Annals of Mathematics and Artificial Intelligence, 1 (1990) 167–187

SOLVING PROPOSITIONAL SATISFIABILITY PROBLEMS

Robert G. JEROSLOW * and Jinchang WANG

College of Management, Georgia Institute of Technology, Atlanta, GA 30332, USA

Abstract

We describe an algorithm for the satisfiability problem of propositional logic, which is significantly more efficient for this problem than is a general mixed-integer programming code. Our algorithm is a list processor using a tree-search method, and is based on Loveland's form of the algorithm of Davis and Putnam.

Keywords: Satisfiability problem, branch-and-bound, propositional logic.

1. Introduction

We shall discuss an algorithm for the satisfiability problem of propositional logic. Our experiments reported below, on the CDC 180/855 CYBER, will show this algorithm to be notably more efficient than the APEX IV code, an excellent general mixed-integer programming algorithm. In fact, on the more difficult problems, our algorithm is more than an order of magnitude faster than APEX IV. Moreover, we shall see that propositional satisfiability problems in a hundred to a few hundred letters and constraints generally are rapidly resolved, confirming the earlier experiments of [2], but with typical reductions from several seconds, with APEX IV, to under a second with the algorithm reported here.

Our algorithm is an all-in-core list processing method based on the propositional logic version of the algorithm DP of Davis and Putnam [4], utilizing Loveland's problem subdivision in place of resolution, and with heuristics we have developed. As shown in [2], the Davis–Putnam algorithm in Loveland's form (DPL) is very closely related to the branch-and-bound (BB) method for mixed-integer programming (MIP) when applied to the "standard representation" of disjunctive clauses as linear inequalities in binary variables. The primary difference is that BB may establish satisfiability at a node of the search tree, via an incumbent which solves the linear relaxation, even though DPL at the corresponding node does not produce an incumbent and proceeds to branch (subdivide the problem at the node).

* Deceased. Research partially supported by a research grant from the National Science Foundation (DMS-8513970).

One of the main roles of the heuristics of our method DPLH is to replace this incumbent-finding feature of linear programming by a list processing procedure which, while not the same as the linear relaxation, is intended to function in the same way within the overall algorithm. In addition, DPLH contains heuristics for what would be called, in MIP terminology, the branching variable choice rule and subproblem selection rule.

The problems on which we tested DPLH are randomly generated according to a probability distribution which does not fit those for which analytical results on satisfiability (as opposed to finding *all* satisfying truth valuations) have thus far appeared. For such results and related work, see e.g. [6,13]. One of these problems could not be solved by APEX IV in over ninety seconds, while it was solved in a third of a second by DPLH; a second problem, which was solved in just over two seconds by DPLH, was not solved in over seven minutes by APEX IV; both were consistent. Thus, these problems are nontrivial.

In terms of the difference between the original algorithm DP of Davis and Putnam and DPL, we recall the following from [8]. Loveland's splitting [10] always improves the remainder of the algorithm as compared to the resolution used in DP (via a smaller linear relaxation in both subproblems), and in many instances the total size of the two subproblems of DPL is less than that of the single problem retained by DP. Thus DPL is the generally preferred method of the two.

Clearly, much of the time saving of our algorithm DPLH, as compared to APEX IV, derives from the fact that DPLH is a list processor which acts directly on relevant data, while APEX IV carries around data structures such as bases and vectors, much of which turn out to be unnecessary to do just propositional logic. However, when logic is compounded by additional constraints, such as capacity restrictions, it seems very likely that relatively more of a linear programming algorithm will be needed for maximum efficiency. Further investigations are needed to address such issues.

The algorithms DPL, DPLH and the branch-and-bound algorithm of APEX IV all contain subroutines which, while limited in their ability to diagnose satisfiability, are rather fast and frequently helpful. They serve to illustrate Nevin's view [12] on the value of using such subroutines, *prior to* initiating steps (such as branching or reasoning by cases) which greatly increase the size of the problem being processed. While the latter kinds of steps are, in principle, adequate to diagnose satisfiability by themselves (without the aid of other subroutines), one uses the size-increasing steps only as a "last resort" when more limited subroutines fail to determine satisfiability.

We now outline the remainder of the paper. Section 2 reviews the Davis–Putnam algorithm in Loveland form (DPL) and standard representation *via integer programming* of a proposition. In section 3, we describe and explain our algorithm. Experimental results are given in section 4.

Appendix A gives details of the DPLH procedure. In appendix B, four more

experiments we did are briefly discussed. These include a "trivial" problem which our algorithm did not solve.

2. The Davis–Putnam algorithm

In this section, Loveland's form [10] of the Davis–Putnam procedure (DPL) and the standard representation (SR) of a proposition via integer programming are reviewed.

2.1. TERMINOLOGY AND ABBREVIATIONS

A propositional *letter* such as P_1, P_2, $P_3, \ldots$ stands for an *atomic proposition*. Other (complex) *propositions* are built up from the letters using the symbols: disjunction, denoted $\vee$ (meaning: "or", in a nonexclusive sense); conjunction, denoted $\wedge$ (meaning: "and"); and negation, denoted $\neg$ (meaning: "not"). Thus, $(P_1 \vee \neg P_2) \wedge (\neg P_1 \vee (P_3 \wedge P_5))$ is a proposition.

A *literal* is an atomic letter or its negation. Thus, P_5 and $\neg P_7$ are literals.

A proposition B is a *conjunctive normal form* (*c.n.f.*) if it is a conjunction $B = B_1 \wedge B_2 \wedge \ldots \wedge B_t$ of propositions B_i $(1 \leqslant i \leqslant t)$, each of which is a disjunction $B_i = \bigcup_{j \in I_i} \pm P_j$ of literals. (Here, $+P_j$ abbreviates P_j while $-P_j$ abbreviates $\neg P_j$). The individual B_i are called (*disjunctive*) *clauses*. A *unit clause* is a clause consisting of one element, i.e. consisting of one literal. (We conceptually differentiate a unit clause from a literal.)

