \text{REDUCTION(PRODUCT}(A))
\]

\[
= +++
-++
-1-
+-1
l+-
\]

In formal-logic notation, this is

\[
(x_1 \land x_2 \land x_3)
\v (\bar{x}_1 \land x_2 \land x_3)
\v (\bar{x}_1 \land \bar{x}_3)
\v (x_1 \land \bar{x}_2)
\v (x_2 \land \bar{x}_3)\)

We see that the results are identical.

\textbf{Notation:} For any function } F, \text{ } FA = F \ A = F(A). \ \text{I now show an important result (which saves rigorously proving the}
equivalence of PROD with one iteration of DP using sub-
sumption):

Lemma 3: PROD preserves truth, i.e. A is a tautology iff
PROD(A) is a tautology.

Proof: Since I have shown that REDUCTION preserves truth,
it remains to show that PRODUCT does also.

(=>) Given that A is a tautology and \( v \) is a valuation func-
tion on \( B = PRODUCT \ A \); to show that for some clause \( T \) of
\( B \), \( V(T) = 1 \). Note that there are two possible exten-
sions of \( v \) to a valuation function on \( A \), namely \( u_1 \) and
\( u_2 \) given by choosing a value true or false for the new
variables:

\[
\begin{align*}
  u_1(i) &= \begin{cases} 
  v(i-1) & \text{if } i > 1 \\
  1 & \text{if } i = 1
  \end{cases} \\
  \text{and } \quad u_2(i) &= \begin{cases} 
  u_1(i) & \text{if } i > 1 \\
  0 & \text{if } i = 1
  \end{cases}
\end{align*}
\]

There are 2 cases:

1. a clause \( R \) of \( A \) exists such that \( U_1(R) = U_2(R) = 1 \)
   (remember \( U \) is the function on DNF's induced by
   \( u \))

2. \( U_1(R) = 1 \) only if \( R[1] = +, \) and \( U_2(R) = 1 \) only if \( R[1] = - \).

In case (1), \( V(R \oplus R) = 1 \) and we are done.

In case (2), let \( R, S \) be chosen so that \( U_1(R) = U_2(S) = 1 \).

Claim: If \( R \) and \( S \) are any clauses of equal length, and
\( V \) is any valuation function, then \( V(R \& S) = V(R) \& V(S) \).

Proof of claim:

We show that for each \( i \), \( (V(R \& S))[i] = (V(R) \& V(S))[i] \)

For convenience, pick \( i \) and let
\[ r = R[i] \]
\[ s = S[i], \text{ and let} \]
\[ W(T[i]) = (V(T))[i] \text{ for any } T \]
We have
\[ (V(R) \& V(S))[i] = (V(R))[i] \& (V(S))[i] \]
\[ = W(R[i]) \& W(S[i]) \]
\[ = W(r) \& W(s) \]
also
\[ (V(R \& S))[i] = W((R \& S)[i]) \]
\[ = W(R[i] \& S[i]) \]
\[ = W(r \& s) \]
Hence we need to show that \( W(r \& s) = W(r) \& W(s) \).
There are 16 possible combinations for \( r \) and \( s \),
and \( v(i) \) could be 0 or 1, but these 32 possibilities can be condensed into 4 cases:

(i) \( r = 1 \). Then no matter what \( s \) is,
\[ W(r \& s) = W(1 \& s) = W(s), \text{ and} \]
\[ W(r) \& W(s) = 1 \& W(s) \text{ (since } (V(R))[i]=1 \text{ if } R[i]=1, \]
for any valuation function \( V \) = \( W(s) \), which equals \( W(r \& s) \) by the previous string of equations.
Since \( x \& y = y \& x \), this also takes care of \( s = 1 \),
by interchanging \( r \) and \( s \).

(ii) \( r = 0 \). Then
\[ W(r \& s) = W(0 \& s) = W(0) = 0 \text{, since } (V(T))[i] = 0 \text{ if } T[i] = 0; \]
and $W(r) \& W(s) = W(0) \& W(s) = 0 \& W(s) = 0$.

Likewise for $s = 0$.

(iii) If $r = s$ then
$$W(r\&s) = W(r\&r) = W(r),$$
and $W(r) \& W(s) = W(r) \& W(r) = W(r)$.

(iv) If $r = +, s = -$ then
$$W(r\&s) = W(+\&-) = W(0) = 0$$
and $W(r) \& W(s) = W(+) \& W(-)$

- case (a): $= 1\&0$ if $V(i) = 1$
  $= 0$
- case (b): $= 0\&1$ if $V(i) = 0$
  $= 0$ again.

Likewise for $r = -, s = +$. This exhausts the possibilities and proves the Claim that
$$W(r\&s) = W(r) \& W(s).$$

Now consider $V(R \oplus S)$. Since $R[i] = +$ and $S[i] = -$,
$$R[i] \bigoplus S[i] = 1,$$
so $(R \oplus S)[i] = (R\&S)[i+1]$, and by choice of $U_1$,
$$(V(R \oplus S))[i] = (U_1(R\&S))[i+1] \text{ (also } = (U_2(R\&S))[i+1],$$
but we don't need both facts.)

$$= (U_1(R))[i+1] \& (U_1(S))[i+1] \text{ by Claim above}$$

$$= (U_1(R))[i+1] \& (U_2(S))[i+1] \text{ since }$$
$$U_1(i) = U_2(i), i>1$$
$$= 1\&1 \text{ since } U_1(R) = U_2(S) = 1.$$
(<=) If \( A \) is not a tautology, then neither is \( \text{PRODUCT}(A) \).

Suppose \( A \) is not a tautology, let \( V(A) \) be a false valuation of \( A \), then I will show that if \( v \) is the valuation function for \( V \), and if \( u(i) = v(i+1) \), \( 1 \leq i \leq \text{VARS}(A) - 1 \), then the valuation \( U(B) \) of \( B = \text{PRODUCT} A \) under \( u \) is false, i.e. every clause \( T \) of \( U(B) \) contains a \( 0 \). Precisely, for each \( T \) of \( B \), we will exhibit an \( i \geq 1 \) such that \( T[i] = 0 \). Now by the definition of \( \oplus \), \( R \oplus S \) will definitely contain 0's unless \( (R[1] = S[1] = 1) \) or \( (R[1] = + \text{ and } S[1] = -) \); so we need only consider \( T = R \oplus S \), with either \( R[i] = S[i] = 1 \) or \( R[i] = +, S[i] = - \). In this case \( T[i] = R[i+1] \& S[i+1] \), and since we proved \( V(R \& S) = V(R) \& V(S) \), we need only show that for some \( j \geq 2 \), either \( (V(R))[j] = 0 \) or \( (V(S))[j] = 0 \) (because then \( T[j-1] = 0 \) for some \( j-1 \geq 1 \)).

If \( R[1] = S[1] = 1 \) then \( (V(R))[1] = 1 \), and since \( V(R) \) contains a \( 0 \), \( (V(R))[j] = 0 \) for some \( j \geq 2 \), as required (indeed \( (V(S))[k] = 0 \) also for some \( k \geq 2 \)). Now suppose \( R[1] = +, S[1] = - \). If \( v(1) = 1 \), then \( (V(R))[1] = 1 \), so again \( (V(R))[j] = 0 \), for some \( j \geq 2 \). If \( v(1) = 0 \), then \( (V(S))[1] = 1 \), so now \( (V(S))[j] = 0 \) for some \( j \geq 2 \), as required. In every case one of \( V(R) \) or \( V(S) \) contains a 0 that will be passed on to \( U(T) \).

Q.E.D.

Cook's Example 1 (appendix) shows that in general it is important to choose the variable of elimination judi-
ciously. Probably the best scheme would be to compute \( \text{PROD}(\sigma A) \) for all of the \( (\text{VARS}(A)) \) possible first columns, and keep the smallest resulting DNF for the next iteration. A much simpler scheme is to choose \( \sigma \) so that the first column minimizes \( \text{(number of '+'s) x (number of '-'s)} \). In my APL implementation I used this criterion, with the added provision that if any clause contained only one literal, that should be the variable of elimination, unless some variable appeared in only one sense. For the class of formulae which I analyzed, the variable of elimination is immaterial, so I shall not specify a particular scheme for choosing the elimination variable, and use the term "Locally optimal form of A" to mean some appropriate \( \sigma A \).

I now give an algorithm for the (dual) Davis-Putnam algorithm (in the CNF interpretation, replace "tautology" by "unsatisfiable" and "not a tautology" by "satisfiable").

**Definition:** DPDNF is the following algorithm:

**DPDNF**

read A
A = REDUCTION(A)

BEGIN: A = locally optimal form of A
A = PROD A
if A = 1 or A = \{+, -\} then say "tautology" and halt.
if A is empty then say "not a tautology" and halt.
go to BEGIN
This completes the algebraization of the Davis-Putnam algorithm. In the next two chapters we will use this formalization to derive a polynomial bound on tautology recognition of symmetrical DNF formulae, using the Davis-Putnam algorithm with the annihilation rule. The first concern will be to describe symmetric formulae, and discover how they are affected by a single iteration of the Davis-Putnam algorithm.
CHAPTER 3

Symmetric DNF's: characterization; effect of PROD.

Definitions:

Symmetric
RSDNF
R⁺, R⁻
clause-type, (R⁺, R⁻) = r = (r⁺, r⁻)
CT(E)
annihilate (clause-types)
⊙, □
PRODⁿE
adjacent, adj
A-B (A,B sets)
□ (clause-types)

Proofs:

L4 Symmetrical iff every permutation of every clause is again a clause.

L5 REDUCTION preserves symmetry

L6 Clause-type annihilation corresponds to DNF annihilation.

L7 For RSDNF's, a ⊙ b or b ⊙ a.

L8 σ(A □ B) = (τA) □ (τB).

L9 PRODUCT preserves symmetry

C9 PROD preserves symmetry

A ⊙ A annihilates A □ B if A[1] = 1, A ≠ B.

Adjacency is unique.
L10 If $a$ in $CT(E)$ with $a^+ + a^- < V$, then some $b \leq a$ in $CT(PROD E)$.

L11 $a$ in $(CT(PROD E)) - CT(E)$ implies $a = b \circ c$.

L12 $a \circ b$ in $PROD E$ if $a^+ + b^- \leq V + 1$. 
In order to avoid concern with "locally optimal form", which complicates analysis of the algorithm, I wish to investigate DNF's A such that every permutation of A is a locally optimal form of A. A simple class satisfying that condition is the class of symmetric DNF's, as defined below (an extension of the class will be discussed later). In terms of propositional formulae, a formula is symmetric iff it is invariant under interchanging names of variables. This is my own definition, but I presume it to be the one Mr. Cook had in mind in his Example 2.

**Definition:** E a DNF, E is **symmetric** iff for all \( \sigma \), \( \sigma E = E \).

Note that local optimization has no effect on symmetric formulae.

**Definition:** RSDNF stands for "reduced symmetric DNF", i.e. a DNF which is both symmetric and reduced.
Example of a symmetric DNF formula

<table>
<thead>
<tr>
<th>Formal Logic Notation</th>
<th>Algebraic DNF Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x₁ &amp; x₂)</td>
<td>++11</td>
</tr>
<tr>
<td>v (x₁ &amp; x₃)</td>
<td>+1+1</td>
</tr>
<tr>
<td>v (x₁ &amp; x₄)</td>
<td>+11+</td>
</tr>
<tr>
<td>v (x₂ &amp; x₃)</td>
<td>l++1</td>
</tr>
<tr>
<td>v (x₂ &amp; x₄)</td>
<td>l1+1</td>
</tr>
<tr>
<td>v (x₃ &amp; x₄)</td>
<td>l1++</td>
</tr>
<tr>
<td>v (x₁ &amp; x₂ &amp; x₃)</td>
<td>---+1</td>
</tr>
<tr>
<td>v (x₁ &amp; x₂ &amp; x₄)</td>
<td>+1+</td>
</tr>
<tr>
<td>v (x₁ &amp; x₃ &amp; x₄)</td>
<td>--1+</td>
</tr>
<tr>
<td>v (x₂ &amp; x₃ &amp; x₄)</td>
<td>++1+</td>
</tr>
<tr>
<td>v (x₁ &amp; x₂ &amp; x₃ &amp; x₄)</td>
<td>++-+1</td>
</tr>
<tr>
<td>v (x₂ &amp; x₃ &amp; x₄ &amp; x₄)</td>
<td>+++1</td>
</tr>
</tbody>
</table>


Example of a symmetric DNF formula, continued

\( \sigma A, \sigma \) given by

\[
\begin{array}{c}
\sigma(i) \\
1234 \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Formal Logic Notation</th>
<th>Algebraic DNF Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>((x_2 &amp; x_3))</td>
<td>1++1</td>
</tr>
<tr>
<td>(v(x_1 &amp; x_2))</td>
<td>++11</td>
</tr>
<tr>
<td>(v(x_2 &amp; x_4))</td>
<td>1+1+</td>
</tr>
<tr>
<td>(v(x_1 &amp; x_3))</td>
<td>+1+1</td>
</tr>
<tr>
<td>(v(x_3 &amp; x_4))</td>
<td>1l++</td>
</tr>
<tr>
<td>(v(x_1 &amp; x_4))</td>
<td>+1l+</td>
</tr>
<tr>
<td>(v(x_1 &amp; x_2 &amp; x_3))</td>
<td>+++1</td>
</tr>
<tr>
<td>(v(x_2 &amp; x_3 &amp; x_4))</td>
<td>l+++</td>
</tr>
<tr>
<td>(v(x_1 &amp; x_2 &amp; x_4))</td>
<td>++1+</td>
</tr>
<tr>
<td>(v(x_1 &amp; x_2 &amp; x_3))</td>
<td>+1++</td>
</tr>
<tr>
<td>(v(x_1 &amp; x_3 &amp; x_4))</td>
<td>+1++</td>
</tr>
<tr>
<td>(v(x_1 &amp; x_2 &amp; x_3))</td>
<td>++1+</td>
</tr>
<tr>
<td>(v(x_1 &amp; x_2 &amp; x_4))</td>
<td>+1++</td>
</tr>
<tr>
<td>(v(x_2 &amp; x_3 &amp; x_4))</td>
<td>l+++</td>
</tr>
<tr>
<td>(v(x_1 &amp; x_2 &amp; x_3))</td>
<td>++1+</td>
</tr>
<tr>
<td>(v(x_1 &amp; x_2 &amp; x_4))</td>
<td>+1++</td>
</tr>
</tbody>
</table>
Example of a symmetric DNF formula, continued

\( \tau A, \tau \) given by \( \begin{array}{c} i \\ \tau(i) \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{bmatrix} \)

<table>
<thead>
<tr>
<th>Formal Logic Notation</th>
<th>Algebraic DNF Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>((x_3 &amp; x_4))</td>
<td>(11++)</td>
</tr>
<tr>
<td>((x_1 &amp; x_3))</td>
<td>(+1+1)</td>
</tr>
<tr>
<td>((x_2 &amp; x_3))</td>
<td>(1++1)</td>
</tr>
<tr>
<td>((x_1 &amp; x_4))</td>
<td>(+1+)</td>
</tr>
<tr>
<td>((x_2 &amp; x_4))</td>
<td>(++1)</td>
</tr>
<tr>
<td>((x_1 &amp; x_2))</td>
<td>(+1-)</td>
</tr>
<tr>
<td>((x_1 &amp; x_3 &amp; x_4))</td>
<td>(+1++)</td>
</tr>
<tr>
<td>((x_2 &amp; x_3 &amp; x_4))</td>
<td>(+++-)</td>
</tr>
<tr>
<td>((x_1 &amp; x_2 &amp; x_3))</td>
<td>(+++)</td>
</tr>
<tr>
<td>((x_1 &amp; x_3 &amp; x_4))</td>
<td>(+l-)</td>
</tr>
<tr>
<td>((x_2 &amp; x_3 &amp; x_4))</td>
<td>(+l++)</td>
</tr>
<tr>
<td>((x_1 &amp; x_2 &amp; x_3))</td>
<td>(+l-)</td>
</tr>
<tr>
<td>((x_2 &amp; x_3 &amp; x_4))</td>
<td>(+l++)</td>
</tr>
<tr>
<td>((x_1 &amp; x_2 &amp; x_3))</td>
<td>(+++)</td>
</tr>
<tr>
<td>((x_1 &amp; x_2 &amp; x_3))</td>
<td>(+l+)</td>
</tr>
<tr>
<td>((x_2 &amp; x_3 &amp; x_4))</td>
<td>(+l+)</td>
</tr>
<tr>
<td>((x_1 &amp; x_2 &amp; x_3))</td>
<td>(+l+)</td>
</tr>
</tbody>
</table>

Notice that \( A \) is not reduced (every 3-literal clause is annihilated by some 2-literal clause). Of course, \( \sigma A \) and \( \tau A \) are not reduced either.

**Convention:** Hereinafter \( \sigma \) denotes an arbitrary permutation, and \( E \) and \( F \) will in general denote RSDNF's (exceptions will be noted or evident from context). Also, \( A, B, R, S, \) and \( T \)
will be used to denote clauses.

A useful characterization of symmetry is

**Lemma 4:** A DNF E is symmetrical iff for every clause R of E and every permutation \( \sigma \), \( \sigma R \) is a clause of E.

**Proof:** \((\Rightarrow)\) If E is symmetrical, and \( \sigma \) is a permutation, then \( \sigma E = E \); by definition, if R is a clause of E, \( \sigma R \) is a clause of \( \sigma E \); hence since \( \sigma E = E \), \( \sigma R \) is a clause of E.

\((\Leftarrow)\) If R a clause of E implies \( \sigma R \) a clause of E, then \( \sigma E \subseteq E \); since permutations are bijective, \( \sigma E = E \).

