Abscon 112
Toward more Robustness

Christophe Lecoutre and Sebastien Tabary

CRIL-CNRS UMR 8188,
Université d'Artois
Lens, France
{*lecoutre*, *tabary*}@cril.fr

Abstract. This paper describes the three main improvements made to the solver Abscon 109 [9]. The new version, Abscon 112, is able to automatically break some variable symmetries, infer $allDifferent$ constraints from cliques of variables that are pair-wise irreflexive, and use an optimized version of the STR (Simple Tabular Reduction) technique initially introduced by J. Ullmann for table constraints.

1 From Local to Global Variable Symmetries

In [10], we have proposed to automatically detect variable symmetries of CSP instances by computing for each constraint scope a partition exhibiting locally symmetrical variables. From this local information that can be obtained in polynomial time, we can build a so-called lsv-graph whose automorphisms correspond to (global) variable symmetries. Interestingly enough, our approach allows us to disregard the representation (extension, intension, global) of constraints. Besides, the size of the lsv-graph is linear wrt the number of constraints (and their arity). To break symmetries from the generators returned by a graph automorphism algorithm, a classical approach is to post lexicographic ordering constraints defined on two vectors of variables. We have proposed a new variant of an algorithm enforcing GAC (generalized arc consistency) on such constraints which is able to deal with shared variables. This algorithm is quite simple to implement and well-adapted to general-purpose constraint solvers. Our experimental results show the robustness of the overall approach with different search heuristics: on a large number of series, more instances can be solved while the cpu time required for symmetry identification is observed as negligible. These results confirm that automatically breaking symmetries constitutes a significant breakthrough for black-box CSP solvers.

In order to show the practical interest of this approach, we have then conducted an extensive experimentation on a cluster of Xeon 3.0GHz with 1GiB of RAM under Linux. Here, performance is measured in terms of cpu time (in seconds) and number of visited nodes. We have integrated to the classical MAC algorithm, that is to say the algorithm that maintains (generalized) arc consistency at each node of the search tree, several variants of the symmetry breaking

		MAC	MAC_{Le}	MAC_{Lex}	MAC^*_{Le}	MAC^*_{Lex}
scen11-f12	*cpu*	1.88	2.0	2.05	1.71	1.82
	nodes	390	614	721	4,140	4,140
scen11-f11	*cpu*	1.77	1.97	1.95	1.82	1.83
	nodes	390	614	721	4,216	4,216
scen11-f10	*cpu*	1.77	1.82	2.08	1.81	1.9
	nodes	468	327	722	109	77
scen11-f9	*cpu*	2.15	1.96	2.23	1.84	1.97
	nodes	1,064	576	922	109	90
scen11-f8	*cpu*	2.1	2.09	2.28	2.02	2.0
	nodes	1,354	558	997	112	115
scen11-f7	*cpu*	4.83	2.28	2.37	1.91	2.05
	nodes	8,369	955	1,247	121	135
scen11-f6	*cpu*	8.29	2.14	2.37	2.1	2.08
	nodes	17,839	571	1,333	172	157
scen11-f5	*cpu*	32.0	2.2	3.13	2.19	2.13
	nodes	85,104	988	3,465	253	226
scen11-f4	*cpu*	112	2.66	3.88	2.36	2.53
	nodes	$345K$	1,983	5,007	593	903
scen11-f3	*cpu*	403	3.41	7.98	2.55	2.45
	nodes	$1,300K$	3,926	17,259	946	696
scen11-f2	*cpu*	*time-out*	4.32	16.4	2.95	2.92
	nodes	–	6,014	40,615	1,700	1,591
scen11-f1	*cpu*	*time-out*	7.56	19.7	3.49	3.4
	nodes	–	14,997	47,318	3,199	2,609

Table 1. Cost of running MAC and its symmetry breaking variants on hard RLFAP instances (38 generators). The variable ordering heuristic is $dom/wdeg$.

approach described in [10]. For this experimentation, no restarts and no nogood recording were activated.

