SOLVING THE INCREMENTAL SATISFIABILITY PROBLEM

J. N. HOOKER

▷ Given a set of clauses in propositional logic that have been found satisfiable, we wish to check whether satisfiability is preserved when the clause set is incremented with a new clause. We describe an efficient implementation of the Davis–Putnam–Loveland algorithm for checking the satisfiability of the original set. We then show how to modify the algorithm for efficient solution of the incremental problem, which is NP-complete. We also report computational results. ◁

Suppose that a given proposition does not follow from a knowledge base encoded in propositional logic. This can be checked by adding the proposition's denial to the knowledge base and verifying that the resulting set of propositions is satisfiable. We wish to determine whether the proposition follows when the knowledge base is augmented by one or more new propositions. Rather than re-solve the satisfiability problem from scratch, we propose to use information gained while solving the original problem, so as to speed the solution of the new problem.

We therefore address the *incremental satisfiability problem* of propositional logic: given that a set S of propositional clauses is satisfiable, check whether $S \cup \{C\}$ is satisfiable for a given clause C. Our investigation was occasioned by efforts to solve logic circuit verification problems [6], but it has application whenever one wishes to check again for logical inferences after enlarging a propositional knowledge base. It also plays a critical role in the solution of inference problems in first-order predicate logic [4, 7].

Our approach is first to solve the original satisfiability problem with the classical Davis–Putnam–Loveland (DPL) algorithm [1, 3, 9, 11, 12]. We then solve the incremented problem with a modified DPL algorithm that takes advantage of the information in the data structure generated during the solution of the original

Address correspondence to J. N. Hooker, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213.

Received April 1991; accepted June 1992.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$5.00

problem. We choose DPL because, as has been reported elsewhere [2, 5, 10], it is quite competitive with more recent algorithms when it is properly implemented, and it has the advantage of simplicity. But satisfiability algorithms other than DPL can be modified in a similar fashion.

The superior performance of our DPL implementation relies critically on the use of an intelligent branching rule (that of Jeroslow and Wang) and an efficient data structure for reconstructing the problem at a node after backtracking. The data structure is straightforward, but since it has not been described elsewhere, we will provide a detailed statement of the implementation of DPL that we have found to be very efficient. We conclude by reporting computational tests that demonstrate the advantage of our approach.

The incremental satisfiability problem is clearly NP-complete because one can solve a classical satisfiability problem on m clauses by solving at most m incremental problems.

1. THE DPL ALGORITHM

If x_j denotes an atomic proposition, then x_j and its denial $\neg x_j$ are *literals*. A *clause* is a nontautologous disjunction of distinct literals. A set of clauses is *satisfiable* if some assignment of truth values to atomic propositions makes all of the clauses true. An *empty* clause contains no literals and is necessarily false.

The DPL algorithm makes essential use of the *unit resolution* procedure, also known as *forward chaining*. Unit resolution is sound, but unlike DPL, incomplete. The procedure is applied to a given set S of clauses as follows. Look for a *unit clause* (a clause containing just one literal l) in S, and fix the value of l to true. Remove all clauses from S containing l and all occurrences of l's negation from the remaining clauses, and repeat. The procedure terminates when S is empty (indicating that the given clause set is satisfiable) or S contains an empty clause (indicating the the given set is unsatisfiable)—or, if neither of these occurs, when S contains no unit clause (in which case the satisfiability issue is unsettled). An empty clause is obtained when a literal is removed from a unit clause.

The DPL procedure searches a binary tree with the following structure. Every node of the tree is associated with a set of clauses, and the root node is associated with the set S to be checked for satisfiability. A nonleaf node associated with clause set S' has two children associated, respectively, with sets $S'' \cup \{x_j\}$ and $S'' \cup \{\neg x_j\}$, where x_j is any variable that occurs in S'', and S'' is the result of applying the unit resolution procedure to S'. (The unit clauses x_j and $\neg x_j$ that are added to S'' are known as *branch cuts*.) S is unsatisfiable if and only if there exists such a tree in which every leaf can be *fathomed*; that is, every leaf is associated with a clause set for which unit resolution yields the empty clause. (Optionally, the unit resolution procedure can be augmented so as to fix *monotone* variables before performing any resolutions. That is, a variable x_j is set to true if it is posited in every occurrence in S', and is set to false if it is negated in every occurrence.)