A letter P_j occurring in a c.n.f. B is called *monotone* (in B) if either P_j does not occur as a literal of B, or $\neg P_j$ does not occur as a literal of B.

DPL will abbreviate the Davis–Putnam procedure in Loveland's form [10].

SR will abbreviate the standard representation of a proposition via integer programming.

DPLH will denote our algorithm to test the satisfiability of a proposition. The abbreviation DPLH derives from "DPL plus heuristic".

2.2. REVIEW OF DPL

DPL is applied to a proposition in c.n.f. It consists of *three subroutines*, for our purposes: *clausal chaining*, denoted CC (this is unit resolution in propositional logic); *monotone variable fixing*, denoted Mon; and *splitting*. We next review these subroutines.

2.2.1. Subroutine CC

Remove all clauses containing both some letter P_j and its negation.

For as long as there is a unit clause, assign the corresponding atomic letter the truth value needed to make this clause true. If two unit clauses are opposite, the

list of clauses is inconsistent. Otherwise, remove all clauses made true by this truth value and remove the opposite of this literal from all clauses then remaining.

If no clauses remain, a satisfying truth valuation has been found. Otherwise, if clauses remain, go to subroutine Monotone.

2.2.2. Subroutine Monotone

As long as there are monotone letters, set these letters to truth valuations, which makes all clauses true that contain them, and remove these clauses from the list.

If no clauses remain, a satisfying truth valuation has been found.

Otherwise, go to subroutine Split.

2.2.3. Subroutine Split

Choose a letter P_j from the list.

The clauses can be divided into three groups:

Group A: $P_j \vee R_1, \ldots, P_j \vee R_a$ – the clauses containing P_j positively.

Group B: $\neg P_j \vee S_1, \ldots, \neg P_j \vee S_b$ – the clauses containing P_j negatively.

Group C: $T_1, \ldots, T_c$ – clauses not containing P_j.

The list is split into two lists of clauses:

$R_1, \ldots, R_a, T_1, \ldots, T_c$ and P_j is set to F

and

$S_1, \ldots, S_b, T_1, \ldots, T_c$ and P_j is set to T.

These lists are added to the set of lists. A list is then withdrawn from this set and one returns to subroutine CC.

If all the lists currently in the set of lists are inconsistent, the proposition we are testing is not satisfiable.

DPL is not so much an algorithm as it is an *algorithm framework*. Its precise implementation depends on the strategy for choosing the letter P_j in subroutine Split (analogous to the branching variable of branch-and-bound), and the strategy for selecting which list is processed next (analogous to the node choice rule).

2.3. THE STANDARD REPRESENTATION OF A DISJUNCTIVE CLAUSE VIA INTEGER PROGRAMMING

Every proposition can be transferred to its equivalent c.n.f. (by the distribution law or linear time procedure of Tseitin which is cited in [2]). Furthermore, every proposition in c.n.f. can be represented via integer programming. The standard representation (SR) is one of such representations.

In the SR, a clause $\bigcup_{j \in J} \pm P_j$ for $J \neq \phi$ is represented by a single linear constraint:

$$\sum_{j \in J} z(\pm P_j) \geqslant 1, \quad z(P_j) \in \{0, 1\},$$

where $z(+P_j)$ is $z(P_j)$ and $z(-P_j)$ abbreviates $1 - z(P_j)$.

For example, the clause $P_1 \vee \neg P_2 \vee \neg P_5 \vee P_7$ is represented by:

$$z(P_1) + (1 - z(P_2)) + (1 - z(P_5)) + z(P_7) \geqslant 1,$$

$$z(P_1),\ z(P_2),\ z(P_5),\ z(P_7) \in \{0, 1\}$$

Clearly, if "1" represents "true" and "0" represents "false", the binary solutions of the SR are exactly the satisfying truth valuations of the clause.

3. The algorithm DPLH

In [2] experiments were described on testing propositional satisfiability, by testing the consistency of the associated system of clauses in standard representation. A quite powerful commercial computer code APEX IV was used to solve several of the integer programs.

As shown in [2], the branch-and-bound (BB) method applied to the SR is quite similar to DPL, once both are equipped with the same variable choice rules and subproblem selection rules. One difference occurs, in that DPL has monotone variable fixing, while BB does not. This fact gives some advantage to DPL. The second difference favors BB; we call it "incumbent finding" and describe it as follows.

Incumbent finding consists of the fact that the linear relaxation (LR) at a node of the BB search tree may yield an integer solution (an "incumbent"). Sometimes when this happens, clausal chaining (CC) of DPL also finds an incumbent, but not always. As shown in [2], the difference between BB and DPL at a node of the search tree is this: any incumbent found by DPL is one where the truth values are *forced* to take on a "true" or a "false" value; while an incumbent can occur in the LR of BB even if no values are forced, provided only that the values are *possible*.

For example, given the c.n.f. $(\neg P_1 \vee P_2) \wedge (\neg P_1 \vee \neg P_2)$ clausal chaining takes no action, since there are no unit clauses. However, in the LR of the standard representation, $z(P_1) = z(P_2) = 0$ is a solution which derives from a basic feasible solution to the LR.

The example just given also illustrates the fact that, while all settings of clausal chaining derive from letters which are forced to values, not all forced values are detected by CC. In our example, P_1 is forced to be false.

Clearly, incumbent finding is an advantage of BB, but it can contribute only to consistent problems; in inconsistent problems, it is not useful.

In an imprecise sense, we may put the relations between DPL and BB into the following formula.

DPL = BB + monotone fixing − incumbent finding.

Of course, monotone variable fixing can be added to BB, as it is valid for (satisfiability in) propositional logic. However, a disadvantage of BB is its need to

carry and manipulate the large data structures such as vectors and bases, when evidence from logic is that these algebraic data structures are irrelevant in this setting.

We sought to develop a method with the incumbent-finding advantages of BB but which would avoid the algebraic data structures of BB.