The remainder of this paper is devoted to a time bound on Davis-Putnam analysis of symmetrical formulae. To this end, we are particularly interested in finding out what general statements can be made about \( \text{PROD}(E) \) for an arbitrary RSDNF E. The first thing we would like to know is that \( \text{PROD}(E) \) is also an RSDNF. In showing this, it will be convenient to have some special notation.

**Definition:** For R a clause, \( R^+ \) denotes the number of +'s in R, and \( R^- \), the number of minuses. We call the ordered pair \((R^+, R^-)\) a **clause-type**, or **type** for short. We also make the conventions that for any clause R, \( r = \) the type of \( R \); \( r = (r^+, r^-) \); if \( r \) is a clause-type, \( R \) is any clause of type \( r \). We say "\( r \) is a type of E" iff E contains a clause of type \( r \).

A little reflection shows that "all permutations of R" is identical to "all clauses of type \( r \)" (precisely
because permutations are bijections). Thus an RSDNF is a reduced DNF E such that if \( r \) is a type of E, every clause of type \( r \) is a clause of E. When we know E is an RSDNF, "\( r \) is a type of E" might as well be defined to mean "every \( R \) of type \( r \) is a clause of E". We make the

**Definition:** \( \text{CT}(E) = \{ r \mid r \text{ is a clause-type of } E \} \).

Thus an RSDNF E is completely characterized by \( \text{VARS}(E) \) and \( \text{CT}(E) \).

Note that if \( r \) is a type of E, then \( r^+ + r^- \leq \text{VARS}(E) \) because R cannot have more literals than E has variables.

Now what is the relation between \( \text{CT}(E) \) and \( \text{CT}(\text{REDUCTION}(E)) \)? Suppose E has 10 variables, and suppose E has the clause-types (2,5), (2,7), (3,6), and (4,3).

Then among the clauses of E are

1. +++---1lll
2. +++-----1
3. +++-----1+
4. +++----1ll++

It is clear that 1 annihilates 2 and 3, and hence 2 and 3 will not be in the reduced form of E. Thus if REDUCTION preserves symmetry, (2,7) and (3,6) must not be clause-types of the reduced form of E.

On the other hand, 1 and 4 cannot be rearranged so that either annihilates the other; hence (2,5) and (4,3) should be clause-types of the reduced form of E, assuming these are all the types of E.
**Lemma 5:** REDUCTION preserves symmetry.

To show that if E is symmetric and F is the reduced form of E, then F is symmetric, i.e. by Lemma 4, if C is a clause of F then every permutation of C is a clause of F.

**Proof:** We shall prove the contrapositive: if C is a clause of E but not a clause of F, then no permutation of C is a clause of F. But this is trivial: if C is in E but not in F, then for some D in E, D > C; but then σD > σC for any σ (proof: for each i, D[i] > C[i], so D[σi] > C[σi], so by definition of σ, (σD)[i] = (σC)[i], whence σD > σC). Since σD is a clause of F, σC is annihilated by σD, and hence σC is not a clause of F.

Q.E.D.

Looking at the example above, it seems plausible to make the

**Definition:** For any types a, b a annihilates b iff a ≤ b (i.e. a⁺ ≤ b⁺, a⁻ ≤ b⁻).

**Lemma 6:** For any types a and b such that a annihilates b, and any clauses A and B (of types a and b respectively, by convention), there is a permutation σ(not necessarily unique) such that σA annihilates B.

**Proof:** Consider for example a = (4,5), b = (5,9), VARS(E) = 16.

Let A = (+1+1---1-1-lll)

Let B = (l+---------+l-+++)

Then σA = (l+------ll+ll+l+l), if σ is given
by $\sigma(i)$ 2 1 10 13 8 3 4 5 9 6 11 15 7 14 12 16 ,
and by inspection, $\sigma A$ annihilates B. Intuitively, we
"move" the +'s and -'s of A "under" the +'s and -'s of B,
with some 1's left over.

More formally, we see that since $A^+ \leq B^+$ we can find
$\sigma$ so that $(\sigma A)[i] = +$ only if $B[i] = +$, and because $A^- \leq B^-$
we can further restrict $\sigma$ so that $(\sigma A)[i] = -$ only if
$B[i] = -$. Since $A[i]$ and $B[i]$ are either +,-, or 1
(not 0, by convention), $B[i] \leq (\sigma A)[i]$, i.e. $B \leq \sigma A$,
i.e. $\sigma A$ annihilates B.

Q.E.D.

Note: $\leq$ has the opposite meaning for types as for clauses:
$A \leq B$ implies $b \leq a$. Thus an RSDNF is a symmetric DNF $E$
such that if $a,b$ are types of $E$, $a$ does not annihilate $b$.
Since $a$ annihilates $b$ iff $a \leq b$, "$a$ does not annihilate $b"$
means either $a^+ < b^+$ and $a^- > b^-$, or $a^+ > b^+$ and $a^- < b^-$.
It will be convenient to have a notation for this.
Definition: $a \bowtie b$ iff $a^+ > b^+$ and $a^- < b^-$. Similarly
$a \bowtie b$ means "$a \bowtie b$ or $a = b$", $a \bowtie b$ iff $b \bowtie a$. It is clear
that $\bowtie$ is a strict order relation. We can restate the
characterization of reduced for a symmetric DNF as

Lemma 7: A DNF $E$ is an RSDNF iff $E$ is symmetric and for
every pair of types $a \neq b$ of CT($E$), either $a \bowtie b$ or $b \bowtie a$.

Proof: Immediate from definitions and Lemma 6.

Q.E.D.

If $E$ is an RSDNF, then $\bowtie$ totally orders the clause-types of
of $E$, and $a \odot b$ iff $a^+ \odot b^+$. If $a \odot b$ for all $b$ of $CT(E)$, then we call $a$ the "$\odot$ - largest" type of $E$ (or of $CT(E)$). When we write down the types of $E$, we will write them in $\odot$ - decreasing order from left to right.

Next we wish to show that $PRODUCT$ preserves symmetry. The major part of the proof is contained in

**Lemma 8:** If $A$, $B$ are clauses of a symmetric DNF $E$, and $\sigma$ is a permutation of $PRODUCT(E)$, then

$$\sigma(A \Box B) = (\tau A) \Box (\tau B)$$

for a certain permutation $\tau$ of $E$. (The symmetry of $E$ is needed only to ensure that $\tau A$ and $\tau B$ are clauses of $E$.)

**Comment:** Intuitively this is obvious: $\sigma(A \Box B)$ is the product of $A$ and $B$ scrambled up, and $\tau A \Box \tau B$ is the product of $(A$ scrambled up) and $(B$ scrambled up); it shouldn't make any difference when you do the scrambling. That is, the same element-by-element operations will be done; they are just done in different columns.

**Proof:** Let $\tau(1) = 1$, $\tau(i) = \sigma(i-1)+1$, $2 \leq i \leq \text{VARS}(E)-1$. We show that for each $i$, $1 \leq i \leq \text{VARS}(E)-1$,

$$(\sigma(A \Box B))[i] = ((\tau A) \Box (\tau B))[i]:$$

$$(\sigma(A \Box B))[i] = (A \Box B)[\sigma i], \text{by definition of } \sigma;$$

$$= A[(\sigma i)+1] \& B[(\sigma i)+1] \&(A[1] \lor B[1]), \text{by definition of } \Box;$$

$$= A[\tau(i+1)] \& B[\tau(i+1)] \&(A[1] \lor B[1]),$$

because $\tau(i+1) = \sigma((i+1)-1)+1 = \sigma(i+1)$
\[= (\tau A)[i+1] \& (\tau B)[i+1] \& (A[l] \lor B[l]),\]

by definition of \(\tau;\)

\[= ((\tau A) \exists (\tau B))[i], \text{ by definition of } \exists;\]

Hence \(\sigma(A \exists B) = (\tau A) \exists (\tau B), \text{ as claimed.}\)

Q.E.D.

Note: this proof can be carried out equally simply in the propositional logic notation. We have \(v_{\sigma i}\) is a literal of \(\sigma A\) iff \(v_i\) is a literal of \(A;\) in logical notation the variables of \(\text{PRODUCT}(E)\) are \(v_2 \ldots v_n,\) so \(\sigma\) operates only on \(2 \ldots n,\) and \(\tau(1) = 1, \tau(i) = \sigma(i), i > 1.\)

Lemma 9: \(\text{PRODUCT}\) preserves symmetry, i.e. if \(E\) is symmetric then \(\text{PRODUCT}(E)\) is symmetric.

Proof: We wish to show that for any permutation \(\sigma\) of \(\text{PRODUCT}(E),\) and any clause \(R\) of \(\text{PRODUCT}(E),\) \(\sigma R\) is a clause of \(\text{PRODUCT}(E).\)

For some \(A\) and \(B\) in \(E, R = A \exists B.\) Hence

\[\sigma R = \sigma(A \exists B)\]

\[= (\tau A) \exists (\tau B), \text{ by Lemma 8}.\]

Because \(E\) is symmetric and \(\tau\) is a permutation, \(\tau A\) and \(\tau B\) are clauses of \(E\) (by Lemma 4), so their product \(\sigma R\) is a clause of \(\text{PRODUCT}(E).\)

Q.E.D.

Corollary 9: \(\text{PROD}\) preserves symmetry.

Proof: \(\text{PRODUCT} \text{ and } \text{REDUCTION} \text{ do.}\)

Since \(\text{PROD}\) preserves symmetry, and since every permu-
tation of a symmetrical formula yields the same formula, the step "A + locally optimal form of A" in DPDNF is superfluous. If we also ignore halting conditions, we see that iterations of DPDNF correspond simply to successive applications of PROD: PROD E, PROD(PROD E), PROD(PROD(PROD E)), ...

Using the usual functional iteration notation, we make the

**Definition:** PROD^0 E = E; for n ≥ 1,

PROD^n E = PROD(PROD^{n-1} E).

This notation will be useful later, after we have made a closer examination of the action of a single application of PROD to an arbitrary RSDNF.

Now, what is the action of PRODUCT on the set of clause-types? We don't care about A ⊕ A; we know what that type is. We also don't care about A ⊕ B if A ⊕ B contains 0's, or if A ⊕ A annihilates A ⊕ B.

**Remark:** A ⊕ A annihilates A ⊕ B if A[1] = 1 and A ≠ B.

**Proof:** If A ⊕ B ≠ 0, then

(A ⊕ B)[i] = (A&B)[i+1]

≤ A[i+1] ; since

A ≠ B, the inequality is strict for some i, and A ⊕ A annihilates A ⊕ B.

Q.E.D.

A ⊗ B must have all of the literals of A and B, i.e. depending on the relative positions of literals in A and B, A ⊗ B might be of any clause-type from \((a^+ - 1) + b^+, a^- + (b^- - 1))\) (subtract one because the first variable is eliminated, and \(A[1] = +\) and \(B[1] = -\)) to \((a^+ - 1, b^- - 1)\). For example, if \(A^+ = 5, A^- = 4, B^+ = 3,\) and \(B^- = 5\), (that is, the type of A is \((5, 4)\) and of B, \((3, 5)\)), then PRODUCT(E) will include these:

\[
\begin{align*}
B & \quad +++-1------1111111 \\
A & \quad +11111111111 \\
B \oplus A & \quad ++-1---------11111 \\
\text{or} & \\
B & \quad +++-----11-1111111 \\
A & \quad +++++-l++1+1l11111 \\
B \oplus A & \quad ++++++++1111111 \\
\text{or} & \\
A & \quad +++-----1ll+111 \\
B & \quad -lll---l1l1111111 \\
A \oplus B & \quad +++++-1llllllll11 \\
\text{or} & \\
A & \quad +++-----llllllll111 \\
B & \quad +++++-l1111ll11 \\
A \oplus B & \quad +++++-llllllllll1111 \\
\text{of type} & \quad (6, 6) \\
\text{of type} & \quad (5, 5) \\
\text{of type} & \quad (6, 7) \\
\text{of type} & \quad (4, 4)
\end{align*}
\]

By the annihilation rule, when we reduce PRODUCT E to get PROD E, only \((4, 4)\) can remain. Suppose in addition to a and b we have another type c in E, \(c=(2, 6)\). What can A ⊗ C yield? Arranging them for the smallest type product, we get

\[
\begin{align*}
A & \quad +++-----llllll1 \\
C & \quad ++lll-----ll \\
A \oplus C & \quad +++-----ll1llllll \\
\text{of type} & \quad (4, 5). \text{ Since} \\
(4, 4) & \text{is a type of PRODUCT E, A ⊗ C cannot be a type of PROD(E). We now formalize, generalize, and prove these}
\end{align*}
\]
observations.

**Definition:** a is adjacent (in E) to b iff a is the immediate successor to b in the $\triangleright$-ordering, i.e. $a \triangleright b$ and for all c in CT(E), not $(a \triangleright c \triangleright b)$. "a is adjacent to b" is denoted $a \text{ adj } b$.

**Remark:** Adjacency is unique, i.e. a $\text{ adj } b$ and a $\text{ adj } c$ implies $b = c$; and d $\text{ adj } f$ and e $\text{ adj } f$ implies d = e.

**Remark:** For any types a, b of E an RSDNF, a $\triangleright b$ iff $a^+ > b^+$ iff $a^- < b^-$; and a $\text{ adj } b$ iff ($a^+ > b^+$ and for no type $r$ of E is $a^+ > r^+ > b^+$) iff ($a^+ > b^+$ and for no clause $R$ of E is $a^+ > R^+ > b^+$)(and likewise iff $a^- < b^-$ and ...).

**Lemma 10:** If a is a type of E with $a^+ + a^- \leq \text{ VARS(E)} - 1$, then for some type $b$ of PROD E, $b \leq a$ (either a is annihilated or a is in CT(PROD E)).

**Proof:** Since $a^+ + a^- \leq \text{ VARS(E)} - 1$, there is some A in E of type a with $A[1] = 1$. For this A, $A \cdot A \in \text{ PRODUCT(E)}$ and has no 0's, so either $A$ $\in$ PROD E OR A is annihilated by some $B$ $\in$ PROD E. If $A$ $\in$ PROD E let $b = a$; if A is annihilated by B, let $b = (B^+, B^-)$. In either case $b \leq a$.

Q.E.D.

**Notation:** A, B sets; $A - B$ means the set difference of A and B, $A \cap (B^C)$. The motivation for the next definition is Lemma 7 below.

**Definition:** If $a \triangleright b$ then $a \text{ \P b} = (a^+ - 1, b^- - 1)$.
The purpose of this lemma is to exhibit a strong limitation on the clause-types of PROD(E).

Lemma 11: If \( a \) is a clause-type of PROD(E) but not of E, then \( a = (b^{+1}, c^{-1}) \), with \( b \) adj \( c \) in E (using the notation defined above, \( a = b \# c \) with \( b \) adj \( c \) in E).

Proof: If \( a \) is a clause-type of PROD E but not of E, then \( a \) is the type of some clause \( A \) in PROD E but not in E; by the Remark on page 96, \( A = B \# C \) with \( B[1] = +, C[1] = - \) (in logical notation, \( x_1 \) is in \( B \), \( \bar{x}_1 \) is in \( C \)). Throughout the remainder of this proof, remember that \( B[1] = +, C[1] = - \).

The proof now proceeds in four segments; the first will imply the second, and the first two imply the last two.

(1) If \( b \bowtie c \) then \((B^{+1}, C^{-1})\) is a clause-type of PRODUCT(E).

(2) If \( b \unlhd c \) and \( B \# C \) does not have type \((B^{+1}, C^{-1})\) then \( B \# C \) is annihilated.

(3) If \( c \uplus b \), then \( B \# C \) is annihilated.

(4) If \( b \uplus d \uplus c \) then \( B \# C \) is annihilated.

--PROOFS--

(1) Suppose \( b \bowtie c \); we wish to show that we can find \( X \) and \( Y \) of types \( b \) and \( c \) respectively such that the type of \( X \# Y \) is \((B^{+1}, C^{-1})\).

By assumption \( B \# C \) is a clause in PROD E, so it has no 0. Hence there is no \( i \neq 1 \) with \( B[i] = + \) and
C[i] = - or vice versa.

Now by symmetry we can "move" the +'s, -'s, and l's around and still get clauses of E. Hence we can move all C's +'s to where B has +'s, and all of B's -'s to where C has -'s, still having B[l] = +, C[l] = -.

This gives B ⊙ C of type \(B^+-1, C^-1\) (because B[l], C[l] are lost).

More formally: By symmetry, every permutation of C is a clause of E. Since \(b \bowtie c\), \(B^+ > C^+\), so we can find Y of type c such that: \(Y[i] = +\) only if \(i \neq 1\) and \(B[i] = +\); and \(Y[i] = -\) iff \(C[i] = -\). We note that we never have \(B[i] = +\), \(Y[i] = -\) for \(i \neq 1\). Similarly, every permutation of B is a clause of E. Since \(b \bowtie c\), \(B^- < C^-\), so we can find X of type b such that \(X[i] = -\) only if \(i \neq 1\) and \(C[i] = -\), and \(X[i] = +\) iff \(B[i] = -\).

Now \((X \bowtie Y)^+ = B^+-1\) (because \((X \bowtie Y)[i] = +\) iff \(X[i+1] = +\) iff \(B[i+1] = +\), and similarly \((X \bowtie Y)^- = C^-1\). Since X and Y are clauses of E, \(X \bowtie Y\) is the desired clause of PRODUCT(E) of type \((B^+-1, C^-1)\).

