DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 00, 19xx

A Branch and Cut Algorithm for MAX-SAT and Weighted MAX-SAT

Steve Joy, John Mitchell, and Brian Borchers

Abstract. We describe a branch and cut algorithm for both MAX-SAT and weighted MAX-SAT. This algorithm uses the GSAT procedure as a primal heuristic. At each node we solve a linear programming (LP) relaxation of the problem. Two styles of separating cuts are added: resolution cuts and odd cycle inequalities.

We compare our algorithm to an extension of the Davis Putnam Loveland (EDPL) algorithm. Our algorithm is more effective than EDPL on some problems, notably MAX-2-SAT. EDPL is more effective on some other classes of problems.

1. Introduction

The satisfiability problem (SAT) is a problem in propositional logic. A logic formula consists of the conjunction of clauses. Each clause consists of a disjunction of literals. Each literal is a variable or its negation. The SAT problem seeks to find an assignment to the variables which satisfies the logic formula, or an indication that no such assignment exists. The satisfiability problem is NP-complete [**5**].

There are a number of exact algorithms for the satisfiability problem. These include Davis-Putnam-Loveland [**4, 23**], resolution [**26**], and integer programming approaches [**1, 16, 18, 20, 21**]. A number of heuristics that use randomization also exist; the first randomized local search algorithm for satisfiability was due to Gu [**9, 10, 11, 12**]. Other algorithms include the GSAT heuristic [**29, 30**] and the GRASP heuristic [**25**]. For surveys of algorithms for SAT problems see [**13, 14**].

In this paper we investigate the related MAX-SAT problem. Given a collection of clauses, we seek a variable assignment that maximizes the number of satisfied clauses. The weighted MAX-SAT problem assigns a weight to each clause, and seeks an assignment that maximizes the sum of the weights of the satisfied clauses. Both of these problems are NP-hard. It is possible to approximate MAX-SAT within a factor of 1.325 in polynomial time [**6**].

Most SAT heuristics have been extended to MAX-SAT. Several heuristics for MAX-SAT are summarized in Hansen and Jaumard [**15**]. The GSAT heuristic has also been extended to weighted MAX-SAT [**22**].

1991 *Mathematics Subject Classification.* 03B05, 49M35, 65K05, 90C10.
Research supported by ONR Grant N00014–94–1–0391 to Rensselaer Polytechnic Institute.

1052-1798/00 $1.00 + $.25 per page

In this paper we investigate a branch and cut approach to MAX-SAT. We then report on computational results using both our approach and an extension of the Davis-Putnam-Loveland procedure [**2**].

1.1. MAX-SAT as an Integer Programming Problem. A logical variable v_i can be TRUE or FALSE. We replace this variable with a corresponding integer variable x_i. This variable takes on value 1 when v_i is TRUE and 0 when it is FALSE.

An unnegated literal v_i is simply replaced with the expression x_i. A negated literal such as $\overline{v_i}$ can be replaced with the expression $1 - x_i$. When v_i is TRUE, $\overline{v_i}$ is FALSE; x_i is 1, and $1 - x_i$ is 0. When v_i is FALSE, $\overline{v_i}$ is TRUE; x_i is 0, and $1 - x_i$ is 1.

A clause is satisfied if, and only if, at least one of its k literals is TRUE. For the integer problem, we sum the corresponding k expressions. The clause is true if, and only if, the sum is one or more. For example, the clause

$$\overline{v_1} \vee v_3 \vee v_7 \vee \overline{v_9}$$

is equivalent to

$$(1 - x_1) + x_3 + x_7 + (1 - x_9) \geq 1$$

We must have some way of handling the maximization of satisfied constraints. We do this by adding variables to the problem. A clause is either satisfied or it isn't. But this just produces a new clause that is always satisfied:

$$\textit{original clause} \vee \textit{original clause not satisfied}$$

we can replace the 2nd term above with a new variable. There will be one such variable per clause.

Maximizing the sum of the weights of satisfied constraints is equivalent to minimizing the sum of the weights of unsatisfied constraints. This sum is simply the sum of the weight of each clause times the variable indicating that the clause is not satisfied. Given the MAX-SAT problem:

$\overline{v_1} \vee \overline{v_2}$ $\quad weight\ 1$
$v_1 \vee v_2 \vee v_3$ $\quad weight\ 4$
$v_1 \vee \overline{v_2}$ $\quad weight\ 3$

we obtain the equivalent Integer Program (IP):

$$\begin{array}{llllllllllll}
\min & & & & & & w_1 & + & 4w_2 & + & 3w_3 & \\
s.t. & (1-x_1) & + & (1-x_2) & & + & w_1 & & & & \geq & 1 \\
 & x_1 & + & x_2 & + & x_3 & & + & w_2 & & \geq & 1 \\
 & x_1 & + & (1-x_2) & & & & & & + & w_3 & \geq 1
\end{array}$$

$x_i = 0$ *or* $1, i = 1, \ldots, 3$. $w_i = 0$ *or* $1, i = 1, \ldots, 3$

or

$$
\begin{array}{llllllllllllll}
\min & & & & & & & & w_1 & + & 4w_2 & + & 3w_3 & \\
\\
s.t. & - & x_1 & - & x_2 & & & + & w_1 & & & & & \geq & -1 \\
 & & x_1 & + & x_2 & + & x_3 & & & + & w_2 & & & \geq & 1 \\
 & & x_1 & - & x_2 & & & & & & & + & w_3 & \geq & 0
\end{array}
$$

$x_i = 0$ *or* $1, i = 1, \ldots, 3$. $w_i = 0$ *or* $1, i = 1, \ldots, 3$

These added variables are referred to as the *weighted* variables. The other variables are the unweighted variables.

