

## Promoting Robust Black-box Solvers through Competitions\*

Christophe Lecoutre · Olivier Roussel ·  
M.R.C. van Dongen

Received: date / Accepted: date

**Abstract** This paper presents the experiences of the organisers of the four constraint solver competitions which were held in conjunction with CP in the previous years. The paper mainly focuses on the competitions which were held in 2008 and 2009, outlines the reasons for organising the competitions, describes how the solvers were evaluated, and presents lessons, observations, and general trends.

**Keywords** solver competition · experimental evaluation · robust constraint solvers · black-box

### 1 Why Organise a Competition?

This paper presents the experiences of the organisers of the four constraint solver competitions, all of which were held in conjunction with the international conferences on principles and practice of Constraint Programming (CP). The paper mainly focuses on the competitions organised in 2008 and 2009.

In the remainder of this section, we outline three main motivations for organising a competition: solver comparison, problem instance exchange, and the promotion of robust constraint solvers.

---

\* <http://cpai.ucc.ie/05/CPAI.Vol.II.pdf>, <http://www.cril.univ-artois.fr/{CPAI06,CPAI08,CSC09}>

C. Lecoutre · O. Roussel  
CRIL – CNRS UMR 8188,  
Université Lille-Nord de France, Artois  
rue de l'université, SP 16, F-62307 Lens  
France  
E-mail: {lecoutre,roussel}@cril.fr

M.R.C. van Dongen  
Computer Science Department  
University College Cork  
Western Road, Cork,  
Ireland  
E-mail: dongen@cs.ucc.ie

---

*Facilitating Solver Comparison* Our first reason for starting the competitions is the desire to have an independent platform for solver comparison. The need for such a platform may not be immediately obvious. For example, there are many papers describing the fine details of how to implement solvers, constraint propagation algorithms, variable and value ordering heuristics, and so on. Research papers also show a trend of comparing the implementation of new algorithms against existing ones.

However, there are several complications when it comes to algorithm comparison. For example, the conditions for algorithm comparison may vary due to differences in hardware, software, and operating systems environment. It takes time to implement a given algorithm, it takes even more time to implement it efficiently, and besides, the implementation may depend on the remaining solver implementation. This may result in a significant delay between the first naive and subsequent optimised implementations of a given algorithm. Finally, it is clearly impossible for anybody to implement all relevant algorithms.

For these reasons, we feel that comparing the state-of-the-art in solver implementation is only possible in the presence of a common platform, which facilitates the evaluation of solvers. With such a platform, one can overcome some of the previous complications. For example, the conditions for algorithm comparison are constant, and, in the presence of sufficiently many contestants, it allows for the comparison of a significant and relevant cross-section of solver implementations.

*Facilitating Problem Instance Exchange* Our second reason for organising constraint solver competitions is the exchange of problem instances. Requiring that all participating solvers be capable of handling a common format creates the conditions for the exchange of problem instances among researchers in the field.

Clearly an agreement on problem format also supports reproducibility of results. We note that since the first competition, several contestants *as well as* others have contributed new and interesting problem instances which are now publicly available for the larger constraint community. This may be viewed as a positive side-effect.

*Facilitating Robust Constraint Solvers* Our third and final reason for organising the competition is that we challenge researchers to implement robust black-box solvers. We develop this idea below.

A solver applies constraints so as to avoid exploring combinations of values that cannot possibly belong to any solution. Ideally, the operation of a solver should be totally transparent to the user; the user should not be aware of specific short-cuts used by the solver. Unfortunately, this ideal vision is not exactly correct in reality because most of the currently available constraint toolkits require the user to guide search, to select algorithms to filter the search space, to break symmetries, etc. Besides, modelling a problem may turn out to be very difficult for the uninitiated user as, for example, the number of specific patterns of constraints, called global constraints, may be unexpectedly large. As pointed out by Puget [19], an important challenge for constraint programming is to achieve greater *simplicity of use*: constraint programming should be made easier for non-specialist users. We believe that enhanced ease of use of constraint solvers will boost the impact of constraint programming on industry and academia and will establish it more firmly as a key software technology for solution of combinatorial problems.

