M.R.C. van Dongen,
Christophe Lecoutre, and
Olivier Roussel (Editors)

Proceedings of the Second International CSP Solver Competition[1]

Held in conjunction with the
Twelfth International Conference on
Principles and Practice of
Constraint Programming (CP 2006)

---

[1] Sponsored by the Association for Constraint Programming.

Preface

The Second International CSP Solver Competition was organised to improve our understanding of the sources of solver efficiency, and the options that should be considered in crafting solvers. In particular, issues of interdependence and interaction among features can perhaps only be elucidated by comparing and testing actual implementations. It is hoped that efforts like this will further our understanding of the important dimensions of performance, for example robustness or versatility as opposed to problem-specific efficiency.

These proceedings present short descriptions of some of the solvers. Also they present the results of the competition and an invited paper of work related to the solver competition.

For this second edition we considered instances involving constraints defined in extension and in intension (i.e. by a predicate). Also the global constraint *allDifferent* and the Max-CSP Class have been introduced.

January 2008

Marc van Dongen
Christophe Lecoutre
Olivier Roussel

Main Organising Committee

Organisation

Organising a CSP solver competition requires much of work and is not possible without the help of others. The organisers should like to thank the following people/institutions for turning the 2006 version of the CSP Solving Competition into a successful event:

- The members of the independent jury, Radoslaw Szymanek (EPFL, Lausanne), Djamal Habet (LSIS, Marseille), and Richard Ostrowski (LSIS, Marseille), for selecting the final instances for the competition.
- Lucas Bordeaux (Microsoft) for providing the SMT instances.
- Rick Wallace (4C) for generating his Max-CSP instances.
- CRIL at University of Artois for providing their machines for the final evaluation.
- The Association for Constraint Programming for donating €1000 for organising the event.
- All members of the programme committee.
- The contestants for participating: without them there would not have been a competition.

Programme Committee

Fred Hemery	Université d'Artois, France
Chris Jefferson	University of Oxford
Christophe Lecoutre	Université d'Artois, France
Marc van Dongen	University College Cork, Ireland
Olivier Roussel	Université d'Artois, France
Radoslaw Szymanek	École Polytechnique Federale de Lausanne, Switzerland
Rick Wallace	Cork Constraint Computation Centre, Ireland

Additional Reviewers

Diarmuid Grimes	Cork Constraint Computation Centre, Ireland
Emmanuel Hebrard	Cork Constraint Computation Centre, Ireland
Deepak Mehta	Cork Constraint Computation Centre, Ireland
Sébastien Tabary	Université d'Artois, France
Julien Vion	Université d'Artois, France

Table of Contents

Papers from the Organisers

Main Results . . . 1
M.R.C. van Dongen, Christophe Lecoutre, and Olivier Roussel

Invited Papers

Combining Multiple Constraint Solvers: Results on the CPAI'06 Competition Data . . . 11
Matthew Streeter, Daniel Golovin, and Stephen F. Smith

Papers from the Contestants

toolbar/toulbar2 . . . 19
S. Bouveret et al.

VALCSP . . . 23
Assef Chmeiss, Vincent Krawczyk, and Lakhdar Saïs

Alternative Methods for Learning in CSP Solvers . . . 29
Diarmuid Grimes and Richard J. Wallace

Mistral . . . 35
Emmanuel Hebrard

CSP2SAT4J: A Simple CSP to SAT Translator . . . 43
Daniel Le Berre and Inês Lynce

Abscon 109 *A Generic CSP Solver* . . . 55
Christophe Lecoutre and Sebastien Tabary

Sugar: A CSP to SAT Translator Based on Order Encoding . . . 65
Naoyuki Tamura and Mutsunori Banbara

Introducing $buggy_{2-5}$ and $buggy^s_{2-5}$ . . . 71
M.R.C. van Dongen

CSP4J: a black-box CSP solving API for Java . . . 75
Julien Vion

A Report on the B-Prolog CSP Solver . . . 89
Neng-Fa Zhou

Results of the Second CSP Solver Competition

M.R.C. van Dongen[1], Christophe Lecoutre[2], and Olivier Roussel[2]

[1] University College Cork
[2] Université d'Artois

Abstract. This paper presents the main results of the Second International CSP Solver Competition, which was held in conjunction with the Twelfth International Conference on Principles and Practice on Constraint Programming (CP'2006).

1 Introduction

This paper presents the main results of the Second International CSP Solver Competition, which was started in 2006 and evaluated in 2007. Due to an initial lack of solvers it was decided to run the competition in two phases: a first warmup round in 2006, and a second deciding round in 2007. It was hoped that this should give prospective contestants more time to prepare. In the end, this approach proved successful. The number of solvers doubled from 16, in the first edition of the competition in 2005, to 32 in the 2006/2007 edition. The results presented in this paper are the results of the second deciding round.

Changes

Compared to the first edition of the competition there were four changes. The first change involved the problem specification format. This edition, all problem instances were represented in XML (Extensible Markup Language), using the format XCSP 2.0. The second change involved the introduction of a new problem class. Besides the class Ordinary CSP, which was already considered in the first edition, we also considered Max-CSP. The third change was the introduction of intensional constraints (this change proved to be the most challenging as it presented problems related to overflows, divisions, etc.). The fourth change involved the introduction of the global constraint $allDifferent$.

Teams, Problems, and Categories

Twelve teams, proposing a total of 23 solvers, participated in the class Ordinary CSP. The solvers were run against a suite of 3425 instances, consisting of binary as well as non-binary instances. The total time spent by all solvers amounts to 343 CPU days.

In addition 9 solvers, proposed by 5 teams, were run against a suite of 1069 Max-CSP instances, consisting of binary and non-binary instances. The total time spent by all Max-CSP solvers amounts to 101 CPU days.

Table 1 lists all CSP and Max-CSP solvers which participated in the competition.

Solver	Version	Author(s)
Class: Ordinary CSP		
Abscon	109 ESAC	Christophe Lecoutre and Sebastien Tabary
Abscon	109 AC	Christophe Lecoutre and Sebastien Tabary
BPrologCSPSolver70a	2006-12-13	Neng-Fa Zhou
$buggy_{2-5}$	2007-01-08	Marc van Dongen
$buggy^s_{2-5}$	2007-01-08	Marc van Dongen
CSP4J-Combo	2006-12-19	Julien Vion
CSP4J-MAC	2006-12-19	Julien Vion
CSP4J-MAC	2007-01-16	Julien Vion
CSPtoSAT+minisat	0.3	Olivier Roussel
Diarmuid-rndi	2006-12-21	Diarmuid Grimes
Diarmuid-rndi	2007-01-22	Diarmuid Grimes
Diarmuid-wtdi	2006-12-21	Diarmuid Grimes
galac	1	Gilles Audemard, Assef Chmeiss, and Lakdhar Saïs
galacJ	beta 1	Gilles Audemard, Assef Chmeiss, and Lakdhar Saïs
Mistral	2006-12-04	Emmanuel Hebrard
rjw-solver	2006-12-09	Richard Wallace
rjw-solver	2007-01-21	Richard Wallace
sat4jCSP	1.7 RC BF3	Daniel Le Berre and Ines Lynce
SAT4JCSP-CACHED	1.7 RC BF3	Daniel Le Berre and Ines Lynce
sugar	0.40	Naoyuki Tamura
Tramontane	2006-12-04	Emmanuel Hebrard
VALCSP	3.0	Vincent Krawczyk, Assef Chmeiss, and Lakhdar Saïs
VALCSP	3.1	Vincent Krawczyk, Assef Chmeiss, and Lakhdar Saïs
Class: Max-CSP		
AbsconMax	109 EPFC	Christophe Lecoutre
AbsconMax	109 PFC	Christophe Lecoutre
aolibdvo	2007-01-17	Radu Marinescu
aolibpvo	2007-01-17	Radu Marinescu
CSP4J-MaxCSP	2006-12-19	Julien Vion
Toolbar	2007-01-12	Javier Larrosa, Emma Rollon, Federico Heras, Matthias Zytnicki, Simon de Givry, and Thomas Schiex
$\text{Toolbar}_{\text{BTD}}$	2007-01-12	Simon de Givry, Federico Heras, Gerard Verfaillie, J. Larrosa, M. Zytnicki , Sylvain Bouveret, and Thomas Schiex
$\text{Toolbar}_{\text{MaxSat}}$	2007-01-19	Federico Heras, Emma Rollon, Javier Larrosa, Matthias Zytnicki, Simon de Givry, and Thomas Schiex
Toulbar2	2007-01-12	Marti Sanchez, Javier Larrosa, Matthias Zytnicki, Simon de Givry, and Thomas Schiex

Table 1. Solvers submitted to the 2006 CSP/Max-CSP solver competition.

Execution Environment

The competition was run on a cluster consisting of 93 nodes. Each node of the cluster is equipped with:

- An Intel XEON 3.0 GHz bi-processor running at 3.0 GHz, and having an 800 MHz FSB and 2 MB cache.
- 2 GB DDR PC2700 ECC Registered RAM.

Each solver was run, using the Linux operating system, under the control of a master program which enforced some limits on the memory and the total CPU time used by the solver. The master program also performed some administrative tasks such as recording the output of the solvers, checking the solutions, and so on. Even if the cluster has 64 bits processors, all solvers were run in 32 bits mode.

Instance Selection

The selection of the problem instances was made by the following three independent judges:

- Radoslaw Szymanek (EPFL, Lausanne).
- Djamal Habet (LSIS, Marseille).
- Richard Ostrowski (LSIS, Marseille).

One requirement about the selection was to guarantee a certain balance between binary and non-binary instances, between instances given in extension and in intension, and between random and structured instances. Problems were selected from the following categories:

`ACAD`**:** instances from academia which do not involve any random generator.
`BOOL`**:** instances with only Boolean variables.
`PATT`**:** instances with a structure following a regular pattern (with some random generation).
`QRND`**:** random instances containing a small structure.
`RAND`**:** “pure” random instances.
`REAL`**:** instances originating from real world applications.

Outline

The remainder of this paper presents the results obtained for the two problems (CSP and Max-CSP) in the different categories of instances. Section 2 presents the results for Ordinary CSP and Section 3 presents the results for Max-CSP. A short conclusion is presented in Section 4.

2 Results for the Ordinary CSP Competition

This section presents the main results for the class Ordinary CSP. Within the class, we distinguished between binary and non-binary instances and between intensional and extensional constraints. In addition, we considered the category consisting of problem instances involving the global constraint $allDifferent$.

To solve each CSP instance, the maximum allowed time was set to 30 minutes and the maximum memory limit to 900MB. In each category, results are not presented for solvers that were found to produce incorrect results. It should be noticed that some incorrect results were caused by problems which were related to translating from the competition format to the solvers' native problem specification format.

2.1 Results for Ordinary Binary CSPs

This section presents the results for the class ordinary binary CSP. Within the class, we distinguish between problems specified in intension and extension.

Rank	Solver	Version	SAT	#Instances	Total Time	Average Time
1	VALCSP	3.0	−	1093	59496.90	54.43
2	$buggy^s_{2-5}$	2007-01-08	−	1093	59632.56	54.56
3	$buggy_{2-5}$	2007-01-08	−	1092	58658.14	53.72
4	Abscon	109 AC	−	1082	57755.56	53.38
5	Abscon	109 ESAC	−	1081	54184.15	50.12
6	Mistral	2006-12-04	−	1060	57473.78	54.22
7	Tramontane	2006-12-04	−	1047	53907.19	51.49
8	CSP4J-MAC	2007-01-16	−	1025	62580.90	61.05
9	galac	1	+	940	84144.15	89.52
10	BPrologCSPSolver70a	2006-12-13	−	938	171619.04	182.96
11	galacJ	beta 1	+	928	88183.23	95.03
12	sat4jCSP	1.7 RC BF3	+	917	112685.03	122.88
13	SAT4JCSP-CACHED	1.7 RC BF3	+	915	109227.69	119.37
14	CSPtoSAT+minisat	0.3	+	877	82401.02	93.96
15	sugar	0.40	+	806	54719.92	67.89

Table 2. Results for the **Ordinary CSP Category** "**Binary Constraints in Extension**." Results for solvers that produced incorrect results are not listed. The Colums "Solver," "Version," and "Rank" list the names of the solvers, version information, and their final ranking. The Column "SAT" indicates the SAT-based solvers. The Column "Instances" lists the total number of instances that were solved within the given timeout limit, which was set to 30 minutes. The Columns "Total Time" and "Average Time" list the total and average solution time for the instances that were solved *within* the given timeout limit.

Table 2 presents the results for ordinary, extensional, binary CSPs. As with all rankings, solvers are ordered by decreasing number of problem instances solved, breaking ties by increasing total solution time. It is interesting to notice that there is only a small margin separating the top solvers.

Rank	Solver	Version	SAT	#Instances	Total Time	Average Time
1	$buggy^s_{2-5}$	2007-01-08	−	627	74687.08	119.12
2	$buggy_{2-5}$	2007-01-08	−	626	34572.13	55.23
3	Abscon	109 ESAC	−	575	61532.88	107.01
4	Abscon	109 AC	−	561	20169.62	35.95
5	Tramontane	2006-12-04	−	457	23726.50	51.92
6	CSP4J-Combo	2006-12-19	−	438	88510.72	202.08
7	CSP4J-MAC	2006-12-19	−	432	90051.92	208.45
8	Mistral	2006-12-04	−	423	23648.22	55.91
9	CSP4J-MAC	2007-01-16	−	405	48439.42	119.6
10	sugar	0.40	+	276	15977.41	57.89
11	sat4jCSP	1.7 RC BF3	+	258	18300.66	70.93
12	galac	1	+	245	41654.44	170.02
13	BPrologCSPSolver70a	2006-12-13	−	226	64400.80	284.96
14	galacJ	beta 1	+	220	16881.98	76.74
15	SAT4JCSP-CACHED	1.7 RC BF3	+	217	8445.94	38.92
16	CSPtoSAT+minisat	0.3	+	55	13612.87	247.51

Table 3. Results for the **Ordinary CSP Category** "**Binary Constraints in Intension**." Results for solvers that produced incorrect results are not listed. The Colums "Solver," "Version," and "Rank" list the names of the solvers, version information, and their final ranking. The Column "SAT" indicates the SAT-based solvers. The Column "Instances" lists the total number of instances that were solved within the given timeout limit, which was set to 30 minutes. The Columns "Total Time" and "Average Time" list the total and average solution time for the instances that were solved *within* the given timeout limit.

Some of the problem instances which were used for the category ordinary, extensional, binary CSP are inherited from the first CSP competition, which only allowed problems in extension. What is interesting to notice is that some of these instances, e.g. `QCP` and `QWH`, are basically extensional specifications of CSPs having equality and "generalised SAT" constraints, i.e. constraints that disallow exactly one tuple. Such constraints are easily expressed intensionally. It should be worth while also considering these "legacy" instances for the class intensional CSP in future editions of the competition.

Table 3 presents the results for ordinary, intensional, binary CSPs. This time there is a clearer difference between the solvers. It is interesting to notice that $buggy^s_{2-5}$ managed to solve one more instance than $buggy_{2-5}$ at the price of spending about twice as much average solution time. This difference can be explained by the fact that $buggy^s_{2-5}$ puts more effort in constraint propagation pre-processing, as opposed to solving the problem by search. In a similar vein, `Abscon ESAC` solves 14 more instances than `Abscon AC` by enforcing a higher consistency level before search. This time it comes at the expense of an average solution time exceeding the average solution time of `Abscon AC` by more than three times.

Overall, a possible lesson learnt from the binary CSP competition is that enforcing higher levels of consistency does pay off as far as the number of instances solved (within the time limit) is concerned. However, the addition time which is required seems to be a severe penalty.

2.2 Results for Ordinary N-ary CSPs

This section presents the results for the class ordinary n-ary CSP. Within this class, we distinguish between problems specified in intension and extension.

Rank	Solver	Version	SAT	#Instances	Total Time	Average Time
1	Abscon	109 AC	$-$	277	24132.71	87.12
2	Abscon	109 ESAC	$-$	276	25923.82	93.93
3	Mistral	2006-12-04	$-$	269	36625.34	136.15
4	Tramontane	2006-12-04	$-$	260	41901.79	161.16
5	CSP4J-MAC	2007-01-16	$-$	239	29234.00	122.32
6	CSP4J-MAC	2006-12-19	$-$	234	30795.56	131.60
7	CSP4J-Combo	2006-12-19	$-$	219	33861.56	154.62
8	galac	1	$+$	191	18017.97	94.33
9	SAT4JCSP-CACHED	1.7 RC BF3	$+$	187	15004.32	80.24
10	sat4jCSP	1.7 RC BF3	$+$	187	15174.91	81.15
11	CSPtoSAT+minisat	0.3	$+$	176	5208.18	29.59
12	galacJ	beta 1	$+$	175	8659.69	49.48
13	sugar	0.40	$+$	161	627.05	3.89
14	BPrologCSPSolver70a	2006-12-13	$-$	150	22206.17	148.04

Table 4. Results for the **Ordinary CSP Category** "N**-ary Constraints in Extension**." The Colums "Solver," "Version," and "Rank" list the names of the solvers, version information, and their final ranking. The Column "SAT" indicates the SAT-based solvers. The Column "Instances" lists the total number of instances that were solved within the given timeout limit, which was set to 30 minutes. The Columns "Total Time" and "Average Time" list the total and average solution time for the instances that were solved *within* the given timeout limit.

Table 4 presents the results for ordinary, n-ary, extensional CSPs. From the top to the bottom of the table there is a smooth transition in the number of instances which are solved. With the exception of the average solution time of `sugar`, which seems very low, the solution times are all of the same order.

The order in which the solver `Abscon AC` and `Abscon ESAC` are ranked is the opposite of their ranking for binary CSPs. The difference is negligible since `Abscon AC` only solves one instance more. The average solution time of the two solvers is also comparable.

Table 5 presents the results for ordinary, n-ary, intensional CSPs. It is clear from the results that `BPrologCSPSolver`[3] is a clear winner. However, it requires much more average solution time than the `Abscon` solvers. It is interesting to notice that the

[3] After the initial online publication of the rankings for ordinary n-ary intensional CSPS, which was before the publication of the proceedings, it was discovered by Naoyuki Tamura that some of the `Fisher` instances were incorrectly classified by `BPrologCSPSolver70a` as unsatisfiable. The reason why this had gone unnoticed before is that the `Fisher` instances had not been classified before and the certificate unsatisfiable cannot always be verified. A comment about incorrect answers for some of the `Fisher` instances has been added to the position paper about `BPrologCSPSolver70a`.

Rank	Solver	Version	SAT	#Instances	Total Time	Average Time
1	BPrologCSPSolver70a	2006-12-13	−	579	185136.57	319.75
2	Abscon	109 ESAC	−	509	43430.74	85.33
3	Abscon	109 AC	−	490	42068.73	85.85
4	sugar	0.40	+	431	14414.00	33.44
5	CSPtoSAT+minisat	0.3	+	395	39764.89	100.67
6	CSP4J-MAC	2006-12-19	−	370	49866.12	134.77
7	CSP4J-Combo	2006-12-19	−	364	64358.81	176.81
8	galac	1	+	352	37116.12	105.44
9	galacJ	beta 1	+	331	38483.58	116.26
10	Tramontane	2006-12-04	−	313	29473.79	94.17
11	Mistral	2006-12-04	−	304	34541.39	113.62
12	sat4jCSP	1.7 RC BF3	+	228	15734.88	69.01

Table 5. Results for the **Ordinary CSP Category** “N-ary Constraints in Intension.” Results for solvers that produced incorrect results are not listed. The Colums “Solver,” “Version,” and “Rank” list the names of the solvers, version information, and their final ranking. The Column “SAT” indicates the SAT-based solvers. The Column “Instances” lists the total number of instances that were solved within the given timeout limit, which was set to 30 minutes. The Columns “Total Time” and “Average Time” list the total and average solution time for the instances that were solved *within* the given timeout limit.

ranking of `BPrologSolver` for n-ary intensional CSP is the complete opposite from its ranking for n-ary extensional CSP, where it was ranked last. Clearly there is room for improvement.

2.3 Results for Global Constraints

Table 6 presents the results for ordinary CSPs with global constraints. It is recalled that this was the first time such constraints were introduced. The only constraint which was allowed was $allDifferent$. Overall, the difference in the number of instances solved and the average solution time are small. It is clear that the benchmarks for ordinary CPSs with $allDifferent$ was not discriminative enough.

3 Max-CSP Competition

This section presents the main results for the class Max-CSP. Within the class, we distinguished between binary and non-binary instances and between intensional and extensional constraints.

Recall that there were 9 solvers participating in the Max-CSP competition. This is quite good since this was the first time we considered Max-CSP. Unfortunately, most solvers were incapable of dealing with constraints in intension.

To solve each Max-CSP instance, the maximum allowed time was set to 40 minutes and the maximum memory limit to 900MB.

Rank	Solver	Version	SAT	#Instances	Total Time	Average Time
1	BPrologCSPSolver70a	2006-12-13	−	127	39.90	0.31
2	Abscon	109 AC	−	127	784.80	6.18
3	Abscon	109 ESAC	−	126	145.99	1.16
4	CSP4J-MAC	2006-12-19	−	126	692.14	5.49
5	CSP4J-Combo	2006-12-19	−	126	1041.41	8.27
6	Mistral	2006-12-04	−	125	33.38	0.27
7	Tramontane	2006-12-04	−	125	45.71	0.37
8	CSP4J-MAC	2007-01-16	−	125	412.16	3.30
9	sugar	0.40	+	118	884.51	7.50
10	sat4jCSP	1.7 RC BF3	+	115	2346.01	20.40
11	CSPtoSAT+minisat	0.3	+	114	645.37	5.66
12	SAT4JCSP-CACHED	1.7 RC BF3	+	114	1198.54	10.51
13	galac	1	+	110	492.21	4.47

Table 6. Results for the **Ordinary CSP Category** "**Global Constraints**." Results for solvers that produced incorrect results are not listed. The Colums "Solver," "Version," and "Rank" list the names of the solvers, version information, and their final ranking. The Column "SAT" indicates the SAT-based solvers. The Column "Instances" lists the total number of instances that were solved within the given timeout limit, which was set to 30 minutes. The Columns "Total Time" and "Average Time" list the total and average solution time for the instances that were solved *within* the given timeout limit.

3.1 Results for Binary Max-CSP

This section presents the results for the class binary Max-CSP. Within the class, we distinguish between problems specified in intension and extension.

Table 7 presents the results for binary, extensional Max-CSP. As with all rankings, solvers are ordered by decreasing number of problem instances solved, breaking ties by increasing total solution time. Two versions of Toolbar are clearly the winners of this category. Table 8 presents the results for binary, intensional Max-CSP. Unfortunately, only three solvers participated to this category, which makes it difficult to say much about the final result.

3.2 Results for N-ary Max-CSP

This section presents the results for the class n-ary Max-CSP. Within the class, we distinguish between problems specified in intension and extension.

Table 9 presents the results for n-ary extensional Max-CSP. The top three solvers clearly outperform the other ones. Interestingly, the next four solvers are quite close although they correspond to significantly different solving approaches. Finally, Table 10 presents the results for n-ary intensional Max-CSP.

Rank	Solver	Version	SAT	#Instances	Total Time	Average Time
1	$\text{Toolbar}_{\text{BTD}}$	2007-01-12	–	589	81834.27	138.94
2	toolbar	2007-01-12	–	588	93217.84	158.53
3	Toulbar2	2007-01-12	–	537	123966.36	230.85
4	aolibdvo	2007-01-17	–	466	119217.59	255.83
5	aolibpvo	2007-01-17	–	452	119213.19	263.75
6	AbsconMax	109 PFC	–	437	190992.91	437.05
7	AbsconMax	109 EPFC	–	418	195840.09	468.52
8	CSP4J-MaxCSP	2006-12-19	–	24	13.22	0.55

Table 7. Results for the **Max-CSP Category** "**Binary Constraints in Extension**." Results for solvers that produced incorrect results are not listed. The Colums "Solver," "Version," and "Rank" list the names of the solvers, version information, and their final ranking. The Column "SAT" indicates the SAT-based solvers. The Column "Instances" lists the total number of instances that were solved within the given timeout limit, which was set to 40 minutes. The Columns "Total Time" and "Average Time" list the total and average solution time for the instances that were solved *within* the given timeout limit.

