APPROXIMATE SOLUTION OF WEIGHTED MAX-SAT PROBLEMS USING GRASP*

M.G.C. RESENDE†, L.S. PITSOULIS‡, AND P.M. PARDALOS§

Abstract. Computing the optimal solution to an instance of the weighted maximum satisfiability problem (MAX-SAT) is difficult even when each clause contains at most two literals. In this paper, we describe a greedy randomized adaptive search procedure (GRASP) for computing approximate solutions of weighted MAX-SAT problems. The heuristic is tested on a large set of test instances. Computational experience indicates the suitability of GRASP for this class of problems.

Key words. Combinatorial optimization, logic, satisfiability, MAX-SAT, greedy, artificial intelligence, local search, GRASP, heuristics, computer implementation

1. Introduction. Let $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m$ be m clauses, involving n Boolean variables $x_1, x_2, \ldots, x_n$, which can take on only the values `true` or `false` (1 or 0). In addition, for each clause $\mathcal{C}_i$, there is an associated nonnegative weight w_i. Define clause i to be

$$\mathcal{C}_i = \bigvee_{j=1}^{n_i} l_{ij},$$

where the literals $l_{ij} \in \{x_j, \bar{x}_j \mid j = 1, \ldots, n\}$. In the weighted *Maximum Satisfiability Problem* (MAX-SAT), one is to determine the assignment of truth values to the n variables that maximizes the sum of the weights of the satisfied clauses. Note that the classical Satisfiability Problem (SAT) is a special case of the MAX-SAT in which all clauses have unit weight and one wants to decide if there is a truth assignment of total weight m.

The satisfiability problem is a central problem in artificial intelligence, mathematical logic, and combinatorial optimization. Problems in computer vision, VLSI design, databases, automated reasoning, computer-aided design and manufacturing, involve the solution of instances of the satisfiability problem. Furthermore, SAT is the basic problem in computational complexity [3, 8]. Developing efficient exact algorithms and heuristics for satisfiability problems can lead to general approaches for solving combinatorial optimization problems.

SAT was the first problem shown to be NP-complete [3, 8]. However, 2SAT, where each clause contains exactly two literals, can be solved in polynomial time. In contrast, MAX-SAT is known to be NP-complete [8] even when each clause contains exactly two literals (MAX-2SAT). Therefore, it is unlikely that any polynomial time algorithm exists that can optimally solve MAX-SAT.

The weighted MAX-SAT has a natural mixed integer programming formulation. Let $y_j = 1$ if Boolean variable x_j is `true` and $y_j = 0$ otherwise. Furthermore, the continuous variable $z_i = 1$ if clause $\mathcal{C}_i$ is satisfied and $z_i = 0$, otherwise. Consider the

*April 10, 1997– Latest version can be found in URL = `ftp://netlib.att.com/math/people/-mgcr/doc/gmaxsat.ps.Z`

†AT&T Research, Florham Park, NJ 07932 USA

‡Center for Applied Optimization, Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL 32611 USA

§Center for Applied Optimization, Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL 32611 USA

mixed integer linear program

$$\max \quad F(y,z) = \sum_{i=1}^{m} w_i z_i$$

subject to

$$\sum_{j \in I_i^+} y_j + \sum_{j \in I_i^-} (1 - y_j) \geq z_i, \quad i = 1, \ldots, m,$$

$$y_j \in \{0,1\}, \quad j = 1, \ldots, n,$$

$$0 \leq z_i \leq 1, \quad i = 1, \ldots, m,$$

where I_i^+ (resp. I_i^-) denotes the set of variables appearing unnegated (resp. negated) in clause $\mathcal{C}_i$.

Although finding an exact solution to the MAX-SAT is NP-Complete, there has been a significant amount of research in developing ϵ-approximation algorithms for the MAX-SAT. Let the optimal solution of a maximization problem be F^*. Then, an ϵ-approximation algorithm is an algorithm that produces, in polynomial time, a solution F such that

