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

Proceedings of the Third International CSP Solver Competition[1]

Held in conjunction with the
Thirteenth International Conference on
Principles and Practice of
Constraint Programming (CP 2008)

[1] Sponsored by the Association for Constraint Programming.

II

Preface

The Third 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 the description of some of the solvers submitted to the competition. Although these descriptions have not been formally reviewed, we believe that they contain valuable information which deserves to be considered with careful attention by current and future developers of constraint systems. Due to time constraints it has been impossible to present the final ranking of the solvers in these proceedings. The rankings will be presented at CP'08 and a paper will be submitted to a journal.

For this third edition, we considered instances involving constraints defined in extension and in intension (i.e. by a predicate). In addition to the global $allDifferent$ constraint, which was already present for the second competition, the global constraints $cumulative$, $element$, and $weightedSum$ have been introduced. Solvers have been evaluated for (Ordinary) CSP and Max-CSP, considering instance categories defined by the arity of the constraints (binary, non-binary) and by their representation (extension, intension, global).

This year there have been three organisational changes. The first change is the creation of a Working Committee which is responsible for the creation of new instance classes and tools for converting instances from different languages to the competition's specification format. The second change is the installation of two independent judges, which are not related to any of the teams. The main task of the judges is to ensure that the selection of the instances and key decisions are made in a fair and transparent way. The third and final organisational change is the creation of a Problem Selection Committee, which was installed by the independent judges. Installing the judges and the Problem Selection Committee has reduced the possibility of interference of the main organisers with problem selection and other decisions which may affect the outcome of the competition.

The original version of these proceedings have been made available on a DVD. The contents of the DVD are the proceedings, the normalised problem instances which were used for the final evaluation, the original instances from which the normalised instances were obtained, and the satisfiability of the problem instances.

September 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 2008 version of the CSP Solving Competition into a successful event:

- The independent judges Pierre Flener (Upsala University), Deepak Mehta (4C), and Rick Wallace (4C) for selecting the final instances for the competition and for guaranteeing a fair competition.
- The members of the Working Group Emmanuel Hebrard (4C), Barry O'Sullivan (4C), Andrea Rendl (University of St Andrews), and Sebastien Tabary (Université d'Artois) for providing new problem instances.
- Leonid Ioffe (4C) and James Little (4C) for providing the Cabinet Problem instances.
- CRIL at University of Artois for providing their machines for the final evaluation.
- The contestants for participating.

Main Organisers

Chirstophe Lecoutre	Centre de Recherches en Informatique de Lens, Université d'Artois, France
Olivier Roussel	Centre de Recherches en Informatique de Lens, Université d'Artois, France
Marc van Dongen	University College Cork, Ireland

Independent Judges

Piere Flener	Upsala University, Sweden
Rick Wallace	Cork Constraint Computation Centre, Ireland

Problem Selection Committee

Deepak Mehta	Cork Constraint Computation Centre, Ireland

Competition Working Group

Emmanuel Hebrard	Cork Constraint Computation Centre, Ireland
Barry O'Sullivan	Cork Constraint Computation Centre, Ireland
Andrea Rendl	University of St Andrews, UK
Sebastien Tabary	Centre de Recherches en Informatique de Lens, Université d'Artois, France

Table of Contents

An overview of mddc-solv

Kenil C.K. Cheng and Roland H.C. Yap

School of Computing
National University of Singapore
{chengchi, ryap}@comp.nus.edu.sg

Abstract. This paper describes the design and implementation of mddc-solv.

1 General Description

Written in C++, mddc-solv is a complete solver participating in the category N-ARY-EXT in CSP 2008 Competition. Its design and implementation aim at handling large extensional constraints with high arity, rather than (small) binary or ternary constraints. The overall architecture is as follows:

The variable ordering heuristic is $dom/wdeg$ [1], which selects the variable x with smallest

$$\frac{|dom(x)|}{\sum_{x \in var(C)} w(C)}$$

where $var(C)$ is the scope of the constraint C and $w(C)$ is the weight of C. Initially, the weight is one. Whenever a constraint is found inconsistent during propagation, its weight is incremented by one. The weights are never reset.