2. Description of Algorithm

2.1. Overview. The basic approach is branch and cut [**7, 8, 17, 24**]. At each node of the tree we solve the linear programming (LP) relaxation obtained by replacing the integrality requirements by the simple bounds:

$0 \leq x_i \leq 1, i = 1, \ldots, n$, $0 \leq w_i \leq 1, i = 1, \ldots, m$

If the solution to the LP is integral, we compare it to the best integral solution so far. If the solution has one or more fractional variables, we branch on one of the fractional variables and repeat the procedure.

The algorithm has several interacting components. A *primal heuristic* is used to obtain an upper bound on the binary solution. This allows us to *fathom* nodes in the branch and cut tree. A *bounds* routine is used to determine if any variable of the LP relaxation can be fixed at zero or one. *Resolution cuts* and *odd cycle inequalities* are added to guide the LP toward a binary solution. Finally, a *branching* routine is used to choose a variable on which to branch.

2.2. Primal Heuristic. The primal heuristic is run once at the beginning of the algorithm. This routine is an efficient randomized local search heuristic, similar to other good heuristics in the literature [**9, 10, 11, 12, 22, 29, 30**]. Several *tries* are attempted. For each try we randomly choose a binary assignment to the variables. We then perform a series of *flips*. By flipping variables (choosing the opposite assignment for a single variable), we attempt to move towards an optimal solution.

There are two types of flips. *Random* flips randomly select a variable to flip. *Best-choice* flips attempt to select the best variable to flip. We compute the effect of a flip, that is the net increase or decrease in the sum of the weights of the satisfied clauses. We choose the variable with the best net increase in this sum. If several variables produce the same net increase, we select one of these variables at random.

Seventy percent of the flips (chosen arbitrarily) are best-choice flips. The random flips are used to assist in escaping from local minima.

2.3. Node Fathoming. The difference between the optimal value of the LP relaxation and that of the incumbent solution is referred to as the *gap*. If the gap is sufficiently small then we can fathom the node. That is, we know that any binary solution along this branch is no better than the incumbent solution.

If the gap is strictly less than some threshold then we can fathom the nodes. Normally this threshold is the greatest common divisor of the clause weights. In the special case where the optimal incumbent solution has one unsatisfied clause of minimal weight, the threshold is this minimum weight.

2.4. Variable bounds. In some cases we can fix variables, thus simplifying the problem. When a variable is fixed, some clauses become *satisfied*.

Let us sum the weighted variables that are fixed at one. This is the *fixed weighted variable sum*, and it represents a lower bound on the LP objective function. Suppose we have a non-satisfied clause whose weight when added to this sum is greater than or equal to the optimal value of the incumbent solution. Since we are seeking a better solution, the weighted variable of the clause must be fixed at zero (or equivalently the unweighted portion of the clause must be satisfied). This clause is a *must satisfy clause*.

A simple form of variable fixing is *sufficient weight fixing*. If we have a *must satisfy clause* then we fix the weighted variable at zero.

If we have a *must satisfy clause* with only a single unfixed unweighted variable then we must fix this variable in such a way as to satisfy the clause. This is known as *unit clause fixing*.

If a variable appears in only the positive sense in the non-satisfied clauses then we can fix the variable at one. Likewise, if the variable is always negated then we can fix it at zero. This is known as *monotone variable fixing*.

First, we examine the clauses that are not satisfied. We compute the number of non-fixed variables in each clause. For each non-fixed variable, we count the number of clauses in which it appears in the positive and negated senses. If a variable appears in only the positive sense then fix it via monotone variable fixing. If there is only a single non-fixed variable in a clause of sufficient weight then fix it via unit clause fixing.

When a variable is fixed, this satisfies some constraints. We decrement the counters for the other variables in this newly satisfied constraint. This potentially allows us to fix additional variables that may now have become monotone. As we fix a variable, all constraints having the opposite sense of the variable are now shorter. If a clause is now of length one and has sufficient weight then we can fix the remaining variable via unit clause fixing.

If an original constraint is satisfied, we set the clause's weighted variable to zero. This is known as *satisfied clause fixing*.

If all the unweighted variables of an original clause are fixed in such a way that none of the unweighted literals satisfy the clause then we set the clause's weighted variable to one. This is known as *unsatisfied clause fixing*.

2.5. Cut generation. There are two types of cuts: *resolution cuts* and *odd cycle inequalities*. The cuts are applied locally (at this node of the branch and bound tree and its descendents) rather than to the whole tree. Resolution cuts are discussed by Hooker and Fedjki [**18, 19, 20**]. Odd cycle inequalities are discussed by Cheriyan et al. [**3**].

