A Parallel GRASP for MAX-SAT Problems ⋆

P. M. Pardalos[1], L.Pitsoulis[1], and M.G.C. Resende[2]

[1] Center for Applied Optimization, Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL 32611-6595 USA
[2] AT&T Research, Florham Park, NJ 07932 USA

Abstract. The weighted maximum satisfiability (MAX-SAT) problem is central in mathematical logic, computing theory, and many industrial applications. In this paper, we present a parallel greedy randomized adaptive search procedure (GRASP) for solving MAX-SAT problems. Experimental results indicate that almost linear speedup is achieved.
Key words: Maximum Satisfiability, Parallel Search, Heuristics, GRASP, Parallel Computing.

1 Introduction and Problem Definition

The Satisfiability Problem (SAT) is a central problem in artificial intelligence, mathematical logic, computer vision, VLSI design, databases, automated reasoning, computer-aided design and manufacturing. In addition, SAT is a core problem in computational complexity, and it was the first problem shown to be NP-complete [2]. Since most known NP-complete problems have natural reductions to SAT [5], the study of efficient (sequential and parallel) exact algorithms and heuristics for SAT can lead to general approaches for solving combinatorial optimization problems.

In SAT problems we seek to find an assignment of the variables that satisfy a logic formula or the maximum number of clauses in the formula. More specifically, let $x_1, x_2, \ldots, x_n$ denote n Boolean variables, which can take on only the values `true` or `false` (1 or 0). Define clause i (for $i = 1, \ldots, n$) to be

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

where the literals $l_{ij} \in \{x_i, \bar{x}_i \mid i = 1, \ldots, n\}$. In addition, for each clause $\mathcal{C}_i$, there is an associated nonnegative weight w_i. In the weighted *Maximum Satisfiability Problem* (MAX-SAT), one has to determine the assignment of truth values to the n variables that maximizes the sum of the weights of the satisfied clauses. 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.

⋆ Invited paper, PARA96 – Workshop on Applied Parallel Computing in Industrial Problems and Optimization, Lyngby, Denmark (August 18–21, 1996)

We can easily transform a SAT problem from the space of `true-false` variables into an optimization problem of discrete 0-1 variables. 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. Then, the weighted MAX-SAT has the following mixed integer linear programming formulation:

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

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 $\mathcal{C}_i$.

Active research during the past decades has produced a variety of exact algorithms and heuristics for SAT problems [7]. Many of these algorithms have been implemented and tested on parallel computers. Efficient parallel implementations can significantly increase the size of the problems that can be solved. For recent advances on parallel processing of discrete optimization problems see the survey article [1] and the books [4, 11].

One successful heuristic for solving large SAT and MAX-SAT problems is GRASP [12, 13]. A greedy randomized adaptive search procedure (GRASP) is a randomized heuristic for combinatorial optimization [3]. In this paper, we describe a parallel 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

```
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. 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. GRASP construction phase pseudo-code

search to attempt to improve each constructed solution. 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.

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 1 shows a GRASP in pseudo-code. Lines 1 and 2 of the pseudo-code input the problem instance and initialize

```
procedure AdaptGreedyFunction(s)
1     if s > 0 →
2         for j ∈ Γ_s^+ →
3             for k ∈ L_j (k ≠ j) →
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 ≠ j) →
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. 3. AdaptGreedyFunction pseudo-code

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.

Next, we describe the GRASP (based on [12, 13]) for the weighted MAX-SAT. To accomplish this, we outline 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.

```
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. 4. The local search procedure in pseudo-code

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 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 $N = \{1, 2, \dots, n\}$ and $M = \{1, 2, \dots, m\}$ 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.

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. 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`. This requires that some of the sets Γ_i^+, Γ_i^-, as well as the γ_i^+ and γ_i^-, be updated. There are two cases, as described in Figure 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

In general, most heuristics for combinatorial optimization problems terminate at a solution which may not be locally optimal. 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. 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, if x is interpreted as a vertex of the n-dimensional unit

