ICOT Technical Report: TR-528

TR-528

A Pruning Condition for the Davis-Putnam Procedure

by
N. Helft

December, 1989

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Pruning Condition for the Davis-Putnam Procedure

Nicolas Helft
ICOT
1-4-28, Mita, Minato-ku, Tokyo 108, Japan.
Phone +813-456-4365 Email helft@icot.jp

November 20, 1989

Abstract

The Davis-Putnam procedure is the most investigated algorithm to test the satisfiability of propositional formulae. We uncover some inefficiencies of this algorithm, showing it can explore the same search space more than once. We then propose a simple modification of the procedure that avoids such unnecessary computation.

1 Introduction

This paper is concerned with SAT, the problem of determining the satisfiability of propositional formulae. This problem arises in many areas of computer science.

The Davis-Putnam Procedure (DP) [2] is the most investigated algorithm to achieve this task. The procedure attempts to find a model of the given formulae by actually constructing one, exploring a binary tree that results from alternatively assigning the truth values *true* or *false* to the propositional variables appearing in the formulae.

SAT is well known to be NP-complete [1]. Much work has been done on speeding up the algorithm, mainly by using heuristics to determine the order in which the propositional variables are assigned a truth value [7]. The result presented here is different, however. We first uncover some inefficiencies of DP by showing that it can explore the same search space more than once. We then propose a remedy to this situation, a simple condition under which some information obtained in a certain part of the search space can be transmitted to another part. This information avoids the unnecessary computation.

The results presented here are analogous to those discovered by Shostak [6] concerning some inefficiencies of linear resolution. The example used in this paper to illustrate the algorithm is taken from Shostak's paper.

The next section briefly introduces DP and shows a source of inefficiency. Section 3 introduces a modification of DP that overcomes this problem. This is illustrated with an example in Section 4. We then make some remarks on implementation, and survey related work.

2 Davis and Putnam's Procedure

This section briefly introduces the Davis-Putnam Procedure. [1] Given a formula in the propositional calculus, we are interested in knowing whether it has a model, that is an assignment of its propositional variables into the set $\{true, false\}$ such that the value of the formula under such assignment is *true*. The formula itself can be put in conjunctive normal form. Each of the conjuncts is then a *clause*, consisting of a disjunction of (negated or unnegated) atomic formulae called *literals*. These are called *positive* or *negative* according to their sign. We denote literals with lower-case letters l, $\neg p$, q, ... If a clause contains only one literal we call both the clause itself and the literal *unary*.

Definition Let C be a set of clauses and l a literal. Then $Transform(C, l)$ is the set obtained from C by performing the following operations.

1. Delete clauses containing l.
2. Delete the literal $\neg l$ [2] from clauses containing it.
3. Delete subsumed clauses [3] from the resulting set.#

For example, if $C = \{a \vee b \vee c, a \vee \neg b, d\}$, then $Transform(C, \neg b)$ is $\{a \vee c, d\}$.

The models of $Transform(C, l)$ are then a subset of those of C, the ones that assign *true* to l.

Let C be a set of clauses. The DP procedure is then the following algorithm.

Procedure $DP(C)$.

1. If C contains an empty clause then fail. If C is empty then exit with success.
2. Choose a literal l appearing in C and return
   $DP(Transform(C, l))$ OR $DP(Transform(C, \neg l))$.#

[1] Additional details can be found in Loveland's book [4].

[2] To be more precise, instead of $\neg l$ we should write *opposite*(l). This notation is simpler, however, and no confusion arises.

[3] We recall that a clause C is subsumed by a clause D if every literal of D appears in C.

In general, the algorithm needs to explore the two branches in step 2. There are, however, two well-known situations in which only one of these branches needs to be explored. If a unary clause $\{l\}$ occurs in C, then choosing $\neg l$ would immediately produce the empty clause; the algorithm can thus safely avoid this choice. If a literal l appears in C and its opposite $\neg l$ does not appear in C, (these literals are called *pure*) l can also be chosen deterministically.