Rank	Solver	Version	SAT	#Instances	Total Time	Average Time
1	AbsconMax	109 EPFC	–	18	5989.58	332.75
2	AbsconMax	109 PFC	–	17	9429.61	554.68
3	CSP4J-MaxCSP	2006-12-19	–	0	—	—

Table 8. Results for the **Max-CSP Category** "**Binary Constraints in Intension**." The Colums "Solver," "Version," and "Rank" list the names of the solvers, version information, and their final ranking. The Column "SAT" indicates the SAT-based solvers. The Column "Instances" lists the total number of instances that were solved within the given timeout limit, which was set to 40 minutes. The Columns "Total Time" and "Average Time" list the total and average solution time for the instances that were solved *within* the given timeout limit.

Rank	Solver	Version	SAT	#Instances	Total Time	Average Time
1	$\text{Toolbar}_{\text{BTD}}$	2007-01-12	–	83	11205.62	135.01
2	Toulbar2	2007-01-12	–	82	37763.06	460.53
3	toolbar	2007-01-12	–	79	10413.75	131.82
4	AbsconMax	109 PFC	–	58	28178.94	485.84
5	$\text{Toolbar}_{\text{MaxSat}}$	2007-01-19	+	56	8899.46	158.92
6	aolibdvo	2007-01-17	–	54	31157.39	576.99
7	AbsconMax	109 EPFC	–	54	32815.47	607.69
8	CSP4J-MaxCSP	2006-12-19	–	4	1.73	0.43

Table 9. Results for the **Max-CSP Category** "N**-ary Constraints in Extension**." Results for solvers that produced incorrect results are not listed. The Colums "Solver," "Version," and "Rank" list the names of the solvers, version information, and their final ranking. The Column "SAT" indicates the SAT-based solvers. The Column "Instances" lists the total number of instances that were solved within the given timeout limit, which was set to 40 minutes. The Columns "Total Time" and "Average Time" list the total and average solution time for the instances that were solved *within* the given timeout limit.

Rank	Solver	Version	SAT	#Instances	Total Time	Average Time
1	AbsconMax	109 EPFC	–	15	5315.92	354.39
2	AbsconMax	109 PFC	–	14	4411.38	315.10
3	CSP4J-MaxCSP	2006-12-19	–	0	—	—

Table 10. Results for the **Max-CSP Category** "N**-ary Constraints in Intension**." Results for solvers that produced incorrect results are not listed. The Colums "Solver," "Version," and "Rank" list the names of the solvers, version information, and their final ranking. The Column "SAT" indicates the SAT-based solvers. The Column "Instances" lists the total number of instances that were solved within the given timeout limit, which was set to 40 minutes. The Columns "Total Time" and "Average Time" list the total and average solution time for the instances that were solved *within* the given timeout limit.

4 Conclusion

This paper presents the main results of the second international CSP solver competition. A paper like this is too small to present all relevant information about an event such as this. The interested reader is invited to visit `http://www.cril.univ-artois.fr/CPAI06/` for more detailed information. XML specifications of all problem instances which were used for this competition may be found at `http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks`.

The third CSP solver competition is planned for 2008. The authors should like to invite anybody to enter the competition. They also wish to invite anybody to submit new problem instances. They are convinced it will be at least as successful an event as the second instance.

Combining Multiple Constraint Solvers: Results on the CPAI'06 Competition Data

Matthew Streeter[1], Daniel Golovin[1], and Stephen F. Smith[2]

[1] Computer Science Department
[2] The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
{matts,dgolovin,sfs}@cs.cmu.edu

Abstract. In a recent paper [5], we presented an algorithm that constructs a schedule for interleaving the execution of two or more solvers, with the goal of obtaining improved average-case running time relative to the fastest individual solver. In this paper, we evaluate this algorithm experimentally using data from the CPAI'06 constraint solver competition.

1 Introduction

Many computational problems that arise in practice are NP-hard and thus are unlikely to admit algorithms with provably good worst-case performance. These problems must nevertheless be solved, and in many problem domains heuristics have been developed that perform much better in practice than a worst-case analysis would guarantee. Unfortunately, the behavior of a heuristic on a previously unseen problem instance can be difficult to predict in advance, and the running times of two different heuristics on the same instance can easily differ by orders of magnitude. For this reason, if a heuristic has been running unsuccessfully for some time it may be worthwhile to suspend the execution of that heuristic and start running a different heuristic instead.

Table 1. Behavior of two solvers on three instances from the CPAI'06 competition.

Instance	`BPrologCSPSolver70a` CPU (s)	`Abscon 109 ESAC` CPU (s)
allIntervalSeries/series-10	0.021	0.72
fisher/FISCHER1-1-fair	0.046	≥ 1800
pseudoSeries/aim/aim-100-1-6-1	≥ 1800	1.089

The potential reduction in average-case running time that can be achieved by interleaving the execution of multiple heuristics is illustrated in Table 1. Here, although both solvers take > 600 seconds on average, a schedule that simply ran the two solvers in parallel would take less than one second on average.

In this paper, we seek to improve the average-case performance of constraint solvers by interleaving the execution of multiple (currently available) constraint solvers according to a *task-switching schedule*. We construct task-switching schedules using a recently-developed algorithm [5] and evaluate their performance using data from the CPAI'06 competition.

1.1 Task-switching schedules

Let $\mathcal{H} = \{h_1, h_2, \ldots, h_k\}$ be a set of deterministic heuristics, and let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of instances of some decision problem. Heuristic h_j, when run on instance x_i, runs for $\tau_{i,j}$ time units before returning a (provably correct) "yes" or "no" answer. A *task-switching schedule* $S : \mathbb{Z}_+ \to \mathcal{H}$ specifies, for each integer $t \geq 0$, the heuristic $S(t)$ to run from time t to time $t+1$. For example, to execute the task-switching schedule depicted in Figure 1 we would run h_1 for two time units; then run h_2 for two time units, then run h_1 for four additional time units, and so on.

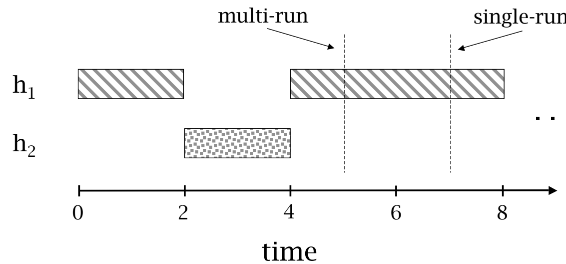

Fig. 1. A task-switching schedule.

A task-switching schedule may either be executed in *single-run mode* or in *multi-run mode*. The two modes differ in what happens to the heuristic (call it h) that is currently running when the task-switching schedule starts running a new heuristic h': in single-run mode, the current run of h is discarded, while in multi-run mode the execution state of h is saved and will be restored if h is run again. For any schedule S, let $c_i^s(S)$ denote the time S takes to solve x_i when S is executed in single-run mode and let $c_i^m(S)$ denote the time it takes when executed in multi-run mode.[3] For example, if heuristics h_1 and h_2 both require 3 time units to solve instance x_i (i.e., $\tau_{i,1} = \tau_{i,2} = 3$), the task-switching schedule S depicted in Figure 1 will require 5 time units to solve x if it is executed in multi-run mode but will require 7 time units if it is executed in single-run mode (i.e., $c_i^m(S) = 5$ and $c_i^s(S) = 7$).

We now consider the problem of computing a good task-switching schedule. That is, given as input the matrix τ, we would like to compute a schedule

[3] Formally, $c_i^m(S)$ is the smallest integer t such that for some heuristic h_j, $|\{t' < t : S(t') = h_j\}| = \tau_{i,j}$. Similarly, $c_i^s(S)$ is the smallest integer t such that, for some heuristic h_j, $S(t - \tau_{i,j}) = S(t - \tau_{i,j} + 1) = S(t - \tau_{i,j} + 2) = \ldots = S(t-1) = h_j$.

that minimizes $\sum_{i=1}^{n} c_i^s(S)$ (or $\sum_{i=1}^{n} c_i^m(S)$). Of course, we would not use the resulting task-switching schedule to solve instances in $\mathcal{X}$ (which we must already have solved in order to fill in the table τ). Rather, we would hope that a task-switching schedule that performs well on the instances in $\mathcal{X}$ would also perform well on similar problem instances, which we would be able to solve more quickly via the task-switching schedule.

Unfortunately, it is NP-hard to compute even an approximately optimal task-switching schedule. This follows from the fact that the problem of computing an optimal task-switching schedule generalizes *min-sum set cover*. Feige et al. [1] showed that it is NP-hard to approximate min-sum set cover within a factor of α for any $\alpha < 4$, and gave a greedy algorithm that achieves the optimal approximation ratio of 4. In [5], we showed how to generalize this greedy algorithm to obtain a 4-approximation to the optimal task-switching schedule. Our results are summarized in the following theorem.

Theorem 1. *Let $C^* = \min_S \sum_{i=1}^{n} c_i^m(S)$. There exists a poly-time greedy approximation algorithm that returns a schedule S^m such that $\sum_{i=1}^{n} c_i^m(S^m) \leq 4C^*$. A different greedy approximation algorithm returns a schedule S^s such that $\sum_{i=1}^{n} c_i^s(S^s) \leq 4C^*$.*

In [5] we also derived bounds on the number of training instances required in order to PAC-learn an optimal (or approximately optimal) schedule for instances drawn independently from a probability distribution. We also developed an online algorithm that receives a sequence $\langle x_1, x_2, \ldots, x_n \rangle$ of problem instances one at a time, and solves each instance (via a task-switching schedule) before moving on to the next.

1.2 Related work

Our work is closely related to previous work on *algorithm portfolios* [2, 3]. An algorithm portfolio consists of a set of heuristics that are run in parallel (or interleaved on a single processor) according to some schedule. The schedules considered in previous work simply run each heuristic in parallel at equal strength and assign each heuristic a fixed restart threshold. The term "algorithm portfolio" has also been used to describe algorithms such as SATzilla [6], which use machine learning to attempt predict which heuristic will solve a given instance the fastest and then run that heuristic exclusively.

Task-switching schedules were introduced in a recent paper by Sayag et al. [4], who gave an exact algorithm for computing an optimal task-switching schedule (as already mentioned, doing so is NP-hard, and the running time of their algorithm is exponential in the number of heuristics). For a more detailed discussion of related work, see [5].

2 Results

In this section, we use the greedy algorithms alluded to in Theorem 1 to construct task-switching schedules for interleaving solvers from the CPAI'06 competition.

To do so, we used the data available on the competition web site[4] to determine the running time of each constraint solver on each benchmark instance. If a solver did not return a solution within the half hour time limit, we artificially set its running time equal to half an hour. We used this data as input to our greedy approximation algorithms. Note that in performing these experiments, we did not actually run any of the constraint solvers.

One might worry that task-switching schedules computed in this way are highly tuned to the specific benchmark instances that were used in the competition. To address this concern, we evaluate our task-switching schedules using leave-one-out cross-validation.

The instances in the CPAI'06 competition were divided into five categories: 2-ARY-EXT, 2-ARY-INT, GLOBAL, N-ARY-EXT, and N-ARY-INT. We performed separate experiments on the instances in each category. We present the results for the category N-ARY-INT in detail, then summarize the results for the other four categories.

2.1 Results for category N-ARY-INT

The CPAI'06 competition included 925 instances in the N-ARY-INT category. Of the 14 solvers that were run on these instances, two produced incorrect answers for one or more instances and were excluded from the competition. 726 of the 925 instances were solved by at least one of the 12 remaining solvers within the half hour time limit. We use these 726 instances and these 12 solvers in our experiments.

Table 2 displays the number of instances solved within the half hour time limit as well as the average CPU time for each of the 12 solvers as well as four schedules: $Greedy^m$, $Greedy^s$, $Parallel^m$, and $Parallel^s$. $Greedy^m$ is the schedule S^m from Theorem 1 executed in multi-run mode, and similarly $Greedy^s$ is the schedule S^s from Theorem 1 executed in single-run mode. $Parallel^m$ is a schedule that runs all 12 heuristics in parallel, each at equal strength. $Parallel^s$ is a single-run version of $Parallel^m$ which first runs each heuristic for 1 second, then runs each heuristic for 2 seconds, then runs each heuristic for 4 seconds, and so on.

As shown in Table 2, the two greedy schedules outperform each of the 12 original solvers as well as the two parallel schedules, both in terms of average CPU time and in terms of the number of instances solved within the half hour time limit. Note that the results listed for the schedules executed in multi-run mode are optimistic in that they assume there is no overhead associated with keeping multiple runs in memory; however **there is no such issue with the schedules executed in single-run mode**. Also note that because we artificially set a solver's CPU time equal to the half hour time limit for instances it did not solve, the values for the average CPU time of the 12 heuristics are actually lower bounds, and using the (unknown) actual values could significantly improve our results. Figure 2 illustrates the task-switching schedule $Greedy^s$.

[4] `http://www.cril.univ-artois.fr/CPAI06/`

Table 2. Results for category N-ARY-INT (cross-validation results are parenthesized).

Solver	Num. solved	Avg. CPU (s)
Greedy^m	706 (701)	338 (407)
Greedy^s	631 (625)	395 (498)
Parallel^m	630	2460
Parallel^s	614	4896
BPrologCSPSolver70a	579	636
Abscon 109 ESAC	509	614
Abscon 109 AC	490	659
sugar	431	766
CSPtoSAT+minisat	395	888
CSP4J - MAC	370	963
CSP4J - Combo	364	998
galac	352	990
galacJ	331	1043
Tramontane	313	1075
Mistral	304	1103
sat4jCSP	228	1264

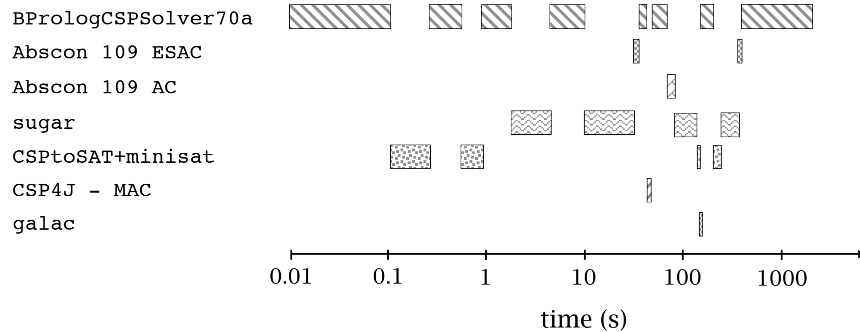

Fig. 2. The task-switching schedule *Greedy*^{s}.

To address the possibility of overfitting, we evaluated the task-switching schedules returned by the greedy algorithm using leave-one-out cross-validation.[5] The cross-validation results appear in parentheses in Table 2. The number of instances solved by *Greedy*^{s} decreased by about 1% under cross-validation, while the average CPU time increased by about 26%. The results for *Greedy*^{m} were similar.

[5] Leave-one-out cross-validation is performed as follows: for each instance, we remove that instance from the matrix τ and run the greedy algorithm on the remaining data to obtain a schedule to use in solving that instance.

2.2 Summary of results for all categories

We performed similar experiments on the instances in the four remaining categories: 2-ARY-EXT, 2-ARY-INT, GLOBAL, and N-ARY-EXT. In each experiment, we removed solvers that produced an incorrect answer on one or more instances, and we removed instances that none of the solvers could solve within the half hour time limit.

The results for all five instance categories are summarized in Table 3. In four out of five categories, the two greedy schedules outperform the corresponding parallel schedules and the best individual solver in terms of the number of instances solved within the time limit. The one exception to this trend is the GLOBAL category, which contained a small number of relatively easy instances. In this category, both the greedy schedules and the parallel schedules solve exactly the same number of instances as the best individual solver. In terms of average CPU time, the greedy schedules consistently outperform the corresponding parallel schedules, and usually (but not always) outperform the best individual solver.

Table 3. Summary of results (cross-validation results are parenthesized).

Category	**Solver**	**Num. solved**	**Avg. CPU (s)**
2-ARY-EXT	$Greedy^m$	1120 (1110)	107 (148)
	$Greedy^s$	1114 (1104)	150 (237)
	VALCSP	1093	126
	$Parallel^m$	1068	588
	$Parallel^s$	1042	1413
2-ARY-INT	$Greedy^m$	682 (674)	127 (167)
	$Greedy^s$	675 (667)	187 (262)
	$Parallel^m$	649	781
	$Parallel^s$	619	1894
	buggy_2_5_s	627	290
GLOBAL	$Greedy^m$	127 (127)	0.13 (1.14)
	$Greedy^s$	127 (127)	0.13 (2.78)
	BPrologCSPSolver70a	127	0.31
	$Parallel^m$	127	1.48
	$Parallel^s$	127	3.61
N-ARY-EXT	$Greedy^m$	298 (296)	298 (425)
	$Greedy^s$	292 (289)	352 (572)
	Abscon 109 AC	277	279
	$Parallel^m$	266	1522
	$Parallel^s$	252	3708
N-ARY-INT	$Greedy^m$	706 (701)	338 (407)
	$Greedy^s$	631 (625)	395 (498)
	$Parallel^m$	632	2109
	$Parallel^s$	614	4896
	BPrologCSPSolver70a	579	636

3 Discussion

In this paper we have investigated the potential for exploiting the complementary strengths of multiple constraint solvers through the use of task-switching schedules. As indicated in Table 3, our results include task-switching schedules that, if entered in the competition, would have run faster on average than any of the individual solvers and would have solved more instances within the half hour time limit. We hope that these results will encourage hybridization of existing constraint solvers.

A natural way to improve on the results presented here would be to use machine learning to take advantage of instance-specific features, as is done in SATzilla [6]. We plan to pursue this approach as future work.

References

1. Uriel Feige, László Lovász, and Prasad Tetali. Approximating min sum set cover. *Algorithmica*, 40(4):219–234, 2004.
2. Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126:43–62, 2001.
3. Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, 1997.
4. Tzur Sayag, Shai Fine, and Yishay Mansour. Combining multiple heuristics. In *Proceedings of the* 23rd *International Symposium on Theoretical Aspects of Computer Science*, pages 242–253, 2006.
5. Matthew Streeter, Daniel Golovin, and Stephen F. Smith. Combining multiple heuristics online. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, pages 1197–1203, 2007.
6. Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla07: The design and analysis of an algorithm portfolio for SAT. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming*, pages 712–727, 2007.

Max-CSP competition 2006: toolbar/toulbar2 solver brief description

S. Bouveret[3], S. de Givry[1], F. Heras[2], J. Larrosa[2], E. Rollon[2], M. Sanchez[1], T. Schiex[1], G. Verfaillie[3], and M. Zytnicki[1]

[1] INRA, Toulouse, France
[2] Dep. LSI, UPC, Barcelona, Spain
[3] ONERA, Toulouse, France

Abstract. This document gives a brief description of the key techniques used in four different versions of `toolbar/toulbar2` solvers submitted to the Max-CSP competition 2006.

All the solvers exploit an initial upper bound found by a local search solver : `maxwalksat` [13] (with 5 tries) for `toolbar/MaxSAT` and `INCOP`[4] [11] for the other solvers. The solvers are implemented in C code, except for `toulbar2` in C++[5].

`toolbar`

The search procedure is *MEDAC** [3], a branch and bound algorithm which maintains the state-of-the-art soft local consistency property *EDAC** during the search. The local consistency enforcement procedure is *à la AC-2001* as described in [9] and it uses specific data structures for efficient binary constraint updating as introduced in [1]. The usual *min domain / max degree* dynamic variable ordering heuristic is employed during the search. Domain values are dynamically ordered by increasing associated unary costs for value enumeration at each node of the search tree.

No particular options are used, except in a preprocessing step where constraints of arity smaller than 10 are projected on binary constraints. Notice that non-binary constraints are delayed from propagation during the search until they become binary.

`toolbar/BTD`

The search procedure is *EDAC-BTD+* [2], a branch and bound algorithm which exploits the problem structure given by a tree decomposition. EDAC-BTD+ extends *BTD* [5] by exploiting local initial upper bounds inside the clusters

[4] The command line parameters are `narycsp result problem.wcsp 0 1 5 idwa 100000 cv v 0 200 1 0 0`.

[5] The solvers are available at the *AlgorithmS* section of `http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/SoftCSP`

and maintaining EDAC* during the search instead of partial forward checking (EDAC* is restricted to the current cluster subtree in order to guarantee time complexity proportional to the tree width).

The *min-degree* heuristic is used to compute a tree decomposition. The root of the tree decomposition is the cluster that minimizes the tree height. Ties are broken by selecting the cluster whose product of domain sizes is maximal. Cluster separators larger than 5 are removed, by merging clusters. The same preprocessing as in `toolbar` for non-binary constraints is performed. The variable ordering heuristic is dynamic (*min domain / max degree*) inside the clusters and follows a compatible order with respect to the cluster tree decomposition. All the variables of a cluster are assigned before its children are examined (in lexicographic order). A hash-table with initial size of 2^{20} is used to memorize cluster lower bounds for pruning and partial solutions for recovering an optimal solution. The value ordering heuristic chooses the last value in the best solution found so far before sorting values by increasing unary costs.

`toolbar/MaxSAT`

The search procedure is *Max-DPLL* [7], a branch and bound algorithm dedicated to Weighted Max-SAT. Max-DPLL is enhanced by several inference rules : *neighborhood resolution* (equivalent to soft AC* in Weighted CSPs), *chain resolution* restricted to binary clauses (equivalent to soft DAC* in Weighted CSPs but with a dynamic DAC ordering), and *cycle resolution* with cycles of triplets of variables, initially proposed in [6]. *Two-sided Jeroslow* dynamic variable ordering heuristic is used during the search.

The usual direct encoding (one Boolean variable per value for non-Boolean variables) is used to convert Weighted CSP instances into Weighted Max-SAT.

`toulbar2`

The search procedure extends the one in `toolbar`, i.e. MEDAC*, in several ways:

- EDAC* also propagates soft ternary constraints as defined in [12].
- The variable ordering heuristic combines a basic form of conflict back-jumping [10] with the usual *min domain / max degree*.
- A limited form of variable elimination (for variables with a degree less than or equal to 2) is applied during the search as proposed in [8].
- The search procedure exploits a binary branching scheme instead of value enumeration. Two different kinds of branching schema are used depending on the domain size of the current chosen variable. If the domain size is greater than 10, then the domain is split into two equal-size parts, creating two new search nodes. Otherwise, the chosen variable is assigned to its *fully supported value* (maintained by EDAC*) or this value is removed from its domain.
- A Limited Discrepancy Search [4] scheme is performed by iteratively running MEDAC* with a power-of-two increasing limit in the number of discrepancies to the value ordering heuristic until optimality proof has been obtained (when no limit occurred).

The first extension yields better lower bounds, especially on the *pedigree* benchmark. The second and third extensions exploit the problem structure better. The forth extension improves propagation and heuristics, especially on problems with large domains as in the *celar* benchmark. The last extension allows to find better upper bounds more rapidly. However, this last option may be counter-productive when a good initial upper bound has been already found by local search as it slows down the time to prove optimality (mainly by a factor of two approximatively).

References

[1] M. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154:199–227, 2004.

[2] S. de Givry, T. Schiex, and G. Verfaillie. Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP. In *Proc. of AAAI-06*, Boston, MA, 2006.

[3] S. de Givry, M. Zytnicki, F. Heras, and J. Larrosa. Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. In *Proc. of IJCAI-05*, pages 84–89, Edinburgh, Scotland, 2005.

[4] W. D. Harvey and M. L. Ginsberg. Limited discrepency search. In *Proc. of the* 14th *IJCAI*, Montréal, Canada, 1995.

[5] P. Jégou and C. Terrioux. Decomposition and good recording. In *Proc. of ECAI-2004*, pages 196–200, Valencia, Spain, 2004.

[6] J. Larrosa and F. Heras. New Inference Rules for Efficient Max-SAT Solving. In *Proc. of AAAI-06*, Boston, MA, August 2006.

[7] J. Larrosa, F. Heras, and S. de Givry. A logical approach to efficient max-sat solving. *Artificial Intelligence*, 172(2–3):204–233, 2008.

[8] J. Larrosa, E. Morancho, and D. Niso. On the practical applicability of bucket elimination: still-life as a case study. *Journal of Artificial Intelligence Research*, 23:421–440, 2005.

[9] J. Larrosa and T. Schiex. Solving Weighted CSP by Maintaining Arc-consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.

[10] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Last conflict based reasoning. In *Proc. of ECAI-2006*, pages 133–137, Trento, Italy, 2006.