For example, if we have

<table>
<thead>
<tr>
<th>CLAUSE NAME</th>
<th>CLAUSE</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>++++-1-+-1ll+</td>
<td>(6,4)</td>
</tr>
<tr>
<td>C</td>
<td>-++ll-1-1+++l</td>
<td>(4,5)</td>
</tr>
<tr>
<td>B ⊙ C</td>
<td>++++++++----+++</td>
<td>(7,6)</td>
</tr>
<tr>
<td>CLAUSE NAME</td>
<td>CLAUSE</td>
<td>TYPE</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>++++-l-+-1l1+</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>-++11l-+-l+-l+</td>
</tr>
<tr>
<td></td>
<td>B ⊗ Y</td>
<td>++++-++l-+-+l+</td>
</tr>
</tbody>
</table>

and the only choice for X,

<table>
<thead>
<tr>
<th>CLAUSE</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>++++l-+-l-+-+l+</td>
</tr>
<tr>
<td>Y</td>
<td>-++11l-+-l+-l+</td>
</tr>
<tr>
<td>X ⊗ Y</td>
<td>+++l-+-l-+-+l+</td>
</tr>
</tbody>
</table>

(2) We wish to show that if \( b \odot c \) and \( A = B \odot C \) does not have type \((B^+ - 1, C^- - 1)\), then \( A \) is annihilated.

We know that \( A[i] = + \) if \( B[i+1] = + \) or \( C[i+1] = + \), and \( A[i] = - \) if \( B[i+1] = - \) or \( C[i+1] = - \).

Hence

\[
A^+ \geq B^+ - 1 \quad \text{(subtract 1 because B} \leftarrow [1] = +, \text{ and)}
\]

\[
A^+ \geq C^+ \quad \text{implying } i+1 \neq 1
\]

\[
A^- \geq B^- \quad \text{and}
\]

\[
A^- \geq C^- - 1
\]

Since \( b \odot c \), \( B^+ - 1 \geq C^+ \) and \( C^- - 1 \geq B^- \), so the four inequalities reduce to two:

\[
A^+ \geq B^+ - 1
\]

\[
A^- \geq C^- - 1
\]

But by part (1), \((B^+ - 1, C^- - 1)\) is a type of PRODUCT(E).

We have \((A^+, A^-) \geq (B^+ - 1, C^- - 1)\), and we have assumed \((A^+, A^-) \neq (B^+ - 1, C^- - 1)\), so by Lemma 6, \((A^+, A^-)\) is not a type of PROD(E), and \( A \) is annihilated.

We now know that \( a \) is a type of PROD(E) but not E only if \( a = b \odot c \) for some \( b, c \) in CT(E), so parts
(3) and (4) can be restated and proved as (3a) and (4a), below.

(3a) If \(c \geq b\), then \(b \circ c\) is annihilated.

The set of four inequalities in Part (2) did not use, \(b \geq c\), so we have again, if \(A = B \circ C\),

\[
\begin{align*}
A^+ & \geq B^+ - 1 \\
A^+ & \geq C^+ \\
A^- & \geq B^- \\
A^- & \geq C^- - 1
\end{align*}
\]

But now since \(c \geq b\), \(C^+ > B^+ - 1\) and \(B^- > C^- - 1\), so

\[
(A^+, A^-) \geq (C^+, B^-)
\]

\[
\geq (B^+, B^-) \quad \text{(and} \geq (C^+, C^-) \text{for that matter)},
\]

So a is annihilated by b or c, if either is a type of PRODUCT(E).

Now, if \(b^+ + b^- \leq \text{VARS}(E) - 1\), then there is a clause \(\sigma B\) of E with \((\sigma B)[1] = 1\); thus \(\sigma B\) is a clause of PRODUCT(E), and b is a type of PRODUCT(E). In other words, b will fail to be a type of PRODUCT(E) only if \(b^+ + b^- = \text{VARS}(E)\); if this happen then \(A^+ + A^- \geq \text{VARS}(E)\) also (because \((A^+, A^-) \geq (B^+, B^-))\), so a cannot be a type of PRODUCT(E). In either event, we see that if \(c \geq b\), then \(b \circ c\) is annihilated.

(4a) We wish to show that if \(b \geq d \geq c\), then \(b \circ c\) is annihilated.
But $b \circ d = (b^+, d^- - 1)$;

$b \circ c = (b^+, c^- - 1)$;

d $\circ c$ implies $c^- > d^-$, so

$b \circ c \geq b \circ d$ and not equal; in other words

$b \circ c$ is annihilated.

We now have that $b \circ c$ is a type of PROD(E) only
if $b \adj c$, as desired.

Q.E.D.

Lemma 12: If $a \adj b$ in $E$, with $a^+ + b^- \leq \text{VARS}(E) + 1$, then
$(a^+ - 1, b^- - 1)$ is a clause-type of PROD(E).

Proof: Since $a \circ b$ with $a^+ + b^- \leq \text{VARS}(E) + 1$, we can find
PRODUCT(E) and of type $(a^+ - 1, b^- - 1)$. We need to show that
$(a^+ - 1, b^- - 1)$ is not annihilated, i.e. for any clause-type
c of PROD(E) it is false that $c \leq a \circ b$.

But by Lemma 11, if c is a type of PROD(E), either c
is a type of E or $c = (d^+ - 1, f^- - 1)$ for some types $d \adj f$
in $E$.

Case 1: $c$ is a type of $E$.

Because $E$ is an RSDNF and $a \adj b$, either $c \geq a \circ b$ or
$a \circ b \geq c$. If $c \geq a$ then by definition $c^+ > a^+$. Hence
"$c^+ \leq a^+ - 1$" is false, so "$(c^+, c^-) \leq (a^+ - 1, b^- - 1)$" is
false, i.e. $c \not\leq (a^+ - 1, b^- - 1)$. Similarly if $b \geq c$ then
$c^- \not<_b - 1$, so again $c \not<_b (a^+ - 1, b^- - 1)$. 
Diagrammatically, we have either:

\[
\begin{align*}
\text{c} \supset a \supset b \\
\text{c} \not\supset a \supset b
\end{align*}
\]

; for example

\[
\begin{align*}
(12,8) \supset (12,8) \supset (11,9) \\
(12,8) \not\supset (11,8)
\end{align*}
\]

--or--

\[
\begin{align*}
\text{a} \supset b \supset c \\
\text{a} \not\supset b \not\supset c
\end{align*}
\]

; for example

\[
\begin{align*}
(12,8) \supset (11,9) \supset (10,10) \\
(11,8) \not\supset (10,10)
\end{align*}
\]

In sum, no type of E annihilates \((a^+-1, b^-1)\)

**Case 2:** \(c = d \supset f\)

Either \(d \supset f \supset a \supset b\)

or \(a \supset b \supset d \supset f\).

In the first case, \(d \supset a\), so \(d^+ > a^+\) and

\[c^+ = d^+ - 1 > a^+ - 1;\]

so \(c \not\leq (a^+-1, b^-1)\).

In the other case, \(b \supset f\), so \(c^- > b^-1\), and again

\[c \not\leq (a^+-1, b^-1)\].

Diagrammatically:

\[
\begin{align*}
d \supset f \supset a \supset b \\
d \not\supset f \not\supset a \not\supset b
\end{align*}
\]

; for example

\[
\begin{align*}
(15,4) \supset (14,8) \supset (12,9) \supset (6,10) \\
(14,7) \supset (11,9)
\end{align*}
\]

--or--

\[
\begin{align*}
a \supset b \supset d \supset f \\
a \not\supset b \not\supset d \not\supset f
\end{align*}
\]

\[
\begin{align*}
(8,6) \supset (6,8) \supset (5,9) \\
(7,7) \supset (5,8)
\end{align*}
\]

Thus no type of PRODUCT(E) annihilates \(a \supset b\).

Q.E.D.

Lemmas 11 and 12 together say that a is a clause-type
of \text{PROD}(E) \text{ but not of } E \text{ iff } a = b \oplus c, \text{ } b \text{ adj } c, \text{ } b^+c^- \preceq \text{VARS}(E)+1.
Chapter 4

CT(PROD\textsuperscript{n}E); a polynomial bound for DPDNF on RSDNF's.

Definitions:

SEMIPROD(E)

(m)

q

CMAX\textsubscript{n}(E)

C\textsubscript{n}(E)

Proofs:

L13 \quad |CT(E)| \leq V+1

L14 \quad a \text{ adj } b \text{ in PROD}(E) \text{ implies }
\quad a^+ b^- \geq VARS(E)+1
\quad \text{or } a^+ = b^++1
\quad \text{or } b^- = a^-+1

L15 \quad (\forall) (a^+a^-)
\quad \frac{a^+!}{a^+!} \frac{a^-!}{a^-!} \text{ clauses of type a}

L16 \quad (N-1)(a^+a^-)
\quad \frac{a^+!}{a^+!} \frac{a^-!}{a^-!} \leq \frac{(a^+a^-+1)}{a^+! (a^-+1)!}

L17 \quad CMAX\textsubscript{n}(E) \leq CMAX\textsubscript{n-1}(E), n \geq 2

T1 \quad |\text{PROD}\textsuperscript{n}E| \leq V \cdot |\text{PROD} E|

T2 \quad |\text{PROD} E| \leq |E|^2

Time for E \leq 9 V^4|E|^4
We can now shift our attention entirely from the RSDNF's themselves to their associated sets of clause-types. An important result is that there are not very many clause-types.

Lemma 13: In any RSDNF E, there are no more than VARS(E)+1 clause-types. (It is possible to show a restriction which bounds the number of types in PROD^nE in terms of the number of types in E: the number of types in PROD^nE is no more than (N+1) times the number of types in E; but in the worst case this is actually weaker than Lemma 13, and it requires several pages to prove.)

Proof: Since E is reduced and symmetric, each clause-type a of E has a distinct first member a⁺, with 0 ≤ a⁺ ≤ VARS(E). There can be at most VARS(E)+1 such first members, hence at most VARS(E)+1 clause-types.

Q.E.D.

This shows that PROD^nE (that is, the result of the n-th iteration of DPDNF) has at most VARS(E)-n types. In order to bound the number of clauses, we will need to know how many clauses belong to each of these types. Knowing what types there are, it is simple combinatorics to find out how many clauses there are. Thus we need to know still more specifically what the types of PROD^nE can be; this will require knowing some restrictions on the possible types of PROD^nE, based on the types of PROD^(n-1)E. We begin with an example.
Lemmas 11 and 12 show that the only clause-types of PROD(E) are either types of E or a \( \circledast \) b for a \( \text{adj} \) b in E. Let us denote by "SEMIPROD(E)" the set of all clauses in VARS(E)-1 variables whose types are in

\[ \text{CT}(E) \cup \{ a \circledast b \mid a \text{ adj} \ b \ \text{in} \ E \}; \]

usually SEMIPROD(E) properly contains PROD(E). Note that SEMIPROD is strictly an expository device; for example CT(SEMIPROD E), interpreted as CT(E) \( \cup \{ a \circledast b \mid a \text{ adj} \ b \ \text{in} \ E \} \), might contain clause-types a with \( a^+ + a^- \rightarrow \)

VARS(PRODUCT E). Recall that PROD\(_n\)E is the result of n iterations of DPDNF on the symmetric DNF E.

In the example below I have chosen a set of clause-types at random and listed in order CT(E), CT(SEMIPROD E), CT(PROD E), CT(SEMIPROD(PROD E)), CT(PROD\(^2\)E), ..., up through CT(PROD\(^9\)E). The types are listed in \( \odot \)-decreasing order from left to right. The passage from SEMIPROD(PROD\(_n\)E) to PROD\(_{n+1}\) E is by visual inspection, using the annihilation rule for clause-types.

In order to fit the example readably on the page, I have changed notation:

\[
(a^+, a^-) \text{ is here denoted} \\
\begin{array}{c}
a^+ \\
a^- \end{array}
\]

Let us say VARS(E) = 100. The types of E bear a * above them.
\[
\begin{align*}
CT(E) & \\
*60 & 50 & 35 & 32 \\
20 & 35 & 40 & 60 \\
CT(SEMIPROD E) &= CT(PROD E) \\
*60 & 59 & 50 & 49 & 35 & 34 & 32 \\
20 & 34 & 35 & 39 & 40 & 59 & 60 \\
CT(SEMIPROD(PROD E)) & \\
*60 & 59 & 58 & 50 & 49 & 48 & 35 & 34 & 33 & 32 \\
20 & 33 & 34 & 35 & 38 & 39 & 40 & 58 & 59 & 60 \\
CT(PROD^2E) &= CT(REDUCTION(SEMIPROD(PROD E))) \\
*60 & 59 & 58 & 50 & 49 & 48 & 35 & 34 & 33 & 32 \\
20 & 33 & 34 & 35 & 38 & 39 & 40 & 58 & 59 & 60 \\
CT(SEMIPROD(PROD^2E)) & \\
*60 & 59 & 58 & 50 & 49 & 48 & 35 & 34 & 33 & 32 \\
20 & 32 & 33 & 34 & 35 & 37 & 38 & 39 & 40 & 57 & 58 & 59 & 60 \\
CT(PROD^3E) &= CT(REDUCTION(SEMIPROD(PROD^2E))) \\
*60 & 59 & 58 & 50 & 49 & 48 & 35 & 34 & 33 & 32 \\
20 & 32 & 33 & 34 & 35 & 37 & 38 & 39 & 40 & 57 & 58 & 59 \\
CT(SEMIPROD(PROD^3E)) & \\
*60 & 59 & 58 & 50 & 49 & 48 & 35 & 34 & 33 & 32 \\
20 & 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 & 57 & 58 & 59 \\
CT(PROD^4E) & \\
*60 & 59 & 58 & 50 & 49 & 48 & 47 & 46 \\
20 & 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \\
\text{**}35 & 34 & 33 & 32 \\
& 40 & 56 & 57 & 58 \\
CT(SEMIPROD(PROD^4E)) & \\
*60 & 59 & 58 & 50 & 49 & 48 & 47 & 46 \\
20 & 30 & 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \\
*45 & 35 & 34 & 33 & 32 & 32 \\
\text{**}39 & 40 & 55 & 56 & 57 & 58
\end{align*}
\]
<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>59</th>
<th>58</th>
<th>57</th>
<th>56</th>
<th>55</th>
<th>49</th>
<th>48</th>
<th>47</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td>39</td>
<td>40</td>
<td>55</td>
<td>56</td>
<td>57</td>
</tr>
</tbody>
</table>

**CT(SEMIPROD(PROD^5E))**

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>59</th>
<th>58</th>
<th>57</th>
<th>56</th>
<th>55</th>
<th>54</th>
<th>49</th>
<th>48</th>
<th>47</th>
<th>46</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>45</td>
<td>44</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CT(SEMIPROD(PROD^6E))**

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>59</th>
<th>58</th>
<th>57</th>
<th>56</th>
<th>55</th>
<th>54</th>
<th>49</th>
<th>48</th>
<th>47</th>
<th>46</th>
<th>46</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>45</td>
<td>44</td>
<td>43</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CT(PROD^7E)**

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>59</th>
<th>58</th>
<th>57</th>
<th>56</th>
<th>55</th>
<th>54</th>
<th>53</th>
<th>47</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>44</td>
<td>43</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td>32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CT(PROD^8E)**

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>59</th>
<th>58</th>
<th>57</th>
<th>56</th>
<th>55</th>
<th>54</th>
<th>53</th>
<th>52</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>44</td>
<td>43</td>
<td>42</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* = types of E
Notice that in the example above, for every type \((a^+, a^-)\) in \(\text{PROD}^nE\), \(n \geq 1\), either \((a^+1, a^-)\) or \((a^+, a^-1)\) is also a type of \(\text{PROD}^nE\). Lemma 10 shows that this is essentially without exception; it is vital to the polynomial bound.

**Lemma 14:** If a \texttt{adj} \(b\) in \(\text{PROD}(E)\) then either
\[
\begin{align*}
a^+ + b^- & \geq \text{VARS}(E) + 1 \\
or & \quad a^+ = b^+ + 1 \\
or & \quad b^- = a^- + 1
\end{align*}
\]

**Proof:** We show this by cases.

(1) \(a\) and \(b\) both clause-types of \(E\).

If a \texttt{adj} \(b\) in \(E\) then \(a \texttt{adj} b\) is not in \(\text{PROD } E\) (if it were, it would be \(\bowtie\) - between \(a\) and \(b\)), so by Lemma 12 \(a^+ + b^- \geq \text{VARS}(E) + 2\).

Otherwise let a \texttt{adj} \(c\) in \(E\). Then \(a^+ + c^- \geq \text{VARS}(E) + 2\) for the same reason. But \(c \bowtie b\) implies \(b^- > c^-\), so \(a^+ + b^- \geq \text{VARS}(E) + 2\) also.

(2) \(a\) a clause-type of \(E\) and \(b\) not.

Then \(b = c \bowtie d\) with \(c \texttt{adj} d\) in \(E\). Now, either \(a = c\) or not.

If \(a = c\) then \(b = (a^+ - 1, d^- - 1)\), so \(b^+ = a^+ - 1\), i.e. \(a^+ = b^+ + 1\) as desired. (This is usually the case; in the example just given, it is always the case.)

Suppose \(a \neq c\). Then diagrammatically, we have
\[
\begin{align*}
c \bowtie d \\
\downarrow \\
a \bowtie b
\end{align*}
\]

\[
\begin{align*}
a \bowtie c \bowtie d \\
\downarrow \\
a \bowtie b
\end{align*}
\]

, hence

\[
\begin{align*}
a \bowtie c \bowtie d \\
\downarrow \\
\end{align*}
\]
In words since \( b = c \circ d \), \( b \triangleright d \). Since a \text{adj} b, 
a \triangleright b. Hence a \triangleright d. Since c \text{adj} d implies \( c \triangleright d \) and
precludes \( c \triangleright a \triangleright d \), we have \( a \triangleright c \triangleright d \) in E; in partic-
ular a \triangleright c. Let m \text{adj} c in E (so m = a or a \triangleright m).
Since m \circ c is not in PROD E, Lemma 12 says \( m^+ + c^- \geq VARS(E) + 2 \).
VARS(E)+2. But \( a^+ \geq m^+ \) and \( b^- \geq c^- \) (because \( b = c \circ d \)),
so \( a^+ + b^- \geq VARS(E) + 2 \) as desired.

