Solving MAX-SAT and Weighted MAX-SAT Problems Using Branch-and-Cut[*]

Steve Joy[†] John E. Mitchell[‡] Brian Borchers[§]

February 28, 1998

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 and a Semi-Definite Programming (SDP) approach. Our algorithm is more effective than EDPL on some problems, notably MAX-2-SAT. EDPL and SDP are 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 [7].

There are a number of exact algorithms for the satisfiability problem. These include Davis-Putnam-Loveland [6, 27], resolution [31], and integer programming approaches [2, 19, 22, 24, 25]. A number of heuristics that use randomization also exist; the first randomized local search algorithm for satisfiability was due to Gu [12, 13, 14, 15]. Other algorithms include the GSAT heuristic [33, 34] and the GRASP heuristic [30]. For surveys of algorithms for SAT problems see [16, 17].

[*]Research supported by ONR Grant N00014–94–1–0391 to Rensselaer Polytechnic Institute

[†]Mathematical Sciences, Rensselaer Polytechnic Institute, Troy, NY 12180.

[‡]Mathematical Sciences, Rensselaer Polytechnic Institute, Troy, NY 12180. Partially supported by a grant from the Dutch NWO and Delft University of Technology for the 1997–98 academic year, while visiting TWI/SSOR at Delft University of Technology.

[§]Mathematics Department, New Mexico Tech, Socorro, NM 87801.

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 [8].

Most SAT heuristics have been extended to MAX-SAT. Several heuristics for MAX-SAT are summarized in Hansen and Jaumard [18]. The GSAT heuristic has also been extended to weighted MAX-SAT [26].

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 [4].

We also compare our algorithm with a semi-definite programming (SDP) approach [9, 20]. This method formulates an SDP relaxation of the MAX-SAT problem, solves the relaxation using an interior point algorithm, and, if necessary, repeatedly adds cutting planes and solves the modified relaxation. The implementation described in this paper may not solve a given instance to optimality, but in those cases, it could be made into an exact algorithm with the addition of a branch-and-bound phase. The SDP approach provides both an upper and a lower bound on the optimal value, so it is possible to terminate with guaranteed optimality if these two bounds agree.

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 $1 - x_i$.

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:

$$original\ clause \vee original\ clause\ not\ satisfied$$

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:

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

we obtain the equivalent Integer Program (IP):

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

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

or

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

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

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 [10, 11, 21, 29]. 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 [12, 13, 14, 15, 26, 33, 34]. 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. After adding cuts, the LP is re-evaluated. This process is repeated until no more cuts can be added. Resolution cuts are discussed by Hooker and Fedjki [22, 23, 24]. Odd cycle inequalities are discussed by Cheriyan *et al.* [5].

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 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.

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

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

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

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 literal. In each iteration this literal is the *start literal.*

- 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 literal. 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 literal.* 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 literal. 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 literal 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 [25] 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 original clauses containing the given sense of the variable. The chance of a random assignment satisfying this clause is:

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

We then consider the relative weights of the clause. For a clause of weight, w, with $k \geq 2$, we assign the following value to the clause:

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

For a singleton clause ($k = 0$) of weight, w, we assign the following value to the clause:

$$w0.05$$

instead of merely using $k = 1$ in the previous formula. This considerably reduces the weight assigned to singleton clauses. This value was determined through experimentation.

We sum the clause values over all clauses in which this sense of the variable occurs. 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 lesser weight to singleton 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 [4] and a semi-definite programming (SDP) approach [3]. All are implemented in C. The branch and cut code uses the MINTO package of Savelsbergh *et al.* [28] (replacing default modules with those discussed in previous sections). All tests are executed on an IBM RS6000/390 with 128 Megabytes of memory. The results are contained in Tables 1–14. The abbreviation "MEM" in the tables indicates that a test terminated due to insufficient memory.