DPL normally traverses the tree depth-first, since this requires that fewer nodes be kept in storage at any one time (namely, those along a path from the current node to the root). Traversal begins at the root node. At any node associated with clause set S', there are three cases. 1) The unit resolution procedure applied to S' generates an empty set of clauses. Then the algorithm terminates, because S is

satisfiable. 2) Unit resolution generates the empty clause. The algorithm *backtracks* along the path to the root until it reaches a node with only one child, whereupon it generates the other child and continues at the new node. If it reaches the root without finding such a node, it terminates because S is unsatisfiable. 3) Neither of these occurs, in which case the algorithm *branches*. It generates a child associated with either $S'' \cup \{x_j\}$ or $S'' \cup \{\neg x_j\}$, where x_j occurs in S''. It then continues at the new node.

The Jeroslow–Wang branching rule [8] indicates which child should be generated first when the algorithm branches. It says roughly that the literal added to S'' should occur in a large number of short clauses in S''. Let the length of a clause be the number of literals in it. If v represents a truth value (0 or 1), define the function

$$w(S'', j, v) = \sum_{k=1}^{\infty} N_{jkv} 2^{-k}$$

where N_{jkv} is the number of clauses of length k that contain x_j (if $v = 1$) or $\neg x_j$ (if $v = 0$). If (j^*, v^*) maximizes $w(S'', j, v)$, then we take the $S'' \cup \{x_j *\}$ branch first if $v^* = 1$, and otherwise take the $S'' \cup \{\neg x_j *\}$ branch first. The rationale is that $w(S'', j, v)$ estimates the probability that a random truth assignment will falsify one of the clauses eliminated when one branches on x_j. Thus, by maximizing $w(S'', j, v)$, we also maximize the probability that a random truth assignment will *satisfy* all of the remaining clauses. This can lead to a satisfying solution earlier in the search process, if S is satisfiable. Conversely, if S is unsatisfiable, then fixing the value of literals in short clauses tends to generate more unit clauses, and therefore to allow unit resolution to detect inconsistency more quickly.

When backtracking to a node, we must restore the clause set S' corresponding to that node. One option is to store the problem associated with every unfathomed node, but this may require excessive storage. Another option is to store only the original problem set S, add to S all the branch cuts between the current node and the root, and perform unit resolution on the resulting set to obtain S'. But this is quite time consuming. A better alternative is to maintain for each clause i the highest level a_i, in the path from the root to the current node, at which the clause is still in S' (the root is level 1). We also maintain for each variable x_j the highest level b_j at which the x_j still occurs in S'. A variable that is no longer in S' at level k may have been removed for either of two reasons: its value was fixed by unit resolution, or all of the clauses containing it were removed by unit resolution (or both).

In the accompanying statement of the DPL procedure (Algorithm 1), we let L_k be the branch cut added to obtain the left child of the node last visited at level k, and R_k the cut added to obtain the right child, with $R_k = 0$ if no right child has been generated. T contains the variables fixed to true at the current node, and F those fixed to false. If a solution is found, it is recovered by setting the variables currently in T to true and those in F to false; each of the remaining variables may be set to either true or false. We use a procedure UPDATE to keep track of the clause set associated with the current node, and a procedure RESTORE to restore the problem associated with the current node.

The algorithm is written in pseudo-Pascal, so that no procedure (except DPL) is executed unless called by a subsequent procedure. Execution therefore starts at the Begin statement following the end of the BUILD TREE procedure.

Our Fortran implementation of the algorithm stores each clause in the form of two linked lists. One contains the indices of positive literals in ascending order, and the other similarly lists the negative literals. Pointers to these lists are stored in a linked list, in ascending order of clause indices. Variables and clauses can therefore be quickly added or deleted, and the running time for RESTORE and UPDATE is linear in the number of literals. The values of the Jeroslow–Wang function $w(S'', j, v)$ are computed simultaneously for all (j, v) on a single pass through the data structure. The sets T and F are maintained simply as an array that records *true*, *false*, or *undetermined* for each variable.