2.5.1. *Resolution cuts. Resolution cuts* arise by combining two clauses: one with the positive sense of a variable, the other clause with the negative sense. There must be exactly one such literal. The *resolvent* consists of literals (even those arising from weighted variables) found in either clause except for the variable of opposite sense. For example, given the clauses:

$$\overline{x_1} \vee x_3 \vee x_7 \vee w_9$$

$$x_2 \vee \overline{x_3} \vee x_7 \vee w_{11}$$

we can resolve to generate the following:

$$\overline{x_1} \vee x_2 \vee x_7 \vee w_9 \vee w_{11}$$

The resolution routine is activated once at each node of the tree. It consists of a sequence of *passes*, each designed to generate resolvents matching certain criteria. In each pass for each variable of each resolvable clause, the list of clauses with the opposite sense of that variable is obtained. This clause is resolved (if possible) with each of the other clauses on the list. If the resolvent matches the given criteria, it is added to the list of resolvable clauses for the next pass. If this clause is of length one or less (counting only the original unweighted variables) and is a sufficiently deep separating cut (violated by 0.3 or more with the current LP solution) then it is added to the list of separating cuts (the length and depth restrictions were determined by experimentation to yield the fastest algorithms). This process continues until a sufficient number of resolvents have been generated or all such clauses have been exhausted.

At the end of each pass, constraints that can not possibly be useful in future passes are removed from consideration. The resolvents that are generated are reduced to a minimal set through *absorption*. In absorption if the unweighted literals of clause A are a subset of the literals of clause B, then clause A implies clause B. In this case, clause B is absorbed by clause A and B is removed from consideration. The list of separating cuts is also reduced to a minimal set through absorption.

The first pass requires that the resolvents match the requirements for the separating cuts, that is, of length one or less and violated by 0.3 or more.

The second pass requires that the resolvents be of length three or less.

The third pass requires that the resolvents be of length two or less.

The fourth pass requires that the resolvents be of length one or less.

2.5.2. *Odd cycle inequalities.* The odd cycle inequalities combine clauses with two unfixed unweighted variables. A clause is considered *odd* if both unfixed unweighted variables are negated, or if both are not negated; otherwise, the clause is considered *even*. A *cycle* $x_{i_1}, \ldots, x_{i_k}$ in the unfixed unweighted variables is sought. The first constraint involves variables x_{i_1} and x_{i_2}, the next x_{i_2} and x_{i_3}, the next x_{i_3} and x_{i_4}, ... the last clause involving x_{i_k} and x_{i_1}. The cycle is said to be odd if when we "add up" these constraints, we obtain an odd total. By insuring that the cycle is odd, we know that the right hand side will be odd, and will be rounded up when we divide the coefficients of the resulting constraint by two.

This is best illustrated by an example. Suppose we have the constraints below. Variables x_1–x_4 are unfixed unweighted variables, the other variables are unfixed and weighted. The first constraint is a cut (generated at a previous branch and cut node) since it involves multiple weighted variables.

$$\begin{array}{rrrrrrrrl}
x_1 & +x_2 & & & +w_{10} & +w_{11} & & & \geq 1 \\
 & +x_2 & -x_3 & & & +w_{11} & & & \geq 0 \\
 & & -x_3 & -x_4 & & & +w_{12} & & \geq -1 \\
-x_1 & & & -x_4 & & & & +w_{13} & \geq -1
\end{array}$$

The first, third, and fourth constraints are odd; the second is even. Adding these constraints together we obtain:

$$2x_2 - 2x_3 - 2x_4 + w_{10} + 2w_{11} + w_{12} + w_{13} \geq -1$$

Dividing by 2 and rounding coefficients we obtain:

$$x_2 - x_3 - x_4 + w_{10} + w_{11} + w_{12} + w_{13} \geq 0$$

We are again seeking only separating cuts. Odd cycles force this increase in the right hand side thus potentially generating a separating cut. Even cycles do not have this potential. If we find an odd cycle of accumulated weight (defined below) $1{-}2\delta$, then we obtain a separating cut of depth δ. We are seeking a cut of depth 0.3, so we are seeking an odd cycle of weight 0.4 or less. We also require that the cut is of length two or less (the length and depth restrictions were determined by experimentation to yield the fastest algorithms).

We use a modified version of Dijkstra's shortest path algorithm to find the separating cuts. Dijkstra's algorithm starts from a given start node of a tree and uses edge weights to find the shortest path from this start node to any other. Initially, the start node is assigned weight 0. All others are assigned infinite weight. All nodes are initially unmarked. The algorithm then picks the lowest weight unmarked node. This node is now marked. For any unmarked neighbors of this chosen node with

$$\textit{chosen weight} + \textit{weight of edge from chosen to neighbor} < \textit{weight of neighbor}$$

we replace the neighbor's weight with this sum. This process continues until all the shortest paths have been found (all nodes are marked). The algorithm can be slightly modified to record the actual shortest path to a node (in addition to its length).

We use the algorithm once per variable. In each iteration this variable is the *start variable*.