To take up the simplicity of use challenge, there is a need for robust and efficient solvers that users can regard as black-boxes. A *black-box* is a system such that the user sees only its input and output data: its internal structure or mechanism is invisible to him. This approach has recently been emphasised by some position papers [19,9]. This black-box approach

---

partially addresses the requirement for simplicity since the user does not have to be aware of (or modify or extend) embedded techniques and algorithms. However, a black-box constraint solver must have a default configuration that yields in most cases the best behaviour that could be obtained by fine tuning of available options. This can be achieved by making the solver robust.

A solver is *robust* when, to some extent, it is able to provide similar results, while consuming similar resources (time, space), given different but equivalent models of the same problem. It is important to note that the user of an ideally robust solver does not need to provide carefully chosen models of constraint networks. Robustness compensates for bad modelling by providing sophisticated solving techniques. Some of these certainly remain to be invented but others are well-known, namely, learning (constraint acquisition, nogood recording), adaptive search heuristics, automatic symmetry breaking, etc. These techniques aim at a particularly clever exploration of the search space, learning much useful information before or during search so as to avoid exploring fruitless combinations of values of variables. Given different formulations of the same problem instance, advanced learning and inference techniques reduce behaviour disparities by increasing the efficiency of the solver. Thus robustness and efficiency are intimately inter-related. Our contribution to take up the simplicity of use challenge is the organization of constraint solver competitions.

In the competitions we organised, contestants have to submit solvers with a default parametrisation file. The solvers are then executed as black-box solvers, without human intervention. This is clearly oriented toward assessing the solver robustness since the solver ranking is based on the number of solved instances within a predefined allotted time per instance. Consequently, contestants aim at crafting solver configurations which increase the solver's ability to solve as many problem instances as possible, within a reasonable time.

## 2 Basis for Solver Comparison

When organising a solver competition it is important to evaluate solvers in the same environment and on the same instances. Unfortunately, even to this day there is no agreement on the representation of CP instances, i.e. instances of the frameworks CSP (Constraint Satisfaction Problem), WCSP (Weighted CSP), QCSP (Quantified CSP), COP (Constraint Optimisation Problem), etc., and most solvers use their own native format. Furthermore, different solvers support different kinds of constraints. Solvers may support constraints in extension and/or in intension, binary and/or non-binary constraints, and solvers greatly vary in the list of global constraints that they accept.

*Common Features* The model retained for the competitions has been to first identify a minimal set of important features of constraints, restrict the first competitions to these core features, and then gradually include new features. Our small-step approach has been motivated by the observation that very few solvers implement all possible forms and types of constraints, and are both capable of dealing with satisfaction, optimisation, quantification, ... To attract a substantial number of contestants (more than 10 teams), we *had to* agree on a small subset of CP and gradually extend it.

Some may have regretted our decision to only consider extensional constraints in the first competition and not to consider global constraints in the first two competitions. Others may have regretted our decision not to include optimisation instances in the first two competitions. However, having organised four competitions, we think that our approach was rather

wise. Indeed, we have been able to gradually extend the scope of the competition (noticeably by including important global constraints, and optimisation problems with Max-CSP and WCSP). And this was possible while maintaining a substantial core of regular contestants as well as attracting new ones (e.g. we had 18 teams in 2008).

The constraint solver competitions have considered the automatic solving of constraints by solvers. This means that solvers are considered as black-boxes which have no other information than the declarative description of the constraints. Some CP toolkits do not target this model. For example, the Gecode team did not enter the 2008 competition because they expected that the order of selection of variables during search would be fixed and given in the instance. In our opinion, posing restrictions on variable orders has two disadvantages. First, it poses restrictions on the solvers because they have to implement the fixed variable orders. Such restrictions may rule out state-of-the-art solvers which implement new and innovative algorithms. Second, it poses restrictions on the possible variable orders. For example, the commonly used variable order *dom/wdeg* [2] depends on the order of the propagation (i.e. order of the revisions of a generic filtering algorithm, or order of the calls to propagators). Therefore two solvers which both agree on *dom/wdeg* as a variable order *and* agree on a tie breaking variable order, may still visit different nodes during search. In general, this problem holds for any variable ordering heuristic which depends on the order of the propagation. As a consequence, it is impossible in general to use such orders to fix the order of visited nodes for two different solvers.