2. AN INCREMENTAL DPL ALGORITHM

Consider the search tree that the above DPL algorithm generates when testing S for satisfiability. When we add a new clause C to S, the clause set S' associated with any fathomed node of this tree remains unsatisfiable. So, there is no point in looking again at any part of the tree already generated, except the path from the root to the last node examined. We simply continue to build the tree, beginning at an appropriate point along this path, and keeping in mind that C must also be satisfied. We must also update the data structure to show at what unfathomed nodes C remains in the problem.

More precisely, when we initially determine that S is satisfiable, we save the data structure by saving the numbers a_i, b_j, the sets T, F, the cuts L_k, R_k, and the level k in the search tree at which the search terminates. Then to test $S \cup \{C\}$ for satisfiability, we first check whether the variables fixed at level k satisfy or falsify C. If they satisfy C, we note that $S \cup \{C\}$ is satisfiable and we quit. If they falsify C, we backtrack, along the path from the current node to the root, to the node at which C is first falsified, and resume the DPL algorithm at that point. If the fixed variables leave the truth value of C undetermined, we branch on one of the variables in C (unless only one variable in C is unfixed, in which case we simply fix its value). We then note that $S \cup \{C\}$ is satisfiable and we quit. In all of these cases, we update the data structure to indicate at which node C is eliminated from the problem.

In Algorithm 2, which is a statement of incremental DPL, k_0 is the level, along the path from the root to the current node, at which C is first falsified (if at all), and k_1 is the level at which it is first satisfied (if at all). The algorithm can be applied repeatedly as new clauses are added.

3. COMPUTATIONAL RESULTS

Algorithms 1 and 2 were tested on part of a collection of satisfiability problems assembled by Radermacher [10, 13] for establishing benchmarks. We solved the first two problems from each of 14 groups of problems, and the results appear in Table 1. Problem groups ulmxxxr0, ulmxxxr1, and ulmxxxr2 encode systems of linear equations modulo 2. Groups real1x12 and real2x12 are stuck-at-zero problems in VLSI design. Groups jnhx, jnh2xx, and jnh3xx are random problems intentionally generated with parameters set to yield hard problems. Groups ulmbcxxx, ulmbpxxx, and ulmbsxxx are chessboard problems. Group twoinvrx consists of logic circuit construction problems, group holex of pigeonhole problems,

TABLE 1. Computational results

			All Increments				Last Increment Only				
	Size		No. Nodes		CPU Sec.			No. Nodes		CPU Sec.*	
Problem	n	m	DPL	IDPL	DPL	IDPL	Sat?	DPL	IDPL	DPL	IDPL
ulm027r0	25	64	638	83	1.16	0.06	Y	7	1	0.04	0.00
ulm027r1	23	64	671	78	1.31	0.07	Y	21	1	0.08	0.00
ulm027r2	25	64	669	86	1.16	0.08	Y	7	1	0.04	0.00
ulm054r0	50	128	2502	299	10.9	0.34	Y	13	1	0.16	0.00
ulm054r1	46	128	3143	291	13.4	0.42	Y	265	1	1.33	0.00
ulm054r2	50	128	2492	336	10.7	0.57	Y	13	1	0.17	0.00
real1a12	18	273	1221	273	7.40	0.05	Y	3	1	0.05	0.00
real1b12	15	110	471	124	1.03	0.05	Y	3	1	0.02	0.00
real2a12	18	275	1226	282	7.32	0.07	Y	3	1	0.06	0.00
real2b12	15	110	471	124	1.01	0.08	Y	3	1	0.02	0.00
jnh201	100	800	35087	6989	2419	266	Y	32	1	8.06	0.00
jnh202	100	778	38623	101897	3095	12036	N	93	8325	50.1	1178
jnh1	100	850	41083	16347	2684	998	Y	126	1	18.7	0.00
jnh2	100	838	35099	9929	2971	1121	N	221	411	81.4	85.0
jnh301	100	900	44619	198730	3248	27770	Y	360	1	72.4	0.00
jnh302	100	756	35621	3920	2286	683	N	109	1665	65.4	634
ulmbc024	24	164	2122	185	5.69	0.09	Y	10	1	0.09	0.00
ulmbc040	40	340	6713	364	47.9	0.31	Y	24	1	0.42	0.00
ulmbp048	48	99	2455	2820	9.33	16.1	N	505	406	3.50	2.79
ulmbp070	70	161	9437	29212	69.7	252	N	4671	28933	42.2	251
ulmbs040	40	70	1197	139	2.39	0.11	Y	11	1	0.06	0.00
ulmbs060	60	120	2997	189	12.2	0.22	Y	16	1	0.18	0.00
twoinvrt	30	79	921	141	2.32	0.14	Y	5	1	0.06	0.00
twoinvr1	30	81	975	116	2.58	0.11	Y	15	1	0.21	0.00
hole6	42	133	9665	14441	32.8	82.1	N	6491	14278	22.4	82.5
hole7	56	204	71826	184714	370	1550	N	65561	184482	326	1550
nod5col4	40	128	2956	394	12.4	2.21	N	205	169	2.09	1.85
nod6col4	60	226	7554	709	73.2	9.60	N	263	99	7.85	2.34

