The Knowledge Engineering Review, Vol. 15:1, 2000, 31–45
Printed in the United Kingdom

Combining satisfiability techniques from AI and OR

HEIDI E. DIXON and MATTHEW L. GINSBERG

CIRL, 1269 University of Oregon, Eugene, OR 97403-1269, USA (email: {dixon, ginsberg}@cirl.uoregon.edu)

Abstract

The recent effort to integrate techniques from the fields of artificial intelligence and operations research has been motivated in part by the fact that scientists in each group are often unacquainted with recent (and not so recent) progress in the other field. Our goal in this paper is to introduce the artificial intelligence community to pseudo-Boolean representation and cutting plane proofs, and to introduce the operations research community to restricted learning methods such as relevance-bounded learning. Complete methods for solving satisfiability problems are necessarily bounded from below by the length of the shortest proof of unsatisfiability; the fact that cutting plane proofs of unsatisfiability can be exponentially shorter than the shortest resolution proof can thus in theory lead to substantial improvements in the performance of complete satisfiability engines. Relevance-bounded learning is a method for bounding the size of a learned constraint set. It is currently the best artificial intelligence strategy for deciding which learned constraints to retain and which to discard. We believe these two elements or some analogous form of them are necessary ingredients to improving the performance of satisfiability algorithms generally. We also present a new cutting plane proof of the pigeonhole principle that is of size n^2, and show how to implement some intelligent backtracking techniques using pseudo-Boolean representation.

1 Introduction

Imagine you are working on a large jigsaw puzzle, and you are busy collecting and fitting together some red pieces that you hope will become a barn settled into a scenic landscape. At some point, it seems as if none of the pieces you have can be combined any further and your progress has halted completely. You then begin to suspect that the person across from you, who you thought was working on the blue sky, might in fact have some of the red pieces you need to finish your section. It is also likely that you have some of their missing pieces. The obvious solution is to have a look at their pieces and see if there is anything that you can use.

This is a fair metaphor for the current relationship between the fields of Artificial Intelligence (AI) and Operations Research (OR). The development of successful methods to solve constraint satisfaction problems is of great interest to both communities, but until recently, the two fields have seldom collaborated. The fields have evolved independently, use different techniques, and each has a unique framework for approaching problems. It is only recently that there have been attempts to build algorithms integrating techniques from both fields. A unifyng framework for understanding the connections between methods combined with new hybrid approaches (Hooker et al., 2000; Wolfman and Weld, 2000), will provide better solutions to difficult problems.

Our goal in this paper is to examine the relative strengths and weaknesses of AI and OR approaches to solving propositional satisfiability problems (SAT). As we will see, the two approaches have different strengths and weakness. We discuss these differences, and suggest ways that the strengths of both fields might be combined.

The SAT problem is a constraint satisfaction problem in which an instance is defined by a set of boolean variables $U = \{v_1, v_2, \ldots, v_n\}$ and a set of clauses $C = \{c_1, c_2, \ldots, c_m\}$. A clause is a

disjunction of literals, where a literal is a boolean variable v_i, or its negation $\overline{v}_i$. A clause is satisfied if and only if any one of its literals evaluates to true. A solution to a SAT problem is an assignment of values to variables that satisfies every clause; if no such assignment exists, the instance is called unsatisfiable. An algorithm for solving satisfiability problems is generally called complete if it is capable of determining conclusively whether any given problem instance is unsatisfiable; in this paper, we restrict our attention to such complete methods.

In fact, we will go further still, restricting our attention to the application of complete methods to unsatisfiable problems. The reason for this is that any complete depth-first backtracking method, when attempting to solve even a satisfiable problem P, will of necessity spend the bulk of its time working on subproblems of P that are in fact unsatisfiable. After all, if the overall goal is to assign values to n variables, and the machine only backtracks when a particular partial assignment cannot be extended to a total solution, there will be only n "forward" steps in the overall search for a solution. Every other step will be part of the analysis of an unsatisfiable subproblem.

The methods used by both AI and OR attempt to show a problem to be unsatisfiable by deriving an explicit contradiction, either the empty disjunction (in the AI case), or $0 \geq 1$ (in the OR case). We will discuss the techniques used by the two fields in two phases:

First, in section 2 we discuss the representations used. The AI community has generally used the syntax of Propositional Logic (PL); the OR community has used Cutting Planes (CP). Along with a representational choice comes a choice of basic inference step; as we will see, inference using cutting planes (where the basic inference is a *cut*) can be exponentially more efficient than inference using propositional logic (where the basic inference is a *resolution step*). The punch line here is that the OR community has it basically right, and the AI community basically wrong: The CP representation is simply more efficient than the PL representation.

In section 3, we discuss the methods used to control the search of an inference engine. Both communities start from the same basic premise, which is to split the subspace in two and show the resulting subproblems to be individually unsatisfiable. Both use similar techniques for doing this. The AI community, however, has developed powerful learning techniques that allow results from one portion of the proof tree to be used effectively in others. So here, it seems that the situation is reversed: the AI camp has made substantial progress beyond basic chronological backtracking, while the OR camp has yet to do so.

Finally, in section 4, we suggest that it is possible to have the best of both worlds. We show that the learning methods developed by AI can be combined effectively with the representation used by OR, hopefully improving the efficiency of the tools available to both groups. Conclusions and suggestions for future work are contained in section 5.

2 Representation and inference

2.1 AI: Conjunctive normal form and resolution

AI approaches to satisfiability typically represent the constraints as a conjunctive list of disjunctions. The fundamental inference step is *resolution*, which we write as:

$$\frac{\begin{array}{c} a_1 \vee \cdots \vee a_k \vee l \\ b_1 \vee \cdots \vee b_m \vee \neg l \end{array}}{a_1 \vee \cdots \vee a_k \vee b_1 \vee \cdots \vee b_m}$$

In other words, given two disjunctions, one of which includes a literal l and the other of which includes its negation $\neg l$, we derive a new disjunction that is formed by disjoining the originals while dropping l and $\neg l$. If a single literal appears in both the a_i and the b_i, it is included only once in the conclusion of the resolution; this is known as *factoring*.