To identify variable symmetries, we have used Saucy. For each generator of the symmetry group returned by Saucy, we have considered four distinct symmetry breaking procedures. For the first one, denoted by MAC_{Le}, a binary constraint of difference Le (constraint of the form $x \leq y$) that involves the two first variables of the first cycle of the generator is posted. For the second one, denoted by MAC_{Lex}, a lexicographic ordering constraint Lex (involving all variables of all cycles of the generator) is posted. Clearly, a Lex constraint is stronger than the corresponding Le constraint: its filtering capability is greater. Notice that when the two first variables of the first cycle of the generator are included in the scope of a (non-global) constraint c of the network, one can merge c with a binary constraint Le. In practice, if c is defined in intension, its associated predicate is modified whereas if c is defined in extension, the set of tuples disallowing the constraint Le are removed from the table associated with c. When such a merging method is applied, one obtains two additional procedures, denoted by MAC^*_{Le} and MAC^*_{Lex}.

Here, we only provide some results (see Table 1) obtained for the hardest instances (which involve 680 variables and a greatest domain size of about 50 values) built from the real-world Radio Link Frequency Assignment Problem (RLFAP). Clearly, the symmetry breaking methods allow us to be far more efficient than the classical MAC algorithm. In practice, MAC^*_{Lex} has been observed as the best method and has been used for the CSP competition.

2 Exploiting Cliques

Some instances contain hidden structures such as backbones, (strong) backdoors and unsatisfiable cores. Cliques also belong to this category. A clique is a graph such that there exists an edge between any two vertices. Interestingly, sometimes, we observe that for any pair (x, y) of variables of a sub-network P' of P whose constraint graph is a clique, the relation associated with the constraint involving x and y is irreflexive. Otherwise stated, we know that $\forall \{x, y\} \subset vars(P'), x \neq y$. We can then infer an additional global constraint $allDifferent$ that can be useful to better prune the search space. However, in some constraint solvers, the filtering procedure (propagator) attached to $allDifferent$ achieves a local consistency weaker than generalized arc consistency. But, even in this case, inferring $allDifferent$ global constraints can be quite effective provided that the following (trivial) proposition is exploited.

Proposition 1. *Let $c : allDifferent(x_1, \ldots, x_r)$ be a constraint. If we have $|\cup_{i=1}^{r} dom(x_i)| < r$, then c is disentailed (i.e. the set of supports of c is empty).*

This approach is quite simple, and to the best of our knowledge, employed by some other solvers engaged in the 2008 competition. It suffices to detect cliques in a greedy manner, determine if irreflexivity is guaranteed between each pair of variables, and post a constraint $allDifferent$ that at least exploits Proposition 1. Interestingly, it is not so rare to find cliques in non-random problems. As an illustration, the instance $blackHole$-4-4-e-0 (see its constraint graph in Figure 1) contains a 16-clique that enables us to infer a global constraint $allDifferent$. As one can show that this additional constraint is disentailed by using Proposition 1, the instance is directly proved to be unsatisfiable.

3 Simple Tabular Reduction

Table constraints play an important role within constraint programming. Recently, many schemes or algorithms have been proposed to propagate table constraints or/and to compress their representation. In [7], we have shown that simple tabular reduction (STR), a technique proposed by J. Ullmann [11] to dynamically maintain the tables of supports, is very often the most efficient practical approach to enforce generalized arc consistency within MAC. We have also described an optimization of STR which allows limiting the number of operations related to validity checking or search of supports. Interestingly enough,

