2006 and 2007 Max-SAT evaluations: contributed instances

Federico Heras, Javier Larrosa, Simon de Givry, Thomas Schiex

Journal on Satisfiability, Boolean Modeling and Computation 4 (2008) 239-250

2006 and 2007 Max-SAT Evaluations: Contributed Instances

Federico Heras fheras@lsi.upc.edu
Javier Larrosa larrosa@lsi.upc.edu
UPC, Barcelona, Spain

Simon de Givry degivry@toulouse.inra.fr
Thomas Schiex Thomas.Schiex@toulouse.inra.fr
INRA, Toulouse, France

Abstract

In this technical report we briefly describe the instances submitted to the 2006 and 2007 Max-SAT Evaluations. First, we introduce the instances that can be directly encoded as Max-SAT. Then, we describe the methods used to translate problem instances coming from other optimization frameworks to Max-SAT. Finally, we present a quick reference table containing short descriptions of each set of problem instances.

KEYWORDS: *Max-SAT problem instances*

Submitted September 2007; revised January 2008; published June 2008

1. Introduction

Max-SAT is the optimization version of SAT where the goal is to satisfy the maximum number of clauses. It is considered as one of the fundamental combinatorial optimization problems because many important problems can be expressed as Max-SAT.

The *First (2006) and Second (2007) Max-SAT Evaluations* were co-located events of the *Ninth and Tenth International Conferences on Theory and Applications of the Satisfiability Testing* (*SAT-2006* and *SAT-2007*), respectively. They were organized by Josep Argelich, Chu Min Li, Felip Manyà and Jordi Planes with the objectives, among others, of identifying successful solving techniques and identifying challenging benchmarks. The purpose of this paper is to describe the instances used in both evaluations. The instances submitted to the 2006 Max-SAT Evaluation were a subset of the instances submitted to the 2007 Max-SAT Evaluation. Hence, in the following we focus on the 2007 instances.

Max-SAT problem instances are expressed using a *propositional logic-like language*. However, there exist more expressive languages such as *weighted constraints* or *pseudo-boolean constraints*. For this reason, some instances in the evaluations were taken from a *Weighted Constraint Satisfaction Problem* (*WCSP*) or *Pseudo-Boolean Optimization* (*PBO*) repositories and reformulated to Max-SAT. In this paper, we explain how to translate automatically instances coming from these frameworks to Max-SAT.

The structure of the paper is as follows: Section 2 provides preliminary definitions on Max-SAT. Section 3 presents the problems that were directly modelled as Max-SAT. Sections 4 and 5 presents problems coming from the *2005 PB Evaluation* [23] and from a

WCSP repository [9] and show the translation mechanism that we used. Finally, section 6 contains a table with a brief description of the benchmarks submitted to the evaluation and presents some concluding remarks.

2. Preliminaries

In the sequel $X = \{x_1, x_2, \ldots, x_n\}$ is a set of boolean variables taking values over the set $\{true, false\}$, which stands for *true* and *false*, respectively. A *literal* is either a variable x_i or its negation $\bar{x}_i$. Given a literal l, its negation $\bar{l}$ is $\bar{x}_i$ if l is x_i and is x_i if l is $\bar{x}_i$. A *clause* C is a disjunction of literals.

A *weighted* clause is a pair (C, w), where C is a clause and w is the cost of its falsification, also called its *weight*. If a problem has clauses that *must* be satisfied, we call such clauses *mandatory* or *hard* and associate with them a special weight $\top$. Non-mandatory clauses are also called *soft*. In practice, $\top$ can be associated to a natural number equal to the sum of all the weights of the soft clauses plus 1.

An *assignment* is an instantiation of a subset of X. If variable x_i is *assigned* to *true*, literal x_i is satisfied and literal $\bar{x}_i$ is falsified. Similarly, if variable x_i is instantiated to *false*, literal $\bar{x}_i$ is satisfied and literal x_i is falsified. An *assignment* is *complete* if it gives values to all the variables in X (otherwise it is partial).

A *weighted formula* in *conjunctive normal form* (WCNF) is a set of weighted clauses. A *model* is a complete assignment that satisfies all mandatory clauses. The *cost of an assignment* is the sum of weights of the clauses that it falsifies. Given a WCNF formula $\mathcal{F}$, *Weighted* Max-SAT is the problem of finding a model of $\mathcal{F}$ of minimum cost. This cost will be called the *optimal cost of* $\mathcal{F}$. Note that if a formula has only mandatory clauses, weighted Max-SAT is equivalent to classical SAT.

