韓國經營科學會誌
第27卷 第4號
2002年 12月

집합 피복 공식화를 이용한 명제논리의 만족도 문제에 대한 계산실험 연구*

조 건**

An Empirical Study for Satisfiability Problems in Propositional Logic Using Set Covering Formulation*

Geon Cho**

■ Abstract ■

A satisfiability problem in propositional logic is the problem of checking for the existence of a set of truth values of atomic propositions that renders an input propositional formula true. This paper describes an empirical investigation of a particular integer programming approach, using the set covering model, to solve satisfiability problems. Our satisfiability engine, SETSAT, is a fully integrated, linear programming based, branch and bound method using various symbolic routines for the reduction of the logic formulas. SETSAT has been implemented in the integer programming shell MINTO which, in turn, uses the CPLEX linear programming system. The logic processing routines were written in C and integrated into the MINTO functions. The experiments were conducted on a benchmark set of satisfiability problems that were compiled at the University of Ulm in Germany. The computational results indicate that our approach is competitive with the state of the art.

Keyword : Propositional Logic, Satisfiability Problem, Set Covering, Branch and Bound

논문접수일 : 2001년 7월 3일　　논문게재확정일 : 2002년 10월 10일

* This study was financially supported by Chonnam National University in 1999.

** School of Business Administration at Chonnam National University, Standing Researcher of Management Research Institute

1. Introduction

Satisfiability problems in propositional logic are well-known as the first NP-complete problems. They have been solved by a number of methods over the years. One of the best known complete methods is Robinson's resolution algorithm, which is designed for first-order predicate logic. Resolution applied to propositional logic is called ground resolution and is essentially a portion of the Quine-McClusky algorithm [10]. The difficulty with resolution-based methods is that their running time tends to explode with problem size. The Quine-McClusky algorithm and subsequently the resolution algorithm were recently shown to have exponential complexity [10]. Another symbolic approach to solving satisfiability has been the search method of Davis-Putnam-Loveland (DPL) which is a more practical method than resolution but also exponential in the worst-case [5].

Recently, more effective methods for solving satisfiability problems have been developed. These methods are based on the fact that the satisfiability problem can be written as an integer program that can be solved with methods that exploit its structure. The two best-known methods for solving integer programs, branch and bound and cutting plane methods, have been applied to the integer program underlying a satisfiability problem. The branch and bound technique was used by Blair, Jeroslow and Lowe [4], who pioneered the mathematical programming approach to inference in propositional logic. They showed that a straightforward branch and bound algorithm can solve a large class of random problems by enumerating only a few nodes in the search tree[12]. The branch and bound method is similar to the DPL method except that one solves a linear programming relaxation of the problem at each node of the search tree, rather than applying unit resolution.

Hooker[9] showed that input resolvents and rank-one cutting plane techniques can solve a class of random satisfiability problems more rapidly than the well-known resolution method of theorem proving in propositional logic. Hooker and Fedjki[12] combined the branch and bound and cutting plane approaches to obtain a branch and cut algorithm and found that the branch and cut methods required substantially less time on random problems than branch and bound when the latter must enumerate a relatively large number of nodes.

Despite this success, resolvents are rather "weak" cuts in the sense that many of these resolvents are redundant and ineffective. In fact, the use of cutting planes that are "deep" or facet-inducing may reduce the size of the enumeration trees remarkably. We need to have an understanding of the facial description of the satisfiability-polytope. But no useful results of this sort have been obtained for the satisfiability-polytope partly because the dimension of the face of this polytope is not well defined [1].

Araque and Chandru [1] formulated the satisfiability problem as a set covering problem, a type of integer program for which many theories have already been developed. In that formulation they use as many decision variables as there are literals in the proposition (twice the number of atomic propositions). Further, the set covering formulation has a full-dimensional convex hull and this gives a technical handle for exploring the polyhedral combinatorics of satisfiability [1].

In this paper we implement a branch-and-bound algorithm with the Jeroslow-Wang branching rule [13] based on the set covering formulation. We utilize an integer programming shell, MINTO(Mixed INTeger Optimizer) written in C language. MINTO is a software system that solves mixed-integer linear programs by a branch and bound algorithm with linear programming relaxations [16]. MINTO is implemented on top of the CPLEX callable library, version 1.2. We find that SETSAT's results in the number of nodes are more efficient than MINTO's defaults. For most problems, SETSAT results in search trees with fewer nodes than DPL(J/W) as implemented by Hooker.

In the next section, we begin with a brief review of some of the basic notions of propositional logic. We also introduce the Davis-Putnam-Loveland (DPL) algorithm and discuss the branch and bound method for the satisfiability problem. We then introduce the set covering formulation of the satisfiability problem. In section 3, we describe our theorem prover (called SETSAT) and present the results of our experiments. Section 4 contains a summary of our conclusions and a list of topics for future research.

2. Preliminaries

We begin with a brief review of some of the basic notions of propositional logic and go on to discuss Resolution and DPL, the two classical symbolic methods for theorem proving. In the second subsection we will review the quantitative or numerical approach based on integer programming formulations. The last subsection will deal with the set covering formulation.

2.1 Symbolic Methods for Satisfiability Problem

Symbolic logic has been studied from both philosophical and mathematical perspectives. J.A. Robinson developed a single inference rule, the resolution principle, which was shown to be complete and easily implemented on computers. Since then, many improvements of the resolution principle have been made. In this subsection, we are interested in the applications of symbolic logic to solving large-scale inference in propositional logic.

A *proposition* is a declarative sentence that is either true or false, but not both. The "true" or "false" assigned to a proposition is called the *truth value* of the proposition. In propositional logic, it is customary to use five logical connectives ;

$$\neg\,(\text{not}),\ \wedge\,(\text{and}),\ \vee\,(\text{or}),\ \rightarrow(\text{if}\cdots\text{then}),$$
$$\text{and}\ \leftrightarrow(\text{if and only if}).$$

These five logical connectives can be used to build compound propositions from simple(atomic) propositions. For instance, suppose that we represent "x is larger than y" by P, "y is larger than z" by Q, and "x is larger than z" by R. Then the sentence "if x is larger than y and y is larger than z, then x is larger than z" may be represented by $((P \wedge Q) \rightarrow R)$. A *literal* is an atomic proposition x_j or its negation $\neg x_j$. A *clause*, such as $x_1 \vee \neg x_2 \vee x_3$, is a disjunction (or) of zero or more literals. A clause with no literals is the *empty* clause, which is always interpreted as taking the value false. A *truth assignment* is an assignment of truth values to every variable. A proposition P is a *conjunctive normal form* (c.n.f.) if it is a conjunction $P = P_1 \wedge P_2 \wedge \cdots \wedge P_m$ of propositions P_i $(1 \leq i \leq m)$, each of which is a (disjunctive) clause.

A set S of clauses is *satisfiable* (or consistent) if there exists a truth assignment which makes every clause in S true. A set S of clauses is *unsatisfiable* (or inconsistent) if it is false under all its truth assignments. It is not difficult to see that the satisfiability question is central to inference in propositional logic. It is also well known that any proposition can be reduced to an equivalent c.n.f. formula by using the De Morgan's Laws along with the distributive laws for disjunction and conjunction. We will therefore use the satisfiability problem of a formula in c.n.f. as being synonymous with "theorem proving" in this paper.