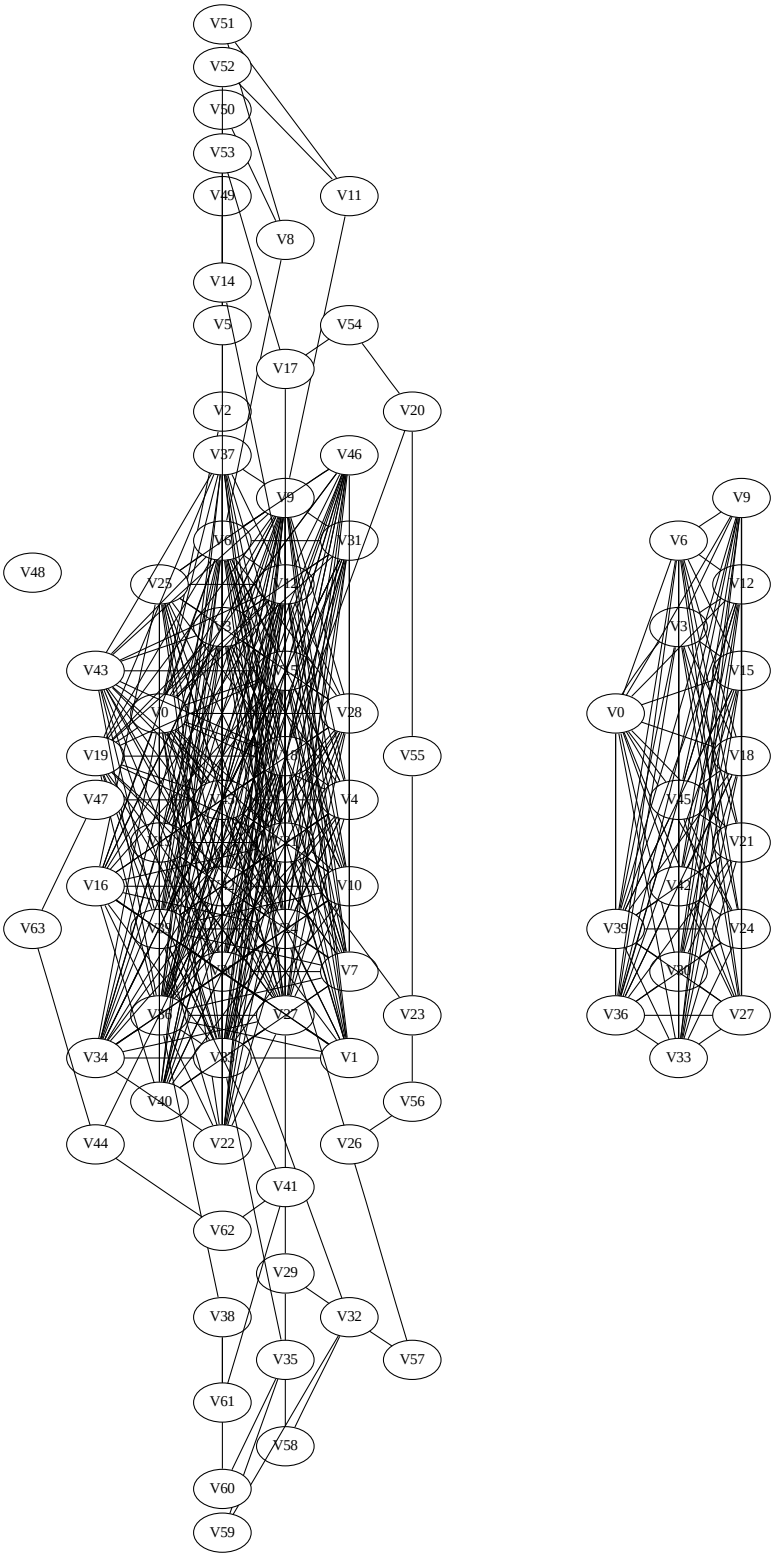

Fig. 1. The constraint graph of the instance *blackHole*-4-4-*e*-0 contains a 16-clique. A constraint $allDifferent$ generated from this clique can be shown to be disentailed.

this optimization makes STR potentially r times faster where r is the arity of the constraint(s). The results of an extensive experimentation that we have conducted with respect to random and structured instances indicate that the optimized algorithm we propose is usually around twice as fast as the original STR and can be up to one order of magnitude faster than previous state-of-the-art algorithms on some series of instances.