Resolution is sound because if l is true, the large disjunction holds as a consequence of the second of the two resolvents; if l is false, it holds as a consequence of the first. Resolution can be shown to be complete as well.

Resolution-based methods are prevalent because (we assume) of their simplicity. Resolution is easier to control and to implement than many stronger systems. Unfortunately, resolution is also a fairly weak proof method in that many unsatisfiable problems do not have short resolution proofs of unsatisfiability.

As an example, consider the pigeonhole problem, which involves showing that you cannot put $n+1$ pigeons into n holes with each pigeon getting its own hole. If we write p_{ij} to mean that pigeon i is in hole j, then the condition that no two pigeons share a hole becomes

$$\neg p_{ik} \vee \neg p_{jk}$$

for any hole k and distinct pigeons $i \neq j$. The requirement that all pigeons have some hole is

$$p_{i1} \vee \cdots \vee p_{in}$$

for each pigeon i.

It is possible to show that these axioms are collectively unsatisfiable, but the shortest resolution proof that does so involves a number of steps that is exponential in n (Haken, 1985). In fact, problems similar to the pigeonhole problem are sufficiently common that it is possible to show that even the *average* number of resolution steps needed to prove unsatisfiability for randomly generated problems grows exponentially with problem size (Chvátal and Szemerédi, 1988). This produces a significant theoretical limitation on resolution-based proof methods, since it is clear that the running time of such a method will never be less than the length of the shortest resolution proof. Many of the systematic methods in use in the AI community appear to be resolution based (Baker, 1995; Mitchell, 1998), and will therefore suffer from this difficulty.

2.2 OR: Cutting planes

The cutting plane proof system (CP) originated from an algorithm for general integer programming created by Gomory (1963). The algorithm was rarely used in practice because it converged slowly, but it was recognized by Chvátal (1983) that the method could function as a proof system. There are many studies examining the complexity and strength of the CP proof system (Bonet et al., 1997; Cook et al., 1987; Goerdt, 1990; Pudlák, 1997), and it was shown early on by Cook that the CP system is properly stronger than resolution in that existence of a polynomial-length resolution proof implies the existence of a polynomial-length CP proof, but the reverse need not hold (Cook et al., 1987).

Constraints in CP are expressed as linear inequalities

$$\sum a_j x_j \geq k$$

where $x_1, x_2, \ldots, x_n$ are non-negative integer variables and $a_1, a_2, \ldots, a_n$ and k are integers. The system has two rules of inference: (i) derive a new inequality by taking a linear combination of a set of inequalities, (ii) given an inequality $\sum a_j x_j \geq k$ derive $\sum (a_j/d) x_j \geq \lceil \frac{k}{d} \rceil$, where d is a positive integer that divides each a_j evenly. The notation $\lceil q \rceil$ denotes the least integer greater than or equal to q. A derived inequality of this type is called a *cut*. If $0 \geq 1$ can be derived in this fashion, the original set of inequalities is inconsistent.

Simulating resolution using cuts We can use the CP system to simulate proofs in propositional logic. Propositional clauses are written as linear inequalities with propositional variables restricted to values of $0 =$ `false`, and $1 =$ `true`. A disjunction of literals

$$x_0 \vee x_1 \vee \cdots \vee x_n$$

can be equivalently written as a linear pseudo-Boolean inequality:

$$x_0 + x_1 + \cdots + x_n \geq 1$$

The variable $\overline{x}$ refers to the negation of the variable x, so that for all literals x, $\overline{x} = 1 - x$. This type of representation is commonly used by the operations research community to describe boolean expressions. The more general form for linear pseudo-Boolean inequalities

$$a_0 x_0 + b_0 \overline{x}_0 + a_1 x_1 + b_1 \overline{x}_1 \cdots + a_n x_n + b_n \overline{x}_n \geq r$$

allows for real coefficients a_i, b_i and r (Hammer and Rudeanu, 1968). Resolution can now be simulated as follows: given two clauses $C_1 = p \vee \bigvee_i x_i$ and $C_2 = \overline{p} \vee \bigvee_i y_i$

$$\frac{p \vee \bigvee_i x_i \quad \overline{p} \vee \bigvee_i y_i}{\bigvee_i x_i \vee \bigvee_i y_i}$$

can be written as

$$\frac{p + \sum_i x_i \geq 1 \quad p + \sum_i y_i \geq 1}{\sum_i x_i + \sum_i y_i \geq 1}$$

The derived inequality follows because $p + \overline{p} = 1$. Factoring takes the following form:

$$\frac{\begin{array}{rl} p + p + \sum_i x_i & \geq 1 \\ \sum_i x_i & \geq 0 \end{array}}{\begin{array}{rl} 2p + 2\sum_i x_i & \geq 1 \\ p + \sum_i x_i & \geq \lceil \frac{1}{2} \rceil \end{array}}$$

Rounding the fraction $\frac{1}{2}$ to 1 by virtue of the CP rule of integer rounding now gives us the inequality $p + \sum_i x_i \geq 1$.

Before we move on to examine proof lengths in CP, note the expressive power of linear inequalities for propositional logic. In addition to disjunctions, linear inequalities can also be used to express more complicated constraints like cardinality constraints

$$x_1 + x_2 + \cdots + x_n \geq k$$

where k is a constant. This requires that at least k of the x_i be true. Here is another example:

$$ny + \sum_{i=1}^{n} x_i \geq n$$

where n is a constant. This is equivalent to the logical expression $y \bigvee (x_1 \wedge x_2 \wedge \cdots \wedge x_n)$. These types of concise expressions can make a constraint set substantially smaller than the equivalent set of disjunctive clauses.

Proof length in CP The CP proof system is properly more powerful than resolution. We saw above that any resolution proof can be simulated as a CP proof of equal length; there are also examples of problems where the shortest resolution proof is exponential but the shortest CP proof is polynomial. CP always does as well as resolution and sometimes does exponentially better.

As an example, let us return to the pigeonhole problem, known to require a proof of exponential length using resolution. It is known that there is a CP proof of length n^3 (Cook et al., 1987); we give an alternate pigeonhole proof in CP of length n^2.