*Problem Representation Format* The first competition in 2005 was restricted to binary and non-binary constraints defined in extension. At that time, two formats were defined: a tabular format and an XML based format. Instances were provided to solvers in both formats. To also accommodate solvers which did not parse any of these formats, *off-line* conversions were allowed from the competition format to other formats, provided these conversions did not add any information. The time which was required for these off-line conversions was not added to the total solution time. The problem with such conversions is that it is extremely difficult to detect if new information is introduced during the translation. Furthermore, small changes in the order of constraints or variables may also result in significant differences in the execution time of a solver. For these reasons, off-line conversions were gradually discouraged.

When the second competition in 2006 introduced constraints in intension, it became clear that the tabular format was not suitable to represent such constraints. Version 2.0 of the XML format (XCSP 2.0) added support for constraints in intension, as well as a representation of the alldifferent global constraint. The third version of the competition introduced new global constraints, as well as the Max-CSP and WCSP problems<sup>1</sup>. To this end, the XCSP 2.1 format [21] introduced a generic representation of global constraints that allowed a direct translation of the syntax used in the catalog of global constraints<sup>2</sup>, as well as support for WCSP and quantified CSP. The fourth competition in 2009 only introduced some more global constraints.

The development of the XCSP format was actually driven by the requirements of the CSP competition and each new version tried to introduce the least number of features needed to support these requirements. As an example, XCSP was never intended to be a modelling language: it only describes constraint networks. The goal of the XCSP format is to reach the right balance between structuration, readability and succinctness. Having an XML based

<sup>1</sup> Unfortunately, too few solvers were submitted to the WCSP track to draw significant conclusions.

<sup>2</sup> <http://www.emn.fr/x-info/sdemasse/gccat/>

---

format ensures that it is well structured, and easily parsable by the solvers. Still, humans can easily read and modify an instance, which is an important feature. Finally, succinctness is achieved by the abridged version of the format which replaces some tags by single characters. This abridged format is significantly shorter than a pure XML representation and preserves the same benefits.

*Problem Instance Selection* The next issue for comparing solvers is to identify a set of instances to use as benchmark. Ideally, these instances should be representative of the kind of instances which are offered to solvers on a regular basis. Besides, we should have a benchmark that covers the largest number of application domains. Finally, the benchmark should not be biased toward a certain kind of solver. It is almost impossible to guarantee that all these requirements are met. The process used in the competition has been to (1) collect a number of various instances that is as large as possible, (2) ask an independent jury to select how many instances should be selected in each series/category, and finally (3) perform a random selection of instances in each series/category. More than 16,000 instances have been collected for the competitions and have been made available on the competition web site. Unfortunately, most of these instances have been submitted by contestants or by the organisers. This contrasts with the SAT (Satisfiability) competition for example where many users submit an instance without submitting a solver. As a final observation, we note that our current problem instance pool is a valuable resource for researchers in the constraint community. In fact, the use of instances from a common problem instance resource for the purpose of experimental comparison of researchers in publications makes it easier to repeat these experiments by others.

*Environment* One of the most important requirement in a competition is the ability to verify the results. There are several issues. First, the answer given by a solver must be verified. Solvers claiming the satisfiability of a given instance are required to also output a solution of that instance. The solution acts as a certificate, a *proof*, that the solver's satisfiability claim is correct. In fact, such certificates are always used for the purpose of cross-checking. Unfortunately, it is much more difficult for a solver to return a certificate or proof that a given instance is unsatisfiable. The only cross-checking of claims of a given instance's unsatisfiability is that we make sure no other solver can provide a solution which provably satisfies all constraints of that instance. This means that we do not question unsatisfiability claims of a given instance unless we have a proof that the instance is satisfiable. The same issue exists for the optimisation problems (Max-CSP and WCSP). We only check if a solver found a better solution or not, since we are not able to check efficiently if the solution is actually the optimum. A solver which gave an incorrect answer in a given category was disqualified from that category because its answers cannot be trusted. Indeed, it appeared that some solvers were fast because they did not handle each possible case and were incorrect on some instances. It would not make sense to compare them with other solvers.

