

A COMPUTATIONAL STRUCTURE FOR THE PROPOSITIONAL CALCULUS

M. J. Shensa*

code 632

Naval Ocean Systems Center
San Diego, CA 92152-5000

Abstract

Despite the long history and descriptive simplicity of the propositional calculus, practical aspects of its implementation on the computer remain an active area of study. We propose a representation of the propositional calculus which is straightforward, yet compact and sufficiently flexible to circumvent the combinatorial difficulties posed by many problems. The methodology does not involve term-rewriting, production systems, or similar, symbol-oriented approaches, but, rather, relies on a non-canonical representation in disjunctive normal form. It has been implemented in PASCAL, and efficiently solves what are described in the literature as "difficult" problems. The approach, which is simple and highly structured, and perhaps more procedural than many others, may be recommended for its ease of interpretation and for incorporation as a tool in larger systems.

1 Introduction

Despite the long history and descriptive simplicity of the propositional calculus, practical aspects of its implementation on the computer remain an active area of study ([1] - [8]). Boolean logic has always been a powerful tool in its own right, and, more recently, boolean modules have come to play an integral role in constraint logic programming ([2], [4], [6], [7]). The methodology described in this paper differs from these others; it does not involve term-rewriting ([3], [5]), production systems ([7]), or similar, symbol-oriented approaches ([1]). Rather, it is somewhat of a throwback, relying on a representation in disjunctive normal form. However, the representation is not canonical (not unique), and, in exchange, it possesses a flexibility which overcomes many of the combinatorial problems associated with truth tables and such techniques (c.f., [8]). The algorithm, which has been implemented in PASCAL, successfully solves what are

described in [7] to be "difficult" problems.¹ Furthermore, its highly structured nature lends itself well to maintaining consistency and, it is hoped, also to tracing reasoning.

Before proceeding further, let us be a bit more specific as to the type and context of the problems which we expect to solve. We visualize a universe of discourse consisting of a set of boolean variables (predicate letters) A, B, C , etc., each with a range of values $\{0, 1\}$. A proposition P may be input by writing a formula employing letters, or variable names, and the connectives $\sim, \wedge, \vee, \Rightarrow$, and $=$ in the standard fashion of the propositional calculus. Such a formula is viewed, in the representation described below, as a constraint on the values of the variables. Having been given a set, $\{P_i\}$, of propositions (that is, a knowledge base or the premise of a theorem), we shall be interested in

- (a) Reducing the knowledge base: that is, providing a succinct representation of the (set of) variable values for which $P = P_1 \wedge P_2 \wedge \dots$ holds.
- (b) Answering queries Q : Does $P \Rightarrow Q$? Does $P \Rightarrow \sim Q$?
- (c) Solving Boolean equations: For example, given $P = P(A, B, C)$, solve for A in terms of B and C ([9]). We shall not treat this topic in this paper, but we note that it involves the task of describing the constraint set of (a) in a particular manner.

In short, the critical issues are the choice of a representation for a set of propositions (i.e., the knowledge base), and the ease with which it lends itself to the subsequent extraction of information, namely the answering of queries and the expression of the constraints relating variables.

A fairly naive, but reasonable, first approach to implementing these concepts might proceed as follows: One assigns a binary digit to each boolean

*This work was funded by code 221, NUSC, Newport, RI under ONT Code 23 program element 62314N.

¹A preliminary version solves several combinatorially difficult problems in times comparable to the boolean module for Prolog III as cited in [7]. For example the salt and mustard problem of Lewis Carroll (for which, according to [7], a solution by saturation would generate 35000 clauses) takes about 1.5 seconds on an Apollo 3550 workstation.

variable so that, for example, in a universe of three variables the elementary conjunction ABC (A and B and not C) is internally represented by the three digit binary word 110. A disjunction ("or") of elementary terms is represented by a list of words, and boolean manipulations of these expressions are accomplished by applying bitwise "and" to pairs of words and concatenating lists. However, a moment's reflection reveals that, even for relatively simple problems, the combinatorial burden can become enormous. For example, in the above context, the simple proposition A (i.e., A is true) would require four terms for its representation, $A = ABC \vee \bar{A}BC \vee ABC\bar{B} \vee ABC\bar{C}$, while in a universe of N variables it would result in 2^{N-1} terms. In contrast, one's intuitive notion of the proposition A might read: (a) $A = 1$ and (b) the other variables are "don't care". We propose to formalize this description by introducing a new symbol $*$ which means "any", i.e., 0 or 1. The proposition A then takes the more succinct form $\underline{1}**$ (or N $*$'s if there are N variables). The "1" is underlined because it no longer represents a single binary digit; each place in this representation must allow for at least three symbols, $\underline{0}$, $\underline{1}$, or $*$. Although not obvious, it turns out that this representation is sufficiently compact to overcome the combinatorial difficulties faced in solving most practical problems.