The value ordering heuristic is static, where the most supported value is tried first [6].

Two-way branching is used, with the branch $x = a$ chosen before $\neq a$.

Geometric restart with nogood learning [5] was implemented. The initial cutoff is 3 times the number of variables, and is subsequently multiplied by 1.5 upon restart, which occurs when the number of fails (due to $x = a$) exceeds the current cutoff.

Generalized arc consistency (GAC) is the only consistency available in the solver; in particular, we implemented mddc [4], a coarse-grained GAC algorithm based on multi-valued decision diagram (MDD) [7]. In mddc-solv, all extensional constraints, binary or non-binary, are represented as MDDs. To enforce GAC on a constraint, mddc traverses the MDD recursively and updates the domains of the relevant variables on the fly. Because of the compactness of MDDs, mddc may run exponentially faster, with exponentially less memory, than GAC algorithms based on tables (arrays or bit vectors). Since our current implementation of mddc-solv has no differentiation between boolean and non-boolean constraints, we didn't implement bddc [3], which is a scale-down version of mddc and includes special features such as caching.

During constraint propagation, the constraint with largest

$$\frac{w(C)}{\prod_{x \in var(C)} |dom(x)|}$$

is checked first. Intuitively speaking, the weight records how often a constraint was found inconsistent in the past, whereas the product of domain's size estimates how likely the constraint is going to be inconsistent with respect to the current search space.

The domain of a variable is represented as a set of pruned values, which was implemented with the same sparse set data structure [2] used in mddc.

We implemented the table-to-MDD procedure (mddReduce) and the GAC algorithm (mddc) as presented in [4]; some competition-specific changes have been made, which are described in the remaining sections. Readers may refer to the original papers [3, 4] for details. Beforehand, we reproduce the definitions of an MDD and an MDD constraint.

Definition 1. *A multi-valued decision diagram (MDD) [7] is either the* t-terminal (`tt`), *the* f-terminal (`ff`), *or a directed acyclic graph of the form*

$$G = \texttt{mdd}(x, \{a_1/G_1, \ldots, a_d/G_d\})$$

where $G_1, \ldots, G_d$ are MDDs and $a_1, \ldots, a_d$ are distinct integers. Each pair a_k/G_k $(1 \leq k \leq d)$ is a branch *of G, and G_k is a* sub-MDD *of G.*

Both `tt` and `ff` are used because in the competition mddc has to deal with both positive and negative constraints (the former is defined as a set of solutions while the latter a set of non-solutions). Fig. 1a depicts a small MDD.

Definition 2. *An MDD constraint (represented by an MDD G) is a logical constraint*

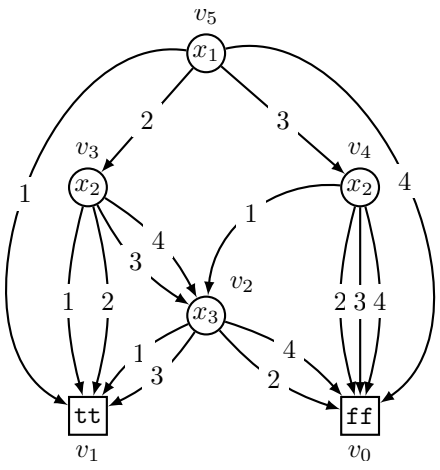
$$\Phi(G) \equiv \begin{cases} \mathit{True} & : \ G = \texttt{tt} \\ \mathit{False} & : \ G = \texttt{ff} \\ \bigvee_{k=1}^{d} (x = a_k \wedge \Phi(G_k)) & : \ G = \texttt{mdd}(x, \{a_1/G_1, \ldots, a_d/G_d\}) \end{cases}$$

2 Building an MDD for an Extensional Constraint

Since in the competition, an extensional constraint is represented as its set of (non-)solutions in lexicographical order, we can build the MDD without first constructing a trie explicitly [4]. Fig. 2 illustrates the ideas.