$$F \geq \epsilon F^*, \text{where } 0 < \epsilon < 1.$$

The first ϵ-approximation algorithm for MAX-SAT is described in Johnson [14], where a $\frac{1}{2}$-approximation algorithm is presented. When each clause has at least k literals, then the algorithm becomes a $(1 - \frac{1}{2^k})$-approximation. Until recently, the best ϵ-approximation algorithms were those by Yannakakis [22] and Goemans and Williamson [9] with $\epsilon = \frac{3}{4}$. Goemans and Williamson [9] proved that the solution of the linear relaxation of the mixed integer linear program given above is a $\frac{3}{4}$-approximate solution. Subsequently, Goemans and Williamson [10] described an improved .758-approximation algorithm based on semidefinite programming. In an extension of the result in [10], Feig and Goemans [4] derive a .931-approximation algorithm for MAX-2SAT. Moreover, it is known that there exists a constant $c < 1$ such that no c-approximation algorithm exists for MAX-SAT and MAX-2SAT, unless $P = NP$ [1]. For MAX-SAT $c = \frac{77}{80}$ while for MAX-2SAT $c = \frac{95}{96}$ [2]. The conclusion that can be drawn from the above discussion of ϵ-approximation algorithms is that although the theoretical bound for approximation algorithms for the MAX-2SAT has been approached, this is not the case for the MAX-SAT, justifying the investigation of heuristic algorithms for it. See [11, 12, 13] for surveys of algorithms for MAX-SAT and SAT.

A greedy randomized adaptive search procedure (GRASP) is a randomized heuristic for combinatorial optimization [5, 6]. In this paper, we describe an implementation of GRASP for solving the weighted MAX-SAT problem. GRASP is an iterative process, with each GRASP iteration consisting of two phases, a construction phase and a local search phase. The best overall solution is kept as the result.

In the construction phase, a feasible solution is iteratively constructed, one element at a time. At each construction iteration, the choice of the next element to be added is determined by ordering all elements in a candidate list with respect to

a greedy function. This function measures the (myopic) benefit of selecting each element. The heuristic is adaptive because the benefits associated with every element are updated at each iteration of the construction phase to reflect the changes brought on by the selection of the previous element. The probabilistic component of a GRASP is characterized by randomly choosing one of the best candidates in the list, but not necessarily the top candidate. This choice technique allows for different solutions to be obtained at each GRASP iteration, but does not necessarily compromise the power of the adaptive greedy component of the method.

As is the case for many deterministic methods, the solutions generated by a GRASP construction are not guaranteed to be locally optimal with respect to simple neighborhood definitions. Hence, it is usually beneficial to apply a local search to attempt to improve each constructed solution. While such local optimization procedures can require exponential time from an arbitrary starting point, empirically their efficiency significantly improves as the initial solutions improve. Through the use of customized data structures and careful implementation, an efficient construction phase can be created which produces good initial solutions for efficient local search. The result is that often many GRASP solutions are generated in the same amount of time required for the local optimization procedure to converge from a single random start. Furthermore, the best of these GRASP solutions is generally significantly better than the solution obtained from a random starting point.

An especially appealing characteristic of GRASP is the ease with which it can be implemented. Few parameters need to be set and tuned (candidate list size and number of GRASP iterations) and therefore development can focus on implementing efficient data structures to assure quick GRASP iterations. Finally, GRASP can be trivially implemented on a parallel processor in an MIMD environment. For example, each processor can be initialized with its own copy of the procedure, the instance data, and an independent random number sequence. The GRASP iterations are then performed in parallel with only a single global variable required to store the best solution found over all processors.

GRASP has been applied successfully to a wide range of combinatorial optimization problems, and a survey of GRASP can be found in [6].

The paper is organized as follows. In Section 2, we describe in detail the GRASP heuristic as applied to MAX-SAT, including the construction and local search phases. In Section 3, we report on computational results on weighted MAX-SAT instances derived from a set of standard SAT test problems with randomly generated weights. The integer programming formulation presented in this section is used to compare the heuristic performance with the solutions produced by the CPLEX `mips` solver. Concluding remarks are made in Section 4.

2. GRASP for the weighted MAX-SAT. As outlined in Section 1, a GRASP possesses four basic components: a greedy function, an adaptive search strategy, a probabilistic selection procedure, and a local search technique. These components are interlinked, forming an iterative method that constructs a feasible solution, one element at a time, guided by an adaptive greedy function, and then searches the neighborhood of the constructed solution for a locally optimal solution. Figure 2.1 shows a GRASP in pseudo-code. Lines 1 and 2 of the pseudo-code input the problem instance and initialize the data structures. The best solution found so far is initialized in line 3. The GRASP iterations are carried out in lines 4 through 8. Each GRASP iteration has a construction phase (line 5) and a local search phase (line 6). If necessary, the solution is updated in line 7. The GRASP returns the best solution found.

```
procedure grasp(RCLSize,MaxIter,RandomSeed)
1    InputInstance();
2    InitializeDataStructures();
3    BestSolutionFound = 0;
4    do k = 1,..., MaxIter →
5        ConstructGreedyRandomizedSoln(RCLSize,RandomSeed);
6        LocalSearch(BestSolutionFound);
7        UpdateSolution(BestSolutionFound);
8    od;
9    return(BestSolutionFound)
end grasp;
```