The algorithm DPLH is based on DPL. A heuristic is added, which plays two roles in the algorithm. The first role is as a splitting rule and a subproblem selection rule. The second role is as an incumbent finder. We shall describe the heuristic from the view that it is an efficient splitting rule.

3.1. THE HEURISTIC FOR SPLITTING

Let S denote a set of disjunctive clauses.

Define the weight of S as

$$W(S) = \sum_p n_p/2^p,$$

where N_p = number of clauses of length p in S, and "a clause has length p" means that there are p literals in the clause.

We give an example. If S consists of the four clauses listed below, its weight is computed as follows:

Clause:		*Length*:
	$\neg P_1 \vee P_2$	2
	$P_1 \vee \neg P_2 \vee P_3$	3
	$P_2 \vee P_5$	2
	$P_1 \vee \neg P_3$	2

$W(S) = 3 \times 2^{-2} + 2^{-3} = 7/8.$

For a propositional letter P_j, and a truth valuation $i = T$(rue) or F(alse), let S_j^i be that subset of S in which P_j occurs positively, respectively negatively, if $i = T$, resp. $i = F$.

Our heuristic for splitting is to pick i^* and j^* so as to maximize $w(S_j^i)$, i.e.:

$$w\left(S_{j^*}^{i^*}\right) = \max_{ij} w\left(S_j^i\right) = m.$$

The branching variable (letter) will then be P_{j^*}. (Ties at the maximum are broken by least index.)

3.2. MOTIVATION OF THIS HEURISTIC

In a list of clauses, there are 2^n truth valuations. A clause of length p rules out exactly 2^{n-p} of these. Thus, S rules out at most $\Sigma_p n_p 2^{n-p} = 2^n w(S)$ of these. Our heuristic chooses that letter P_{j^*} which can yield a system of minimum weight (by setting to i^*), hence, in an intuitive sense, a system *most likely to be satisfiable*. Note that when $w(S) < 1$, S must be satisfiable.

We illustrate these ideas by an example. Suppose that S has the two clauses given below:

$$P_1 \vee P_2 \vee \neg P_3, \quad \neg P_2 \vee P_3.$$

Here we have three letters in S, so $n = 3$, and there are $2^3 = 8$ possible truth valuations. The first clause has length $p = 3$ and therefore rules out $2^{3-3} = 2^0 = 1$ truth valuation, which is ($P_1 =$ false, $P_2 =$ false, $P_3 =$ true). The second clause has $p = 2$ and rules out $2^{3-2} = 2^1 = 2$ truth valuations. They are ($P_1 =$ true, $P_2 =$ true, $P_3 =$ false), ($P_1 =$ false, $P_2 =$ true, $P_3 =$ false).

Here $w(S) = 2^{-3} + 2^{-2} = 3/8$. Since $w(S) = 3/8 < 1$, S must be satisfiable.

For each letter P_j and its truth valuation $i = T$ or F, $w(S_j^i)$ may be viewed as an indicator of the extent to which the remaining set of S is relaxed after fixing P_j to i. A larger $w(S_j^i)$ implies that there are more truth valuation choices in the remaining set of S after fixing P_j to i. In this sense, the likelihood of a satisfying truth valuation is increased by setting P_{j*} to i^*.

In the example above,

$$w(S_1^T) = w(S_2^T) = w(S_3^F) = 1/8, \quad w(S_2^F) = w(S_3^T) = 1/4.$$

With our heuristic, we may choose to fix P_2 to F or to fix P_3 to T. If we set P_2 to F, we have three choices of truth valuations:

$$(P_1 = T,\ P_3 = T), \quad (P_1 = F,\ P_3 = F), \quad (P_1 = T,\ P_3 = F).$$

But if we chose to fix $P_2 = T$, the remaining clause would be $\neg P_2 \vee P_3$, which could be simplified to P_3. The choices of the truth valuation are only two:

$$(P_1 = T,\ P_3 = T), \quad (P_1 = F,\ P_3 = T).$$

The concept of the "weight" $w(S)$ of a set S of clauses, and the sets S_j^i, occur in [1] and our heuristic of this subsection is motivated by results in [1].

3.3. LINEAR TIME ALGORITHM OF CLAUSAL CHAINING (CC)

The following is a linear time algorithm for a clausal chaining subroutine, which adapts the data structures for Horn clauses as given in [5]. We used it in our algorithm DPLH to implement CC.

For each clause, one maintains a list clauselist(i) which lists all the letters occurring in clause i with their signs. For example, if the fifth clause is $\neg P_1 \vee P_4 \vee \neg P_2$, then clauselist(5) = $\{-1, 4, -2\}$.

For each letter P_j, one also maintains a list letterlist(j) which lists all the clauses in which P_j occurs with signs. In the example we cited in section 3.1, letterlist(1) = $\{-1, 2, 4\}$, letterlist(2) = $\{1, -2, 3\}$ letterlist(3) = $\{2, -4\}$, letterlist(5) = $\{3\}$.

For each clause i, we use $L(i)$ to denote its length. If $L(i) = 0$, this clause is dropped from the list of clauses.

For each letter P_j, we use setting(j) to indicate whether P_j has been fixed, and if so, what truth valuation it is fixed to. Setting(j) = 0 means P_j has not been fixed. Setting(j) = -1, respectively setting (j) = 1 indicates that P_j has been fixed to "false", resp. "true".

To implement clausal chaining with these data structures, we use the following procedure, which is based on [5].

Algorithm DG (for CC)

Initially, put all unit clauses in a "stack". When the clause $\pm P_j$ reaches the top of the stack, "pop" it and give it value T if it is $+P_j$ and F if it is $\neg P_j$. Check that this setting does not contradict an earlier truth assignment; if it does, inconsistency is proved. Otherwise examine letterlist(j), where there are two cases:

Case 1: P_j is fixed to T.

Drop all the clauses which have (+) sign from the list of clauses. For each clause(i) which is in the remaining list of clauses and has a -1 sign in letterlist(j), erase $-j$ from clauselist(i) and decrement $L(i)$ by 1. If $L(i) = 1$, add the clauselist to the bottom of the stack.