```
procedure ParallelGRASP(n,dat)
1       GenerateRandomNumberSeeds(s_1,...,s_N);
2       do i = 1,...,N →
3           GRASP(n,dat,s_i,val_i,x_i);
4       od;
5       FindBestSolutions(x_i,val_i,max,x*);
6       return(max,x*);
end ParallelGRASP(n,dat);
```

Fig. 5. Parallel GRASP for MAX-SAT

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.

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 same process, until $G_k = 0$ where the local search procedure is terminated. The procedure is described in the pseudo-code in Figure 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 Parallel Implementation

The GRASP heuristic has an inherent parallel nature, which results in an effective parallel implementation [9, 10]. Each GRASP iteration can be regarded as a search in some region of the feasible space, not requiring any information from previous iterations. Therefore, we can perform any number of iterations (searches) in parallel, as long as we make sure that no two searches are performed in the same region. The region upon which GRASP performs a search is chosen randomly, so by using different random number seeds for each iteration performed in parallel we avoid overlapping searches.

Given N processors that operate in parallel, we distribute to each processor its own problem input data, a random number seed, and the GRASP procedure.

Each processor then applies the GRASP procedure to the input data using its random number seed, and when it completes the specified number of iterations it returns the best solution found. The best solution among all N processors is then identified and used as the solution of the problem. It is readily seen that the cost of processor interaction is completely absent since each GRASP procedure operates independently, resulting in no communication between the processors. This in turn, results in an almost linear speedup.

Based on the above discussion, the implementation of GRASP to solve the MAX-SAT in parallel is presented in pseudo code in Figure 5. In line 1 the random numbers seeds $s_1, \ldots, s_N$ are generated for each of the N processors, while in lines 2 to 4 the GRASP procedure is distributed into the processors. Each processor i takes as input the problem size n, the input data `data`, and its random number seed s_i, and returns its best solution found in val_i, with the corresponding truth assignment x_i. In line 5 the best solution among all val_i is found together with the corresponding truth assignment vector x^*.

The code for the GRASP parallel procedure is written in Fortran (`f77`), and was implemented in a Parallel Virtual Machine (PVM) framework [6]. PVM utilizes a network of Unix workstations preferably sharing the same filesystem, by treating each workstation as a different processor enabling parallel execution. The advantages of PVM are its portability, the ease of implementing existing codes, and the fact that even parallel machines in a network could be used. The main disadvantage is that network communication could cause slow or problematic execution since the communication between the processors is based on the network status. But for inherently parallel algorithms with minimal amount of communication required, PVM presents and ideal framework for implementation and testing purposes.

4 Computational Results

Table 1. Solutions for the `jnh` problem class (error-% is expressed in units of $\times 10^{-3}$).

		1-proc.		5-proc.		10-proc.		15-proc.	
name	optimal	soln	error-%	soln	error-%	soln	error-%	soln	error-%
jnh201	394238	394154	0.21	394238	0.00	394171	0.17	394238	0.00
jnh202	394170	393680	1.24	393708	1.17	393706	1.17	393883	0.72
jnh203	393881	393446	1.10	393289	1.50	393695	0.47	393695	0.47
jnh205	394063	393890	0.43	393958	0.26	394060	0.07	394060	0.07
jnh207	394238	394030	0.52	393929	0.78	393813	1.07	394090	0.37
jnh208	394159	393893	0.67	393585	1.45	393622	1.36	393483	1.71
jnh209	394238	393959	0.70	393805	1.10	393884	0.89	393985	0.64
jnh210	394238	393950	0.73	394238	0.00	394238	0.00	394238	0.00
jnh301	444854	444403	1.01	444577	0.62	444577	0.62	444577	0.62
jnh302	444459	443555	2.03	443911	1.23	443911	1.23	443911	1.23

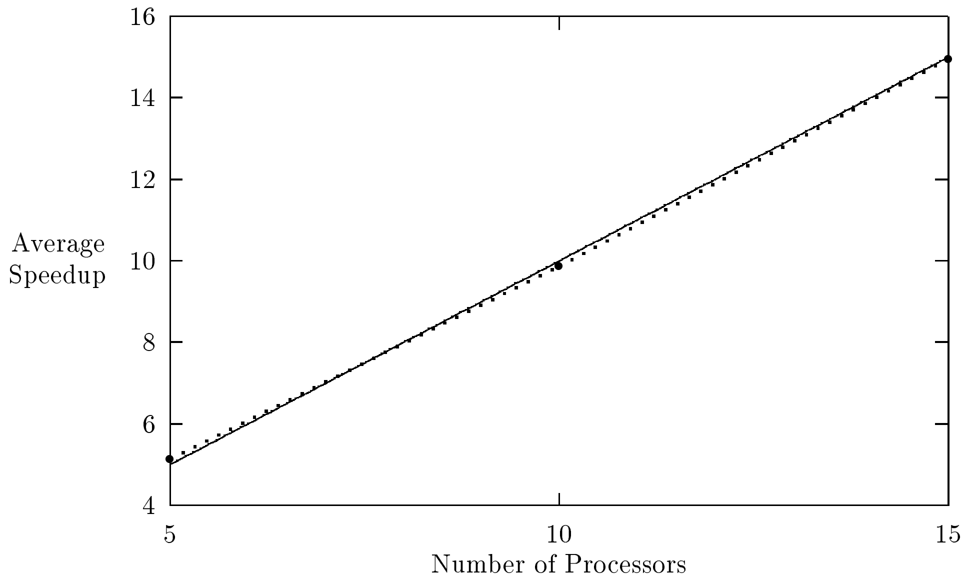

Fig. 6. Average Speedup

Table 2. CPU time (in seconds) and speedup for the `jnh` problem class

	1-proc.	5-proc.		10-proc.		15-proc.	
name	time	time	speedup	time	speedup	time	speedup
jnh201	310.4	62.8	4.9	30.5	10.2	22.2	14.0
jnh202	312.2	59.8	5.2	31.2	10.0	23.4	13.3
jnh203	351.2	72.3	4.9	35.2	10.0	23.2	15.1
jnh205	327.8	63.4	5.2	32.1	10.2	22.5	14.6