We tested these algorithms on a set of unweighted problems generated by the MWFF package of Selman [32]. Both MAX-2-SAT and MAX-3-SAT problems were examined. Various numbers of variables and clauses were tried. This produced some problems that were satisfiable and others with several unsatisfiable clauses. The results were sorted on the number of unsatisfied clauses, as this gave a strong indication of the difficulty of the problem. The semi-definite programming approach can only be directly applied to MAX-2-SAT problems.

For MAX-2-SAT problems (see Tables 1–5), 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 trees while EDPL generates large ones. As a result, branch and cut performs dramatically better. SDP appears to fare poorly on most of these problems. However, on very dense problems (see Table 5), SDP outperforms the branch and cut code. The run time of the SDP approach appears almost insensitive to the number of clauses.

Our algorithm does not perform nearly so well for MAX-3-SAT problems (see Tables 6–13). 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 6–8); however, even here, EDPL performs better.

We also examined the Steiner "D" weighted tree problems [26] (see Table 14). Our current implementation of the primal heuristic is too primitive to handle these large variable problems well. So, for these problems, we ran 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.

problem name	clauses	unsat. clauses	B+C		EDPL		SDP		
			CPU	nodes	CPU	backtracks	CPU	lower	upper
p2180_5	180	0	0.62	1	0.1	1	366.85	180	180
p2180_8	180	1	4.96	1	0.7	2	1015.99	179	179
p2180_6	180	1	4.89	1	0.7	47	911.65	178	178
p2180_3	180	2	5.18	1	0.7	40	1417.08	178	178
p2180_9	180	2	5.23	1	0.7	34	2497.35	178	179
p2180_2	180	2	5.29	1	0.7	10	2152.11	178	179
p2180_1	180	3	4.99	1	0.7	162	2299.32	177	178
p2180_7	180	4	5.25	1	0.8	964	1322.69	176	176
p2180_4	180	4	5.19	1	0.9	1773	2315.90	176	177
p2180_10	180	4	5.31	1	0.9	1933	1106.76	176	176
p2200_8	200	4	5.24	1	0.8	1065	613.46	196	196
p2200_6	200	4	5.27	1	0.9	1304	2249.11	196	197
p2200_3	200	4	5.58	1	0.9	2219	2231.58	196	197
p2200_5	200	5	5.39	1	1.5	7982	643.29	195	195
p2200_1	200	5	5.48	1	2.3	12303	2465.95	195	196
p2200_7	200	5	5.41	1	2.3	13549	2169.85	195	196
p2200_2	200	6	5.67	1	4.2	32700	2373.41	194	195
p2200_10	200	6	6.02	3	8.2	73117	2146.13	194	196
p2200_9	200	6	5.56	1	8.9	80824	2162.91	194	195
p2200_4	200	7	5.56	1	20.1	217362	2549.42	193	194

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

problem name	clauses	unsat. clauses	B+C		EDPL		SDP		
			CPU	nodes	CPU	backtracks	CPU	lower	upper
p2220_5	220	4	5.81	1	0.8	1045	2279.29	216	217
p2220_3	220	4	5.71	1	1.2	3699	424.00	216	216
p2220_9	220	4	5.80	1	1.0	2233	1924.46	216	216
p2220_2	220	5	5.51	1	1.8	10039	2134.21	215	216
p2220_7	220	6	5.94	1	5.7	48360	2083.10	214	215
p2220_10	220	7	5.91	1	9.8	91322	1450.78	213	213
p2220_1	220	7	5.83	1	10.3	90586	2240.66	213	214
p2220_8	220	7	6.46	5	21.6	197734	2190.08	213	214
p2220_4	220	8	5.74	1	17.4	172550	2156.89	212	213
p2220_6	220	8	5.93	1	41.5	418421	2127.74	212	213
p2240_7	240	7	6.12	1	26.6	242434	2219.66	233	234
p2240_9	240	9	6.22	1	33.9	337982	906.15	231	231
p2240_10	240	9	6.05	1	84.2	817004	2660.03	231	232
p2240_2	240	9	6.22	1	115.4	1150960	2245.81	231	232
p2240_4	240	9	6.22	1	163.9	1767247	1557.34	231	231
p2240_5	240	9	6.70	1	150.4	1634963	2149.84	231	232
p2240_6	240	9	6.47	1	181.4	1775023	2200.40	231	232
p2240_8	240	9	6.35	1	126.4	1357751	1110.96	231	231
p2240_3	240	11	6.57	1	1205.4	13050216	2115.74	229	230