Case 2: P_j is fixed to F (done similarly to Case 1).

Linear time is established for DG, by observing that every cycle of the algorithm shortens the list of clauses by at least a number of occurrences of some letter which is proportional to the number of computation steps in the cycle. This is true, since only those clauses in letterlist(j) are examined during the cycle.

We remark that subroutine Monotone (i.e. monotone variable fixing) can also be done in linear time, using the same data structures as for DG. We used such a linear time algorithm to implement that subroutine. We leave details to the reader.

3.4. DPLH – OUR ALGORITHM

The algorithm DPLH consists of DPL modified in this manner: after subroutine Monotone and prior to subroutine Split, the heuristic of section 3.1 is used *iteratively* to attempt to find an incumbent solution. Only if it fails, is subroutine Split entered.

In this_manner, use of the heuristic is a substitute for the role of the linear relaxation of branch-and-bound. However, its iterative (or repeated) use leads to some subtleties in this simple description. The heuristic sets only a single letter (i.e. P_{j*} to the value i^*). After this setting, DPLH enters subroutine CC and then subroutine Mon *prior* to setting the next letter (if needed) via the heuristic. This process is repeated until either an incumbent is found (and the problem is thus proven satisfiable) or until an inconsistency is encountered. During the iterative use of the heuristic, there is *no backtracking* until it is finished.

The fact that the iterative use of the heuristic also serves to develop part of the search tree complicates our discussion of DPLH, although conceptually it is as described above. While in the iterative phase of the heuristic, one must keep track of the subproblem node which led to this phase, i.e. from which the heuristic tracing decends. It is called the "active problem", and if the iterative phase ends with inconsistency in its attempt to find an incumbent, backtracking occurs to the active node.

The node currently being processed (for CC, or fixing, or computations for the heuristic) is called the "current problem". Thus, when the "current problem" is found inconsistent, backtracking jumps control to the "active problem", which may be a considerable distance up the search tree from the "current problem".

When backtracking to the active problem occurs, the branching (splitting) letter is still P_{J*}, however, one proceeds in the direction *opposite* to the heuristic to develop a new active subproblem.

Precise details of the control structure of DPLH, and its elaborate backtracking procedures, are provided in appendix A.

Since the core memory is not large enough to store every whole problem corresponding to each node, we store only the original problem, and the setting list for each node. Thus, each node can be recreated by using its setting list on the original problem and then simplifying it. This "all-in-core" procedure is also faster than retrieval from disk files.

For each node, we maintain the following information:

- whether it is fathomed,
- whether it has child nodes,

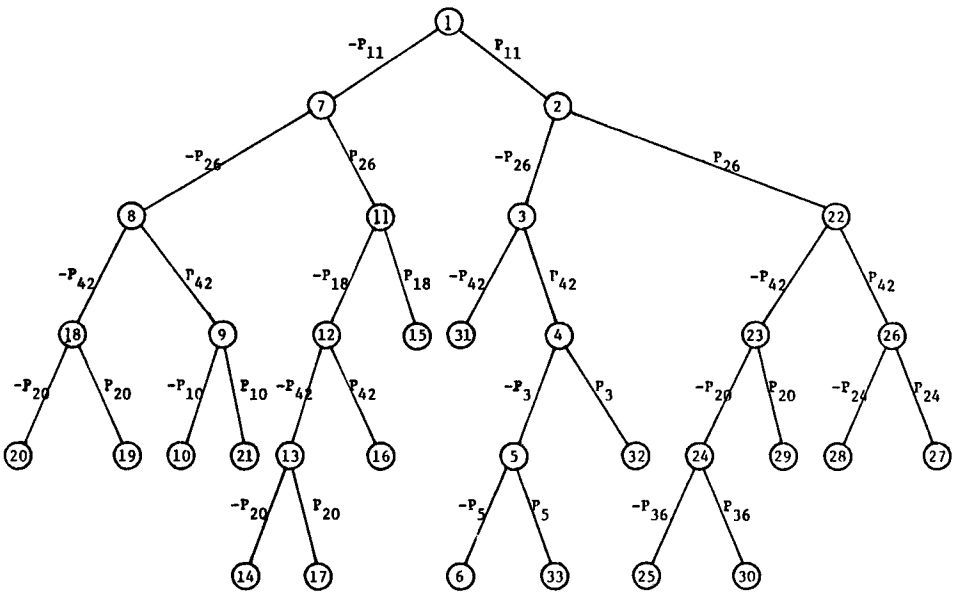

Fig. 1. An example of search tree by DPLH, an inconsistent problem with size $400 \times 50 \times 3$.

- branching letter if it has child nodes,
- node number of its left child node which results by negating the branching letter,
- node number of its right child node which results by setting the branching letter "true",
- whether its left child node is fathomed,
- whether its right child node is fathomed,
- node number of its parent node.

Fig. 1 is an example to illustrate DPLH. The numbers corresponding to nodes agree with the sequence of processing in our algorithm.

4. Experiments and results

We made an effort to obtain real-world satisfiability problems by contacting several researchers. We found that, aside from the Horn clause instances which occur in expert systems, such problems have evidently not been collected.

While in principle (by NP-completeness of satisfiability [3,7,9]) any integer program is polynomial-time convertible to a satisfiability problem, such test problems would compound the effects of the reductions with the algorithm DPLH. Moreover, for the clausal chaining (i.e. unit resolution) subroutine of DP, we used an adaptation of a linear-time algorithm of Dowling and Gallier [5], which is known to be very efficient on Horn clauses. For these reasons, we chose to use randomly-generated non-Horn problems.

We did experiments on randomly generated problems on CDC CYBER 180/855 System B. The problems were run on DPLH and APEX IV, for purposes of comparison.

FORTRAN V was used to coding. Bit mapping, maximum compilation, and other code optimizing techniques (other than all-in-core) were not used.

