Solving Satisfiability Problems with Less Searching

by

Paul W. Purdom, Jr.

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT NO. 117

SOLVING SATISFIABILITY PROBLEMS WITH LESS SEARCHING

Paul W. Purdom, Jr.

Revised July 1982

*Research reported herein was supported in part by the National Science Foundation under grant number MCS 7908110.

Abstract

A new technique, complement searching, is given for reducing the amount of searching required to solve satisfiability (constraint satisfaction) problems. Search trees for these problems often contain subtrees that have approximately the same shape. When this occurs, knowledge that the first subtree does not have a solution can be used to reduce the searching in the second subtree. Only the part of the second subtree which is different from the first needs to be searched. The pure literal rule of the Davis-Putnam procedure is a special case of complement searching. The new technique greatly reduces the amount of searching required to solve conjunctive normal form predicates that contain almost pure literals (literals with a small number of occurrences).

1 Introduction

The generalized satisfiability problem is to determine whether there exists any assignment of values to variables such that a predicate with the following form evaluates to *true*:

$$P \equiv \bigwedge_{1 \le i \le t} R_i(x_{i1}, \ldots, x_{ij_i}) \tag{1}$$

where the variables x_{ik} are elements of a finite set $\{x_j \mid 1 \le j \le v\}$, each R_i is a relation that depends on some of the variables, and the values for each x_j are elements of a corresponding finite set Y_j. The set of problems with form (1) is obviously NP-complete. Garey and Johnson [6] give many interesting and important problems with this form.

Two special cases of (1) are conjunctive normal form (CNF) predicates where each R_i is a clause and constraint satisfaction problems [9] where each R_i depends on only two variables. These special cases are as difficult as the general problem; they have the advantage that certain algorithms are easier to state for the special cases. It is usually straightforward (but at times tedious) to reexpress such algorithms for the general case.

Two basic approaches have been developed for solving problems with form (1). The first approach is called searching. In searching, some variable (called the splitting variable) is selected, and a subproblem is formed for each value of the variable. Each subproblem is formed by using the corresponding values of the splitting variable to simplify each R_i that depends on the splitting variable. (For example, to simplify an R_i which is a clause, if R_i contains a *true* literal associated with the value of the splitting variable, then R_i is replaced with *true*, while if it contains a *false* literal, then R_i is replaced with the clause that results from dropping the *false* literal.) The original problem has a solution if and only if one or more of the subproblems has a solution. The version of the Davis-Putnam procedure in [3] is an example of a searching algorithm.

The second basic approach to solving problems of form (1) is to replace the relations by an equivalent set of relations. If repetition of this process leads to the constant relation *false*, then the original problem does not have a solution. The original version of the Davis-Putnam procedure [4] and resolution [14] use this approach. Bibel [1] has a unified presentation of several recent advances in resolution type methods.

The method developed in this paper, which I call complement searching, combines these two approaches. Complement searching is a searching method in which all the subproblems (except the first) are modified to avoid researching some parts of the subproblem which are known to be the same as the corresponding parts of previous subproblems. Complement searching is a generalization of the pure literal rule [3].

The pure literal rule says, in effect, that if some value for a variable makes all of the relations that depend on the variable simplify to *true* (such a value is called a safe value), then one does not need to consider any other values for that variable. If the problem has a solution then it has a solution in which the variable is assigned its safe value. This paper develops the following generalization of the pure literal rule: if some value for a variable makes almost all of the relations that depend on the variable simplify to *true*, then one can greatly reduce the amount of searching needed to investigate the remaining values for the variable. The general insight that leads to this result has been noticed before [2,5]: the various subproblems generated by a searching algorithm are often similar. Here I use one aspect of the similarity to speed searching on problems with almost pure literals.

2 An Example

The techniques of this paper will be illustrated with the relations:

$$\begin{array}{lll} R_1 = a \vee \neg b \vee \neg c & R_6 = b \vee \neg c \vee d & R_{11} = \neg b \vee c \vee d \\ R_2 = \neg a \vee b \vee \neg e & R_7 = b \vee \neg c \vee e & R_{12} = \neg b \vee c \vee \neg d \\ R_3 = \neg a \vee \neg b \vee d & R_8 = b \vee \neg c \vee \neg e & R_{13} = \neg b \vee \neg d \vee \neg e \\ R_4 = \neg a \vee \neg b \vee \neg e & R_9 = b \vee \neg d \vee e & R_{14} = c \vee \neg d \vee \neg e \\ R_5 = b \vee c \vee d & R_{10} = b \vee \neg d \vee \neg e & R_{15} = \neg c \vee d \vee e \end{array} \quad (2)$$

For these relations problem (1) is satisfied with $a, b, c, d = \mathit{true}$ and $e = \mathit{false}$.