2 Representation

We introduce four symbols $\underline{E}$, $\underline{0}$, $\underline{1}$, and $*$ which, semantically, are intended to stand for "inconsistent (empty)", "false", "true", and "any (true or false)" with respect to a single boolean variable. More precisely, they represent sets of values which the variable may assume

$$\begin{aligned}\underline{E} &\triangleq \{\} \\ \underline{0} &\triangleq \{0\} \\ \underline{1} &\triangleq \{1\} \\ * &\triangleq \{0, 1\}.\end{aligned}\quad (1)$$

Implementation on a computer requires four numbers which we assign by means of two binary digits:

$$\begin{aligned}\underline{E} &= 00 \\ \underline{0} &= 01 \\ \underline{1} &= 10 \\ * &= 11.\end{aligned}\quad (2)$$

Some useful computational aspects of the representation (2) will be described later. For the moment, we note that it was specifically chosen to possess the following property: set intersections in (1) correspond to a bitwise "AND" of two-digit binary words in (2). That is,

$$\begin{aligned}\underline{0} \wedge \underline{0} &= \underline{0}; & \underline{1} \wedge \underline{1} &= \underline{1}; & \underline{0} \wedge \underline{1} &= \underline{E} \\ * \wedge \underline{0} &= \underline{0}; & * \wedge \underline{1} &= \underline{1}; & * \wedge * &= * \\ \underline{E} \wedge \underline{0} &= \underline{E}; & \underline{E} \wedge \underline{1} &= \underline{E}; & \underline{E} \wedge \underline{E} &= \underline{E}\end{aligned}\quad (3)$$

where $\wedge$ is taken to mean either set intersection or bitwise AND, depending on whether we are speaking of the interpretation, (1), or of the internal representation, (2). More precisely, let a two-bit computer word w_A represent the set of values of A and w_B those of B, then the set of values of $A \wedge B$ is represented by w_A AND w_B .

More generally, we shall be dealing with N variables, each represented by a "digit" of the form (2) giving a total of N quaternary digits (2N binary digits). As described in the introduction these digits are interpreted as a conjunction ("anding") of the corresponding variables. For example, with $N = 3$, the statement $AC \wedge B$ takes the form $\underline{1}\underline{1}\underline{1} \wedge * \underline{0} * = \underline{101}$. Denoting these words by lower case letters, we write general propositional expressions in the form $x_1 \vee x_2 \vee x_3 \dots$ and store them as lists of words x_i . This is simply a disjunctive normal form with conjunctions of variables (terms) represented by words, and a disjunction of terms stored as a list of those words. Thus, if P and Q are two propositions given by $P = \bigvee_i x_i \rightarrow$ list of x_i and $Q = \bigvee_j y_j \rightarrow$ list of y_j , then we have

$$P \wedge Q = \bigvee_{i,j} (x_i \wedge y_j) = \bigvee_{i,j} x_i y_j \rightarrow \text{list of } x_i y_j \quad (4a)$$

$$P \vee Q = (\bigvee_i x_i) \vee (\bigvee_j y_j) \rightarrow (\text{list of } x_i) \text{ concat } (\text{list of } y_j) \quad (4b)$$

where $x_i y_j$ is the bitwise AND of x_i and y_j , and the arrow stands for "is represented by."

The actual construction of a proposition in the computer is straightforward. Negation is passed through to predicate letters which are realized by

$$A_k \rightarrow * \dots * \underline{1} * \dots * \quad (5a)$$

$$\sim A_k \rightarrow * \dots * \underline{0} * \dots * \quad (5b)$$

where A_k is the k^{th} variable. The expression itself is built up, recursively, by performing $\wedge$ and $\vee$ operations on subexpressions according to (4a) and (4b). Equals and implication are replaced by an appropriate combination of $\sim$, $\wedge$, and $\vee$. There is also some merging of terms as described in the next section.

² One could use the symbol ϕ instead of $\underline{E}$ to reflect that this is the null set; however, in (1), $\underline{E}$ is visualized as one of four symbols describing the constraints put on the range of a single boolean variable. We reserve for ϕ the more general meaning of the null set of values in the (vector space) of all the variables.