4.1. EXPERIMENTS ON DPLH

A program called XPGE is used to randomly generate a propositional logic problem in c.n.f. (i.e., a set of clauses). Our method is as follows:

Parameters are the number of clauses, number of letters and a lower and upper bound on clause size (which may be set equal). Each clause is generated at random as follows. Clause size is chosen equiprobably between lower and upper bounds. Then that number of letters is drawn without replacement from the set of all letters. The signs of the letters are initially chosen randomly but subsequent occurrences of a letter have sign opposite the previous one.

The DPLH control program, named XCOR, tests satisfiability of the problem generated by XPGE. XCOR consists of a main control and calling program and nearly a dozen subroutines, such as input data structure transforming, CC, MON,

HEURISTIC and node recreating, etc. The output of XCOR includes the testing result – whether the problem is satisfiable; determining an incumbent if satisfiable; and making a trace of the search tree development process.

4.2. EXPERIMENTS ON APEX IV

The propositional problems are generated by XPGE as in section 4.1. These problems are then transformed into the standard representation via integer programming before the software APEX IV is used to test satisfiability. The time for the transformation is not counted as APEX time.

4.3. RESULTS

The results of our experiments are given in tables 1–5.

In the tables, we represent problem size by: number of clauses, number of letters, and length of each clause. For example: $100 \times 50 \times 3$ means there are 100 clauses, a total of 50 letters appear in this problem, and each clause has length 3.

The column APEX/XCOR is the ratio between APEX CPU time to XCOR CPU time. Each row in the tables represents a problem with the indicated size. Tables 1 and 2 report satisfiability tests.

We also ran implications tests, which are a special kind of satisfiability test in which some of the letters are fixed to certain truth values. Such implications tests

Table 1
Satisfiability tests with three literals per clause

Problem size	Satisfiable?	DPLH		APEX		APEX/XCOR
		nodes	XCOR cpu time [s]	nodes	APEX cpu time [s]	
$100\times50\times3$	Y	14	0.109	3	0.689	6.32
$200\times50\times3$	N	79	0.586	104	21.029	35.88
$450\times50\times3$	N	33	0.515	35	19.11	37.10
$100\times100\times3$	Y	1	0.065	1	0.738	11.35
$200\times100\times3$	Y	20	0.222	10	2.287	10.30
$400\times100\times3$	N	603	7.083	Terminated without conclusion at node 748	406	> 57
$600\times100\times3$	N	219	5.020	Terminated without conclusion at node 838	940	>187

Table 2
Satisfiability tests with four literals per clause

Problem size	Satis fiable?	DPLH		APEX		APEX/ XCOR
		nodes	XCOR cpu time [s]	nodes	APEX cpu time [s]	
100×50×4	Y	11	0.109	1	0.676	6.20
200×50×4	Y	18	0.197	13	3.203	16.25
400×50×4	Y	20	0.347	Terminated without conclusion at node 201	91.971	> 265
100×100×4	Y	4	0.090	1	0.758	8.42
200×100×4	Y	22	0.256	1	1.216	4.75
400×100×4	Y	46	0.576	29	14.604	25.35
600×100×4	Y	114	2.140	Terminated without conclusion at node 618	456	> 213

parallel, for general disjunctive clause logic, the inference mechanisms of expert systems on Horn clause logic, the latter being a more restrictive framework. In this conceptualization, the clauses represent the "knowledge base" of production rules, while the fixes of letters represent database facts.

In general disjunctive clause logic, one allows for rules with conclusions that may have *more* than one conclusion, as e.g. $\neg P_1 \vee \neg P_2 \vee P_3 \vee P_4$ may represent

Table 3
Implications test (at least three free letters per clause)

Problem size	Letters fixed	Satis- fiable?	DPLH		APEX		APEX/ XCOR
			Nodes	XCOR cpu time [s]	Nodes	APEX cpu time [s]	
300×160×7	61 to 1	Y	1	0.180	1	2.324	12.9
300×160×7	61 to 1	Y	1	0.181	1	2.634	14.55
300×120×7	42 to 1	Y	1	0.174	1	2.235	12.84
300×100×7	38 to 1	Y	1	0.171	1	2.207	12.9
400×100×7	34 to 1	Y	8	0.251	1	2.777	11.06
400×60×7	19 to 1	Y	12	0.257	1	2.661	10.35
500×60×7	18 to 1	Y	14	0.321	1	3.264	10.16
500×50×7	13 to 1	Y	15	0.334	1	3.236	9.68
600×60×7	17 to 1	Y	16	0.391	1	3.881	9.92

Table 4
Search for region of "hard problems"

Problem size	Satis-fiable?	DPLH		APEX		APEX/ XCOR
		Nodes passed	XCOR cpu time [s]	Nodes passed	APEX cpu time [s]	
100×100×3	Y	1	0.066	1	0.738	11.18
100×90×3	Y	5	0.082	1	0.736	8.97
100×80×3	Y	7	0.09	1	0.728	8.08
100×70×3	Y	5	0.082	1	0.689	8.40
100×60×3	Y	11	0.107	4	0.769	7.18
100×50×3	Y	14	0.113	3	0.692	6.12
100×40×3	Y	27	0.107	12	1.546	10.51
100×35×3	Y	9	0.091	43	3.685	40.49
100×30×3	Y	18	0.119	35	3.148	26.45
100×29×3	N	33	0.182	27	2.44	13.40
100×28×3	Y	30	0.170	19	1.698	9.98
100×27×3	Y	17	0.119	29	2.572	21.61
100×26×3	N	27	0.158	41	2.874	18.18
100×25×3	N	29	0.162	26	2.41	18.87
100×23×3	N	19	0.125	27	2.219	17.75
100×20×3	N	17	0.119	19	1.693	14.22
100×17×3	N	9	0.086	9	1.151	13.38
100×14×3	N	11	0.095	12	1.078	11.34
100×12×3	N	9	0.086	7	0.925	10.75
100×10×3	N	7	0.077	7	0.853	11.08
100×8×3	N	7	0.084	5	0.715	8.51
100×5×3	N	7	0.078	6	0.611	7.83

an implication "P_1 and P_2 implies P_3 or P_4." In Horn clause logic, as in expert systems, at most one conclusion is possible for a rule. Horn clause logic is much simpler than general logic, since it is solved by clausal chaining and never requires branching [8].