[11] B. Neveu and G. Trombettoni. INCOP: An Open Library for INcomplete Combinatorial OPtimization. In *Proc. of CP-03*, pages 909–913, Cork, Ireland, 2003.

[12] M. Sanchez, S. de Givry, and T. Schiex. Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. *Constraints*, 13(1), 2008. Special issue on Bioinformatics and Constraints.

[13] B. Selman, H. Kautz, and B. Cohen. Noise Strategies for Improving Local Search. In *Proc. of AAAI-94*, pages 337–343, Seattle, WA, 1994.

VALCSP Solver : a combination of Multi-Level Dynamic Variable Ordering with Constraint Weighting

Assef Chmeiss, Vincent Krawczyk, Lakhdar Saïs

CRIL - University of Artois - IUT de Lens
Rue Jean Souvraz - SP 18
62307 LENS Cedex 3, France
{chmeiss, krawczyk, sais}@cril.univ-artois.fr

Abstract. The usual way for solving constraint satisfaction problems is to use a backtracking algorithm. One of the key factors in its efficiency is the rule it will use to decide on which variable to branch next (namely, the variable ordering heuristics). In this paper, we propose a solver for binary CSPs: `VALCSP`. It uses and combines two powerful and complementary heuristics. The first one [BCS01] is a look-ahead based heuristic called multi-level variable ordering heuristic, a variable is selected according to a measure that takes into account the properties of the neighborhood of the given variable. The second one [BHLS04] is based on constraint weighting (*Wdeg*). More precisely, a higher weight is given to constraints violated at some previous steps of the search process. Such weighted constraints are used to guide the dynamic variable ordering heuristics of a backtrack search-like algorithms. Our solver is based on the well known MAC algorithm. Arc-consistency is maintained using the AC8 algorithm. In this paper, we give a description of our solver presented to the second International CSP Solver Competition.

1 Introduction

Constraint satisfaction problems (*CSPs*) are widely used to solve combinatorial problems appearing in a variety of application domains.

The usual technique to solve CSPs is the systematic backtracking. It repeatedly chooses a variable, attempts to assign it one of its values, and then goes to the next variable, or backtracks in case of failure. This technique is at the basis of almost all the CSP solving engines. But if we want to tackle highly combinatorial problems, we need to enhance this basic search procedure with clever improvements.

A crucial improvement to be added is look-ahead value filtering, which consists in removing from future domains values that cannot belong to a solution extending the current partial instantiation. Many works have studied the different levels of filtering that can be applied at each node of the search tree. Two famous algorithms maintaining different levels of consistency at each node are forward checking (FC), and maintaining arc consistency (MAC). Several papers

have discussed their performances [SF94,BR96,GS96]. Our solvers use MAC as a search algorithm with the AC8 [CJ98] arc consistency algorithm.

A second kind of improvement that can be added to a backtrack search procedure is to use the knowledge obtained from deadends to avoid future failures coming from the same reason. Backjumping based algorithms [Pro93] are the most famous of these "look back" techniques. In [BHLS04], a simple and efficient criterion is used to direct the search on the most hard and probably inconsistent subpart of the CSP is proposed. It selects the next variable to assign according to its occurrence in the most violated constraint during search. This heuristic is originally proposed in [BGS99] for solving the satisfiability problem.

Another improvement that has been shown to be of major importance is the ordering of the variables (VO), namely, the criterion under which we decide which variable will be the next to be instantiated. Many variable ordering heuristics for solving CSPs have been proposed over the years. However, the criteria used in those heuristics to order the variables are often quite simple, and concentrated on the characteristics inherent to the variable to be ordered, and not too much on the influence its neighborhood could have. Those that used more complex criteria, essentially based on the constrainedness or the solution density of the remaining subproblem, need to evaluate the tightness of the constraints, and so, need to perform many constraint checks.

Our `VALCSP` solver aims to use the influence of the neighborhood in the criterion of choice of a variable [BCS01], while remaining free of any constraint check. It, also, combines this heuristic with constraint weight.

2 Definitions and notations

A constraint network is defined by a set of variables $\mathcal{X}$, each taking values in its finite domain $D_i \in \mathcal{D}$, and a set of constraints $\mathcal{C}$ restricting the possible combinations of values between variables. The set of variables implied in a constraint c will be denoted by Vars(c).

Any constraint network can be associated with a *constraint graph* in which the nodes are the variables of the network, and an edge links nodes if and only if there is a constraint on the corresponding variables.

$\Gamma_{init}(x_i)$ denotes the set of nodes sharing an edge with the node x_i (its initial neighbors). We define the set $\Gamma(x_i)$ as the current neighborhood of x_i, namely, the neighbors remaining uninstantiated once a backtracking search procedure has instantiated the set $Y = \{X_{i_1}, \ldots, X_{i_k}\}$ of variables, i.e., $\Gamma(x_i) = \Gamma_{init}(x_i) - Y$. The size of $\Gamma(x_i)$ (resp. $\Gamma_{init}(x_i)$) is called the *degree* (resp. initial degree) of x_i.

3 Search heuristic

In the `VALCSP` solver, we combine the Multi-Level ordering heuristic and the Wdeg heuristic. This solver is based on the MAC algorithm which uses the following skills:

- the AC8 [CJ98] arc consistency algorithm is used to maintain arc consistency during search.
- The multi-level variable ordering [BCS01] is the basic heuristic used to choose the next variable to assign (defined in 3.1)
- The solver uses the lexicographic heuristic to choose the next value in the domain of the current variable.
- The branching scheme is d-way branching for domains of size *d*.

Let us recall the notion of Multi-Level ordering heuristic proposed in [BCS01] and the Wdeg heuristic proposed in [BHLS04].

3.1 Multi-Level variable ordering heuristics

Let us first define $W(C_{ij})$ as the weight of the constraint C_{ij} and,
(1) $W(x_i) = \frac{\sum_{x_j \in \Gamma(x_i)} W(C_{ij})}{|\Gamma(x_i)|}$
as the mean weight of the constraints involving x_i. In order to maximize the number of constraint involving a given variable and to minimize the mean weight of such constraints, the next variable to branch on should be chosen according to the minimum value of
(2) $H(x_i) = \frac{W(x_i)}{|\Gamma(x_i)|}$
over all uninstantiated variables.

For complexity reasons, the weight we will associate to a constraint must be something cheap to compute (e.g., free of constraint checks). it can be defined by $W(C_{ij}) = \alpha(x_i) \circledcirc \alpha(x_j)$, where $\alpha(x_i)$ is instantiated to a simple syntactical property of the variable such as $|D_i|$ or $\frac{|D_i|}{|\Gamma(x_i)|}$, and $\circledcirc \in \{+, \times\}$.

We obtain the new formulation of (2):
(3) $H_\alpha^\circledcirc(x_i) = \frac{\sum_{x_j \in \Gamma(x_i)} \alpha(x_i) \circledcirc \alpha(x_j)}{|\Gamma(x_i)|^2}$

Multi-level generalization In the formulation of the dynamic variable orderings (DVOs) presented above, the evaluation function $H(x_i)$ considers only the variables at distance one from x_i (first level or neighborhood). however, when arc consistency is maintained (MAC), the instantiation of a value to a given variable x_i could have an immediate effect not only on the variables of the first level, but also on those at distance greater than one.

To maximize the effect of such a propagation process on the CSP, and consequently to reduce the difficulty of the subproblems, we propose a generalization of the DVO $H_\alpha^\circledcirc$ such that variables at distance k from x_i are taken into account. This gives what we call a "multi-level DVO", $H_{(k,\alpha)}^\circledcirc$. To obtain this multi-level DVO, we simply replace $\alpha(x_j)$ in formula **(3)** by a recursive call to $H_{(k-1,\alpha)}^\circledcirc$. The recursion terminates with $H_{(0,\alpha)}^\circledcirc$, equal to α. This is formally stated as follows:
(4) $H_{(0,\alpha)}^\circledcirc(x_i) = \alpha(x_i)$
(5) $H_{(k,\alpha)}^\circledcirc(x_i) = \frac{\sum_{x_j \in \Gamma(x_i} \alpha(x_i)) \circledcirc H_{(k-1,\alpha)}^\circledcirc(x_i)}{|\Gamma(x_i)|^2}$

3.2 Wdeg heuristic

The main goal behind the `Wdeg` heuristic is to exploit informations about previous step of the search and to direct the search to the most constrained subproblem. More precisely, a counter, called $W(C_{ij})$, with any constraint C_{ij} of the problem. These counters will be updated during search whenever a dead-end (domain wipeout) occurs. As systematic solvers such as FC or MAC involve successive revisions of variables in order to remove values that are no more consistent with the current state, it suffices to introduce a test at the end of each revision. If the constraint under test is violated, its counter is increased by one.

Using these counters, it is possible to define a new variable ordering heuristic, denoted `Wdeg`, that gives an evaluation $H_{wdeg}(x_i)$, called weighted degree, of any variable x_i as follows:

$$H_{wdeg}(x_i) = \sum_{x_j \in \Gamma(x_i)} W(C_{ij})$$

3.3 `VALCSP` solver : combining multi-level DVO with Constraint weighting

Our `VALCSP` solver combines both Multi-Level and Constraint weighting heuristics. It defined as following:

$$H_{lw}(x_i) = \frac{|D_i| + \sum_{x_j \in \Gamma(x_i)} |D_j|}{W(C_{ij})}.$$

To compute $H_{lw}(x_i)$, we consider for each constraint C_{ij}, the ratio between the sum of the domains size of all x_j (and x_i) and the weight $W(C_{ij})$. A sum is calculated on all the constraints involving a given variable x_i.

4 Filtering algorithm

A preprocessing step achieves arc consistency on the CSP. We also maintain arc consistency during search with the MAC algorithm.

The main arc consistency algorithm used in our solvers is AC8. It's proposed by Chmeiss and Jegou in [CJ98].
AC8 is based on supports but without recording them. When a value $a \in D_i$ is removed from its domain, AC8 records the reference of the variable x_i, that is the number i, in the queue of propagation denoted *Queue-AC*. Propagations will be realized with respect to variables in this queue. Suppose that a variable x_i is removed from the queue. Then, all neighbooring variables will be considered, i.e. for all $x_j \in \mathcal{X}$ such that $C_{ji} \in \mathcal{C}$, and for each value $b \in D_j$, AC8 will ensure that there is a value $a \in D_i$ such that $(a, b) \in R_{ij}$ holds. Unlike AC6, AC8 has to start again the search from the first value of the domains. If no support a of b is found in D_i, then b must be deleted, and the number of the variable, namely j must be inserted in *Queue-AC*. To ensure that j is not duplicated in *Queue-AC*, we must maintain an array of booleans (a bit vector), denoted *Status-AC*,

recording the status of variables. So, the data structures used to AC8 are the *Queue-AC* containing variables which have lost some values in their domain and not propagated yet, and the boolean table *Status-AC* that always verifies $\{i \in \textit{Queue-AC} <=> \textit{Status-AC}[i]\}$.

5 Summary: submitted solver

In this paper, we have given a description of our solver for binary CSP which we submit to the second International CSP Solver Competition. It's a dedicate solver to binary CSPs. This solver uses the AC8 algorithm to maintain arc-consistency during search. `VALCSP` solver combines Multi-Level and constraint weighting heuristics (see sections 4.2 and 4.3). The constraints in intention are changed into equivalent constraints in extension.

Finally, we mention that this solver is implemented using the C ANSI programming language.

References

[BCS01] C. Bessière, A. Chmeiss, and L. Sais. Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of CP'01*, pages 565–569, 2001.

[BGS99] L. Brisoux, E. Gregoire, and L. Sais. Improving backtrack search for sat by means of redundancy. In *Proceedings of ISMIS'99*, pages 301–309, 1999.

[BHLS04] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.

[BR96] C. Bessiere and J-C. Regin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Principles and Practice of Constraint Programming*, pages 61–75, 1996.

[CJ98] A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):121–142, 1998.

[GS96] S.A. Grant and B.M. Smith. The phase transition behavior of maintaining arc consistency. In *Proceedings of ECAI'96*, pages 175–179, 1996.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problems. *Computational Intelligence*, 9(3):268–299, 1993.

[SF94] D. Sabin and E.C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In Alan Borning, editor, *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94, Rosario, Orcas Island, Washington, USA*, volume 874, pages 10–20, 1994.

Alternative methods for learning in CSP Solvers

Diarmuid Grimes and Richard J. Wallace

Cork Constraint Computation Centre and Department of Computer Science
University College Cork, Cork, Ireland
{d.grimes,r.wallace}@4c.ucc.ie

Abstract. This paper outlines the major features of three solvers entered in this years CPAI solver competition. The solvers shared the same underlying architecture which is described in detail in the following paper, the base solver is *rjw* (submitted by Richard Wallace) which *Diarmuid-rndi* and *Diarmuid-wtdi* (both submitted by Diarmuid Grimes) are built on. Furthermore the solvers all used constraint weighting as a means for identifying sources of contention in the problem. However two of the solvers (*Diarmuid-rndi* and *Diarmuid-wtdi*) used restarting to approximate sources of global difficulty, while *rjw* learnt local information.

1 Introduction

The solver *rjw* is an implementation of the classical treesearch algorithm for solving CSPs, consisting of backtracking plus lookahead (using MAC-3 for consistency maintenance). In other words, it is a complete algorithm that uses depth-first backtracking with d-way branching, interleaved with propagation. Currently it is restricted to problems involving only extensional binary constraints.

This solver is designed for experimentation so the features that have received the most attention are related to this aim. Naturally efficiency is always an issue as the efficiency impacts the experiments that can be performed. The present version represents fairly mature code which has not been changed in any fundamental way over the past few years. We firstly describe *rjw*, then the solvers *Diarmuid-rndi* and *Diarmuid-wtdi* which build on *rjw* by combining learning with restarts in quite different ways [GW07].

2 Basic Features of *rjw*

The solvers are implemented in Common Lisp and are designed to run on a Unix machine. Since many lisp compilers only compile into a kind of intermediate code, the program cannot run with the speed of a C or a C++ program.

The present version of *rjw* has a 'backbone' that is a recursive procedure (which naturally limits the size of the problems that can be handled). The basic structure is shown in Algorithm 1; as indicated, it uses recursion to run through a list of variables and a list of values in the domain of the current variable.

In the actual code, this structure is elaborated to:

- heuristically choose the next variable (in clause 3).

Algorithm 1: Basic recursive structure underlying tree search in the present solver.

```
Search(variables, domain, solution)
if variables = nil then                                   /* clause 1 */
    save-solution
    return t
else if domain = nil then                                 /* clause 2 */
    reset data structures
    return nil                                            /* backtrack */
else if
arc-consistency(next-variable, next-domain-value, remaining-variables)
AND
Search(remaining-variables, new-domain, solution+next-assignment) then
/* clause 3 */
    return t
else return Search(remaining-variables, remaining-domain, solution)
```

- set up data structures for handling arc consistency (in clause 3). (These are reset in clause 2, as indicated.)
- handle all-solutions as well as one-solution search.

In addition, the MAC solver tests for singleton domains and only does arc consistency when the current domain has more than one value. Incidentally, during search consistency maintenance is only carried out following each new instantiation (i.e. not after a value has been discarded). As per usual only arcs between a variable whose domain has changed and its unassigned neighbors are added to the queue (so the initial queue comprises solely of arcs between the variable just assigned and its unassigned neighbors). A full arc consistency is, of course, carried out prior to search.

3 Data Structures

Domain values are kept in simple lisp lists, accessed via an array. A list of variables is also used, as indicated in Algorithm 1. The current (partial) assignment is stored as an array, with nil values for currently unassigned variables. This array is accessed by variable-numbers, so it accommodates dynamic variable ordering. (Hence, the resetting in clause 2 in Algorithm 1 includes setting the value for the current variable to nil.)

In the present implementation, constraint relations are represented as arrays of binary values; the size is set by the size of the largest domain. The arrays themselves are accessed via a hash table, where the hash key is based on the two variable-numbers. Two arrays are stored for each relation, so the program does not have to put the variables in any particular order when computing the hash key.

The constraint representation is a global data structure that can be accessed by any function in connection with search or heuristic selection. This is also true for the current assignment. Global structures are also used to maintain a list of the original variables and the original domains.

Another important global data structure is the set of adjacency lists, which are lists of variables adjacent to a given variable in the constraint graph. Again, these are kept in an array so they can be accessed by variable-numbers. There are also data structures for certain parametric features like the degree of each variable, and the current domain sizes, and the original tightness of each constraint, which are used by certain heuristics.

A critical data structure used in connection with propagation stores current domains during search. The basic strategy is to maintain lists of lists within an array. Each list of lists is handled as a stack, with the current domain at the top. Using an array allows the program to access the current domain via the variable-number.

In order to use this structure during recursive search, the setup function (in clause 3 in Algorithm 1) adds a duplicate of the current domain to the top of each stack. (For forward checking this need only be the domains of variables adjacent to the current variable.) As successive assignments are made at a given level of search, the arc consistency functions take the domain just below the top of the stack before support testing and replace the top-most list with the adjusted domain afterwards.

This means that no special (setting-up) code is required for this purpose when reassigning a variable. If the program backtracks from a given level of search (clause 2 in Algorithm 1), the stacks are cut back so the domains are as they were when this level was entered.

4 Heuristics

One of the major uses to which this solver has been put in the last few years has been the experimental study of variable ordering heuristics. So there are a large number of heuristics - and anti-heuristics coded. These are organized in an elaborate but tedious manner for selecting a particular heuristic during a given run of the program.

For the competition, rjw used the domain over weighted-degree heuristic ($dom/wdeg$) of Boussemart et al. [BHLS04], in this case, the heuristic code is in the same file as the search code and the heuristic is called directly. For all solvers, values were chosen lexically (as were arcs during consistency maintenance).

For the implementation of the weighted-degree heuristic all constraints have a weight, initially set to 1, associated with them. The weights are stored in a hash-table with a hash key based on the variable-numbers of the variables in the constraint. Whenever a constraint causes a domain wipeout during consistency maintenance the weight gets incremented by 1. The weighted-degree of a variable is the sum of the weights on constraints between the variable and its uninstantiated neighbors. The heuristic then chooses the variable with minimum ratio of domain to weighted-degree.

5 Environment and I/O

The solvers normally run either interactively or in batch mode, where they take the same instructions from a command file. There is an i/o module that currently accepts two kinds of problem formats, both involving extensional constraints. The top-level of the program (not used in the competition) is a menu-driven system.

Different top-level commands either read in the next problem (and set up most of the global data structures), or generate the next problem, or call for a solver of some generic type (e.g. backtrack, hybrid tree search, local search). During this interactive process, after an algorithm is selected, further menu options allow the heuristic to be selected for the run.

6 Restarting Strategies

Both restarting strategies follow the logic of Refalo [Ref04] who emphasises the importance of making good choices at the top of search. Refalo suggests that, in general, to achieve this one must perform some preprocessing on a problem. By collating information regarding contentious variables through their constraint weights, and then restarting one moves these contentious variables to the top of the search tree.

The solver *Diarmuid-wtdi* works as follows: there is an initial failure cutoff C_0 where search runs until C_0 failures have occurred. It then restarts with a new cutoff which is the previous cutoff multiplied by a constant factor z. Thus the failure cutoff C_R for the Rth restarted search is: $C * (z^{R-1})$.

The only difference between this solver and the solver *rjw* is the restart procedure, so MAC-3 is the consistency algorithm and $dom/wdeg$ is the variable ordering heuristic. The solver stops as soon as it finds a solution or proves the problem insoluble. It is complete since the cutoff is increased with every restart.

Since weights are consistently being updated, the variable ordering is always changing, thus search is unlikely to revisit an identical part of the search space upon restarting. This allows the solver to visit different parts of the search space while still maintaining a large degree of confidence in the variables selected. In fact, since at each restart it has more information available for $dom/wdeg$ to make its early decisions, it should improve its ordering with each restart. However this is contingent on the information learnt being of uniform quality which is not necessarily the case.

The solver *Diarmuid-rndi* is an automated learning approach [Gri07] to problem solving that aims to boost the power of the $dom/wdeg$ heuristic by randomly probing the search space for information prior to a complete search using $dom/wdeg$. A pseudocode description is given in Algorithm 2.

The solver works as follows: similarly to *Diarmuid-wtdi* there is an initial failure cutoff C. However this cutoff is never incremented. Search runs identically to *rjw* with the one exception that variables are chosen randomly at each selection point (line 5). Constraint weights are updated throughout but are never used to guide search. If the problem is solved (line 9) or the problem proven insoluble (line 7) during these "random probes" then the solver stops (although insolubility is unlikely to be proven since the failure cutoff is normally quite low, for the competition the cutoff was 100 failures per probe).

After every R restarts (where R modulo $10 = 0$ and $R > 10$) the stability of the variables weights is checked (line 12) between the Rth restart and the $(R-10)^{th}$ restart until either the stability criteria has been satisfied or the number of restarts is equal to a predefined maximum number of restarts (line 4). Thus there is a minimum number of

Algorithm 2: Automated Probing Algorithm

```
   input  : A constraint satisfaction problem P, cutoff C, maximum number of restarts R
   output: Solution or insoluble
1  RestartedSearch (P, C, R)
2  restarts = 0
3  stabilised = false
4  while ((restarts < R) ∧ (¬ stabilised)) do
5      Solution = Solve (P, SelectRandomVariable, Value Heuristic, C)
6      if Solution ≠ Cutoff then
7          if Solution = false then
8              return insoluble
9          else
10             return Solution
11     else if (restarts > 10) ∧ (restarts mod 10 = 0) then
12         stabilised = CheckStability
13         restarts ++
14     else
15         restarts ++
16 Solution = Solve (P, SelectDomWdegVariable, Value-Heuristic, ∞)
17 if Solution = false then
18     return insoluble
19 else return Solution
```

random probes per problem, equal to 20. The maximum number of probes per problem used in the competition was 100.

When either the stability criteria has been satisfied or the maximum number of restarts has been reached, search restarts for the final time (line 16). The cutoff is removed (i.e. search runs to completion), and $dom/wdeg$ is used for variable selection with the weights learnt during preprocessing being interleaved with weights learnt during this final search.

7 Results Discussion

There are some points of interest regarding the results of the solvers submitted. Firstly, and somewhat surprisingly, *Diarmuid-wtdi* solved fewest problems. Although *Diarmuid-rndi* solved a few more problems than *rjw*, it was not a significantly greater number.

As expected *rjw* was much quicker for easier problems where the cost of preprocessing was an unnecessary hindrance to *Diarmuid-rndi*. For the harder problem sets (e.g. problem sets "random" and "tightness") the random probing approach solved more and was quicker than *rjw*. However the magnitude of improvement was not as large as initially expected.

This is partially because the automated probing approach was still in its infancy when the solver was submitted. As mentioned the cutoff for the probes was 100 failures which, in retrospect, was somewhat excessive and led to the preprocessing being quite costly to perform. We have found that using a smaller cutoff of 30 failures for this strategy does not result in degradation of information learnt and results in improved overall performance [Gri07].

Acknowledgment. This work was supported in part by Science Foundation Ireland under Grant 00/PI.1/C075.

References

[BHLS04] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proc. Sixteenth European Conference on Artificial Intelligence-ECAI'04*, pages 146–150, 2004.

[Gri07] Diarmuid Grimes. Automated within-problem learning for constraint satisfaction problems. In *First Workshop on Autonomous Search, in conjunction with CP 2007*, 2007.

[GW07] D. Grimes and R. J. Wallace. Learning to identify global bottlenecks in constraint satisfaction search. In *20th International FLAIRS Conference*, 2007.

[Ref04] P. Refalo. Impact-based search strategies for constraint programming. In M. Wallace, editor, *Principles and Practice of Constraint Programming-CP'04. LNCS No. 3258*, pages 557–571, 2004.

Mistral, a constraint satisfaction library

Emmanuel Hebrard

Cork Constraint Computation Center & University College Cork

Abstract. Mistral is a constraint library written in C++, and initially evolved from EFC [9]. It is an open source light weight and relatively efficient toolkit providing many of the functionalities offered by most well known solvers such as Ilog Solver, Gecode, Choco, etc. For instance, several efficient algorithms for filtering global constraints (Alldifferent [11], Global Cardinality [12], NValue [2] or the Edge Finder algorithm for scheduling [5]) are implemented in Mistral.