S *logically* implies a clause C if every truth assignment that makes all the clauses in S true also makes C true. A clause C *absorbs* (or *dominates*) a clause D if every literal of C occurs in D. It is not difficult to see that a clause C logically implies a clause D if and only if C absorbs D. Note that a set of clauses that implies the empty clause is unsatisfiable. When two clauses C and D are such that exactly one variable x_j occurs negated in one clause and posited in the other, their resolvent is the clause containing all the literals in C and D except x_j and $\neg x_j$. For instance, the resolvent of (1) and (2) below is (3).

$$x_1 \vee x_2 \vee \neg x_3 \qquad (1)$$

$$\neg x_1 \vee x_2 \qquad \vee x_4 \qquad (2)$$

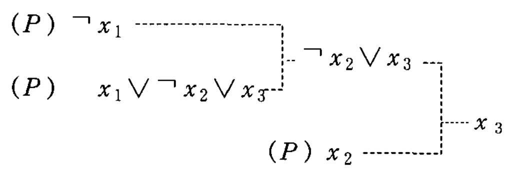
$$x_2 \vee \neg x_3 \vee x_4 \qquad (3)$$

The resolvent is implied by the conjunction of its two parents but by neither individually. In fact, resolution is reasoning by cases. In the above example, if x_1 is false, then x_2 or $\neg x_3$ must be true by (1), whereas if x_1 is true, then x_2 or x_4 is true by (2) ; it follows that x_2, $\neg x_3$, or x_4 must be true.

A *resolution proof* of a clause C from a set of premises is a finite sequence of clauses beginning with the premises and ending with a clause that absorbs C, such that every clause is the resolvent of two earlier clauses in the sequence. A *refutation* of a set of premises is a proof of the empty clause from these premises. W.V. Quine showed that resolution is a complete inference method for propositional logic, in the following sense ; if a given set S of clauses is unsatisfiable, then resolution will generate the empty clause ; if it is satisfiable, any conclusion that follows from S is absorbed by one of the clauses obtained in the resolution process. Resolution is therefore a complete refutation method, since there is a refutation for any unsatisfiable set of premises.

A *prime implication* of a set of clause S is a clause if it is not absorbed by any clause that is an implication of S. In fact, when S is satisfiable, the clauses that remain when the procedure terminates are the prime implications of S. Note that prime implications of S are in a sense the strongest possible implications of S, and every implication of S is absorbed by at least one prime implication. There is a simple finite procedure for generating all prime implications of a given set of clauses. If there is exactly one literal x_j that occurs negated in clause C and posited in clause D, then C and D are the parents of a resolvent (on x_j). By repeatedly applying this resolution step and throwing away absorbed clauses we can eventually find all prime implications of a set of clauses. Unfortunately the number of prime implications grows exponentially with the number of variables in the worst case.

A *unit resolution* is a resolution in which a resolvent is obtained by using at least one unit parent clause. A *unit proof* is a proof that consists entirely of unit resolutions. For instance, there is a unit proof of x_3 from the premises marked (P).



A *unit refutation* is a unit proof that is a refutation. Unit resolution is complete for the satisfiability problem on Horn Formulas[5].

As for general satisfiability problems, some subclasses of them seem clearly hard. There seems to be no algorithm, for instance, that can easily solve the famous pigeon-hole problems (one of the results of our computational experiments notes a welcome exception to this statement). The pigeon hole problem is to place n pigeons in $n-1$ holes so that no hole contains more than one pigeon. For instance, the pigeon hole problem for $n=3$ is,

$$\begin{array}{llllll}
x_{11} & & & \vee\, x_{12} & & \\
 & x_{21} & & & \vee\, x_{22} & \\
 & & x_{31} & & & \vee\, x_{32} \\
\neg x_{11} \vee \neg x_{21} & & & & & \\
\neg x_{11} & & \vee \neg x_{31} & & & \\
 & \neg x_{21} \vee \neg x_{31} & & & & \\
 & & & \neg x_{12} \vee \neg x_{22} & & \\
 & & & \neg x_{12} & & \vee \neg x_{32} \\
 & & & & \neg x_{22} \vee \neg x_{32} &
\end{array} \qquad (4)$$

Here x_{ij} is true when pigeon i placed in hole j. Thus the first three clauses in (4) assert that each pigeon is placed in a hole. The remaining clauses assert, for pair of pigeons, that both do not occupy the same hole. In general, the pigeon hole problem for n is,

$$\bigvee_{j=1}^{n-1} x_{ij}, \quad \text{for all } i \in \{1, 2, \cdots, n\}$$

$$\neg x_{ik} \vee \neg x_{jk}, \quad \text{for all } k \in \{1, 2, \cdots, n-1\},$$
$$i, j \in \{1, 2, \cdots, n\}, \quad i \neq j.$$

In spite of pathologically hard problems like the pigeon hole problems, computational experience over the last few years indicates that a wide variety of satisfiability problems can be practically solved in large instances. There are several methods that have successfully solved satisfiability problems. They include tree search methods and cutting plane methods related to the resolution method of theorem proving. In this paper we will be mostly concerned with tree search methods.

The Davis-Putnam-Loveland (DPL) procedure is a classic inference method well-known to computer science and artificial intelligence. DPL is very closely related to the branch and bound method for satisfiability problem. The DPL procedure generates a binary search tree in order to find a satisfying solution for a set of clauses. Some terminology is helpful here.

A *binary tree* consists of a set of nodes, each of which has at most two other nodes as immediate successors. Every node except the root node is the immediate successor of exactly one node, called its immediate predecessor. Node A is a *predecessor* of node B, and B a *successor* of A, if A is the first and B the last in a series of nodes, each of which is the immediate predecessor of the next. To generate the DPL search tree, we associate the original problem with root node of the tree and apply unit resolution procedure to simplify the problem. If we don't prove unsatisfiability, we set a chosen variable to true and to false, generating two immediate successors.

DPL optionally uses *monotone variable fixing*, which simply looks for a monotone variable (a variable that appears only posited or only negated) and assumes that the variable is false if the variable is negated and the variable is true if the variable is posited. In either case we remove all clauses containing the variable. Clearly this does not affect the satisfiability of the original problem.

The performance of DPL can be improved by using an efficient branching rule. One heuristic for choosing on which variable to branch is that proposed by Jeroslow and Wang (we will call it the *Jeroslow-Wang branching rule*). The Jeroslow-Wang branching rule says roughly that one should branch on a variable that occurs in a large number of short clauses. If v represents a truth value (0 or 1), define the function

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

where N_{jkv} is the nㅅumber of k-literal clauses in S' in which x_j occurs positively (if $v = 1$) or negatively(if $v = 0$). If (j^*, v^*) maximizes $w(S', j, v)$, then we branch on x_j^*, taking the $S' \cup \{x_j^*\}$ branch first if $v^* = 1$, and otherwise taking the $S' \cup \{\neg x_j^*\}$ branch first.

Jeroslow and Wang [13] found that the speed of the branch and bound procedure can apparently be increased on random satisfiability problems by branching using the rule above and by replacing the LP step with unit resolution. Their approach differs from DPL only on the choice of which node to expand next (DPL is generally depth-first). In fact, it turns out that the two methods generate search trees of identical size when the problem is unsatisfiable. However, the Jeroslow-Wang method may find a solution sooner than DPL if the problem is satisfiable. We now introduce branch and bound method for integer programming and discuss the solution of satisfiability problem and branch and bound technique.

2.2 Traditional Integer Programming Approaches for Satisfiability Problem

Satisfiability problems have been traditionally solved by the nonnumeric methods discussed above. But recently several researchers have noted that mathematical programming methods provide more efficient approaches for solving hard inference problems. These methods are based on the fact that satisfiability problem can be written as an integer program that can be solved with methods that exploit its structure.

The two best-known methods for solving integer programs, branch and bound and cutting plane methods, have been applied to inference in propositional logic. The branch and bound technique was used by Blair, Jeroslow and Lowe

[4], who pioneered the mathematical programming approach to inference in propositional logic. Their motivating idea is that by solving the linear programming (LP) relaxation of the problem at each node of the branch and bound tree, they may find an "incumbent" integer solution early in the process and thereby solve the problem quickly. The branch and bound approach creates a binary search tree in a way very similar to the DPL algorithm. But rather than using unit resolution at each node, it solves the linear relaxation of the problem. We now introduce how to formulate inference problems in propositional logic as integer programs and discuss how the integer programs are solved.