- The edges represent clauses with two unfixed unweighted literals. The weight of an edge is the surplus of the constraint plus the sum of the weights of the clause's weighted variables (as these are likely to appear in the sum exactly once and so will have their coefficient rounded up thus increasing the slack).
- The weight of a node is the accumulated edge weights on the path from the start node.
- There are two nodes per variable. One node represents the length of the shortest odd path to the node. The other that of the shortest even path.
- The *start node* corresponds to an even length path to the *start variable*. This null path is considered even.
- The algorithm terminates when either:
    1. The chosen node (most recently marked) has a weight exceeding 0.4 (indicating that any separating cut is no deeper than 0.3, a failure), or has a length greater than 2 (a failure), or
    2. The chosen node is the odd sense of the start variable. Thus, we have an odd path from the start node to itself (a cycle).

The odd cycle cuts can only be applied if we start with a large number of two variable clauses or after a significant number of variables have been fixed. Hence, these cuts tend to be applied only deep into the tree. We will investigate the effects of a more robust approach in a future paper.

2.6. Branching. After generating cuts and applying bounds, we branch if there are still fractional variables in the LP solution.

The branching scheme we use is a modification of the Jeroslow-Wang [**21**] scheme.

We examine the probability of satisfying all the constraints if we randomly assign values to the remaining unfixed variables. For simplification, we assume that the probabilities of such an assignment satisfying each constraint are independent, and that each binary variable assignment has a probability of one half. We attempt to pick a variable whose assignment will have the greatest change in this probability.

For each fractional variable, we find a weight for the positive and negative sense as follows. We investigate all the clauses (original plus the resolvents) containing the given sense of the variable. We consider the effect of picking the opposite sense of the variable (making it harder to satisfy this constraint). By selecting the wrong choice we have turned the k variable constraint into a k−1 variable constraint. The probability of satisfying this constraint is:

$$1 - \left(\frac{1}{2}\right)^{k-1}$$

We wish to minimize the product of such terms (hence to minimize the sum of their logs). A similar approach is used for the opposite sense of the variable. We sum these two values and branch on the variable with the greatest sum. Such a variable will have a significant effect along either of the two branches and should, therefore, tend to generate a smaller branch and bound tree. For this variable, we first explore the more heavily weighted branch.

This approach differs from Jeroslow-Wang in that:

1. We give greater weight to the shorter clauses.
2. We consider the effect on both branches.

3. Test Results

In this section we compare our branch and cut code (B+C) to a MAX-SAT extension of the Davis-Putnam-Loveland (EDPL) algorithm [**2**]. Both are implemented in C. The branch and cut code uses the MINTO package of Savelsbergh et al. [**27**] (replacing default modules with those discussed in previous sections). All tests are executed on an IBM RS6000/390 with 128 Megabytes of memory. The abbreviation "MEM" in the tables indicates that a test terminated due to insufficient memory.

We tested both algorithms on a set of unweighted problems generated by the MWFF package of Selman [**28**]. Both MAX-2-SAT and MAX-3-SAT problems were examined. Various numbers of clauses were tried. This produced some problems that were satisfiable and others with several unsatisfiable clauses. For MAX-3-SAT, we also tried different number of variables. The results were sorted on the number of unsatisfied clauses, as this gave a strong indication of the difficulty of the problem.

For MAX-2-SAT problems, the branch and cut code appears superior. EDPL only performs better on problems with a small number of clauses. Both algorithms take more CPU time as the number of clauses grow; however, the execution time of EDPL grew explosively. The branch and cut code tends to generate very small

problem name	clauses	unsat. clauses	B+C		EDPL	
			CPU	nodes	CPU	backtracks
p2180_5	180	0	0.84	1	0.1	1
p2180_8	180	1	5.20	1	0.7	2
p2180_2	180	2	5.44	1	0.7	10
p2180_3	180	2	5.25	1	0.7	40
p2180_6	180	2	5.03	1	0.7	47
p2180_9	180	2	5.39	1	0.7	34
p2180_1	180	3	5.09	1	0.7	162
p2180_7	180	4	5.41	1	0.8	964
p2180_4	180	4	5.43	1	0.9	1773
p2180_10	180	4	5.29	1	0.9	1933
p2200_8	200	4	5.23	1	0.8	1065
p2200_6	200	4	5.34	1	0.9	1304
p2200_3	200	4	5.43	1	0.9	2219
p2200_5	200	5	5.40	1	1.5	7982
p2200_1	200	5	5.62	1	2.3	12303
p2200_7	200	5	5.42	1	2.3	13549
p2200_2	200	6	5.57	1	4.2	32700
p2200_9	200	6	5.52	1	8.9	80824
p2200_10	200	6	5.95	3	8.2	73117
p2200_4	200	7	5.88	3	20.1	217362

TABLE 1. Computational results for 100 variable MAX-2-SAT problems with a small number of clauses

trees while EDPL generates large ones. As a result, branch and cut performs dramatically better.

Our algorithm does not perform nearly so well for MAX-3-SAT problems. The search tree is generally smaller than that of EDPL. However, the evaluation at each node is much more expensive thus resulting in much greater execution times. EDPL's advantage diminishes with increasing numbers of unsatisfied clauses (for example the series of problems in Tables 5–7); however, even here, EDPL performs better.