Consider solving (2) using a as the first splitting variable. Let Q_{i1} be the result of simplifying R_i by setting $a = \mathit{false}$ and let Q_{i2} be the corresponding result for $a = \mathit{true}$. Then

$$\begin{array}{llll} Q_{11} = \neg b \vee \neg c & Q_{21} = \mathit{true} & Q_{31} = \mathit{true} & Q_{41} = \mathit{true} \\ Q_{12} = \mathit{true} & Q_{22} = b \vee \neg e & Q_{32} = \neg b \vee d & Q_{42} = \neg b \vee \neg e \end{array} \quad (3)$$

$$Q_{i1} = Q_{i2} = R_i \text{ for } 5 \leq i \leq 15.$$

The search algorithm reduces problem (2) to determining whether $\bigwedge_i Q_{i1}$ or $\bigwedge_i Q_{i2}$ is satisfiable. Figure 1 shows the search tree that results for problem (2) when the splitting variables are selected in alphabetical order. Each non-solution leaf node is labelled with the R_i of smallest index which has terminated the search by simplifying to *false*.

The following parts of this paper explain why there is no need to search the dashed and dotted portions of the tree in Fig. 1. It will be shown that, since $R_1 = a \vee \neg b \vee \neg c$ is the only clause with the literal a, and since the $a = \mathit{false}$ branch does not contain a solution, the $a = \mathit{true}$ branch does not contain any solution with $b = \mathit{false}$ or $c = \mathit{false}$. Also, using R_5 and R_{14} one can conclude that the dotted part of the tree contains no solutions. This type of observation can greatly reduce the amount of searching that is required to solve some large problems.

3 Searching

The *ordinary search method* for determining whether a predicate P in form (1) is satisfiable is:

1. If P is obviously satisfiable (all $R_i \equiv true$) or if P is obviously unsatisfiable (some $R_i \equiv false$) then stop.
2. Select a splitting variable. For each value of the splitting variable form a subproblem by simplifying P. Simplifying P consists of replacing each R_i with R'_i where R'_i is R_i restricted by assigning the chosen value to the splitting variable.
3. Solve the subproblems recursively. If one or more of the subproblems is satisfiable, then the original problem is satisfiable; otherwise it is not. (It is possible to stop this step as soon as one satisfiable subproblem is found.)

The following definitions are useful:

x_j = the splitting variable.

Y_j = the set of possible values for x_j.

y_{jr} = the rth element in Y_j for some ordering of Y_j.

k = the number of elements in Y_j.

$Q_{ir} = R_i$ simplified by replacing x_j with its value y_{jr}.

R_i *depends on* x_j if and only if $Q_{ir} \not\equiv Q_{is}$ for some r and s.

$S = \{i \mid R_i \text{ depends on } x_j\}$.

$P_0 = \bigwedge_{i \notin S} R_i$.

$P_r = \bigvee_{i \in S} Q_{ir}$ for $1 \leq r \leq k$.

Notice that the arguments to functions denoted by capital letters P, Q and R are often suppressed in this notation. Also, the definitions depend on the choice of x_j and the ordering of Y_j. Moreover, a different ordering of Y_j may be used each time x_j is used for splitting.

To illustrate the definitions consider problem (2) with splitting variable a and ordering (*false*, *true*). Then the Q_{ir} are given by (3), $S = \{1, 2, 3, 4\}$, $P_0 = \bigwedge_{5 \leq i \leq 15} R_i$, $P_1 = Q_{11}$, and $P_2 = Q_{22} \wedge Q_{32} \wedge Q_{42}$.

Searching tests whether P is satisfiable by testing each disjunct in

$$\bigvee_{1 \leq n \leq k} (P_0 \wedge P_n). \tag{4}$$

Step (2) of searching computes the simplification of P for each value y_{jr} of x_j; the simplification is $P_0 \wedge P_r$. Step (3) of searching *or*s the result from the subproblems. Stopping at the first solution corresponds to using a short-circuit evaluation of the *or* in (4).

4 Complement Searching

First I will give the mathematical basis for complement searching.