3 Modified mddc for Negative Constraints

We slightly modified mddc (lines 1, 2 in Fig. 3) for negative constraints.



(a)

	1	2	3	4
0	-	-	-	-
1	-	-	-	-
2	1	0	1	0
3	1	1	2	2
4	2	0	0	0
5	1	3	4	0

(b)

Fig. 1: (a) A graphical representation of a multi-valued decision diagram (MDD). Each non-terminal node v_k is labeled with a variable x_i. An outgoing edge of v_k with label a depicts an assignment (x_i, a). (b) An MDD is implemented as a static two-dimensional transition table, where the row k corresponds to the MDD node v_k and the column a corresponds to the assignment (x_i, a), i.e., each cell corresponds to an MDD edge.

References

1. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *European Conference on Artificial Intelligence*, pages 146–150, 2004.
2. P. Briggs and L. Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1–4):59–69, 1993.
3. K. C. K. Cheng and R. H. C. Yap. Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints. In *European Conference on Artificial Intelligence*, pages 78–82, 2006.
4. K. C. K. Cheng and R. H. C. Yap. Maintaining generalized arc consistency on ad hoc r-ary constraints. In *International Conference on Principles and Practice of Constraint Programming*, 2008, to appear.
5. C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Recording and minimizing nogoods from restarts. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:147–167, 2007.
6. D. Mehta and M. R. C. van Dongen. Static value ordering heuristics for constraint satisfaction problems. In *CPAI'05 workshop held with CP'05*, pages 49–62, 2005.
7. A. Srinivasan, T. Kam, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In *Computer Aided Design*, pages 92–95, 1990.