FIG. 2.1. *A generic GRASP pseudo-code*

```
procedure ConstructGreedyRandomizedSoln(RCLSize,RandomSeed,x)
1    do k = 1,..., n →
2        MakeRCL(RCLSize);
3        s = SelectIndex(RandomSeed);
4        AssignVariable(s,x);
5        AdaptGreedyFunction(s);
6    od;
end ConstructGreedyRandomizedSoln;
```

FIG. 2.2. *GRASP construction phase pseudo-code*

In this section, we describe the GRASP for the weighted MAX-SAT. The algorithm presented in this paper is a generalization of the GRASP for SAT described in [20]. To accomplish this, we describe in detail the ingredients of the GRASP, i.e. the construction and local search phases. To describe the construction phase, one needs to provide a candidate definition (for the restricted candidate list), provide an adaptive greedy function, and specify the candidate restriction mechanism. For the local search phase, one must define the neighborhood and specify a local search algorithm.

2.1. Construction phase. The construction phase of a GRASP builds a solution, around whose neighborhood a local search is carried out in the local phase, producing a locally optimal solution. This construction phase solution is built, one element at a time, guided by a greedy function and randomization. Figure 2.2 describes in pseudo-code a GRASP construction phase. Since in the MAX-SAT problem there are n variables to be assigned, each construction phase consists of n iterations. In `MakeRCL` the restricted candidate list of assignments is set up. The index of the next variable to be assigned is chosen in `SelectIndex`. The variable selected is assigned a truth value in `AssignVariable`. In `AdaptGreedyFunction` the greedy function that guides the construction phase is changed to reflect the assignment just made. To describe these steps in detail, we need some definitions. Let $N = \{1, 2, \ldots, n\}$ and $M = \{1, 2, \ldots, m\}$ be sets of indices for the set of variables and clauses, respectively. Solutions are constructed by setting one variable at a time to either 1 (`true`) or 0 (`false`). Therefore, to define a restricted candidate list, we have 2 potential candidates for each yet-unassigned variable: assign the variable to 1 or assign the variable to 0.

```
procedure AdaptGreedyFunction(s)
1     if s > 0 →
2           for j ∈ Γ_s^+ →
3                 for k ∈ L_j (k ≠ s) →
4                       if x_k is unnegated in clause j →
5                             Γ_k^+ = Γ_k^+ − {j}; γ_k^+ = γ_k^+ − w_j;
6                       fi;
7                       if x_k is negated in clause j →
8                             Γ_k^− = Γ_k^− − {j}; γ_k^− = γ_k^− − w_j;
9                       fi;
10                rof;
11          rof;
12          Γ_s^+ = ∅; Γ_s^− = ∅;
13          γ_s^+ = 0; γ_s^− = 0;
14    fi;
15    if s < 0 →
16          for j ∈ Γ_{−s}^− →
17                for k ∈ L_j (k ≠ −s) →
18                      if x_k is unnegated in clause j →
19                            Γ_k^+ = Γ_k^+ − {j}; γ_k^+ = γ_k^+ − w_j;
20                      fi;
21                      if x_k is negated in clause j →
22                            Γ_k^− = Γ_k^− − {j}; γ_k^− = γ_k^− − w_j;
23                      fi;
24                rof;
25          rof;
26          Γ_{−s}^+ = ∅; Γ_{−s}^+ = ∅;
27          γ_{−s}^+ = 0; γ_{−s}^− = 0;
28    fi;
29    return
end AdaptGreedyFunction;
```

FIG. 2.3. `AdaptGreedyFunction` *pseudo-code*

```
procedure LocalSearch(x,BestSolutionFound)
1     BestSolutionFound = C(x);
2     GenerateGains(x,G,0);
3     G_k = max{G_i | i = 1, ..., n} ;
4     for G_k ≠ 0 →
5           Flip value of x_k;
6           GenerateGains(x,G,k);
7     rof;
8     BestSolutionFound = C(x);
9     return;
end LocalSearch;
```

FIG. 2.4. *The local search procedure in pseudo-code*

We now define the adaptive greedy function. The idea behind the greedy function is to maximize the total weight of yet-unsatisfied clauses that become satisfied after the assignment of each construction phase iteration. For $i \in N$, let Γ_i^+ be the set of unassigned clauses that would become satisfied if variable x_i were to be set to `true`. Likewise, let Γ_i^- be the set of unassigned clauses that would become satisfied if variable x_i were to be set to `false`. Define

$$\gamma_i^+ = \sum_{j \in \Gamma_i^+} w_j \text{ and } \gamma_i^- = \sum_{j \in \Gamma_i^-} w_j.$$