The second Max-SAT Evaluation consisted of four categories:

- Unweighted Max-SAT (U): All clauses are soft and have the same weight 1.
- Weighted Max-SAT (W): All clauses are soft and may have different weights.
- Unweighted Partial Max-SAT (P): There is a set of mandatory clauses and a set of soft clauses. All soft clauses have the same weight 1.
- Weighted Partial Max-SAT (WP): There is a set of mandatory clauses and a set of soft clauses. Soft clauses may have different weights.

The Partial Max-SAT terminology was first introduced in [6]. Note that all the instances in the Unweighted Partial and Weighted Partial categories have models (i.e., are satisfiable with respect to hard clauses).

3. Problem instances directly encoded as (Weighted) Max-SAT

In this section we describe those problem instances that can be naturally encoded directly as (Weighted) Max-SAT. Obviosly, it may exist other problems not described in this report that can be directly encoded as Max-SAT such as the *set covering problem* or the *set packing problem*, but we will focus only on the instances submitted to the 2007 Max-SAT Evaluation.

3.1 Random Max-k-SAT

A k-SAT CNF formula is a CNF formula in which all clauses have size k. Random unsatisfiable 2-SAT and 3-SAT formulas were generated with three different generators: *Cnfgen* [31], [26] and [28]. We fixed the number of variables and varied the number of clauses. Instances generated with [31] and [26] may contain repeated clauses, while this does not occur with [28]. The name of each problem instance specifies how many literals per clause and how many variables and clauses per instance. The name of each random Max-k-SAT problem contains the appropriate information in order to know how many literals per clause and how many variables and clauses are included in the instance. Regarding the random weighted Max-k-SAT instances, only the generator [26] is used and a uniform random weight is associated to each clause. All the weights range from 1 to 10.

3.2 Random Partial Max-SAT

These instances were created using a random partial Max-SAT generator [3]. Basically, it generates random Max-k-SAT instances with [26] and then n clauses are declared hard and the rest are declared soft. Note that n is the number of variables. For the unweighted Partial Max-SAT case, all soft clauses have weight 1. Regarding the weighted Partial Max-SAT case, all clauses have a uniform random weight ranging between 1 and 10.

3.3 Max-one

Given a satisfiable CNF formula, *max-one* is the problem of finding a model with a maximum number of variables set to true. This problem can be encoded as Max-SAT by considering the clauses in the original formula as mandatory and adding a weighted unary clause $(x_i, 1)$ for each variable in the formula. We considered the max-one problem over two types of CNF formula. First, over random 3-SAT instances of 150 variables (generated with *Cnfgen* [31]). Second, we selected structured satisfiable SAT instances coming from the 2002 SAT Competition [29] submitted by J.D. Pehoushek. We considered the following sets of instances: 3col80, 3col100, 3col120, 3col140, cnt, dp and ezfact32.

3.4 Max-cut

Given a graph $G = (V, E)$, a *cut* is defined by a subset of vertices $U \subseteq V$. The size of a cut is the number of edges (v_i, v_j) such that $v_i \in U$ and $v_j \in V - U$. The *Max-cut* problem consists on finding a cut of maximum size. It can be encoded as Max-SAT associating one variable x_i to each graph vertex. Value *true* (respectively, *false*) indicates that vertex v_i belongs to U (respectively, to $V - U$). For each edge (v_i, v_j), there are two clauses $x_i \vee x_j, \bar{x}_i \vee \bar{x}_j$. Given a complete assignment, the number of violated clauses is $|E| - S$ where S is the size of the cut associated to the assignment. We considered Max-Cut instances extracted from random graphs (generator [15]) of 60 nodes with varying number of edges.