Any clause can be written as a linear inequality in binary variables. For instance, $x_1 \vee \neg x_2 \vee x_3$ can be written,

$$x_1 + (1 - x_2) + x_3 \geq 1 \quad \text{or} \quad x_1 - x_2 + x_3 \geq 0$$

where each $x_j \in \{0, 1\}$. We interpret $x_j = 1$ to mean x_j is true, and $x_j = 0$ to mean is false. Let a set S of m clauses in n variables be written as an $m \times n$ system $Ax \geq a$ of linear inequalities. Clearly, S is satisfiable if and only if the minimum value of the objective function is zero in the following integer programming formulation.

$$\begin{aligned} &\min \ x_0 \\ &\text{s.t.} \ x_0 e + Ax \geq a \qquad (5) \\ &\qquad x_j \in \{0, 1\}, \ j = 1, 2, \cdots, n \end{aligned}$$

where e is a column vector of ones.

Ordinarily a branch and bound algorithm bounds as well as branches. That is, each time we obtain an incumbent solution we get a new upper bound on the minimum value of the objective function value. So, if at some node the LP relaxation yields an objective function value that is greater than or equal to the best upper bound obtained so far, then we can fathom that node, since the objective function value at any successor of that node cannot be better. The search continues until we have branched at or fathomed every node in the current search tree, and the best incumbent solutions solve the original integer program. However, this sort of bounding serves no purpose in a satisfiability problem. Even if we find an integer solution with objective function value 1, this is a useless bound, because the LP relaxation cannot have a greater solution value than 1.

As we mentioned earlier, the branch and bound method for satisfiability problem is quite similar to DPL in the sense that solving the LP relaxation at a node is equivalent to applying unit resolution, since the LP relaxation and unit resolution detect unsatisfiability in the same instances. In fact, x_0 is strictly positive in the LP relaxation of (5) if and only if unit resolution detects unsatisfiability.

It is advantageous to apply unit resolution before solving the LP relaxation. If a contradiction is detected, we can backtrack without solving the relaxation. Otherwise we obtain a simplified problem, and we solve its LP relaxation in hope of obtaining an integer solution with $x_0 = 0$. The advantage of this approach is not only that we solve fewer LP problems, but also that the problems become smaller as we move deeper into the search tree. The disadvantage is that we must solve each LP relaxation from scratch.

It may happen that once unit resolution has been performed without finding a contradiction, the LP relaxation does not necessarily give any information about the satisfiability of the problem. In fact, if every clause has at least two literals other than x_0, then it can always be solved by setting $x_0 = 0$ and every other $x_j = 1/2$, even if the problem is unsatisfiable. But even though this solution provides no information, it does not mean that there is no point in solving the linear relaxation. For linear programming algorithms typically find extreme point solutions, and the solution with all $x_j = 1/2$ is seldom an extreme point solution. An extreme point solution can be quite useful, because it may be an integral solution with $x_0 = 0$.

2.3 Set Covering Formulation of Satisfiability Problem

We just saw that checking satisfiability in propositional logic can be stated as checking feasibility of an integer linear program. Polyhedral combinatorics, a central theme in integer programming, is concerned with the structure of the convex hull of incidence vectors that represent feasible solutions to an integer linear program. Previous attempts at the polyhedral combinatorics of the "satisfiability polytope" have run into the difficulty that the dimension of this polytope is not well defined (The convex hull of incidence vectors of truth assignments is called the satisfiability polytope). Araque and Chandru [1] constructed a set covering formulation of satisfiability problem such that the satisfiability polytope is the projection of a face (possibly empty) of the dominant of the set covering polytope.

The set covering problem can be stated as

$$(SC) \quad \min\{cx \mid Ax \geq \mathbf{1}, x \in \{0,1\}^n\},$$

where $A = (a_{ij})$ is an $m \times n$ matrix with $a_{ij} \in \{0,1\}$, for all i, j, and $\mathbf{1}$ is the m-vector of 1's. If $Ax \geq \mathbf{1}$ is replaced by $Ax = \mathbf{1}$, the problem is called set partitioning. If we reverse the inequality in the definition of (SC), we obtain the set packing problem

$$(SP) \quad \max\{cx \mid Ax \leq \mathbf{1}, x \in \{0,1\}^n\}.$$

Quite surprisingly, unlike the closely related set packing problem, the study of the facial structure of the set covering polytope $Q(A)$, i.e., the linear description of the convex hull of the feasible solutions of (SC), has received very little attention in the literature [11]. Recently, motivated by the successful applications of the polyhedral approach to some, NP-hard, combinatorial problems, several authors [2, 3, 6, 7, 14] addressed the problem of providing a partial linear description of the polytope $Q(A)$.

Now we introduce the set covering formulation of satisfiability problem as presented in Araque and Chandru [1]. In this formulation we use as many decision variables as there are literals in the proposition (twice the number of atomic propositions). These additional freedom permits at least two advantages. We are able to formulate satisfiablity problem as a set covering problem, a type of integer program for which specialized theory has already been developed. Further, the set covering formulation has a full-dimensional convex hull and this gives us a technical handle for exploring the polyhedral combinatorics of satisfiability problem.

Given a set of clauses $C = \{C_1, \cdots, C_m\}$ on the atomic propositions $\{x_1, \cdots, x_n\}$, the satisfiability problem is to determine if there exists a set of truth assignments (a model) that satisfies all clauses in C. We define a set of $2n$ variables $\{y_1, z_1, \cdots, y_n, z_n\}$ which have the interpretation

$$y_j = \begin{cases} 1 & \text{if } x_j \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

$$z_j = \begin{cases} 1 & \text{if } x_j \text{ is false} \\ 0 & \text{otherwise} \end{cases}$$

Hence there is an 1-1 correspondence between a set of truth assignments for $\{x_1, \cdots, x_n\}$ and non-negative integer y, z meeting the condition

$$y_j + z_j = 1 \quad \text{for } j = 1, 2, \cdots, n \qquad (6)$$

Given a clause C_i of C that is not a tautology, i.e. is nonempty and does not contain a literal and its negation, we can write down a clausal inequality

$$(a^i, b^i)(y, z) \geq 1, \quad \text{where}$$

$$a^i_j = \begin{cases} 1 & \text{if } x_j \text{ is in } C_i \\ 0 & \text{otherwise} \end{cases}$$

$$b^i_j = \begin{cases} 1 & \text{if } \neg x_j \text{ is in } C_i \\ 0 & \text{otherwise} \end{cases}$$

Satisfiability of C can be resolved by solving the following integer programming problem (IP).

$$\begin{array}{lll} & \min & \sum_{j=1}^{n} (y_j + z_j) \\ (IP) & \text{s.t.} & (a^i, b^i)(y, z) \geq 1 \quad \text{for } C_i \in C \\ & & y_j + z_j \geq 1 \quad \text{for } j = 1, 2, \cdots, n \\ & & (y, z) \in \{0, 1\}^{2n} \end{array}$$

Clearly C is satisfiable if and only if the optimal objective value of (IP) is n. Let $F(y, z)$ denote the set of feasible solutions to (IP) and let $Q_C = \text{conv}\{F(y, z)\}$ be the convex hull of $F(y, z)$. Then Q_C is nonempty even if C is unsatisfiable since the vector of 1's $(\mathbf{1}, \mathbf{1})$ is always in $F(y, z)$. The main result proved in [1] is that prime implications of C define facets of Q_C. This result on the set covering polytope related to satisfiability problems illustrate another nice connection between logic and the geometry of optimization : prime implications which are the "strongest implications" give rise to facets which are the "strongest inequalities".