So step 2 can be replaced by the following:

2. If there is a unary or pure literal l in C,
   then return $DP(Transform(C, l))$;
   else return $DP(Transform(C, l))$ OR $DP(Transform(C, \neg l))$.#

Figure 1 illustrates the algorithm. Clauses written as a juxtaposition of literals, with the "V" omitted.

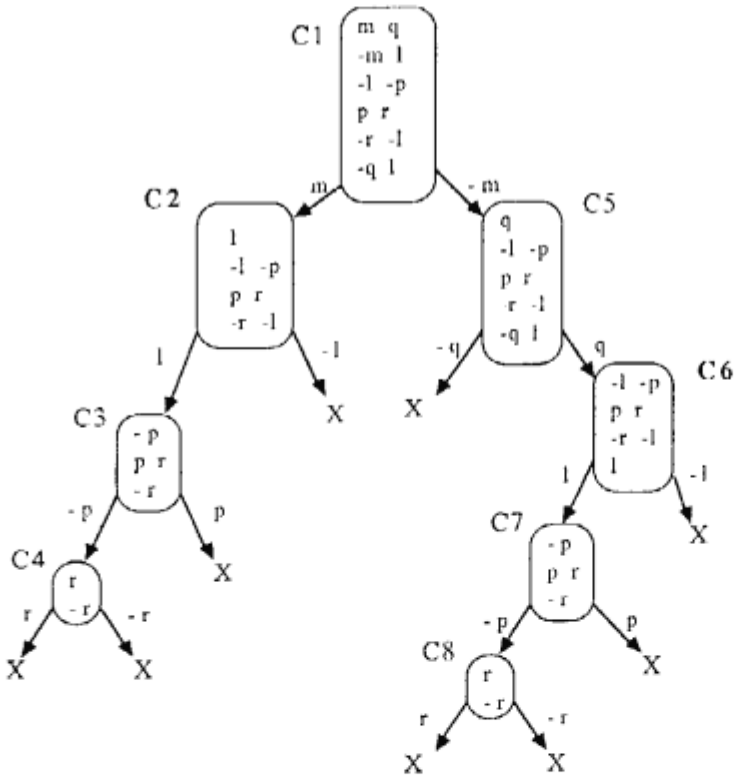

Figure 1: DP Procedure

In this example, observe that nodes $C2$ and $C6$ are exactly the same. Moreover, $C2$ was found to be inconsistent, and the information produced while proving so is lost after the algorithm backtracks to $C1$. The subtree rooted at $C6$ is then explored, producing the same search space as the one rooted at $C2$. Obviously, the example is kept small for understandability, but if $C2$ and $C6$ contained additional clauses, the tree explored twice could be much larger. The condition for this to happen, as in the example, is that the inconsistency of $C2$ should be independent of the choices made to generate it (in the example, choosing the literal m, that is, assigning *true* to the propositional variable m). The next section shows how we can easily transmit information produced while proving the inconsistency of a certain node to another one higher in the tree.

3 Main Result

The improvement over the the procedure described above follows from observations below:

Observation 1

Let C be a certain node of the DP search tree. If $DP(Transform(C,l))$ fails, then $C \models \neg l$.

Proof If $DP(Transform(C,l))$ fails, then no model of C assigns true to l; thus all models of C satisfy $\neg l$.#

Observation 2

If a set of clauses C implies a unary clause l, then the set obtained by removing from C all the clauses containing $\neg l$ also implies l.

Proof Call D the set obtained from C by removing clauses containing $\neg l$. We have to show that an arbitrary model M of D satisfies l.

Suppose not. Then M would satisfy $\neg l$, and thus all the clauses in $C - D$. As M satisfies D, if it satisfied $C - D$, it would satisfy C and thus l as well, a contradiction.#

These two results can now be used in the following way.

Application of the above observations: Let C be a node in the search tree, l the literal chosen to transform C, and D be C with clauses containing l removed. Let C' be a node higher in the tree, that is, an ancestor of C.

If $D \subseteq C'$ [4] and $DP(Transform(C,l))$ fails, then $C' \models \neg l$.#

If the condition applies, when the algorithm backtracks to C' to examine the remaining choice on it, the unary clause $\neg l$ can be added to it. This may simplify C' and avoid redundant computation.