We also considered Max-Cut instances from the DIMACS graphs [16]. In Figure 1 a detailed description of these graphs is shown. Since solving the Max-CUT problem of the original DIMACS graphs cannot be handled by current solvers, we generated subgraphs from them. Specifically, we removed nodes from the original graphs and their related edges, until the resulting graphs had about 40 nodes. Then, we encoded the Max-CUT problem

Problem	*nodes*	*density*	*Problem*	*nodes*	*density*
*brock*200_1	200	74.54	*MANN_a*27	378	99.01
*brock*200_2	200	49.63	*MANN_a*45	1035	99.63
*brock*200_3	200	60.54	*MANN_a*81	3321	99.88
*brock*200_4	200	65.77	*p_hat*1000 − 1	1000	24.48
*brock*400_1	400	74.84	*p_hat*1000 − 2	1000	49.01
*brock*400_2	400	74.92	*p_hat*1000 − 3	1000	74.42
*brock*400_3	400	74.79	*p_hat*1500 − 1	1500	25.34
*brock*400_4	400	74.89	*p_hat*1500 − 2	1500	50.61
*brock*800_1	800	64.93	*p_hat*1500 − 3	1500	75.36
*brock*800_2	800	65.13	*p_hat*300 − 1	300	24.38
*brock*800_3	800	64.87	*p_hat*300 − 2	300	48.89
*brock*800_4	800	64.97	*p_hat*300 − 3	300	74.45
$c - fat200 - 1$	200	7.71	*p_hat*500 − 1	500	25.31
$c - fat200 - 2$	200	16.26	*p_hat*500 − 2	500	50.46
$c - fat200 - 5$	200	42.58	*p_hat*500 − 3	500	75.19
$c - fat500 - 1$	500	3.57	*p_hat*700 − 1	700	24.93
$c - fat500 - 10$	500	37.38	*p_hat*700 − 2	700	49.76
$c - fat500 - 2$	500	7.33	*p_hat*700 − 3	700	74.80
$c - fat500 - 5$	500	18.59	*san*1000	1000	50.15
*hamming*10 − 2	1024	99.02	*san*200_0.7_1	200	70.00
*hamming*10 − 4	1024	82.89	*san*200_0.7_2	200	70.00
*hamming*6 − 2	64	90.48	*san*200_0.9_1	200	90.00
*hamming*6 − 4	64	34.92	*san*200_0.9_2	200	90.00
*hamming*8 − 2	256	96.86	*san*200_0.9_3	200	90.00
*hamming*8 − 4	256	63.92	*san*400_0.5_1	400	50.00
*johnson*16 − 2 − 4	120	76.47	*san*400_0.7_1	400	70.00
*johnson*32 − 2 − 4	496	87.88	*san*400_0.7_2	400	70.00
*johnson*8 − 2 − 4	28	55.56	*san*400_0.7_3	400	70.00
*johnson*8 − 4 − 4	70	76.81	*san*400_0.9_1	400	90.00
*keller*4	171	64.91	*sanr*200_0.7	200	69.69
*keller*5	776	75.15	*sanr*200_0.9	200	89.76
*keller*6	3361	81.82	*sanr*400_0.5	400	50.11
*MANN_a*9	45	92.73	*sanr*400_0.7	400	70.01

Figure 1. The 66 graphs from the Second DIMACS challenge [16].

of such subgraphs. Let $\alpha = |V|/40$ and let $V = \{v_1, v_2, \ldots, v_i, \ldots, v_n\}$ be the set of vertices of the original graph. For each node v_i of the original graph, we conserved it if $i\%\alpha = 0$, otherwise it was removed.

The weighted Max-CUT version of the random graphs and of the reduced DIMACS graphs are built by associating a uniform random weight to each edge. Such weight ranges from 1 to 10.

As noted in [30], the *spin glass* problem can be reformulated as computing the Max-CUT of a specific graph. Frauke Liers provided us 5 unweighted and 5 weighted spin glass instances and other 20 unweighted instances were created by Han Lin using the *Spin Glass Server* [17]. While all those instances can be solved in zero time with a *branch-and-cut algorithm* [30, 12], they are quite challenging for current Max-SAT solvers [4].

3.5 The Minimum Vertex Covering problem

Given a graph $G = (V, E)$, a *vertex covering* is a set $U \subseteq V$ such that for every edge (v_i, v_j) either $v_i \in U$ or $v_j \in U$. The size of a vertex covering is $|U|$. The *minimum vertex covering* problem is a well-known NP-Hard problem. It consists in finding a covering of minimal size. It can be naturally formulated as (weighted) Max-SAT. We associate one variable x_i to each graph vertex v_i. Value *true* (respectively, *false*) indicates that vertex x_i belongs to U (respectively, to $V - U$). There is a binary weighted clause $(x_i \vee x_j, \top)$ for each edge (v_i, v_j). It specifies that at least one of these two vertices have to be in the covering because there is an edge connecting them. To minimize the number of vertices in the covering, there is a unary clause $(\bar{x}_i, 1)$ for each variable x_i in order to specify that it is preferred not to add vertices to U.