The previous checks are easy to automate. Other checks, such as identifying if an external event caused a solver to fail on a specific instance, cannot be automated. Only the authors of the solver can do this. This is why they were given the chance to view the results of their own solvers and report potential problems before the results were made final. When a problem was identified, the solver was run again on the instance.

Another issue during the competition is the reliable counting of the CPU time used by a process, which is not so obvious when a solver can run several processes or threads. In order to make sure that the CPU time was meaningful, we also limited the memory that could be used by a solver so as to prevent it from swapping. Since 2006, the competition has used

the framework that was set up for the competition of pseudo-Boolean solvers in 2005. This framework has generated a huge amount of detailed information which is publicly available on the competition web sites, so that the competition be as transparent as possible.

Solvers were ranked on the number of instances they solved, and ties were broken on the cumulated CPU time. This ranking, which is very simple, intuitive and easy to check, targets the identification of robust solvers. Other rankings (such as the purse based scoring) try to aggregate several criteria and promote other solvers features, such as speed, or innovation. Unfortunately, these rankings are not always intuitive<sup>3</sup> because the relative ranking of two solvers can be changed by the results of a third solver.

*Comparing Like with Like* The final issue is to compare solvers which are comparable. It would not make sense to compare a solver which only solves constraints in extension with a solver which only accepts constraints in intension. Besides, even if two solvers are able to solve the same instances, it is still relevant to identify if a solver is faster than the other one on a given set of instances. For these reasons, two levels of categories were defined. The first level is based on syntactic criteria: binary or non-binary instances, and constraints in extension or intension. This makes up for 4 categories. Global constraints pose problems regarding the fair comparison of solvers. For example, solvers may, or may not implement a given global constraint. This means we may end up creating an exponential number of categories, which we can't afford given the huge number of global constraints. The approach taken in the last competition was to cluster solvers which were capable of solving the same global constraints. This ensures that we are comparing solvers that are actually comparable.

It has been suggested that solvers not implementing a given kind of constraint should be evaluated on a rewritten instance where the unsupported constraints are replaced by a decomposition into simpler constraints that the solver supports. We believe that this approach is incorrect for two reasons. First, it violates the requirement to evaluate solvers on the very same set of instances. Second, we can hardly expect a solver running on a rewritten instance to be competitive with a solver running on the initial instance (e.g. rewriting constraints in intension into extensional constraints necessarily changes the complexity of the problem). Clearly, such a comparison would not be fair.

The second level of categories is based on the origin and the properties of the instances. As an example, categories were defined for concrete problems, mathematical problems, torture test problems, randomly generated problems and structured ones.

### 3 Lessons, Observations, and Trends

In this section, we attempt to draw some general lessons, observations, and trends from the last conducted competitions. We start with two sections which confirm the importance of adaptive heuristics and restarts. We continue by pointing out a trend toward filtering algorithms associated with non-binary extensional constraints and a trend toward strong consistencies. We conclude by pointing out that the competitions have revealed the effectiveness of novel approaches through some remarkable solvers.

*Search-guiding Heuristics* Traditionally, variable-ordering heuristics exploit information about the root or the current node of the search tree which is built by a backtracking algorithm. They are respectively called *static* and *dynamic*. A well-known dynamic heuristic is

---

<sup>3</sup> As an illustration, competitors of the SAT 2009 competition voted against the purse based scoring.

*dom* [12]. Recently, two adaptive heuristics have been introduced. The first is based on constraint weighting [2] and the second on impacts [20, 8]. Intuitively, adaptive heuristics seem more appropriate as efficient general-purpose heuristics than static and dynamic heuristic. Here, we claim that the two beforementioned adaptive heuristics may be considered as the current state-of-the-art in variable ordering heuristics. Indeed, all best ranked CSP solvers at the last competitions implement some form of *dom/wdeg* as variable ordering heuristic or combine it with *impact*.

*Restarting Search* Restarts are important in modern SAT and CSP solvers because they can take account of the erratic behaviour of backtrack search algorithms on many combinatorial problems [10]. The introduction of restarts requires *diversification* of the exploration of the search space, that is, solvers should make sure they use different search paths each time a new run is started. One way to achieve this is by randomization and another way is by learning. One simple learning approach [1, 7, 16] identifies and records nogoods from the last branch of the search tree before each restart, ensuring that subsequent runs do not explore parts of the the search tree that have been explored previously. The importance of restarts (with or without nogood recording) is confirmed by the last competitions, where most of the constraint solvers embed this technique.