We also examine the Steiner "D" weighted tree problems [**22**] (see Table 13). Our current implementation of the primal heuristic is too primitive to handle these large variable problems well. So, for these problems, we run our code with the primal heuristic disabled. The EDPL code takes in excess of 12 hours on all of these problems. This is true even if the primal heuristic is disabled and the code is provided with the correct incumbent value. Our branch and cut code handles many of these problems with ease.

4. Conclusions and Future Directions

We compared two algorithms for solving MAX-SAT. Neither one was universally superior. The general trend was that branch and cut works best on MAX-2-SAT and the Steiner "D" problems (where the average clause length is small). Trends suggest that branch and cut may also be better if the system contains a large

problem name	clauses	unsat. clauses	B+C		EDPL	
			CPU	nodes	CPU	backtracks
p2220_5	220	4	5.73	1	0.8	1045
p2220_3	220	4	5.76	1	1.2	3699
p2220_9	220	4	5.83	1	1.0	2233
p2220_2	220	5	5.70	1	1.8	10039
p2220_7	220	6	5.81	1	5.7	48360
p2220_1	220	7	6.01	3	10.3	90586
p2220_10	220	7	6.23	3	9.8	91322
p2220_8	220	7	7.02	9	21.6	197734
p2220_4	220	8	6.19	3	17.4	172550
p2220_6	220	8	6.33	3	41.5	418421
p2240_1	240	6	5.91	1	5.0	12402
p2240_7	240	7	6.85	3	26.6	242434
p2240_9	240	9	6.82	3	33.9	337982
p2240_10	240	9	6.96	3	84.2	817004
p2240_2	240	9	6.94	3	115.4	1150960
p2240_4	240	9	6.81	3	163.9	1767247
p2240_5	240	9	8.04	5	150.4	1634963
p2240_6	240	9	6.87	3	181.4	1775023
p2240_8	240	9	6.61	3	126.4	1357751
p2240_3	240	11	9.27	7	1205.4	13050216

TABLE 2. Computational results for 100 variable MAX-2-SAT problems with a somewhat small number of clauses

number of unsatisfied clauses. EDPL appears to generally work best for MAX-3-SAT. The EDPL code generates larger search trees, but spends much less time per node.

There are a number of possible directions to explore:

One possibility is some kind of hybrid algorithm, perhaps using a limited EDPL step for node fathoming and/or variable fixing. This should help reduce the tree size with a relatively low cost per node.

Another possibility is the addition of deeper cuts such as max-clique [**3**] inequalities.

A third possibility is a branch step that takes into account the slacks on the LP relaxation. It may also be worthwhile to investigate branching on several variables, perhaps reducing the total number of LP evaluations.

A final possibility is to reduce or eliminate cuts in the first few levels of the tree. Despite the addition of a large number of cuts here, the tree expanded in a binary fashion for several levels (see table 14). These cuts increase the size of the LP's and, therefore, the time to solve them.

problem name	clauses	unsat. clauses	B+C		EDPL	
			CPU	nodes	CPU	backtracks
p2260_10	260	6	6.49	1	4.1	29932
p2260_4	260	8	6.80	3	12.5	109971
p2260_7	260	8	6.10	1	36.4	340233
p2260_3	260	9	6.24	1	61.7	588508
p2260_2	260	9	6.54	3	66.2	631689
p2260_6	260	11	9.12	5	510.7	5214818
p2260_8	260	11	9.33	5	939.1	9722169
p2260_1	260	11	11.37	9	1543.1	15149775
p2260_9	260	12	12.36	9	2556.3	26165444
p2260_5	260	12	11.38	7	4293.8	46674569
p2280_10	280	10	7.49	3	227.2	2118261
p2280_5	280	11	7.98	3	1398.0	14876069
p2280_8	280	11	8.50	5	495.6	4770966
p2280_9	280	11	8.24	3	407.3	3853794
p2280_7	280	12	8.87	5	3437.0	35347297
p2280_1	280	13	12.29	7	2738.0	28997281
p2280_3	280	13	13.34	11	3151.6	32091970
p2280_6	280	14	19.82	15	29111.1	313402437
p2280_2	280	15	19.53	13	32085.0	332713465
p2280_4	280	15	20.81	15	45268.2	503462478

TABLE 3. Computational results for 100 variable MAX-2-SAT problems with a medium number of clauses

problem name	clauses	unsat. clauses	B+C		EDPL	
			CPU	nodes	CPU	backtracks
p2300_2	300	13	8.27	3	2229.1	21334438
p2300_3	300	13	8.22	3	2497.9	23764177
p2300_4	300	14	10.96	5	4618.5	47629271
p2300_1	300	15	20.04	11	10564.3	104553113
p2300_5	300	15	17.63	11	29738.0	309999407
p2300_9	300	15	18.61	11	8746.0	85235320
p2300_10	300	15	19.16	11	12693.1	125325067
p2300_6	300	17	64.57	35	69325.3	704246333
p2300_8	300	17	37.23	19	Not Run	Not Run
p2300_7	300	20	92.97	45	Not Run	Not Run
p2400_3	400	25	80.08	27	Not Run	Not Run
p2400_7	400	26	102.35	29	Not Run	Not Run
p2400_6	400	27	117.51	35	Not Run	Not Run
p2400_2	400	28	83.44	27	Not Run	Not Run
p2400_4	400	28	115.30	29	Not Run	Not Run
p2400_1	400	29	122.42	37	Not Run	Not Run
p2400_5	400	29	226.72	57	Not Run	Not Run
p2400_10	400	30	565.93	135	Not Run	Not Run
p2400_8	400	33	518.31	103	Not Run	Not Run
p2400_9	400	34	1518.91	281	Not Run	Not Run