Written as a system of inequalities, the pigeonhole problem becomes

$$\sum_{k=1}^{n} p_{ik} \geq 1 \quad i = 1, \ldots, n+1 \tag{1}$$

$$\overline{p}_{ik} + \overline{p}_{jk} \geq 1 \quad i \neq j, k = 1, \ldots, n \tag{2}$$

The variable $p_{ik} = 1$ continues to mean that pigeon i is in hole k. Inequality (1) says that every

pigeon is in some hole. Inequality (2) says that two different pigeons cannot share the same hole. To solve the problem, it is sufficient to derive the inequalities

$$\sum_{i=1}^{j+1} \overline{p}_{ik} \geq j \tag{3}$$

for $j < n + 1$. These inequalites tell us that at most 1 of the first $j + 1$ pigeons can be in a given hole k. To state the general case that at most 1 of the pigeons can be in a given hole, we let $j = n$ and write,

$$\sum_{i=1}^{n+1} \overline{p}_{ik} \geq n \tag{4}$$

The inequality $0 \geq 1$ can be obtained by summing (1) and (4) over i and k respectively and adding the results.

We will derive (3) by induction on j. The base case $j = 1$ is contained in the initial inequalities. For the inductive step, assume (3) is true for a given j:

$$\begin{array}{llllll}
j(\overline{p}_{1k}+ & \overline{p}_{2k}+ & \cdots & +\overline{p}_{j+1,k}) & & \geq j^2 \\
\overline{p}_{1k}+ & & & & \overline{p}_{j+2,k} & \geq 1 \\
& \overline{p}_{2k}+ & & & \overline{p}_{j+2,k} & \geq 1 \\
& & & \overline{p}_{j+1,k}+ & \overline{p}_{j+2,k} & \geq 1 \\
\hline
\overline{p}_{1k}+ & & \cdots & & +\overline{p}_{j+2,k} & \geq \frac{j^2+j+1}{j+1} = j + \frac{1}{j+1}
\end{array}$$

The right-hand side of the final equation rounds up to $j + 1$. ■

It is important to observe the role factoring plays in these proofs. In the CP system, factoring corresponds to integer rounding. Descriptions of resolution often focus on the resolution rule alone, downplaying the role of factoring in proof construction. This is somewhat misleading: Were it not for factoring, inference using CP would consist simply of taking linear combinations of earlier conclusions to produce new ones. These steps could all be combined, providing proofs of length one in all cases (and implying that NP = co-NP). The strength of the CP system may be that it does not require the use of resolution or cancellation steps to locate factoring opportunities. In the pigeonhole proofs, for example, factoring opportunities are derived from clauses that cannot be resolved together at all. In resolution proofs, factoring can only be performed on a resolvent of two clauses.

State-of-the-art satisfiability algorithms in AI can be adapted to use linear pseudo-Boolean inequalities. We will see in section 4, that changing the underlying representation has very little effect on the efficiency of the algorithm, so there is little to lose for the AI community in making this change. With this change comes the possibility of taking advantage of the stronger cutting techniques that pseudo-Boolean allows.

Very little is known about how to control the construction of cutting plane proofs for satisfiability. It is possible to generate polynomial time constructions for certain polynomial proofs, including the pigeonhole proof above, but it is currently unclear whether such methods will lead to practical implementations.

3 Controlling search

3.1 OR: Branching, backtracking and cuts

The goal of this section is to familiarize AI readers with some standard integer programming techniques from OR, and to then show how these techniques have been applied to solve the satisfiability problem. The OR community solves satisfiability problems by converting clauses in CNF to linear pseudo-Boolean inequalities and then applying the techniques that we are about to describe. A pseudo-Boolean problem is an instance of a general problem class known as *integer programming*.

An integer programming problem is an optimization problem in which all variables are restricted

to have nonnegative integer values. It can be described by a set of constraints and an objective function of the form

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to:} & Ax \leq b \\ & x \in Z_+^n \end{array}$$

where $A \in \Re^{m \times n}$ is an $m \times n$ real-valued array, and $b \in \Re^m$ and $c \in \Re^n$ are real-valued vectors. The goal is to find a solution that does not violate any constraints and gives the optimal value of the objective function.

Branch-and-bound is the classic approach to solving the integer programming problem. Within the standard framework of a branch-and-bound algorithm, there are many different strategies for partitioning subproblems, node selection and preprocessing constraints. There are also many methods that are derivatives of the branch-and-bound approach such as branch-and-cut. There are too many such variations for us to cover them all in any depth; our focus here will be on those strategies that are most commonly used and bear most directly on satisfiability. For a more detailed description of linear programming and integer programming, we suggest texts by Chvátal (1983) and Nemhauser and Wolsey (1988).

Branch and Bound The underlying idea of the branch-and-bound method is to find a feasible integer solution early in the search process, and use this solution to prune unproductive areas of the search space.

Somewhat more specifically, any integer solution to $Ax \leq b$ places a lower bound on the optimal integer solution. An integer solution used in this way is called an *incumbent* solution. If the value of the objective function for any given subproblem is less than the current lower bound for the optimal solution, then the subproblem cannot contain the optimal solution and can be pruned from the search space.

Branch-and-bound is an enumeration-based algorithm in which a relaxation of the integer problem is solved at each node. The most common relaxation used is the linear programming relaxation, which removes the integer constraints and allows variables to have fractional values. This creates a continuous version of the problem that can be solved by a linear programming solver. Different solvers can be used, but the simplex method is the most prevalent.

The solution to the linear relaxation is considered an approximation to the integer solution. Nonintegral solutions found in this fashion can be used to direct the search process. If the solution to the continuous problem is integer, then it may become the incumbent solution used to prune unpromising areas of the search space. The branch-and-bound algorithm is outlined below:

1 *Initialization*. The original integer problem is added to the list of subproblems L to be solved. There is no incumbent solution.
2 *Termination*. If the list of subproblems L is empty, the algorithm terminates. The current incumbent solution is the optimal solution. If no incumbent solution has been found, the problem is infeasible.
3 *Problem Selection and Relaxation*. A subproblem is selected from the list L and a linear programming relaxation is run to find the optimal solution x^* to the continuous problem.
4 *Pruning and Fathoming*.