Let us recapitulate in the form of several remarks:

i) An expression in the above representation may be viewed on three levels: as a proposition involving predicate variables, as the set of possible values assumed by the variables when they satisfy the proposition, or as a list of words with binary (alternatively quaternary) digits. For example, a single variable A , under no constraints is represented by $\underline{*} = \{0, 1\} = \underline{0} \cup \underline{1} = \neg A \vee A$. In a world of two variables this becomes $**$. If we add to that world the proposition $A = 1$, that is constrain ourselves to the set $\underline{1*}$, then we get $** \wedge \underline{1*} = \underline{1*}$. Next, asserting $B = 0$ we get $AB = \underline{1*} \wedge \underline{*0} = \underline{10}$. Finally, if we insist that also $A = 0$, we have $ABA = \underline{10} \wedge \underline{0*} = \underline{E0} = \phi$, the empty set. That is, we have a contradiction. As propositions (constraints) are added, we get nested, decreasing sets of variable values.

ii) Adding new variables is no problem since one universe may be embedded in a larger one simply by adding digits with their values set to $*$.

iii) The reader may have noticed that we did not supply a set of relations similar to (3) for the operator $\vee$. One can do this most easily by taking its dual; namely, transforming by $\underline{E} \rightarrow *$, $\underline{0} \rightarrow \underline{1}$, $\underline{1} \rightarrow \underline{0}$, $* \rightarrow \underline{E}$, and $\wedge \rightarrow \vee$. However, in the above representation, the resulting relations are not particularly useful since they do not extend to words: whereas the conjunction of two words $x \wedge y$ is given by the conjunction of its digits, the disjunction $x \vee y$ does not equal the disjunction of its digits. That is, $\underline{0*} \vee \underline{*1} \neq **$ even though $\underline{0V*} = *$ and $\underline{1V*} = *$.

iv) Nevertheless, it is worth noting that $\underline{0V1} = *$. In fact, it is quite natural to speak of set inclusion; e.g., $\underline{0} \subseteq *$, $\underline{1} \subseteq *$, etc. These relations play a role in reducing the number of terms in an expression.

v) Our representation for a set of values (i.e., for a proposition) is not unique. For example $A \vee B$ is given by $\underline{1*V*1} = \underline{10V*1} = \underline{11V*0}$.

vi) A word containing at least one digit $\underline{E}$ implies the null set; i.e., that there is no set of values for the variables in the universe of discourse satisfying the constraints of that word. Said differently, if the set of possible values of at least one component of a vector is empty, then the set of values (i.e., of N -tuples) for that vector must also be empty. Words which are ϕ are irrelevant when they appear in a disjunction, so they are simply omitted in the corresponding list. In symbols, $x \vee \phi = x$. If all the words in an expression reduce to ϕ , then that expression is the empty list, and the proposition(s) which it represents contains a contradiction. For example, $(A \vee B) \wedge \neg B$ becomes $(\underline{1*V*1}) \wedge \underline{*0} = \underline{10V1E} = \underline{10V\phi} = \underline{10} = A(-B)$, while $A \wedge \neg B \wedge B = \underline{1*} \wedge \underline{*1} \wedge \underline{*0} = \underline{1E} = \phi$.

³ This is a consequence of using words to represent conjunctions (intersections) of variables (values). The alternative would be words as disjunctions and lists as conjunctions which would lead to a conjunctive normal form.

3 Merging

Except for the introduction of $*$ and $\underline{E}$, our representation is fairly trivial. The outstanding question is whether it is sufficiently powerful to overcome the combinatorial difficulties that beset many practical problems. (Since the general problem is NP-complete, it is in grave doubt whether these difficulties can ever be completely mastered [10].) The answer, as indicated in the introduction, is a qualified "yes". It does, however, seem necessary to remove null words and to include the simple merging operation described below.

When they occur, combinatorial problems exhibit themselves in the length of the "OR" lists. For example, ANDing a list of m words with one of n words has the potential for producing a list of mn words. Some products will be ϕ and, hence, disappear, but, at the very least, we must also remove duplications of words and, preferably, perform other reductions as well. It turns out that for most problems it is sufficient to recognize set inclusions; i.e., if $x \subseteq y$, then one replaces $x \vee y$ with y . This process is straightforward. In fact, letting $x^{(s)}$ denote the s^{th} digit of x , we have

$$x \subseteq y \Leftrightarrow \{y^{(s)} = * \text{ or } x^{(s)} \text{ for all } s\} \quad (6)$$

The implementation of (6) is inexpensive (see section 4), and the combinatorial savings are enormous. This is true even though a list of n words can require up to $n(n+1)$ comparisons to check for all inclusions.⁴ In fact, experience indicates that testing for inclusion is critical in large problems.