TABLE 4. Computational results for 100 variable MAX-2-SAT problems with a large number of clauses

problem name	clauses	unsat. clauses	B+C		EDPL	
			CPU	nodes	CPU	backtracks
test215_1	215	0	0.81	1	0.1	1
test215_3	215	1	5.29	15	0.5	23
test215_2	215	1	5.76	23	0.5	25
test215_5	215	1	6.52	33	0.5	31
test215_4	215	2	34.47	353	0.6	562
test250_3	250	0	2.63	1	0.2	1
test250_1	250	2	17.27	89	0.6	314
test250_4	250	2	21.52	123	0.6	306
test250_5	250	2	28.53	177	0.6	362
test250_2	250	4	253.76	1467	3.2	20001

TABLE 5. Computational results for 50 variable MAX-3-SAT problems with a small number of clauses

problem name	clauses	unsat. clauses	B+C		EDPL	
			CPU	nodes	CPU	backtracks
test300_3	300	4	78.80	275	2.1	10116
test300_1	300	4	147.10	577	2.3	11073
test300_2	300	5	249.48	891	5.9	42151
test300_5	300	5	325.48	1107	8.4	59216
test300_4	300	6	493.78	1583	16.2	130802
test350_4	350	5	168.53	417	4.8	28409
test350_2	350	6	460.75	1195	15.1	112539
test350_1	350	8	1718.31	3867	103.3	915731
test350_3	350	8	1877.87	4363	97.8	864778
test350_5	350	8	2369.56	5659	118.0	1048876
test400_3	400	8	758.43	1349	62.8	482523
test400_2	400	8	1163.13	2101	68.9	521495
test400_5	400	8	1020.13	1895	76.1	594532
test400_4	400	11	4249.03	6341	519.8	4541184
test400_1	400	11	6501.42	9701	690.1	6196183

TABLE 6. Computational results for 50 variable MAX-3-SAT problems with a medium number of clauses

problem name	clauses	unsat. clauses	B+C		EDPL	
			CPU	nodes	CPU	backtracks
test450_4	450	10	1522.54	2139	162.6	1237498
test450_5	450	11	2326.24	3003	357.9	2817502
test450_3	450	11	3676.28	4549	533.7	4268781
test450_1	450	12	5480.85	6197	868.9	7129533
test450_2	450	14	13440.50	14077	2752.4	24408217
test500_1	500	15	9324.21	7505	2465.1	19442201
test500_3	500	16	17271.87	13443	4081.6	32959363
test500_4	500	16	17460.65	13387	3923.8	31096996
test500_5	500	19	MEM	MEM	15832.7	136515702

TABLE 7. Computational results for 50 variable MAX-3-SAT problems with a large number of clauses

problem name	clauses	unsat. clauses	B+C		EDPL	
			CPU	nodes	CPU	backtracks
s323_1	323	0	1.37	1	0.2	1
s323_2	323	0	2.02	1	0.1	1
s323_3	323	0	1.75	1	0.1	1
s323_4	323	0	1.01	1	0.1	1
s323_5	323	1	18.39	77	0.8	94
s350_1	350	1	15.80	45	0.9	85
s350_2	350	1	20.68	79	0.9	76
s350_3	350	1	21.09	89	0.9	77
s350_4	350	1	21.05	87	0.9	73
s350_5	350	2	143.99	749	1.3	1176
s375_1	375	0	2.41	1	0.1	1
s375_2	375	2	139.37	557	1.5	1527
s375_3	375	2	199.87	907	1.5	1649
s375_5	375	3	437.32	1477	4.0	11649
s375_4	375	3	537.77	1917	4.7	14669

TABLE 8. Computational results for 75 variable MAX-3-SAT problems with a small number of clauses

problem name	clauses	unsat. clauses	B+C		EDPL	
			CPU	nodes	CPU	backtracks
s400_3	400	2	190.70	733	1.7	1894
s400_5	400	3	653.70	2049	4.1	11460
s400_2	400	4	1300.32	3463	20.3	90478
s400_4	400	4	1874.79	5369	25.9	118064
s400_1	400	5	4276.55	10353	104.1	554165
s425_1	425	4	1689.55	3869	20.4	83776
s425_3	425	4	1841.67	4439	21.8	88537
s425_4	425	5	3378.52	7815	84.6	442233
s425_2	425	5	5214.78	11643	110.0	559420
s425_5	425	6	MEM	MEM	478.5	2799709
s450_4	450	5	3761.84	7149	97.9	481736
s450_5	450	6	7208.57	13225	295.9	1603476
s450_1	450	6	MEM	MEM	466.4	2619212
s450_2	450	7	MEM	MEM	1639.2	10218689
s450_3	450	7	MEM	MEM	1212.7	7362249