- If the value of the objective function on x^* is less than or equal to the bound provided by the incumbent solution or there is no x^* because the problem is infeasible, then this subproblem can be pruned or *fathomed*. Go to step 2.
- If the value of the objective function of the relaxed problem is greater than the minimum bound provided by the incumbent solution and the solution x^* is integer, then it becomes the new incumbent solution. The list of subproblems L is scanned for any problems that have objective function values less than the new incumbent. These problems are removed from the list. Go to step 2.

- If the value of the objective function of the relaxed problem is better than that of the incumbent but is fractional, the current problem is partitioned into subproblems. These problems are added to the list L.

The most common node selection strategy is depth first search with backtracking. If the solution to the linear relaxation yields a fractional value x_i^* as the optimal value for variable x_i, then the range

$$\lfloor x_i^* \rfloor < x_i^* < \lceil x_i^* \rceil \tag{5}$$

cannot contain any feasible solutions to the integer problem. The problem is partitioned into two new subproblems, one with the constraint $x_i \leq \lfloor x_i^* \rfloor$ and one with $x_i \geq \lceil x_i^* \rceil$.

Branch and Cut Branch and cut is a branch and bound algorithm that applies additional cutting techniques. Again, a linear relaxation is solved at each node, but if the solution is fractional, there is the option of generating and adding separating cuts to the constraint set. A separating cut is a linear inequality that is satisfied by all the integer points, but is violated by the fractional solution generated by the linear relaxation. The addition of separating cuts eliminates parts of the solution space that contain no integer solutions. The resulting relaxation approximates the integer programming problem more closely, and can provide better direction for the search process.

Cuts can be generated through a variety of different methods. A common method is the generation of Gomory cuts described in the beginning of section 2.2. A thorough discussion of cut generation methods is beyond the scope of this paper, but we refer the interested reader to Nemhauser and Wolsey (1988).

3.2 Integer programming methods and SAT

We can formulate an instance of a satisfiability problem with variables $\{x_1, x_2, \ldots, x_n\}$ and clauses $\{c_1, c_2, \ldots, c_m\}$ as an integer programming problem as follows:

$$\begin{array}{lll} \text{minimize} & w & \\ \text{subject to:} \quad w + \sum_{y_i \in c_j^+} x_i + \sum_{y_i \in c_j^-} (1 - x_i) \geq 1 & & j = 1 \ldots m \\ & x \in 0, 1 & i = 1 \ldots n \\ & w \geq 0 & \end{array}$$

where c_j^+ is the set of positive literals in c_j, and c_j^- is the set of negative literals in c_j. The variable w is an auxiliary variable that is added to ensure we have a feasible starting solution. An instance is unsatisfiable if the value of w is greater than zero in the optimal solution to the linear relaxation.

Branch-and-bound and branch-and-cut methods have provided good solutions to many integer problems, but they do not perform well on satisfiability problems. This is because the integer programming formulation of the SAT problem is weak. A feasible solution to the linear relaxation can be obtained by fixing variables that appear in unit clauses to one or zero as necessary, and setting the remaining variables to 1/2. This provides very little direction to the search. The linear relaxation adds some value, but is equivalent to the AI technique of unit propagation (described in the next subsection) (Blair et al., 1986). In addition, the substantial overhead needed to run a linear programming solver makes it slow and impractical. Because of this, most OR methods for SAT are hybrids, often incorporating some of the standard AI techniques as well (Gallo and Urbani, 1989; Hooker and Fedjki, 1990; Jeroslow and Wang, 1990).

An example of this is Hooker's implementation of branch and cut (Hooker and Fedjki, 1990). In this branch and cut algorithm, unit propagation is applied to the constraint set before the linear relaxation is solved and again applied after generated cuts are added. The method used to find seperating cuts is based on resolution. Cuts are generated by simulating resolution with linear inequalities. Usually, a number of cuts must be generated before a separating cut is found. Cuts are not generated for nodes below depth 4 because cuts generated at deep nodes do not decrease the search space enough to warrant the cost of their generation. Below nodes of depth 4 the linear relaxation is no longer solved and instead a standard AI algorithm (Davis–Putnam–Loveland) is run.

We note in passing that the column subtraction method (Harche and Thompson, 1994) is an

integer programming algorithm that seems to be doing something unique. It was originally written to solve set covering problems, but can be extended to solve satisfiability problems (Harche et al., 1994). It is based on a standard branch and bound algorithm where a linear relaxation is performed at each node, but it adds the technique of subtracting some nonbasic columns from the right-hand side of the optimal simplex tableau. The column subtraction method was included in a computational study of SAT algorithms developed by the operations research community (Harche et al., 1994). The algorithm did not always perform the best, but consistently gave good performance overall. It did particularly well on some combinatoric problems (such as the pigeonhole problem), where it was the only algorithm that could solve some of the instances. A better understanding of how this method differs from other methods would be useful.

3.3 AI: Learning and restricted learning

As the basic OR technique for solving satisfiability problems is branch-and-bound, the basic AI technique is the classic Davis–Putnam–Loveland method (Davis and Putnam, 1960; Loveland, 1978).

Davis–Putnam–Loveland The DPL procedure takes a valid partial assignment and attempts to extend it to a valid total assignment by incrementally assigning values to variables. This creates a binary search tree where each node corresponds to a set of variable assignments. If the algorithm reaches a dead end (where there is no valid assignment for a variable), it backtracks. DPL uses a procedure called unit propagation, which identifies clauses with two properties: First, they have a single unvalued literal and second, all other literals are valued to false. The procedure locates these clauses and sets the unvalued literal to `true`. It continues to do this until no clause with these properties exists, or an inconsistency is detected. The algorithm terminates when a valid total assignment has been found or when it can prove that no solution exists. DPL is a relatively simple algorithm and is easy to implement. The method underlies almost all complete approaches to solving the satisfiability problem. In pseudocode, we have:

Procedure 3.1 (Davis–Putnam–Loveland) *Given a SAT problem S and a partial assignment of values to variables P*, to compute `solve`(C, P):

if `unit-propagate`(P) *fails*, **then** return failure
else set $P :=$ `unit-propagate`(P)
if all clauses are satisfied by P, **then** return P
$v :=$ an atom not assigned a value by P
if `solve`$(C, P \cup (v :=$ `true`$))$ succeeds, **then** return it
else return `solve`$(C, P \cup (v :=$ `false`$))$

As noted above, the procedure begins by *unit propagating*, a polynomial-time procedure that assigns forced values to atoms. If unit propagation reveals the presence of a contradiction, we return failure. If it turns P into a solution, we return that solution. Otherwise, we pick a branch variable and try binding it to true and to false in succession. In practice, some effort is made to select v and order the values for it in a way that is likely to lead to a solution quickly.