The greedy choice is to select the variable x_k with the largest γ_k^+ or γ_k^- value. If $\gamma_k^+ > \gamma_k^-$, then the assignment $x_k = 1$ is made, else $x_k = 0$. Note that with every assignment made, the sets Γ_i^+ and Γ_i^- change for all i such that x_i is not assigned a truth value, to reflect the new assignment. This consequently changes the values of γ_i^+ and γ_i^-, characterizing the adaptive component of the heuristic.

Next, we discuss restriction mechanisms for the restricted candidate list (RCL). The RCL is set up in `MakeRCL` of the pseudo-code of Figure 2.2. We consider two forms of restriction: value restriction and cardinality restriction.

Value restriction imposes a parameter based *achievement level*, that a candidate has to satisfy to be included in the RCL. In this way we ensure that a random selection will be made among the best candidates in any given assignment. Let $\gamma^* = \max\{\gamma_i^+, \gamma_i^- \mid x_i \text{ yet unassigned}\}$. Let α $(0 \leq \alpha \leq 1)$ be the restricted candidate parameter. We say a candidate $x_i =$ `true` is a *potential candidate* for the RCL if $\gamma_i^+ \geq \alpha \cdot \gamma^*$. Likewise, a candidate $x_i =$ `false` is a potential candidate if $\gamma_i^- \geq \alpha \cdot \gamma^*$. If no cardinality restriction is applied, all potential candidates are included in the RCL.

Cardinality restriction limits the size of the RCL to at most `maxrcl` elements. Two schemes for qualifying potential candidates are obvious to implement. In the first scheme, the best (at most `maxrcl`) potential candidates (as ranked by the greedy function) are selected. Another scheme is to choose the first (at most `maxrcl`) candidates in the order they qualify as potential candidates. The order in which candidates are tested can determine the RCL if this second scheme is used. Many ordering schemes can be used. We suggest two orderings. In the first, one examines the least indexed candidates first and proceeds examining candidates with indices in increasing order. In the other, one begins examining the candidate with the smallest index that is greater than the index of the last candidate to be examined during the previous construction phase iteration.

Once the RCL is set up, a candidate from the list must be selected and made part of the solution being constructed. `SelectIndex` selects, at random, the index s from the RCL. In `AssignVariable`, the assignment is made, i.e. $x_s =$ `true` if $s > 0$ or $x_s =$ `false` if $s < 0$.

The greedy function is changed in `AdaptGreedyFunction` to reflect the assignment made in `AssignVariable`. In this procedure, let L_j denote the indices of the literals in clause $j \in M$. The procedure updates some of the sets Γ_i^+, Γ_i^-, as well as the values γ_i^+ and γ_i^-. There are two cases, as described in Figure 2.3. If the variable just assigned was set to `true` then Γ^+, Γ^-, γ^+ and γ^- are updated in lines 5, 8, 12, and 13. If the variable just assigned was set to `false` then Γ^+, Γ^-, γ^+ and γ^- are updated in lines 19, 22, 26, and 27.

2.2. Local search phase. The GRASP construction phase described in Subsection 2.1 computes a feasible truth assignment that is not necessarily locally optimal

with respect some neighborhood structure. Consequently, local search can be applied with the objective of finding a locally optimal solution that may be better than the constructed solution. In fact, the main purpose of the construction phase is to produce a good initial solution for the local search. It is empirically known that simple local search techniques perform better if they start with a good initial solution. This will be illustrated in the computational results section, where experiments indicate that local search applied to a solution generated by the construction phase, rather than random generation, produces better overall solutions, and GRASP converges faster to an approximate solution.

To define the local search procedure, some preliminary definitions have to be made. Given a truth assignment $x \in \{0,1\}^n$, define the *1-flip neighborhood* $N(x)$ to be the set of all vectors $y \in \{0,1\}^n$ such that $\| x - y \|_2 = 1$. If x is interpreted as a vertex of the n-dimensional unit hypercube, then its neighborhood consists of the n vertices adjacent to x. If we denote by $C(x)$ the total weight of the clauses satisfied by the truth assignment x, then the truth assignment x is a *local maximum* if and only if $C(x) \geq C(y)$, for all $y \in N(x)$. Starting with a truth assignment x, the local search finds the local maximum y in $N(x)$. If $y \neq x$, it sets $x = y$. This process is repeated until no further improvement is possible.