These results when combined with the body of knowledge on facets of general set covering polytopes [2, 3, 7, 15, 17] gives us some hope that effective cutting plane methods for satisfiability are possible. It was with this motivation that we set about building a satisfiability checker based on set covering. This paper reports on the first phase of this project in which we have implemented SETSAT, a theorem prover for propositional logic using linear programming and symbolic techniques on this set covering formulation. The details of this implementation as well as the results of our empirical study are the topics of the next section.

3. Experiments with the Set Covering Formulation

3.1 The Inference Engine

In this section we describe the design of our "satisfiability engine" SETSAT. SETSAT is written in C, for compatibility with the integer programming shell, MINTO(Mixed INTeger Optimizer) developed by Savelsbergh, Sigismondi and Nemhauser in Georgia Institute of Technology and also written in C. MINTO is a software system that solves mixed-integer linear programs by a branch and bound algorithm with linear programming relaxations. MINTO works with a minimization problem. However, if the original formulation describes a maximization problem, MINTO will change the signs of all the objective function coefficients by utilizing a command line option. MINTO is implemented on top of the CPLEX callable library, version 1.2. MINTO requires the mixed integer programming formulation to be specified in a MPS format file.

To be as effective and efficient as possible when used as a general purpose mixed integer optimizer, MINTO attempts to improve the formulation by preprocessing, construct feasible solutions, generate strong valid inequalities, perform variable fixing based on reduced prices, and control the size of the linear programs by managing active constraints. A set of application functions has to be compiled and linked with the MINTO library in order to produce an executable version of MINTO. These functions give the application program the opportunity to incorporate problem specific knowledge and thereby increase the overall performance [16].

We have used MINTO in order to implement SETSAT, a branch and bound algorithm with Jeroslow-Wang's branching rule applied to the set covering formulation of the satisfiability problem. SETSAT was created by including four subroutines as application programs in MINTO. These were needed to exploit the possibility of problem reductions based on logic processing. The four subroutines correspond to

- BIPARTITE : convert c.n.f. formula to set covering
- UNITRES : unit resolution
- DOMINATION : domination (absorption)
- MVF : monotone variable fixing

3.1.1 Subroutine BIPARTITE

In this subroutine we translate satisfiability of a propositional formula in c.n.f. to a set covering problem represented by a bipartite graph structure. Every clausal inequality has its own structure called CONSTRAINT which store information about the clause, such as the number of variables and the indices of the variables (positive for y-variables, negative for z-variables). The CONSTRAINTs are maintained in doubly-linked lists. Each variable in each structure CONSTRAINT has also its own structure VARIABLE which is a singly-linked list. Every variable also has its own structure called MONOVAR which stores pointers which point to clauses containing the variable. Every pointer in the structure MONOVAR has also its own structure CLAUSE which is a singly-linked list. For example, consider the following set of clauses.

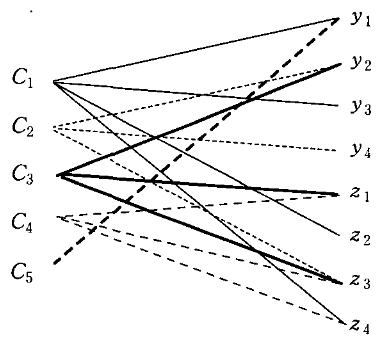
$$\begin{array}{lllll}
(C_1) & x_1 \vee \neg x_2 \vee & x_3 \vee \neg x_4 & & \\
(C_2) & & x_2 \vee \neg x_3 \vee & x_4 & \\
(C_3) & \neg x_1 \vee & x_2 \vee \neg x_3 & & \qquad (7)\\
(C_4) & \neg x_1 & \vee \neg x_3 \vee \neg x_4 & & \\
(C_5) & x_1 & & &
\end{array}$$

The set covering formulation of the above problem (7) is

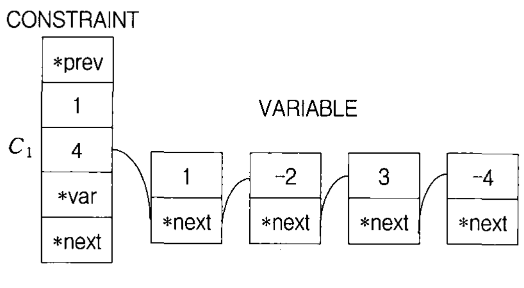
$$\min \sum_{j=1}^{4} (y_j + z_j)$$

$$
\begin{aligned}
\text{s.t. } & y_1 + z_2 + y_3 + z_4 \geq 1 \\
& y_2 + z_3 + y_4 \geq 1 \\
& z_1 + y_2 + z_3 \geq 1 \\
& z_1 + z_3 + z_4 \geq 1 \\
& y_1 \geq 1 \\
& y_j + z_j \geq 1 \quad \text{for } j = 1,2,3,4 \\
& y_j,\ z_j \in \{0,1\} \quad \text{for } j = 1,2,3,4
\end{aligned}
\tag{8}
$$

Then we translate this formula into a bipartite graph structure as shown in [Figure 1]. Each C_i's, y_i's and z_i's have their own structures. For example, C_1 and y_1 have the following structures as shown in [Figure 2] and [Figure 3], respectively.



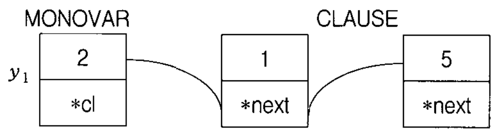
[Figure 1] Original Bipartite Graph Structure



[Figure 2] Structure CONSTRAINT and VARIABLE

Here,

* prev = a pointer which points to the previous structure CONSTRAINT in the doubly linked list
* var = a pointer which points to the structure VARIABLE
* next = a pointer which points to the next structure CONSTRAINT in the doubly linked list or to the next structure VARIABLE in the singly linked list



[Figure 3] Structure MONOVAR and CLAUSE

Here,

* cl = a pointer which points to the structure CLAUSE
* next = a pointer which points to the next structure CLAUSE in the singly linked list

3.1.2 Subroutine UNITRES

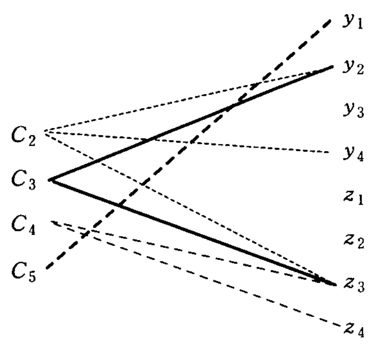
This subroutine implements the unit resolution method on the BIPARTITE representation. We search the list of CONSTRAINTs for a unit clause. As long as there is a unit clause, we assign the corresponding variable the truth value needed to make this clause true and delete all clauses containing the variable and also delete the opposite variable from all clauses. If we create a new unit clause, then we add the unit clause at the tail so that we can do unit resolution for the new atomic variable later. If two unit clauses are opposite, then the set of clauses is unsatisfiable. If no clauses remain except unit clauses, a satisfying truth assignment has been found. For example, consider the set of clauses (7). Since there is a unit clause C_5 containing an atomic variable y_1, we delete clauses C_1 containing y_1. We need to update the structure MONOVAR before we delete C_1 because if we delete C_1 then it will not be in the MONOVAR any more. Now we delete z_1 from C_3 and C_4. Thus we have the following set of clauses (9).

$$\begin{array}{ll} (C_2) & x_2 \vee \neg x_3 \vee x_4 \\ (C_3) & x_2 \vee \neg x_3 \\ (C_4) & \neg x_3 \vee \neg x_4 \\ (C_5) & x_1 \end{array} \tag{9}$$

The set covering formulation of (9) is

$$\begin{aligned} \min \ & \sum_{j=1}^{4} (y_j + z_j) \\ \text{s.t.} \ & y_2 + z_3 + y_4 \geq 1 \\ & y_2 + z_3 \geq 1 \\ & z_3 + z_4 \geq 1 \\ & y_1 \geq 1 \\ & y_j + z_j \geq 1 \quad \text{for } j = 1,2,3,4 \\ & y_j,\ z_j \in \{0,1\} \quad \text{for } j = 1,2,3,4 \end{aligned} \tag{10}$$

And the corresponding bipartite graph structure is shown in [Figure 4].