*The computer does not measure CPU times less than 0.01 second.

and group nodxcolx of graph coloring problems. Table 1 shows the number n of atomic propositions in each problem. A more detailed description of the problems can be found in [2, 10, 13].

For each problem, the order of clauses was randomized, and a subproblem consisting of clauses $1,\ldots,k$ of the original problem was solved for each $k = 1,\ldots,m$. When the original problem is satisfiable, m is the number of clauses in the problem, and otherwise m is the number of clauses that had been added when the subproblem first became unsatisfiable. The series of subproblems was solved once by solving each subproblem from scratch using the DPL algorithm stated above (the results are marked DPL in Table 1). The subproblems were then re-solved by repeated application of the incremental DPL algorithm (results marked IDPL). Table 1 shows results not only for the entire series of subproblems $1,\ldots,m$ (left side of the table), but for the last problem alone as well (right side). It also indicates by "Y" or "N" whether the last subproblem is satisfiable.

The DPL algorithm is the same as tested in [2,10]. It and the incremental algorithm were coded in Fortran and run on a Sun Sparc Station 330 operating under SunOS 4.1.

The incremental algorithm is substantially faster for 22 out of 28 problems. In at least half the problems, incremental DPL solved all of the subproblems in about the same time it took DPL to solve only the last subproblem.

When choosing a variable on which to branch, DPL can apply the Jeroslow-Branching rule to the entire problem, whereas incremental DPL can look only at the clauses that have been added so far. One might therefore expect incremental DPL to use a less efficient branching order than DPL, and this could offset its advantage of using the search tree accumulated in prior solutions. But as we noted, this happened for only 6 of the 28 problems. In fact, one would not expect incremental DPL to perform well on at least two of the six, namely, the pigeonhole problems, since for these the subproblems are quite easy until the very last clause is added. Incremental DPL can also result in *more* efficient branching order than DPL, since for three of the test problems, incremental DPL takes less time to solve all of the incremental problems than DPL takes to solve the last one only.

This work was supported in part by the U.S. Air Force Office of Scientific Research Grant AFOSR-91-0287. GSIA Working Paper 1991-9.