3.6 The Maximum Clique Problem

Given a graph $G = (V, E)$, a *clique* is a set $U \subseteq V$ such that for every vertex $v \in U$, v is connected to all the vertices in U. The size of a clique is $|U|$. The *maximum clique* problem (Max-Clique) consists in finding a clique of maximal size. This problem is deeply related to the Minimum Vertex Covering. As noted in [13], finding the maximum clique of a graph $G = (V, E)$ is equivalent to finding a minimum vertex covering of the complementary graph $\bar{G}$. Given a graph $G = (V, E)$, its complementary graph is noted by $\bar{G} = (V, \bar{E})$. It is constructed with the same set of vertices V and $(v_i, v_j) \in \bar{E}$ iff $(v_i, v_j) \notin E$. Hence, we model Max-Clique problems as Minimum Vertex Covering problems over the complementary graph. Observe that the maximum size of the maximum clique is equivalent to $|V| - S$, where S is the size of the minimum vertex covering.

We considered maximum clique instances extracted from random graphs [15] with 150 nodes and varying number of edges. We also considered the 66 Max-Clique instances from the Second DIMACS challenge [16] (See Table 1 for details).

3.7 Combinatorial Auctions

A *combinatorial auction* is defined by a set of goods G and a set of bidders that bid for indivisible subsets of goods. Each bid i is defined by the subset of requested goods $G_i \subseteq G$ and the amount of money offered. The bid-taker, who wants to maximize its revenue, must decide which bids are to be accepted. Note that if two bids request the same good, they cannot be jointly accepted [25]. In its Max-SAT encoding, there is one variable x_i associated to each bid. There are unit clauses (x_i, u_i) indicating that if bid i is not accepted there is a loss of profit u_i. Besides, for each pair i, j of conflicting bids, there is a mandatory clause $(\bar{x}_i \vee \bar{x}_j, \top)$.

We used the CATS generator [18] that allows to generate random instances inspired from real-world scenarios. In particular, we generated instances from the *Regions*, *Paths* and *Scheduling* distributions. The number of goods was fixed to 60 and we increased the number of bids. By increasing the number of bids, instances become more constrained (namely, there are more conflicting pairs of bids) and harder to solve.

4. Problem instances coming from the 2006 PB Evaluation

A *(linear constrained) pseudo-boolean optimization problem* (*PBO*) [27, 11] has the form:

(1) minimize $\sum_{j=1}^{n} c_j \cdot x_j$

(2) subject to $\sum_{j=1}^{n} a_{ij} l_j \geq b_i, \quad i = 1 \ldots m$

where variables x_j can take values in $\{0, 1\}$, l_j is either x_j or $1 - x_j$, and c_j, a_{ij} and b_i are non-negative integers. (1) is the *objective function* and (2) are the *pseudo-boolean constraints*. A PBO instance can be translated into a Max-SAT formula as follows: each pseudo-boolean constraint is translated into a set of *hard clauses* using MiniSAT+ [11] (the algorithm heuristically decides the most appropriate translation choosing among *adders*, *sorters* or *BDDs*). The objective function is translated into a set of *soft unit clauses*. Each summand $c_j \cdot x_j$ becomes a new soft unit clause $(\bar{x}_j, c_j)$.

The following problem instances were taken from the 2006 PB Evaluation [22]: *logic synthesis* [34], *misc* (*garden*), *routing* [2], *MPI* (*Minimum-size Prime Implicant)* [24], *MPS* (*miplib*) [1]. They were translated to Max-SAT as specified above.