Theorem 1.

$$X \vee (A \wedge B) \vee (A \wedge C) = X \vee (A \wedge B) \vee (A \wedge \neg B \wedge C) \tag{5}$$

Proof: The left and right side have the same truth table. ∎

When X, A, B and C are relations (with unspecified arguments) Theorem 1 says informally that if the $A \wedge B$ part does not lead to a solution then in the $A \wedge C$ part one can ignore the possibility that B is *true*.

Theorem 2.

$$\bigvee_{1 \le r \le k} (P_0 \wedge P_r) = \bigvee_{1 \le r \le k} [(P_0 \wedge P_r) \wedge (\bigwedge_{n \in I_r} \neg P_n)] \tag{6}$$

where I_r is any subset of the integers from 1 to $r-1$. (The conjunction $\bigwedge_{n \in I_r} \neg P_n$ is *true* if I_r is the empty set.)

Proof: Apply Theorem 1 repeatedly to (4). First use Theorem 1 to convert

$$[\bigvee_{1 \le r \le k-1} (P_0 \wedge P_r)] \vee (P_0 \wedge P_k)]$$

to

$$[\bigvee_{1 \le r \le k-1} (P_0 \wedge P_r)] \vee [(P_0 \wedge P_k) \wedge (\bigwedge_{n \in I_r} \neg P_n)].$$

Continue in like manner for $r = k-1, k-2, \ldots, 1$. ∎

Theorem 2 shows how the idea of Theorem 1 can be applied repeatedly.

Corollary 3.

$$\bigvee_{1 \le r \le k} (P_0 \wedge P_r) = \bigvee_{1 \le r \le k} [(P_0 \wedge P_r) \wedge (\bigwedge_{1 \le n \le r-1} \neg P_n)] \tag{7}$$

.

The corollary is the special case of Theorem 2 that uses complement search to the fullest extent possible. However, complement search can result in extra overhead as well as reduced searching. Theorem 2 permits one to vary the extent to which complement search is applied by selection of the I_r, so that large overhead can be avoided (by using I_r = *empty*) in unfavorable situations.

The *complement search method* is the same as the ordinary search method given in section 3 except that step 2 is modified in the way suggested by Theorem 2. Whereas ordinary searching uses

$$P_0 \wedge P_r \tag{8}$$

as the r-th subproblem, complement searching uses

$$P_0 \wedge P_r \wedge (\bigwedge_{n \in I_r} \neg P_n) \tag{9}$$

with some set I_r. Under favorable conditions complement searching can be much faster than ordinary search because the extra conjuncts ($\bigwedge_{n \in I_r} \neg P_n$) in (9) (as compared to (8)) can eliminate much of the searching. In unfavorable cases little or no savings will result, and moreover substantial overhead may be needed to process ($\bigwedge_{n \in I_r} \neg P_n$). The overhead (along with all of the savings) can be avoided by using I_r = empty. To obtain a fast complement searching algorithm it is necessary to choose I_r wisely.

5 Efficient Complement Searching

Define the number of *unfavorable occurrences* of the splitting variable to be

$$\min_r \{\text{number of } i \text{ such that } i \in S \text{ and } Q_{ir} \not\equiv \mathit{true}\}. \tag{10}$$

Assume for the rest of this section that Y_j is ordered so that $r = 1$ gives the minimum value in (10).

Suppose the splitting variable has zero unfavorable occurrences. Then $P_1 \equiv \mathit{true}$. For $r \geq 2$ let each I_r contain the index one. Then $P_0 \wedge P_r \wedge (\bigwedge_{n \in I_r} \neg P_n) \equiv \mathit{false}$ for $r \geq 2$, so in this case complement searching reduces testing P to testing $P_0 \wedge P_1$ (which has one less variable than P). Using complement searching (with I_r formed as indicated) is extremely effective when the splitting variable has zero unfavorable occurrences. In fact, for this case, complement searching is reduced to the pure literal rule [3] for CNF problems.

For many sets of random CNF problems use of the pure literal rule can reduce the average search time from exponential in the number of variables to polynomial [8,12,13]. The pure literal rule is also an important component of the algorithm of Monien and Speckenmeyer [10], which has a good worst case time for CNF problems with three literals per clause. So previous work has established that complement searching is an important algorithm, at least in the special case where it coincides with the pure literal rule.