1 Introduction

Mistral is a constraint library written in C++. It was created as an evolution, or rather a simplification of EFC [9], adapted to find *super solutions* [8]. EFC (for Extended Forward Checking) was originally written by Fahiem Bacchus and later developed by George Katsirelos. The current version of *Mistral* was entirely rewritten, and thus it now shares very little with EFC. The initial idea was to write a light weight constraint solver, implementing MAC (*Maintain Arc Consistency*) as well as the usual techniques that made the success of constraint programming (variable and value ordering heuristics, global constraints, etc). Over time, some additional features, such as restarts and branch & bound search were added. Besides a few global constraints, there is no major novelty implemented in *Mistral*.

2 Implementation

2.1 Problem Modelling

Mistral is a constraint library and not a language, hence if offers very limited modelling shortcut or syntactic sugar. Figure 1 illustrates a typical implementation of the Nqueens problem. The types `IntVar` and `SVar` correspond respectively to a general integer variable, and to a specific implementation. A model usually involves three steps. First, the CSP and the variables are declared (lines 4 to 9). Next the constraints are posted (lines 11 to 18). Finally a solver is declared and the method `solve()` is called (lines 20 to 22), effectively solving the model.

2.2 Data Structures

Four different types of variables are implemented in *Mistral*, all subclasses of `IntVar`, in other words all are finite domain integer variables.

Integer Variables as bit-vectors (SVar) In this implementation, domain $D(x)$ is represented as vector of bits. All set operations, such as union, intersection or difference can be performed in $O(n/32)$ where $n = max(D(x)) - min(D(x))$. Of course membership, insertion and deletion can be performed in constant time.

For each variable, an array of size $min(maxlevel, |D(x)|)$ of such domains is created (memory is statically allocated befor search begins). Whenever the domain change for the first time for a given level in the search tree, it is first copied in the next available index of this array. The space complexity for each variable implemented in this way is therefore:

$$min(maxlevel, |D(x)|).\frac{max(D(x)) - min(D(x))}{32}$$

```
1   int main(int argc, char *argv[]) {
2     int N = ( argc > 1 ? atoi(argv[1]) : 50 );
3
4     // model and variables
5     CSP model;
6     IntVar* queens[N];
7     for( int i = 0; i < N; ++i )
8       queens[i] = new SVar(N);
9     model.add(queens, N);
10
11    // constraints
12    model.post( new AllDiffConstraint(queens, N) );
13    for(int i=0; i<N-1; ++i)
14      for(int j=i+1; j<N; ++j) {
15        IntVar *scope[2] = {queens[i], queens[j]};
16        model.post( new DiagonalConstraint(scope,
17                    queens[i]->id-queens[j]->id) );
18      }
19
20    // solver
21    MACSolver s(&model, "dom/deg");
22    s.solve();
23  }
```

Fig. 1. A model for the NQueens problem in *Mistral*.

Integer Variables as lists (LVar) In this implementation, a domain $D(x)$ is represented both as vector of bits mainly for checking membership, and as a doubly linked list for faster iteration. Notice that the order of the values in the list is not guaranteed to be lexicographical. This is because when backtracking, the list of deleted values is appended at the head of the domain list. Deletion of multiple

values at once (such as setting the maximum or the minimum) will tend to be more expensive in this representation, however, iterating through the values of a domain is linear in the number of values.

For each variable, an array of size $min(maxlevel, |D(x)|)$ of `int` is created. Whenever the domain change at a given level, the value is removed form the domain list, appended to the head of *delta* list which is pointed by next available index in this array. When backtracking to this level, the delta list appended to the head of the domain list. The space complexity for each variable implemented in this way is therefore:

$$min(maxlevel, |D(x)|) + (max(D(x)) - min(D(x)))$$

Boolean Variables (BVar) Boolean variables have their own specific implementation for optimisation purpose. For instance, the variable does not store any data for backtracking.

Their space complexity is therefore constant (in $O(1)$).

Interval Variables (RVar) In this implementation, only a lower bound and an upper bound are stored in order to represent large intervals.

However, two array of integers (one for the lower bound and one for the upper bound) of size *maxlevel* are allocated. The space complexity of an interval variable is therefore $O(maxlevel)$.

2.3 Algorithms

Binary Backtrack Search A standard two-ways branching algorithm is implemented. It is given in integrality in Figure 2. First, the termination conditions are checked. If either the cutoff is reached (lines 2,3) or a solution is found (lines 4,5) then the status is changed accordingly and the search ends. In lines 6 to 8, the variable and value ordering heuristic is called and the next decision is selected. This decision (that is, a *left branch*) is explored in lines 10 to 18. When this branch is explored exhaustively, the complementary *right branch* is explored in line 20 to 28. Notice that when the left branch was unsuccessful because the cutoff was reached, the right branch is not explored (condition in line 22). When both branches were unsuccessful, a backtrack occurs (lines 30 to 32).

Generic Arc Consistency Algorithm The *arc consistency* algorithm used for extensionally defined constraints is a slightly modified version of the simple `AC3` algorithm. This modification is sometime called *residual* `AC3` [10]. Whenever a support is found for a given pair ⟨ variable, value ⟩, it is stored. Then the next time a support need to be found for this pair, the stored support is checked for validity (i.e., whether the values involved in this support are still in their respective domains). If not, the regular `AC3` algorithms proceeds as normal. Since nothing needs to be done when backtracking (as opposed to `AC2001` for instance), the overhead in time complexity is marginal, whilst the gain is in practice noticeable.

```
1   // // End of search?
2   if( limitsExpired() )
3     return (status = LIMITOUT);
4   if( allAssigned() )
5     return solutionFound();
6
7   // // Select a variable and a value
8   int idx, value = Variable::NOVAL;
9   Variable *curvar = future[(idx = heuristic->select(value))];
10
11  // // Left branch
12  curvar->makeDecision(value);
13  if( curvar->assigned )
14    bound(idx);
15  if( filtering() ) {
16    ++level;
17    if( genericBacktrack() == SAT ) return SAT;
18    --level;
19  }
20
21  // // Right branch
22  restoreTo(level-1);
23  if( status != LIMITOUT ) {
24    if( curvar->makeComplementary(value) && filtering() ) {
25      ++level;
26      if( genericBacktrack() == SAT) return SAT;
27    } else ++level;
28    // // Backtrack
29    restore();
30    ++BACKTRACKS;
31  } else ++level;
32
33  return status;
```

Fig. 2. *Mistral*'s backtracking procedure.

3 Solver Competition

3.1 Representations of Variables

The different implementations of variables were used according to heuristic rules. The simplest one being that Boolean Variables were always represented using `BVar`. The range variables (`RVar`) were only used when extra variables with very large domains (larger than "*maxlevel*") were required. This is explained in more detail in Section 3.2. Finally the choice between `SVar` and `LVar` was down to the size and the density of the domain. The list implementation `LVar` is intuitively better when the ratio $\frac{D(x)}{max(D(x))-min(D(x))}$ is low, since being able to iterate in linear time through a sparse domain is valuable. Moreover, when the size of the domain increase, the space used for the list is outbalanced by the conciseness of the backtracking structure (a single integer as opposed to a domain for `SVar`). Therefore `LVar` was used for large and/or sparse domains, whilst `SVar` was used in all other cases.

3.2 Representations of Constraints

Relations: Relations are represented as an n dimensional matrix flattened into a vector of bits. The worst case space complexity is not very good, since sparse or dense matrix have the same size. Moreover, in order to keep the membership operation in constant time, the size of a binary relation between x and y is stored using $(max(D(x)) - min(D(x))).(max(D(y)) - min(D(y)))$ bits, instead of $|D(x)|.|D(y)|$. However this was never a problem during the first round of the competition.

Predicates: Two different representations of predicates were used, both of them pretty standard. During the first round of the competition only the first version was implemented, and comported several bugs. Given a tree of binary and unary predicates, a set of as many respectively ternary and binary reified constraints and extra variables are created. For instance for the predicate:

$$eq(add(mul(X_0, X_1), X_2), X_3)$$

the following constraints will be posted:

$$mul(X_0, X_1, Y_0)$$
$$add(Y_0, X_2, Y_1)$$
$$eq(Y_1, X_3)$$

where Y_0 and Y_1 are extra variables; $mul(X_0, X_1, Y_0)$ constrains the product of X_0 and X_1 to be equal to Y_0; $add(Y_0, X_2, Y_1)$ constrains the sum of Y_0 and X_2 to be equal to Y_1; and $eq(Y_1, X_3)$ will substitute X_3 to Y_1 in all other constraints. Notice that the latter substitution is only possible because the constraint eq is at the root of the predicate tree, otherwise a ternary constraint $eq(Y_1, X_3, Y_2)$ would be posted, constraining Y_2 to be the truth value of the relation $Y_1 = X_3$.

The second representation is used for low arity constraints encoded as a predicates tree. In this case instead of extra variables and constraints, a unique constraint is posted. This constraint is propagated using the generic arc consistency algorithm, that is, the algorithm used for extensional constraints. However, instead of implementing constraint checks using a Boolean matrix, the predicate tree is stored and queried at each constraint check. This is essentially equivalent to transforming the predicate into a table constraint, albeit with slightly worse time complexity and better space complexity.

Predefined Constraints: Most of the usual arithmetic and logic predicates ($+, -, \leq, \neq, \&, \leftrightarrow$) are predefined. For instance, we illustrate in Figure 3 the propagation algorithm for the $\neq$ predicate.

```
1  bool NotEqualConstraint::propagate(IntVar *v, const int event)
2  {
3    return scope[(scope[0] == v)]->remove(v->value());
4  }
```

Fig. 3. *Mistral*'s Not-Equal propagator.

Notice that this propagator is called only on assignment of the variable v (the second parameter "event" is used to describe the type of event, when needed).

3.3 Search Strategy

No specific value heuristic was used in the competition, the values are therefore visited in lexicographical order for `SVar`, `RVar` and `BVar` and in an undefined order for `LVar`. The variable ordering heuristic is a slightly modified version of *domain over weighted degree* [4]. As in the regular framework, each constraint $C(V)$ over a set of variable V is associated with a weight $w(C)$ and the variable with minimum ratio *domain size* over *sum of neighbouring constraints weights* is chosen:

$$\text{choose } x \text{ such that } \frac{|D(x)|}{\sum_{x \in V} w(C(V))} \text{ is minimum}$$

However, on failure during the GAC closure procedure, the constraint responsible for the failure gets its weight incremented by $maxlevel - level$ instead of 1. The intuition behind this choice is that a failure early in the search is more meaningful than a failure later, since less decisions have been taken.

Two versions of *Mistral* were submitted in the competition, one of them implementing a geometric restart policy. The initial cutoff was set to $\frac{2}{3}.maxlevel$, and then multiplied by $1 + \frac{1}{3}$ upon every restart. It is worth noticing that a very limited form of nogood learning naturally happens when restarting the basic backtracking procedure illustrated in Figure 2. Indeed, the procedure is

called with the level set to 1. When a left branch, for instance the decision $X = v$, is unsuccessful on level 1, then the complementary decision $x \neq 1$ is explored. However, this decision is globally consistent and therefore never withdrawn, therefore $x \neq 1$ is a unary nogood, reused upon subsequent restarts. Moreover, observe that arbitrarily many left branches and many variables may be chosen at level 1. Therefore, several unary nogood, involving different variables may be "learnt".

4 Features

Some novel global constraint propagation algorithm have been implemented in *Mistral* in order to perform empirical evaluation. The NValue constraint, for example, as described in [2] is implemented in *Mistral*.

The NValue constraint counts the number of distinct values used by a vector of variables. It is a generalisation of the widely used AllDifferent constraint [6, 13]. It was introduced in [7] to model a musical play-list configuration problem so that play-lists were either homogeneous (used few values) or diverse (used many). There are many other situations where the number of values used are limited. For example, if values represent resources, we may have a limit on the number of values used at the same time. A NValue constraint can thus aid both modelling and solving many real world problems.

Definition 1. NValue$(N, [X_1, \ldots, X_m])$ *holds iff* $N = |\{X_i|\ 1 \leq i \leq m\}|$

Enforcing generalised arc consistency (GAC) on the NValue constraint is NP-hard [3]. One way to deal with this intractability is to decompose the constraint or to approximate the pruning. The NValue constraint can be decomposed into two other global constraints: the AtMostNValue and the AtleastNValue constraints. Unfortunately, while enforcing GAC on the AtLeastNValue constraint is polynomial, enforcing GAC on the AtMostNValue constraint is also NP-hard. We will therefore focus on various approximation methods for propagating the AtMostNValue constraint.

Mistral features three new approximations. Two are based on graph theory while the third exploits a linear relaxation encoding. These algorithms compare favourably to a previous approximation method due to Beldiceanu based on intervals that runs in $O(n \log(n))$ [1]. The two new algorithms based on graph theory are incomparable with Beldiceanu's, though one is strictly tighter than the other. Both algorithms, however, have an $O(n^2)$ time complexity. However, the linear relaxation method dominates all other approaches in terms of the filtering, but with a higher computational cost.

References

1. N. Beldiceanu. Pruning for the *Minimum* Constraint Family and for the *Number of Distinct Values* Constraint Family. In Toby Walsh, editor, *Proceedings of the*

7th International Conference on Principles and Practice of Constraint Programming (CP-01), volume 2239 of *Lecture Notes in Computer Science*, pages 211–224, Paphos, Cyprus, 2001. Springer-Verlag.

2. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. Filtering Algorithms for the NValue Constraint. In Roman Barták and Michela Milano, editors, *Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-05)*, volume 3524 of *Lecture Notes in Computer Science*, pages 79–93, Prague, Czech Republic, 2005. Springer-Verlag.
3. C. Bessiere, E. Hebrard, B. Hnich, and T. Walsh. Tractability of Global Constraints. In Mark Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP-04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 716–720, Toronto, Canada, 2004. Springer-Verlag. Short paper.
4. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting Systematic Search by Weighting Constraints. In Ramon López de Mntaras and Lorenza Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04)*, pages 482–486, Valencia, Spain, August 2004. IOS Press.
5. J. Carlier and E. Pinson. Adjustment of Heads and Tails for the Job-Shop Problem. *European J ournal of Operational Research*, 78:146–161, 1994.
6. M. Dincbas, P. van Hentenryck, H. Simonis, and A. Aggoun. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 693–702, Tokyo, Japan, 1988.
7. P. Roy F. Pachet. Automatic Generation of Music Programs. In Joxan Jaffar, editor, *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP-99)*, volume 1713 of *Lecture Notes in Computer Science*, pages 331–345, Alexandria, VA, USA, 1999. Springer-Verlag.
8. E. Hebrard, B. Hnich, and T. Walsh. Robust Solutions for Constraint Satisfaction and Optimization. In Ramon López de Mntaras and Lorenza Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04)*, pages 186–190, Valencia, Spain, 2004. IOS Press.
9. G. Katsirelos and F. Bacchus. A library for solving CSPs. http://www.cs.utoronto.ca/ gkatsi/efc/, 2001.
10. C. Likitvivatanavong, Y. Zhang, J. Bowen, and E.C. Freuder. Arc Consistency in MAC: a new perspective. In Mark Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP-04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 93–107, Toronto, Canada, October 2004. Springer-Verlag.
11. A. Lopez-Ortiz, C-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In Georg Gottlob and Toby Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 245–250. Morgan Kaufmann, 2003.
12. C-G. Quimper, A. Lopez-Ortiz, P. van Beek, and A. Golynski. Improved Algorithms for the Global Cardinality Constraint. In Mark Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP-04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 542–556, Toronto, Canada, October 2004. Springer-Verlag.
13. J.C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In Barbara Hayes-Roth and Richard E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, Seattle, WA, USA, 1994. AAAI Press.

CSP2SAT4J: A Simple CSP to SAT translator

Daniel Le Berre[1] and Inês Lynce[2]

[1] CRIL-CNRS FRE 2499
Rue Jean Souvraz SP 18
62307 Lens Cedex FRANCE*
[2] IST/INESC-ID,
Technical University of Lisbon, Portugal

Abstract. SAT solvers can now handle very large SAT instances. As a consequence, many translations into SAT have been shown successful in recent years: Planning and Bounded Model Checking are two examples of applications in which SAT engines are reported to be as good as or even better than dedicated software. During the first CSP competition, SAT-based approaches were demonstrated competitive with the other CSP solvers on binary constraints. Constraints being provided in extension, it was a big advantage for techniques based on grounding predicates since only benchmarks that could be grounded in a reasonable space were available. As such, the comparison between SAT-based solvers (that need to ground predicates) and the approaches developed by the CSP community (that usually handle directly the constraints as expressed) was not fair. For the second edition of the competition, the constraints can now be given in intension, and global constraints such as *allDifferent* are available. The idea behind the submission of CSP2SAT4J is to show when SAT-based CSP solvers can still compete in some cases against "traditional" CSP solvers under those new conditions.

1 Introduction

The idea behind the submission of CSP2SAT4J is to show when SAT-based CSP solvers can still compete against "traditional" CSP solvers under those new conditions: on non-randomly generated binary constraints benchmarks, when the domain of the variables is not too big (a few hundred variables max) to allow grounding the predicates in reasonable time. The translation from CSP to SAT has been improved since last year submission: the solver can outperform another CSP solver (namely Abscon) on last year competition's benchmarks. The underlying SAT solver can handle cardinality constraints, which minimizes the number of constraints used in the translation. The evaluation of the constraints given in intention is done using a JavaScript to Java bytecode compiler, in order to keep the "Keep It Simple Stupid" approach of last year submission. The experimental results on the new benchmarks available in July 2006 do show the limit

* The first author has been supported in part by the IUT de Lens, the CNRS and the Region Nord/Pas-de-Calais under the TAC Programme and the COCOA project. The two authors were partly founded by the PAI-PESSOA project MUSICA.

of the approach when the size of the domains increases: for some categories of benchmarks in which the solver performed well last year (Queens and Knight for instance), the solver is unable to ground the bigger instances available in intention this year. Since the participation to the competition also implies submitting benchmarks, we included in the last section the description of our encoding of the *Social Golfer Problem*.

2 On the power of SAT solvers

Very recently, the translation from pseudo boolean into SAT was shown competitive and sometimes better than dedicated solvers during the PB05 evaluation [20]. Indeed, the MiniSAT+ solver shown surprising good performances on most of the benchmarks, and it can be considered as the “winner” of the evaluation. The MiniSAT solver, together with the SatELite preprocessor were the winners of the SAT competitions in the industrial and crafted categories. The solver MiniSAT+ has shown that using SAT engines as efficient generic problem-solving engines was a reality. For that reason, we decided to submit a SAT-based CSP solver to the first (and second) CSP competition.

The SATisfiability problem (SAT) gained interest from the industry a few years ago when SAT solvers where used to solve Bounded Model Checking problems [6] instead of BDDs. That interest pushed people to design solvers for those particular types of problems. One of the particularities of those benchmarks is their size: they are“huge” compared to the classical pathological pigeon hole or random k-SAT problems. As a consequence, the complexity of the algorithms and data structures becomes even more important. That observation was the origin of the design of the head-tail lazy data structure in SATO[26, 27], the Watched Literals and cheap VSIDS heuristic in Chaff [17, 28]. Grounding CSP problems into SAT does generate huge CNF, so it makes sense to use ”industrial” SAT solvers for solving those formulas.

Another particularity of SAT instances coming from BMC, or more generally practical problems translated into SAT compared to the seminal 3-CNF instances, is the lack of real characterization of those SAT instances: while the theory around 3-CNF allows to build powerful heuristics-based SAT solvers for 3-SAT (TABLEAU, CSAT, POSIT, SATZ, CNFS, etc), non-chronological backtracking and learning looks like the best approach to tackle SAT-encoded problems (SATO,RELSAT,GRASP,CHAFF). Note that usually non-chronological backtracking and learning is useless if the heuristic is good enough, which explains why solvers using those techniques are outperformed on random 3-SAT instances by heuristics based solvers. Furthermore, it seems that the lack of “structure” in the problem makes the VSIDS heuristic ineffective. On the other hand, non-chronological backtracking can repair mistakes made by the heuristics by analyzing conflicts.

One of the consequence of using an “industrial” SAT solver to power a SAT-based CSP solver in the CSP competition is to have poor performances on randomly generated CSP benchmarks. While a randomly generated CSP benchmark

encoded into a SAT benchmark cannot be considered as a randomly generated SAT benchmark, we conjecture that the bad behavior of our approach on randomly generated CSP benchmarks is due to the lack of "structure" needed by conflict driven clause learning SAT solvers.

All the parts of a SAT solver received a huge interest from both an algorithmic and an implementation point of view so current SAT solvers are now heavily tuned and they should not be considered as prototype software but rather as production software. We use the SAT4J library[3], an open-source library of conflict driven clause learning (aka "industrial") SAT solvers in Java. The library is mature and competitive with state-of-the-art SAT solvers: it participated to the 2004 and 2005 SAT competitions in which it went in the second stage in the industrial category, and passed the qualification step of the SAT Race 2006, a competition of SAT solvers especially dedicated to industrial SAT benchmarks.

In the rest of the paper, we first describe how our SAT library follows the current trend to generalize SAT solvers to handle constraints more general than clauses. Then we explain the CSP to SAT encodings used in our solver and provide some experimental results.

3 From clauses to pseudo boolean constraints

Researchers are pushing the limit beyond SAT: Quantified Boolean Formulas (QBF) and Stochastic SATisfiability (SSAT) for instance are two extensions of SAT being studied recently. Another extension of SAT received some attention a decade ago: using pseudo boolean constraints (linear constraints with boolean variables) instead of plain clauses [3, 4]. Most of the solvers for those extensions to SAT are developed using techniques that were demonstrated powerful for SAT. Those solvers in the early 90s were based on DPLL[9, 8] while the solvers developed today are often related to Chaff-like solvers.

This is especially true for pseudo boolean solvers: Barth first developed a DPLL version of a pseudo boolean solver [4]. Walser [24] and later Prestwich [18, 19] developed local search or hybrid pseudo boolean solvers. Aloul et al [22] developed a version of Chaff handling pseudo boolean constraints instead of clauses as input, plus symmetry breaking predicates, with clause learning (same thing for the recent MiniSAT [13]). Dixon and Ginsberg[10] developed a pseudo boolean version of Relsat (PRS), which was the first pseudo boolean solver including true pseudo boolean learning. They developed a pseudo boolean version of Chaff (PBChaff[11]) in the same spirit while Chai and Kuehlmann [7] did extended all Chaff techniques (learning scheme and data structures) in the pseudo boolean solver Galena. Recent work from Dixon et al [12] describes a generic conflict driven constraint learning solver based on group theory while Thiffault et al [23] describe a conflict driven clause learning solver working with arbitrary boolean gates.

SAT4J uses some principles taken from both Chai and Kuehlmann and Dixon to allow some of its solvers to use cutting planes instead of resolution when using

[3] http://www.sat4j.org/

linear pseudo boolean constraints. As a result, those solvers can solve in a linear number of conflicts the pigeon hole problem when it is expressed by linear pseudo boolean constraints, which is not possible with a solver based only on resolution. But that power comes with a high price to pay in practice: on benchmarks with few pseudo boolean constraints, such approach is not as efficient as a solver applying resolution on pseudo boolean constraints.

As a consequence, many SAT solvers are currently using general constraints to express the problem in a more compact way than with pure clauses (e.g. for cardinality constraints), without using the full power of those constraints, but without additional running time either. This is how we setup our own SAT solver for the CSP competition.

4 From CSP to SAT

Our CSP to SAT translator uses two different encodings: direct encodings [25] and support encoding for binary clauses [14]. Note that we use a single cardinality constraint instead of using binary (so-called "at most") clauses to express that no more than one value can be chosen in a domain.

The encoding used depends on the way the constraints are expressed:

extension (conflict) In that case, it is straightforward to use a direct encoding since each tuple is translated into a clause.

extension (support) If the constraint is binary, then we use the binary support encoding, else the direct encoding, by generating all conflicting tuples.

intension The constraint is grounded by generating all the possible input values and checking if it satisfies or not the constraint. If the constraint is binary, then the binary support encoding is used, else the direct encoding is used. A better option might be to approximate the number of allowed or forbidden tuples and to select the encoding accordingly. A more sophisticated and efficient way to generate the tuples to be considered is also a possible way of improvement.

The main drawback of our method is the way we handle n-ary constraints: for a constraint of arity 8 with domains of size 10, 10^8 tuples need to be generated. It is currently impossible to simply enumerate all those tuples in a reasonable time.

We also implemented the generalized support encoding [5] for n-ary constraints. However, the cost for generating that encoding is much higher than the direct encoding. We are aware of another CSP to SAT encoding that we have not experimented because it relies on a very specific way to describe the problem in terms of disjunction of forbidden values [19].