5. Problem instances coming from a WCSP repository

A WCSP can be roughly defined as a generalization of the Max-SAT problem that contains variables with a finite number of values rather than boolean variables and contains weighted constraints rather than weighted clauses. Formally, a *weighted constraint satisfaction problem* (WCSP) [20] is a tuple $P = (\top, \mathcal{X}, \mathcal{D}, \mathcal{C})$. $\mathcal{X} = \{1, \ldots, n\}$ is a set of *variables*. Each variable $i \in \mathcal{X}$ has a finite domain $D_i \in \mathcal{D}$ of values that can be assigned to it. (i, a) denotes the assignment of value $a \in D_i$ to variable i. A *weighted tuple* is a pair $(t; w)$ where t is an assignment to a subset of variables and w is a *cost* expressed as a natural number. A *weighted constraint* (also called *cost function*) C_k contains all the weighted tuples defined over the same subset of variables. When a weighted tuple has cost $\top$, it means that the assignment indicated by such tuple is forbidden, otherwise the assignment is permitted with the corresponding cost. $\mathcal{C}$ is the set of weighted constraints of the WCSP problem. The *cost of an assignment* is the sum of the costs of the weighted tuples that are a subset of such assignment. The usual task of interest is to *find a complete consistent assignment with minimum cost*, which is NP-hard.

Similarly to the translation of a *CSP* to *SAT* [32], there are two possible ways of encoding a WCSP problem as Max-SAT: the *direct* and the *log* encoding. The first one was shown to maintain nice propagation properties from the original CSP problem [32]. Hence, we translated most of the WCSP instances using the direct encoding.

In what follows, we present how the direct encoding works. We associate a boolean variable x_{ij} with each value j that can be assigned of the WCSP variable x_i. Then, we create new hard clauses to assure that each WCSP variable is given a value. That is, for each WCSP variable x_i with values $\{1, 2, \ldots, d\}$ we add the hard clause $(x_{i1} \vee x_{i2} \vee \cdots \vee x_{id}, \top)$. Furthermore, we add hard clauses that ensure that a WCSP variable is not given more than one value: For each WCSP variable x_i and each pair of different values j, k of variable x_i we create the clause $(\bar{x}_{ij} \vee \bar{x}_{ik}, \top)$. Finally, for each weighted tuple we add a new weighted clause that represents its contribution.

Example 1 *As an example, consider a WCSP defined with variables* $\{x_1, x_2, x_3\}$ *each one with three values* $\{a, b, c\}$*. Let be the constraint* C_1 *defined over variables* $\{x_1, x_2, x_3\}$ *with weighted tuples* $(x_1 = a, x_2 = a, x_3 = a; 1)$*,* $(x_1 = b, x_2 = a, x_3 = c; 3)$ *and* $(x_1 = a, x_2 = b, x_3 = b; \top)$*. We need to create 9 boolean variables:* $x_{1a}, x_{1b}, x_{1c}, x_{2a}, x_{2b}, x_{2c}, x_{3a}, x_{3b}, x_{3c}$*. We add 3 hard clauses that assure each WCSP variable takes a value:* $(x_{1a} \vee x_{1b} \vee x_{1c}, \top)$*,* $(x_{2a} \vee x_{2b} \vee x_{2c}, \top)$ *and* $(x_{3a} \vee x_{3b} \vee x_{3c}, \top)$*. We assure we do not assign more than one value to each WCSP variable* x_i *(with* $i = 1, 2, 3$*):* $(\bar{x}_{ia} \vee \bar{x}_{ib}, \top)$*,* $(\bar{x}_{ia} \vee \bar{x}_{ic}, \top), (\bar{x}_{ib} \vee \bar{x}_{ic}, \top)$*. Finally, we add the contribution of each weighted tuple:* $(\bar{x}_{1a} \vee \bar{x}_{2a} \vee \bar{x}_{3a}, 1)$*,* $(\bar{x}_{1b} \vee \bar{x}_{2a} \vee \bar{x}_{3c}, 3)$ *and* $(\bar{x}_{1a} \vee \bar{x}_{2b} \vee \bar{x}_{3b}, \top)$*.*

Now, we present the log encoding. For each WCSP variable x_i with $|D_i|$ values, we create $x_{i1}, \ldots, x_{im}$ where $m = \lceil log_2(|D_i|) \rceil$ propositional variables. Each of the 2^m combinations represents a possible value of x_i.