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

problem name	clauses	unsat. clauses	B+C		EDPL		SDP		
			CPU	nodes	CPU	backtracks	CPU	lower	upper
p2260_10	260	6	6.53	1	4.1	29932	419.95	254	254
p2260_4	260	8	6.54	1	12.5	109971	1155.04	252	252
p2260_7	260	8	6.23	1	36.4	340233	442.69	252	252
p2260_3	260	9	6.37	1	61.7	588508	902.17	251	251
p2260_2	260	9	6.53	1	66.2	631689	2206.08	251	252
p2260_6	260	11	6.89	1	510.7	5214818	2148.64	249	250
p2260_8	260	11	7.92	1	939.1	9722169	2144.15	249	250
p2260_1	260	11	7.43	1	1543.1	15149775	2294.05	249	250
p2260_9	260	12	7.70	1	2556.3	26165444	2264.76	248	249
p2260_5	260	12	7.39	1	4293.8	46674569	662.34	248	248
p2280_10	280	10	6.83	1	227.2	2118261	2401.60	270	272
p2280_8	280	11	7.54	1	495.6	4770966	2169.24	268	270
p2280_9	280	11	7.11	1	407.3	3853794	1161.27	269	269
p2280_5	280	11	7.06	1	1398.0	14876069	1181.26	269	269
p2280_7	280	12	6.98	1	3437.0	35347297	2105.63	268	269
p2280_1	280	13	7.89	1	2738.0	28997281	2241.37	267	268
p2280_3	280	13	7.56	1	3151.6	32091970	2102.66	267	268
p2280_6	280	14	16.37	15	29111.1	313402437	2199.46	266	268
p2280_2	280	15	12.68	5	32085.0	332713465	2212.57	265	266
p2280_4	280	15	10.39	3	45268.2	503462478	2243.10	265	267

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		SDP		
			CPU	nodes	CPU	backtracks	CPU	lower	upper
p2300_2	300	13	7.18	1	2229.1	21334438	2307.46	287	288
p2300_3	300	13	7.50	1	2497.9	23764177	2399.92	287	288
p2300_4	300	14	7.29	1	4618.5	47629271	2202.70	286	287
p2300_9	300	15	7.51	1	8746.0	85235320	1220.04	285	285
p2300_1	300	15	8.18	1	10564.3	104553113	2086.05	285	286
p2300_10	300	15	8.24	1	12693.1	125325067	2292.71	285	286
p2300_5	300	15	11.35	3	29738.0	309999407	2263.76	285	287
p2300_6	300	17	9.67	1	69325.3	704246333	2200.03	283	284
p2300_8	300	17	9.42	1	Not Run	Not Run	2123.03	283	284
p2300_7	300	20	32.33	11	Not Run	Not Run	2266.63	280	281
p2400_3	400	25	13.36	1	Not Run	Not Run	2097.99	375	376
p2400_7	400	26	30.69	11	Not Run	Not Run	2127.00	374	375
p2400_6	400	27	18.70	3	Not Run	Not Run	2134.13	373	374
p2400_4	400	28	13.58	1	Not Run	Not Run	640.15	372	372
p2400_2	400	28	18.29	3	Not Run	Not Run	2134.42	372	373
p2400_1	400	29	21.75	3	Not Run	Not Run	2157.36	371	372
p2400_5	400	29	45.13	15	Not Run	Not Run	2167.33	371	373
p2400_10	400	30	64.82	11	Not Run	Not Run	2141.70	370	372
p2400_8	400	33	71.13	11	Not Run	Not Run	2166.69	367	369
p2400_9	400	34	121.73	19	Not Run	Not Run	2074.66	365	368