4.1 Common encoding

The translator takes the new XML representation (XML CSP 2.0 format) of the problem as input and outputs a set of constraints (mixing clauses and cardinality constraints) to feed our extended SAT solver.

Variables For each variable $v_i \in V$, and each domain value $d_j \in Domain(v_i)$, a propositional variable $p_{i,j}$ is created.

Domains For each variable $v_i \in V$, a cardinality constraint denotes that only one value from the domain can be chosen: $\sum_x p_{i,x} = 1$. In practice, that constraint is expressed in the solver using the clause $\vee_x p_{i,x}$ and the cardinality constraint $\sum_x p_{i,x} \leq 1$

4.2 Direct encoding [25]

Forbidden tuples (nogoods) Each tuple representing a forbidden combination of values is represented by a propositional clause composed by the negation of the propositional variables representing those values. So the length of the generated clause is the arity of the constraint. Note that in case of binary constraints, binary clauses will be generated.

Allowed tuples (supports) When a relation is represented by allowed tuples, we deduce all the forbidden tuples and translate them into clauses as described above.

The main drawback of that translation is the translation of the allowed tuples. It can take a lot of time to generate them when the arity of the constraint increases.

4.3 Support encoding for binary constraints [14]

Forbidden tuples (nogoods) Each tuple representing a forbidden combination of values is represented by a propositional clause composed by the negation of the propositional variables representing those values.

Allowed tuples (supports) For binary constraints, we create a clause $\neg a \vee b_1 \vee \ldots \vee b_k$ for each variable a that appears in tuples $(a, b_1), (a, b_2), \ldots (a, b_k)$

4.4 From intension to extension

Our solver grounds predicates in intension into tuples, in order to apply the above translation. Compared to the first competition, the cost of grounding the predicate is added to the CSP solver, and it might not be possible to ground some of the problems in reasonable time or space. The biggest issue for dealing with constraints in intention in our SAT-based approach is to evaluate the expressions. Since our aim is simply to evaluate them for a complete assignment of the variables, and since we want to keep as low as possible the portion of our code dedicated to CSP solving, we decided to have both a pragmatic and extensible approach to do it. Indeed, it can be expected that the next versions of the input format will see more and more built-in functions. Furthermore, the new version of the Java virtual machine will ship with a Java Script interpreter called Rhino[4]. So we decided to interpret the predicates defined in the input file as a javascript expression. This can be easily achieved by defining in

[4] http://www.mozilla.org/rhino/

JavaScript the built-in functions and load them before evaluating the expression. That framework also has the good property to allow to compile directly the JavaScript expression into Java bytecode, so the cost of evaluating the expression is reduced. The main drawback of that approach in our opinion is that the Rhino framework is 700Kb big while the SAT4J library is only 400Kb big. Adding a dependency to such a package makes our CSP solver temporarily more than 1MB big on current JVM).

4.5 The allDifferent global constraint

One of the new features of the second version of the CSP input format is the ability to express global constraints. For the second competition, only the *allDifferent* constraint is available. That constraint has some nice properties and is very useful to eliminate values in domains. A translation into SAT of the *allDifferent* constraint preserving some of those properties was proposed in [16]. However, we decided to use a simpler approach: for each *allDifferent* constraint we simply add the binary clauses ensuring that no couple of variables share a common value: it is a sort of local direct encoding of the constraint, since the forbidden tuples of the global constraint can be easily expressed by binary constraints ($allDiff(x_1, x_2, \ldots, x_n) \equiv \wedge_{i<j} x_i \neq x_j$). In some sense, we are not taking advantage that way of the constraint being global.

Note that a specific data structure proposed by Lawrence Ryan [21] is used in our SAT solver to handle binary clauses because the implementation of the *allDifferent* constraint is likely to produce many of them.

5 A few experimental results

We present here some experimental results comparing our SAT-based CSP solver against Abscon, one of the strongest CSP solvers that participated in the first CSP competition. Note that Abscon and our own solver are to the best of our knowledge the only CSP solvers freely available for research purpose that are compatible with the first and second CSP competition input format. Note also that the two solvers are written in Java.

All the results were obtained on a cluster of Bi-Xeon 2.6 GHz with 2GB of memory (1GB per processor) running Linux, using Java 1.5.0_06 for 32 bits architecture. The timeout was 20 mn per benchmark.

5.1 First CSP competition benchmarks (extension)

These results were obtained on January 2006 on the set of benchmarks used for the first CSP competition, plus some additional random benchmarks. The version of SAT4J used was 1.5_01. We used a developer version of Abscon. The benchmarks were given in extension. The first part of the table (*All* column) summarizes the results of the two solvers on all the benchmark (number of problems solved) classified into binary and n-ary ones. Abscon is far better than SAT4J

overall, and especially on n-ary satisfiable benchmarks. The second column restricts the results to the benchmarks that were not randomly generated. In that case, SAT4J is slightly better than Abscon on binary benchmarks.

	All		Non-random	
	SAT4J	Abscon	SAT4J	Abscon
non binary constraints				
	(186 benchmarks)		(150 benchmarks)	
UNSAT	27	**28**	27	**28**
SAT	61	**125**	48	**108**
binary constraints				
	(2031 benchmarks)		(1041 benchmarks)	
UNSAT	842	**995**	**400**	396
SAT	760	**827**	**560**	536

These results simply show that provided that grounding the problem is possible, a SAT-based approach is competitive with Abscon for binary benchmarks non-randomly generated.

5.2 Benchmarks in XML 2.0 format

These results were obtained in July 2006. The version of SAT4J used was a CVS snapshot tagged OBJECTWEB 1.0.90 (the one submitted to the CSP competition) and the version of Abscon was 105.

	SAT4J	Abscon
non binary constraints (978 benchmarks)		
UNSAT	69	**78**
SAT	273	**453**
binary constraints (2673 benchmarks)		
UNSAT	613	**1053**
SAT	861	**1285**

Unfortunately, we do not have the details of random/non-random benchmarks. However, a few remarks can help reading these results:

Queens/Knights During the first CSP competition, the direct encoding gave poor results on those benchmarks (none of them solved). Using the support encoding allowed SAT4J to solve all them quickly (in less than 2 minutes overall). The benchmarks proposed this year are much bigger: queensKnights-50 has for instance a domain size of 2500. As a consequence, enumerating 2500^2 tuples just makes the SAT-based approach hopeless on those bigger benchmarks.

Fapp There are 40 series of 11 benchmarks for the FAPP benchmarks (binary benchmarks). For the first CSP competition, only the first two series (the smaller ones) were submitted because the other ones were too big to be expressed in extension. Our approach is only able to solve the benchmarks of the first series and a few from the second series, and runs out of memory on the other ones. On the other hand, Abscon is able to solve almost all of them. Those particular series of benchmarks represents 1/6 of the total number of benchmarks, while there are more than 25 different sets of benchmarks: the difference in number of problems solved in the table should be considered at the light of that fact. We expect to have closer results between SAT4J and Abscon during the second CSP competition because the number of problems will be close for each kind of benchmarks.

Out of Memory happened in 633 cases on binary benchmarks, and 204 cases on n-ary benchmarks, i.e. respectively in 24% and 21% of the total number of benchmarks! It happened starting at domino-2000, fapp-02, knights-50, queens-knights-50 and js-taillard-15 for binary benchmarks. For n-ary benchmarks, it happened mostly on pseudo boolean benchmarks translated into CSP and on traveling salesman problems, golomb ruler, all interval series and mknap. It happened even on some problems given in extension in n-ary benchmarks, because we need to generate forbidden tuples when supports tuples are given.

6 The social golfer problem

The social golfer problem is derived from a question posted to `sci.op-research` in May 1998:

The coordinator of a local golf club has come to you with the following problem. In her club, there are 32 social golfers, each of whom play golf once a week, and always in groups of 4. She would like you to come up with a schedule of play for these golfers, to last as many weeks as possible, such that no golfer plays in the same group as any other golfer on more than one occasion.

In other words, this problem can be described more explicitly by enumerating four constraints which must be satisfied:

- The golf club has 32 members.
- Each member plays golf once a week.
- Golfers always play in groups of 4.
- No golfer plays in the same group as any other golfer twice.

Since 1998, this problem has become a famous combinatorial problem. It is problem number 10 in CSPLib (http://www.csplib.org/). A solution is said to be optimal when maximum socialisation is achieved, i.e. when one golfer plays with as many other golfers as possible. Clearly, since a golfer plays with three new golfers each week, the schedule cannot exceed 10 weeks. This follows from the fact that each golfer plays with three other golfers each week. Since there is a total of 31 other golfers, this means that a golfer runs out of opponents

after 31/3 weeks. For some years, it was not known if a 10 week (and therefore optimal) solution for 32 golfers exists. In 2004, Aguado found a solution using design-theoretic techniques [1].

Even though the social golfer problem was described for 32 golfers playing in groups of 4, it can be easily generalized. An instance to the problem is characterized by a triple w-p-g, where w is the number of weeks, p is the number of players per group and g is the number of groups. The original question therefore is to find a solution to the w-4-8 problem, with w being the maximum, i.e. to find a solution to 10-4-8 (or prove that none exists).

The social golfer problem is related with other well-known combinatorial problems. Indeed, this problem is a generalisation of the problem of constructing a round-robin tournament schedule, the main difference being that in the social golfer problem the number of players in a group may be greater than two. Also, the social golfer problem of finding a 7 week schedule for 5 groups of 3 players (7-3-5) is the same as Kirkman's Schoolgirl Problem, where the main goal is to arrange fifteen schoolgirls in rows of three so that each schoolgirl walks in the same row with every other schoolgirl exactly once a week.

The encoding used is the one proposed by Walser available on CSPLIB that can be summarized as follows:

- 0-1 variables $Golfer_{i,j,k} = 1$ indicate that golfer i plays in group j in week k.
- 0-1 variables $Meet_{i,j,k} = 1$ indicate that golfers i and j meet in week k (thus are in the same group).
- constraints relating the above variables: $Golfer_{i,k,l} + Golfer_{j,k,l} - Meet_{i,j,l} \leq 1$.
- golfers play in exactly one group per week: $\sum_j Golfer_{i,j,k} = 1$.
- each pair of golfer plays only once: $\sum_k Meet_{i,j,k} = 1$.
- each group has exactly p golfers: $\sum_i Golfer_{i,j,k} = p$.

Note that while the domain of the variables is small (boolean), some constraints have a big arity (the number of golfers $p*g$) which makes the SAT-based approach inefficient (enumerating 2^{32} tuples for a 8-4-8 problem for instance is out of reach for our solver).

The problems 8-4-8, 9-4-8 and 10-4-8 that we have submitted are quite challenging. It would be a good news if some competitors were able to solve them. Some recent work on a SAT encoding with symmetry breaking predicates can be found in [15]. [2] has proposed to dynamically break symmetries in the social golfers problem. This new approach is often able to outperform the traditional approaches, although at the cost of eliminating some solutions. Hence, the proposed method is incomplete.

7 Conclusion

We presented our new SAT-based CSP solver as submitted to the second CSP competition. We presented some experimental results showing that a SAT-based

approach for CSP is quite competitive provided that the problems are not randomly generated and contain binary constraints with "reasonable size" domain to make the grounding of the predicates possible. We also presented our encoding of the *Social Golfer Problem* for which we provided a generator and 10 samples benchmarks for the competition. We believe that one solution to cope with the predicates given in intension that cannot be grounded in reasonable time or space is to manage them as a new kind of constraint in the SAT solver. Two manipulations are needed in the case of a conflict driven constraint learning solver:

value propagation the constraint should be able to cope with partial assignment (of boolean variables provided by the SAT solver) and to detect which values in the domains need to be assigned/forbiden as a result of a domain assignment (thus leading to unit propagation in the SAT solver).

reason computation Conflict analysis is an important part of the SAT solver. It relies on computing for each propagated assignment a reason for that assignment in the form of a set of literals (the set of falsified literals in a clause). The biggest issue in our opinion will be to make sure that such a reason can be computed in a predicate given in intention and to see how a possible solution relates with CSP backjumping and nogood learning.

We hope to be able to compare our solver against numerous other CSP solvers in the future: it would be nice if the solvers that participate in the CSP competition could be freely available for research purpose after the competition.

References

1. Alexandro Aguado. A 10 days solution o the social golfer problem., 2004. Manuscript.
2. Francisco Azevedo. An Attempt to Dynamically Break Symmetries in the Social Golfers Problem. In Francisco Azevedo, editor, *11th Annual ERCIM Workshop on Constraint Programming (CSCLP'2006)*, pages 101–115, 2006.
3. Peter Barth. Linear 0-1 inequalities and extended clauses. Technical Report MPI-I-94-216, Saarbrcken, Germany, 1994.
4. Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI–I–95–2–003, Saarbrcken, 1995.
5. Christian Bessière, Emmanuel Hebrard, and Toby Walsh. Local consistencies in sat. In *Proceedings of the Sixth International Conference on Theory and Application of Satisfiability Testing (SAT'2003)*, volume 2919 of *LNCS*, pages 299–314. Springer, May 2003.
6. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of bdds. In *Proceedings of Design Automation Conference (DAC'99)*, 1999.
7. Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *ACM/IEEE Design Automation Conference (DAC'03)*, pages 830–835, Anaheim, CA, 2003.
8. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. In *Communications of the Association for Computing Machinery 5*, pages 394–397, 1962.

9. M. Davis and H. Putnam. A computing procedure for quantification theory. In *Journal of the ACM, 7*, pages 201–215, 1960.
10. Heidi E. Dixon and Matthew L. Ginsberg. Combining satisfiability techniques from AI and OR. In *The Knowledge Engineering Review 15*, page 53, 2000.
11. Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *Proceedings of The Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, pages 635–640, 2002.
12. Heidi E. Dixon, Matthew L. Ginsberg, and Andrew J. Parkes. Generalizing boolean satisfiability I: Background and survey of existing work. In *Journal of Artificial Intelligence Research 21*, 2004.
13. Niklas Een and Niklas Sorensson. An extensible sat solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing*, LNCS 2919, pages 502–518, 2003.
14. Ian Gent. Arc consistency in sat. In *Proceedings of ECAI'2002*, pages 121–125, 2002.
15. Ian P. Gent and Ines Lynce. A sat encoding for the social golfer problem. In *Proceedings of the IJCAI'05 workshop on Modeling and Solving Problems with Constraints*, July 2005.
16. Ian P. Gent and Peter Nightingale. A new encoding of alldifferent into SAT. In *3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems (CP2004)*, pages 95–110, 2004.
17. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, June 2001.
18. S. Prestwich. Randomised backtracking for linear pseudo-boolean constraint problems. In *Proceedings of Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'2002)*, pages 7–20, 2002.
19. S. Prestwich. Incomplete dynamic backtracking for linear pseudo-boolean problems: Hybrid optimization techniques. *Annals of Operations Research*, 130(1-4):57–73, August 2004.
20. Olivier Roussel and Vasco Manquinho. The first pseudo boolean solver evaluation. http://www.cril.univ-artois.fr/PB05/.
21. Lawrence Ryan. Efficient algorithms for clause learning sat solvers. Master's thesis, SFU, February 2004. Available at http://www.cs.sfu.ca/∼mitchell/papers/ryan-thesis.ps.
22. Fadi A. Aloul Arathi Ramani Igor L. Markov Karem A. Sakallah. Symmetry-breaking for pseudo-boolean formulas. In *International Workshop on Symmetry on Constraint Satisfaction Problems (SymCon)*, pages 1–12, County Cork, Ireland, 2003.
23. Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving Non Clausal Formulas with DPLL Search. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP'2004)*, Toronto, Canada, September 2004.
24. J. P. Walser. Solving Linear Pseudo-Boolean Constraint Problems with Local Search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 269–274, 1997.
25. Toby Walsh. Sat vs csp. In Springer, editor, *Proceedings of CP'2000*, pages 441–456, 2000.

26. H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996.
27. Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.
28. L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, November 2001.

Abscon 109
A generic CSP solver

Christophe Lecoutre and Sebastien Tabary

CRIL-CNRS FRE 2499,
Université d'Artois
Lens, France
{lecoutre,tabary}@cril.univ-artois.fr

Abstract. This paper describes the algorithms, heuristics and general strategies used by the two solvers which have been elaborated from the *Abscon* platform and submitted to the second CSP solver competition. Both solvers maintain generalized arc consistency during search, explore the search space using a conflict-directed variable ordering heuristic, integrate nogood recording from restarts and exploit a transposition table approach to prune the search space. At preprocessing, the first solver enforces generalized arc consistency whereas the second one enforces existential SGAC, a partial form of singleton generalized arc consistency.

1 Introduction

A constraint network (CN) is composed of a set of variables (each of them with an associated domain corresponding to a set of values) and a set of constraints (defining the tuples of values allowed for variables of each constraint). Finding a solution to a constraint network involves assigning a value to each variable such that all constraints are satisfied. The Constraint Satisfaction Problem (CSP) is the task to determine whether or not a given constraint network, also called CSP instance, is satisfiable (i.e. admits a solution).

A CSP solver is a program which deals with satisfiability of CSP instances. It is said complete when it can prove that an instance is either satisfiable or unsatisfiable. Most of the CSP solvers are composed of two main components: Inference and Search. Inference is used to transform an instance into an equivalent form which is simpler than the original one, while search is used to traverse the search space of the instance in order to find a solution. For (most of the) complete CSP solvers, it respectively corresponds to constraint propagation and depth-first search with backtracking guided by some heuristics.

In this document, we quickly introduce the inference strategy (Section 2) and the search strategy (Section 3) used by the two solvers that we have presented at the second CSP solver competition.

2 Inference Strategy

Many works in the area of Constraint Programming have focused on inference, and more precisely, on filtering methods based on properties of constraint net-

works. The idea is to exploit such properties in order to identify some nogoods where a nogood corresponds to a partial assignment (i.e. a set of variable assignments) that can not lead to any solution. Properties that allow identifying nogoods of size 1 are called domain filtering consistencies [7]. The interest of exploiting domain filtering consistencies is that it is quite easy to deal with nogoods of size 1. Indeed, as such nogoods correspond to inconsistent values, it suffices to remove them from domains of variables.

Generalized Arc consistency (GAC) is a domain filtering consistency which guarantees that each value admits at least a support in each constraint. GAC remains a fundamental property of constraint networks. It is called AC (Arc Consistency) when constraints are binary (i.e. only involve 2 variables). M(G)AC is the algorithm that maintains the (G)AC property at each node of a search tree.

2.1 AC3^{bit+rm} and GAC3^{rm}

In a (coarse-grained) Arc Consistency (AC) algorithm, *revise* is the procedure which determines if a value is supported by a constraint. A residual support, or residue, is a support that has been found and stored during a previous execution of the procedure *revise*. The point is that a residue is not guaranteed to represent a lower bound of the smallest current support of a value. In [15], a study about the theoretical impact of exploiting residues with respect to the basic algorithm AC3 is given. First, it is proved that AC3^{rm} (AC3 with multi-directional residues) is optimal for low and high constraint tightness. Second, it has been shown that during a backtracking search, MAC2001 presents, with respect to MAC3^{rm}, an overhead in $O(\mu ed)$ per branch of the binary tree built by MAC, where μ denotes the number of refutations of the branch, e the number of constraints and d the greatest domain size of the constraint network. One consequence is that MAC3^{rm} admits a better worst-case time complexity than MAC2001 for a branch involving μ refutations when either $\mu > d^2$ or $\mu > d$ in the case of constraints with low or high tightness.

In [21], we have proposed to exploit bitwise operations to speed up some important computations such as looking for a support of a value in a constraint, or determining if a value is substitutable by another one. Considering a computer equipped with a x-bit CPU, one can then expect an increase of the performance by a coefficient up to x (which may be important, since x is equal to 32 or 64 in many current processors). To show the interest of enforcing arc consistency using bitwise operations, we have introduced a new variant of AC3, denoted by AC3^{bit}, which can be used when constraints are (or can be) represented in extension. Importantly, we have also shown how to combine bitwise operations with residues, which happens to be quite useful when domains become large (approximately more than 300 values). The new algorithm, denoted by AC3^{bit+rm}, is quite robust. We do believe that, for solving binary instances, when constraints are given in extension or can be efficiently converted into extension, the generic algorithm MAC, embedding AC3^{bit+rm} is the most efficient approach. One rea-

son is that, like $MAC3^{rm}$, no maintenance of data structures is required upon backtracking by $MAC3^{bit+rm}$,

For the competition, $MAC3^{bit+rm}$ is the algorithm used by the solver *Abscon* 109. More precisely, it was used for binary instances involving constraints in extension and constraints in intention that can be converted efficiently into extension. For non binary constraints, the algorithm that we have adopted is $MGAC3^{rm}$ (but, for positive table constraints, we have used the algorithm described in the next section). Remark that the propagation scheme we used is variable-oriented with *dom* as a revision ordering heuristic [5]. We have also used the variant $R1$ [6] which allows avoiding useless revisions.

2.2 GAC for positive table constraints

In [20], we have introduced a new algorithm to establish GAC on positive table constraints. A table constraint is a constraint which is defined in extension by a set of tuples - when tuples are allowed (resp. disallowed) for the variables involved in the constraint, the table constraint is said positive (resp. negative). Table constraints are commonly used in configuration applications or applications related to databases.

The approach that we propose is a refinement of two approaches called GAC-valid and GAC-allowed. In order to find supports, GAC-valid iterates over valid tuples (i.e. tuples that can be built from the current domains of constraint variables) whereas GAC-allowed iterates over allowed tuples (i.e. combinations of values which are allowed by a constraint). Recall that a tuple is called a support if and only if it is valid and allowed. Roughly speaking, GAC-valid and GAC-allowed respectively correspond to GAC-schema-predicate and GAC-schema-allowed presented in [3].

The principle of the algorithm proposed in [20] is to avoid considering irrelevant tuples (when a support is looked for) by jumping over sequences of valid tuples containing no allowed tuple and over sequences of allowed tuples containing no valid tuple. It has been shown that the new schema (GAC-valid+allowed) admits on some instances a behaviour quadratic in the arity of the constraints whereas classical schemas (GAC-valid and GAC-allowed) admit an exponential behaviour.

On the practical side, the results that we have obtained demonstrate the robustness of GAC-valid+allowed. In fact, they are comparable to the ones obtained with a GAC-allowed+valid scheme [22] which allows to skip irrelevant allowed tuples from a reasoning about lower bounds on valid tuples. On the one hand, we believe that our model is simpler, and, importantly, can be easily grafted to any generic GAC algorithm. On the other hand, as the two approaches are different, it should be worthwhile combining them.

For the competition, $GAC3^{rm}$-valid+allowed is the algorithm used by the solver *Abscon* 109 for positive table constraints.

2.3 Existential SAC

It is natural to conceive algorithms to enforce partial forms of singleton arc consistency such as First-SAC, Last-SAC and Bound-SAC [16]. Indeed, it suffices to remove all values detected as arc inconsistent and bound values (only the minimal ones for First-SAC and the maximal ones for Last-SAC) detected as singleton arc inconsistent. When enforcing a constraint network P to be First-SAC, Last-SAC or Bound-SAC, one then obtains the greatest sub-network of P which is First-SAC, Last-SAC or Bound-SAC. However, enforcing Existential-SAC on a constraint network is meaningless. Either the network is (already) Existential-SAC, or the network is singleton arc inconsistent. It is then better to talk about checking Existential-SAC. An algorithm to check Existential-SAC will have to find a singleton arc consistent value in each domain. As a side-effect, if singleton arc inconsistent values are encountered, they will be, of course, removed. However, we have absolutely no guarantee about the network obtained after checking Existential-SAC due to the non-deterministic nature of this consistency.

In [14], an original approach to establish SAC has been proposed. The principle is to perform several runs of a greedy search, where at each step arc-consistency is maintained. As a result, the incrementality of arc-consistency algorithms is exploited but in a different manner that the one proposed in [1]. Unfortunately, a bound-SAC version of this approach does not seem to be feasible. Indeed, the main goal is to build branches (corresponding to greedy runs) as long as possible in order to benefit from incrementality, and potentially to find solutions during inference. When we are exclusively maintaining Bound-SAC via this approach the resultant propagation branches tend to be short, and therefore uneconomical. However, using a greedy approach to check Existential-SAC seems to be quite appropriate. In particular, it is straight forward to adapt the algorithm SAC3 [14] to guarantee ∃-SAC. This is what is done in [16].