An example will clarify this.

[4] $D \subseteq C'$ of course means that every clause of D belongs of C'. This should not be confused with saying that every clause of D is included in some of C'.

4 Example

Figure 2 shows nodes $C1$ and $C2$. The three clauses in a circle imply the literal $\neg l$.

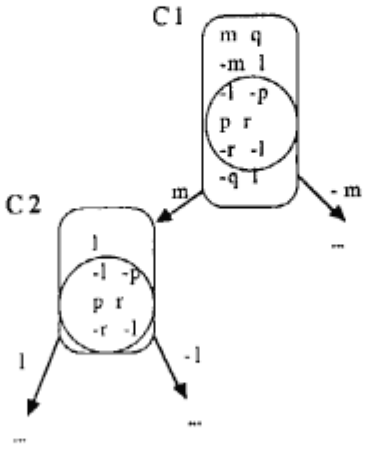

Figure 2: Nodes $C1$ and $C2$ imply $\neg l$

From Observation 1, we know that $C2 \not\models \neg l$ because the tree that results from choosing l fails. By Observation 2 we know that the unary clause l appearing in $C2$ has no responsability in such a proof: we can discard it and the resulting set still implies $\neg l$. Now, this resulting set appears in $C1$. Thus $C1 \not\models \neg l$ as well.

Now at node C1, literal $\neg l$ can be chosen deterministically, and the execution proceeds as indicated in Figure 3.

Thus only two inferences are needed instead of the four in the previous situation.

The reason is that the set of clauses in $C1$ was simplified by the derivation of a unary clause. This clause enables us to delete two clauses and reduce two others to unary clauses, that is, to produce deterministic choices. In general, a set of clauses can be greatly simplified by adding to it clauses implied by the set and that subsume other clauses of it. Much attention has been given to this problem in resolution-based theorem proving [4, 6]. The result presented here is based on the same motivation.

5 Implementation Note

A pruning condition such as the one proposed here must be cheap to implement, otherwise the cost may outweigh the benefits. The implementation of the test to verify the pruning condition

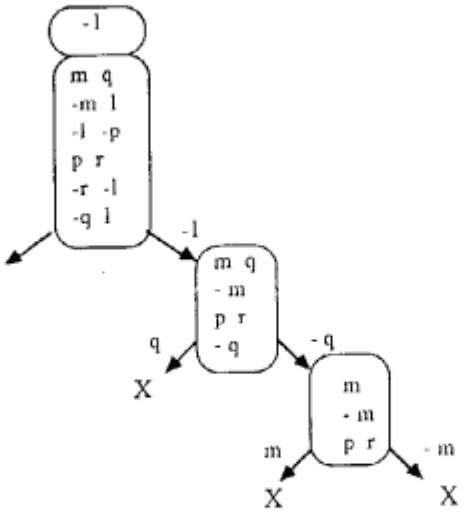

Figure 3: The Proof Once $\neg l$ has been added to $C1$

needs two operations in addition to the normal DP procedure.

1. Compute the set of clauses that result from deleting from a node those clauses that contain the chosen literal.

2. Test whether such a set is included in some ancestor.

The first condition comes with absolutely no cost: this operation has to be performed by DP in any case. Remember that new nodes are computed by deleting clauses containing the chosen literal and shortening the ones containing its opposite. The set we are looking for is the one that results from the deleting operation. We can do the inclusion test and then shorten the clauses containing the opposite of the chosen literal.

There is a cost to the second of the operations above, that of testing for inclusion. But there are well-known techniques for achieving this efficiently. In [3] it is proposed not to actually delete the clauses but rather to replace them by a marker. For example, in a Prolog implementation, a set of clauses can be represented as a list, and deleting a clause from the list is performed by replacing the clause by a free variable, thus leaving unchanged the actual size and order of the list. The inclusion test is then performed by a single unification. In [5], an efficient implementation for a boolean representation of propositional calculus formulae is described.