Suppose the splitting variable has one unfavorable occurrence. Then $P_1 = Q_{x1}$, where x is the index of the Q_{i1} that is not *true*. Again let each $I_j (j \geq 2)$ contain the index one. Suppose P_1 is *true* for most values of its variables. Then $\neg P_1$ is usually *false* and complement searching will operate rapidly on all but the first branch of the search tree. Moreover, $\neg P_1$ is easy to compute. If P_1 is *false* for most values, then complement searching (and also ordinary searching) is fast on the first branch. For splitting variables with one unfavorable occurrence, complement searching guarantees that a fast search is possible for at least one branch of the search tree. This generalizes the pure literal rule to almost pure literals—those that have one unfavorable occurrence.

An algorithm that uses complement searching whenever the splitting variable has no more than one unfavorable occurrence will clearly be fast whenever it is used on problems with many variables that have no more than one unfavorable occurrence. It will take about the same time as ordinary searching when used on problems with few such variables. Analytical studies should be done to determine how much better this version of complement searching is than the version that uses complement searching only when variables have no unfavorable occurrences.

As the number of unfavorable occurrences of the splitting variable increases, the advantages of using complement searching decrease because $\neg P_1$ becomes more complex. It is less likely to help reduce the search, while it is more time-consuming to process it (see section 6). Anyone using complement searching on large problems will probably want to investigate further the relation between the number of unfavorable occurrences of the splitting value and the use of complement search.

6 Clauses

Now let us consider complement searching when P is in CNF and when each $\neg P_n$ from (6) is expressed in CNF. In this case each R_i is the disjunction of a set of literals, and the definitions in section 3 simplify as follows:

$$S = \{i \mid R_i \text{ contains } x_j \text{ or } \neg\, x_j\}$$

$$Y_j = \begin{cases} \text{either} & (false, true) \\ \text{or} & (true, false) \end{cases}$$

$$Q_{ir} = \begin{cases} true & \text{if } R_i \text{ contains } x_j \text{ and } y_{jr} \text{ is } true \text{ or} \\ & \text{if } R_i \text{ contains } \neg\, x_j \text{ and } y_{jr} \text{ is } false, \text{ otherwise} \\ R_i & \text{with } x_j \text{ and } \neg\, x_j \text{ omitted.} \end{cases}$$

If Q_{ir} contains no literals, then it is *false*. When no Q_{ir} is *false*, the effort required to convert $\neg\, P_i$ to CNF is proportional to the length (in literals) of the Q_{ir} that are not equivalent to *true*.

Suppose some R_i contain the literal x_j and none contain $\neg\, x_j$. One then has no unfavorable occurrences of x_j, so (6) with $Y_j = (true, false)$ and $I_2 = \{1\}$ shows that there is no need to search for solutions with $x_j = false$. A similar savings occurs when no R_i contain x_j and one uses $Y_j = (false, true)$. In this case complement searching is the same as the pure literal rule.

Suppose one $R_i (i \in S)$, say R_1, contains x_j and some of the rest contain $\neg\, x_j$, i.e., there is one unfavorable occurrence of x_j. For the ordering $(false, true)$, $\neg\, P_1 = \neg\, Q_{11}$. If R_1 contains w literals, then $\neg\, P_1$ is the conjunction of $w - 1$ literals. For example, in (2) the literal a occurs only in $R_1 = a \vee b \vee c$ so $\neg\, P_1 = b \wedge c$. Thus, if one uses (6) with $I_2 = \{1\}$, on the $a = true$ branch of the search one can set $b = true$ and $c = true$, greatly speeding up the search of the $a = true$ branch. Notice that the larger w is, the more the search of the second branch is speeded up.

If there are z unfavorable occurrences of the splitting variable then $\neg\, P_1$ (when expressed in CNF) is the conjunction of a set of clauses, where each clause has z literals (unless some Q_{i1} is *false*, in which case $\neg\, P_1 = true$). The number of clauses in the set is equal to the product of the lengths of the Q_{i1} that are not equivalent to *true*. The time to convert $\neg\, P_1$ to CNF increases exponentially with z. When z is small (particularly for $z \leq 2$) complement searching can be done with modest overhead, and it can lead to great reductions in searching time. When z is large, large overhead results, and complement searching does not lead to significant savings on the searching. To produce an efficient algorithm it is therefore important to bypass complement searching (by setting I_r to the empty set) at those nodes where the splitting variable has a large number of unfavorable occurrences.