For the competition, we have used ∃-SAC3 [16] at preprocessing for *Abscon 109 ESAC*.

3 Search Strategy

3.1 Search heuristics

The order in which variables are assigned by a backtracking search algorithm such as MAC has been recognized as a key issue for a long time. Using different variable ordering heuristics to solve a CSP can lead to drastically different results in terms of efficiency. Traditional dynamic variable ordering heuristics benefit from information about the current state of the search such as current domain sizes and current variable degrees. For instance, *dom/ddeg* [2] involves selecting first the variable with the smallest ratio current domain size to current dynamic degree. One limitation of this approach is that no information about previous states of the search is exploited.

In [4], inspired from [25–27], it is proposed to record such information by associating a counter with any constraint of the problem. These counters are used

as constraint weighting. Whenever a constraint is shown to be unsatisfied (during the constraint propagation process), its weight is incremented by 1. Using these counters, it is possible to define a new variable ordering heuristic, denoted *wdeg*, that gives an evaluation called weighted degree of any variable. The weighted degree of a variable V corresponds to the sum of the weights of the constraints involving V and at least another uninstantiated variable. In order to benefit, at the beginning of the search, from relevant information about current variable degrees, all counters are initially set to 1. Finally, combining weighted degrees and domain sizes yields *dom/wdeg*, an heuristic that selects first the variable with the smallest ratio current domain size to current weighted degree. The experimental results of [4, 13] show that MAC-*wdeg* and MAC-*dom/wdeg*, i.e., MAC combined with *wdeg* or *dom/wdeg* (called conflict-directed heuristics), is a generic backtracking approach which is quite robust to solve constraint networks.

Value-ordering heuristics have received less attention than variable ordering heuristics. Apart from *lexico*, *mc* [8] (see also [24, 9]) is certainly the most employed heuristic. It involves selecting the value which has the highest number of compatible values in the domains of other variables.

For the competition, we have used the variable ordering heuristic *dom/wdeg* and a static version [23] of the value ordering heuristic *mc*. Note that our solvers use a binary branching scheme. At each node of the search tree, two alternatives are successively tried: the first one corresponds to an assignment while the second one corresponds to the refutation of the assignment. A mechanism of restarts has been used (see below). Whatever the instance, the cutoff value is initially set to 10 backtracks and is increased at each new run by 50%. From one run to the next one, weighted degrees are not reinitialized.

3.2 Nogood Recording from Restarts

In [10], Gomez et al. have shown that runtime distributions of backtrack search algorithms exhibit on some instances a large variability in performance and are characterized by long tails with some infinite moments, called heavy-tailed phenomena. They also show that it is possible to boost search by introducing randomization and restarts. The principle is that if the search algorithm does not terminate within some number of allowed backtracks (or any other related criterion), referred as the cutoff value, the current run is stopped and a new run is started. Introducing randomization allows that runs behave differently. It can be used when breaking ties of variables to be selected, for example, and initialized with a random seed associated with each run. It is important to note that the cutoff value can be updated from one run to the next one. In particular, when it is systematically increased, the completeness of the backtrack search algorithm is preserved.

Using weighted degrees of variables is an alternative to randomization. Indeed, it suffices to keep the weighted degrees from one run to the next one. When restarting, one can expect to solve the instance with more facility when the hard part of the instance, i.e. the back-door, do correspond to variables with highest weighted degrees.

In [18], nogood recording is investigated for CSP within the randomization and restart framework. The goal is to avoid the same situations to occur (that is to say to explore several times the same part of the search space) from one run to the next ones. Nogoods are recorded when the current cutoff value is reached, i.e. before restarting the search algorithm. Such a set of nogoods is extracted from the last branch of the current search tree and exploited using the lazy data-structure of watched literals originally proposed for SAT. We prove that the worst-case time complexity of extracting such nogoods at the end of each run is only $O(n^2d)$ where n is the number of variables of the constraint network and d the size of the greatest domain, whereas for any node of the search tree, the worst-case time complexity of exploiting these nogoods to enforce Generalized Arc Consistency (GAC) is $O(n|\mathscr{B}|)$ where $|\mathscr{B}|$ denotes the number of recorded nogoods. As the number of nogoods recorded before each new run is bounded by the length of the last branch, the total number of recorded nogoods is polynomial in the number of restarts. Interestingly, the minimization of the nogoods is envisioned with respect to an inference operator ϕ, and it is possible to directly identify some nogoods that cannot be minimized. For $\phi = AC$ (i.e. for MAC), the worst-case time complexity of extracting minimal nogoods is slightly increased to $O(en^2d^3)$ where e is the number of constraints of the network. Experimentations over a wide range of CSP instances demonstrate the effectiveness of this approach. Recording nogoods (and in particular, minimal nogoods) from restarts significantly improves the robustness of the solver.

For the competition, we have used nogood recording from restarts.

3.3 Transposition Tables

In [19], we provided the proof of concept of the exploitation, for constraint satisfaction, of a well-known technique widely used in search: pruning from transpositions. This has not been addressed so far since, in CSP, contrary to search, two branches of a search tree cannot lead to the same state (that is to say the same domains for each variable of a given constraint network). This led us to define some reduction operators that keep partial information from each node of the search tree, sufficient to detect some nodes that do not need to be explored. We actually addressed the theoretical and practical aspects of how to exploit these operators in terms of equivalence between nodes.

Note that one can associate a constraint network with each node of a search tree. The reduction operator we used for the competition (called ρ^{red}), extracts a constraint sub-network from each node proved to be the root of an unsatisfiable sub-tree. Theses sub-networks are recorded in a so-called transposition table. The reduction operator removes some selected variables with either a singleton domain involved in constraints binding at most one non-singleton domain variable or with a domain that remains unchanged (after taking a set of decisions and applying an inference operator). Interestingly enough, when a constraint network P'' is extracted with the ρ^{red} operator from a binary CN P', variables of P'' satisfy the following property : $1 < |dom^{P'}(X)| < |dom^{P}(X)|$ where P is the initial constraint network.

The transposition table is implemented as a hash table, and before expanding a new node we just check if the current constraint network (associated with the current node) is equivalent (or not) to another one previously recorded in the transposition table. The hash-key is the concatenation of pairs (id,dom) where id is a unique integer associated with each variable and dom the domain of the variable itself represented as a bit vector. This approach allows us to dynamically break some kinds of symmetries (e.g. neighborhood interchangeability) and prune similar states of the search space. On some series, when no inference is performed using this approach, the extraction procedure is stopped and the memory (allocated to the transposition table) is released.

For the competition, we have used transposition tables for constraint satisfaction.

4 What about Max-CSP?

In order to participate to the part of the competition dedicated to Max-CSP, we have implemented in *Abscon* a variant of the PFC-MRDAC algorithm [12]. This variant lies between PFC-MRDAC and PFC-MPRDAC [11].

For preprocessing, we have used a tabu search algorithm in order to obtain an initial lower bound of good quality. For (complete) search, we have used our PFC-MRDAC variant and exploited nogood recording from restarts. The variable ordering heuristic was *dom/wdeg* while the value ordering heuristic selects the value with the smallest number of inconsistencies computed during filtering (as in [12, 11]).

Unlike *AbsconMax* 109 PFC, *AbsconMax* 109 EPFC integrates a mechanism similar to existential SAC and adapted to PFC. Also, last-conflict based reasoning [17] has been used.

5 Conclusion

In this paper, we have presented the strategies of the two solvers that we have submitted to the second CSP solver competition. They are derived from *Abscon*, a generic constraint programming platform which has been developed in Java (J2SE 5.0). You will find:

- the executable at:
  http://www.cril.univ-artois.fr/~lecoutre/research/tools/abscon.html
- the results obtained at the 2006 competition at:
  http://www.cril.univ-artois.fr/CPAI06

Acknowledgements

This paper has been supported by the CNRS, the “programme COCOA de la Région Nord/Pas-de-Calais” and by the “IUT de Lens”.

References

1. C. Bessiere and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, pages 54–59, 2005.
2. C. Bessiere and J. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of CP'96*, pages 61–75, 1996.
3. C. Bessiere and J. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, 1997.
4. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
5. F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the Constraint Satisfaction Problem. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 29–43, 2004.
6. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Support inference for generic filtering. In *Proceedings of CP'04*, pages 721–725, 2004.
7. R. Debruyne and C. Bessiere. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
8. D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of IJCAI'95*, pages 572–578, 1995.
9. P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of ECAI'92*, pages 31–35, 1992.
10. C.P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100, 2000.
11. J. Larrosa and P. Meseguer. Partition-Based lower bound for Max-CSP. *Constraints*, 7:407–419, 2002.
12. J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107(1):149–163, 1999.
13. C. Lecoutre, F. Boussemart, and F. Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Proceedings of ICTAI'04*, pages 549–557, 2004.
14. C. Lecoutre and S. Cardon. A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI'05*, pages 199–204, 2005.
15. C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, pages 125–130, 2007.
16. C. Lecoutre and P. Prosser. Maintaining singleton arc consistency. In *Proceedings of CPAI'06*, pages 47–61, 2006.
17. C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Last conflict-based reasonning. In *Proceedings of ECAI'06*, pages 133–137, 2006.
18. C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Recording and minimizing nogoods from restarts. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 1:147–167, 2007.
19. C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Transposition Tables for Constraint Satisfaction. In *Proceedings of AAAI'07*, pages 243–248, 2007.
20. C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, pages 284–298, 2006.
21. C. Lecoutre and J. Vion. Enforcing arc consistency using bitwise operations. *Submitted*, 2007.
22. O. Lhomme and J.C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of AAAI'05*, pages 405–410, 2005.

23. D. Mehta and M.R.C. van Dongen. Static value ordering heuristics for constraint satisfaction problems. In *Proceedings of CPAI'05 workshop held with CP'05*, pages 49–62, 2005.
24. S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint-satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.
25. P. Morris. The breakout method for escaping from local minima. In *Proceedings of AAAI'93*, pages 40–45, 1993.
26. B. Selman and H. Kautz. Domain-independent extensions to GSAT: solving large structured satisfiability problems. In *Proceedings of IJCAI'93*, pages 290–295, 1993.
27. J.R. Thornton. *Constraint weighting local search for constraint satisfaction.* PhD thesis, Griffith University, Australia, 2000.

Sugar: A CSP to SAT Translator Based on Order Encoding

Naoyuki Tamura and Mutsunori Banbara

Information Science and Technology Center, Kobe University, JAPAN
{tamura,banbara}@kobe-u.ac.jp

Abstract. This paper gives some details on the implementation of *sugar* constraint solver submitted to the Second International CSP Solver Competition. The sugar solver solves a finite linear CSP by translating it into a SAT problem using the *order encoding* method and then solving the translated SAT problem with the MiniSat solver. In the order encoding method, a comparison $x \le a$ is encoded by a different Boolean variable for each integer variable x and integer value a.

1 Introduction

This paper gives some details on the implementation of *sugar* constraint solver submitted to the Second International CSP Solver Competition.

The sugar solver solves a finite linear CSP by translating it into a SAT problem by using *order encoding* method [1] and then solving the translated SAT problem by the MiniSat solver [2].

The method of the order encoding is basically the same with the one used for Job-Shop Scheduling Problems by Crawford and Baker in [3] and studied by Soh, Inoue, and Nabeshima in [4–6]. It encodes a comparison $x \le a$ by a different Boolean variable for each integer variable x and integer value a.

The benefit of this encoding is the natural representation of the order relation on integers. Axiom clauses with two literals, such as $\{\neg(x \le a), x \le a+1\}$ for each integer a, represent the order relation of an integer variable x. Clauses, for example $\{x \le a, \neg(y \le a)\}$ for each integer a, can be used to represent the constraint among integer variables, i.e. $x \le y$.

2 Order encoding

The order encoding uses Boolean variables p_{xi} meaning $x \le i$ for each CSP variable x and each integer constant i ($\ell(x)-1 \le i \le u(x)$) where $\ell(x)$ and $u(x)$ are the lower and upper bounds of x respectively[1].

[1] $p_{x\ell(x)-1}$ and $p_{xu(x)}$ are redundant because they are always false and true respectively. However, we use them for simplicity of the discussion.

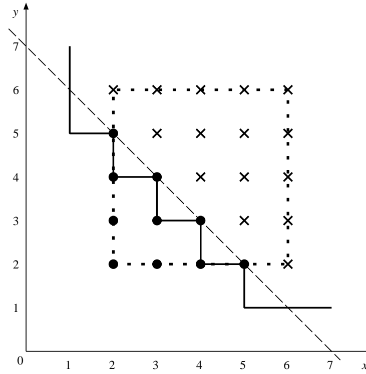

Fig. 1. Encoding $x + y \le 7$

Consider an example of encoding $x + y \le 7$ when $x, y \in \{2, 3, 4, 5, 6\}$. The following 12 Boolean variables are used to encode the example.

$$\begin{array}{cccccc} p_{x1} & p_{x2} & p_{x3} & p_{x4} & p_{x5} & p_{x6} \\ p_{y1} & p_{y2} & p_{y3} & p_{y4} & p_{y5} & p_{y6} \end{array}$$

The following clauses are used as axioms representing the bounds and the order relation for each CSP variable x.

$$\begin{array}{ll} \neg p_{x\,\ell(x)-1} & \\ p_{x u(x)} & \\ \neg p_{x\,i-1} \lor p_{xi} & (\ell(x) \le i \le u(x)) \end{array}$$

Therefore, the following 14 clauses are required for the example.

$$\begin{array}{ccccc} & \neg p_{x1} & & p_{x6} & \\ \neg p_{x1} \lor p_{x2} & \neg p_{x2} \lor p_{x3} & \neg p_{x3} \lor p_{x4} & \neg p_{x4} \lor p_{x5} & \neg p_{x5} \lor p_{x6} \end{array}$$
$$\text{(similar clauses for } y\text{)}$$

Constraints are encoded into clauses representing conflict regions instead of conflict points. When all points $(x_1, \ldots, x_n)$ in the region $i_1 < x_1 \le j_1, \ \ldots, \ i_n < x_n \le j_n$ violate the constraint, the following clause is added.

$$p_{x_1 i_1} \lor \neg p_{x_1 j_1} \lor \cdots \lor p_{x_n i_n} \lor \neg p_{x_n j_n}$$

Therefore, the following 5 clauses are used to encode $x + y \le 7$ (Fig.1).

$$p_{x1} \lor p_{y5} \qquad p_{x2} \lor p_{y4} \qquad p_{x3} \lor p_{y3} \qquad p_{x4} \lor p_{y2} \qquad p_{x5} \lor p_{y1}$$

When a_i's are non-zero integer constants, c is an integer constant, and x_i's are mutually distinct integer variables, any finite linear comparison $\sum_{i=1}^{n} a_i\, x_i \le c$

can be encoded into the following CNF formula [1].

$$\bigwedge_{\sum_{i=1}^{n} b_i = c-n+1} \bigvee_{i} (a_i\, x_i \le b_i)^{\#}$$

Parameters b_i's range over integers satisfying $\sum_{i=1}^{n} b_i = c-n+1$ and $\ell(a_i x_i)-1 \le b_i \le u(a_i x_i)$ for all i where functions ℓ and u give the lower and upper bounds of the given expression respectively. The translation $()^{\#}$ is defined as follows.

$$(a\,x \le b)^{\#} \;\equiv\; \begin{cases} x \le \left\lfloor \dfrac{b}{a} \right\rfloor & (a > 0) \\ \neg\left(x \le \left\lceil \dfrac{b}{a} \right\rceil - 1\right) & (a < 0) \end{cases}$$

3 System Description of Sugar

Sugar is a CSP to SAT solver based on the order encoding. It consists of the front-end Perl program and the encoder program written in Java[2]. The MiniSat 1.4 [2] is used as the backend SAT solver in the submitted version.

CSP instances are encoded into SAT instances in the following ways.

Encoding m-ary linear comparisons: As described in the previous section, comparisons of the form $\sum_{i=1}^{m} a_i x_i \le b$ can be encoded into $O(d^{m-1})$ clauses in general where d is the domain size.

However, it is possible to reduce the number of integer variables in each comparison to at most three, by introducing new integer variables. Therefore, each comparison $\sum_{i=1}^{m} a_i\, x_i \le b$ can be encoded by at most $O(m\, d^2)$ clauses even when $m \ge 4$.

Encoding other expressions: Expressions other than $\sum a_i x_i \le b$ are encoded to SAT formulas by using the conversion described in the Fig.2 where E div c and E mod c are integer quotient and remainder of E divided by an integer constant c.

Note that non-linear expressions such as $x \times y$ can not be handled by the sugar program submitted to the competition.

Keeping clausal form: When encoding a clause of CSP to SAT, the encoded formula is no more a clausal form in general. As it is well known, introduction of new Boolean variables is useful to solve this problem.

Consider an example of encoding a clause $\{x - y \le -1, -x + y \le -1\}$ when $x, y \in \{0, 1, 2\}$. Comparisons $x - y \le -1$ and $-x + y \le -1$ are converted into $S_1 = (x \le -1 \vee \neg(y \le 0)) \wedge (x \le 0 \vee \neg(y \le 1)) \wedge (x \le 1 \vee \neg(y \le 2))$ and

[2] The package is available at `http://bach.istc.kobe-u.ac.jp/sugar/`

Expression	Replacement	Extra condition
$E < F$	$E+1 \le F$	
$E = F$	$(E \le F) \land (E \ge F)$	
$E \ne F$	$(E < F) \lor (E > F)$	
$\max(E,F)$	x	$(x \ge E) \land (x \ge F) \land ((x \le E) \lor (x \le F))$
$\min(E,F)$	x	$(x \le E) \land (x \le F) \land ((x \ge E) \lor (x \ge F))$
$\mathrm{abs}(E)$	$\max(E,-E)$	
E div c	q	$(E = c\,q + r) \land (0 \le r) \land (r < c)$
E mod c	r	$(E = c\,q + r) \land (0 \le r) \land (r < c)$

Fig. 2. Encoding expressions other than $\sum a_i x_i \le b$

$S_2 = (\neg(x \le 2) \lor y \le 1) \land (\neg(x \le 1) \lor y \le 0) \land (\neg(x \le 0) \lor y \le -1)$ respectively. Expanding $S_1 \lor S_2$ generates 9 clauses. However, by introducing new Boolean variables p and q, we obtain the following seven clauses.

$$
\begin{array}{ccc}
 & \{p,q\} & \\
\{\neg p, x \le -1, \neg(y \le 0)\} & \{\neg p, x \le 0, \neg(y \le 1)\} & \{\neg p, x \le 1, \neg(y \le 2)\} \\
\{\neg q, \neg(x \le 2), y \le 1\} & \{\neg q, \neg(x \le 1), y \le 0\} & \{\neg q, \neg(x \le 0), y \le -1\}
\end{array}
$$

Encoding extensional constraints: Extensional constraints with conflict tuples and support tuples are encoded by a simple way in the submitted version of sugar.

Conflict tuples $\{(a_1,b_1),\ \dots,\ (a_n,b_n)\}$ for variables (x,y) can be expressed by the following formula.

$$\neg(x = a_1 \land y = b_1) \land \cdots \land \neg(x = a_n \land y = b_n)$$

This formula is encoded into the following n clauses.

$$
\begin{array}{c}
\{x \le a_1 - 1,\ \neg(x \le a_1),\ y \le b_1 - 1,\ \neg(y \le b_1)\} \\
\cdots \\
\{x \le a_n - 1,\ \neg(x \le a_n),\ y \le b_n - 1,\ \neg(y \le b_n)\}
\end{array}
$$

Support tuples $\{(a_1,b_1),\ \dots,\ (a_n,b_n)\}$ for variables (x,y) can be expressed by the following formula.

$$(x = a_1 \land y = b_1) \lor \cdots \lor (x = a_n \land y = b_n)$$

This formula is encoded into the following $4n+1$ clauses by introducing n new Boolean vairables s_i.

$$
\begin{array}{cccc}
 & \{s_1, s_2, \dots, s_n\} & & \\
\{\neg s_1, x \le a_1\} & \{\neg s_1, \neg(x \le a_1 - 1)\} & \{\neg s_1, x \le b_1\} & \{\neg s_1, \neg(x \le b_1 - 1)\} \\
 & \cdots & & \\
\{\neg s_n, x \le a_n\} & \{\neg s_n, \neg(x \le a_n - 1)\} & \{\neg s_n, x \le b_n\} & \{\neg s_n, \neg(x \le b_n - 1)\}
\end{array}
$$

4 Conclusion

In this paper, we have described some details on the implementation of sugar constraint solver submitted to the Second International CSP Solver Competition. The sugar solver solves a finite linear CSP by translating it into a SAT problem by using order encoding method and then solving the translated SAT problem by the MiniSat solver. Although the system is still under development, we hope it gives some research directions for CSP to SAT encoding systems.

Acknowledgments

We would like to give thanks to the competition organizers for their efforts and Katsumi Inoue, Hidetomo Nabeshima, Takehide Soh, and Shuji Ohnishi for their helpful suggestions.

References

1. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. In: Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP 2006). (2006) 590–603
2. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003). (2003) 502–518
3. Crawford, J.M., Baker, A.B.: Experimental results on the application of satisfiability algorithms to scheduling problems. In: Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94). (1994) 1092–1097
4. Soh, T., Inoue, K., Banbara, M., Tamura, N.: Experimental results for solving job-shop scheduling problems with multiple SAT solvers. In: Proceedings of the 1st International Workshop on Distributed and Speculative Constraint Processing (DSCP'05). (2005)
5. Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M., Tamura, N.: A competitive and cooperative approach to propositional satisfiability. Discrete Applied Mathematics (2006) (to appear).
6. Nabeshima, H., Soh, T., Inoue, K., Iwanuma, K.: Lemma reusing for SAT based planning and scheduling. In: Proceedings of the International Conference on Automated Planning and Scheduling 2006 (ICAPS'06). (2006) 103–112

Introducing $buggy_{2-5}$ and $buggy^{s}_{2-5}$

M.R.C. van Dongen

University College Cork

Abstract. This paper presents a brief introduction to two solvers, $buggy_{2-5}$ and $buggy^{s}_{2-5}$, which were submitted to the binary extensional and binary intensional categories of the Second International CSP Solver Competition. Both solvers are a combination of preprocessing followed by MAC search. The preprocessing consists of domain squashing and enforcing weak k-singleton arc consistency.

1 Main Description

This position paper briefly describes the two solvers, $buggy_{2-5}$ and $buggy^{s}_{2-5}$, which were submitted to the Second International CSP Solver Competition for the binary extensional and binary intensional categories.

The paper is deliberately kept to a minimum. The main reason is that there is not much that can be said about the solvers. Another reason, which is related to the first, is that the basic solver and its data types are currently undergoing a major re-implementation. As a result, the basic solver, which underlies $buggy_{2-5}$ and $buggy^{s}_{2-5}$, is only half finished. What is worse, it is not clear if $buggy_{2-5}$ and $buggy^{s}_{2-5}$ are bug-free.

The solvers are best described as follows. Both do some preprocessing, enforce consistency and, if needed, they start a MAC search [Sabin and Freuder, 1994]. The only difference between the solvers is the level of preprocessing. The remainder of this paper describes the local consistency which is enforced by the solvers, the preprocessing,[1] and the solution strategy.

2 Consistency

This section briefly describes the local consistency which is enforced by the solvers.

Enforcing local consistency means enforcing weak k-singleton arc consistency (weak k-SAC) [van Dongen, 2006] for increasing levels of k, $2 \leq k \leq 5$ using greedy search. Weak k-sac is equivalent to SAC [Debruyne and Bessière, 1997; 2001, Bessière and Debruyne, 2005, Lecoutre and Cardon, 2005, Prosser *et al.*, 2000] if $k = 1$ but stronger if $k > 1$. They start by enforcing arc consistency, followed by

[1] Here preprocessing should not be confused with converting the competition's `XML` format to the solvers' native format. this conversion was carried out at solution-time.

weak 2-SAC, weak 3-SAC, weak 4-SAC and weak 5-SAC. The difference between the solvers is that $buggy^s_{2-5}$ may enforce higher levels of consistency. Both solvers terminate in case of an inconsistency. In the process of establishing weak k-SAC it is possible that a solution is found, in which case the algorithms terminate. The algorithms for enforcing weak k-SAC [van Dongen, 2006] are closely related to Lecoutre and Cardon's algorithms for establishing SAC [Lecoutre and Cardon, 2005].