6 Related Work

6.1 Linear Resolution and the DP Procedure

The following is a linear refution of clause $m \vee q$, using background theory $\{\neg m \vee l, \neg l \vee \neg p, p \vee r, \neg r \vee \neg l, \neg q \vee l\}$, and the GC (graph-construction) procedure of Shostak [6].

The given clauses are the same as those in our example, and a slightly simplified version of those given by Shostak; this simplification keeps the example shorter while preserving the features needed to illustrate the comparison. We denote A-literals in brackets, C-literals in parenthesis, omit the "$\vee$" between literals, and order the clauses from left to right.

$m\, q$	Origin clause.
$l\, [m]\, q$	Resolution with $\neg m \vee l$
$\neg p\, [l]\, [m]\, q$	Resolution with $\neg l \vee \neg p$
$r\, [\neg p]\, [l]\, [m]\, q$	Resolution with $p \vee r$
$\neg l\, [r]\, [\neg p]\, [l]\, [m]\, q$	Resolution with $\neg r \vee \neg l$.
$q\, (\neg l)\, (\neg m)$	A-resolution on $\neg l$ and creation of C-literals.
$l\, [q]\, (\neg l)\, (\neg m)$	Resolution with $\neg q \vee l$.
$\square$	Resolution on l using C-literal.

Now consider the following correspondence between this linear refutation and DP. Resolved-upon literals in the linear refutation are associated with literals chosen to transform nodes in DP. Now, C-literals in the GC procedure play a role analoguous to our literals transmitted to a node higher in the search tree. Both are "lemmas" produced in some part of the search space that are kept for use later. Without the information on the C-literals, the above refutation would examine twice the same search space, in a way analoguous to DP without the added literal at node $C1$.

6.2 Other Pruning Conditions

In [3], a similar pruning condition for the DP procedure is presented. It is shown that if a certain node of the DP search tree C is included in a higher node D, then C is inconsistent if and only if D is. Thus if C fails, the algorithm need not consider D. As an example, consider again Figure 1. If the unary clause l were not present in node $C2$, then $C2$ would be included in $C1$; as $C2$ failed, $C1$ would not need to be considered and thus the algorithm could stop.

The result presented here is a particular case of the result in [3]. If C is included in D, then obviously the set resulting from C by removing clauses containing the chosen literal is also included in D. If inclusion holds, D can be removed. This is stronger than just adding to D a

unary clause, so it should be preferred.

However, our result can be applied in many situations in which the plain inclusion test does not succeed. The example of Figure 1 is such a case. Note that no node in this search tree is included in a higher one, and thus the condition of [3] does not apply.

7 Conclusion

We presented a simple condition that enables the DP procedure to transmit information from one part of the search tree to another part. An example was shown in which this extra information saves unnecessary computation.

We still have no information on how often the condition applies in real problems, that is, how much can be gained from it. This needs to be related to other attempts to improve the efficiency of DP.

Regarding this, it was pointed out in the last section how our results complements that of [3]. Concerning heuristics, our approach suggests the use of one that tries to maximize the chances of applicability of the pruning condition. Further research needs to be done on the interaction of such a heuristic and the one of, for example [7].

References

[1] Cook, S., The Complexity of Theorem Proving Procedures, *3rd. Annual ACM Symposium on Theory of Computation*, (1970) 151-158.

[2] Davis, M. and Putnam, H., A Computing Procedure for Quantification Theory, *Journal of the ACM* 7 (1960) 201-215.

[3] Jeannicot, S., Oxusoff, L. and Rauzy, A., Evaluation Sémantique: Une Propriété Pour Rendre Efficace la Procédure de Davis et Putnam. *Revue d'Intelligence Artificielle* 2 (1988) 41-60.

[4] Loveland, D., *Automated Theorem Proving: A Logical Basis*. North Holland, 1978.

[5] Shensha, M., A Computational Structure for the Propositional Calculus, *Proceedings of IJCAI-89* 384-388.

[6] Shostak, R., Refutation Graphs, *Artificial Intelligence* 7 (1976) 51–64.

[7] Zabih, R. and McAllester, D., A Rearrangement Search Strategy for Determining Propositional Satisfiability. *Proceedings of AAAI-88* 155-160.