Example 2 *Consider a WCSP variable* x_1 *with 3 values* $\{a, b, c\}$*. We need to create* $m = \lceil log_2(|D_i|) \rceil = \lceil log_2(3) \rceil = 2$ *variables to represent each value of variable* x_1*. That is, we create propositional variables* x_{11} *and* x_{12} *that allow 4 combinations. Precisely,* $x_{11} \vee x_{12}$ *represents value a,* $x_{11} \vee \bar{x}_{12}$ *represents value b and* $\bar{x}_{11} \vee x_{12}$ *represents value c. Observe that combination* $\bar{x}_{11} \vee \bar{x}_{12}$ *is not used.*

The log encoding does not require to ensure that each WCSP variable is assigned a value or that it is given only one value. However, it may happen as stated above that the number of values $|D_i|$ of a WCSP variable x_i is not power of 2. In that case, we have to add hard clauses to forbid that non-existing values are assigned.

Example 3 *Consider the WCSP variable* x_1 *with 3 values* $\{a, b, c\}$ *of the previous example. It is clear that a possible fourth value represented by* $\bar{x}_{11} \vee \bar{x}_{12}$ *does not exist because our variable has only 3 values. Hence, we add hard clause* $(\bar{x}_{11} \vee \bar{x}_{12}, \top)$ *to assure that such a value is never assigned.*

Finally, we add the clauses that represent each weighted tuple.

Example 4 *As an example, consider a WCSP defined with variables* $\{x_1, x_2, x_3\}$ *each one with three values* $\{a, b, c\}$*. Let be the constraint* C_1 *defined over variables* $\{x_1, x_2, x_3\}$ *with weighted tuples* $(x_1 = a, x_2 = a, x_3 = a; 1)$*,* $(x_1 = b, x_2 = a, x_3 = c; 3)$ *and* $(x_1 = a, x_2 = b, x_3 = b; \top)$*. We create 6 boolean variables. Variables* x_{11} *and* x_{12} *represent the values of the WCSP variable* x_1*. Variables* x_{21} *and* x_{22} *represent the values of the WCSP variable* x_2*. Variables* x_{31} *and* x_{32} *represent the values of the WCSP variable* x_3*. All three WCSP variables have only 3 values, hence we have to forbid a fourth value is assigned. Hence, we add three hard clauses* $\{(\bar{x}_{11} \vee \bar{x}_{12}, \top), (\bar{x}_{21} \vee \bar{x}_{22}, \top), (\bar{x}_{31} \vee \bar{x}_{32}, \top)\}$*. Finally, we add (large) clauses representing each weighted tuple:* $\{(x_{11} \vee x_{12} \vee x_{21} \vee x_{22} \vee x_{31} \vee x_{32}, 1), (x_{11} \vee \bar{x}_{12} \vee x_{21} \vee x_{22} \vee \bar{x}_{31} \vee x_{32}, 3), (x_{11} \vee x_{12} \vee x_{21} \vee \bar{x}_{22} \vee x_{31} \vee \bar{x}_{32}, \top)\}$*.*

We created *Max-CSP* random instances generated using the protocol specified in [19]. We distinguish 4 different sets of problems: *Dense Loose* (*DL*), *Dense Tight* (*DT*), *Sparse Loose* (*SL*) and *Sparse Tight* (*ST*). Each set contains 10 instances with 3 values and 10

instances with 4 values per variable. Note that a Max-CSP instance can be seen as a WCSP in which all weighted tuples have a cost 1.

We considered structured instances that were taken from a *Weighted Constraint Satisfaction Problem* (*WCSP*) repository [9] including *Weighted Queens* [21], *Planning* [8] and *Satellite Photograph Schedulling* (*SPOT5*) [5].

All the previous instances were translated to Max-SAT using the direct encoding. For the SPOT5 case, we also considered the log encoding.

6. Concluding remarks

In this paper we have overviewed most of the instances submitted to the 2007 Max-SAT Evaluation. Figure 2 briefly describes them. The first column specifies the category, the second column shows the number of instances of each set and the third column gives a short description. The description contains a reference to a generator (if it was needed) and/or to some paper in which the benchmark is described in detail. 2007 Max-SAT Evaluation results and a comparison of all the submitted solvers can be found in [4]. Note that the authors of this technical report submitted almost all the instances. However, some instances were submitted by other authors:

- Random Max-2-SAT (U), random Max-3-SAT (U), random Weighted Max-2-SAT (W), random Weighted Max-3-SAT (W), random unweighted Partial Max-2-SAT (P), random weighted Partial Max-3-SAT (P), random weighted Partial Max-2-SAT (WP), random weighted Partial Max-3-SAT (WP) and the Quasigroup Completion Problem instances were submitted by the organizers.
- Some random Max-3-SAT (U) instances and 20 unweighted spin glass (U) instances were submitted by Han Lin.