For example, let V=20, then

\[
\begin{align*}
\text{CT}(E) & \quad (13,6) \quad (10,10) \quad (7,11) \quad V=20 \\
\text{CT(SEMIPROD E)} & \quad (13,6) \quad (12,9) \quad (10,10) \quad (9,10) \quad (7,11) \quad V=19 \\
\text{CT(PROD E)} & \quad (13,6) \quad (9,10) \quad (7,11) \quad V=19
\end{align*}
\]

Neither (12,9) nor (10,10) can be a type of PROD(E)
because 12+9 = 21 and 10+10=20; both are greater than
VARS(PRODUCT E) = 19.
Note that 13+10 \geq VARS(E)+1.

(3) a is not a clause-type of E but B is (This is a
"clause-type dual" to (2); the proof is identical in
form).

We have a = c \circ d with c \text{adj} d in E.
Either b = d or b \neq d.
If b = d then a = (c^+ - 1, b^- - 1), so \( a^- = b^- - 1 \), i.e.
\( b^- = a^- + 1 \) as desired.

Suppose b \neq d.

Since a = c \circ d, c \triangleright a. Since a \text{adj} b, a \triangleright b.
Hence c \triangleright b. Since c \text{adj} d implies c \triangleright d and pre-
cludes c ⊢ b ⊢ d, we must have c ⊢ d ⊢ b in E.
Thus we have a ⊢ d adj m with a adj b in PROD E, 
so d ⊢ m is not in PROD E; hence by Lemma 12, 
d⁺m⁻ ≥ VARS(E)+2. But a⁺ > d⁺; also b⁻ ≥ m⁻, so 
a⁺+b⁻ ≥ VARS(E)+2, as claimed.

(4) Neither a nor b is a clause-type of E.

Let a = w ⊢ x, b = y ⊢ z.
Either x = y or x ≠ y.

Suppose x ≠ y. Let x adj m in E (so either m = y or 
m ⊢ y). Now a ⊢ x ⊢ m ⊢ b but a adj b in PROD(E), 
so x ⊢ m is not in PROD(E); hence by Lemma 8 x⁺m⁻ 
≥ VARS(E)+2. But a⁺ ≥ x⁺ and b⁻ ≥ m⁻, so a⁺+b⁻ 
≥ VARS(E)+2 as claimed.

Suppose x = y, so 
a = w ⊢ x = (w⁺⁻,x⁻⁻) 
and b = x ⊢ z = (x⁺⁻,z⁻⁻)

Then we will find that the situation must be like 
one of the following three examples:
x = y, so we have:

\[ w \xrightarrow{x} x \xrightarrow{z} \]
\[ a \xrightarrow{b} \]

Let \( V = 20 \)

(1) \( x^+ + x^- = \text{VARS}(E) \)

\[
\begin{align*}
E & (12,6) & (11,9) & (7,10) & v = 20 \\
\text{SEMIPROD E} & (12,6) (11,8) (11,9) (10,9) (7,10) & v = 19 \\
\text{PROD E} & (12,6) (11,8) (10,9) (7,10) & v = 19
\end{align*}
\]

(2) \( x^+ + x^- \leq \text{VARS}(E) - 1 \)

\[
\begin{align*}
E & (10,3) (9,6) (8,8) \\
\text{SEMIPROD E} & (10,3) (9,5) (9,6) (8,7) (8,8) \\
\text{PROD E} & (10,3) (9,5) (8,7)
\end{align*}
\]

(3) \( x^+ + x^- \leq \text{VARS}(E) - 1 \)

\[
\begin{align*}
E & (11,3) (9,6) (8,7) \\
\text{SEMIPROD E} & (11,3) (10,5) (9,6) (8,6) (8,7) \\
\text{PROD E} & (11,3) (10,5) (8,6)
\end{align*}
\]

Now \( a \boxtimes x \boxtimes b \) but \( a \text{ adj } b \) in PROD(E), so \( x \) is not a clause-type of PROD E. Thus either \( x^+ + x^- \) > \text{VARS(PROD E)} or for some \( e \) in PROD E, \( e \leq x \) and \( e \neq x \).
If $x^+ x^- \geq \text{VARS(PROD E)}$, then

$$x^+ x^- \geq \text{VARS(PROD E)} + 1$$

$$= \text{VARS(E)}.$$ 

Moreover $x^+ x^- \leq \text{VARS(E)}$ (for any $x$ in $\text{CT(E)}$),
so $x^+ x^- = \text{VARS(E)}$.

Also $a^- = x^- - 1$.

$$b^+ = x^+ - 1$$

$$a^+ \geq x^+$$

$$b^- \geq x^-$$

Adding pairs of these we get

$$a^+ a^- \geq x^+ x^- - 1 = \text{VARS(PROD E)}$$

and $b^+ b^- \geq x^+ - 1 + x^- = \text{VARS(PROD E)}$.

Since $a$ and $b$ are clause-types of $\text{PROD E}$,

$$a^+ a^- \leq \text{VARS(PROD E)} \text{ and}$$

$$b^+ b^- \leq \text{VARS(PROD E)} \text{ so}$$

$$a^+ a^- = \text{VARS(PROD E)} = b^+ b^-.$$ 

Since $a^- = x^- - 1$, we have $a^+ = x^+ = b^+ + 1$;

since $b^+ = x^+ - 1$ we have $b^- = x^- = a^- + 1$. Thus
we have both $a^+ = b^+ + 1$ and $b^- = a^- + 1$.

If $x^+ x^- \leq \text{VARS(PROD E)}$ then there is an $e$ in $\text{PROD E}$
but not in $E$ such that $e \leq x$ and $e \neq x$.

Let $e = p \oplus q$, $p \text{ adj } q$ in $\text{CT(E)}$.

Claim either $p = w$, $q = x$, and $e = a$,

or $p = x$, $q = z$, and $e = b$.

Note that once this claim is established, we are done: we know $a^+ \geq x^+$ and $b^- \geq x^-$. 
(because \( a = w \bigcirc x \) and \( b = x \bigcirc z \)). If \( e = a \) then \( a^+ \leq x^+ \) so \( a^+ = x^+ \); since \( b^+ = x^+-1 \), \( a^+ = b^+ + 1 \) as desired. Similarly if \( e = b \) then \( b^+ = b^+ \); since \( a^- = x^-1 \), \( b^- = a^- + 1 \).

Since \( w \mathrel{adj} x \), either \( p \bigcirc w, p = w, p = x, \) or \( x \bigcirc p \). If \( p = w \) then since \( p \mathrel{adj} q, w \mathrel{adj} q \) also; but \( w \mathrel{adj} x \), so by uniqueness, \( q = x \) and \( e = p \bigcirc q = w \bigcirc x = a. \)

Similarly if \( p = x \), then \( q = z \) and \( e = p \bigcirc q = b. \)

If \( p \bigcirc w \) then \( e^+ \geq w^+ > x^+ \) so \( e^+ \) is greater than \( x^+ \), which contradicts the assumption \( e \leq x. \)

Likewise \( z \bigcirc p \) implies \( e^- \geq z^- > x^- \), which also contradicts \( e \leq x. \)

This proves the claim, and ends part (4).

For \( a \) and \( b \) being exhausted, the Lemma is proved.

Q.E.D.

It is appropriate now to calculate the number of clauses belonging to a given clause-type.

If \( E \) is an RSDNF, \( \text{VARS}(E) = V, a \) a clause-type

then the number of clauses in \( E \) of type \((a^+, a^-)\) is

\[
\frac{(V)}{(a^+ + a^-)} \text{, where} \frac{a^+ !}{a^- !}
\]
\[(m)_{q}^{m{l}} = m \cdot (m-1) \cdot \ldots \cdot (m-q+1) \]
\[
\frac{1}{(m-q)!}
\]

**Proof:** Since \(E\) is symmetric, every possible clause of length \(V\) with \(a^+\)'s and \(a^-\)'s is a clause in \(E\) of type \(a\). How many such clauses are there? There are \(V\) places to put the first \(+\) or \(-\), \((V-1)\) to put the second, \(\ldots\), \((V-a^+a^-+1)\) places to put the last, and these choices are independent, so there are \(V \cdot (V-1) \cdot \ldots \cdot (V-a^+a^-+1) = (V)_{(a^++a^-)}\) ways to pick spots for the \(+\)'s and \(-\)'s. Since \(+\)'s are indistinguishable this number must be divided by the \(a^+!\) ways to interchange \(+\)'s, and also by \(a^-!\) for the \(-\)'s, whence \((V)_{(a^++a^-)}\). (For a more careful proof, see \(\frac{a^+!}{a^-!}\).

Feller, *Introduction to Probability*, 3rd Ed., Vol. 1, Ch. II.4, Thm. 2 (p.37). Next, a simple combinatorial lemma which, in conjunction with the two previous lemmas, will enable us to bound the clauses-per-clause-type of \(\text{PROD}^n E\) in terms of the clauses-per-clause-type in \(\text{PROD}^{n-1} E\).

**Lemma 16:** If \(N \geq a^-+1\), then

\[
\frac{(N-1)}{a^+! a^-!} \leq \frac{(N)}{a^+! (a^-+1)!} \leq \frac{(N)}{a^+! (a^-+1)!}
\]

**Proof:** \(A \leq B \iff A \cdot \frac{1}{B} \leq 1\). So consider

\[
\frac{(N-1)}{a^+! a^-!} \leq \frac{a^+! (a^-+1)!}{(N)} \leq \frac{(N)}{a^+! (a^-+1)!}
\]
\[
\begin{align*}
= (N-1) \left( a^+ + a^- \right) & \quad \frac{a^+ (a^- + 1)}{a^+ a^-} \quad \frac{a^+ (a^- + 1)}{a^+ a^-} \\
& \quad \frac{a^+ (a^- + 1)}{a^+ a^-} \\
= a^- + 1 & \quad \frac{a^+ (a^- + 1)}{a^+ a^-} \\
& \quad \frac{a^+ (a^- + 1)}{a^+ a^-} \\
\leq 1 & \quad \text{by hypothesis.}
\end{align*}
\]

Q.E.D.

The next lemma is the final step in the proof of the polynomial bound for DPDNF on symmetric DNF's. 

**Definition:** Let \(\text{CMA}_n(E)\) be the greatest number of clauses of any given clause-type in \(\text{PROD}^nE\), i.e. \(\text{CMA}_n(E) = \text{MAX}\{\text{number of clauses of clause-type } a \mid a \text{ is a clause-type of } \text{PROD}^nE\}\).

**Lemma 17:** If \(n \geq 2\) then \(\text{CMA}_n(E) \leq \text{CMA}_{n-1}(E)\). (This is false for \(n = 1\)).

**Proof:** Note: throughout this proof, "less than" means "\(\leq\)".

For some clause-type \(a\) of \(\text{PROD}^{n-1}E\), \(\text{CMA}_{n-1}(E) = \text{the number of clauses of clause-type } a\) in \(\text{PROD}^{n-1}E\) (because \(\text{CT}(\text{PROD}^{n-1}E)\) is finite). By Lemma 15, for this \(a\),

\[
\text{CMA}_{n-1}(E) = \frac{(\text{VARS}(E)-n+1)(a^++a^-)}{a^+a^-}.
\]

Since this is the maximum, for every \(b\) in \(\text{CT}(\text{PROD}^{n-1}E)\),

\[
\frac{(\text{VARS}(E)-n+1)(b^++b^-)}{b^+b^-} \leq \frac{(\text{VARS}(E)-n+1)(a^++a^-)}{a^+a^-}.
\]
Let $c$ be a clause-type in $\text{PROD}^n E$ having $\text{CMAX}_n (E)$
classes. Then since $c$ is a clause-type of $\text{PROD}^n E$, either
$c$ is a clause-type of $\text{PROD}^{n-1} E$, or $c = d \uparrow e$, with $d \text{ adj}_e$
in $\text{PROD}^{n-1} E$. By Lemma 15, the number of clauses of type
$c$ in $\text{PROD}^n E$ is
\[ \frac{(\text{VARS}(E)-n)(c^+c^-)}{c^+!c^-!} \]
If $c$ is in $\text{PROD}^{n-1} E$, we note that this is clearly less
than
\[ \frac{(\text{VARS}(E)-n+1)(c^+c^-)}{c^+!c^-!} \], which is the
number of clauses of type $c$ in $\text{PROD}^{n-1} E$, which by
definition is less than $\text{CMAX}_{n-1} (E)$.
If $c = d \uparrow e$ then by definition of $\uparrow$, $c^+ = d^-1$ and
$c^- = e^-1$. Restated, $d^+ = c^+1$ and $e^- = c^-1$. Also
by Lemma 13, since $d \text{ adj}_e$ in $\text{PROD}^n E$ and $n \geq 2$, either
d$^+ = e^+1$, or $e^- = d^-1$.
In the first case we have
\[ e^+1 = d^+ = c^+1, \text{ so } e^+ = c^+ \text{ and } e = (c^+, c^-1) \]
a clause-type of $\text{PROD}^{n-1} E$. Thus
\[ \text{CMAX}_n (E) = \frac{(\text{VARS}(E)-n)(c^+c^-)}{c^+!c^-!} \leq \frac{\text{VARS}(E)-n+1)(c^+c^-+1)}{c^+; (c^-1)!} \]
by Lemma 16.
\[ = \text{ the number of clauses of type } e \text{ in } \text{PROD}^{n-1} E \]
\[ \leq \text{CMAX}_{n-1} (E) \text{ by definition of } \text{CMAX}. \]
In the second case, \( e^- = d^- + 1 \), we have \( d^- + 1 = e^- = c^- + 1 \) and \( d = (c^+ + 1, c^-) \) is a clause-type of PROD\(^{n-1}E\), so

\[
C_{\text{MAX}}_n(E) = \frac{(\text{VARS}(E) - n)(c^+ + c^-)}{c^+! c^-!}.
\]

\[
\leq \frac{(\text{VARS}(E) - n + 1)(c^+ + 1 + c^-)}{(c^+ + 1)! c^-!}
\]

by Lemma 16

= the number of clauses of type \( d \) in PROD\(^{n-1}E\)

\[
\leq C_{\text{MAX}}_{n-1}(E)
\]

by definition of CMAX

Thus for every type \( c \) in PROD\(^{n}E\), there are fewer than

\( C_{\text{MAX}}_{n-1}(E) \) clauses of type \( c \), so \( C_{\text{MAX}}_n(E) \leq C_{\text{MAX}}_{n-1}(E) \), as claimed.

Q.E.D.

Thus we see that except for the first application of PROD, the number of clauses can increase only by increasing the number of clause-types; but the number of clause-types can increase to at most VARS(E) - n at the \( n^{\text{th}} \) iteration. This in essence finishes the proof. The formalities follow:

**Definition:** \( C_n(E) = |\text{CT(PROD}^nE)| \) (i.e. the number of clause-types in PROD\(^nE\)).

**Theorem 1:** The number of clauses in PROD\(^nE\) is no more than \((\text{VARS}(E)) \cdot (\text{the number of clauses in PROD } E)\).

**Proof:** The number of clauses in PROD\(^nE\) is equal to
\[ \sum_{a \in \text{CT}(\text{PROD}^nE)} (\text{number of clauses of type } a) \]

by definition of \(\text{CMA}_n(E)\)

\[ \leq \sum_{a \in \text{CT}(\text{PROD}^nE)} \text{CMA}_n(E) \]

\[ = \text{C}_n(E) \cdot \text{CMA}_n(E) \]

by definition of \(\text{C}_n(E)\)

\[ \leq (\text{VARS}(E)+1-n) \cdot \text{CMA}_n(E) \]

by Lemma 13

\[ \leq (\text{VARS}(E)+1-n) \cdot \text{CMA}_1(E) \]

by Lemma 17

\[ \leq (\text{VARS}(E)+1-n) \cdot (\text{the number of clauses in PROD } E), \]

because \(\text{CMA} \) is the number of clauses of one type, which is surely less than the number of clauses of all types put together.

\[ \leq (\text{VARS}(E)) \cdot (\text{the number of clauses in PROD } E) \]

since \( n \geq 1 \)

Q.E.D.

Remark: For \( E \) an RSDNF, \(|\text{PROD } E| \leq |E|^2\).

Proof: Trivial by definition of PRODUCT.

Note that for \( \text{CT}(E) = \{(\text{VARS}(E)/3,0), (0,\text{VARS}(E)/3)\} \) we get \(|\text{PROD } E| \) nearly equal to \(|E|^2\); see Appendix.

Theorem 2: The time required by DPDDNF for an RSDNF \( E \) is no more than \( 9 \cdot (\text{VARS}(E))^4 |E|^4 \), assuming element operations to take unit time.

Proof: Let \( V = \text{VARS}(E) \).

The procedure terminates in no more than \( V \) steps, so \( n \leq V \), below.

The \( n^{\text{th}} \) application of PRODUCT takes \( (|\text{PROD}^nE|)^2 \) applications of \( \oplus \); in this iteration, each application of
(3) takes \(2(V-n+1)\) & operations and one \(\bigsqcup\) operation.

Take upper bounds: By Theorem 1 and the remark above, 
\[|\text{PROD}^n E| \leq V|E|^2,\] so the \(n\textsuperscript{th}\) application of \text{PRODUCT} takes no more than \((V|E|^2)^2 = V^2|E|^4\) applications of (3), and an application of (3) takes no more than \(2V+3\) operations, so each application of \text{PRODUCT} takes no more than \((2V+3) \cdot V^2|E|^4 \leq 3 \cdot V \cdot V^2 |E|^4\) operations, for \(V \geq 3\).

Each application of the reduction procedure takes no more than twice as long as each application of \text{PRODUCT} (you do as many element operations, and then possibly that many transfers as well), so each application of \text{PROD} takes no more than \(9 V^3 |E|^4\). There are at most \(V\) iterations, so the whole procedure terminates in
\[
V \cdot 9 \cdot V^3 |E|^4 = 9 V^4 |E|^4 \text{ operations.}
\]

Q.E.D.