[Figure 4] Bipartite Graph Structure after UNITRES

3.1.3 Subroutine DOMINATION

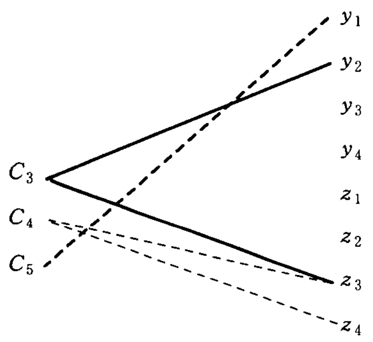
This subroutine checks for dominated (absorbed) clauses and deletes them. We first check the number of clauses containing a variable, say y_i. If the number is greater than one, then we check the number of variables contained in each clause containing y_i. If a clause C_j (containing y_i) has more variables than a clause C_k (containing y_i), then we check if C_j contains all variables in C_k. If it does, then C_j is dominated by C_k, so we delete C_j. Before we delete C_j, we need to update the structure MONOVAR. In the above example, since C_2 is dominated by C_3, we delete C_2 after we delete C_2 from the structure MONOVAR. Therefore the resulting set of clauses is

$$\begin{array}{ll} (C_3) & x_2 \vee \neg x_3 \\ (C_4) & \neg x_3 \vee \neg x_4 \\ (C_5) & x_1 \end{array} \tag{11}$$

And the set covering formulation of (11) is

$$\begin{aligned} \min \ & \sum_{j=1}^{4} (y_j + z_j) \\ \text{s.t.} \ & y_2 + z_3 \geq 1 \\ & z_3 + z_4 \geq 1 \\ & y_1 \geq 1 \\ & y_j + z_j \geq 1 \quad \text{for } j = 1,2,3,4 \\ & y_j,\ z_j \in \{0,1\} \quad \text{for } j = 1,2,3,4 \end{aligned} \tag{12}$$

The corresponding bipartite graph structure is shown in [Figure 5].



[Figure 5] Bipartite Graph Structure after DOMINATION

3.1.4 Subroutine MVF

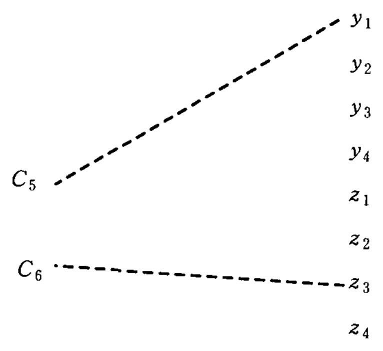
This subroutine implements monotone variable fixing, which the reader may recall is the idea of fixing an atomic proposition to true (false) if it only appears as a positive (negative) literal in all clauses. We check the structure MONOVAR to see if there is a monotone variable. We first make a list of monotone variables so that we can apply MVF to new monotone variables efficiently. Whenever we detect a monotone variable we add it to the list. We use an array for the list of monotone variables. Now if y_i is an monotone variable, then after updating the structure MONOVAR, we delete every clause containing y_i by setting y_i to true and add a unit clause with an atomic variable y_i at the tail of the linked list of CONSTRAINT so that we can keep track of the variable fixed by true. If no clauses remain except those monotone variables, a satisfying truth assignment has been found. With the continuation of the above example, since y_2, z_3, and z_4 are monotone variables, we delete C_3 and C_4 after updating MONOVAR and add one of these monotone variables (we use z_3) to the tail of the linked list of CONSTRAINT. Therefore the resulting set of clauses is

$$(C_5) \quad x_1$$
$$(C_6) \quad \neg x_3$$

And the set covering formulation of this is as follows.

$$\begin{aligned} \min \quad & \sum_{j=1}^{4}(y_j + z_j) \\ \text{s.t.} \quad & y_1 \geq 1 \\ & z_3 \geq 1 \\ & y_j + z_j \geq 1 \quad \text{for } j = 1,2,3,4 \\ & y_j,\ z_j \in \{0,1\} \quad \text{for } j = 1,2,3,4 \end{aligned}$$

The corresponding bipartite graph structure is shown in [Figure 6].



[Figure 6] Bipartite Graph Structure after MVP

Notice that we have solved example (7) by just applying these logic processing subroutines. In fact, if we set x_1 to true and x_3 to false, the problem is satisfiable. This is equivalent to the fact that the set covering formulation of (7) has an integer solution $(y_1, y_2, y_3, y_4, z_1, z_2, z_3, z_4) = (1,1,0,1,0,0,1,0)$ with optimum objective value 4 (note that there are several integer solutions with objective value 4).

Now we observe that there is a natural order for these subroutines to be invoked.

UNITRES $\Rightarrow$ DOMINATION $\Rightarrow$ MVF

- **Remark (i)** : Monotone variable fixing (MVF) does not enable unit resolution (UNITRES). Suppose y_j is a monotone variable. Then we delete every clause containing y_j and add a unit clause y_j at the tail of the structure CONSTRAINT by MVF. Therefore, we can only create some unit clauses containing monotone variables Thus, UNITRES is not enabled by MVF.
- **Remark (ii)** : Domination (DOMINATION) does not enable unit resolution (UNITRES). Suppose a clause C_i is dominated by a clause C_j. Then C_i is deleted by DOMINATION.

Since we delete a clause, not a variable, the result of DOMINATION can not create unit clauses. Therefore DOMINATION does not enable a new unit clause.

- **Remark (iii)** : Monotone variable fixing (MVF) does not enable domination (DOMINATION). Suppose y_j is a monotone variable. Then we delete every clause containing y_j and add a unit clause y_j at the tail of the structure CONSTRAINT by MVF. Since no clauses are dominated by any added unit clauses after MVF, DOMINATION is not enabled by MVF.
- **Remark (iv)** : UNITRES may enable MVF. For example, consider the following set of clauses.

$$\begin{array}{lllll}
(C_1) & x_1 \vee & x_2 \vee & x_3 & \\
(C_2) & \neg x_1 \vee & \neg x_2 & & \vee\ x_4 \\
(C_3) & & & \neg x_3 & \vee \neg x_4 \\
(C_4) & x_1 & & &
\end{array}$$

If we apply UNITRES, then we have

$$\begin{array}{llll}
(C_2') & \neg x_2 & & \vee\ x_4 \\
(C_3) & & \neg x_3 & \vee \neg x_4 \\
(C_4) & x_1 & &
\end{array}$$

Note that $\neg x_2$ and $\neg x_3$ are new monotone variables. Therefore, MVF is now applicable, which means UNITRES has enabled MVF.

- **Remark (v)** : UNITRES may enable DOMINATION. For example, consider the following set of clauses :

$$\begin{array}{ll}
(C_1) & \neg x_1 \vee x_2 \vee x_3 \\
(C_2) & \qquad\ x_2 \vee x_3 \vee x_4 \\
(C_3) & x_1
\end{array}$$

After we apply UNITRES, we have the following set of clauses.

$$\begin{array}{ll}
(C_1') & \quad x_2 \vee x_3 \\
(C_2) & \quad x_2 \vee x_3 \vee x_4 \\
(C_3) & x_1
\end{array}$$

Note that the clause C_2 is dominated by the clause C_1'. Therefore, UNITRES has enabled MVF.