We would like to remark that we cannot explain in detail how the *Ramsey Number* instances and the *Quasi Group Completion problem* instances were generated.

Recently, we have collected new problem instances that we plan to submit to future evaluations. They include unsatisfiable DIMACS instances, protein alignment instances, hidden optimum value instances and time tabling instances, to name a few.

Acknowledgements

We would like to thank to the organizers, Han Lin, Frauke Liers and Mutsunori Yagiura. They contributed in some way to create some of the instances described in this paper.

Cat.	N.I.	Short Description
U	330	Random Max-2-SAT. Generators [31, 26, 28].
U	310	Random Max-3-SAT. Generators [31, 26, 28].
U	62	Max-CUT of reduced DIMACS graphs [16].
U	40	Max-CUT of random graphs. Graph generator [15].
U	25	Spin-Glass [30] modelled as Max-CUT.
U	48	Ramsey Number. Generator [33].
W	90	Random Weighted Max-2-SAT. Generator [26].
W	80	Random Weighted Max-3-SAT. Generator [26].
W	62	Weighted Max-CUT of reduced DIMACS graphs [16].
W	40	Weighted Max-CUT of random graphs. Graph generator [15].
W	5	Spin-Glass [30] modelled as Weighted Max-CUT.
W	48	Weighted Ramsey Number. Generator [33].
P	90	Random Partial Max-2-SAT. Generator [3].
P	60	Random Partial Max-3-SAT. Generator [3].
P	96	Max-Clique of random graphs. Graph generator [15].
P	62	Max-Clique of DIMACS graphs [16].
P	80	Max-One of random satisfiable instances. Generator [31].
P	60	Max-One of 2002 SAT Competition satisfiable instances [29].
P	7	Garden instances from 2005 PB Evaluation [23].
P	17	Logic Synthesis instances [34] from 2005 PB Evaluation [23].
P	148	Min. Prime Imp. [24] on DIMACS instances [16] from 2005 PB Evaluation [23].
P	15	Routing instances [2] from 2005 PB Evaluation [23].
P	20	Dense Loose Max-CSP instances. Generator [10].
P	20	Dense Tight Max-CSP instances. Generator [10].
P	20	Sparse Loose Max-CSP instances. Generator [10].
P	20	Sparse Tight Max-CSP instances. Generator [10].
P	7	Weighted Queens instances from WCSP repository [9]. Generator [21].
WP	90	Random Weighted Partial Max-2-SAT. Generator [3].
WP	60	Random Weighted Partial Max-3-SAT. Generator [3].
WP	88	Comb. Auction instances following PATHS distribution. Generator [18].
WP	84	Comb. Auction instances following REGIONS distribution. Generator [18].
WP	84	Comb. Auction instances following SCHEDULING distribution. Generator [18].
WP	16	MIPLIB instances [1] from 2005 PB Evaluation [23].
WP	186	Factor instances from 2005 PB Evaluation [23].
WP	71	Planning instances [7] from WCSP repository [9].
WP	42	SPOT5 instances [5] from WCSP repository [9].
WP	25	Quasigroup Completion Problem instances [14].

Figure 2. Instances submitted to the 2007 Max-SAT Evaluation.

References

[1] Alexander Martin Tobias Achterberg and Thorsten Koch. Miplib - mixed integer problem library. http://miplib.zib.de/, 2005.

[2] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Generic ilp versus specialized 0-1 ilp: an update. In *ICCAD*, pages 450–457, 2002.

[3] Josep Argelich. Random partial max-sat generator. Not available.

[4] Josep Argelich, Chu Min Li, Felip Manyá, and Jordi Planes. The first and second max-sat evaluations. *Journal on Satisfiability, Boolean Modeling and Computation*, this issue.

[5] E. Bensana, M. Lemaitre, and G. Verfaillie. Earth observation satellite management. *Constraints*, **4(3)**:293–299, 1999.

[6] Byungki Cha, Kazuo Iwama, Yahiko Kambayashi, and Shuichi Miyazaki. Local search algorithms for partial MAXSAT. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-97*, pages 263–268, 1997.