The constant 9 could be reduced by using tighter bounds, but not by very much; and it is the degree of the polynomial that is of interest; using the best bounds I have derived here will not improve this degree; neither will the obvious simplifications of \text{PROD} (e.g. omit \text{R}(\text{PROD}) S unless: \(R=S\) and \(R(1)=1\); or \(R(1)=+\), \(S(1)=-\)). An improvement on the remark that \(|\text{PROD} E| \leq |E|^2\) would improve the degree. There is an example and rough estimate in the Appendix bearing on this and on Theorem 1.
Chapter 5

Extensions and Corollaries

Definitions:

parity, opposite, \( \dagger \)
parity change
A represents f

Proofs:

Change of parity preserves size.

L18 Change of parity commutes with PROD.

Time for DPDNF on symmetrizable formulae \( \leq 9|A|^4 \sqrt[4]{|V|} \)

L19 If A represents a symmetrical function, then we can extend A in the obvious way to a symmetrical DNF.

L20 Cook's \( D_3 \) Lemma

L21 symmetry p-reducible to tautology

T3 symmetry p-equivalent to tautology

L21 A special algorithm for tautology recognition of symmetric functions
Theorems 1 and 2 are easily extended to "symmetrizable" DNF formulae. In terms of formal logic, a formula is symmetrizable if uniformly replacing certain variables by their negations renders the formula symmetric. In DNF representation, a DNF is symmetrizable if interchanging + with - in certain columns produces a symmetric formula.

Definition: f is the opposite (in parity) of an element e iff

\[
f = \begin{cases} 
0 & \text{if } e = 0 \\
1 & \text{if } e = 1 \\
- & \text{if } e = + \\
+ & \text{if } e = - 
\end{cases}
\]

The opposite of e is denoted \( \hat{e} \).

Definition: B is a parity change (over I) of A iff

\( I \subseteq \{1, \ldots, \text{VARS}(A)\} \), and R is a clause of B iff there is an S in A with \( R[i] = S[i] \) if \( i \in I \), \( 1 \leq S[i] \) if \( i \notin I \).

Remark: If B is a parity change of A, then VARS(B) = VARS(A) and \( |B| = |A| \).

Proof: Clearly VARS(B) = VARS(A). Also \( |B| \leq |A| \). If \( S \neq S' \) and R,R' are parity changes over I of S and S' respectively, then R \( \neq R' \), since \( S[i] \neq S'[i] + R[i] = R'[i] \). Hence \( |B| \geq |A| \) and \( |B| = |A| \).

Q.E.D.

Lemma 18: Change of parity commutes with PROD, i.e. If B is a parity change over I of A, then PROD B is the parity change of PROD A over \( \{j|j+1 \in I\} \).
Proof: It is clear that change of parity commutes with reduction. To show it commutes with PRODUCT. Suppose for the moment that \( l \not\in I \). Then if \( i + 1 \in I \), and \( R, S \in B \) with \( R[l] = S[l] = l \) or \( R[l] = + \) and \( S[l] = - \), then

\[
(R \oplus S)[i] = R[i+1] \& S[i+1] \text{ by definition}
\]

\[
= \oplus R'[i+1] \& S'[i+1], \text{ where}
\]

\( T' \) means the clause in \( A \) corresponding to \( T \) in \( B \)

\[
= \oplus (R'[i+1] \& S'[i+1]) \text{ by inspection}
\]

\[
= \oplus ((R' \oplus S')[i]) \text{ by definition}
\]

and this is what we want.

If \( l \in I \), then \( R \oplus S = S' \oplus R' \), and we still get the same clauses since

\[ a \& b = b \& a. \]

Q.E.D.

Corollary 18-1: If \( B \) is a parity change of \( A \), then \( \text{DPDNF} \) takes exactly as long as \( B \) as on \( A \).

Definition: \( B \) is symmetrizable iff \( B \) is a parity change of a symmetric DNF \( A \).

Corollary 18-2: If \( A \) is symmetrizable then the time required for \( \text{DPDNF} \) to halt on \( A \) is no more than

\[ 9|A|^4(VARS(A))^4. \]

Proof: Theorem 1 and Corollary 18-1.
Another corollary to Theorem 1 is that deciding whether a DNF formula represents a symmetric Boolean function is p-equivalent to the tautology problem.

**Definition:** A represents \( f \) means that \( f : \{\text{true}, \text{false}\}^V \rightarrow \{\text{true}, \text{false}\} \), and \( f(x_1, \ldots, x_V) = \text{true} \) iff \( A \) is true for those same choices of \( x_1, \ldots, x_V \). \( f \) is symmetric iff there is an RSDNF \( B \) such that \( B \) represents \( f \).
There is a quicker proof of the aforementioned Corollary, but I think the next Lemma is interesting in its own right.

**Lemma 19:** If \( A \) is a DNF that represents a symmetrical Boolean function \( f \), and if \( A' \) is obtained from \( A \) by appending to \( A \) every permutation of every clause of \( A \), then \( A' \) represents \( f \) also.

**Proof:** It clearly suffices to show that if \( B = A \lor C \), where \( C \) is a single clause equal to an arbitrary permutation of an arbitrary clause of \( A \), then \( B \) represents \( f \). How could \( B \) represent a different function from \( f \)? Only if there is a valuation function \( v \) which makes \( A \) false and \( B \) true (\( A \) true, \( B \) false is impossible since true \( v \) anything = true). But if \( v \) makes \( A \) false then it makes \( C \) false.

Let \( \sigma, D \) be such that \( D \) is a clause of \( A \) and \( C = \sigma D \).

Suppose \( v(A) = \text{false} \). Then if \( u = v(\sigma(\ )) \), we have

\[
\begin{align*}
U(A) &= \text{false} \quad \text{by symmetry, so} \\
U(D) &= \text{false} \quad \text{since } D \text{ is a clause of } A, \text{ and} \\
& \quad \text{U(A) is false iff every clause of } U(A) \text{ is false.}
\end{align*}
\]

By definition, \( U(D) = V(\sigma(D)) \)

\[
\begin{align*}
= V(C) \quad \text{so } V(C) = \text{false as claimed. Hence if } V(A) = \text{false then} \\
V(B) &= V(A) \lor V(C) \\
& = \text{false} \lor \text{false} \\
& = \text{false.}
\end{align*}
\]
Hence we can add permutations of clauses of A to A until we obtain a symmetrical DNF.

Q.E.D.

Lemma 20: (due to Stephen Cook): If A is any DNF then in polynomial time and space one can write down a formula B in DNF with at most 3 literals per clause such that B is a tautology iff A is a tautology.

Proof: (This is essentially the Davis-Putnam algorithm in reverse).

For any clause C = D & E, C can be replaced by

($\overline{a} & D) v (a & E)$, where a is not a variable in A

(justification: Davis-Putnam rule (3) says ($\overline{a} & D)v(a & E)$ can be replaced by D&E = C). But this makes two shorter clauses as long as C has four or more literals. Repeating this systematically, with D always containing two literals and E always containing at least two literals, takes at most (number of literals in the clause) new variables for each clause of A, and a new clause of length 3 for each new variable, so at worst B is 3 times as long as A (and has at most (3·VARS(A)) times as many clauses as A).

We are now ready to prove the corollary.

Recall $P_1$ is p-equivalent to $P_2$ iff $P_1$ is p-reducible to $P_2$ and $P_2$ is p-reducible to $P_1$. We show the "obvious" one first:

Lemma 21: The problem of determining whether a formula in
DNF represents a symmetric Boolean function is p-reducible to the tautology problem.

Proof: Recall that $P_1$ is p-reducible to the tautology problem if either $P_1$ or "not $P_1$" can be recognized in polynomial time by a non-deterministic algorithm. I will exhibit an algorithm which non-deterministically accepts $A$ iff $A$ is not symmetrical. The algorithm is:

Pick $n$, $1 \leq n \leq \text{VARS}(A)$

Pick $I$ with $|I| = n$, and let

$x_i = \text{true if } i \in I$, false if $i \notin I$.

$V1 + \text{valuation of } A \text{ for those choices for } x_i$

($V1 = \text{true or false}$)

Pick $I'$ with $|I'| = n$, and let $x_i = \text{true if } i \in I'$, false if $i \notin I'$.

$V2 + \text{valuation of } A \text{ for those choices of } x_i$.

If $V1 \neq V2$ then accept $A$.

This algorithm can accept $A$, for some $n$, $I$, and $I'$, iff $A$ is not symmetrical. It clearly operates in polynomial (indeed linear) time.

Q.E.D.

The proof of Theorem 3, below, will thus consist of showing that the tautology problem is p-reducible to the symmetry problem.

Theorem 3: The problem of determining whether a formula in DNF represents a symmetric Boolean function is p-equivalent to the tautology problem. (Note that it is easy to
determine in polynomial time whether the formula is itself symmetric).

Proof: We wish to show that use of an oracle for "Does A represent a symmetric Boolean function?" will enable us
to determine in polynomial time whether A is a tautology.

First we use the method of Lemma 20 to replace A
by a DNF B with at most 3 literals per clause. Then we
ask the oracle if B is symmetric. If not, we say,
"no, A is not a tautology," because if A were a tautology,
B would be a tautology, and if B were a tautology, it
would be symmetric ($V(\sigma B) = true=V(B)$ for all $V$ and $\sigma$).
If B is symmetric we use the method of Lemma 19 to
replace B by a symmetrical DNF C (this takes at most
polynomial time/space because there are only $(2 \cdot (VARS(C) +1))^3$ distinct clauses of length no greater than 3 in
VARS(C) variables, so C can have at most $(2 \cdot (VARS(C)+1))^3$
clauses). Then we use the Davis-Putnam algorithm to
determine whether C is a tautology; this takes polynomial
time by Theorem 1.

Note that theorem 3 can be obtained more directly
as follows:

Lemma 21: (suggested by Hopcroft and Williams of Cornell):
There is a (special-purpose) polynomial algorithm for de-
ciding whether a formula is a tautology, given that it
represents a symmetric Boolean function.
Proof: The valuation of the formula depends only on the number of variables chosen to be true, so just evaluate the formula for

(1) no variable true
(2) \( a_1 \) true, all other false
(3) \( a_1 \) and \( a_2 \) true, all others false
...
(V+1) \( a_1 \ldots a_V \) all true

The formula is a tautology iff all results are true.

Alternate proof of theorem 3:

Use the oracle to decide whether \( A \) is symmetric; if so, then use Lemma 21 to decide if \( A \) is a tautology.

Q.E.D.

I do not see an obvious extension of Lemma 21 to include symmetrizable DNF's, unless you know in advance which literals (columns) to reverse. I also do not see how to extend Theorem 3 to replace "symmetrical" by "symmetrizable."

Another possible extension of these results is to "block-diagonally symmetric" DNF's. Call a DNF "block-diagonal" if it is a block-diagonal array, as the term is used in matrix theory. I think it is straightforward to show, using the same techniques as in this paper, that if \( A \) is a block-diagonal DNF in which all the blocks are symmetric, and if each block is identical to every other,
then DPDNF is polynomial on $\Lambda$. An interest in this extension is motivated by the observation that Stephen Cook's Turing machine description (in his May, 1971 ACM paper) is a perturbed block-diagonal formula, if the variables are arranged in order of increasing times.
Glossary

\[ a = (a^+, a^-); A^+, A^- \] - see clause-type

A[i] means the \( i^{th} \) literal of the clause A.

adj, adjacent: \( a \text{ adj b} \) (a is adjacent to b) iff \( a \odot b \) and

(for all \( c \) in CT(E)) not \( a \odot c \odot b \). \( a \odot b \) is a type of

PROD E only if \( a \text{ adj b} \) (Lemmas 11 and 12). \( a \text{ adj b} \)

in PROD E implies either \( a^+ = b^+ + 1 \), or \( 5 = a^- + 1 \), or

\( a \oplus b \) not in PROD \( ^2 E \) (Lemma 14).

annihilation - Clause A annihilates clause B if B contains

all literals of A, i.e. \( B \subseteq A \). For clause-types, a

annihilates b iff \( a \preceq b \) (Lemma 6).

clause (DNF notation) - a vector over the lattice \( \{0, 1, +, -\} \).

The formal-logic interpretation is that the clause A

contains \( x_i \) if \( A[i] = + \), \( \overline{x_i} \) if \( A[i] = - \),

\( x_i \& \overline{x_i} \) if \( A[i] = 0 \), and does

not contain \( x_i \) if \( A[i] = 1 \).

clause-type - If A is a clause then \( A^+ \) is the number of

+'s or positive literals in A, and \( A^- \) is the number

of -'s or negative literals in A. \( (A^+, A^-) \) is the

clause-type of A; or, A is a clause of type \( (A^+, A^-) \).

By convention if \( a \) is a clause-type, \( a = (a^+, a^-) \), and

A is any clause of type \( a \); likewise \( a = (A^+, A^-) \).

CNF, conjunctive normal form.

CT(E) is the set of all \( (A^+, A^-) \) such that A is a clause

of E. For a symmetric DNF, every clause, whose type

is in CT(E), is a clause of E (Lemma 4). For an RSDNF,
if a, b are in \( \text{CT}(E) \), then either \( a = b \) or \( a \lor b \) or \( b \lor a \) (Lemma 7).

DNF, disjunctive normal form. As a noun, a DNF means a set of vectors over \( \{0,1,+,-,\} \), as described under "clause". "DNF notation" means the representation of a DNF formula as a DNF (Chapter 2).

DPDNF - The Davis-Putnam algorithm in DNF notation (Chapter 2).

equivalent - The DNF A is equivalent to the DNF B iff they are true or false under the same set of valuations. In particular, if A is equivalent to B then A is a tautology iff B is a tautology.

lattice - The clauses in DNF notation are made up of the elements of the lattice

\[
\begin{array}{c}
1 \\
+ \\
- \\
0
\end{array}
\]

which is a Boolean algebra.

literal - In DNF notation, a literal is an element of the lattice \( \{0,1,+,-,\} \). The \( i \)th literal of the clause A is denoted \( A[i] \). 0 corresponds to \( x \& \overline{x} \), 1 to \( x \lor \overline{x} \) (i.e. no occurrence of \( x \)), + to \( x \), and - to \( \overline{x} \).

permutation - For any permutation \( \sigma \) on \( \{1,\ldots,\text{VARS}(A)\} \), \( \sigma A \) is formed by replacing \( R[i] \) by \( R[\sigma i] \) for every clause \( R \) of A (informally, we say \( \sigma A \) is A with the \( i \)th column replaced by the \( \sigma i \)th column).
is symmetric iff for every permutation \( \sigma \), \( \sigma E = E \).

\[ \text{PROD} = \text{PROD}(A) \], for the DNF \( A \) (also written \( \text{PROD} A \))

is equal to \( \text{REDUCTION}(\text{PRODUCT}(A)) \). For RSDNF's

\( \text{PROD}^n E = \text{PROD}^0 E = \text{PROD} E, \text{PROD}^n E = \text{PROD}(\text{PROD}^{n-1} E) \).

For RSDNF's, \( \text{PROD}^n \) is equivalent to \( n \) iterations of DPDNF.

\[ \text{PRODUCT} = \text{the DNF notation for Davis-Putnam rule}(3). \]

\[ \text{PRODUCT}(A) = \{ R \oplus S \mid R, S \text{ are clauses of } A \}. \]

\( \text{PRODUCT} \) preserves tautologies (Lemma 3).

product rule - Another name for Davis-Putnam rule(3),

incorporating "expansion" back to CNF.

reduced, \( \text{REDUCTION} \) - In formal logic notation, the

reduced form of \( A \) is the result of applying the

subsumption rule (annihilation, rule(4)) to \( A \).

\( \text{REDUCTION}(A) = \text{the reduced form of } A \) (Lemma 1).

\( \text{REDUCTION} \) preserves tautologies (Lemma 2). If

\( E \) is reduced and symmetric then \( \ominus \) is a total

ordering of \( \text{CT}(E) \). (Lemma 7).

RSDNF - reduced symmetric DNF.

rule(1) - A special case of rules (3) and (4), in which

the variable \( x \) of elimination occurs only as \( x \)
or only as \( \bar{x} \).

rule(2) - A special case of rules (3) and (4), in which

the variable \( x \) of elimination occurs alone, as
the clauses \( x \) or the clause \( \bar{x} \).
rule(3) – An application of consensus. Equivalent to the product rule; called PRODUCT in DNF notation. (Chapter 1).

rule(4) – The subsumption rule; annihilation; reduction. Added to the Davis-Putnam algorithm to overcome an exponential analysis time for certain symmetric formulae. (Chapter 1).

SEMIPROD E – The set of all clauses in PRODUCT(E) whose types are in CT(E) or equal a ⊕ b for some a adj b in E. An expository device.

σ(A) – See "permutation".

subsume, subsumption – An ambiguous term for annihilation.
symmetrical – E is symmetrical iff for every permutation σ, σE=E. The Davis-Putnam algorithm with subsumption rule is polynomial on symmetric formulae.

valuation – A DNF valuation is a DNF whose only literals are 0 and 1; it is true iff some clause contains only 1's. A valuation function is (1) a mapping v:{l,...,VARS(A)}→{true,false}; (2) a mapping V:{DNF's}→{valuations}. V is induced by v in the following way: S is a clause of V(A) iff there is a clause R of A such that we have for each i,

\[
\begin{array}{c|cc}
R[i] & S[i] \\
\hline
+ & 1 & 0 \\
- & 0 & 1 \\
0 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

A is a tautology iff for every V, V(A) is true.

VARS(A) = the number of variables in A; in DNF terminology, the number of elements in every clause.

adj - "is adjacent to". See "adjacent".

⊙ - a ⊙ b iff a = (a⁺,a⁻), b = (b⁺,b⁻), and a⁺ > b⁺ and a⁻ < b⁻. A strict order relation. See "clause-types".