3 Preprocessing

The preprocessing which is done by the solvers involves domain squashing [Gault and Jeavons, 2004]. Here it is assumed, without loss of generality, that the domains have unique values, allowing us to squash more values [Bulatov and Jeavons, 2003]. This preprocessing more or less amounts to eliminating some substitutable values. The domain squashing is performed after enforcing arc consistency. Since the domains squashing may be quite a large overhead if the domains are large, $buggy_{2-5}$ only smashes the extremal values, that is to say the smallest and largest values in the domains. The following paragraph spends a few more words on extremal value squashing. The same strategy is adopted by $buggy^s_{2-5}$, which then attempts to also smash the remaining values. It is allowed a fantastic magic number of 490 seconds maximum domain squashing time. For many instances much more is needed to ensure that no more squashing is possible. It turned out that its extra squashing enabled $buggy^s_{2-5}$ to solve two more instances than $buggy_{2-5}$ but at the price of much more solution time.

Squashing the extremal values is *very* effective, especially for the jobshop and openshop instances, since many (almost) extremal values, v, satisfy the property that if v is part of any solution then so is $v + 1$ or $v - 1$, in which case we can eliminate v. For some of the instances, including modified Radio Link Frequency Assignment Problem instances, squashing made the difference between solving the problem or not.

4 Solution Strategy

This section briefly describes the solution strategy of the solvers, each of which have a different objective. The main goal of $buggy_{2-5}$ is to find a solution. For $buggy^s_{2-5}$ this is different. Its main objective is to make the problem *inverse consistent* [Freuder and Elfe, 1996], i.e. remove the values which do not participate in any solution. In the process of making the problem inverse consistent, $buggy^s_{2-5}$ outputs the first solution if there is one.

Both solvers terminate as soon as they decide the problem is unsatiafiable. They enforce arc consistency using AC-3 with residues [Lecoutre and Hemery, 2007]. The following two sections describe how the solvers work for satisfiable problem instances.

4.1 $\boldsymbol{buggy_{2-5}}$

In the case of $buggy_{2-5}$ the algorithm starts by enforcing arc consistency. Next it starts by enforcing weak 2-SAC, weak 3-SAC, weak 4-SAC, and weak 5-SAC, stopping if it finds a solution or finds the problem unsatisfiable and starting a MAC search otherwise. During MAC search variables are ordered using a combination of variable ordering heuristics. The main component of the variable ordering is used which minimises the ratio of domain to weighted-degree[Boussemart *et al.*, 2004]. The branching scheme which is used is k-way branching [Mitchell, 2003].

4.2 $\boldsymbol{buggy^s_{2-5}}$

The strategy of $buggy^s_{2-5}$ is different. It enforces arc consistency followed by weak 2-SAC, weak 3-SAC, and so on. In the process of doing this, it marks all values which participate to a solution and terminates if there are no more values left which do not participate to a solution.

5 Future Work

The solver's data types are still in a development state. They still are not ideal, let alone optimal. It is expected that much can be improved. As soon as the implementation is finalised, it is the author's intention to write a paper describing the data types and how they affect the implementation of the algorithms which are built on top of them. Work related to the domain squashing is underway.

Acknowledgment

Bessière *et al.* also define the notion of k-singleton arc consistency and the equivalent of weak k-singleton arc consistency. In their paper, weak k-SAC is is called k-SAC(v). At the moment of writing [van Dongen, 2006] the author was unaware of this work.

Bibliography

C. Bessière and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 54–59, 2005.

C. Bessière, R. Coletta, and T. Petit. A generic framework for learning implied constraints.

F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In Ramon López de Mántaras and Lorenza Saitta, editors, *Proceedings of the Sixteenth Eureopean Conference on Artificial Intelligence*, pages 146–150, 2004.

A. Bulatov and P.G. Jeavons. An algebraic approach to multi-sorted constraints. In F. Rossi, editor, *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming, CP'2003*, pages 183–198, 2003.

R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In M.E. Pollack, editor, *Proccedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 412–417, 1997.

R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.

E.C. Freuder and C.D. Elfe. Neighborhood inverse consistency preprocessing. In W.J. Clancey and D. Weld, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 202–208, 1996.

R. Gault and P.G. Jeavons. Implementing a test for tractability. *Journal of Constraints*, 9:139–160, 2004.

C. Lecoutre and S. Cardon. A greedy approach to establish singleton arc consistency. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, 2005.

C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 125–130, 2007.

David G. Mitchell. Resolution and constraint satisfaction. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming, CP'2003*, pages 555–569, 2003.

P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In R. Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming, CP'2000*, pages 353–368, 2000.

D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 125–129. John Wiley and Sons, 1994.

M.R.C. van Dongen. Beyond singleton arc consistency. In *Proceedings of the Seventeenth European Conference on Artificial Intelligence*, 2006.

CSP4J: a Black-box CSP Solving API for Java

http://cspfj.sourceforge.net/

Julien Vion

CRIL-CNRS FRE 2499,
Université d'Artois
Lens, France
vion@cril.univ-artois.fr

Abstract. We propose an API, namely CSP4J for *Constraint Satisfaction Problem for Java*, that aims to solve a CSP problem part of any Java application. CSP4J is distributed online using the LGPL license [16]. We intend our API to be a "black box", i.e. to be able to solve any problem without tuning parameters or programming complex constraints. We intend CSP4J to move towards the Graal of AI: the ability to solve any problem in a reasonable time with a minimal expertise from the user.

1 Introduction

Many problems arising in the computing industry involve constraint satisfaction as an essential component. Such problems occur in numerous domains such as scheduling, planning, molecular biology and circuit design. Problems involving constraints are usually NP-Complete and need, if able, powerful Artificial Intelligence techniques to be solved in reasonable time. Problems involving constraints are usually represented by so-called constraint networks. A constraint network is simply composed of a set of variables and of a set of constraints. Finding a solution to a constraint network involves assigning a value to each variable such that all constraints are satisfied. The Constraint Satisfaction Problem (CSP) is the task to determine whether or not a given constraint network, also called CSP instance, is satisfiable. The Maximal Constraint Satisfaction Problem (Max-CSP) is the task to find a solution that satisfies as much constraints as possible, and eventually proving that a given solution is optimal, i.e. no other solution exists that can satisfy more constraints than the given one.

CSP4J has been in development since 2005 and is quickly acquiring maturity. We intend our API to be a "black box" solving CSP and Max-CSP. Given this assumption, CSP4J does not focus on problem-specific global constraints, although the Object design of CSP4J permits to develop such constraints. For example, CSP4J is shipped with the well known "all-different" global constraint including a simple specific propagator.

CSP4J proposes powerful engines based on the latest refinements of current research in AI.

- **MGAC**, a complete solver based on the well known MGAC-$dom/wdeg$ algorithm [13]. It can solve any CSP in a complete way: if given enough time, a feasible solution, if it exists, will be found. If no solution exists, this engine is able to prove it.

- **MCRW**, an incomplete local search solver based on the *Min-Conflicts Hill-Climbing with Random Walks* algorithm [11]. This engine can be used to solve optimization problems that can be formalized as a Max-CSP problem in an "anytime" way: the algorithm can be stopped after a given amount of time, and the best solution found so far will be given.
- **Tabu**, an incomplete local search solver performing a Tabu search [3]. **Tabu** have similar characteristics as **MCRW**.
- **WMC**, an incomplete local search solver based on the *Breakout Method* [12], that show similar characteristics as **MCRW** and **Tabu**, although not really suited for Max-CSP problems.
- **Combo**, a complete solver based on the hybridization of MGAC-$dom/wdeg$ with WMC [21].

In order to prove the interest of our library, we developed a few test applications, all distributed online using the GPL license [15]. One of these test applications is dedicated to participate to the International CSP Solver Competitions, and tries to solve problems delivered under the XCSP 2.0 format [18]. This solver participated to the two first International CSP Solver Competitions. This "competitor" version of CSP4J is shipped with a particular constraint called "Predicate Constraint", that compiles intentional constraints as defined by the XCSP 2.0 format.

Other example applications include :

- a random problem generator and solver, which is very useful to benchmark algorithms and computers,
- a Minimal Unsatisfiable Core (MUC) extractor, able to extract a minimally unsatisfiable set of variable and constraints from a larger incoherent CSP,
- an Open-Shop solver, able to find feasible and optimal solutions to Open-Shop problems
- last but not least, a Sudoku solver

2 Solving a CSP in a black-box

In order to be able to solve any kind of problem, CSP4J focuses on two main topics: genericity and flexibility. Flexibility was obtained by the choice of an object-oriented language for its development: Java 5. The object-oriented conception of CSP4J permits to model problems using a fully object-oriented scheme.

A few classes and interfaces are in the heart of CSP4J, as described by the UML diagram on Figure 1: The $Problem$, $Variable$ and $Constraint$ classes define a CSP instance. The Solver interface is implemented by all engines provided with CSP4J.

The $Variable$ class: It can be used directly through its constructor. $domain$ simply contains the domain of the variable (i.e. the set of value the variable can take its value in) under the form of an array of integers.

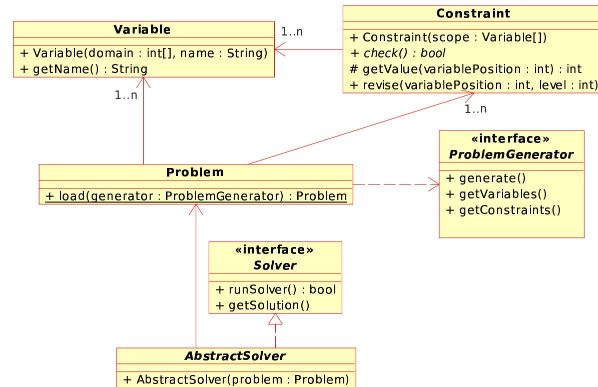

Fig. 1. UML sketch of CSP4J

```
public final class DTPConstraint extends Constraint {

  final private int duration0;
  final private int duration1;

  public DTPConstraint(final Variable[] scope,
      final int duration0, final int duration1) {
    super(scope);
    this.duration0 = duration0;
    this.duration1 = duration1;
  }

  @Override
  public boolean check() {
    final int value0 = getValue(0);
    final int value1 = getValue(1);

    return (value0 + duration0 < value1
      || value1 + duration1 < value0);
  }

}
```

Listing 1.1. The Disjunctive Temporal Constraint

```
final Predicate predicate = new Predicate();
predicate.setExpression("(X0 + X1 < X2) || (X2 + X3 < X0)");
predicate.setParameters("int X0 int X1 int X2 int X3");

final PredicateConstraint dtpConstraint =
  new PredicateConstraint(scope, predicate);
dtpConstraint.setParameters(scope[0].getName() + " " + duration0
  + " " + scope[1].getName() + " " + duration1);
try {
  dtpConstraint.compileParameters();
} catch (FailedGenerationException e) {
  System.err.println("Failed to compile constraint");
  System.exit(1);
}
```

Listing 1.2. Defining a DT Constraint with predicates

The $Constraint$ class . It consists of an abstract class that must be extended to define the constraints that define the problem. In particular, the abstract method $check()$ must be overridden. $check()$ must return whether the current tuple is allowed by the constraint. The current tuple is accessible through the $getValue(int\ variablePosition)$ method, $variablePosition$ corresponding to the position of the variable in the constraint, as defined by the *scope* in the constructor. Listing 1.1 gives an example on how to easily define a constraint. Alternatively, one could use the $PredicateConstraint$ to define such a constraint as shown on Listing 1.2. Notice, however, that source code from $PredicateConstraint$ is released amongst the Competitor test application for CSP4J under the GPL, and not directly with the CSP4J API.

If desired, one may also override the $revise(int\ variablePosition, int\ level)$ method in order to develop constraint-specific propagators. If not, a revision using the AC3rm algorithm (see section 3.1) is done.

The $Problem$ class: It defines a CSP. The $ProblemGenerator$ interface permits to define classes that will be intended to generate problems to solve. To define a problem to be solved with CSP4J, one has to implement the ProblemGenerator interface. An instance of the problem is then loaded by calling the static method $Problem.load(ProblemGenerator)$. The $ProblemGenerator$ interface only defines three methods.

- $generate()$: this method is called upon loading of the Problem, it can be used to create constraints and variables
- $Collection\ \langle Variable \rangle\ getVariables()$: this method must return the set of variables that defines the problem
- $Collection\ \langle Constraint \rangle\ getConstraints()$: this method must return the set of constraints that defines the problem

The $Solver$ interface and the $AbstractSolver$ helper class: These permit to define additional engines for CSP4J. The MGAC and MCRW engines that come with CSP4J

Algorithm 1: revise-rm(X: Variable): Boolean

1. $domainSize \leftarrow |dom(X)|$
2. **foreach** $C \mid X \in vars(C)$ **do**
3.   **foreach** $v \in dom(X)$ **do**
4.   **if** $supp[C, X, v]$ *is valid* **then continue**
5. $tuple \leftarrow seekSupport(C, X_v)$
6.   **if** $tuple = \top$ **then** remove v from $dom(X)$
7.   **else**
8.   **foreach** $Y \in vars(C)$ **do**
9. $supp[C, Y, tuple[Y]] \leftarrow tuple$

            `/* for wdeg: */`

10.   **if** $dom(X) = \emptyset$ **then** $wght[C]$++
11. **return** $domainSize \neq |dom(X)|$

Algorithm 2: GAC3rm ($P = (\mathscr{X}, \mathscr{C})$: CN)

1. $Q \leftarrow \mathscr{X}$
2. **while** $Q \neq \emptyset$ **do**
3.   pick X from Q
4.   **foreach** $Y \in \mathscr{X} \mid \exists C \in \mathscr{C} \mid X \in C \wedge Y \in C \wedge X \neq Y$ **do**
5.   **if** $revise\text{-}rm(Y)$ **then**
6.   **if** $dom(Y) = \emptyset$ **then return false**
7. $Q \leftarrow Q \cup Y$

are classes that extends $AbstractSolver$. The $runSolver()$ method launches the resolution and returns **true** if the problem is satisfiable and **false** if it is not. The method $getSolution()$ returns the last found solution (the best solution found so far for Max-CSP). To use CSP4J as an incomplete Max-CSP solver, one has to launch $runSolver()$ from a thread to control its execution.

To illustrate how CSP4J can be used in a Java application, Listing 1.3 defines the well-known Pigeons problem, using a clique of $different$ constraints defined as predicates. Once the problem has been defined and loaded, the solving process can be launched in a few lines of code, as shown on Listing 1.4.

3 Under the hood

3.1 The MGAC engine

Generalized Arc Consistency guarantees the existence of a support of each value in each constraint. Establishing Generalized Arc Consistency on a given network P involves removing all generalized arc inconsistent values.

Many algorithms establishing Arc Consistency have been proposed in the literature. We believe that GAC3rm [8] is a very efficient and robust one. GAC3rm is a refinement

```
public class Pigeons implements ProblemGenerator {
  final private int size;
  final private List<Variable> variables;
  final private Collection<Constraint> constraints;
  final private Predicate predicate;

  public Pigeons(int size) {
    this.size = size;
    variables = new ArrayList<Variable>(size);
    constraints = new ArrayList<Constraint>();
    predicate = new Predicate();
    predicate.setExpression("X0 != X1");
    predicate.setParameters("int X0 int X1");
  }

  public void generate() throws FailedGenerationException {
    final int[] domain = new int[size - 1];
    for (int i = size - 1; --i >= 0;) { domain[i] = i; }
    for (int i = size; --i >= 0;) {
      variables.add(new Variable(domain, "V" + i));
    }
    for (int i = size; --i >= 0;) {
      for (int j = size; --j >= i + 1;) {
        constraints.add(diff(variables.get(i), variables
            .get(j)));
      }
    }
  }

  private Constraint diff(final Variable var1,
    final Variable var2) throws FailedGenerationException {
    PredicateConstraint constraint = new PredicateConstraint(
        new Variable[] { var1, var2 }, predicate);
    constraint.setParameters(var1.getName() + " "
        + var2.getName());
    constraint.compileParameters();
    return constraint;
  }

  public Collection<Variable> getVariables() {
    return variables;
  }

  public Collection<Constraint> getConstraints() {
    return constraints;
  }
}
```

Listing 1.3. The Pigeons problem

```
public static void main() throws
  FailedGenerationException, IOException {
  final Problem problem = Problem.load(10);
  final Solver solver = new MGAC(problem);
  final boolean result = solver.runSolver();
  System.out.println(result);
  if (result) {
    System.out.println(solver.getSolution());
  }
}
```

Listing 1.4. Solving the Pigeons-10 problem

Algorithm 3: MGAC(P = $(\mathscr{X},\mathscr{C})$: CN, $maxBT$: Integer): Boolean

1 **if** $maxBT < 0$ **then throw** $Expiration$
2 **if** $\mathscr{X} = \emptyset$ **then return true**
3 **select** $(X, v) \mid X \in \mathscr{X} \wedge a \in dom(X)$
4 $P' \leftarrow GACrm(P|_{X=a})$
5 **if** $P' \neq \perp \wedge MGAC(P' \backslash X, maxBT)$ **then return true**
6 $P' \leftarrow GACrm(P|_{X \neq a})$
7 **return** $P' \neq \perp \wedge MGAC(P', maxBT - 1)$

of GAC3 [9]. They both admit a worst-case time complexity of $O(er^3d^{r+1})$. GAC2001 [1] admits a worst-case time complexity of $O(er^2d^r)$ and has been proved to be an optimal algorithm for establishing Generalized Arc Consistency.

The GAC3rm algorithm is described in Algorithm 2. Every variable of the CN is put in a queue in order to be revised one by one using Algorithm 1. If an effective revision is done (i.e. at least one value is removed from the variable), all neighbors of the variable are put in the queue. The algorithm continues until a fix-point is reached, i.e. no more value can be removed in the CN. A neighbor variable is one that shares at least one constraint with the current variable.

Residual supports ($supp[C, X, v]$) are used during the revision in order to speed up the search. Contrary to GAC2001, if the residue is no longer valid, the search for a valid tuple is restarted from scratch, which allow us to keep the residues from one call to another, even after a backtrack. Although GAC3rm by itself is not optimal, [8] shows that maintaining GAC3rm during search (see below) is more efficient than maintaining GAC2001.

The MGAC algorithm [13] aims at solving a CSP instance and performs a depth-first search with backtracking while maintaining (generalized) arc consistency. More precisely, at each step of the search, a variable assignment is performed followed by a filtering process called constraint propagation which corresponds to enforcing generalized arc-consistency.

Recent implementations of MGAC use a binary (2-way) branching scheme [6]: at each node of the search tree, a variable X is selected, a value $a \in$ dom(X) is selected,

Algorithm 4: initP($P = (\mathscr{X}, \mathscr{C})$: CN): Integer

1 **foreach** $X \in \mathscr{X}$ **do**
2   **select** $v \in dom(X) \mid countConflicts(P|_{X=v})$ *is minimal*
3     $P \leftarrow P|_{X=v}$
4 **return** $countConflicts(P)$

and two edges are considered: the first one corresponds to $X = a$ and the second one to $X \neq a$.

Algorithm 3 corresponds to a recursive version of the MGAC algorithm (using binary branching). A CSP instance is solved by calling the $MGAC$ function: it returns **true** iff the instance is satisfiable. $P|_{X=a}$ denotes the constraint network obtained from P by restricting the domain of X to the singleton $\{a\}$ whereas $P|_{X\neq a}$ denotes the constraint network obtained from P by removing the value a from the domain of X. $P \backslash X$ denotes the constraint network obtained from P by removing the variable X.

The heuristic that allows the selection of the pair (X, a) has been recognized has a crucial issue for a long time. Using different variable ordering heuristics to solve a CSP instance can lead to drastically different results in terms of efficiency.

In [2], it is proposed to associate a counter, denoted $wght[C]$, with any constraint C of the problem. These counters are used as constraint weighting. Whenever a constraint is shown to be unsatisfied (during the constraint propagation process), its weight is incremented by 1 (see line 11 of Algorithm 1).

The weighted degree of a variable X is then defined as the sum of the weights of the constraints involving X and at least another uninstantiated variable. The adaptive heuristic $dom/wdeg$ [2] involves selecting first the variable with the smallest ratio current domain size to current weighted degree. As search progresses, the weight of hard constraints becomes more and more important and this particularly helps the heuristic to select variables appearing in the hard part of the network. This heuristic has been shown to be quite efficient [19].

3.2 Local Search algorithms

Although there also has been some interest in using Local Search techniques to solve the CSP problem [11, 3, 4, 17], these algorithms have not been studied a fraction as much as MGAC. Contrary to systematic backtracking algorithms like MGAC, local search techniques are incomplete by nature: if a solution exists, it is not guaranteed to be found, and the absence of solution can usually not be proved. However, on very large instances, local search techniques have been proved to be the best practical alternative. We also found that local learch algorithms are far more efficient than MGAC on quite small, dense instances.

A local search algorithm works on *complete assignments*: each variable is assigned with some value, then the assignment is iteratively *repaired* until a solution is found. A repair generally involves changing the value assigned to a variable so that as few constraints as possible are violated [11]. The initial variable assignments may be randomly generated. However, in order to make the first repairs more significant, we use

Algorithm 5: initγ(P = $(\mathscr{X}, \mathscr{C})$: CN)

```
1 foreach X ∈ 𝒳 do
2     foreach v ∈ dom(X) do
3         γ(X, v) ← 0
4         foreach C ∈ 𝒞 | X ∈ vars(C) do
5             if ¬check(C|_{X=v}) then γ(X, v) ← γ(X, v) + wght[C]
```

Algorithm 6: updateγ(X : Variable, v_{old}: Value)

```
1 foreach C ∈ 𝒞 | X ∈ vars(C) do
2     foreach Y ∈ vars(C) | X ≠ Y do
3         foreach v_y ∈ dom(Y) do
4             if check(C|_{Y=v_y}) ≠ check(C|_{Y=v_y ∧ X=v_old}) then
5                 if check(C|_{Y=v_y}) then
6                     γ(Y, v_y) ← γ(Y, v_y) − wght[C]
7                 else
8                     γ(Y, v_y) ← γ(Y, v_y) + wght[C]
```

Algorithm 4 to build the initial variable assignment. The algorithm tries to minimize the number of conflicting constraints after initialization. $countConflicts(P)$ returns the number of falsified constraints involving only assigned variables.

Designing efficient local search algorithms for CSP requires the use of clever data structures and powerful incremental algorithms in order to keep track of the efficiency of each repair. [3] proposes to use a data structure $\gamma(X, v)$ which at any time contains the number of conflicts a repair would lead to. Algorithms 5 and 6 describes the management of γ ($check(C)$ controls whether C is satisfied by the current assignments of $vars(C)$). Since each assignation has an impact only on the constraints involving the selected variable, we can count conflicts incrementally at each iteration with a worst-time complexity of $O(\Gamma_{max} rd)$.

There are many cases where no value change can improve the current assignment in terms of constraint satisfaction. In this case, we have reached a *local minimum*. The main challenge over local search techniques is to find the best way to avoid or escape local minima and carry on the search. A $maxIterations$ parameter is given to each local search algorithm. It mostly allows to define a restart strategy: if no solution is found after a fixed number of iterations, the search is restarted with a new initial assignment. The best value of $maxIterations$ is highly dependant on the nature of the problem. This comes against our view of a "black box" CSP solver, and future progress on CSP4J will be aimed to eliminate that kind of parameter. However, default values are given to each algorithms and we found them to be quite robust.