In order to show the practical interest of simple tabular reduction, and in particular the optimization we propose, we have then experimented using a cluster of Xeon 3.0GHz with 1GiB of RAM under Linux, employing MAC with *dom/ddeg* and *lexico* as variable[1] and value ordering heuristics, respectively. We have compared classical schemes to enforce GAC on (positive) table constraints with STR. More precisely, we have implemented the three schemes GACv, GACa and GACva described in [8]. We do believe that GACva is a representative state-of-the-art algorithm for table constraints. Our own experience confirms the results reported in [4]: GACva and the trie approach are quite robust and close in terms of performance.

Here, we only provide some results obtained for some series of Crossword puzzles. For each grid, there is a variable per white square which can be assigned any of the 26 letters of the Latin alphabet, and a constraint for any sequence of white squares which corresponds to a word that we must put in the grid. Such constraints are defined by a table which contains all words of the right length. The series prefixed by cw-m1c are defined from blank grids and only contain positive table constraints (contrary to model m1 in [1] where no two identical words can be put in the grid, which is then naturally expressed in intension). The arity of the constraints is given by the size of the grids: for example, cw-m1c-lex-vg5-6 involves table constraints of arity 5 and 6 (the grid being 5 by 6).

The results that we have obtained (see Table 2) with respect to 4 dictionaries (lex, words, uk, ogd) of different length show the good performance of STR for such series. GACstr is the original algorithm, GACstr2 is the optimized version and GACstr2+ is GACstr2 made incremental. On the most difficult instances, GACstr2+ is about two times faster than GACstr and one order of magnitude faster than GACva. Note that we do not provide mean results for these series because many instances cannot be solved within $1,200$ seconds.

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 [6]. This variant lies between PFC-MRDAC and PFC-MPRDAC [5].

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. We have integrated the pruning approach presented in [2].

[1] In our implementation, using *dom/wdeg* does not guarantee exploring the same search tree with classical and STR schemes.

		Classical GAC schemes			Simple Tabular Reduction		
		$GACv$	$GACa$	$GACva$	$GACstr$	$GACstr2$	$GACstr2+$
Crossword puzzles with dictionary lex (24,974 words)							
cw-m1c-lex-vg5-6	*cpu*	$> 1,200$	38.8	54.2	14.3	12.4	10.7
#nodes=26,679	*mem*		$2,889K$	$2,928K$	$2,932K$	$2,935K$	$2,968K$
cw-m1c-lex-vg5-7	*cpu*	$> 1,200$	357	875	134	114	96.3
#nodes=$171K$	*mem*		$4,134K$	$4,173K$	$8,005K$	$8,055K$	$8,059K$
cw-m1c-lex-vg6-6	*cpu*	$> 1,200$	2.98	4.29	1.28	1.05	0.91
#nodes=1,602	*mem*		$4,422K$	$4,344K$	$4,226K$	$4,203K$	$4,296K$
cw-m1c-lex-vg6-7	*cpu*	$> 1,200$	436	1,174	176	143	118
#nodes=$152K$	*mem*		$5,887K$	$5,692K$	$9,458K$	$9,437K$	$9,555K$
Crossword puzzles with dictionary words (45,371 words)							
cw-m1c-words-vg5-5	*cpu*	$> 1,200$	0.04	0.05	0.05	0.05	0.04
#nodes=38	*mem*		$4,969K$	$4,987K$	$4,823K$	$4,791K$	$4,809K$
cw-m1c-words-vg5-6	*cpu*	$> 1,200$	1.19	1.46	0.48	0.37	0.33
#nodes=718	*mem*		$6,508K$	$6,526K$	$6,348K$	$6,273K$	$6,348K$
cw-m1c-words-vg5-7	*cpu*	$> 1,200$	18.6	36.0	6.61	5.21	4.03
#nodes=6,957	*mem*		$8,470K$	$8,489K$	$8,276K$	$8,145K$	$8,237K$
cw-m1c-words-vg5-8	*cpu*	$> 1,200$	866	$> 1,200$	273	229	187
#nodes=$256K$	*mem*		$4,604K$		$10M$	$10M$	$10M$
Crossword puzzles with dictionary uk (225,349 words)							
cw-m1c-uk-vg5-5	*cpu*	$> 1,200$	0.05	0.05	0.1	0.07	0.07
#nodes=28	*mem*		$12M$	$12M$	$12M$	$12M$	$12M$
cw-m1c-uk-vg5-6	*cpu*	$> 1,200$	0.55	0.5	0.21	0.17	0.17
#nodes=145	*mem*		$17M$	$17M$	$16M$	$16M$	$16M$
cw-m1c-uk-vg5-7	*cpu*	$> 1,200$	2.97	5.18	0.51	0.37	0.34
#nodes=408	*mem*		$22M$	$22M$	$22M$	$22M$	$22M$
cw-m1c-uk-vg5-8	*cpu*	$> 1,200$	82.5	71.9	7.08	5.71	4.78
#nodes=8,148	*mem*		$12M$	$12M$	$11M$	$11M$	$11M$
Crossword puzzles with dictionary ogd (435,705 words)							
cw-m1c-ogd-vg6-6	*cpu*	$> 1,200$	0.37	0.31	0.23	0.17	0.15
#nodes=98	*mem*		$46M$	$47M$	$46M$	$46M$	$48M$
cw-m1c-ogd-vg6-7	*cpu*	$> 1,200$	95.3	56.1	12.0	8.01	6.81
#nodes=9,522	*mem*		$11M$	$11M$	$11M$	$11M$	$11M$
cw-m1c-ogd-vg6-8	*cpu*	$> 1,200$	53.0	6.44	2.91	2.0	1.72
#nodes=2,806	*mem*		$24M$	$23M$	$22M$	$22M$	$24M$
cw-m1c-ogd-vg6-9	*cpu*	$> 1,200$	727	214	35.1	25.1	19.1
#nodes=23,283	*mem*		$42M$	$41M$	$39M$	$37M$	$40M$