TABLE 9. Computational results for 75 variable MAX-3-SAT problems with a medium number of clauses

problem name	clauses	unsat. clauses	B+C		EDPL	
			CPU	nodes	CPU	backtracks
s475_2	475	6	6105.25	9767	345.5	1844920
s475_3	475	6	6258.74	9765	282.3	1500066
s475_5	475	7	7668.66	10287	678.2	3741527
s475_1	475	7	12541.21	18867	856.5	4841953
s475_4	475	8	MEM	MEM	2651.3	15909584
s500_2	500	7	9331.30	12099	758.0	4205882
s500_4	500	7	MEM	MEM	985.3	5693206
s500_3	500	8	MEM	MEM	2652.9	15799725

TABLE 10. Computational results for 75 variable MAX-3-SAT problems with a large number of clauses

problem name	clauses	unsat. clauses	B+C		EDPL	
			CPU	nodes	CPU	backtracks
o430_4	430	0	1.66	1	0.1	1
o430_2	430	0	3.67	1	0.1	1
o430_3	430	0	1.75	1	0.2	1
o430_1	430	1	120.85	545	1.4	320
o430_5	430	2	1833.07	7991	5.2	8834
o450_5	450	0	1.50	1	0.1	1
o450_4	450	1	65.81	247	1.3	199
o450_3	450	1	87.50	347	1.4	312
o450_1	450	2	988.54	3589	3.5	5221
o450_2	450	2	1336.35	4667	3.9	5712
o475_3	475	1	72.03	251	4.1	6168
o475_4	475	1	72.81	253	1.4	230
o475_5	475	1	65.66	213	1.4	281
o475_1	475	2	593.13	2059	3.2	4212
o475_2	475	2	1028.67	3347	3.3	4265
o500_1	500	0	2.47	1	0.2	1
o500_3	500	2	1068.16	3389	2.9	3345
o500_2	500	3	4795.02	12683	31.3	87816
o500_4	500	4	MEM	MEM	162.5	572489
o500_5	500	4	MEM	MEM	203.1	735535

TABLE 11. Computational results for 100 variable MAX-3-SAT problems with a small number of clauses

problem name	clauses	unsat. clauses	B+C		EDPL	
			CPU	nodes	CPU	backtracks
o525_1	525	2	493.49	1287	2.3	1910
o525_4	525	3	3640.80	8073	21.3	54483
o525_3	525	3	5379.72	12301	29.0	74311
o525_5	525	3	5596.78	11893	25.7	66390
o525_2	525	5	MEM	MEM	1122.6	4525046
o550_2	550	3	2162.29	3969	11.2	24487
o550_1	550	3	3746.04	7367	30.0	78768
o550_5	550	4	16601.80	28423	147.9	479767
o550_3	550	5	MEM	MEM	837.4	3304456
o550_4	550	6	MEM	MEM	7526.6	33281390
o575_5	575	4	12477.14	19139	151.9	491192
o575_3	575	5	MEM	MEM	602.2	2191647
o575_1	575	6	MEM	MEM	4370.9	18334787
o575_2	575	7	MEM	MEM	10488.5	47346157

TABLE 12. Computational results for 100 variable MAX-3-SAT problems with a large number of clauses

problem name	variables	clauses	ave clause length	optimal value	B+C	
					CPU	nodes
steind2	1295	1765	1.31	220	4423.23	7599
steind3	1416	1885	1.25	1646	3.63	1
steind4	1499	2074	1.28	2044	5.08	1
steind5	1749	2646	1.34	3419	14.47	1
steind7	2045	2325	1.15	103	6.53	7
steind8	2166	2544	1.15	1180	3.46	1
steind9	2249	2797	1.20	1585	4.84	1
steind10	2499	3261	1.23	2219	12.68	1
steind11	5120	5882	1.17	29	2374.98	523
steind12	5009	5039	1.01	42	2.78	1
steind13	5166	5499	1.06	544	4.76	1
steind14	5249	5709	1.08	740	6.33	1
steind15	5499	6202	1.11	1193	13.49	1
steind16	25032	25133	1.01	13	208.37	4
steind18	25166	25412	1.01	262	13.32	1
steind19	25249	25585	1.01	359	15.59	1
steind20	25499	26041	1.02	558	21.45	1

TABLE 13. Computational results for Steiner "D" tree problems

depth	nodes	resolution cuts/node	odd cycle cuts/node
0	1	0.000	0.000
1	2	1.500	0.000
2	4	6.250	0.500
3	8	19.000	1.250
4	16	12.875	2.625
5	32	13.344	3.875
6	64	11.594	6.719
7	128	8.438	7.781
8	256	7.516	8.609
9	512	7.227	9.416
10	971	7.270	9.747
11	1461	7.600	9.721
12	1674	7.224	9.648
13	1428	7.429	9.148
14	991	7.272	8.954
15	593	6.390	8.155
16	316	5.551	6.965
17	180	6.172	5.989
18	102	7.686	5.912
19	67	6.060	5.776
20	40	4.775	4.525
21	27	4.704	3.111
22	10	3.700	2.300
23	4	2.750	1.750
24	1	0.000	0.000

TABLE 14. Cuts per node in the branch and cut tree of problem test500_4 in table 7

References

[1] C. E. Blair, R. G. Jeroslow, and J. K. Lowe. Some results and experiments in programming techniques for propositional logic. *Computers and Operations Research*, 13(5):633–645, 1986.