*Table Constraints* Extensional constraints are defined by explicitly listing the allowed or disallowed tuples. *Table constraints* are extensional constraints which usually are understood to be non-binary. They are important because they are easily handled by end-users of constraint systems. Some recent research articles have focused on theoretical and practical aspects of table constraints. As a result, there are many new ways to enforce Generalised Arc Consistency (GAC) on table constraints and/or to compress their representation. Observing the results obtained at the 2008 and 2009 competitions, we identified some effective approaches among the non-portfolio solvers.

In 2008, the best non-portfolio solver in category N-ARY-EXT (non-binary extensional constraints) was *mddc-solv* [5]. The benefit of converting tables into MDDs (Multi-valued Decision Diagrams) and using them to enforce GAC [3, 4] is clearly visible on some series of instances (series BDD, MDD and also some random instances). *Mistral* [13] and its variants (e.g. *MDG* [11]) were very efficient in 2008 on many instances of the category N-ARY-EXT. *Mistral* won this category in 2009. It used two algorithms for table constraints: *GAC3'-valid*, which uses residual supports and stores relations as bitsets, and *GAC2001-allowed*, which traverses the list of allowed tuples. Finally, the algorithm *STR* [26, 15] implemented in *Absson* was shown to be very efficient on some hard instances (e.g. see the crossword problem).

*Strong Consistencies* Most constraint solvers which were submitted to the last competitions maintain GAC during search. This confirms the transition, observed from the mid-90's, from FC (which maintains a partial form of arc consistency) to MAC (which maintains full arc consistency). The general trend is to extend the level of local reasoning in constraint solvers, so as to more efficiently solve hard problem instances. The next generation of solvers should certainly less restrictively enforce consistencies which are stronger than GAC (because for some problem instances, enforcing a "strong" consistency may have a dramatic impact).

This is already the case for some solvers which enforce derived forms of Singleton Arc Consistency (SAC). Nevertheless, we have to admit that practical progress has been confined mainly to preprocessing.

We notice that Abscon enforces existential SAC at preprocessing (a constraint network is checked to be existential SAC if and only if some value in the domain of each variable is proved to be singleton arc-consistent), Buggy<sub>2-5</sub> (from the 2006 competition) enforces a form of weak  $k$ -SAC [27] at preprocessing which is stronger than SAC, and CaSPER [6] enforces a time-bounded SAC at preprocessing (RSAC [18]). A restricted form of SAC is also employed by CaSPER when evaluating the search heuristics.

*Remarkable Solvers* Finally, the last competitions allowed us to identify some remarkable solvers. These solvers have been shown to be either quite robust (cpHydra, Mistral) or dominating in a particular category (Sugar).

cpHydra [17] is an algorithm portfolio that embeds several solvers and uses case-based reasoning to determine how to produce a schedule of solvers when solving an unknown problem instance. A case base of problem solving experience is built by solving various typical instances with each solver in the portfolio. The results obtained by cpHydra at the third solver competition show the effectiveness of this approach. If the interest of this solver cannot be denied, the correct evaluation of this kind of solver appears as a challenge because it must be evaluated mainly on new instances which were not used during its training. Clearly, it is the community's responsibility to generate a significant number of new instances each year in order to correctly evaluate this kind of solver.

Mistral [13] is a library that automatically chooses variable representations and filtering algorithms. For example, domains of integer variables can be implemented by bitsets or lists. The right implementation choice may have a dramatic effect [14]. Mistral and its variants obtained good results at the third and fourth solver competitions.

Sugar [24] is a SAT-based constraint solver that translates CSP instances into SAT instances using the order encoding method [23], and then runs a SAT solver such as MiniSat. Even if the quality of the evaluation benchmark for the category of global constraints can be questioned (because of the limited number of problems available), the result obtained by Sugar for this category is promising and opens new perspectives.

Finally, among solvers not already referenced earlier, let us mention the overall good behaviour of Choco [25] in different CSP categories, and Toulbar2 [22] for Max-CSP. More details about rankings may be found on the competition's website, which is available through <http://cpai.ucc.ie>.