- **Remark (vi)** : DOMINATION may enable MVF. For example, consider the following set of clauses.

$$\begin{array}{lllll}
(C_1) & x_1 \vee & x_2 & & \\
(C_2) & \neg x_1 \vee & x_2 & \vee\ x_3 & \\
(C_3) & \neg x_1 \vee & \neg x_2 & \vee \neg x_3 \vee & \neg x_4 \\
(C_4) & x_1 & & \vee \neg x_3 \vee & x_4
\end{array}$$

Since the clause C_2 is dominated by the clause C_1, we can apply DOMINATION. The resulting set of clauses is as follows.

$$\begin{array}{lllll}
(C_1) & x_1 \vee & x_2 & & \\
(C_3) & \neg x_1 \vee & \neg x_2 \vee & \neg x_3 \vee & \neg x_4 \\
(C_4) & x_1 & & \vee \neg x_3 \vee & x_4
\end{array}$$

Note that $\neg x_3$ has became a monotone variable after DOMINATION. Therefore, MVF is now enabled.

The above remarks imply that the most efficient order of processing is

UNITRES $\Rightarrow$ DOMINATION $\Rightarrow$ MVF

We now attempt a brief description of the SETSAT control structure. First, try to simplify the original problem by calling UNITRES, DOMINATION, and MVF in that order and then create a MPS format file for MINTO to begin at the root node of the search tree. Solve the LP relaxation of the set covering formulation at current node inside MINTO. If the optimum objective value is equal to half of the number of variables in the formRulation with 0-1 integer solution, stop. A set S of clauses is satisfiable. If the optimum objective value is strictly greater than half of the number of variables in the formulation, fathom the node. If the optimum objective value is equal to half of the number of variables in the formulation with fractional solutions, then call a function 'a_bnds.c' which contains subroutines BIPARTITE, UNITRES, and MVF (MINTO keeps track of fixed variables from a_bnds.c). Now call a function 'a_divide.c' which contains Jeroslow-Wang branching rule and returns a branching variable, and then branch on the variable. Continue until either satisfiability or unsatisfiability is established or the enumeration limit MAXNODES is reached.

3.2 Computational Results

As was mentioned earlier, our empirical testing of SETSAT has been conducted on a set of benchmark satisfiability problems that was compiled at the University of Ulm in Germany. In 1991, Harche, Hooker and Thompson [8] reported a careful comparative study of the performance of a variety of algorithms on this test set. Their implementations are in FORTRAN and the CPU times reported are for a SUN Sparc Station 330. The LP solver used in their study was the XMP system.

The following tables report the results of our experiments on a part of the Ulm test set. We have run SETSAT and MINTO in two default versions by utilizing the set covering formulation of each problem in the test set. MINTO A utilizes all the default cutting plane techniques built into MINTO which may generate clique inequalities, knapsack covers, and flow covers and thus may be regarded as a general purpose branch and cut system for mixed integer programming. In MINTO B we turned off the cut generators and hence MINTO B may be regarded as a general purpose LP-based branch and bound solver.

SETSAT is a branch and bound algorithm and no cutting plane techniques built in MINTO are utilized. We use the following fathoming rule for both SETSAT and MINTO ; if LP solution is greater than half of the number of variables in the set covering formulation, then we fathom the node. Otherwise, we continue. SETSAT C utilizes the Jeroslow-Wang branching rule without a tie-breaking rule(we branch on a z-variable with the largest index whenever we have a tie on weight). SETSAT D utilizes the same branching rule with the following tie-breaking rule : whenever we have a tie on weight, we branch on a variable that appears in the clauses with average shortest length. If a tie occurs here, then we branch on the variable that appears in more clauses.

Our implementations ran on a SUN Sparc Station IPC. For purposes of comparison we have also listed the performance of DPL (with the Jeroslow-Wang branching rule) and Branch and Cut on the benchmark problems as reported in [8]. Hooker and Fedjki [12] developed B & CUT method appeared in the following tables by utilizing the integer programming formulation (5). They not only used the heuristic of Jeroslow-Wang branching rule which selects a branching variable only from variables with a fractional value in the LP relaxation, but also used the so-called rank one cut, a cut which is a positive linear combination of inequalities in the system and rounding up any nonintegers that result (for more details, refer to Harche, Hooker and Thompson [9]).

As we can see from the tables there is no absolute domination by any of the methods. However, some notable trends in the results are :

(i) MINTO default, version B appears to be the weakest method. This is not surprising since it is a general purpose LP-based branch and bound method. It does not take any advantage of the special structure in the set covering formulation.
(ii) MINTO default, version A appears to be only slightly better than version B. A notable exception is the amazing performance of version A on Pigeon Hole Problems. It solved every Pigeon Hole Problem in one node, i.e., with no branching.
(iii) SETSAT D seems to be improved version of SETSAT C. SETSAT also seems to be slightly better than DPL in the size of the enumeration trees it generates. However, SETSAT does a lot more work at each node (solving an LP) and so the smaller enumeration is to be expected.
(iv) SETSAT, on the other hand, generates a slightly larger enumeration tree than Branch and Cut on most problems. This is also to be expected since Branch and Cut uses the power of input resolution in tightening the LP relaxation while SETSAT, in its current version, uses only unit resolution.
(v) An important observation is that the CPU times for SETSAT and MINTO seem not to be competitive in relation to Hooker's implementations of DPL and Branch and Cut. Part of this is perhaps due to the speed of XMP written in FORTRAN over CPLEX on these problems. It should be noted however that considerable speed up in CPLEX may be possible by changing pivot options to deal with the degeneracy in the set covering linear relaxations. Also, since the number of constraints are typically larger than the number of variables, it is perhaps advantageous to solve the dual of the linear relaxations. These improvements along with several others may guarantee the improvement of CPU times for SETSAT and MINTO and therefore are proposed as future research issues.

4. Conclusion

4.1 Summary

We implemented a branch and bound algorithm with the Jeroslow-Wang branching rule based on the set covering formulation. Because of the standardized test problems, we could compare the performance of SETSAT not only with MINTO defaults but also with DPL and Cutting Plane methods as reported by Harche, Hooker and Thompson in [8]. We found that SETSAT results in a smaller number of nodes than MINTO's defaults (except on Pigeon Hole Problems). We believe that the main reason for this is that MINTO's defaults do not exploit the logical reduction techniques that we have built into SETSAT. There is also ample evidence that the Jeroslow-Wang branching rule is more efficient than MINTO's default branching scheme, depth-first search. For most problems, SETSAT results in search trees with fewer nodes than DPL (Improved) as implemented by Hooker. This is because SETSAT goes beyond DPL by solving an LP relaxation at each node. We observed that Hooker and Fedjki's Branch and Cut implementation created smaller enumeration trees than SETSAT. We therefore need to focus on efficient techniques for the generation of deep cutting planes for the set covering polyhedra defined in this paper. This will require refinements of the theory from a computational perspective as well as fresh algorithmic techniques for cut generation.

4.2 Future Research

(i) The phenomenal success of MINTO A on the pigeon hole problems requires further examination. We suspect that the "clique" cutting planes, generated automatically in MINTO A, are the reason that the linear relaxation is able to prove unsatisfiability. This intuitive idea should be made rigorous since it suggests that combinatorial theories can have a substantial impact on logic.

(ii) As has been stated all along, SETSAT represents only the first phase of theorem proving in propositional logic on set covering formulations. Future versions will have to incorporate effective cutting plane methods in SETSAT to exploit the body of knowledge on the polyhedral combinatorics of the set covering formulation of satisfiability. This would combine the power of some types of cutting planes with the logic processing capabilities that we have given SETSAT.

(iii) It may be worthwhile to "fine tune" the CPLEX solver to handle the degeneracy in the set covering linear relaxations more efficiently. Ultimately, we would like to adapt SETSAT to solve the dual of the LP relaxation at each node to exploit the many constraints/fewer variables nature of these model. Ideally, we would also hope that specialized (combinatorial) linear programming methods will be developed for LP relaxations of set covering models that will replace CPLEX completely.