[2] B. Borchers and J. Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. Technical report, Mathematics Department, New Mexico Tech, Socorro, NM 87801, October 1995.

[3] J. Cheriyan, W. H. Cunningham, L. Tunçel, and Y. Wang. A linear programming and rounding approach to max 2-sat. Technical report, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Canada N2L 3G1, 1996. To appear in: *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, D. S. Johnson and M. A. Trick (eds), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS, 1995.

[4] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. Assoc. Comput. Mach.*, 7:201–215, 1960.

[5] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[6] Michel X. Goemans and David P. Williamson. Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming. *J. Assoc. Comput. Mach.*, 42(6):1115–1145, 1995.

[7] M. Grötschel and O. Holland. Solution of large-scale travelling salesman problems. *Mathematical Programming*, 51(2):141–202, 1991.

[8] M. Grötschel, M. Jünger, and G. Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32:1195–1220, 1984.

[9] J. Gu. *Parallel Algorithms and Architectures for Very Fast AI Search*. PhD thesis, University of Utah, 1989.

[10] J. Gu. Efficient local search for very large-scale satisfiability problem. *SIGART Bulletin*, 3(1):8–12, January 1992, ACM Press.

[11] J. Gu. Local search for satisfiability (SAT) problem. *IEEE Trans. on Systems, Man, and Cybernetics*, 23(4):1108–1129, Jul. 1993, and 24(4):709, Apr. 1994.

[12] J. Gu. Global optimization for satisfiability (SAT) problem. *IEEE Trans. on Knowledge and Data Engineering*, 6(3):361–381, Jun. 1994, and 7(1):192, Feb. 1995.

[13] J. Gu. Parallel algorithms for satisfiability (SAT) problem. *DIMACS Series on Discrete Mathematics and Theoretical Computer Science – Parallel Processing on Discrete Optimization Problems*, 22:105–161, Jul. American Mathematical Society, 1995.

[14] J. Gu, P.W. Purdom, J. Franco, and B.W. Wah. Algorithms for satisfiability (SAT) problem: A survey. *DIMACS Volume Series on Discrete Mathematics and Theoretical Computer Science: The Satisfiability (SAT) Problem*, American Mathematical Society, 1996.

[15] P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.

[16] F. Harche and G. L. Thompson. The column subtraction algorithm: An exact method for solving weighted set covering, packing and partitioning problems. *Computers and Operations Research*, 21:689–705, 1994.

[17] K. L. Hoffman and M. Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39(6):657–682, 1993.

[18] J. N. Hooker. Resolution vs. cutting plane solution of inference problems: some computational experience. *Operations Research Letters*, 7(1):1–7, 1988.

[19] J. N. Hooker. Resolution and the integrality of satisfiability problems. Technical report, Carnegie Mellon University, 1995. To appear in: *Mathematical Programming*.

[20] J. N. Hooker and C. Fedjki. Branch-and-cut solution of inference problems in propositional logic. *Annals of Mathematics and Artificial Intelligence*, 1:123–139, 1990.

[21] R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and AI*, 1:167–187, 1990.

[22] Y. Jiang, H. Kautz, and B. Selman. Solving problems with hard and soft constraints using a stochastic algorithm for MAX-SAT. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 07974, 1995.

[23] D. Loveland. *Automated Theorem-Proving: A Logical Basis*. North-Holland, New York, 1978.

[24] M. W. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[25] Mauricio G. C. Resende and Thomas A. Feo. A grasp for satisfiability. Technical report, Department of Combinatorics and Optimization, University of Waterloo, 1995. To appear in: *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, D. S. Johnson and M. A. Trick (eds), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS, 1995.

[26] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[27] M. W. P. Savelsbergh, G. C. Sigismondi, and G. L. Nemhauser. MINTO: a Mixed INTeger Optimizer. Technical Report Memorandum COSOR 91–18, Eindhoven University of Technology, 1991.

[28] B. Selman. Mwff: Program for generating random max k-sat instances. available from DIMACS.

[29] B. Selman and H. A. Kautz. An empirical study of greedy local search for satisfiability testing. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-92), San Jose, CA*, 1993.

[30] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), San Jose, CA*, pages 440–446, July 1992.

Mathematics Department, Rensselaer Polytechnic Institute, Troy, NY 12180
E-mail address: joys@rpi.edu

Mathematics Department, Rensselaer Polytechnic Institute, Troy, NY 12180
E-mail address: mitchj@rpi.edu

Mathematics Department, New Mexico Tech, Socorro, NM 87801
E-mail address: borchers@nmt.edu