Table 3 gives the implications tests. This is implemented by fixing some letters to certain truth values before executing XCOR or APEX IV. The numbers of fixed letters are restricted by the requirement that they have at least three free letters per clause after fixing. In the second column, "61 to 1" means fixing 61 letters to positive. From the test runs, implications testing is easier than satisfiability testing, and it apparently can be done with excellent response time for general disjunctive clause logic.

In Lowe's thesis [11] and in [2], a technique was given for creating a hard problem by fixing many variables of a satisfiability problem to create a "very inconsistent" problem. This inconsistent problem is then relaxed (by letting variables free) until the "boundary of consistency" is reached, where one expects

Table 5
Search for region of "hard problems" (at least one free letter in a clause after fixing)

Problem size	Letters fixed	Satisfiable?	Nodes passed	XCOR cpu time [s]
100×100×3	0	Y	1	0.066
	10 to 1	Y	1	0.080
	20 to 1	Y	1	0.093
	30 to 1	Y	1	0.073
	40 to 1	Y	1	0.082
	42 to 1	Y	1	0.079
	43 to 1	N	1	0.076
	45 to 1	N	1	0.078
	50 to 1	N	1	0.081
100×40×3	0	Y	27	0.163
	5 to 1	Y	6	0.085
	6 to 1	N	1	0.057
	7 to 1	N	1	0.059
	10 to 1	N	1	0.061
100×30×3	0	Y	18	0.125
	1 to 1	Y	10	0.106
	2 to 1	N	21	0.145
	3 to 1	N	3	0.068
	4 to 1	N	1	0.058

maximum run times. This phenomenon did not occur with DPLH or, when it did, it was much less pronounced than in [2]. (See the three problems in table 5.)

In table 4, we record another series of experiments in an attempt to locate harder problems than normal (for the size cited).

The above experimental results suggest that, on problems of the size we studied, DPLH is typically about 8 to 10 times faster than APEX IV. The harder the problem is, the more efficient DPLH is relative to the general MIP code.

Although APEX IV is a fast code of BB, it is designed for general MIP rather than for the special satisfiability problems. DPLH may have less order of magnitude improvement if compared to a BB code cleverly designed for satisfiability problems. This BB code for satisfiability problems will exploit features of propositional logic. For example, it may include unit resolution and monotone fixing as simplifications before solving each linear relaxation, or it may apply stronger rules for node fathoming.

Appendix A

The procedure of DPLH

Let Active-Problem = Current-Problem = the original problem.

LOOP-I.
  Call subroutine CC on Current-Problem.
  If inconsistency is found, then go to LOOP-II.
  If all letters are set to truth values by CC, then
    Stop, the original problem is satisfiable.
  Call subroutine MON on Current-Problem.
  If all letters are set to truth values by MON, then
    Stop, the original problem is satisfiable.
  Store the Current-Problem.
  Call subroutine HEURISTIC on Current-Problem to fix a truth value
    of one letter and derive a new problem.
  Let Current-Problem = the new problem.
  Return to the Beginning of LOOP-I.
LOOP-II.
  If Current-Problem is the original problem, then
    Stop, the original problem is inconsistent.
  Note fathoming on Current-Problem.
  If Current-Problem = Active-Problem, then
  Let Active-Problem = the parent problem of Current-Problem.
  Let Current-Problem = Active-Problem.
  If both the left and right child of Current-Problem are
    fathomed, then
    Return to the beginning of LOOP-II.
LOOP-III.
  If Current-Problem has not been branched by HEURISTIC, then
    Return to the beginning of LOOP-I.
  If one of the child problems of Current-Problem does not exist,
    then
    Let Active-Problem = Current-Problem = the opposite of the existing child
    problem,
    Return to the beginning of LOOP-I.
  If both child problems of Current-Problem are unfathomed, then
    Let Active-Problem = Current-Problem = the child problem generated more
    lately;
  otherwise
    Let Active-Problem = Current-Problem = the unfathomed child problem.
  Return to the beginning of LOOP-III.

Appendix B

Some other experiments

Experiment 1: Storing nodes out of core memory.

In the code XCOR, we "stored" the problem corresponding to each node in

Table 6
Comparison of XCOR and code XTT3 with out-of-core problem storing

Problem size	Satis-fiable?	Nodes	Out-of-core storing	In-core storing
			XTT3 cpu time [s]	XCOR cpu time [s]
100×50×3	Y	14	0.366	0.109
200×50×3	N	79	5.573	0.586
400×50×3	N	33	4.873	0.515
100×50×4	Y	11	0.315	0.109
200×50×4	Y	18	0.751	0.197
400×100×4	Y	46	2.986	0.576
100×29×3	N	33	1.301	0.182
100×28×3	Y	30	1.069	0.170
100×27×3	Y	17	0.584	0.119
100×26×3	N	27	1.078	0.158

core memory by storing the list of settings of the node. In case of re-using the node, the original problem and the list of settings are put together to recreate the problem for the node.

We tried another way. We directly stored the problems themselves on external devices (due to limited core space). Table 6 shows some experiment results in contrast to XCOR.

We see from table 6 that storing a problem in out-of-core memory is approximately 3 times or more slower than XCOR-storing in core. Although in XCOR a recreating calculation is required whenever a node is reviewed, the recreating procedure is just a linear time list processing. It is much more efficient than accessing the out-of-core memory.

Experiment 2: Another heuristic rule.

We tried to replace the heuristic rule in section 3.1 by the following rule, which was shown in [1] to produce a satisfying valuation for S in polynomial time, when $w(S) < 1$.

For each propositional letter P_j, compute

$$m_j = \left| w\left(S_j^T\right) - w\left(S_j^F\right) \right|,$$

where $w(S_j^T)$ and $w(S_j^F)$ are defined in section 3.1.