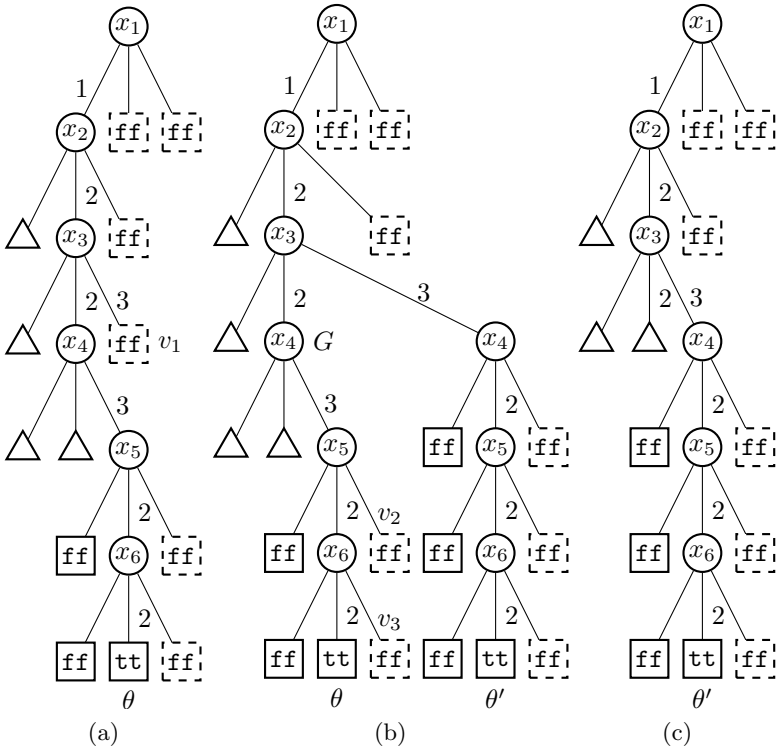


Fig. 2: (a) A semi-constructed MDD for a positive constraint. Every triangle depicts a fully constructed sub-MDD. The terminal node in a dashed box means it is default and temporary (see below). For a positive (negative) constraint, the default is `ff` (`tt`). Arrowheads are omitted and only relevant edges are labeled. The last input solution is $\theta = \{(x_1, 1), (x_2, 2), (x_3, 2), (x_4, 3), (x_5, 2), (x_6, 2)\}$. Now, suppose another solution $\theta' = \{(x_1, 1), (x_2, 2), (x_3, 3), (x_4, 2), (x_5, 2), (x_6, 2)\}$ has just been read. In order to insert θ' into the MDD, the temporary node v_1 is replaced with a new branch. The modified MDD is in (b). We then compress G into a new sub-MDD because v_2 and v_3 are fixed (no solution will share the prefix $\{(x_1, 1), (x_2, 2), (x_3, 2), (x_4, 3)\}$). This is due to the competition rule that requires the solutions of an input constraint to be in lexicographic order. The updated MDD is shown in (c). Notice that θ' is the new frontier: all sub-MDDs on its left are fully constructed. An MDD for a negative constraint can be made in a similar fashion.

```
mddc(G) /* MDD constraint Φ(G)(x_1, ..., x_r) */
begin
    𝒢^YES := ∅
    restore(𝒢^NO)
    for i := 1 to r do S_i := dom(x_i) /* values that have no support yet */
    mddcSeekSupports(G) /* update S_1, ..., S_r */
    for i := 1 to r do dom(x_i) := dom(x_i) \ S_i
end

mddcSeekSupports(G) /* recursive: Φ(G)(x_i, ..., x_r) */
begin
1   if G = tt then
        for j := i to r do
            S_j := ∅ /* True admits any domain values */
        return YES
2   if G = ff then return NO
    if G ∈ 𝒢^YES then return YES /* visited and consistent */
    if G ∈ 𝒢^NO then return NO /* pruned */
    /* let G = mdd(x_i, {a_1/G_1, ..., a_d/G_d}) */
    res := NO
    for k := 1 to d do
        if a_k ∈ dom(x_i) then
            if mddcSeekSupports(G_k) = YES then
                res := YES
                S_i := S_i \ {a_k}
                if ∀i' ≥ i : S_i' = ∅ then break /* Φ(G)(x_i, ..., x_r) is GAC */
    𝒢^res := 𝒢^res ∪ {G}
    return res
end
```

Fig. 3: Modified mddc and mddcSeekSupports (cf. [4])

choco: an Open Source Java Constraint Programming Library

The choco team*
website: choco.emn.fr
contact: choco@emn.fr

École des Mines de Nantes
LINA CNRS
4 rue Alfred Kastler – BP 20722
F-44307 Nantes Cedex 3, France

Abstract. choco is a java library for constraint satisfaction problems (CSP) and constraint programming (CP). It is built on a event-based propagation mechanism with backtrackable structures. choco is an open-source software, distributed under a BSD licence and hosted by sourceforge.net. choco is mainly developped by people at École des Mines de Nantes (France) and is financially supported by Bouygues SA and Amadeus SA.

1 Introduction

choco originated in 1999 within the OCRE project, a French national initiative for an open constraint solver for both teaching and research involving researchers and practitionners from Nantes (École des Mines), Montpellier (LIRMM), Toulouse (INRA and ONERA) and Bouygues SA. Its first implementation was in CLAIRE [CJL02], Yves Caseau's language that compiled into C++. It has been used since then as a teaching tool and the main constraint programming developping tool in the Constraint Programming research group in Nantes.

In 2003, choco went through its premiere major modification when it has been implemented into the Java programming language. The objective was to ensure a greater portability and to ensure an easier takeover for newcomers. As for this time, choco really started its worldwide expansion.