## 4 Conclusion

The goal of constraint solver competitions is to boost research on the implementation of efficient solvers, as the SAT competitions did for SAT solvers. This is a real challenge in the CP community because of the limited number of publicly available solvers, the lack of a unified representation format, the huge number of constraints defined in the literature, and the different views on what a solver must do. We believe that the focus should be on generic, black-box solvers which do not require human intervention to perform well. The competitions have been organised in this spirit and we believe they have contributed by identifying interesting techniques and by drawing a fair picture of the current state-of-the-art. There is still room for improvement: collecting more instances and solvers is a necessary condition for the success of future competitions.

## Acknowledgements

We would like to thank the judges who served at different competitions, all people that helped organizing the competitions, the ACP, and 4C and CRIL for their support.

## References

1. L. Baptista, I. Lynce, and J.P. Marques-Silva. Complete search restart strategies for satisfiability. In *Proceedings of SSA'01 workshop held with IJCAI'01*, 2001.
2. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
3. M. Carlsson. Filtering for the case constraint, 2006. Talk given at the school on global constraints.
4. K. Cheng and R. Yap. Maintaining generalized arc consistency on ad-hoc r-ary constraints. In *Proceedings of CP'08*, pages 509–523, 2008.
5. K. Cheng and R. Yap. An overview of mddc-solve. In [28], pages 3–7, 2008.
6. M. Correia and P. Barahona. Overview of the CaSPER constraint solvers. In [28], pages 15–24, 2008.
7. A.S. Fukunaga. Complete restart strategies using a compact representation of the explored search space. In *Proceedings of SSA'03 workshop held with IJCAI'03*, 2003.
8. P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of ECAI'92*, pages 31–35, 1992.
9. I.P. Gent, C. Jefferson, and I. Miguel. Minion: A fast, scalable constraint solver. In *Proceedings of ECAI'06*, pages 98–102, 2006.
10. C. 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. D. Grimes. A comparison of boosted versus unboosted weighted degree search. In [28], pages 25–30, 2008.
12. R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
13. E. Hebrard. Mistral, a constraint satisfaction library. In [28], pages 31–39, 2008.
14. E. Hebrard. Implementing a constraint solver: a case study. Slides of the workshop of the third international constraint solver competition, <http://www.cril.univ-artois.fr/CPAI08>, 2008.
15. C. Lecoutre. Optimization of simple tabular reduction for table constraints. In *Proceedings of CP'08*, pages 128–143, 2008.
16. 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.
17. E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In [28], pages 53–62, 2008.
18. P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In *Proceedings of CP'00*, pages 353–368, 2000.
19. J.F. Puget. Constraint programming next challenge: simplicity of use. In *Proceedings of CP'04*, pages 5–8, Invited Talk, 2004.
20. P. Refalo. Impact-based search strategies for constraint programming. In *Proceedings of CP'04*, pages 557–571, 2004.
21. O. Roussel and C. Lecoutre. XML representation of constraint networks: Format XCSP 2.1. Technical Report arXiv:0902.2362, CoRR, 2009.
22. M. Sanchez, S. Bouveret, S. de Givry, F. Heras, P. Jegou, J. Larrosa, S. Ndiaye, E. Rollon, T. Schiex, C. Terrioux, G. Verfaillie, and M. Zytnecki. Max-CSP competition 2008: toulbar2 solver description. In [28], pages 63–70, 2008.
23. N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. In *Proceedings of CP'06*, pages 590–603, 2006.
24. N. Tamura, T. Tanjo, and M. Banbara. System description of a SAT-based CSP solver: Sugar. In [28], pages 71–75, 2008.
25. The Choco Team. Choco: an open source Java constraint programming library. In [28], pages 8–14, 2008.
26. J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177:3639–3678, 2007.
27. M.R.C. van Dongen. Beyond singleton arc consistency. In *Proceedings of ECAI'06*, pages 163–167, 2006.
28. M.R.C. van Dongen, C. Lecoutre, and O. Roussel, editors. *Proceedings of the third constraint solver competition*. <http://www.cril.univ-artois.fr/CPAI08/Competition-08.pdf>, 2008.
29. Neng-Fa Zhou bp-solver 2008. In [28], pages 83–90, 2008.