[ ] - A[i] = the iᵗʰ literal of A. See "clause".

& - "inf" or "and". A function on the lattice {0,1,+,-}, extended elementwise to clauses. For arbitrary

x,y, +&− = x&0 = 0, +&1 = +, −&1 = −, x&y ≤ x.

v - "sup" or "or". A function on the lattice {0,1,+,-}, extended elementwise to clauses. For arbitrary

x,y, +v− = x v1 = 1, +v0 = +, −v0 = −, x×x v y.

⊙ - "circled sup". A "selector" operator to replace the searching mechanism of rule(3). x ⊙ y = 0 unless x=y=1 or x=+, y=−. See also ⊗.

≤ - less than or equal to; less true than; is annihilated by (for clauses); annihilates (for clause-types). x ≤ y iff x v y = y.

⊗ - "Prod". Consensus, adapted to DNF notation. For clauses A ⊗ B is given by


For clause-types, a ⊗ b = (a⁺-1,b⁻-1). See also "product".

⊙ - See "permutation".
+, - plus, minus. See "lattice".

a⁺, a⁻ - See "clause-type".

A⁺, A⁻ - See "clauses".
Appendix

Cook's Examples 1 and 2
Cook's Method I for tautology recognition
Cook's p-reducibility theorem
A note on $|\text{PROD } E|$ compared to $|E|^2$
An APL program for the Davis-Putnam algorithm
Some examples using the APL program
Examples for the Davis-Putnam Procedure

Stephen A. Cook

Example 1. The atoms are \( P_1, \ldots, P_m \), together with \( Q_{i_1}, \ldots, i_m \) where each of the \( m \) subscripts \( i_j \) ranges over \( \{0,1\} \).

Let \( F_m \) be the formula

\[
\bigvee \left( \bigwedge_{i_1, \ldots, i_m = 0,1} \hat{i}_1 P_1 \land \hat{i}_2 P_2 \land \ldots \land \hat{i}_m P_m \land \hat{j} Q_{i_1}, \ldots, i_m \right)
\]

where \( \hat{1} \) represents \( \neg \) (negation) and \( \hat{0} \) represents the empty string.

For example, \( F_2 \) is

\[
(P_1 \land P_2 \land \neg Q_{00}) \lor (P_1 \land \neg P_2 \land Q_{00}) \lor (P_1 \land \neg P_2 \land \neg Q_{01}) \lor (P_1 \land \neg P_2 \land \neg Q_{01})
\]

\[

V(\neg P_1 \land P_2 \land Q_{10}) \lor (\neg P_1 \land \neg P_2 \land Q_{10}) \lor (\neg P_1 \land \neg P_2 \land \neg Q_{11})
\]

\[
V(\neg P_1 \land \neg P_2 \land \neg Q_{11}).
\]

In general, \( F_m \) is a disjunction of \( 2^{m+1} \) clauses, where each clause is a conjunction of \( m+1 \) literals.

If each of the variables \( P_1, \ldots, P_m \) is eliminated in turn in the manner prescribed by the Davis-Putnam procedure, the resulting formula in factored form will be

\[
i_1, \ldots, i_m = 0,1 \quad (Q_{i_1}, \ldots, i_m \lor \neg Q_{i_1}, \ldots, i_m)
\]
When this is "multiplied out" to put it in disjunctive
normal form, the resulting formula has $2^2^n$ clauses each
with $2^m$ literals. Thus we have the following:

Proposition: For infinitely many $n$, there are DNF
tautologies with $n$ clauses and $\log n$ literals per clause
such that for at least one order of elimination of variables,
the Davis-Putnam procedure yields intermediate formulas of
more than $(\sqrt{2})^n$ clauses.

Unfortunately, if the $Q$'s are eliminated before the
$P$'s, the intermediate formulas are all shorter than the
original formula.

Example 2. The atoms are $P_1, P_2, \ldots, P_m$. Let $F_m^L$ be the
formula

$$\bigvee_{1 \leq i_1 < i_2 < \ldots < i_L \leq m} \left( (P_{i_1} \& P_{i_2} \& \ldots \& P_{i_L}) \lor (\neg P_{i_1} \& \neg P_{i_2} \& \ldots \& \neg P_{i_L}) \right).$$

Thus $F_m^L$ has $2^m$ clauses of length $L$ each, and is a
tautology iff $m \geq 2L-1$. Since the formula is symmetric in
all atoms, the order of elimination is irrelevant. The
original Davis-Putnam procedure did not include a subsumption
rule; i.e. a rule which eliminates every clause some subset
of whose literals comprises another clause. If we stick to
the original procedure and eliminate the atoms $P_1, P_2, \ldots, P_{L-1}$,
then among the resulting clauses will be all clauses of the
form

$$F_{i_1} \& \ldots \& F_{i_t} \& \neg P_{i_1} \& \ldots \& \neg P_{i_t}$$

where $i_1 \leq \ldots \leq i_t$. If $i_1 = \ldots = i_t$
then among the resulting clauses will be all clauses of the
form

$$P_{i_1} \& \ldots \& P_{i_t} \& \neg P_{i_1} \& \ldots \& \neg P_{i_t}$$

where $i_1 \leq \ldots \leq i_t$. If $i_1 = \ldots = i_t$
then among the resulting clauses will be all clauses of the
form

$$P_{i_1} \& \ldots \& P_{i_t} \& \neg P_{i_1} \& \ldots \& \neg P_{i_t}$$

where $i_1 \leq \ldots \leq i_t$. If $i_1 = \ldots = i_t$
then among the resulting clauses will be all clauses of the
form
where \( l \leq i_1 < i_2 < \ldots < i_t \leq m \) and \( l \leq j_1 < \ldots < j_t \leq m \),
\( t = 2^{l-2} - 2^{l-1} + 1 \), and \( i_r \neq j_s \), any \( r \neq s \), assuming \( m \) is sufficiently large. Thus the number of such clauses
is at least the binomial coefficient
\[
\binom{m-l}{l2^{l-1} - 2^l + 2}
\]
Setting \( k = l2^{l-1} - 2^l + 2 = \) (bottom term in the coefficient)
and taking \( m - l = 2k \), the number of clauses is at
least \( \binom{2k}{k} \geq \frac{1}{2k} 2^{2k} > 2^{(l-2)2^l} \) for large \( l \).

Note that \( F_m^l \) originally had \( 2^{m} < m < l2^{l}2^l \) clauses.
If \( n \) is the number of clauses in the original formula and
\( f(n) \) is the number of clauses after \( P_1, \ldots, P_{l-1} \) have been
eliminated, we see that \( f(n) \) grows much faster than any
polynomial, but somewhat slower than an exponential.

Unfortunately most of the resulting clauses are
supersets of smaller clauses in the same formula, so if the
subsumption rule is allowed, the \( F_m^l \) only require polynomial
time.
Stephen Cook's Method I, from his Spring 1972 course notes, University of Toronto

It is suggested by Robinson's lemma and its proof follows from it.

It operates on an expanding set of clauses \( C_1, C_2, \ldots, C_n \) (starting with the initial ones).

It has a push down stack of literals \( l_1, \ldots, l_m \), consistent with no repetitions.

At every time we have a partial truth assignment obtained by assigning true to \( l_1, \ldots, l_m \) on the stack. We think of the objective as finding a satisfying assignment.

1) \( m = 0 \); \( C_1, \ldots, C_n \) given (we start with initial clauses \( C_1, \ldots, C_n \) and empty stack)

Comment:

At every time before we enter step 2) no clause is completely falsified by the partial truth assignment which verifies all \( l_1, \ldots, l_m \).

2) \( m = m + 1 \)

Select a literal \( l_m \) such that neither \( l_m \) nor \( \overline{l_m} \) appears on the stack.

If no such \( l_m \) exists then we have found a satisfying truth assignment, so \( C_1, \ldots, C_n \) are consistent.

If no clause is falsified by verifying \( l_1, \ldots, l_m \), go to step 2).

If some clause is falsified go to step 3).

3) (Assume \( C_i \) is falsified when \( l_1, \ldots, l_m \) are verified)

Replace \( l_m \) by \( \overline{l_m} \) on the stack.
If no clause is falsified now, go to step 2).
Otherwise some clause $C_j$ will be falsified.
Let $n + n+1$, set $C_n$ equal to the resolvent of $C_i, C_j$.
(Note that we can resolve $C_i, C_j$:
They are not falsified by verifying $l_1, \ldots, l_{m-1}$, but
$C_i$ is falsified by verifying $l_1, \ldots, l_m$, and $C_j$ is falsified by verifying $l_1, \ldots, \overline{l}_m$, therefore $m$, $m$
is the only complementary pair of $C_i, C_j$)
If $C_n = \Box$ then stop - formula is inconsistent.
Otherwise let $t$ be the largest number so that $\overline{l}_t$
occurs in $C_n$.
Let $m + t$, $i + n$, to to step 3.
(Note that $C_n$ is falsified by verifying $l_1, \ldots, l_t$)
Remark: $C_n$ must be a new clause, because if it had appeared earlier, it would have been falsified earlier, and in step 3) either it would have been resolved or the stack literals would have been changed.
Therefore the method must terminate -- with a resolution proof, or a satisfying assignment.
Remark: Subsumption rule is not necessary for this method.
(Subsumption rule says that we can eliminate a clause $D$ if there is a clause $C$ such that $C \leq D$)
If $C \leq D$ then we never look at $D$, because $C$ would be falsified before $D$.
Also note that the resolvent $C_n$ will never contain an already existing clause.
In step 2) we did not specify how to select $l_m$.

**Conjecture:**

Given any inconsistent set of clauses $C_1, \ldots, C_n$ there is a selection strategy for the $l_m$ such that the resulting method I proof is the shortest possible resolution proof of the original clauses.
A Brief Formal Proof of Cook's p-reducibility Theorem

Conventions:
We have a Turing machine with set of states \( K \), whose elements are \( q, q', q'', q_1, q_2, \ldots \) with \( q_0 = \text{initial state} \) and \( q_f = \text{final state} \).
The tape is infinite to the right, and we will think of the squares as numbered 1, 2, 3, \ldots from the left end.
The tape alphabet is \( \Sigma \); its elements are \( b=\text{blank}, \sigma, \sigma', \sigma'', \ldots \).
The input is \( X \); its \( i^{th} \) symbol is \( X[i] \); \( X[i]=b \) for \( i > \lg(X) \).
The state and tape symbols are each alphabetically ordered (arbitrarily; we will not specify the ordering).

Our logical variables are \( T(t,i,\sigma) \) and \( P(t,i,q) \), with \( i \leq t \) (since other configurations are inaccessible, as will be seen from the definitions):
\[
T(t,i,\sigma) = \text{true iff at time } t \text{ the } i^{th} \text{ square of the tape contains } \sigma;
\]
\[
P(t,i,q) = \text{true iff at time } t \text{ the machine is in state } q \text{ and the head is on the } i^{th} \text{ tape square}.
\]

The state table is restricted (with at worst polynomial loss of time over an arbitrary table) to having deterministic move and print, and at most two-way nondeterminism in the state-change function.

State change is denoted by \( \delta \), move by \( m \), and print by \( p \).
Thus \( \delta(q,\sigma) = \{q',q''\} \), \( m(q,\sigma) = \begin{cases} +1 & \text{move right} \\ 0 & \text{no move} \\ -1 & \text{move left} \end{cases} \).
and \( p(q, \sigma) = \sigma' \). We also make the convention that
\[ \delta(q, \sigma) = \{q_f\}. \]

The lines of the formula are numbered for reference to the table that follows the formula.

\[
\begin{align*}
\& T(i, i, X[i]) \quad \text{1} \\
\& P(1, 1, q_0) \quad \text{2} \\
\& \quad \vee P(t, i, q_f) \quad \text{3} \\
\& \quad \text{l} \in \text{S} \\
\& \quad \bar{T}(t, i, \sigma) \vee T(t, i, \sigma') \quad \text{5} \\
\& \quad \text{2} \in \text{S} \\
\& \quad \text{S} \subseteq \Sigma, \sigma, \sigma' \in \Sigma \\
\& \quad \left( \vee P(t, i, q) \vee \bar{T}(t, i, \sigma) \vee T(t+1, i, \sigma) \right) \quad \text{6} \\
\& \quad \text{2} \in \text{S} \\
\& \quad \text{S} \subseteq \Sigma \\
\& \quad \left( \text{P}(t+1, i+m, q') \right) \quad \text{7} \\
\& \quad \text{P}(t+1, i+m, q") \\
\& \quad \{q', q"\} = \delta(q, \sigma) \\
\& \quad \text{l} \in \text{S} \\
\& \quad \text{q} \in \text{K} \\
\end{align*}
\]

\( \text{TABLE} \)

(references to line numbers in the preceding formula)

(1) Since squares beyond the \( i^{th} \) are not accessible to the head until \( i \) time units have passed, we do not want to talk about those inaccessible configurations. Thus we just want to know that at the first time
that the square could be scanned, it has the correct symbol. After this time, maintenance and updating are done by clauses 5, 6, and 8. (Of course there is nothing wrong with talking about those inaccessible configurations; this just shortens the formula.)

(2) The machine starts on square one at time one in the initial state.

(3) Once entering $q_f$, the machine remains in $q_f$. Hence acceptance can be stated as acceptance at time $T$, on some square.

(4) The head can be in at most one position at time $t$, and the machine in at most one state. We restrict $i \neq j$ so that the same disjunction does not occur twice. We require $(i, q) \neq (j, q')$ (meaning either $i \neq j$ or $q \neq q'$) because $(i, q) = (j, q')$ would say the machine was nowhere in no state.

(5) The tape can contain at most one symbol in a given square at a given time.

(6) If the head is not on a given square, that square will not have its content changed. $t \leq T - 1$ because otherwise $T + 1$ would occur, which we do not care about.

(7) State and move table. If $q' = q$ then the clause really contains only three literals.

(8) Print table.

Note that it is superfluous to add clauses saying "the machine is in some state at each time" or "each tape
square contains some symbol at each time", since the initial conditions and state table ensure this.

We can say overall that the formula contains a copy of the whole state table for the machine for the transition from time $t$ to time $t+1$ for each $t$, from $1$ to $T-1$. If we write this in my "algebraic" notation, with all $P(t+1,i,q)$ and $T(t+1,i,\sigma)$ following all $P(t,i,q)$ and $T(t,i,\sigma)$, we see that the formula becomes a perturbed "block-diagonal" array; for times $t$ and $t+1$ it looks like this:

```
  |
  |
  |
  |
  table for t-1 to t
  |
  |
  |
  |
  all l's
  |
  |
  |
  |
  table for t to t+1
  |
  |
  |
  |
  all l's
```

We have also a few oddments associated with initial and final conditions.

We see from this that another "general" class of DNF's (in the sense of $p$-reducibility) to add to Cook's class of DNF's with at most three literals per clause) is the class of perturbed block-diagonal DNF's, in which every block is identical except the upper left and lower right, which are lacking some clauses; and with the further exception of the input and final conditions
(the latter can of course be reduced to the one-literal clause \( P(T,1,q_f) \) if we make the head move back to square one to accept). This means that in looking for difficult formulae, we can restrict our attention to very highly structured formulae, with each variable occurring in only a few clauses. It is clear that a non-overlapping block-diagonal DNF is a disjunction of formulae over disjoint sets of variables, and hence is a tautology iff some block is a tautology.

I ran a formula similar to this, describing directly "this number is prime"; although the results were not conclusive, it is noteworthy that changing one literal (one of the bits of the input, which changed it from prime to non-prime) affected the running time and order of elimination to a significant degree. I found that although the REDUCTION consumed most of the time in each iteration, it was disastrous to omit it for even three or four iterations.
A Note on $|\text{PROD E}|$ Compared to $|E|^2$

It appears that the upper bound $\text{PROD E}$ cannot be significantly improved for RSDNF's. Taking $V$, $p$, and $m$ to be large, we can use Stirling's approximation, $n! \approx n^n e^{-n} \sqrt{2\pi n}$, to derive

$$\frac{(V)}{(p+m)} \approx \frac{V^V V}{2\pi (V-p-m) (V-p-m)^{V-p-m} \sqrt{p(m(V-p-m))}}$$

If $m=0$, but $p$ is still large, we get instead

$$\frac{(V)}{p} \approx \frac{V^V V}{\sqrt{2\pi} (V-p) (V-p)^{p} \sqrt{p(V-p)}}$$

If we let $p=m=V/3$, and let $E$ have the clause-types $(p,0)$, $(0,m)$, then the second equation gives $|E| \approx \frac{3}{\sqrt{\pi V}} 3 V^2 - 2V/3$

The clause-types of $\text{PROD E}$ are $(V/3,0)$, $(V/3-1,V/3-1)$, $(0,V/3)$, but there are far more clauses of the type $(V/3-1,V/3-1)$ than of the other two types, so

$$|\text{PROD E}| \approx \frac{(V-1)(2V/3-2)}{((V/3-1)!)^2}$$

by the first equation.

Now substituting $(V-1)^x \approx x$ for $x$ much smaller than $V$, and $(V+c)^y \approx v^y e^c$ for $y \approx c$ (from the binomial expansion of $(V+c)^y$), we get

$$|\text{PROD E}| \approx \frac{\sqrt{3}}{2\pi V} 3^V$$
If we set $|\text{PROD } E| = K V^C E^P$, we find that

$$p = \frac{\ln(3)}{\ln(3)-(2\ln(2)/3)} \approx 1.73,$$

$$K = \frac{\pi^{p/2-1}}{2} \approx 0.111,$$

and $c = p/2-1 \approx -0.137$.