Procedure 3.2 (Unit propagation) to compute `unit-propagate`(P):

while there is a currently unsatisfied clause $c \in C$ that contains
at most one literal unassigned a value by P **do**
if every atom in c is assigned a value by P, **then** return failure
else $a :=$ the atom in c unassigned by P
augment P by valuing a so that c is satisfied
end if
end while
return P

When applied to satisfiability problems, DPL is essentially indistinguishable from the branch-and-bound method used in OR (Blair et al., 1986). The DPL partitioning strategy of breaking the problem into two subproblems, one with $v = \texttt{true}$, and one with $v = \texttt{false}$, is equivelent to the branch-and-bound strategy of partitioning the subproblem around some non-integer x^*, creating two subproblems with constraints $x^* \leq 0$ and $x^* \geq 1$ added, respectively. We have already remarked that unit propagation on sets of disjunctions derives consequences identical to those found by linear programming techniques applied to the corresponding set of inequalities.

Performance of DPL can be improved in a variety of ways; once again, there is not adequate space for a comprehensive survey of all these methods, so we have chosen instead to highlight two:

1 **Branching heuristics:** the original DPL procedure used a fixed order to select a variable for branching. The addition of a simple branching heuristic can have a substantial impact on the size of the search tree. We describe current branching heuristics and discuss their relative merits.
2 **Backtracking schemes:** backtracking algorithms are particularly susceptible to a condition called thrashing. This refers to a variety of situations in which time is spent exploring parts of the search space that cannot contain solutions. Intelligent backtracking procedures are designed to avoid these problems. The two intelligent backtracking techniques we will describe are learning and restricted learning.

Branching Heuristics Branching rules play an important role in reducing the size of the search space for satisfiability problems (Crawford and Autan, 1996; Hooker and Vinay, 1995). A survey of the references in this section show that the AI and OR communities have both contributed to the literature on branching heuristics. The branching decisions made by algorithms used by both fields are analogous, and both fields have reached similar conclusions: the branch variable should be chosen so as to minimize the size of the resulting subproblem. Most branching rules are designed to encourage a cascade of unit propagations. The result of such a cascade is a smaller and more tractable subproblem.

There are two popular classes of branching rules that are based on this idea. One is the MOMS rule, which branches on the variable that has maximum occurrences in minimum size clauses (Crawford and Autan, 1996; Dubois et al., 1993; Hooker and Vinay, 1995; Jeroslow and Wang, 1990; Pretolani, 1993). Another heuristic is the unit propagation rule (Crawford and Autan, 1996; Freeman, 1995). This rule calculates the full amount of propagation caused by a branching choice. Given a branching candidate v_i, the variable is independently fixed to `true` and `false` and the unit propagation procedure is run on each subproblem. The number of unit propagations caused by an assignment becomes a weight used to evaluate branching choices.

Rules of the MOMS type approximate the number of unit propagations that a particular variable assignment is likely to cause; the unit propagation rule computes the number exactly. While there is clear computational expense in performing this calculation, Li and Anbulagan found that this cost is more than balanced by the benefit of reducing the size of the search space (Li and Anbulagan, 1997). Applying the unit propagation rule to all free variables of every search node significantly outperforms the MOMS rule for random 3-SAT problems. Better performance still can be achieved by using a MOMS-like heuristic to select a few good candidates and then selecting among these candidates using an exact computation (Crawford and Autan, 1996; Li and Anbulagan, 1997). The MOMS-like heuristic that appears to be the most effective is the simple one of counting the number of occurrences of a given literal in clauses with exactly two unsatisfied literals (in other words, the number of "immediate" unit propagations that would be enabled by setting the variable) (Crawford and Autan, 1996; Li and Anbulagan, 1997).

Learning A drawback to using naïve backtracking algorithms in solving a satisfiability problem is that one may end up solving the same subproblems repeatedly. To understand how this can happen, consider the following example.

We are solving a SAT problem with variables $x_1, x_2, \ldots, x_{100}$, and have successfully valued the variables $x_1, \ldots, x_{49}$. In this problem there happens to be a subset of constraints involving only the

variables $x_{50}, \ldots, x_{100}$ that together imply that $x_{50} = \texttt{true}$. If we begin by setting $x_{50} = \texttt{false}$, it will require some degree of searching to discover our mistake. When we finally do, we backtrack to x_{50}, set it to `true`, and continue on. Unfortunately, if later we need to backtrack to a variable set before x_{50}, for instance x_{49}, we are in danger of setting x_{50} to `false` again. We could potentially solve the same subproblem many times. To avoid this, we record the reason why a particular assignment failed by creating a new constraint called a *nogood*. In the example above, we would record the simple nogood x_{50}. Adding this clause to our constraint set will allow us to immediately prune any subproblem with $\{x_{50} = \texttt{false}\}$. This technique was introduced by Stallman and Sussman in dependency directed backtracking (Stallman and Sussman, 1977).

On the implementation level, a nogood is created every time we backtrack. The need for the backtrack indicates that we have a variable x together with clauses indicating simultaneously that x must be true and that it must be false. By resolving these two clauses together, we get a clause that does not mention x and is falsified by the current partial assignment. This clause is both cached in the overall clausal database and used to drive further backtracks.

As an example, suppose we have successfully valued the variables $\{a = \texttt{true}, b = \texttt{false}, d = \texttt{true}, e = \texttt{false}\}$. We try to value c by setting it to `false` and discover that this violates the constraint

$$\overline{a} \vee b \vee c \vee e$$

We try to set c to `true` instead and find that this violates the constraint

$$\overline{c} \vee \overline{d}$$

Resolving these two expressions produces the nogood

$$\overline{a} \vee b \vee e \vee \overline{d}$$

which is violated by the current partial assignment. The algorithm now backtracks.

Caching the nogood ensures that the set of assignments leading to the inconsistency will be avoided as the search proceeds.