Note that a straightforward implementation of the local search procedure described above would require n function evaluations to compute a local maximum, where for a given assignment, each function evaluation computes the total sum of the weights of the satisfied clauses. Moreover, this process is repeated until the local maximum is the initial solution itself, a process that could result in an exponential number of computational steps [15, 17]. We can, however, exploit the structure of the neighborhood to reduce the computational effort.

Given an initial solution x, define G_i to be the gain in total weight resulting from flipping variable x_i in x, for all i. Let $G_k = \max\{G_i \mid i \in N\}$. If $G_k = 0$ then x is the local maximum and local search ends. Otherwise, the truth assignment resulting from flipping x_k in x, is a local maximum, and hence we only need to update the G_i values such that the variable x_i occurs in a clause in which variable x_k occurs (since the remaining G_i values do not change in the new truth assignment). Upon updating the G_i values we repeat the process, until $G_k = 0$ where the local search procedure is terminated. The procedure is described in the pseudo-code in Figure 2.4. Given a truth assignment x and an index k that corresponds to the variable x_k that is flipped, procedure `GenerateGains` is used to update the G_i values returned in an array G. Note that, in line 2, we pass $k = 0$ to the procedure, since initially all the G_i values must be generated (by convention variable x_0 occurs in all clauses). In lines 4 through 7, the procedure finds a local maximum. The value of the local maximum is saved in line 8.

3. Experimental results. In this section, we report on experimental testing of the GRASP described in the previous sections. The objective of this experiment is to verify that the GRASP using RCL parameter $\alpha = 0.5$ produces truth assignments of high quality and is computationally superior to both the pure greedy construction ($\alpha = 0$) followed by local search and pure random construction ($\alpha = 1$) followed by local search.

The test problems [1] in this experiment are derived from the SAT instance class `jnh` of the 2nd DIMACS Implementation Challenge [16]. These instances all have 100

[1] The test problems can be found at `http://www.research.att.com/~mgcr/data/maxsat.tar.gz`.

```
12 250 MHZ IP19 Processors
CPU: MIPS R4400 Processor Chip Revision:  6.0
FPU: MIPS R4010 Floating Point Chip Revision:  0.0
Data cache size:  16 Kbytes
Instruction cache size:  16 Kbytes
Secondary unified instruction/data cache size:  4
Mbytes
Main memory size:  2560 Mbytes, 2-way interleaved
```

FIG. 3.1. *Hardware configuration (partial output of system command* `hinv`*)*

variables and from 800 to 900 clauses. Some of the instances in this collection are satisfiable, while others are not. Clause weights are integers uniformly distributed between 1 and 1,000. The portable pseudo-random number generator of Schrage [21] was used to generate the weights with the initial seed 270001.

This experiment was limited to a subset of the `jnh` test set. For each instance we ran 10,000 iterations of pure greedy, pure random, and GRASP construction (using RCL parameter $\alpha = 0.5$), all followed by the local search described in Subsection 2.2. No cardinality restriction is used to form the RCL. Furthermore, on a subset of these problem, the GRASP was run for 100,000 iterations to show that quasi-optimal solutions can be found if the number of iterations is allowed to increase. To enable us to observe how close to optimality the GRASP solutions are, we ran the CPLEX `mip` mixed integer programming solver (version 3.0) on the entire set of test problems. We used the mixed integer linear programming formulation of MAX-SAT given in Section 1. On those runs, the CPLEX parameters were set to default values. CPLEX found the optimal solution to all problems in the test set.

The experiment was conducted on a Silicon Graphics Challenge computer (250 MHz MIPS M4400 processor), whose hardware configuration is summarized in Figure 3.1. The code was compiled on the SGI Fortran compiler `f77` using compiler flags `-O2 -Olimit 800 -static`. Processes were limited to a single processor. CPU times in seconds were computed by calling the system routine `etime()`. Reported CPU times exclude problem input time, which is negligible for these test problems.

The experiment is summarized in four tables. Table 3.1 lists the objective function values for the three heuristics (pure random, GRASP, and greedy) after 10,000 iterations. For each instance, this table also lists the total sum of weights and the optimal objective function value obtained by CPLEX. Table 3.3 summarizes computation times for the three heuristics on the three problem classes (800 clauses, 850 clauses, and 900 clauses). Each entry in this table is the average time to run 10,000 iterations of the corresponding heuristic on the corresponding problem class. Table 3.2 lists, for every instance and heuristic, running time and number of iterations taken to find the best solution found by that heuristic when the number of iterations is limited to 10,000. Table 3.4 lists the optimal values obtained by the GRASP if the number of iterations is allowed to go to 100,000, compares that value with the solution found after 10,000 and the optimal. The relative error of the 100,000 iteration solution with the optimal, in percentage, is computed.