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

variables	clauses	B+C	SDP		
		CPU	CPU	lower	upper
50	100	3.32	56.15	96	96
	200	5.70	46.61	184	184
	300	9.51	93.63	268	268
	400	12.65	34.45	355	355
	500	26.94	78.56	434	434
	1000	64.76	18.75	838	838
	1500	1933.55	250.39	1228	1229
	2000	1350.11	84.56	1614	1614
	2500	3378.69	348.00	2007	2008
	3000	523.75	38.77	2419	2419
	3500	10142.31	88.67	2783	2783
	4000	5761.59	58.76	3169	3169
	4500	16958.45	352.68	3540	3541
	5000	37241.84	298.27	3933	3933
100	1000	22915.42	5153.65	857	859
	1500	MEM	5321.79	1270	1272
	2000	MEM	1549.69	1685	1685
	2500	MEM	6422.84	2055	2060
150	1000	MEM	24445.65	1285	1297
	1500	MEM	23274.42	1309	1310
	2000	MEM	30934.79	1702	1711
	2500	MEM	21335.99	2104	2112

Table 5: Computational results on dense MAX-2-SAT problems

problem name	clauses	unsat. clauses	B+C		EDPL	
			CPU	nodes	CPU	backtracks
test215_1	215	0	0.64	1	0.1	1
test215_3	215	1	5.99	23	0.5	23
test215_2	215	1	6.74	25	0.5	25
test215_5	215	1	7.18	33	0.5	31
test215_4	215	2	31.19	281	0.6	562
test250_3	250	0	1.53	1	0.2	1
test250_1	250	2	19.13	71	0.6	314
test250_4	250	2	21.56	85	0.6	306
test250_5	250	2	24.64	119	0.6	362
test250_2	250	4	172.02	759	3.2	20001

Table 6: 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	70.41	169	2.1	10116
test300_1	300	4	116.84	285	2.3	11073
test300_2	300	5	193.81	469	5.9	42151
test300_5	300	5	250.22	615	8.4	59216
test300_4	300	6	351.27	773	16.2	130802
test350_4	350	5	208.03	323	4.8	28409
test350_2	350	6	334.62	553	15.1	112539
test350_1	350	8	1052.44	1583	103.3	915731
test350_3	350	8	1074.87	1639	97.8	864778
test350_5	350	8	1346.03	2289	118.0	1048876
test400_3	400	8	556.56	609	62.8	482523
test400_5	400	8	664.45	787	76.1	594532
test400_2	400	8	765.23	821	68.9	521495
test400_4	400	11	1808.15	1811	519.8	4541184
test400_1	400	11	3199.41	3455	690.1	6196183

Table 7: 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	924.22	791	162.6	1237498
test450_5	450	11	1353.03	1095	357.9	2817502
test450_3	450	11	1965.93	1579	533.7	4268781
test450_1	450	12	2572.63	1911	868.9	7129533
test450_2	450	14	5298.33	3795	2752.4	24408217
test500_1	500	15	4446.50	2343	2465.1	19442201
test500_3	500	16	7999.25	3793	4081.6	32959363
test500_4	500	16	6712.91	3381	3923.8	31096996
test500_5	500	19	19831.93	10681	15832.7	136515702

Table 8: 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_4	323	0	0.72	1	0.1	1
s323_2	323	0	0.93	1	0.1	1
s323_3	323	0	1.22	1	0.1	1
s323_1	323	0	4.26	1	0.2	1
s323_5	323	1	22.72	81	0.8	94
s350_1	350	1	24.24	69	0.9	85
s350_2	350	1	24.38	67	0.9	76
s350_4	350	1	29.49	107	0.9	73
s350_3	350	1	160.09	765	0.9	77
s350_5	350	2	128.47	639	1.3	1176
s375_1	375	0	0.92	1	0.1	1
s375_2	375	2	136.71	583	1.5	1527
s375_3	375	2	187.54	783	1.5	1649
s375_5	375	3	308.18	829	4.0	11649
s375_4	375	3	355.89	1061	4.7	14669