〈Table 1〉 Stuck-at-Zero Problems : Number of Nodes in Search Tree

Problem	Sat. (?)	n	m	MINTO A	MINTO B	SETSAT with B&B(J/W) C	SETSAT with B&B(J/W) D	J. Hooker DPL(J/W)	J. Hooker B&CUT
real1a 12	Y	18	273	1	1	1	1	3	1
real1b 12	Y	15	110	1	1	1	1	3	1
real1o 11	Y	8	20	1	1	1	1	4	1
real1q 11	Y	16	100	1	1	1	1	5	1
real1r 11	Y	16	84	1	1	1	1	5	2
real1u 11	Y	8	24	1	1	1	1	5	1
real1v 11	Y	8	17	1	1	1	1	4	1
real1x 11	Y	14	77	1	1	1	1	4	1
real1y 11	Y	7	32	1	1	1	1	4	1
real2a 12	Y	18	275	1	1	1	1	3	1
real2b 12	Y	15	110	1	1	1	1	3	1
real2c 12	Y	17	342	1	1	1	1	3	1
real2d 12	Y	17	76	1	1	1	1	5	1
real2f 12	Y	16	306	1	1	1	1	4	1
real2g 12	Y	18	123	1	1	1	1	3	1
real2j 12	Y	17	200	1	1	1	1	6	1
real2k 12	Y	17	226	1	1	1	1	7	1
real2l 12	Y	15	119	1	1	1	1	6	1
real2m 12	Y	11	12	1	1	1	1	4	1
real2n 12	Y	18	62	1	1	1	1	4	1
real2o 12	Y	15	37	1	1	1	1	6	1
real2o 11	Y	6	18	1	1	1	1	4	1
real2p 11	Y	8	42	1	1	1	1	3	1
real2r 11	Y	16	84	1	1	1	1	5	2
real2q 11	Y	16	100	1	1	1	1	5	1
real2u 11	Y	8	24	1	1	1	1	5	1
real2v 11	Y	8	17	1	1	1	1	4	1
real2w 11	Y	7	32	1	1	1	1	4	1
real2x 11	Y	14	77	1	1	1	1	4	1

Note) Sat.(?) : Checking satisfiability('Y' stands for 'satisfiable problem' and 'N' stands for 'unsatisfiable problem')
n : The number of automic propositions
m : The number of clauses
A : MINTO's default with cutting planes like clique, knapsack cover, flow cover
B : MINTO's default without cutting planes
C : SETSAT's original result without tie-breaking in Jeroslow-Wang branching rule
D : SETSAT's result with tie-breaking in Jeroslow-Wang branching rule
B&B(J/W) : Branch and Bound with Jeroslow-Wang branching rule based on set covering formulation
DPL(J/W) : DPL with Jeroslow-Wang branching rule
B&CUT : Branch and Cut

〈Table 2〉 Stuck-at-Zero Problems : CPU Seconds

Problem	Sat. (?)	*n*	*m*	MINTO		SETSAT with B&B(J/W)		J. Hooker	
				A	B	C	D	DPL(J/W)	B&CUT
real1a 12	Y	18	273	3	3	3	2	0.07	0.03
real1b 12	Y	15	110	3	3	4	3	0.03	0.04
real1o 11	Y	8	20	3	4	3	3	0.02	0.04
real1q 11	Y	16	100	3	4	3	3	0.03	0.04
real1r 11	Y	16	84	3	4	3	3	0.03	0.12
real1u 11	Y	8	24	3	3	3	3	0.02	0.03
real1v 11	Y	8	17	3	3	3	2	0.02	0.04
real1x 11	Y	14	77	4	4	3	3	0.03	0.07
real1y 11	Y	7	32	3	2	3	3	0.02	0.03
real2a 12	Y	18	275	3	3	3	3	0.05	0.02
real2b 12	Y	15	110	3	2	3	3	0.03	0.06
real2c 12	Y	17	342	3	4	3	3	0.07	0.08
real2d 12	Y	17	76	3	4	3	3	0.03	0.07
real2f 12	Y	16	306	3	4	2	3	0.09	0.07
real2g 12	Y	18	123	3	3	3	3	0.06	0.08
real2j 12	Y	17	200	4	4	3	4	0.05	0.05
real2k 12	Y	17	226	3	3	4	3	0.06	0.08
real2l 12	Y	15	119	3	3	3	3	0.04	0.08
real2m 12	Y	11	12	3	3	3	3	0.02	0.05
real2n 12	Y	18	62	3	3	3	3	0.03	0.05
real2o 12	Y	15	37	3	3	3	3	0.03	0.03
real2o 11	Y	6	18	3	2	3	3	0.02	0.03
real2p 11	Y	8	42	3	3	3	3	0.02	0.02
real2r 11	Y	16	84	4	3	3	3	0.03	0.05
real2q 11	Y	16	100	3	3	3	3	0.03	0.04
real2u 11	Y	8	24	3	3	3	3	0.02	0.05
real2v 11	Y	8	17	3	3	3	2	0.01	0.07
real2w 11	Y	7	32	3	2	3	3	0.03	0.06
real2x 11	Y	14	77	3	3	3	3	0.03	0.04

〈Table 3〉 Random Problems : Number of Nodes in Search Tree

Problem	Sat. (?)	n	m	MINTO		SETSAT with B & B(J/W)		J. Hooker	
				A	B	C	D	DPL(J/W)	B & CUT
jnh 201	Y	100	800	25	544	21	21	32	64
jnh 202	N	100	800	131	700*	109	95	77	7
jnh 203	N	100	800	700*	700*	139	139	153	191
jnh 204	Y	100	800	700*	700*	122	88	32	106
jnh 205	Y	100	800	347	700*	39	39	45	46
jnh 206	N	100	800	700*	700*	285	295	409	135
jnh 207	N	100	800	700*	700*	275	285	343	97
jnh 208	N	100	800	700*	700*	103	101	141	43
jnh 209	Y	100	800	700*	700*	202	202	161	172
jnh 210	Y	100	800	173	700*	49	40	70	57
jnh 1	Y	100	850	700*	700*	109	91	122	69
jnh 2	N	100	850	13	459	15	15	17	5
jnh 3	N	100	850	700*	700*	269	275	817	295
jnh 5	N	100	850	700*	700*	87	47	69	23
jnh 6	N	100	850	700*	700*	143	143	129	103
jnh 7	Y	100	850	12	18	13	13	31	44
jnh 8	N	100	850	629	161	127	119	113	27
jnh 9	N	100	850	700*	700*	87	87	109	15
jnh 10	N	100	850	700*	700*	37	37	55	41
jnh 11	N	100	850	700*	700*	181	181	269	139
jnh 12	Y	100	850	98	7	12	12	11	8
jnh 13	N	100	850	700*	469	75	75	81	7
jnh 14	N	100	850	700*	700*	197	183	137	33
jnh 15	N	100	850	700*	700*	167	167	157	41
jnh 16	N	100	850	700*	700*	285	303	1825	1395
jnh 17	Y	100	850	700*	700*	57	57	52	85
jnh 18	N	100	850	700*	700*	261	279	409	201
jnh 19	N	100	850	557	700*	119	119	169	101
jnh 20	N	100	850	613	700*	73	73	249	101
jnh 301	Y	100	900	700*	700*	274	191	342	300
jnh 302	N	100	900	1	469	11	11	11	1
jnh 303	N	100	900	700*	700*	187	175	203	99
jnh 304	N	100	900	175	581	29	29	25	5