Learning and Cuts Adding Nogoods to the constraint set is analagous to the OR method of adding cuts to a branch and bound algorithm. In both cases, a new constraint that eliminates an unproductive part of the search space is generated and added to the constraint set. A cut is added to eliminate a non-integer solution to the linear relaxation, and a nogood is added to eliminate a partial solution that cannot be part of a solution. Both additions prune the search space and give direction to the search. Cuts and nogoods may be more than analogous in the case of SAT problems, because the two techniques may generate many of the same constraints, although in different representations. A good discussion of the connection between nogoods and cuts can be found in Hooker et al. (2000).

Restricted learning Learning methods speed search by eliminating redundant work. Unfortunately, the fact that a nogood is learned with every backtrack means that unrestricted learning can exhaust memory resources (which are often more limited than time resources). Unrestricted learning methods also suffer from performance degradation as the algorithm tries to manage the excessively large database of generated constraints.

To address these problems, it would be useful to have a way to ensure that the set of cached nogoods is polynomially bounded in the size of the problem. To achieve this we need a method to determine when to cache a nogood, and when to discard a nogood from the existing cache. There are currently two such methods: k-order learning and relevance-bounded learning.

In k-order learning (Dechter, 1990; Frost and Dechter, 1994), we discard all nogoods with length greater than some constant bound k. The motivation for this policy is that short clauses are generally more useful for pruning the search space than long clauses. On the face of it, a clause of length l will prune $\frac{1}{2^l}$ of the possible assignments of values to variables. Since short clauses prune more of the space, they should be retained in preference to long ones.

What this argument overlooks is that what matters is not how much each clause prunes from the overall search space, but how much it prunes from the space *that the search engine will need to examine*. For any particular subproblem, there may exist long clauses that are more useful for pruning within the subproblem than some shorter clauses.

To see this, consider the following example. We have just expanded the node with the partial assignment $\{a = \texttt{false}, b = \texttt{false}, c = \texttt{false}, d = \texttt{false}, e = \texttt{false}\}$. All other variables are unvalued. Suppose also that we have generated the following two nogoods in our search thus far:

$$a \vee b \vee c \vee d \vee e \vee f \tag{6}$$

$$\overline{a} \vee \overline{b} \vee g \tag{7}$$

The algorithm must now solve the subproblem below the node given by the current partial assignment. If at any time in this subproblem we attempt to value f, the first nogood (6) will tell us to set f to `true` and to prune the branch with $f = \texttt{false}$. If the subproblem is large, this nogood may be used many times. The second nogood (7) cannot be used to prune the subproblem's search space at all, since it is already satisfied by the current partial assignment. The longer nogood is more useful than the shorter one in solving the immediate subproblem. Even though long nogoods can play an important role in pruning, k-order learning discards them.

Relevance-bounded learning avoids this problem by keeping nogoods based on their relevance to the current position in the search space. The idea originated in dynamic backtracking (Ginsberg, 1993). That algorithm deletes a nogood when one or more variable assignment pairs are no longer a member of the current partial assignment. Bayardo and Miranker (1996) defined a generalized form of relevance in which a nogood is i-relevant if it differs from the partial assignment in at most i variables. Keeping all nogoods that are i-relevant is called *i-relevance-bounded learning*.

We can view the relevance measure of a clause as the number of variables we must unbind before the clause can be useful, in that the clause can potentially contribute to a unit propagation. If we look again at the example above, we see that under the given partial assignment, the longer nogood (6) is actually 0-relevant and can be used immediately through unit propagation, allowing us to conclude f. The shorter nogood (7) is only 2-relevant. It has no potential for pruning until we backtrack to the variables a and b and change their values.

Imagine we have set a relevance bound of $i = 3$, and we have descended into the search tree arriving at the subproblem described above. If the subproblem in our example proves to be infeasible, we will have to backtrack. Should we ever find ourselves working with the new partial assignment $\{a = \texttt{true}, b = \texttt{true}, c = \texttt{true}, d = \texttt{true}\}$, the nogood (7) will be 0-relevant and the nogood (6) will be 4-relevant. This latter nogood will therefore be dropped because it has exceeded the relevance bound of 3. Dropping clauses that exceed the relevance bound enables us to maintain polynomial space usage (Bayardo and Miranker, 1996). Experimentally, relevance-bounded learning makes better use of space resources than does k-order learning, and even when relevance-bounded learning is restricted to linear space, the performance is comparable to that of unrestricted learning. Both relevance-bounded and k-order learning are far more effective for satisfiability problems than the backtracking schemes typically used by the OR community.

Although relevance bounded learning methods have been highly successful, little is known about optimal relevance policies. Such policies are generally determined experimentally for a given problem of interest. Bounds of $i = 3$ or $i = 4$ seem to give good results.

4 AI search techniques in a pseudo-Boolean setting

The advantages of the pseudo-Boolean representation are clear: It can represent constraint sets more compactly, and thereby make more efficient use of memory resources. Changing representation is also important if we want to begin to build more efficient non-resolution based methods. In this section, we show that the standard AI techniques discussed earlier can be lifted to use pseudo-Boolean representations. We note in passing that there is already a pseudo-Boolean version of the

incomplete SAT algorithm WSAT (Walser, 1997), and pseudo-Boolean constraints have been incorporated into constraint programming languages (Bockmayr, 1993). We can adapt our intelligent backtracking algorithms to manage the more expressive pseudo-Boolean constraints as well.

Even having done so, of course, the same underlying issues remain. We must still select branch variables. We must still infer new constraints to help prune subsequent search, and we still must find ways to bound the size of the learned constraint set. We show here how to do these things using the representation of linear inequalities.

4.1 Unit propagation

Recall that a unit propagation occurs when a clause contains a single unvalued literal, and all other literals are valued false. This forces the remaining literal to be true. When we choose a variable to branch on, we'd like to select the one that maximizes the number of subsequent unit propagations, initially approximating this number by counting the number of immediate unit propagations a branching choice will cause. As noted earlier, this avoids doing full unit propagation on each free variable.