Table 9: 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	216.12	723	1.7	1894
s400_5	400	3	447.54	1195	4.1	11460
s400_2	400	4	807.87	1615	20.3	90478
s400_4	400	4	1042.92	2611	25.9	118064
s400_1	400	5	2207.02	4529	104.1	554165
s425_3	425	4	1057.23	2183	21.8	88537
s425_1	425	4	1146.52	2147	20.4	83776
s425_4	425	5	1971.31	3787	84.6	442233
s425_2	425	5	2862.11	5617	110.0	559420
s425_5	425	6	4390.22	7133	478.5	2799709
s450_4	450	5	2286.86	3551	97.9	481736
s450_5	450	6	3113.21	4757	295.9	1603476
s450_1	450	6	5932.71	8739	466.4	2619212
s450_3	450	7	5011.38	6999	1212.7	7362249
s450_2	450	7	5152.23	7921	1639.2	10218689

Table 10: 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	2944.16	3821	345.5	1844920
s475_3	475	6	3108.99	3901	282.3	1500066
s475_5	475	7	3522.77	3671	678.2	3741527
s475_1	475	7	4687.80	5803	856.5	4841953
s475_4	475	8	7074.77	8055	2651.3	15909584
s500_2	500	7	4001.40	3997	758.0	4205882
s500_4	500	7	MEM	MEM	985.3	5693206
s500_3	500	8	MEM	MEM	2652.9	15799725

Table 11: 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_3	430	0	0.97	1	0.2	1
o430_4	430	0	1.08	1	0.1	1
o430_2	430	0	2.19	1	0.1	1
o430_1	430	1	169.68	543	1.4	320
o430_5	430	2	1429.61	7541	5.2	8834
o450_5	450	0	1.06	1	0.1	1
o450_4	450	1	88.37	233	1.3	199
o450_3	450	1	126.68	361	1.4	312
o450_1	450	2	818.97	3073	3.5	5221
o450_2	450	2	1144.84	4023	3.9	5712
o475_5	475	1	90.83	201	1.4	281
o475_4	475	1	97.27	235	1.4	230
o475_3	475	1	112.73	297	4.1	6168
o475_1	475	2	457.51	1571	3.2	4212
o475_2	475	2	983.93	3171	3.3	4265
o500_1	500	0	3.65	1	0.2	1
o500_3	500	2	983.06	3139	2.9	3345
o500_2	500	3	2908.52	7095	31.3	87816
o500_5	500	4	7927.12	16889	203.1	735535
o500_4	500	4	9873.80	18793	162.5	572489

Table 12: 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	450.48	1085	2.3	1910
o525_4	525	3	2330.12	5087	21.3	54483
o525_3	525	3	3290.84	6953	29.0	74311
o525_5	525	3	3864.73	7551	25.7	66390
o525_2	525	5	MEM	MEM	1122.6	4525046
o550_2	550	3	1290.38	1985	11.2	24487
o550_1	550	3	2411.89	4231	30.0	78768
o550_5	550	4	8769.65	13813	147.9	479767
o550_3	550	5	17128.79	24793	837.4	3304456
o550_4	550	6	MEM	MEM	7526.6	33281390
o575_5	575	4	6488.93	8609	151.9	491192
o575_3	575	5	13342.57	17253	602.2	2191647
o575_1	575	6	MEM	MEM	4370.9	18334787
o575_2	575	7	MEM	MEM	10488.5	47346157