Note) * : Not solved because of excessive CPU time and limit on number of nodes

〈Table 4〉 Random Problems : CPU Seconds

Problem	Sat. (?)	*n* *m*	MINTO		SETSAT with B & B(J/W)		J. Hooker	
			A	B	C	D	DPL(J/W)	B & CUT
jnh 201	Y	100 800	115	1588	106	103	6.3	28.4
jnh 202	N	100 800	1000	2168*	674	575	47.4	42.7
jnh 203	N	100 800	3045*	2480*	939	932	66.6	186.3
jnh 204	Y	100 800	2280*	2842*	468	396	8.4	78.1
jnh 205	Y	100 800	1289	1736*	251	249	12.7	57.2
jnh 206	N	100 800	2558*	2428*	1404	1449	126.6	96.4
jnh 207	N	100 800	2630*	2081*	1578	1591	119.8	65.1
jnh 208	N	100 800	2524*	2498*	653	653	51.6	63.6
jnh 209	Y	100 800	2364*	2075*	1124	1109	50.8	77.9
jnh 210	Y	100 800	794	1964*	229	207	9.3	37.6
jnh 1	Y	100 850	2974*	2215*	502	427	18.6	20.8
jnh 2	N	100 850	459	1930	196	193	15.4	26.3
jnh 3	N	100 850	3689*	3624*	1385	1519	239.1	148.0
jnh 5	N	100 850	3167*	2491*	762	423	42.8	88.3
jnh 6	N	100 850	3141*	2722*	1061	1047	84.2	149.9
jnh 7	Y	100 850	147	148	109	107	7.2	51.4
jnh 8	N	100 850	3304	752	879	860	62.8	58.7
jnh 9	N	100 850	3524*	2922*	760	753	78.9	82.6
jnh 10	N	100 850	3153*	2879*	349	344	36.9	82.0
jnh 11	N	100 850	3177*	2403*	1292	1273	135.1	165.0
jnh 12	Y	100 850	603	94	130	128	5.1	28.8
jnh 13	N	100 850	3288*	2206	542	535	45.3	34.6
jnh 14	N	100 850	3532*	3271*	1358	1256	69.0	76.7
jnh 15	N	100 850	3549*	2835*	1193	1178	83.1	65.3
jnh 16	N	100 850	2721*	2708*	1304	1475	542.4	573.6
jnh 17	Y	100 850	2734*	1979*	345	340	10.8	58.1
jnh 18	N	100 850	3566*	2949*	1516	1581	158.2	132.0
jnh 19	N	100 850	3115*	3253*	959	947	87.5	153.8
jnh 20	N	100 850	2866	2218*	553	552	124.5	126.3
jnh 301	Y	100 900	2829*	4566*	1303	1077	65.8	116.0
jnh 302	N	100 900	32	2204	199	196	13.0	17.0
jnh 303	N	100 900	3856*	3794*	1512	1364	111.4	98.2
jnh 304	N	100 900	1415	3436	331	325	27.3	43.4

〈Table 5〉 Logic Circuit Construction : Number of Nodes in Search Tree

Problem	Sat. (?)	n m	MINTO		SETSAT with B&B(J/W)		J. Hooker	
			A	B	C	D	DPL(J/W)	B&CUT
twoinvr 1	Y	30 79	5	2	2	5	8	2
twoinvrt	N	30 81	20	9	17	17	15	7
tomstest	Y	35 53	1	16	1	1	13	2
newtes 1	Y	56 86	1	1	2	2	23	1
newtest	N	56 87	102	251	7	7	1153	7

〈Table 6〉 Logic Circuit Construction : CPU Seconds

Problem	Sat. (?)	n m	MINTO		SETSAT with B&B(J/W)		J. Hooker	
			A	B	C	D	DPL(J/W)	B&CUT
twoinvr 1	Y	30 79	5	4	4	4	0.12	0.50
twoinvrt	N	30 81	7	5	7	6	0.21	2.7
tomstest	Y	35 53	4	5	4	4	0.07	0.15
newtes 1	Y	56 86	4	4	4	4	0.16	0.02
newtest	N	56 87	18	28	5	5	8.6	1.4

〈Table 7〉 Pigeon Hole Problems : Number of Nodes in Search Tree

Problem	Sat. (?)	n m	MINTO		SETSAT with B&B(J/W)		J. Hooker	
			A	B	C	D	DPL(J/W)	B&CUT
hole6	N	42 133	1	2749	**	1595	6491	6299
hole7	N	56 204	1	5000*	**	**	65561	65061
hole8	N	72 297	1	5000*	**	**	756687	***
hole9	N	90 415	1	5000*	**	1215	***	***
hole10	N	110 561	1	5000*	**	877	***	***

Note) ** : Not solved because memory allocation failed inside MINTO
*** : Not solved by J. Hooker

〈Table 8〉 Pigeon Hole Problems : CPU Seconds

Problem	Sat. (?)	n m	MINTO		SETSAT with B&B(J/W)		J. Hooker	
			A	B	C	D	DPL(J/W)	B&CUT
hole 6	N	42 133	4	315	**	262	23.9	26.5
hole 7	N	56 204	5	810*	**	**	288.7	324.0
hole 8	N	72 297	5	1112*	**	**	4020.4	***
hole 9	N	90 415	6	1511*	**	646	***	***
hole 10	N	110 561	8	2018*	**	657	***	***

REFERENCES

[1] Araque, J.R. and V. Chandru, "Some Facets of Satisfiability," *Annals of Mathematics and Artificial Intelligence*, 1992.

[2] Balas, E. and S.M. Ng, "On the Set Covering Polytope I : All Facets with Coefficients in {0, 1, 2}," *Mathematical Programming*, 43(1989), pp.57-69.

[3] Balas, E. and S.M. Ng, "On the Set Covering Polytope II : Lifting the Facets with Coefficients in {0, 1, 2}," *Mathematical Programming*, 45(1989), pp.1-20.

[4] Blair, C., Jeroslow, R.G. and J.K. Lowe, "Some Results and Experiments in Programming Techniques for Propositional Logic," *Computers and Operations Research*, 13(1988), pp.633-645.

[5] Chandru, V. and J.N. Hooker, *Optimization Methods for Logical Inference*, Wiley Interscience Series in Discrete Mathematics and Optimization, 1999.

[6] Conforti, M., Corneil, D.G. and A.R. Majoub, "K_i-covers I : Complexity and Polytopes," *Discrete Mathematics*, 58(1986), pp.121-142.

[7] Cornuejols, G. and A. Sassano, "On the 0,1 Facets of the Set Covering Polytope," *Mathematical Programming*, 45(1989), pp.45-56.

[8] Harche, F., Hooker, J.N. and G.L. Thompson, "A Computational Study of Satisfiability Algorithms for Propositional Logic," Management Science Research Report MSRR-567, Carnegie Mellon University, June 1991.

[9] Hooker, J.N., "Generalized Resolution and Cutting Planes," *Annals of Operations Research*, 12(1988), pp.217-239.

[10] Hooker, J.N., "A Quantitative Approach to Logical Inference," *Decision Support Systems*, 4(1988), pp.45-69.

[11] Hooker, J.N., "Input Proofs and Rank One Cutting Planes," *ORSA Journal on Computing*, 1(1989), pp.137-145.

[12] Hooker, J.N. and C. Fedjki, "Branch-and-Cut Solution of Inference Problems in Propositional Logic," Working Paper 77-88-89, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213.

[13] Jeroslow, R.E. and J. Wang, "Solving Propositional Satisfiability Problems," *Annals of Mathematics and Artificial Intelligence*, 1(1990), pp.167-187.

[14] Ng, S.M., "On the Set Covering Problem," Summer Paper, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213, 1982.

[15] Nobili, P. and A. Sassano, "Facets and Lifting Procedures for the Set Covering Polytope," *Mathematical Programming*, 45 (1989), pp.111-137.

[16] Salvelsbergh, M.W.P., Sigismondi, G.C., and G.L. Nemhauser, "Functional Description of MINTO, a Mixed INTeger Optimizer," Report COC-91-03C, Georgia Institute of Technology, Atlanta, Georgia 30332.

[17] Sassano, A., "On the Facial Structure of the Set Covering Polytope," *Mathematical Programming*, 44(1989), pp.181-202.