If we represent constraints as linear inequalities, determining whether we can unit propagate on a given constraint is more complex. As before, we'd like to identify constraints where the current partial assignment forces a particular value for an unvalued variable. Because we allow inequalities of the form

$$\sum a_i x_i \geq c$$

where the coefficients and the right hand side of the inequality can have values greater than 1, counting the number of satisfied literals and the number of unvalued literals no longer suffices. Instead we define two new counts: the current surplus and the possible surplus.

$$\texttt{curr} = \sum_{x_i \in V} a_i x_i - c$$

$$\texttt{poss} = \sum_{x_i \in V} a_i x_i + \sum_{x_j \notin V} b_j - c$$

V is the set of valued variables, c is the right-hand side of the inequality, and the b_j are the coefficients of the unvalued variables. When $\texttt{curr} \geq 0$ the constraint is satisfied. If the current surplus is negative then the constraint is not yet satisfied by the current partial assignment. In this case, curr tells us how much more we need to satisfy the constraint. The value of poss tells us what the surplus would be if the remaining variables were all set to 1.

The value of poss will always be greater than or equal to 0 since a negative value of poss implies that the constraint cannot be satisfied. If $\texttt{poss} = 0$, then we can clearly value all of the unvalued variables in the clause. Somewhat more generally, we can value any unvalued variable x_j if the value of its coefficient $a_j > \texttt{poss}$. As an example, consider the constraint

$$2x_1 + x_2 + 2x_3 + x_4 \geq 4$$

under the partial assignment $x_1 = 1, x_2 = 0$. The current surplus and possible surplus have values $\texttt{curr} = -2$ and $\texttt{poss} = 1$. We can thus assign a value of 1 to x_3; if $x_3 = 0$, we get $\texttt{poss} < 0$ and the clause will be unsatisfiable.

This procedure can be made more efficient by ordering the variables by decreasing weight. Now we can simply walk along the clause, stopping when we reach the first variable for which $a_j \leq \texttt{poss}$. For clausal inequalities, where all weights and the right hand side are 1, there is no increase in complexity relative to the Boolean case: if $\texttt{poss} > 0$, we see immediately that no unit propagation is possible because the weight of the first variable is no greater than poss. If $\texttt{poss} = 0$, we walk the clause, setting every unvalued variable to 1.

4.2 Nogood generation

When working with linear pseudo-Boolean inequalities, nogoods are generated in way similar to clausal nogoods. As before, a nogood is created every time we backtrack and again the need for the backtrack indicates that we have a variable x together with inequalities indicating simultaneously that $x = 1$ and that $x = 0$. Here, instead of resolving clauses, we add the two linear inequalities in a way that causes the variable x to be cancelled out of the resulting constraint.

Consider the following example. Suppose we have a partial assignment $\{b = 0, c = 0,\ d = 0,\ e = 0\}$. We attempt make the assignment $a = 0$, but this violates the constraint

$$a + d + e \geq 1$$

We try instead to make the assignment $a = 1$ and find that this violates the constraint

$$2\overline{a} + b + c \geq 2$$

We generate a nogood by taking the following linear combination of the constraints:

$$\begin{array}{r} 2(a + d + e) \geq (1)(2) \\ 2\overline{a} + b + c \geq 2 \\ \hline 2d + 2e + b + c \geq 2 \end{array}$$

The generation of the nogood ensures that the assignment which led to the conflict with variable a will be avoided in the future.

4.3 Relevance-bounded learning

Relevance bounded learning can be implemented using linear pseudo-Boolean inequalities in a way very similar to its use with clausal representation. As we saw earlier, the strength of a clausal constraint is correlated with the length of the clause, but strength can also be defined relative to a position in the search space. The number of variable assignments a linear pseudo-Boolean inequality eliminates cannot be determined solely by the length of the constraint. This makes determining the relative strengths of constraints more challenging. A constraint is clearly stronger if it subsumes another constraint, and strength can be easily inferred when all variable coefficients equal 1 by considering both the number of variables appearing in the constraint and the value of the right hand side of the inequality. Consider the inequalities:

$$a + b + c \geq 2 \tag{8}$$

$$a + b \geq 1 \tag{9}$$

$$c + d + e \geq 1 \tag{10}$$

The inequality (8) is the strongest constraint in this set, although it is not the shortest. It subsumes the constraint (9) and is stronger than (10), because it has a larger right hand side. Evaluating the relative strength of constraints when the coefficients are not restricted is not as easy. It is unclear how to determine the number of assignments removed by such a constraint other than enumerating them.

As the following example shows, it still makes sense to define the relevance of a constraint in relation to the current position in the search space. Consider the following nogoods given the partial assignment $\{a = 1, b = 1, c = 1\}$:

$$a + b + c \geq 2 \tag{11}$$

$$a + b \geq 1 \tag{12}$$

$$\overline{a} + \overline{b} + e + f \geq 1 \tag{13}$$

Constraints (11) and (12) cannot be used for pruning anywhere in the subproblem below. The constraint (13), which is the weakest constraint, can be used to prune any node in the subproblem

that contains $\{e = 0, f = 0\}$ as part of its partial assignment, making it the most useful to the immediate subproblem.

An ideal relevance policy will identify the relative usefulness of constraints for pruning the immediate search space. Our current approach is to define the "irrelevance" of a particular nogood as simply `poss`, so that the nogood is dropped when `poss` $> r$, where r is the relevance bound. This policy mimics the clausal policy discussed earlier and is convenient because the value of `poss` is readily available.

Determining optimal policies will require some experimentation as it does when using clausal representation. Here we have to contend with the additional expressiveness of the representation. It is likely that the OR community could provide helpful insight into the development of optimal policies because they have more experience with the representation.

5 Conclusion

The intelligent backtracking algorithms developed by the AI community can be easily adapted to handle the pseudo-Boolean constraints focused on by OR. We would argue that this is a necessary step toward improving the efficiency of satisfiability search engines, but also that it is only a first step. If a backtracking algorithm adapted to use pseudo-Boolean representation is given a problem instance that has been translated to clausal inequalities from a constraint set in conjunctive normal form, it will perform almost identically to its clausal counterpart, constructing cutting plane simulations of resolution proofs. The advantage that such an algorithm has over the clausal version is that it can process problem instances that contain the more expressive pseudo-Boolean constraints as well as inequalities translated from CNF. Some problems, however (such as the pigeonhole problem), can only be solved efficiently by deriving pseudo-Boolean constraints from clausal axioms.