Compute $m^* = \max_j m_j = m_{j^*}$, and set P_{j^*} to T if $w(S_{j^*}^T) \geqslant w(S_{j^*}^F)$; set P_{j^*} to F if $w(S_{j^*}^F) > w(S_{j^*}^T)$.

The results are given in table 7. The new heuristic is comparable to the old one, but generally somewhat slower.

Experiment 3: Non-standard representation via IP.

The motivation to do this experiment is to represent the proposition with a tighter set of constraints than the standard representation.

Table 7
Comparison between two heuristics

Problem size	Satisfiable?	New heuristic		Heuristic in section 3.1	
		Nodes passed	XH2 cpu time [s]	Nodes passed	XCOR cpu time [s]
100×30×3	Y	41	0.212	18	0.119
100×29×3	N	61	0.369	33	0.182
100×28×3	Y	29	0.15	30	0.17
100×27×3	Y	47	0.23	17	0.119
100×26×3	N	63	0.293	27	0.158
100×100×4	Y	5	0.098	4	0.090
200×100×4	Y	22	0.27	22	0.256
100×50×4	Y	11	0.111	11	0.109
200×50×4	Y	24	0.24	18	0.197
400×50×4	Y	28	0.423	20	0.347

We next outline our procedure for a certain non-standard representation. The steps are:

(1) For each pair of letters, P and Q, tabulate $n(P, Q)$ = the number of clauses in which both $\pm P$ and $\pm Q$ occur together.

(2) Pair off letters in order of their $n(P, Q)$ using a threshold k. Specifically, first pair P and Q with maximum $n(P, Q)$. Remove all numbers $n(P, R)$ and $n(Q, R)$ for all R. Repeat as long as $n(P, Q) \geqslant k$.

(3) For each pair of paired letters, P and Q, introduce four binary variables $z(P \wedge Q)$, $z(P \wedge \neg Q)$, $z(\neg P \wedge Q)$, $z(\neg P \wedge \neg Q)$. Enter the constraint

$$z(P \wedge Q) + z(P \wedge \neg Q) + z(\neg P \wedge Q) + z(\neg P \wedge \neg Q) = 1$$

In each clause where only P occurs, enter $z(P \wedge Q) + z(P \wedge \neg Q)$; for only $\neg P$, enter $z(\neg P \wedge Q) + z(\neg P \wedge \neg Q)$. Similar for only Q or $\neg Q$.

In each clause where both P and Q occur, enter $z(P \wedge Q) + z(P \wedge \neg Q) + z(\neg P \wedge Q)$ (i.e. do *not* enter the *direct opposite* $z(\neg P \wedge \neg Q)$). If both P and $\neg Q$ occur, enter $z(P \wedge \neg Q) + z(\neg P \wedge \neg Q) + z(\neg P \wedge \neg Q)$. Similarly, where both $\neg P$ and Q appear and where $\neg P$ and $\neg Q$ appear.

(4) For a letter P which was not paired with any $Q \neq P$, due to the threshold, use $z(P)$ for P and $1 - z(P)$ for $z(\neg P)$, as in standard representations.

(5) After transforming clauses as in 3 and 4, all constraints are $\geqslant$ and all right-hand-sides are "1", as in standard treatment.

Example: A list of clauses:

$$P \vee Q, \quad \neg P \vee Q, \quad P \vee \neg Q, \quad \neg P \vee \neg Q, \quad P \vee S \vee \neg W, \quad \neg S \vee W.$$

Note that this list of clauses is not satisfiable. Compute:

$$n(P, Q) = 4, \quad n(P, S) = 1, \quad n(P, W) = 1,$$
$$n(Q, S) = 0, \quad n(Q, W) = 0, \quad n(S, W) = 2.$$

With $k = 3$, only P and Q are paired, we obtain the constraints:

$$z(P \wedge Q) + z(P \wedge \neg Q) + z(\neg P \wedge Q) + z(\neg P \wedge \neg Q) = 1, \quad (1)$$

$$z(P \wedge Q) + z(P \wedge \neg Q) + z(\neg P \wedge Q) \geqslant 1, \quad (2)$$

$$z(\neg P \wedge Q) + z(\neg P \wedge \neg Q) + z(P \wedge Q) \geqslant 1, \quad (3)$$

$$z(P \wedge \neg Q) + z(\neg P \wedge \neg Q) + z(P \wedge Q) \geqslant 1, \quad (4)$$

$$z(\neg P \wedge \neg Q) + z(P \wedge \neg Q) + z(P \wedge \neg Q) \geqslant 1, \quad (5)$$

$$z(P \wedge Q) + z(P \wedge \neg Q) + z(S) + 1 - z(W) \geqslant 1, \quad (6)$$

$$1 - z(S) + z(W) \geqslant 1. \quad (7)$$

If we add constraints (2), (3), (4), (5), we have

$$3z(P \wedge Q) + 3z(\neg P \wedge Q) + 3z(P \wedge \neg Q) + 3z(\neg P \wedge \neg Q) \geqslant 4,$$

which contradicts the constraint (1) in the linear relaxation. This means the LR will be inconsistent for this new representation (but not for the standard one). Therefore, the non-standard representation is tighter than the standard one.

Table 8
Standard vs non-standard representation

Problem size	Satis-fiable	Standard representation		Non-standard representation			
		Nodes	APEX time [s]	K	Number of paired variables	Nodes	APEX time [s]
100×60×3	Y	4	0.769	3	0	4	0.774
100×35×3	Y	43	3.685	4	10	43	3.692
				3	3	65	5.578
100×27×3	Y	29	2.572	4	4	16	2.235
				3	7	18	2.614
100×26×3	N	41	2.874	5	0	41	2.882
				4	5	29	3.109
				3	8	69	7.654
100×20×3	N	19	1.693	5	1	21	1.811
				4	6	21	2.337
				3	8	32	3.266
				2	9	26	3.088
				1	9	26	3.074
100×10×3	N	7	0.853	15	0	7	0.799
				11	1	6	0.855
				10	2	4	0.919
				5	4	8	1.128
				3	4	8	1.158
				1	4	8	1.177
100×5×3	N	6	0.611	15	2	5	0.818
				10	2	5	0.817
				5	2	5	0.830

In table 8, we give the experimental results along with the corresponding results of using the standard representation. We can see that by properly choosing the threshold k, sometimes a problem solves in fewer nodes, but not always. Even though a smaller number of nodes is created, the total CPU time is usually longer than with the standard form. As a special-purpose code was not developed for this non-standard representation, we used APEX IV to solve these problems.