The MCRW Engine With a probability p, the repair is chosen randomly instead of being selected into the set of repairs that improves the current assignment. The first al-

Algorithm 7: MCRW($P = (\mathscr{X}, \mathscr{C})$: CN, $maxIterations$: Integer): Boolean

1 $nbConflicts \leftarrow initP(P)$; $init\gamma(P)$; $nbIterations \leftarrow 0$
2 **while** $nbConflicts > 0$ **do**
3   **select** X *randomly* | X *is in conflict*
4   **if** $random[0, 1] < p$ **then**
5   **select** $v \in dom(X)$ *randomly*
6   **else**
7   **select** $v \in dom(X)$ | $\gamma(X, v)$ *is minimal*
8   $v_{old} \leftarrow$ current value for X
9   **if** $v \neq v_{old}$ **then**
10     $P \leftarrow P|_{X=v}$
11     $nbConflicts \leftarrow \gamma(X, v)$
12     $update\gamma(X, v_{old})$
13   **if** $nbIterations$++$> maxIterations$ **then throw** *Expiration*
14 **return true**

Algorithm 8: Tabu($P = (\mathscr{X}, \mathscr{C})$: CN, $maxIterations$: Integer): Boolean

1 $nbConflicts \leftarrow initP(P)$; $init\gamma(P)$; $nbIterations \leftarrow 0$
2 init $TABU$ randomly
3 **while** $nbConflicts > 0$ **do**
4   **select** $(X, v) \notin TABU \vee$ *meets the aspiration criteria* | $\gamma(X, v)$ *is minimal*
5   $v_{old} \leftarrow$ current value for X
6   insert (X, v_{old}) in $TABU$ and delete oldest element from $TABU$
7   $P \leftarrow P|_{X=v}$
8   $nbConflicts \leftarrow \gamma(X, v)$
9   $update\gamma(X, v_{old})$
10   **if** $nbIterations$++$> maxIterations$ **then throw** *Expiration*
11 **return true**

gorithm implementing this technique was described in [11] and we call it *Min-Conflicts Random Walk* (MCRW). Algorithm 7 performs a MCRW local search. At each iteration, a variable in conflict is selected (line 3). A variable X is in conflict if any constraint involving X is in conflict. Then, with a probability p, a random value (line 5) or, with a probability $1 - p$, the best value (line 7) is selected. p is one additional parameter we aim to eliminate in further versions of CSP4J. Again, the default value ($p = 0.04$) is quite robust for most problems.

The Tabu engine: Previous repairs are recorded so that we can avoid repairs that lead back to an already visited assignment. A limited number of repairs is remembered, and older ones are forgotten, allowing us to always have a fairly high number of repairs available at each iteration. The size of the Tabu List is arbitrary fixed before search. Note that the *aspiration criterion* allows to select a repair in the Tabu list if it permits to achieve a new best assignment. There have been previous works that mention the

Algorithm 9: WMC($P = (\mathscr{X}, \mathscr{C})$: CN, $maxIterations$: Integer): Boolean

```
1  nbConflicts ← initP(P) ; initγ(P) ; nbIterations ← 0
2  while nbConflicts > 0 do
3      select (X, v) | γ(X, v) is minimal
4      v_old ← current value for X
5      if γ(X, v) ≥ γ(X, v_old) then
6          foreach C ∈ 𝒞 | C is in conflict do
7              wght[C]++ ; nbConflicts++
8              foreach Y ∈ vars(C) do
9                  foreach w ∈ dom(Y) do
10                     if ¬check(C|_{Y=w}) then γ(Y, w)++
11     else
12         P ← P|_{X=v}
13         nbConflicts ← γ(X, v)
14         updateγ(X, v_old)
15     if nbIterations++ > maxIterations then throw Expiration
16 return true
```

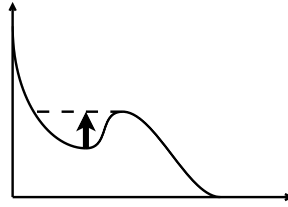

Fig. 2. Escaping from a local minimum

efficiency of Tabu search for Constraint Optimization problems (Max-CSP) [3, 4]. Algorithm 8 performs a Tabu search. The size of the Tabu list is one additional parameter we aim to eliminate in further versions of CSP4J. Again, the default value (30) is quite robust for most problems.

The WMC Engine Another efficient way to escape from local minima, called the Breakout method, has also been proposed [12]. We use this method to design a local search algorithm aimed to find solutions to satisfiable CSPs.

The resulting algorithm, *Weighted Min-Conflicts* (WMC) is described in Algorithm 9. Line 5 detects local minima. When a local minimum is encountered, all conflicting constraints are weighted (line 12). Note that a main advantage of WMC over Tabu search or MCRW is that it involves no parameter outside of $maxIterations$.

Incrementing the weight of constraints permits to effectively and durably escape from local minima, as illustrated by Figure 2. Incrementing the constraints "fills" the local minimum until another parts of the search space are reached. Constraints that are

Algorithm 10: Hybrid(P = $(\mathscr{X}, \mathscr{C})$): CN, $maxIter$: Integer, α: Float): Boolean

```
1  maxTries ← 1 ; maxBT ← maxIter × 8n/ed
2  repeat
3      startTime ← now()
4      repeat ⌊maxTries⌋ times
5          try
6              return WMC(P, maxIter)
7          catch Expiration
8      WMCDuration ← now() − startTime
9      startTime ← now()
10     try
11         return MGAC(P, maxBT)
12     catch Expiration
13     MGACDuration ← now() − startTime
14     maxTries ← α × maxTries
15     maxBT ← α × maxBT × WMCDuration/MGACDuration
```

heavily weighted are expected to be the "hardest" constraints to satisfy. By weighting them, there importance is enhanced and the algorithm will try to satisfy them in priority.

The Combo engine It is well known that the main drawback of systematic backtracking strategies such as MGAC is that an early bad choice may lead to explore a huge sub-tree that could be avoided if the heuristic had lead to focus on a rather small, very hard or even inconsistent sub-problem. In this case, the solver is said to be subject to "thrashing": it rediscovers the same inconsistencies multiple times. On the other hand, it is important to note that some instances are not inherently very difficult. These often show a "heavy tailed" behavior when they are solved multiple times with some randomization [5]. The $dom/wdeg$ heuristic was designed to avoid thrashing by focusing the search on one hard sub-problem [2, 17]. This technique is reported to work quite well on structured problems.

On the other hand, the main drawback of local search algorithms is quite straightforward: their inability to prove the unsatisfiability of problems and the absence of guarantee, even on satisfiable problems, that a solution will be found. The development of hybrid algorithms, hopefully earning the best from each world, has been devised as a great challenge in both satisfiability and constraint satisfaction problems [14].

Constraint weighting used by $dom/wdeg$ heuristic and WMC work in a similar way. Both help to identify hard sub-problems. [10] reports that statistics earned during a failed run of local search can be successfully as an oracle to guide a systematic algorithm in the search of a solution or to extract an incoherent core. We propose to use directly the weights of the constraints obtained at the end of a WMC run to initiate $dom/wdeg$ weights. We devise a simple hybrid algorithm, described by Algorithm 10 based on this assumption. This algorithm is more toroughly described in [21] (in French) and [20].

4 Conclusion and perspectives

We presented CSP4J, an API for Java 5, intended to solve CSPs as part on any Java application, in a "black-box" scheme. We introduced clues on CSP4J usage and given some examples of use, the we presented the five engines shipped with CSP4J and their respective interest.

We will continue to develop CSP4J, by optimizing the algorithms as well as refining them according to the latest refinements of fundamental research is Constraint Programming, and especially SAT and CSP solving. Next developments of CSP4J will focus on preprocessing, especially using promising algorithms such as Dual Consistency [7]. We will also try to eliminate any user-supplied parameter from our algorithms and will focus towards merging the advantages of all engines so that no expertise at all should be needed from the user, in the spirit of CLP(FD) used in Prolog interpreters.

References

1. C. Bessière, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
2. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sas. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
3. P. Galinier and J.K. Hao. Tabu search for maximal constraint satisfaction problems. In *Proceedings of CP'97*, pages 196–208, 1997.
4. P. Galinier and J.K. Hao. A General Approach for Constraint Solving by Local Search. *Journal of Mathematical Modelling and Algorithms*, 3(1):73–88, 2004.
5. C.P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100, 2000.
6. J. Hwang and D.G. Mitchell. 2-way vs d-way branching for CSP. In *Proceedings of CP'05*, pages 343–357, 2005.
7. C. Lecoutre, S. Cardon, and J. Vion. Conservative Dual Consistency. In *Proceedings of AAAI'07*, pages 237–242, 2007.
8. C. Lecoutre and F. Hemery. A Study of Residual Supports in Arc Consistency. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'2007)*, pages 125–130, 2007.
9. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
10. B. Mazure, L. Sas, and . Grgoire. Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, 22:319–331, 1998.
11. S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint-satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.
12. P. Morris. The breakout method for escaping from local minima. In *Proceedings of AAAI'93*, pages 40–45, 1993.
13. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.
14. B. Selman, H. Kautz, and D. McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of IJCAI'97*, 1997.
15. R.M. Stallman. GNU General Public License. GNU Project–Free Software Foundation, http://gnu.org/licenses, 1991.

16. R.M. Stallman. GNU Lesser General Public License. GNU Project–Free Software Foundation, http://gnu.org/licenses, 1999.
17. J.R. Thornton. *Constraint weighting local search for constraint satisfaction*. PhD thesis, Griffith University, Australia, 2000.
18. M. van Dongen, C. Lecoutre, O. Roussel, R. Szymanek, F. Hemery, C. Jefferson, and R. Wallace. Second International CSP Solvers Competition. http://cpai.ucc.ie/06/Competition.html, 2006.
19. M. R. C. van Dongen, editor. *Proceedings of CPAI'05 workshop held with CP'05*, volume II, 2005.
20. J. Vion. Breaking out CSPs. In *Proceedings of the CP 2007 Doctoral Programme*, pages 175–180, 2007.
21. J. Vion. Hybridation de prouveurs CSP et apprentissage. In *Actes des troisièmes Journées Francophones de Programmation par Contraintes (JFPC '07)*, 2007.

A Report on the B-Prolog CSP Solver

Neng-Fa Zhou

CUNY Brooklyn College & Graduate Center

Abstract. This paper provides details of the B-Prolog CSP solver submitted to Second International CSP Solvers Competition. It also attempts to shed some light on why the solver performed well in two of the categories and why it didn't do so well in other categories.

1 Introduction

This paper provides details of the B-Prolog solver submitted to Second International CSP Solvers Competition, and attempts to give a quick analysis of the results. The constraint propagators used in the solver are implemented in AR (action rules) [3, 5], a language available in B-Prolog, and the search part is implemented using `labeling_mix`, a built-in in B-Prolog, that allows for the use of mixed strategies and time limits in labeling variables.

The results of the B-Prolog solver are mixed: On the one hand, it was unexpectedly ranked top in two of the categories (global and n-ary intensional), and on the other hand, it was placed only 13th in the binary intensional category. The propagators from last year's solver [4] were used for extensionally defined constraints. Since the procedures on tables were implemented in Prolog, the poor performance on extensionally defined constraints was expected.

B-Prolog's finite-domain solver has the reputation for good performance. As described in [3], the high performance is partially attributed to the efficient event-handling architecture. This high performance is normally revealed on not only n-ary constraints but also binary constraints. The implementation of the `all_distinct` constraint is based on a weak version of the hall-set finding algorithm [5], which is weaker in terms of pruning power than Régin's filtering algorithm [1]. Channeling constraints, which are facilitated by the `dom_any` event in AR, are used to remedy the weakness of the algorithm. It is unclear if Régin's algorithm is used in any other participating solvers. If so, it would be worthwhile to investigate why the B-Prolog solver outperformed them.

2 Action Rules and Events

The AR (*Action Rules*) language is designed to facilitate the specification of event-driven functionality needed by applications such as constraint propagators and graphical user interfaces where interactions of multiple entities are essential [3]. An action rule takes the following form:

$$Agent, Condition, \{Event\} => Action$$

where *Agent* is an atomic formula that represents a pattern for agents, *Condition* is a conjunction of conditions on the agents, *Event* is a non-empty disjunction of patterns for events that can activate the agents, and *Action* is a sequence of arbitrary subgoals. An action rule degenerates into a *commitment rule* if *Event* together with the enclosing braces are missing. In general, a predicate can be defined with multiple action rules. For the sake of simplicity, we assume in this paper that each predicate is defined with only one action rule possibly followed by a sequence of commitment rules.

Definition 1. A subgoal is called an *agent* if it can be suspended and activated by events. For an agent α, a rule "*H*, *C*, $\{E\}$ => *B*" is *applicable* to the agent if there exists a matching substitution θ such that $H\theta = \alpha$ and the condition $C\theta$ is satisfied.

When an agent is created, the system checks if the action rule in its predicate is applicable to it.[1] If so, the agent will be suspended until it is *activated* by one of the events specified in the rule.

Whenever the agent is activated by an event, the condition of the action rule is tested *again*. If it is met, the action is executed. The agent does not vanish after the action is executed, but instead sleeps until it is activated again. There is no primitive for killing agents explicitly. An agent vanishes only when a commitment rule is applied to it. The reader is referred to [3] for a detailed description of the language and its operational semantics.

The following event patterns are supported for programming constraint propagators:

- `generated`: After an agent is generated but before it is suspended for the first time. The sole purpose of this pattern is to make it possible to specify preprocessing and constraint propagation actions in one rule.
- `ins`(X): when the variable X is instantiated.
- `bound`(X): when a bound of the domain of X is updated. There is no distinction between lower and upper bounds changes.
- `dom`(X,E): when an *inner* value E is excluded from the domain of X. Since E is used to reference the excluded value, it must be the first occurrence of the variable in the rule.
- `dom`(X): same as `dom`(X,E) but the excluded value is ignored.
- `dom_any`(X,E): when an arbitrary value E is excluded from the domain of X. Unlike in `dom`(X,E), the excluded value E here can be a bound of the domain of X.
- `dom_any`(X): equivalent to the disjunction of `dom`(X) and `bound`(X).

Note that when a variable is instantiated, no `bound` or `dom` event is posted. Consider the following example:

[1] Notice that since one-directional matching rather than full-unification is used to search for an applicable rule and only tests are allowed in the condition, the agent will remain the same after an applicable rule is found.

```
p(X),{dom(X,E)} => write(dom(E)).
q(X),{dom_any(X,E)} => write(dom_any(E)).
r(X),{bound(X)} => write(bound).
go:-X :: 1..4, p(X), q(X), r(X), X #\= 2, X #\= 4, X #\= 1.
```

The query `go` gives the following outputs: `dom(2)`, `dom_any(2)`, `dom_any(4)` and `bound`.[2] The outputs `dom(2)` and `dom_any(2)` are caused by `X #\= 2`, and the outputs `dom_any(4)` and `bound` are caused by `X #\= 4`. After the constraint `X #\= 1` is posted, `X` is instantiated to 3, which posts an `ins(X)` event but not a `bound` or `dom` event.

A rule is allowed to specify multiple event patterns, but the `dom`(X,E) and `dom_any`(X,E) patterns are allowed to co-exist with `ins` patterns only. For each co-existing `ins`(X) pattern, there must be a condition `var`(X) in the guard so that the action is never executed when the rule is triggered by an `ins` event.

Note also that the `dom_any`(X,E) event pattern should be used only on small-sized domains. If used on large domains, constraint propagators could be flooded with a huge number of `dom_any` events. For instance, for the propagators defined in the previous example, the query

```
X :: 1..1002, q(X), X #>1000
```

posts 1000 `dom_any` events, while it would post only one `bound` event if `q(X)` were `p(X)` or `r(X)`. For this reason, in the real implementation propagators for handling `dom_any`(X,E) events are generated only after constraints are preprocessed and the domains of variables in them become small.

For each event type, each domain variable has a slot for the list of watching propagators. Therefore, the `dom` event imposes little space overhead: one slot for `dom`(X, E) and another slot for `dom_any`(X, E) for each domain variable X. There is almost no time overhead because an event is posted only when the watching list is not empty.

3 Propagators for Extensional Constraints

3.1 Conflict constraints

A conflict relation is represented as a hashtable. Let $R = [T_1, \ldots, T_n]$ be a conflict relation where $T_1, \ldots, T_n$ are the tuples, and let R_i be the projection of R onto the columns except for column i. For each column i and for each tuple T in R_i, there is an element in the hashtable with the key $k(i, T)$ and the value $[A_1, \ldots, A_k]$ which is a list of no-good values for column i if T is a partial solution.

The propagator for a conflict constraint is implemented as follows:

[2] In the current implementation of AR, when more than one agent is activated the one that was generated first is executed first. This explains why `dom(2)` occurs before `dom_any(2)` and also why `dom_any(4)` occurs before `bound`.

```
conflict_constraint(Rel,Constr),
    no_vars_gt(1,1),{ins(Constr)}
    =>
    true.
conflict_constraint(Rel,Constr)
    =>
    project_constr_on_one_arg(Constr,T,I,Xi),
    conflict_constraint_action(Rel,k(I,T),Xi).
```

where `Rel` is the conflict relation represented as a hashtable and `Constr` is the list of variables of a constraint. The propagator is suspended if there are more than one free variables in the constraint. The second rule is executed if the constraint contains at most one variable. The call `project_constr_on_one_arg (Constr,T,I,Xi)` extracts the free variable `Xi`, its column number `I`, and the list of ground arguments `T` from `Constr`, and the call `conflict_constraint_action (Rel,k(I,T),Xi)` retrieves the no-good values for `Xi` from `Rel` and excludes them from the domain of `Xi`.

3.2 Support constraints

Given a support relation, we extract the following information from it: (1) *static bounds information*: the minimum and maximum elements in each column; (2) *dynamic bounds information*: for each value x in each column i, the minimum and maximum support elements in each column j ($j \neq i$); and (3) *projected binary relations*: the projected binary relations of the original relation onto each two columns. Each projected binary relation is represented as a hashtable, so that for each value in a column we can retrieve its support values in the other column in the binary relation.

For binary support constraints, each value in the domain of a variable can have multiple supporting values in the domain of the other variable. We set up a counter for each value in each domain for counting the support values in the other domain. Whenever the counter of a value becomes zero, the value is excluded from its domain. So the job of maintaining arc consistency reduces to maintaining the counters.

Let `BinaryRelation` be a hashtable representation of the binary relation on two variables `X` and `Y`. For each value in the domain of `X`, it takes constant time to retrieve its supporting values and their associated counters. The propagator for maintaining `Y`'s counters is implemented easily as follows:

```
ac4(BinaryRelation,X,Y),var(X),var(Y),
    {dom_any(X,E)}
    =>
    decrement_counters(BinaryRelation,E,Y).
ac4(BinaryRelation,X,Y) => true.
```

Whenever a value `E` is excluded from the domain of `X`, the counters of the values in the domain of `Y` supported by `E` are decremented. If the counter of a value becomes zero, the value is excluded from the domain of `Y`.

4 Propagators for Intensional Constraints

The B-Prolog solver performs forward checking on disequality ($\neq$) constraints, maintains interval consistency for inequality ($>$,$\geq$,$<$, and $\leq$) constraints, arc consistency for binary equality constraints, and a hybrid of interval and arc consistency for n-ary constraints [3].

The `dom`(X,E) event facilitates implementing propagators for maintaining arc consistency for binary equality constraints. For an equality binary constraint, there is only one supporting value for each value in a domain. Therefore, whenever a value is excluded from a domain, we only need to exclude its counterpart from the other domain to maintain arc consistency. Consider, for example, the constraint `X+Y #= C` where `X` and `Y` are domain variables and `C` is an integer. The propagator defined in the following propagates exclusions of values from the domain of `Y` to `X` to achieve arc consistency:

```
'X in C-Y_ac'(X,Y,C),var(X),var(Y),
    {dom(Y,Ey)}
    =>
    Ex is C-Ey,
    exclude(X,Ex).
'X in C-Y_ac'(X,Y,C) => true.
```

For the original constraint `X+Y #= C`, we need to generate two propagators, namely, `'X in C-Y_ac'(X,Y,C)` and `'X in C-Y_ac'(Y,X,C)`, to maintain the arc consistency. Note that in addition to these two propagators, we also need to generate propagators for maintaining interval consistency since no `dom(Y,Ey)` event is posted if the excluded value happens to be a bound. Note also that we need to preprocess the constraint to make it arc consistent before the propagators are generated.

5 The `all_distinct` Constraint

Many algorithms have been proposed for maintaining different levels of consistency for the `all_distinct` constraint [2]. The filtering algorithm by Régin [1] achieves hyper-arc consistency. However, because of the almost cubic order of complexity, B-Prolog adopts a Hall-set finding algorithm.

Definition 2. For the constraint `all_distinct`($[X_1,\ldots,X_n]$) where X_i has the domain D_i ($1 \leq i \leq n$), a set H is a *Hall set* if the number of subsets of H among D_1, …, D_n is greater than or equal to the size of H. Formally, H is a Hall set if $|\{D_i \mid D_i \subseteq H\}| \geq |H|$.

Since there are an exponential number of potential Hall sets, we have to rely on some heuristics to choose what sets to test. The implementation presented in [3] checks if the domain of each variable is a Hall set when a constraint is installed and when the domain is updated. Understandably, since no union of

domains is considered, this heuristic has its limitations. Consider, for example, the constraint `all_distinct`($[X_1, X_2, X_3, X_4]$) where the variables have the following domains:

X_1	X_2	X_3	X_4
$\{1,2\}$	$\{1,3\}$	$\{2,3\}$	$\{1,2,3,4\}$

The heuristic fails to find the Hall set $\{1,2,3\}$ and thus fails to bind X_4 to 4.

B-Prolog uses channeling constraints to increase the pruning power. By adding the constraints `primal_dual`(Xs,Ys) and `all_distinct`(Ys), the dual variables have the following domains:

Y_1	Y_2	Y_3	Y_4
$\{1,2,4\}$	$\{1,3,4\}$	$\{2,3,4\}$	$\{4\}$

After Y_4 is instantiated to 4, 4 is excluded from the domains of Y_1, Y_2, and Y_3, and X_4 is instantiated to 4 because of the existence of the channeling constraint. As demonstrated by this example, using dual models can to some extent remedy the limitation of the Hall-set finding algorithm.

With the `dom` event, we can use only $2 \times n$ propagators to implement the channeling constraint $\forall_{i,j}(X_i \neq j \Leftrightarrow Y_j \neq i)$. Let `DualVarVector` be a vector created from the list of dual variables. For each primal variable `Xi` (with the index `I`), a propagator defined below is created to handle exclusions of values from the domain of `Xi`.

```
primal_dual(Xi,I,DualVarVector),var(Xi),
   {dom_any(Xi,J)}
   =>
   arg(J,DualVarVector,Yj),
   exclude(Yj,I).
primal_dual(Xi,I,DualVarVector) => true.
```

Each time a value `J` is excluded from the domain of `Xi`, assume `Yj` is the `J`th variable in `DualVarVector`, then `I` must be excluded from the domain of `Yj`. We need to exchange primal and dual variables and create a propagator for each dual variable as well. Therefore, in total $2 \times n$ propagators are needed.

Note that a preprocessing phase is needed to ensure that the channeling constraints are consistent before any propagator is generated. The preprocessing phase takes $O(n^2)$ time.

6 Future Improvements

There is plenty of room for improvement of the solver, whether in the two categories it won or in other categories where it performed poorly. The B-Prolog solver was disappointedly ranked only 13th among 16 participating solvers in the binary intensional category. The following table shows the instances that the B-Prolog solver failed to solve within the time limit:

Problem class	# failed instances
fapp	245
taillard	148
haystack	48
rlfap	30
queensKnight	18
knights	15
pigeons	12
os-qp	10

A closer look reveals the reason: almost all of the failed instances contain non-linear (e.g., $X * Y = C$, $abs(X - Y) = C$, and $X \ mod \ Y = C$) and disjunctive constraints which were not efficiently implemented in the submitted version of the solver.

It was found later that the answers found for the FISCHER series by the B-Prolog solver were wrong. Because of the existence of min/2 and max/2, some variables that belong to the same SCC (Strongly-Connected-Component) are wrongly put into different SCCs. Since search never backtracks over two different SCCs, the final solution will be reported to be UNSAT if one of the SCCs is found to be UNSAT. This problem is unrelated to the B-Prolog system itself.

Future improvements include: (1) refining the propagators for non-linear and disjunctive constraints; (2) introducing certain constraint reasoning ability to the solver; (3) and tuning the labeling strategies.

Acknowledgement

The competition would be impossible without the huge amount of time and energy put into it by the organizers. So thanks go to the organizers. Special thanks go to Olivier Roussel for obtaining all the results and feedbacks.

References

1. J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the National Conference on Artificial Intelligence(AAAI-94)*, pages 362–367. AAAI Press, 1994.
2. W J van Hoeve. The alldifferent constraint: A survey. Technical report, 2001.
3. Neng-Fa Zhou. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming (TPLP)*, 6(5):483–508, 2006.
4. Neng-Fa Zhou and Mark Wallace. A simple constraint solver in action rules for the CP'05 solver competition. In *Proceedings of the CP workshop on Constraint Propagation and Implementation*, page 6 pages, 2005.
5. Neng-Fa Zhou, Mark Wallace, and Peter J. Stuckey. The `dom` event and its use in implementing constraint propagators. Technical report TR-2006013, CUNY Compute Science, (http://www.cs.gc.cuny.edu/tr/techreport.php?id=208) 2006.