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 networks. 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 multidirectional 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(μed) 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 (approximatively 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 reason 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 subnetwork from each node proved to be the root of an unsatisfiable subtree. Theses subnetworks 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. 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.