On the 10,000 iteration runs, GRASP found a better solution than the other two heuristics in 27 of the 44 instances. The pure random heuristic found a better solution in 13 instance, while in four GRASP and the pure random found an equal valued solution. Both GRASP and pure random found optimal solutions in 3 of the

44 instances, two of which were for the same instances. The pure greedy found the worst solution on all instances. Overall, after 10,000 iterations, GRASP produced the best solution, followed by pure random.

When the number of iterations was fixed to 10,000, the pure random variant was able to produce better solutions than GRASP on some instances. However, it did so at the expense of longer computation times. This is so, because of two factors. First, the number of iterations required by the pure random heuristic to converge to its best solution is larger than that of GRASP. Furthermore, in each pure random iteration, local search requires twice as much CPU time than the local search of GRASP. This is so because the solution quality of the GRASP constructed solution is much superior to that of the pure random construction.

At the other extreme, the pure greedy heuristic was the fastest. However, as discussed earlier it produces the worst solutions. It is faster because its constructed solutions are close to low valued local maxima to which local search converges. The small amount of diversity in the greedily constructed solutions restricts the amount of the solution space explored.

Allowing the number of GRASP iterations to increase (to 100,000) in general improves the value of the best solution found. For the ten instances in Table 3.4, the solutions improved in 8 cases (in one of the instances where it did not improve GRASP had produced the optimal before 10,000 iterations). With respect to the optimal solution (found by CPLEX), the relative error ((`optimal` − `grasp`)/`optimal`) of the GRASP solution was always less than 0.08%.

4. Concluding remarks and discussion. In this paper, the GRASP for satisfiability of Resende and Feo [20] has been extended to solve instances of weighted maximum satisfiability problem. A new local search algorithm was introduced, substantially reducing the computational effort of this implementation with respect to the one in [20].

Computational results indicated the superiority of the GRASP heuristic (with RCL parameter $\alpha = 0.5$) to both the pure random heuristic (GRASP with RCL parameter $\alpha = 0$) and the pure greedy heuristic (GRASP with RCL parameter $\alpha = 1$). With 100,000 iterations, GRASP never produced approximate solutions more than 0.08% from the optimal.

GRASP can be easily parallelized [7, 18, 19]. Because of the independent sampling nature of GRASP, parallelization of this heuristic achieves average linear speedup.

A surprising observation in this computational study was the fact that the CPLEX `mips` solver (a general purpose integer programming solver) was able to find the optimal solutions within 30 minutes of CPU time for each instance. We also observed that other SAT instances from the 2nd DIMACS Implementation Challenge test set can be solved by applying CPLEX `mips` to the corresponding weighted MAX-SAT problem with appropriately chosen weights.

The key to using CPLEX to solve instances of SAT is the choice of weights. By using unit weights, CPLEX is unable to solve any of the SAT instances that we considered. On the other hand, by randomly generating weights uniformly in the interval $[1, U]$, where U depends on the number of clauses in the SAT instance, CPLEX was successful. The ratio U/m must be high enough to make it unlikely that any two clauses have the same weight.

[1] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof verification and hardness of approximation problems*, in Proceedings of the 33rd Annual Symposium on Foundations of Computer Science, 1992, pp. 14–23.

[2] M. Bellare, O. Goldreich, and M. Sudan, *Free bits, PCP and non-approximability – Towards tight results*, Unpublished manuscript, (1995).

[3] S. Cook, *The complexity of theorem-proving procedures*, in Proceedings of the Third annual ACM Symposium on Theory of Computing, 1971, pp. 151–158.

[4] U. Feige and M. Goemans, *Approximating the value of two proper proof systems, with applications to MAX-2SAT and MAX-DICUT*, in Proceeding of the Third Israel Symposium on Theory of Computing and Systems, 1995, pp. 182–189.

[5] T. Feo and M. Resende, *A probabilistic heuristic for a computationally difficult set covering problem*, Operations Research Letters, 8 (1989), pp. 67–71.

[6] T. Feo and M. Resende, *Greedy randomized adaptive search procedures*, Journal of Global Optimization, 6 (1995), pp. 109–133.