I have calculated exact values of $|E|$ and $|\text{PROD } E|$ up to $V=45$ using the APL factorial function and Lemma 15. I also calculated $|\text{PROD } E|/K V^C |E|^P$, using the values just derived for $K$, $c$, and $p$:

| $V$  | $|E|$ | $|\text{PROD } E|$ | $|\text{PROD } E|/K V^C |E|^P$ |
|-----|------|----------------|---------------------------------|
| 6   | 30   | 40            | 1.30                            |
| 9   | 168  | 532           | .933                            |
| 15  | 6006 | 214,214       | .840                            |
| 24  | 1,470,942 | 2,805,576,708 | .883                            |
| 27  | 9,373,650 | 6.84·10^{10}  | .895                            |
| 30  | 60,090,030 | 1.69·10^{12}  | .904                            |
| 45  | 6.90·10^{11} | 1.67·10^{19}  | .934                            |
An APL program for the Davis-Putnam Algorithm

V PROD
[1] ***PROD REPLACES EXPR BY PROD(A LOCALLY OPTIMAL
FORM OF EXPR).
[2] ELIM+ELIMCOL
[3] ***ELIMCOL YIELDS AN OPTIMAL VARIABLE FOR ELIMA-
TION.
[4] 'THE VARIABLE TO BE ELIMINATED IS ';ELIM
[5] PREPEXPR
[6] ***PREPEXPR GENERATES THREE SUBSETS OF EXPR, THOSE
CONTAINING THE VARIABLE OF ELIMINATION
[7] a  (1)POSITIVELY,(2)NEGATIVELY,(3)NOT AT ALL.
[8] 'OCURRENCES OF +: ';(pPLUSEXPR)[2];' :- ';(p
MINUSEXPR)[2];' 1: ';(pONEXPR)[2]
[9] PRODUCT
[10] ***PRODUCT REPLACES EXPR BY PRODUCT(EXPR) AS
THOUGH V[ELIM] WERE THE FIRST VARIABLE, USING
[12] 'BEFORE REDUCTION, EXPR HAS ';(pEXPR)[2];' CLAUSES
IN ';(pEXPR)[3];' VARIABLES:'
[13] INTERPEXPR
[14] ***INTERPEXPR PRINTS EXPR IN DNF FORMALISM.
[15] REDUCTION
[16] ***REDUCTION REPLACES EXPR BY REDUCED FORM OF EXPR
AND PRINTS NEW CLAUSES NOT ANNIHILATED.

V
\[39\] \rightarrow \text{INCR} : \text{COUNT} \leq \text{STOP}
\[40\] \text{DON'T : EXP+}((,\text{(ONEXP}[1,;]))^{}, \text{NEWTOPV},(,\text{(ONEXP}[2,;]), \text{NEWTV})
\[41\] **Too long for one line.** \text{EXPR} \text{ IS A STRING OF ALL TOPS FOLLOWED BY ALL BOTTOMS.}
\[42\] \text{EXPR}+:((,\text{(PONEXP})[2])+((\text{PNEWTV}),((\text{PONEXP})[3]))^{}, \text{PEXPR})
\[43\] **EXPR IS NOW IN NORMAL SHAPE, (2*CLUSES*VARS).**
\[44\] \text{OLDONES}+:((\text{PONEXP})[2])
\[45\] **OLDONES IS EQUAL TO THE NUMBER OF CLAUSES IN \text{ONEXPR, I.F. THAT WERE ALREADY IN EXPR.}}
\[46\] \text{A BEFORE APPLICATION OF PRODUCT AND HAD C[ELIM]=1.}
\[47\] \text{ONEXP}+\text{PLUSEXPR}+\text{MINUSEXP}+0
\[48\] **SET UNNEEDED GLOBAL VARIABLES TO 0.**

\[49\] \text{REDUCTION} : \text{STOP} ; \text{COUNT} ; \text{COMPARE} ; \text{NEWNULL} ; \text{NEXTGE} ; \text{NEXTEQ} ; \text{NEXTNULL} ; \text{NULLVEC} ; \text{VARS} ; \text{COMPEXPR} ; \text{COMPVARS} ; \text{COMPROW} ; \text{NOTONES}
\[50\] **REDUCTION REPLACES EXPR BY REDUCED FORM OF EXPR.**
\[51\] \text{NULLVEC}+\text{OLDONES}^p1
\[52\] **NULLVEC WILL HAVE ZEROS FOR CLAUSES THAT ARE ANNIHILATED.**
\[53\] \text{VARS}+:((\text{PEXPR})[3])
\[54\] \text{STOP}+:((\text{PEXPR})[2])
\[55\] \text{STOP}+\text{NEW} ; \text{OLDONES} \geq \text{STOP}
\[56\] \text{COUNT} + \text{OLDONES}
\[57\] **START ON FIRST NEW CLAUSE.**
\[58\] \text{INCR} : \text{COUNT} = \text{COUNT} + 1
\[59\] \text{COMPROW}+\text{EXPR}[; \text{COUNT};]
\[60\] **COMPROW IS CLAUSE TO BE COMPARED WITH ALL PRECEDING CLAUSES.**
\[61\] \text{NOTONES}++(\text{COMPROW}[1,;]) \land \text{COMPROW}[2,;]
\[62\] **NOTONES[I]=1 IFF ITH ELEMENT OF COMPROW \neq 1**
\[63\] \text{COMPVARS}\mathcal{P} / \text{NOTONES}
\[64\] **COMPVARS IS NUMBER OF LITERALS IN COMPROW.**
\[65\] \text{COMPEXPR}+\text{NOTONES}[3])(2, \text{COUNT} - 1, \text{VARS})+\text{EXPR}
\[66\] **EXTRACT RELEVANT PORTION OF EXPR (ONLY CERTAIN COLUMNS OF PREVIOUS ROWS: THOSE**
\[67\] \text{COLUMNS I FOR WHICH COMPROW[I]=1).}
\[68\] \text{COMPARE}+(((\text{(COUNT} - 1) \times \text{COMPVARS})p(\text{NOTONES})(\text{COMPROW}[1,;]))^{},((\text{(COUNT} - 1) \times \text{COMPVARS})p(\text{NOTONES})(\text{COMPROW}[2,;]))^{}
\[69\] **TOO LONG FOR ONE LINE.**
\[70\] \text{COMPARE}+((\text{COUNT} - 1), \text{COMPVARS})p \text{COMPARE}
\[71\] **COMPARE IS (COUNT-1) COPIES OF THE NON-ONE ELEMENTS OF THE COUNTTH CLAUSE.**
[24] a***NEWNULL HAS ZEROES FOR CLAUSES PREVIOUS TO AND
   < THE COUNTH CLAUSE OF EXPR.
[25] ONEONE = [2](~[1])(~NOTONES) /[3)((2,(COUNT-1),VARS
   +EXPR))
[26] a***ONEONE[I] = 1 IFF (JTH CLAUSE OF EXPR HAS ONLY
   ONES WHERE COMPROW HAS ONES).
[27] NEXTGE = [2](~[1] COMPEXPR >= COMPARE))
[29] NEXTNULL = (~[ONEONE,NEXTGE,NEXTEQ])
[30] a***NEXTNULL = 0 IFF SOME PREVIOUS CLAUSE IS >
   COUNTH CLAUSE OF EXPR.
[31] NULLVEC = (NULLVEC \ NEWNULL), NEXTNULL
[32] a***UPDATE NULLVEC. NULLVEC[I] = 0 IFF FOR SOME
   J < K < COUNT, EITHER I = K AND JTH CLAUSE > KTH
   CLAUSE, OR I = J AND KTH CLAUSE = JTH CLAUSE.
[33] a INCR COUNT < STOP
[34] 'THE NUMBER OF OLD CLAUSES DELETED IS ' +/(OLDONES+
   ~NULLVEC)
[35] 'THE NUMBER OF NEW CLAUSES DELETED IS ' +/(OLDONES+
   ~NULLVEC)
[36] EXPR = NULLVEC/[2] EXPR
[37] a***DELETE REDUNDANT CLAUSES.
[38] 'THE NEW CLAUSES AFTER REDUCTION ARE'
[39] PRINTEXPR(+/(OLDONES+NULLVEC))+1
[40] a
[41] NEW = 'NO NEW CLAUSES WERE FORMED.'
\(\text{ANS+VECEXPR A}\)

1. **A**  
   **A** **E**  **C**  **E**  **R**  **E**  **X**  **P**  **R**  **A**  **T**  **T**  **I**  **O**  **N**  **T**  **O**  
   **R**  **I**  **T**  **R**  **E**  **P**  **R**  **E**  **S**  **E**  **N**  **T**  **A**  **T**  **I**  **O**  **N**  

2. \(\text{ANS+}(2, (\rho A)) \rho ((A \epsilon -1'), ((A \epsilon +1'))))\)

\(\text{ANS+INTERP A}\)

1. **A**  
   **A** **M**  **E**  **R**  **P**  **S**  **I**  **T**  **I**  **O**  **N**  **T**  **O**  
   **R**  **I**  **T**  **T**  **R**  **E**  **P**  **R**  **E**  **S**  **E**  **N**  **T**  **A**  **T**  **I**  **O**  **N**  

2. \(\text{ANS+}'0+-1'[1+(A[2;]+2\times A[1;])]\)

\(\text{INTERP_EXPR}\)

1. **A**  
   **A** **M**  **E**  **R**  **P**  **S**  **I**  **T**  **I**  **O**  **N**  **T**  **T**  **I**  **O**  
   **R**  **I**  **T**  **T**  **R**  **E**  **P**  **R**  **E**  **S**  **E**  **N**  **T**  **A**  **T**  **I**  **O**  **N**  
   **T**  **I**  **O**  **N**  **N**  **S**  **S**  **L**  **E**  **E**  **S**  **S**  **S**  **P**  **A**  **C**  **E**  **N**  
   **T**  **H**  **A**  **N**  **T**  **I**  **O**  **N**  

2. \(\text{ANS+}'0+-1'[1+(EXPR[2;]+2\times EXPR[1;])]\)

\(\text{PRINT_EXPR START}\)

1. **A**  
   **A** **M**  **E**  **R**  **P**  **S**  **I**  **T**  **I**  **O**  **N**  **T**  **T**  **I**  **O**  
   **R**  **I**  **T**  **T**  **R**  **E**  **P**  **R**  **E**  **S**  **E**  **N**  **T**  **A**  **T**  **I**  **O**  
   **T**  **I**  **O**  **N**  **N**  **S**  **S**  **L**  **E**  **E**  **S**  **S**  **P**  **A**  **C**  **E**  **N**  
   **T**  **H**  **A**  **N**  **T**  **I**  **O**  **N**  

2. \(\text{PRINT_EXPR START}\)

3. \(\text{PRINT Thị START>1}\)

4. \(\text{PRINT Thị START>1}\)

5. \(\text{PRINT**:INTERP(0,(START-1),0)+EXPR}\)

\(\text{ANS+TIME}\)

1. **A**  
   **A** **M**  **E**  **R**  **P**  **S**  **I**  **T**  **I**  **O**  **N**  **T**  **T**  **I**  **O**  
   **R**  **I**  **T**  **T**  **R**  **E**  **P**  **R**  **E**  **S**  **E**  **N**  **T**  **A**  **T**  **I**  **O**  

2. \(\text{ANS+}(\text{TIME}+60)\)

\(\text{ANS+SPACE}\)

1. **A**  
   **A** **M**  **E**  **R**  **P**  **S**  **I**  **T**  **I**  **O**  **N**  **T**  **T**  **I**  **O**  
   **R**  **I**  **T**  **T**  **R**  **E**  **P**  **R**  **E**  **S**  **E**  **N**  **T**  **A**  **T**  **I**  **O**  

2. \(\text{ANS+I22}\)

\(\text{ANS+TIME}\)

1. **A**  
   **A** **M**  **E**  **R**  **P**  **S**  **I**  **T**  **I**  **O**  **N**  **T**  **T**  **I**  **O**  
   **R**  **I**  **T**  **T**  **R**  **E**  **P**  **R**  **E**  **S**  **E**  **N**  **T**  **A**  **T**  **I**  **O**  

2. \(\text{ANS+}(\text{TIME}+60)\)

\(\text{ANS+SPACE}\)

1. **A**  
   **A** **M**  **E**  **R**  **P**  **S**  **I**  **T**  **I**  **O**  **N**  **T**  **T**  **I**  **O**  
   **R**  **I**  **T**  **T**  **R**  **E**  **P**  **R**  **E**  **S**  **E**  **N**  **T**  **A**  **T**  **I**  **O**  

2. \(\text{ANS+I22}\)
\( \text{INITIALEXPR} \)

1. This function initializes \( \text{EXPR} \) to be a character array of all ones, and sets
2. a global variables.
3. \( \text{SHAPE}+2\times0 \)
4. 'NUMBER OF VARIABLES?'
5. \( \text{SHAPE}[2]+[] \)
6. 'NUMBER OF CLAUSES?'
7. \( \text{SHAPE}[1]+[] \)
8. \( \text{EXPR}+\text{SHAPE}+1 \)
9. \( \text{FINISHT} 0 \)
10. \( \text{VARMAX}+\text{SHAPE}[2] \)
11. \( \text{CLMAX}+\text{SHAPE}[1] \)

\( \text{MAKEXPR} \)

1. MAKEXPR is used to build an expression in character representation, assuming it has
2. been initialized by INITIALEXPR. READCLAUSE does all the work; MAKEXPR provides
3. an entry point, and a means of correcting clauses.
4. \( \text{READ}:'\text{NUMBER OF CLAUSE}?' \)
5. \( \text{WHERE}+\text{STARTMC}+[] \)
6. ENTER clause number which you want to have read
7. in. READCLAUSE will continue by
8. increments of 1. Enter 0 to exit.
9. \( \text{READCLAUSE} \)
10. \( \text{READ} \)

\( \text{READCLAUSE;}\text{COUNT;}\text{ENTRIES} \)

1. READCLAUSE is used with MAKEXPR to read in an
2. expression in indexed notation.
3. F.G. \( A_1=A_3A_4 \) IS READ IN AS \( 1-3 4 \), AND GOES
4. INTO \( \text{EXPR} \) AS \( +1-4 \).
5. TO EXIT OR TO CORRECT A CLAUSE, ENTER 0.
6. \( \text{COUNT}+\text{WHERE}+\text{STARTMC} \)
7. \( \text{READ}:'\text{TEXT OF CLAUSE '};\text{COUNT}';?' \)
8. \( \text{ENTRIES}+[] \)
9. \( \text{A} ; \text{EXPR}+\text{COUNT} ; (\text{ENTRIES}>0) / \text{ENTRIES} \)+''
10. \( \text{EXPR}+\text{COUNT} ; (\text{ENTRIES}<0) / \text{ENTRIES} \)+'-'
11. \( \text{COUNT}+\text{COUNT}+1 \)
12. \( \text{READ} \)
\( \text{v ANS+NUMINTERP} \space ANEXPR;SKEXPR;PEEXPR;INVEC;INVAL \)

[1] \( \text{A***NUMINTERP CONVERTS FROM BIT REPRESENTATION TO INDEXED REPRESENTATION.} \)

[2] \( \text{a IF CHARACTER FORM WOULD HAVE ANEXPR[I;J]=1,} \)
\( \text{INDEXED FORM WILL HAVE ANS[I;J]=0;} \)

\( \text{ANS[I;J]=-J.} \)

[4] \( \text{SKEXPR+((pANEXPR)[2;3])p1(pANEXPR)[3]} \)

[5] \( \text{PEEXPR+(~(ANEXPR[1;})])xSEXPR} \)

[6] \( \text{PEEXPR+PEEXPR-(~(ANEXPR[2;})])xSEXPR} \)

[7] \( \text{INVEC+([a/2](v/[1] ANEXPR))} \)

[8] \( \text{INVAL+([+/INVEC)+INVEC} \)

[9] \( \text{OKx1(pINVAL)=0} \)

[10] \( \text{\'INVALID CLAUSES: \'}\text{INVAL} \)

[11] \( \text{OK:ANS+PEEXPR} \)

\( \text{v ANS+PRINTINDEXED A;CLG;LITERALS;LENGTHS;M;EXPAND; SELECT} \)

[1] \( \text{A***0 - 2 0 4 - 5 6 PRINTS AS \'-2 4 -5 6 0 0 ... \') \)

[2] \( \text{A***A MUST BE IN INDEXED REPRESENTATION,} \)

[3] \( \text{A***I WOULD HAVE LIKED TO USE \& \& \& NOT SUPPORT \& \& ON ARRAYS.} \)

[4] \( \text{LITERALS-\sim A\&0} \)

[5] \( \text{A***LITERALS HAS ONES WHERE A IS NOT 0.} \)

[6] \( \text{LENGTHS+LITERALS} \)

[7] \( \text{A***LENGTHS IS A VECTOR WHOSE ITH ENTRY TELLS THE LENGTH OF THE ITH CLAUSE OF A.} \)

[8] \( \text{CLG+/LENGTHS} \)

[9] \( \text{A***CLG = LENGTH OF LONGEST CLAUSE OF A.} \)

[10] \( \text{SELECT+((CLG,CLG)p\&CLG)\&\&((CLG,CLG)p\&CLG)} \)

[11] \( \text{A***SELECT IS A CLG BY CLG ARRAY WITH ONES ON AND BELOW THE MAIN DIAGONAL, ZEROS ABOVE.} \)

[12] \( \text{M+\&LITERALS)/A} \)

[13] \( \text{A***M IS THE CLAUSES OF A STRUNG OUT, WITH NO 0'S.} \)

[14] \( \text{EXPAND+,SELECT[LENGTHS;]} \)

[15] \( \text{A***EXPAND TELLS WHERE 0'S BELONG, TO DELINEATE CLAUSES.} \)

[16] \( \text{ANS+((pLENGTHS),CLG)pEXPAND\&M} \)

[17] \( \text{A***ALL CLAUSES ARE FILLED IN WITH 0'S TO THE SAME} \)
\( \text{LENGTH AS THE LONGEST CLAUSE.} \)
A computer session with the Davis-Putnam algorithm

INITIAL_EXPR
NUMBER OF VARIABLES?