In 2008, choco is being taken a step further. Thanks to the hiring of a full-time engineer on the solver (financed by École des Mines de Nantes, Bouygues SA and Amadeus SA), choco enters a new period of its development. As the v2 version is shipped on Sep. 10^{th}, 2008, it offers a clear separation between the model and the solving machinery (providing both modelling tools and innovative

* The choco team is composed of people from École des Mines de Nantes (including Narendra Jussien and Charles Prud'homme), Cork Constraint Computation Center (including Hadrien Cambazard), Bouygues e-lab (including Guillaume Rochart) and Amadeus SA (including François Laburthe).

solving tools), a complete refactoring improving its general performance, and a more user-friendly API for both newcomers and experienced CP practionners.

choco v2 is an open system distributed as a sourceforge project under a BSD license authorizing all possible usages. It is a glass box (all the sources are provided) for teaching (illustrating and implementing all major concepts of Constraint Programming), research (its open API allows an easy integration of personal state-of-the-art algorithms and concepts within the solver) and problem solving (it is now used in real-life contexts in several companies).

In a few words, choco is an efficient yet readable constraint system for research and development ; choco is a readable yet efficient constraint system for teaching.

2 choco's general features

choco is a problem modeler and a constraint programming solver available as a Java library. Moreover, its architecture allows the plugin of other (non CP based) solvers.

2.1 A Problem Modeler

choco is a problem modeler able to manipulate a wide variety of variable types (all considered here as first-class citizens):

- integer variables;
- set variables representing sets of integer values;
- real variables representing variables taking their value in an interval of floats;
- expressions representing a integer- or real-based expression using operators such as *plus*, *mult*, *minus*, *scalar*, *sum*, etc.

choco 's modeler accepts over 70 constraints (provided that the called solver will be a CP-based solver):

- all classical arithmetical constraints (or integers or reals): equal, not equal, less or equal, greater or equal, etc.;
- reified constraints *i.e.* boolean operations between (possibly reified) constraints;
- table constraints defining the sets of tuples that (do not) verify the intended relation for a set of variables;
- a large set of useful classical global constraints including the `alldifferent` [Rég94] constraint, the `global cardinality` [Rég96] constraint, the `nvalue` [BHH+05] constraint, the `element` [BC94] constraint, the `cumulative` [AB93] constraint, etc.

Moreover, choco provides access to the most recent state-of-the-art implementations of global constraints produced in Nantes, France including the `tree` [BLF08] constraint and the `geost` [BCP+07] constraint.

Finally, the implementation of the `regular` [Pes04] and `cost-regular` [DPR06] constraints provide an automatic[1] access to all global constraints whose checker is either a deterministic finite automaton or a DFA with (array of) counters as described in the Global Constraint Catalog [BCDP07].

2.2 A Constraint Programming solver

`choco` is (as expected) a constraint programming solver. It provides:

- several implementations of the various domain types (*eg.* enumerated, bounded, list-based, ... integer variables);
- several algorithms for constraint propagation (state-of-the-art AC algorithms for table constraints, full and bound `alldifferent`, parameterized `cumulative`, etc.).

`choco` can either be used in satisfaction mode (computing one solution, all solutions or iterating them) or in optimization mode (maximisation and minimisation). Search can be parameterized using a set of predefined variable and value selection heuristics (including impact-based search [Ref04] and domain over weighted degree [BHLS04]). User's parameterization includes designing her own variable and/or value selectors, as well as precising which should be the decision variables and even designing cascading variable/value selectors for different sets of variables.

Finally, when converting the model into a solver-specific problem, `choco` can enter into a *pre-processor* mode that will perform some automatic improvements[2] in the model. `choco` is initially a solver working with intensional constraints and therefore the *pre-processor* attempts to use the intensional constraints available in choco whenever this is possible. It does the following operations:

- choose a level of consistency e.g : `alldifferent` or `boundAlldifferent`; arc-consistency or forward-checking for extensionnal constraints; arc-consistency on complex expressions or a weaker form of consistency resulting from the decomposition of the expression by introducing intermediate variables;
- compute maximal cliques in the constraint graph of binary differences or disjunctions to state the `alldifferent` global constraint or the `disjunctive` global constraint;
- recognize intentional constraints stated as expressions (or predicates) to state the appropriate intentional constraint : distances ($|x - y| < z$), linear equations, min/max constraints. Some constraints stated extensionnaly such as differences or equalities are also recognized;
- simple value symmetry breaking in case of pure coloring problems.