jnh207	304.7	56.7	5.4	29.6	10.3	19.8	15.4
jnh208	355.2	65.6	5.4	33.2	10.7	21.0	16.9
jnh209	339.0	60.5	5.6	33.6	10.1	21.6	15.7
jnh210	318.5	57.6	5.5	32.5	9.8	20.8	15.3
jnh301	414.5	85.3	4.9	45.2	9.2	28.3	14.6
jnh302	398.7	88.6	4.5	48.2	8.3	27.0	14.7

In this section, computational experience regarding the parallel implementation of GRASP for solving MAX-SAT instances is reported. The purpose of this experiment is not to demonstrate the overall performance of GRASP for solving MAX-SAT instances, which is reported in [13], but rather to show the efficiency of the parallel implementation of the heuristic in terms of speedup and solution quality. A sample of ten test problems was used for calculating the average speedups of the parallel implementation, which were derived from the SAT instance class `jnh` of the 2nd DIMACS Implementation Challenge [8]. These

problems were converted to MAX-SAT problems by randomly assigning clause weights between 1 and 1000, while their size ranges from 800 to 900 clauses. Furthermore the optimal solution for each instance is known from [13].

The parallel implementation was executed on 15 SUN-SPARC 10 workstations, sharing the same file system, and communication was performed using PVM calls. For each instance we run GRASP in 1, 5, 10 and 15 processors, with maximum number of iterations 1000, 200, 100 and 66 respectively. The amount of CPU time required to perform the specified number of iterations, and the best solution found was recorded.

The computational results are shown in Tables 1 and 2. In Table 1 we can see that that the parallel GRASP with 15 processors always produces better solution than the the serial (1 processor) except in one case. On the average the solutions obtained from the 1, 5, 10 and 15 processors are $0.864 \times 10^{-3}, 0.811 \times 10^{-3}, 0.705 \times 10^{-3}$ and 0.58×10^{-3} percent from the optimal solutions. The solution quality increases on the average, as the number of available processors increases. Moreover, in Table 2 we can see clearly that the speedup of the parallel implementation is almost linear, as illustrated in figure 6 where the average speedup for 5, 10 and 15 processors is shown.

References

1. G.Y. Ananth, V. Kumar, and P.M. Pardalos. Parallel processing of discrete optimization problems. In *Encyclopedia of Microcomputers*, volume 13, pages 129–147. Marcel Dekker Inc., New York, 1993.
2. S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.
3. T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.
4. A. Ferreira and P.M. Pardalos, editors. *Solving Combinatorial Optimization Problems in Parallel: Methods and Techniques*, volume 1054 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
5. M.R. Garey and D.S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Company, New York, 1979.
6. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine A Users Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Press, Massachusetts Institute of Technology, 1994.
7. J. Gu. Parallel algorithms for satisfiability (SAT) problems. In P.M. Pardalos, M.G.C. Resende, and K.G. Ramakrishnan, editors, *Parallel Processing of Discrete Optimization Problems*, volume 22 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 105–161. American Mathematical Society, 1995.
8. D.S. Johnson and M.A. Trick, editors. *Cliques, coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

9. P.M. Pardalos, L. Pitsoulis, T. Mavridou, and M.G.C. Resende. Parallel search for combinatorial optimization: Genetic algorithms, simulated annealing and GRASP. *Lecture Notes in Computer Science*, 980:317–331, 1995.
10. P.M. Pardalos, L.S. Pitsoulis, and M.G.C. Resende. A parallel GRASP implementation for the quadratic assignment problem. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems – Irregular'94*, pages 111–130. Kluwer Academic Publishers, 1995.
11. P.M. Pardalos, M.G.C. Resende, and K.G. Ramakrishnan, editors. *Parallel Processing of Discrete Optimization Problems*, volume 22 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1995.
12. M. G. C. Resende and T. A. Feo. A GRASP for satisfiability. In M.A. Trick and D.S. Johnson, editors, *The Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 499–520. American Mathematical Society, 1996.
13. M.G.C. Resende, L. Pitsoulis, and P.M. Pardalos. Approximate solution of weighted MAX-SAT problems using GRASP. In DingZu Du, Jun Gu, and Panos M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996.

This article was processed using the LaTeX macro package with LLNCS style