8

NUMBER OF CLAUSES?

11

MAKE_EXPR
NUMBER OF CLAUSE?

1

TEXT OF CLAUSE 1?

1 2

TEXT OF CLAUSE 2?

-1 3 4

TEXT OF CLAUSE 3?

-2 3 4

TEXT OF CLAUSE 4?

-3 5 6

TEXT OF CLAUSE 5?

-4 5 6

TEXT OF CLAUSE 6?

-5 7 8

TEXT OF CLAUSE 7?

-6 7 8

TEXT OF CLAUSE 8?

-5 -7

TEXT OF CLAUSE 9?

-5 -8

TEXT OF CLAUSE 10?

-6 -7

TEXT OF CLAUSE 11?

-6 -8

TEXT OF CLAUSE 12?

0

NUMBER OF CLAUSE?

0
(continued)

\text{NUMINTERP EXPR}
\begin{align*}
1 & 2 \ 0 \ 0 \ 0 \ 0 \\
-1 & 0 \ 3 \ 4 \ 0 \ 0 \\
0 & -2 \ 3 \ 4 \ 0 \ 0 \\
0 & 0 \ -3 \ 0 \ 5 \ 6 \\
0 & 0 \ 0 \ 4 \ 5 \ 6 \\
0 & 0 \ 0 \ 0 \ -5 \ 0 \ 7 \\
0 & 0 \ 0 \ 0 \ 0 \ -6 \ 7 \\
0 & 0 \ 0 \ 0 \ -5 \ 0 \ 7 \\
0 & 0 \ 0 \ 0 \ -5 \ 0 \ -8 \\
0 & 0 \ 0 \ 0 \ 0 \ -6 \ -7 \\
0 & 0 \ 0 \ 0 \ 0 \ -6 \ 0 \\
\end{align*}

\text{PRINTINDEXED NUMINTERP EXPR}
\begin{align*}
1 & 2 \ 0 \\
-1 & 3 \ 4 \\
-2 & 3 \ 4 \\
-3 & 5 \ 6 \\
-4 & 5 \ 6 \\
-5 & 7 \ 8 \\
-6 & 7 \ 8 \\
-7 & 7 \ 0 \\
-5 & 8 \ 0 \\
-6 & -7 \ 0 \\
-6 & -8 \ 0 \\
\end{align*}

\text{PROD}
\text{THE VARIABLE TO BE ELIMINATED IS 1}
\text{OCURRENCES OF +: 1; -: 1; 1: 9}
\text{BEFORE REDUCTION, EXPR HAS 10 CLAUSES IN 7 VARIABLES:}
\begin{align*}
++1111 \\
1-1++11 \\
11-++11 \\
111-1++ \\
1111-++ \\
111-1-1 \\
111-11- \\
1111--1 \\
11111-1 \\
+++1111 \\
\end{align*}
\text{THE NUMBER OF OLD CLAUSES DELETED IS 0}
\text{THE NUMBER OF NEW CLAUSES DELETED IS 0}
\text{THE NEW CLAUSES AFTER REDUCTION ARE}
\begin{align*}
++1111 \\
\end{align*}
(continued)

PROD
THE VARIABLE TO BE ELIMINATED IS 1
OCCURRENCES OF +: 1; -: 1; 1: 8
BEFORE REDUCTION, EXPR HAS 9 CLAUSES IN 6 VARIABLES:

-1++11
1-++11
11-1++
11-++
11-1-1
11-11-
111-1-
111-1-
++1111

THE NUMBER OF OLD CLAUSES DELETED IS 0
THE NUMBER OF NEW CLAUSES DELETED IS 0
THE NEW CLAUSES AFTER REDUCTION ARE

++1111

PROD
THE VARIABLE TO BE ELIMINATED IS 1
OCCURRENCES OF +: 1; -: 1; 1: 7
BEFORE REDUCTION, EXPR HAS 8 CLAUSES IN 5 VARIABLES:

-++11
1-1++
11-++
1-1-1
1-11-
11--1
11-1-
++11

THE NUMBER OF OLD CLAUSES DELETED IS 0
THE NUMBER OF NEW CLAUSES DELETED IS 0
THE NEW CLAUSES AFTER REDUCTION ARE

++11

PROD
THE VARIABLE TO BE ELIMINATED IS 1
OCCURRENCES OF +: 1; -: 1; 1: 6
BEFORE REDUCTION, EXPR HAS 7 CLAUSES IN 4 VARIABLES:

-1++
1-++
-1-1
-1-1
1-1-
1-1-
++11

THE NUMBER OF OLD CLAUSES DELETED IS 0
THE NUMBER OF NEW CLAUSES DELETED IS 0
THE NEW CLAUSES AFTER REDUCTION ARE

++11
(continued)

**PROD**

THE VARIABLE TO BE ELIMINATED IS 1

OCCURRENCES OF +: 1; -: 3; 1: 3

BEFORE REDUCTION, EXPR HAS 6 CLAUSES IN 3 VARIABLES:

++
--1
-1-
+++  
+-1

THE NUMBER OF OLD CLAUSES DELETED IS 0
THE NUMBER OF NEW CLAUSES DELETED IS 0
THE NEW CLAUSES AFTER REDUCTION ARE

+++  
+-1

**PROD**

THE VARIABLE TO BE ELIMINATED IS 2

OCCURRENCES OF +: 2; -: 2; 1: 2

BEFORE REDUCTION, EXPR HAS 4 CLAUSES IN 2 VARIABLES:

--
+-
++

THE NUMBER OF OLD CLAUSES DELETED IS 0
THE NUMBER OF NEW CLAUSES DELETED IS 0
THE NEW CLAUSES AFTER REDUCTION ARE

++

**PROD**

THE VARIABLE TO BE ELIMINATED IS 1

OCCURRENCES OF +: 2; -: 2; 1: 0

BEFORE REDUCTION, EXPR HAS 2 CLAUSES IN 1 VARIABLES:

-
+

THE NUMBER OF OLD CLAUSES DELETED IS 0
THE NUMBER OF NEW CLAUSES DELETED IS 0
THE NEW CLAUSES AFTER REDUCTION ARE

pEXPR=2 2 1
-
+

@TAUTOLOGY.

@NEXT PAGE BEGINS A NEW EXAMPLE.
Another DP computer session

TIME
0.0333333333

INITIALFXPR

NUMBER OF VARIABLES?
4

NUMBER OF CLAUSES?
16

EXPR + VECEXPR

EXPR

TIME

0.15

EXPR[1;;] + q2 2 2 2 + 16

TIME

0.2

INTERPEXPR

+++1
++1+
++11
+1++
+1+1
+11+
+111
1+++1
1++1
1+1+
1+11
11++
11+1
111+
1111

TIME

0.3

EXPR[2;;] + q2 2 2 2 + 16

TIME

0.3333333333

INTERPEXPR
(continued)

<table>
<thead>
<tr>
<th>INTERPEXPR</th>
<th>NUMINTERP</th>
<th>EXPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>+++-</td>
<td>1 2 3 - 4</td>
<td></td>
</tr>
<tr>
<td>++-</td>
<td>1 2 3 - 4</td>
<td></td>
</tr>
<tr>
<td>+++</td>
<td>1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>+--</td>
<td>1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>+--</td>
<td>1 2 3 - 4</td>
<td></td>
</tr>
<tr>
<td>+--</td>
<td>1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>-++</td>
<td>1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>-+-</td>
<td>1 2 3 - 4</td>
<td></td>
</tr>
<tr>
<td>-+-</td>
<td>1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>-+-</td>
<td>1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>-+-</td>
<td>1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>-+-</td>
<td>1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>-+-</td>
<td>1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>-+-</td>
<td>1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>+++++</td>
<td>1 2 3 4</td>
<td></td>
</tr>
</tbody>
</table>

TIME
0.4333333333

TIME
0.6333333333

PROD

THE VARIABLE TO BE ELIMINATED IS 1
OCCURRENCES OF +: 8; -: 8; 1: 0
BEFORE REDUCTION,EXPR HAS 8 CLAUSES IN 3 VARIABLES:

+++ 
+-- 
++- 
-++ 
-- 
--- 

THE NUMBER OF OLD CLAUSES DELETED IS 0
THE NUMBER OF NEW CLAUSES DELETED IS 0
THE NEW CLAUSES AFTER REDUCTION ARE
pEXPR=2 8 3

+++ 
+-- 
++- 
-- 
-++ 
+- 
--- 

TIME
3.3666666667
(continued)

```
NUMINTERP EXPR
  1 2 3
  1 2 3
  1 2 3
  1 2 3
  1 2 3
  1 2 3
  1 2 3
  TIME
3.483333333
PROD
THE VARIABLE TO BE ELIMINATED IS 1
OCCURRENCES OF +: 4; -: 4; 1: 0
BEFORE REDUCTION, EXPR HAS 4 CLAUSES IN 2 VARIABLES:
  ++
  +- 
-+
-

THE NUMBER OF OLD CLAUSES DELETED IS 0
THE NUMBER OF NEW CLAUSES DELETED IS 0
THE NEW CLAUSES AFTER REDUCTION ARE
ρEXPR=2 4 2
++
+-
-+
-

TIME
5.066666667
    NUMINTERP EXPR
  1 2
  1 2
  -1 2
  -1 2
  TIME
5.183333333
PROD
THE VARIABLE TO BE ELIMINATED IS 1
OCCURRENCES OF +: 2; -: 2; 1: 0
BEFORE REDUCTION, EXPR HAS 2 CLAUSES IN 1 VARIABLES:
  +
-

THE NUMBER OF OLD CLAUSES DELETED IS 0
THE NUMBER OF NEW CLAUSES DELETED IS 0
THE NEW CLAUSES AFTER REDUCTION ARE
ρEXPR=2 2 1
+
-

TIME
6.083333333

#TAUTOLOGY.
#NEXT PAGE BEGINS A NEW EXAMPLE.
```
Another DP computer session

TIME

2.55

PRINTEXPR 1

\[ \text{PRINTINDEXED} \quad \text{NUMINTERP} \quad \text{EXPR} \]

\[
\begin{array}{cccc}
1 & 0 & 0 \\
-1 & 2 & 3 \\
2 & 3 & 4 \\
2 & 3 & 9 \\
3 & 4 & 5 \\
3 & 4 & 9 \\
2 & 5 & 6 \\
2 & 7 & 9 \\
4 & 5 & 6 \\
4 & 5 & 8 \\
3 & 6 & 7 \\
3 & 5 & 9 \\
5 & 9 & 11 \\
6 & 10 & 11 \\
2 & 7 & 8 \\
2 & 7 & 10 \\
5 & 6 & 7 \\
5 & 6 & 7 \\
4 & 7 & 8 \\
4 & 6 & 8 \\
6 & 8 & 9 \\
5 & 6 & 7 \\
3 & 8 & 10 \\
3 & 5 & 10 \\
4 & 6 & 9 \\
4 & 6 & 11 \\
4 & 5 & 10 \\
5 & 7 & 11 \\
4 & 7 & 8 \\
5 & 7 & 8 \\
-2 & 9 & 10 \\
2 & 6 & 10 \\
\end{array}
\]

TIME

3.5833333333

PROD

THE VARIABLE TO BE ELIMINATED IS 1

OCCURRENCES OF +: 1; -: 1; 1: 30

BEFORE REDUCTION, EXPR HAS 31 CLAUSES IN 10 VARIABLES:

PROD[14]
TIME
9.7

PROD
THE VARIABLE TO BE ELIMINATED IS 9
OCCURRENCES OF +: 4; -: 2; 1: 22
BEFORE REDUCTION, EXPR HAS 26 CLAUSES IN 8 VARIABLES:

++11111
+-11111+
1+-+111
1++1111
-11++111
+1111+1-
11-++111
11+-11+1
1-11++11
1+1++111-
-1111++1
111-++11
111+-111
11-11++1
11+1+1-1
1111-1++
111++-11
11+1+11-
1+11-+1
111+1+-1
++111111
11+-+11+
+11+1-11
1-++11+1
-1++111+
+111+-11
+-11+1+1
++1-+111

THE NUMBER OF OLD CLAUSES DELETED IS 0
THE NUMBER OF NEW CLAUSES DELETED IS 2
THE NEW CLAUSES AFTER REDUCTION ARE

+1++1-11
-1++111+
+111+-11
+-11+1+1

TIME
14.11666667
TIME
14.11666667

THE VARIABLE TO BE ELIMINATED IS 7

OCCURRENCES OF +: 6; -: 2; 1: 18

BEFORE REDUCTION, EXPR HAS 26 CLAUSES IN 7 VARIABLES:
-++1111
+-1111+
1-++111
1+-111+
-11++11
+1111-
11-++11
1-11++1
1+1+11-
111--++1
111+-+1
1111+-
111++1-
++1111
11+++-1
+1++1-1
-1++11+
+111+-1
11+-++1
-1+1++1
11+1+-1
++1+11
-11+1+1
11-+1+1
111+++
+-1+++1

PA.: GOING DOWN UP IN TEN

THE NUMBER OF OLD CLAUSES DELETED IS 1

THE NUMBER OF NEW CLAUSES DELETED IS 2

THE NEW CLAUSES AFTER REDUCTION ARE
11+-+11
-1+1++1
11+1+-1
++1+11
-11+1+1
11+-+1+1

TIME
18.56666667

NEXT PAGE IS A NEW SESSION, CONTINUING THIS
EXAMPLE.
\[ \text{TIME} \]
\[ 0.06666666667 \]
\[ \text{\texttt{\textbf{\smaller I WILL CARRY OUT THE STEPS OF PROD TO SHOW TIMES INVOLVED IN EACH STEP. \}}}} \]
\[ \text{TIME} \]
\[ 0.3 \]
\[ \text{\texttt{\textbf{\smaller PREPEXPR}}} \]
\[ \text{\texttt{\textbf{\smaller TIME}}} \]
\[ 0.48333333333 \]
\[ (\text{\texttt{\textbf{\smaller PLUSEXPR}}})[2] \]
\[ 3 \]
\[ (\text{\texttt{\textbf{\smaller MINUSEXPR}}})[2] \]
\[ 3 \]
\[ (\text{\texttt{\textbf{\smaller ONEEXPR}}})[2] \]
\[ 17 \]
\[ \text{\texttt{\textbf{\smaller PRODUCT}}} \]
\[ \text{\texttt{\textbf{\smaller TIME}}} \]
\[ 0.88333333333 \]
\[ \text{\texttt{\textbf{\smaller EXPR}}} \]
\[ 2 \]
\[ 23 \]
\[ 6 \]
\[ \text{\texttt{\textbf{\smaller INTERPEXPR}}} \]
\[ -++111 \]
\[ 1+++11 \]
\[ -11+++1 \]
\[ 11-++1 \]
\[ 1-11++ \]
\[ 111-++ \]
\[ 111+-+ \]
\[ 111+++ \]
\[ ++1111 \]
\[ +1++1- \]
\[ +111+- \]
\[ 11+-+1 \]
\[ -1+1++ \]
\[ 11+1+- \]
\[ ++-+1+ \]
\[ -11+1+ \]
\[ 11+-+1+ \]
\[ +-111+ \]
\[ ++-11+ \]
\[ 1+-+11 \]
\[ -++++1 \]
\[ +++++1 \]
\[ -1+++1 \]
\[ \text{\texttt{\textbf{\smaller TIME}}} \]
\[ 1.05 \]
\[ \text{\texttt{\textbf{\smaller REDUCTION}}} \]
\[ \text{\texttt{\textbf{\smaller THE NUMBER OF OLD CLAUSES DELETED IS 1}}} \]
\[ \text{\texttt{\textbf{\smaller THE NUMBER OF NEW CLAUSES DELETED IS 3}}} \]
\[ \text{\texttt{\textbf{\smaller THE NEW CLAUSES AFTER REDUCTION ARE}}} \]
\[ +111+ \]
\[ 1+++11 \]
\[ ++1+1 \]

The document continues on the next page.
(continued)

TIME

3.9

NOTICE THAT MOST TIME IS USED BY REDUCTION.

PROD

THE VARIABLE TO BE ELIMINATED IS 5

OCCURRENCES OF +: 10; -: 1; 1: 0

BEFORE REDUCTION, EXPR HAS 13 CLAUSES IN 5 VARIABLES:

-11++
1-++1
++111
+1++-
-11++
11-++
++-11+
11-++
-11++
11-++
1-1++
-11++

THE NUMBER OF OLD CLAUSES DELETED IS 2

THE NUMBER OF NEW CLAUSES DELETED IS 2

THE NEW CLAUSES AFTER REDUCTION ARE

-11++
11-++
1-1++

TIME

6.2

PROD

THE VARIABLE TO BE ELIMINATED IS 4

OCCURRENCES OF +: 6; -: 0; 1: 3

BEFORE REDUCTION, EXPR HAS 3 CLAUSES IN 4 VARIABLES:

-++1
++11
+-1+

NO NEW CLAUSES WERE FORMED.

PROD

THE VARIABLE TO BE ELIMINATED IS 3

OCCURRENCES OF +: 1; -: 0; 1: 2

BEFORE REDUCTION, EXPR HAS 2 CLAUSES IN 3 VARIABLES:

++1
+-+

NO NEW CLAUSES WERE FORMED.

PROD

THE VARIABLE TO BE ELIMINATED IS 1

OCCURRENCES OF +: 2; -: 0; 1: 0

BEFORE REDUCTION, EXPR HAS 0 CLAUSES IN 2 VARIABLES:

NO NEW CLAUSES WERE FORMED.

TIME

7.266666667

NOT A TAUTOLOGY.
Bibliography


Reiter, Raymond, (bibliography to an article), JACM Vol. 18, No. 4 (Oct., 1971), pp. 645-646.


Abbreviations:
ACM--Association for Computing Machinery.
JACM--Journal of the ACM.