REFERENCES

1. Davis, M. and Putnam, H., A Computing Procedure for Quantification Theory, *Journal of the ACM* 7:201–215 (1960).
2. Harche, F., Hooker, J. N., and Thompson, G. L., Computational Comparison of Satisfiability Algorithms, Working Paper, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1991.
3. Hooker, J. N., A Quantitative Approach to Logical Inference, *Decision Support Systems* 4:45–69 (1988).
4. Hooker, J. N., New Methods for Computing Inferences in First-Order Logic, Working Paper, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1991.
5. Hooker, J. N. and Fedjki, C., Branch-and-Cut Solution of Inference Problems in Propositional Logic, *Annals of Mathematics and AI* 1:123–139 (1990).
6. Hooker, J. N. and Yan, H., Logic Circuit Verification by Benders Decomposition, Working paper, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1991.
7. Jersolow, R. E., Computation-Oriented Reductions of Predicate to Propositional Logic, *Decision Support Systems* 4:183–197 (1988).
8. Jersolow, R. E. and Wang, J., Solving Propositional Satisfiability Problems, *Annals of Mathematics and AI* 1:167–187 (1990).
9. Loveland, D. W., *Automated Theorem Proving: A Logical Basis*, North-Holland, 1978.
10. Mitterreiter, I. and Radermacher, F. J., Experiments on the Running Time Behavior of Some Algorithms Solving Propositional Logic Problems, Working Paper, Forschungsinstitut für anwendungsorientierte Wissensverarbeitung, Ulm, Germany, 1991.
11. Quine, W. V., The Problem of Simplifying Truth Functions, *American Mathematical Monthly* 59:521–531 (1952).
12. Quine, W. V., A Way to Simplify Truth Functions, *American Mathematical Monthly* 62:627–631 (1955).

13. Truemper, K. and Radermacher, F. J., Analyse der Leistungsfähigkeit eines neuen Systems zur Auswertung aussagenlogischer Probleme, Report FAW-TR-90003, Forschungsinstitut für anwendungsorientierte Wissensverarbeitung, Ulm, Germany, 1991.

ALGORITHM 1 (DPL)

```
Procedure DPL; {Check clause set S for satisfiability.}
  Procedure UNIT RESOLUTION; {S' will contain results of unit resolution.}
  Begin
    If S' = ∅ or S' contains the empty clause, stop;
    While S' contains a unit clause C do
      Begin
        If S' = ∅ or S' contains the empty clause, stop;
        Else
          If the variable x_j in C is positive, then
            Begin
              Add x_j to T;
              Remove from S' all clauses containing literal x_j and all occur-
              rences of literal ¬ x_j;
            End
          Else
            Begin
              Add x_j to F;
              Remove from S' all clauses containing literal ¬ x_j and all occur-
              rences of literal x_j;
            End;
      End;
  End UNIT RESOLUTION;
  Procedure UPDATE; {Record variables and clauses that are in S'.}
  Begin
    For each clause i in S' do
      Begin
        Let a_i := k;
        For each variable x_j in clause i do
          Let b_j := k;
      End;
  End UPDATE;
Procedure RESTORE; {Reconstruct problem at current level k node.}
Begin
  For all j such that b_j ≥ k do
    Begin
      Remove x_j from T or F (whichever contains it);
      Let b_j := k;
    End;
  Let S' := 0;
  For all clauses i in S do
    If a_i ≥ k then
      Begin
```

```
            Let a_i := k;
            Delete from clause i all literals involving a variable x_j for which b_j < k,
            and add the resulting clause to S';
          End;
End RESTORE;
Procedure BUILD TREE; {Conduct tree search beginning at level k.}
Begin
  While k > 0 do
    Begin
      Perform UNIT RESOLUTION;
      Perform UPDATE; {Record variables and clauses left in the problem.}
      If S' is empty, then
        Stop, because S is satisfiable;
      Else
        If S' contains the empty clause, then
          {Backtrack to a node with no right child.}
          Begin
            Let R_k := 1;
            While k > 0 and R_k ≠ 0 do
              k := k - 1;
            If k > 0 then
              Begin
                {Reconstruct the problem at this node and then generate right child.}
                Perform RESTORE;
                Let R_k := ¬ L_k and S' := S' ∪ {R_k};
                Let k := k + 1;
              End;
          End
        Else {Branch as recommended by Jeroslow–Wang heuristic.}
          Begin
            Pick j*, v* so that
              w(S', j*, v*) = max_{j,v}{w(S', j, v)};
            If v* = 1 then let L_k := x_j *
            Else let L_k := ¬ x_j *;
            Let R_k := 0, S' := S' ∪ {L_k} and k := k + 1;
          End;
    End;
  Stop, because S is unsatisfiable;
End BUILD TREE;
Begin {Execution starts here.}
  Begin with a set S of clauses containing variables x_1, ..., x_n;
  Let S' := S, k := 0, T := 0, F := 0, and perform UPDATE;
  Set the level k := 1;
  Perform BUILD TREE;
End DPL.
```