Obviously, many other merging operations are possible. For example, one may consider combining words which differ in a single digit; e.g., $\underline{01*1V00*1} = \underline{0**1}$. This computation is only slightly more demanding than inclusion; however, if one wishes to carry the reduction as far as possible, repeated passes on expressions are necessary. More significantly, limited testing seems to indicate that such combining is not particularly beneficial; the increased expense outweighs combinatorial benefits. Still, it does seem reasonable to employ this procedure at strategic points; for example, as the final operation in incorporating a knowledge base.

4 Implications and Queries

A basic goal of our system is to prove theorems $P \Rightarrow Q$, or, equivalently, given a knowledge base P , to verify a query Q . Proving an implication or establishing a proposition is tantamount to showing that it holds for all values of its variables. More precisely, $P \Rightarrow Q$ if and only if the set of values $QV-P$ is the entire universe. The non-uniqueness of our representation makes direct verification difficult, but this causes no inconvenience

⁴ Note that repeated passes are not necessary since one only deletes redundant words, never creating new ones.

since we may employ the equivalent statement $\neg Q \wedge P = \phi$. Often P will be a knowledge base which has already been input (and reduced). In that case various queries Q can be checked by forming $\neg Q$, intersecting it with P , and checking whether the resulting expression is ϕ . More generally, we can also intersect P with Q , to obtain one of three conclusions, " Q follows", " $\neg Q$ follows", or " Q may neither be proved true nor false from the data P ." Such a system evaluates expressions as logically valid, logically false, or satisfiable but not valid; thus, negation as failure never becomes an issue.

There is a projection procedure which allows one to consider only the relevant variables in proving a proposition and which has the potential to speed up processing, particularly in the presence of a large number of variables. We note that if x contains a variable not appearing in y (i.e., whose digit in y is $*$), then the intersection of the two digits is not $\underline{E}$ and, hence, may be omitted in the determination of whether $x \wedge y = \phi$. Consequently, one would expect to simplify the proof of $P \Rightarrow Q$ by projecting P and Q onto their common variables. This removes irrelevant constraints and, hopefully, a number of terms. In the presence of a large number of variables requiring several computer words for their representation, computational gains at a more basic level may also be achieved.

Formalizing the above suggestion, we define the varset of a word x to be the set of variables active in (constrained by) x and denote it by

$$V_x \triangleq \{s \mid x^{(s)} \neq *\}. \quad (7)$$

(Note that the null set ϕ , conceived as a word containing all $\underline{E}$'s, has all its variables constrained; each variable's set of values is empty.) Next, the varset of an expression is defined as the union of the varsets of its words

$$V_P \triangleq \bigcup_i V_{x_i} \quad \text{where } P = \bigvee_i x_i \quad (8)$$

The projection of a proper (non-null) word x on a varset V is defined by

$$(\Pi_V x)^{(s)} \triangleq \begin{cases} x^{(s)} & \text{for } s \in V \\ * & \text{otherwise} \end{cases}, \quad (9)$$

that is, it retains the constraints only on those variables which are in V . The others become unconstrained; they do not appear when the expression is written as a function of predicate letters. This is best illustrated by an example. Let $x = AC = \underline{1*0}$. Then the active variables in x are A and C , and $V_x = \{1, 3\}$. Now, suppose that we wish to project x on the varset $V_C = \{3\}$, we have $\Pi_{V_C} AC = \Pi_{\{3\}}(\underline{1*0}) = \underline{**0} = \bar{C}$. Proceeding to propositions, we simplify things by supposing that the words in an expression are always proper; i.e., all words equivalent to ϕ are removed as they occur.⁵ We then

define the projection of a proposition $P = \bigvee_i x_i$ onto the varset V by

$$\Pi_V P \triangleq \begin{cases} \bigvee_i \Pi_V(x_i) & P \neq \phi \\ \phi & P = \phi \end{cases} \quad (10)$$

We remark that Π is a true projection operator (idempotent) and has the property $\Pi_1 \Pi_2 = \Pi_2 \Pi_1$. Also, several useful relationships hold: $V_Q = V_{\neg Q}$, $\Pi_Q P = \Pi_{Q \wedge P}$, etc.