Nevertheless, we feel that modifying existing search engines to use both pseudo-Boolean representations and sophisticated learning techniques will be a step forward for both sides. Our focus has been on modifying AI search engines to use pseudo-Boolean representation, but adapting integer programming branch-and-bound methods to use relevance bounded learning would be useful as well.

For clauses of a fixed length, the pseudo-Boolean representation used by OR is properly more expressive than the CNF representation typically used in AI. The learning techniques such as relevance-bounded learning developed by AI provide demonstrable performance improvements over the relatively simple branch-and-backtrack approach in use in OR. We have shown that there is no fundamental reason not to work with algorithms that combine the best of these techniques.

References

Baker, AB, 1995. "Intelligent backtracking on constraint satisfaction problems: experimental and theoretical results" *PhD thesis*, University of Oregon.

Bayardo, RJ and Miranker, DP, 1996. "A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem" *Proceedings of the Thirteenth National Conference on Artificial Intelligence* 298–304.

Blair, CE, Jeroslow, RG and Lowe, JK, 1986. "Some results and experiments in programming techniques for propositional logic" *Computers and Operations Research* **13**(5) 633–645.

Bockmayr, A, 1993. "Logic programming with pseudo-boolean constraints" in F Benhamou and A Colmerauer (eds.), *Constraint Logic Programming. Selected Research* MIT Press, 327–350.

Bonet, M, Pitassi, T and Raz, R, 1997. "Lower bounds for cutting plane proofs with small coefficients" *Jounal of Symbolic Logic* **62** 708–728.

Chvátal, V, 1975. "Edmonds polytopes and weakly Hamiltonian graphs" *Mathematical Programming* **5** 29–40.

Chvátal, V, 1983. *Linear Programming* WH Freeman.

Chvátal, V and Szemerédi, E, 1988. "Many hard examples for resolution" *Journal of the Association for Computing Machinery* **35** 759–768.

Cook, W, Coullard, C and Turán, G, 1987. "On the complexity of cutting plane proofs" *Journal of Discrete Applied Math* **18** 25–38.

Crawford, JM and Auton, LD, 1996. "Experimental results on the crossover point in random 3SAT" *Artificial Intelligence* **81**.

Davis, M and Putnam, H, 1960. "A computing procedure for quantification theory" *Journal of the Association for Computing Machinery* **7** 201–215.

Dechter, R, 1990. "Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition" *Artificial Intelligence* **41**(3) 273–312.

Dubois, O, Andre, P, Boufkhad, Y and Carlier, J, 1993. "SAT versus UNSAT" *Second DIMACS Challenge: Cliques, Colorings and Satisfiability* Rutgers University, NJ.

Freeman, JW, 1995. "Improvements to propositional satisfiability search algorithms" *PhD thesis*, University of Pennsylvania, PA.

Frost, D and Dechter, R, 1994. "Dead-end driven learning" *Proceedings of the Twelfth National Conference on Artificial Intelligence* 294–300.

Gallo, G and Urbani, G, 1989. "Algorithms for testing the satisfiability of propositional formulae" *Journal of Logic Programming* **7** 45–61.

Ginsberg, ML, 1993. "Dynamic backtracking" *Journal of Artificial Intelligence Research* **1** 25–46.

Goerdt, A, 1990. "Cutting plane versus frege proof systems" *Lecture Notes in Computer Science* **533**.

Gomory, R, 1963. "An algorithm for integer solutions to linear programs" *Recent Advances in Mathematical Programming* McGraw-Hill, New York, 269–302.

Haken, A, 1985. "The intractability of resolution" *Theoretical Computer Science* **39** 297–308.

Hammer, P and Rudeanu, S, 1968. *Boolean Methods in Operations Research and Related Areas* Springer-Verlag, Berlin.

Harche, F, Hooker, JN and Thompson, GL, 1994. "A computational study of satisfiability algorithms for propositional logic" *ORSA Journal on Computing* **6**(4) 423–435.

Harche, F and Thompson, GL, 1994. "The column subtraction algorithm: an exact method for solving weighted set covering, packing and partitioning problems" *Computers and Operations Research* **21**(6) 689–705.

Hooker, J, Ottosson, G, Thorsteinsson, ES and Kim, H-J, 2000. "A scheme for unifying optimization and constraint satisfaction methods" *The Knowledge Engineering Review*, this issue.

Hooker, JN and Fedjki, C, 1990. "Branch-and-cut solution of inference problems in propositional logic" *Annals of Mathematics and Artificial Intelligence* **1** 123–139.

Hooker, JN and Vinay, V, 1995. "Branching rules for satisfiability" *Journal of Automated Reasoning* **15** 359–383.

Jeroslow, R and Wang, J, 1990. "Solving the propositional satisfiability problem" *Annals of Mathematics and Artificial Intelligence* **1** 167–187.

Li, CM and Anbulagan, 1997. "Heuristics based on unit propagation for satisfiability problems" *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*.

Loveland, DW, 1978. *Automated Theorem Proving: A Logical Basis* North Holland.

Mitchell, DG, 1998. "Hard problems for CSP algorithms" *Proceedings of the Fifteenth National Conference on Artificial Intelligence* 398–405.

Nemhauser, GL and Wolsey, LA, 1988. *Integer and Combinatorial Optimization* Wiley, New York.

Pretolani, D, 1993. "Satisfiability and hypergraphs" *PhD thesis*, Universita di Pisa, Italy.

Pudlák, P, 1997. "Lower bounds for resolution and cutting plane proofs and monotone computations" *Journal of Symbolic Logic* **62** 981–998.

Stallman, RM and Sussman, GJ, 1977. "Forward reasoning and dependency directed backtracking in a system for computer aided circuit analysis" *Artificial Intelligence* **9**(2) 135–196.

Walser, JP, 1997. "Solving linear pseudo-boolean constraint problems with local search" *Proceedings of the Fourteenth National Conference on Artificial Intelligence*.

Wolfman, SA and Weld, DS, 2000. "Combining linear programming and satisfiability solving for resource planning" *The Knowledge Engineering Review* this issue.