[7] T. Feo, M. Resende, and S. Smith, *A greedy randomized adaptive search procedure for maximum independent set*, Operations Research, 42 (1994), pp. 860–878.

[8] M. Garey and D. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, W.H. Freeman and Company, New York, 1979.

[9] M. Goemans and D. Williamson, *A new $\frac{3}{4}$ approximation algorithm for the maximum satisfiability problem*, SIAM Journal on Discrete Mathematics, 7 (1994), pp. 656–666.

[10] ———, *Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming*, Journal of Association for Computing Machinery, 42 (1995), pp. 1115–1145.

[11] J. Gu, *Parallel algorithms for satisfiability (SAT) problem*, in Parallel Processing of Discrete Optimization Problems, P. Pardalos, M. Resende, and K. Ramakrishnan, eds., vol. 22 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1995, pp. 105–161.

[12] J. Gu, P. Purdom, J. Franco, and B. Wah, *Algorithms for the Satisfiability (SAT) Problem: a survey*, in Satisfiability Problem: Theory and Applications, D.-Z. Du, J. Gu, and P. Pardalos, eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1996.

[13] P. Hansen and B. Jaumard, *Algorithms for the maximum satisfiability problem*, Computing, 44 (1990), pp. 279–303.

[14] D. Johnson, *Approximation algorithms for combinatorial problems*, Journal of Computer and System Sciences, 9 (1974), pp. 256–278.

[15] D. Johnson, C. Papadimitriou, and M. Yannakakis, *How easy is local search?*, Journal of Computer and System Sciences, 37 (1988), pp. 79–100.

[16] D. Johnson and M. Trick, eds., *Cliques, coloring, and Satisfiability: Second DIMACS Implementation Challenge*, vol. 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1996.

[17] M. Krentel, *The complexity of optimization problems*, Journal of Computer and System Sciences, 36 (1988).

[18] P. Pardalos, L. Pitsoulis, , T. Mavridou, and M. Resende, *Parallel search for combinatorial optimization: Genetic algorithms, simulated annealing and GRASP*, Lecture Notes in Computer Science, 980 (1995), pp. 317–331.

[19] P. Pardalos, L. Pitsoulis, and M. Resende, *A parallel GRASP implementation for the quadratic assignment problem*, in Parallel Algorithms for Irregularly Structured Problems – Irregular'94, A. Ferreira and J. Rolim, eds., Kluwer Academic Publishers, 1995, pp. 111–130.

[20] M. G. C. Resende and T. A. Feo, *A GRASP for satisfiability*, in The Second DIMACS Implementation Challenge, M. Trick and D. Johnson, eds., vol. 26 of DIMACS Series on Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1996, pp. 499–520.

[21] L. Schrage, *A more portable Fortran random number generator*, ACM Transactions on Mathematical Software, 5 (1979), pp. 132–138.

[22] M. Yannakakis, *On the approximation of maximum Satisfiability*, in Proceedings of the Third ACM-SIAM Symposium on Discrete Algorithms, 1992, pp. 1–9.

TABLE 3.1
Runs for the `jnh` *problem class*

Name	$\sum w_j$	random ($\alpha = 0$)	GRASP ($\alpha = 0.5$)	greedy ($\alpha = 1$)	optimal
jnh1	420925	420632	420737	420207	420925
jnh4		420620	420615	417864	420830
jnh5		420488	420488	418708	420742
jnh6		420759	420815	419307	420826
jnh7		420925	420925	419255	420925
jnh8		420114	419885	418534	420463
jnh9		420314	420078	418629	420592
jnh10		420463	420565	419496	420840
jnh11		420398	420642	419397	420753
jnh12		420518	420737	418561	420925
jnh13		420783	420533	419977	420816
jnh14		420592	420510	419595	420824
jnh15		420429	420360	418666	420719
jnh16		420809	420851	418758	420919
jnh17		420712	420807	420119	420925
jnh18		420389	420372	418388	420795
jnh19		420307	420323	419383	420759
jnh201	394238	394238	394238	393581	394238
jnh202		393823	393983	392947	394170
jnh203		393881	393889	392915	394199
jnh205		394063	394224	393020	394238
jnh207		394100	394101	392735	394238
jnh208		393873	393987	392531	394159
jnh209		394003	394031	392627	394238
jnh210		393942	394238	393701	394238
jnh211		393529	393739	392536	393979
jnh212		394011	394043	393114	394238
jnh214		393737	393701	392828	394163
jnh215		393818	393858	393023	394150
jnh216		394042	394029	393229	394226
jnh217		394232	394232	393832	394238
jnh218		394009	394099	391490	394238
jnh219		393792	393720	392905	394156
jnh220		393951	394053	393767	394238
jnh301	444854	444612	444670	443713	444854
jnh302		443906	444248	442254	444459
jnh303		444063	444244	442704	444503
jnh304		444310	444214	442806	444533
jnh305		444112	443503	442065	444112
jnh306		444603	444658	443435	444838
jnh307		443836	444159	443544	444314
jnh308		444215	444222	441878	444724
jnh309		444273	444349	441959	444578
jnh310		444010	444282	442092	444391