Finally, we incorporate these concepts into a procedure for testing whether $P \wedge Q = \phi$: (a) Determine the varsets of P and Q , and intersect them to form $V \triangleq V_P \cap V_Q$.⁶ (b) Project P and Q onto V obtaining $P' = \Pi_V P$ and $Q' = \Pi_V Q$. (c) Check whether $P' \wedge Q' = \phi$. (a), (b), and (c) may be formally justified by the following argument: From (7) and (9), with $x \neq \phi$ and $y \neq \phi$, we have $xy = \phi$ if and only if $\exists s \text{ s.t. } x^{(s)} y^{(s)} = \phi$ which implies $x^{(s)} \neq *$ and $y^{(s)} \neq *$ which is true if and only if $s \in V_x \cap V_y$. The remainder follows from (1).

Aside from reducing the number of variables in a proof, projections may be used in their own right. An interesting example lies in their application to the so-called fields of production which were introduced in [7]. To get all statements Q which follow from P and which only contain certain variables or contain a particular number of variables (e.g., at most three predicate letters), one simply projects P onto the desired variable space. Other methods for speeding up deductions, such as the special treatment of binary relations between primitives (e.g., $A \Rightarrow B$ and $A = B$), have not yet been pursued; however, the bare skeleton, just with inclusion and no projections, seems adequate for solving many, or even most, practical problems efficiently.

5 Software Notes

It is clearly important that the various operations described in this paper be efficiently implemented. One of the most frequent, but less obvious, is testing a word for the presence of the digit $\underline{E}$. To facilitate this test our software employs two words of N bits each rather than one word of $2N$ bits to represent N variables. We designate these words by x^o and x^e to indicate the odd and even digits of x . Under such an implementation, one has

$$x \neq \phi \text{ if and only if } (x^e \text{ OR } x^o) = 111\dots 1, \quad (11)$$

⁵ Such a scheme is advisable since it results in much more efficient computation. By $P = \phi$ we shall mean the empty list. Note that a proposition reduces to ϕ if and only if it contains a contradiction.

⁶ Note that if $V = \phi$, then $P \neq Q$ and $P \neq \neg Q$.

where OR is bitwise "or". This representation is also useful for projection. A varset is implemented as a single binary word ν of N bits with a bit of 1 indicating the presence of a variable. The even and odd parts of the projection of x onto ν are then given by $(x^o \text{ OR } \sim \nu)$ and $(x^e \text{ OR } \sim \nu)$ respectively. Finally, recognizing an inclusion is straightforward: $x \subseteq y$ if and only if $(x \text{ OR } y) = y$; i.e., this relation must hold for the even and odd parts of x and y respectively.

6 Conclusion

We have presented a practical, structured representation for the propositional calculus. The notation is such that a given expression may be interpreted as a symbolic formula, as a union of sets, or as a list of binary (quaternary) words implemented on a computer. The workability of our methodology, however, is empirical. It springs from the fact that, despite its naiveté, the implementation is sufficiently powerful to circumvent the combinatorial difficulties posed by many problems. It is felt that this approach, which is simple and highly structured, and perhaps more procedural than many others, may be recommended for its ease of interpretation and for incorporation as a tool in larger systems.

References

- [1] Buttner, W. and Simonis, H., "Embedding Boolean Expressions into Logic Programming," *J. Symbolic Comp.*, 1987, 4, pp 191-205.
- [2] Colmerauer, A., "Opening the Prolog III Universe," *BYTE Magazine*, August, 1987, pp 177-182.
- [3] Dershowitz, N., Hsiang, J., Josephson, N., Plaisted, D., "Associative-Commutative Rewriting," *Proc. 8th IJCAI*, 1983, pp 940-944.
- [4] Dincbas, M., Simonis, H., Van Hentenryck, P., "Solving a Cutting-Stock Problem in Constraint Logic Programming," *Proc. 5th Int. Conf. Logic Programming*, Seattle, 1988.
- [5] Hsiang, J., "Refutational Theorem Proving using Term-Rewriting Systems," *Artificial Intelligence*, 1985, 25, pp 255-300.
- [6] Lassez, C., "Constraint Logic Programming," *BYTE Magazine*, August, 1987, pp 171-176.
- [7] Siegel, P., "Représentation et utilisation de la connaissances en calcul propositionnel," *Doctoral Thesis*, University Aix-Marseille II, July, 1987.
- [8] Turksen, I., "The set of Solutions for Boolean Equations," *Discrete Mathematics*, 1973, 5, pp 261-282.
- [9] Rudeanu, S., *Boolean Functions and Equations*, North Holland, 1974.
- [10] Garey, M. and Johnson, D., *Computers and Intractability*, W. H. Freeman and Co., 1979.