Table 2. Representative results obtained on series of Crossword puzzles using dictionaries of different length. Cpu time is given in seconds and mem(ory) in MiB. The number of nodes (#nodes) explored by MAC is given below the name of each instance.

As a consequence, a requirement was that the value ordering heuristic always selects the value with the lowest aic (arc-inconsistency count). Two variable ordering heuristics were tested: $dom/wdeg$ [3] and $dom * gap/ddeg$ that involves the aic gap of the variables [2]. More precisely, the ratio $dom/ddeg$ is multiplied by the aic gap in order to favour variables for which there is a large gap between the best value and the following one.

Unfortunately, we omitted to remove a trace used for debugging. Consequently, the solvers have been considerably slowed down.

5 Some Deficiencies

Abscon 112 has suffered from two main deficiencies. First, in the category of global constraints, the solver was not ready altogether. Indeed, the constraint cumulative was not implemented and the filtering procedure for the constraint element not optimized. Second, as mentioned above, for Max-CSP, a trace output by the solver has considerably slowed down the resolution.

Acknowledgements

This work has been supported by the CNRS and by the "IUT de Lens".

References

1. A. Beacham, X. Chen, J. Sillito, and P. van Beek. Constraint programming lessons learned from crossword puzzles. In *Proceedings of Canadian Conference on AI*, pages 78–87, 2001.
2. H. Bennaceur, C. Lecoutre, and O. Roussel. A decomposition technique for solving Max-CSP. In *Proceedings of ECAI'08*, 2008.
3. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
4. I.P. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI'07*, pages 191–197, 2007.
5. J. Larrosa and P. Meseguer. Partition-Based lower bound for Max-CSP. *Constraints*, 7:407–419, 2002.
6. J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107(1):149–163, 1999.
7. C. Lecoutre. Optimization of simple tabular reduction for table constraints. In *Proceedings of CP'08*, 2008.
8. C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, pages 284–298, 2006.
9. C. Lecoutre and S. Tabary. Abscon 109: a generic CSP solver. In *Proceedings of the 2006 CSP solver competition*, pages 55–63, 2007.
10. C. Lecoutre and S. Tabary. Des symétries locales de variables aux symétries globales. In *Proceedings of JFPC'08 (in french)*, pages 181–190, 2008.
11. J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177:3639–3678, 2007.