ALGORITHM 2 (INCREMENTAL DPL)

```
Procedure INCREMENTAL DPL; {Check S ∪ {C} for satisfiability.}
```

```
Procedure SCAN NEW CLAUSE;
{Check which literals in C are fixed, and let C' contain those remaining. Determine
at what level in the tree C is first satisfied or falsified (if either) by fixed literals.}
Begin
  For each literal l in C'' do
    Begin
      Let x_j be the variable in l;
      {If x_j is true and occurs posited in C, then C is already satisfied at a level no
      higher than where x_j is fixed.
      But if x_j is negated in C, then it is removed from C, and C is already falsified
      at a level no lower than where x_j is fixed.}
      If x_j ∈ T then
        Begin
          If l = x_j then
            Let k_1 := min{k_1, b_j}
          Else
            Remove x_j from C' and let k_0 := max{k_0, b_j};
        End
      {Analogously if x_j is false.}
      If x_j ∈ F then
        Begin
          If l = ¬ x_j then
            Let k_1 := min{k_1, b_j}
          Else
            Remove ¬ x_j from C' and let k_0 := max{k_0, b_j};
        End;
    End;
End SCAN NEW CLAUSE;
Procedure BRANCH; {Branch on an undetermined variable in C.}
Begin
  Let a_i := k;
  If C' is a unit clause, then
    {In this case, there is no need to branch, because the remaining literal must be
    true.}
    Begin
      Let x_j be the variable in C';
      If C' = x_j then add x_j to T
      Else add x_j to F;
    End
  Else
    {Otherwise create a new child node, in preparation for the next time INCRE-
    MENTAL DPL is executed.}
    Begin
      For each variable x_j in C' do
        If x_j ∉ T ∪ F then b_j := k;
      Pick a literal l in C' and let x_j be the variable in l;
      If l = x_j then
        Add x_j to T and set L_k := x_j, R_k := 0;
      Else
```

```
            Add x_j to F and set L_k := ¬ x_j, R_k := 0;
          Let a_i := k, k := k + 1;
        End;
      Stop, because S is satisfiable;
    End BRANCH;
  Begin {Execution begins here.}
    Let k_0 := 0, k_1 := ∞, C' := C; Let C be numbered clause i;
    Let S := S ∪ {C};
    {Check which literals of C have been fixed, let C' contain those remaining, and
    determine at what level C is first satisfied or falsified (if either).}
    Perform SCAN NEW CLAUSE;
    {If C is already satisfied, record removal of C at the level at which it is first satisfied,
    and stop.}
    If k_1 < ∞ then
      Begin
        For each variable x_j in C do
          If x_j ∉ T ∪ F then let b_j := max{b_j, k_1};
        Set a_k := k_1;
        Stop, because S ∪ {C} is satisfiable;
      End
    {If C is already falsified, backtrack to the level at which it is first falsified.}
    Else
      If C' is empty then
        Begin
          For each variable x_j in C do
            If x_j ∉ T ∪ F then let b_j := max{b_j, k_0};
          Let k := k_0 + 1, R_k := 1, a_i := k_0;
          While k > 0 and R_k ≠ 0 do k := k − 1;
          {Restore problem at node backtracked to, generate the right child, and
          continue the tree search in DPL.}
          If k > 0 then
            Begin
              Perform RESTORE;
              Let R_k := ¬ L_k and S' := S' ∪ {R_k};
              Let k := k + 1;
              Perform BUILD TREE;
            End
          {S is unsatisfiable if backtracking is all the way to root.}
          Else
            Stop, because S is unsatisfiable;
        End
      Else
        {Branch if C is neither satisfied or falsified by fixed literals.}
        Perform BRANCH;
  End INCREMENTAL DPL.
```