TABLE 3.2
Time and number of iterations to find best solution

	random ($\alpha = 0$)		GRASP ($\alpha = 0.5$)		greedy ($\alpha = 1$)	
name	time	itr	time	itr	time	itr
jnh1	1525.2	9692	192.1	2538	29.3	1089
jnh4	207.8	1412	467.9	5894	0.6	16
jnh5	972.5	6078	30.9	301	59.0	1187
jnh6	1431.8	9290	504.2	6041	1.3	24
jnh7	174.2	1123	188.1	2240	5.7	241
jnh8	791.4	5038	546.7	5985	9.1	139
jnh9	571.1	3923	41.2	526	33.6	788
jnh10	964.4	5925	591.1	6595	0.4	7
jnh11	1330.1	8870	757.7	9313	1.5	31
jnh12	564.1	3638	679.2	8456	1.3	22
jnh13	567.0	3595	12.9	146	0.5	6
jnh14	982.6	6519	197.7	2466	2.0	34
jnh15	899.2	5734	424.6	4991	8.9	132
jnh16	18.7	121	392.8	4876	0.2	4
jnh17	1062.8	6773	448.0	5493	21.7	417
jnh18	560.5	3587	142.9	1650	1.0	21
jnh19	287.3	1928	611.3	7854	0.8	12
jnh201	1099.6	7862	604.3	9473	19.2	436
jnh202	261.2	1886	348.7	4875	0.4	13
jnh203	1352.3	9350	265.3	3372	63.1	779
jnh205	1163.5	8232	227.6	3096	0.4	11
jnh207	41.8	305	460.8	6609	17.9	353
jnh208	890.1	6220	335.1	4307	6.0	103
jnh209	1740.0	9539	170.1	2397	0.1	2
jnh210	219.3	1568	130.7	1852	0.8	18
jnh211	281.8	1929	270.0	3431	26.4	423
jnh212	171.5	1266	244.1	3459	36.3	710
jnh214	1272.2	9105	486.4	6515	74.6	1189
jnh215	350.8	2584	601.8	8131	5.7	113
jnh216	1215.7	8463	441.1	5465	9.5	167
jnh217	438.8	3266	125.4	1855	19.4	276
jnh218	825.2	5950	155.7	2207	0.1	2
jnh219	1308.3	9223	502.8	6586	7.4	120
jnh220	1055.1	7935	513.5	7558	3.2	90
jnh301	347.6	1944	522.1	5323	101.2	2077
jnh302	46.6	270	493.9	5434	12.9	288
jnh303	1046.7	6040	416.1	4265	0.1	2
jnh304	142.5	850	96.7	1054	12.3	144
jnh305	1465.8	8258	424.9	4357	0.6	11
jnh306	1003.0	6062	567.8	6757	1.7	27
jnh307	972.3	5684	353.5	3706	1.5	28
jnh308	607.9	3696	50.2	531	0.1	1
jnh309	564.6	3275	86.8	930	0.9	20
jnh310	1290.4	7646	44.7	465	2.2	48

TABLE 3.3
Average time for 10,000 iterations

Clauses	random ($\alpha = 0$)	GRASP ($\alpha = 0.5$)	greedy ($\alpha = 1$)
800	1402.7s	692.9s	511.8s
850	1461.2s	799.8s	506.5s
900	1712.1s	937.8s	563.4s

TABLE 3.4
Solutions found with 10,000 and 100,000 GRASP iterations

name	10K itr	100K itr	optimal	rel. error
jnh1	420737	420848	420925	0.02%
jnh10	420565	420581	420840	0.06%
jnh11	420642	420642	420753	0.03%
jnh12	420737	420871	420925	0.01%
jnh201	394238	394238	394238	0.00%
jnh202	393983	394029	394170	0.04%
jnh212	394043	394188	394238	0.01%
jnh304	444214	444533	444533	0.00%
jnh305	443503	443744	444112	0.08%
jnh306	444658	444775	444838	0.01%