Experiment 4: Experiment on "balls and slots problem".

The "balls and slots" problem was suggested to us by R.M. Karp. It is described as follows.

Suppose we have b balls to be placed in s slots, each of which can hold exactly one ball. Let $P(i, j)$ denote that ball i is placed in slot j. We may describe this problem as a set of logic clauses:

(1) Every ball is placed somewhere:

$$P(i, 1) \vee P(i, 2) \vee \ldots \vee P(i, s); \quad \text{for } i = 1, \ldots, b.$$

(2) No ball occurs in two different slots:

$$\neg P(i, j) \vee \neg P(i, j'), \text{ where } j \neq j'; \quad \text{for } i = 1, \ldots, b.$$

(3) No two different balls occur in the same slot:

$$\neg P(i, j) \vee \neg P(i', j), \text{ where } i \neq i'; \quad \text{for } j = 1, \ldots, s.$$

Thus, with b balls and s slots, there are $b + b\binom{s}{2} + s\binom{b}{2} = \frac{1}{2}b(2 + s^2 - 2s + sb)$ clauses.

Obviously, if $s < b$, the task of placing balls in slots cannot be accomplished since there are too few slots, i.e. the above set of clauses cannot be satisfied.

Our experiments are on a series of problems with b balls and $b - 1$ slots. Each of the problems is inconsistent. Table 9 shows the results of the experiments. For each problem, we use two codes, DPLH and DPLH-SEQ, in order to make a comparison. The difference between the two codes is on the heuristic rule for

Table 9
Results on the balls and slots problem

b	s	Number of clauses	DPLH-SEQ		DPLH	
			nodes (tree size)	CPU time [s]	nodes (tree size)	CPU time [s]
2	1	3	1	–	1	–
3	2	12	3	0.041	3	0.045
4	3	34	11	0.045	13	0.061
5	4	75	47	0.125	61	0.222
6	5	141	239	0.525	327	1.127
7	6	238	1439	3.293	2055	9.601

splitting. In DPLH-SEQ, whenever we need to branch, we use the lowest-indexed available branching variable, with this ordering among indices:

$(i, j) < (i', j')$ if (1) $i' > i$, or (2) $i' = i$, $j' > j$.

Remarks on the results:

(1) Based on the above data, the empirical formula for the size, N_b, of the search tree by the code DPLH-SEQ may be put into a recursive form:

$N_2 = 1$,

$N_b = (N_{b-1} + 1)(b - 1) - 1$ for $b \geqslant 3$.

Using this formula for extrapolation, the expected sizes of search trees of larger problem are listed below:

b	s	tree size	clauses	variables
8	7	10079	372	56
9	8	80639	549	72
10	9	725759	775	90

(2) DPLH is always less efficient in our test problems than DPLH-SEQ, due to the specific structure of these problems.

(3) Considering the problem size and tree size, this series of problems is the hardest problem we have met.

Propositional logic codes like DPLH have no means of "recognizing the pattern" in the "balls and slots" problem. For example, if the fact that $(b - 1)$ balls cannot be put into $b - 2$ slots were stored in some "long term memory", DPLH-SEQ would require only $2b - 3$ nodes to discover that b balls cannot be put into $(b - 1)$ slots. With "induction schema" in addition (which goes beyond propositional logic), proofs for the "balls and slots" problem can be made of constant length.

The deficiency of DPLH as revealed by this experiment, we believe, reflects a deficiency of unstructured, memoryless propositional logic to represent general forms of human reasoning, rather than any inefficiency of the algorithm itself.

References

[1] C.E. Blair and R.G. Jeroslow, unpublished notes.

[2] C.E. Blair, R.G. Jeroslow and J.K. Lowe, Some results and experiments on programming techniques for propositional logic (January 1986) to appear in Comp. Oper. Res.

[3] S.A. Cook, The complexity of theorem proving procedures, *Proc. 3rd SIGACT Symp.* (1971) pp. 151–158.

[4] M. Davis and H. Putnam, A computing procedure for quantification theory, J. ACM 8 (1960) 201–215.

[5] W.F. Dowling and J.H. Gallier, Linear-time algorithms for testing the satisfiability of propositional Horn formulae, Logic Programming 3 (1984) 267–284.

[6] J. Franco and M. Paull, Probabilistic analysis of the Davis–Putnam procedure for solving the satisfiability problem, Discr. Appl. Math. 5 (1983) 77–87.
[7] M. Garey and D. Johnson, *Computers and Intractibility* (W.H. Freeman, 1979).
[8] R.G. Jeroslow, Notes for Rutgers Lectures, Mixed integer model formulation for logic-based decision support (1986).
[9] R.M. Karp, Reducibility among combinatorial problems, in: *Complexity of Computer Computations*, eds. R.E. Miller and J.W. Thatcher (Plenum, New York, 1972) pp. 85–104.
[10] D.W. Loveland, *Automated Theorem Proving: A Logical Basis* (North-Holland, Amsterdam, 1978).
[11] J.K. Lowe, Modelling with integer variables, Ph.D. Thesis, Georgia Institute of Technology (March 1984).
[12] A.J. Nevins, A human oriented logic for automatic theorem-proving, J.ACM 21 (1974) 606–621.
[13] P.W. Purdom, Jr., Solving satisfiability with less searching correspondence, IEEE Trans. Pattern Anal. Machine Intell. PAMI-6 (July 1984).