[7] M. Cooper. High-order consistency in valued constraint satisfaction. *Constraints*, **10**:283–305, 2005.

[8] M. Cooper, S. Cussat-Blanc, M. de Roquemaurel, and P. Régnier. Soft arc consistency applied to optimal planning. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP-06*, pages 680–684, 2006.

[9] S. de Givry, F. Heras, J. Larrosa, E. Rollon, M. Sanchez, and T. Schiex. The wcsp repository. http://mulcyber.toulouse.inra.fr/plugins/scmcvs/cvsweb.php/benchs/?cvsroot=toolbar, 2003.

[10] Simon de Givry. Random binary max-csp generator, file random_vcsp.c. http://mulcyber.toulouse.inra.fr/plugins/scmcvs/cvsweb.php/benchs/generators/?cvsroot=toolbar, 1995.

[11] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**:1–26, 2006.

[12] Matthias Elf, Carsten Gutwenger, Michael Jünger, and Giovanni Rinaldi. Branch-and-cut algorithms for combinatorial optimization and their implementation in abacus. In *Computational Combinatorial Optimization*, pages 157–222, 2001.

[13] Torsten Fahle. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In *Proceedings of ESA-2002*, pages 485–498, 2002.

[14] Carla Gomes. lsencode: A generator of quasigroup completion problem and related problems. http://www.cs.cornell.edu/gomes/new-demos.htm.

[15] Federico Heras and Jordi Planes. Random graph generator, file random_graph.c. http://mulcyber.toulouse.inra.fr/plugins/scmcvs/cvsweb.php/benchs/generators/?cvsroot=toolbar, 2005.

[16] D. S. Johnson and M. A. Trick. Second dimacs implementation challenge: Finding cliques in graphs, coloring graphs, and solving satisfiability problems. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, http://mat.gsia.cmu.edu/challenge.html, 1994.

[17] Michael Jünger and Frauke Liers. Spin glass server. http://www.informatik.uni-koeln.de/ls_juenger/research/spinglass/.

[18] M. Pearson K. Leyton-Brown and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. *ACM Conference on Electronic Commerce*, pages 66–76, 2000.

[19] J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted CSP. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI-03*, Acapulco, Mexico, August 2003.

[20] Javier Larrosa. Node and arc consistency in weighted CSP. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-02*, pages 48–53, 2002.

[21] Javier Larrosa. Weighted queens generator, file wqueens.c. http://mulcyber.toulouse.inra.fr/plugins/scmcvs/cvsweb.php/benchs/generators/?cvsroot=toolbar, 2004.

[22] Vasco Manquinho and Olivier Roussel. 2006 pseudo-Boolean evaluation. http://www.cril.univ-artois.fr/PB06/, 2006.

[23] Vasco Manquinho and Olivier Roussel. The first evaluation of pseudo-Boolean solvers (pb'05). *Journal on Satisfiability, Boolean Modeling and Computation*, **2**:103–143, 2006.

[24] Clara Pizzuti. Computing prime implicants by integer programming. In *ICTAI*, pages 332–336, 1996.

[25] Tuomas Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI-99*, pages 542–547, 1999.

[26] Bart Selman. Program for generating random instances of sat problems. 1992.

[27] H. M. Sheini and Karem A. Sakallah. Pueblo: A hybrid pseudo-Boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**:165–189, 2006.

[28] Haiou Shen and Hantao Zhang. Study of lower bounds for Max-2-SAT. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-04*, 2004.

[29] Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. 2002 sat competition. http://www.satcompetition.org/2002/, 2002.

[30] Caterina De Simone, Martin Diehl, Michael Jünger, Petra Mutzel, Gerhard Reinelt, and Giovanni Rinaldi. Exact ground states in spin glasses: New experimental results with a branch-and-cut algorithm. *Journal of Statistical Physics*, **80**:487–496, 1995.

[31] A. van Gelder. Cnfgen formula generator. ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances, 1993.

[32] Toby Walsh. SAT v CSP. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP-00*, pages 441–456, 2000.

[33] Mutsunori Yagiura. Ramsey number problem generator. http://www-or.amp.i.kyoto-u.ac.jp/~yagiura/sat/Ramsey/, 1998.

[34] Saeyang Yang. Logic synthesis and optimization benchmarks user guide version 3.0. Microelectronics Center of North Carolina, 1991.