Table 13: 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	86.35	105
steind3	1416	1885	1.25	1646	3.08	1
steind4	1499	2074	1.28	2044	4.44	1
steind5	1749	2646	1.34	3419	11.42	1
steind7	2045	2325	1.15	103	2.90	3
steind8	2166	2544	1.15	1180	3.14	1
steind9	2249	2797	1.20	1585	4.73	1
steind10	2499	3261	1.23	2219	10.46	1
steind11	5120	5882	1.17	29	467.52	333
steind12	5009	5039	1.01	42	2.57	1
steind13	5166	5499	1.06	544	4.29	1
steind14	5249	5709	1.08	740	5.81	1
steind15	5499	6202	1.11	1193	11.41	1
steind16	25032	25133	1.01	13	18.41	2
steind18	25166	25412	1.01	262	13.42	1
steind19	25249	25585	1.01	359	15.02	1
steind20	25499	26041	1.02	558	20.54	1

Table 14: Computational results for Steiner "D" tree problems

depth	nodes	resolution cuts/node	odd cycle cuts/node
0	1	0.00	0.00
1	2	24.00	0.00
2	4	12.25	0.00
3	8	12.88	3.50
4	16	13.00	3.88
5	32	7.84	6.66
6	64	7.23	11.63
7	128	5.91	14.77
8	256	5.05	16.60
9	508	4.47	18.27
10	947	3.58	18.77
11	1393	3.09	18.10
12	1503	2.42	16.58
13	1128	2.01	15.26
14	623	1.78	14.02
15	253	1.70	11.93
16	105	1.31	9.72
17	26	1.46	6.00
18	10	2.30	8.10
19	1	4.00	5.00

Table 15: Cuts per node in the branch and cut tree of problem test500_5 in table 8

4 Conclusions and Future Directions

We compared three algorithms for solving MAX-SAT. None 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 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. The SDP approach works well on MAX-2-SAT problems with a very large number of clauses. It may be possible to improve the performance of an SDP algorithm for problems with fewer clauses by implementing an algorithm that exploits sparsity, for example a dual algorithm as used by Benson *et al.* [1] for MAXCUT problems.

There are a number of ways in which the performance of the branch-and-cut code might be improved:

- 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 [5] 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 15). These cuts increase the size of the LP's and, therefore, the time to solve them.

References

[1] S. J. Benson, Y. Ye, and X. Zhang. Solving large-scale sparse semidefinite programs for combinatorial optimization. Technical report, Department of Management Sciences, University of Iowa, Iowa City, Iowa 52242, September 1997.

[2] 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.

[3] B. Borchers. CSDP, a C library for semidefinite programming. Technical report, Mathematics Department, New Mexico Tech, Socorro, NM 87801, March 1997.

[4] 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. Revised: March 1997. To appear in Journal of Combinatorial Optimization.

[5] J. Cheriyan, W. H. Cunningham, L. Tunçel, and Y. Wang. A linear programming and rounding approach to max 2-sat. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, DIMACS Series In Discrete Mathematics and Theoretical Computer Science*, volume 26, pages 395–414. AMS, 1996.

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

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

[8] 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.

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

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

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

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] P. Hansen, B. Jaumard, and V. Mathon. Constrained nonlinear 0–1 programming. *ORSA Journal on Computing*, 5:97–119, 1993.

[19] F. Harche, J. N. Hooker, and G. L. Thompson. A computational study of satisfiability algorithms for propositional logic. *ORSA Journal on Computing*, 6:423–435, 1994.

[20] C. Helmberg and F. Rendl. Solving quadratic (0,1)-problems by semidefinite programs and cutting planes. Technical Report SC-95-35, Konrad-Zuse-Zentrum fuer Informationstechnik, Berlin, 1995.

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

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

[23] J. N. Hooker. Resolution and the integrality of satisfiability problems. *Mathematical Programming*, 74:1–10, 1996.

[24] 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.

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

[26] 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.

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

[28] G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.

[29] M. 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.

[30] Mauricio G. C. Resende and Thomas A. Feo. A grasp for satisfiability. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, DIMACS Series In Discrete Mathematics and Theoretical Computer Science*, volume 26, pages 499–520. AMS, 1996.

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

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

[33] 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.

[34] 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-91), Anaheim, CA*, pages 440–446, July 1992.