7 Relation to Worst Case Time

The work of Monien and Speckenmeyer [10] suggests that an algorithm with a fast worst-case time for CNF problems should combine several techniques. Their work also suggests that steps that generate lots of short clauses are good. Usually the splitting rule has this effect, but when the splitting variable has a small number of occurrences only a few shorter clauses are generated. For this reason they included the pure literal rule in their algorithm, so that the algorithm could make rapid progress even when all variables had only a few occurrences.

Complement searching was developed when I was trying, along with Edward Robertson and Cynthia Brown, to develop an improved algorithm for solving CNF problems. Complement searching appears to be particularly important for anyone trying to improve on the algorithm of Monien and Speckenmeyer, because it can be used to generate short clauses whenever the splitting variable has a small number of occurrences. It will be necessary, however, to do additional work on algorithm development and on algorithm analysis before one has an algorithm that is provably better than that of Monien and Speckenmeyer.

8 Conclusions

Rapid searching methods, such as the one in [10], use clever ideas both to limit which branches of the search tree need to be explored and to select which variable to use for splitting. Complement search is quite suitable for use with earlier methods. Since complement searching includes the pure literal rule as a special case, use of complement searching does not greatly complicate searching algorithms which already need the pure literal rule.

Complement searching is quite different from other techniques for speeding searching. The methods in [9,11] concentrate on discovering that there are no solutions at a node, rather than using the fact that one branch has no solutions to limit searching on a second branch. The method of Gaschnig [7] does use the fact that a failure is found to short-circuit part of a search, but it is less powerful than the method of Haralick and Elliot [9].

The algorithm of Monien and Speckenmeyer [10] contains a feature that is a generalization of the pure literal rule, but their generalization is quite different from mine. They search for safe combinations of values-combinations that result in *all* the clauses that depend on any variables in the combination evaluating to *true*. My generalization provides speed when a variable has an almost safe value—a value that makes most of the relations evaluate to *true*. Both generalizations are important for developing fast algorithms.

Acknowledgement

I wish to thank Doctors Edward Robertson and Cynthia Brown for the intellectual stimulation that led me to produce this algorithm. Also, their advice on various aspects of this paper is greatly appreciated.

References

[1] Wolfgang Bibel, *On Matrices with Connections*, JACM **28** (1981), pp. 633–645.

[2] D. G. Bobrow and B. Raphael, *New Programming Languages for Artificial Intelligence Research*, Comput. Surv. **6** (1974), pp. 153–174.

[3] Martin Davis, George Logemann, and Donald Loveland, *A Machine Program for Theorem-Proving*, CACM **5** (1962), pp. 394–397.

[4] Martin Davis and Hilary Putnam, *A Computing Procedure for Quantification Theory*, JACM **7** (1960), pp. 201–215.

[5] Eugene C. Freuder, *A Sufficient Condition for Backtrack-Free Search,* JACM **29** (1982), pp. 24–32.

[6] Michael R. Garey and David S. Johnson, *Computers and Intractability,* W. H. Freeman and Co., San Franciso (1979).

[7] John Gaschnig, *Performance Measurement and Analysis of Certain Search Algorithms,* Ph.D. Thesis, Carnegie-Mellon (1979).

[8] Allen Goldberg, Paul Walton Purdom Jr., and Cynthia A. Brown, *Average Time Analysis of Simplified Putnam-Davis Procedures,* Information Processing Letters (to appear).

[9] Robert M. Haralick and Gordon L. Elliot, *Increasing Tree Search Efficiency for Constraint Satisfaction Problems,* Artifical intelligence **14**(1980), pp. 263–313.

[10] Burkhard Monien and Ewald Speckenmeyer, *Three-Satifiablity is Testable in* $O(1.62^r)$ *Steps,* Theoretical Informatics Series, University of Paderborn (1979).

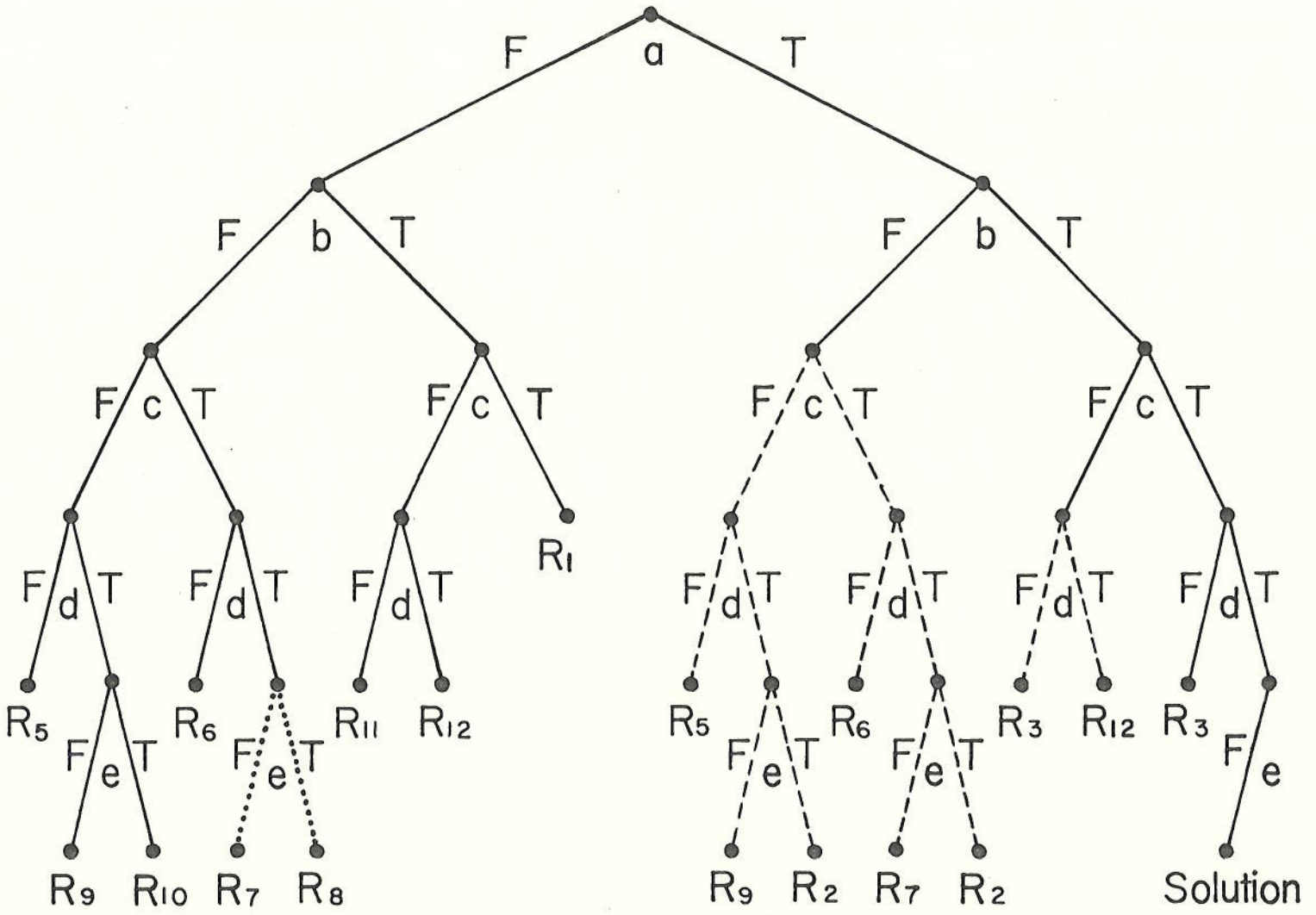
[11] Paul Walton Purdom Jr., Cynthia A. Brown, and Edward L. Robertson, *Backtracking with Multi-Level Dynamic Search Rearrangement,* Acta Informatica **15** (1981), pp. 99–113.

[12] Paul Walton Purdom Jr. and Cynthia A. Brown, *Evaluating Search Methods Analytically,* Proc. National Conference on Artificial Intelligence, (to appear).

[13] Paul Walton Purdom Jr. and Cynthia A. Brown, *The Pure Literal Rule and Polynomial Average Time,* Indiana University Computer Science Technical Report No. 128 (1982).

[14] J. A. Robinson, *A Machine-Oriented Logic Based on the Resolution Principle,* JACM **12** (1965), pp. 23–41.

Fig. 1. The backtrack tree for problem 2 in the text. There is no need to search the dashed part of the tree because of the lack of a solution on the $a = \mathit{false}$ branch implies that the $a = \mathit{true}$ branch does not have a solution when $b = \mathit{false}$ or $c = \mathit{false}$. Also there is no need to explore the dotted branches.


a
F
T
b
c
d
e
R1
R2
R3
R5
R6
R7
R8
R9
R10
R11
R12
Solution