[1] This automatization will be available on the January 2009 release of `choco` .
[2] Those improvements were investigated during the solver competition.

3 choco's design

choco is a Java library that chose to provide a clear separation between modeling and solving. Figure 1 represents the overall architecture of the choco library. There are two separate parts:

- the first part (from the user's point of view) is devoted to expressing the problem. The idea is to manipulate variables and relations to be verified for these variables (constraints) disregarding their potential implementation (either from the variable point of view or the constraint point of view). A complete API is provided to be able to state a problem in a way as user-friendly as possible.
- the second part is devoted to actually solve the problem. In Figure 1, only CP related information is provided. Solving includes specific memory management for tree-based search (as in CP).

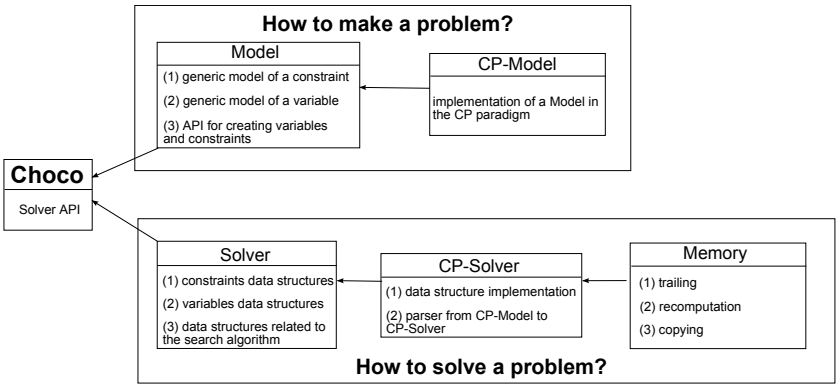


Fig. 1. choco 's general architecture. The separate parts are clearly identified: a modelling part for stating the problem and a solving part (here only the CP related information is described) for actually solving the modelled problem.

This clear separation between model and solver has been introduced to ease the usage of constraint programming to newcommers. This architecture is meant to let newcommers focus on the modeling part of their problem and rely on the *pre-processor* of choco that will take over the user to translate its model into a more CP-like model to be automatically solved by the solver. However, any CP practitionner or CP specialists is left the right to:

- make annotations within the model to force the *pre-processor* to use specific implentations and ways of handling constraints (for example when considering expressions) ;

– use the solver's API to directly program or modify the default behavior of the solver.

choco in its new version has been designed for allowing tree search related solvers to be integrated within the platform. For example, we are currently porting PaLM [JB00], an explanation-based constraint solver which does not rely on a tree-based exploration of the search space. The idea is to provide to the end user of choco a natural and effortless way to use a given constraint model in different contexte (explanation-based constraint programming, local search, etc.)

4 choco in practice

Here is a few lines of code to get the essence of using choco in practice. Notice the use of annotations when building variables (cp:enum) and the explicit separation between Model and Solver.

```
//1- Create the model
Model m = new CPModel();
int n = 6;
//2- declaration of variables
IntegerVariable[] vars = makeIntVarArray("v", n, 0, 5, "cp:enum");
IntegerVariable obj = makeIntVar("obj",0,100,"cp:bound");

//3- add some constraints
String regexp = "(1|2)(3*)(1|4|5)";
m.addConstraint(regular(regexp, vars));
m.addConstraint(neq(vars[0], vars[5]));
m.addConstraint(eq(scalar(new int[]{2,3,1,-2,8,10}, vars), obj));

//4- Create the solver
Solver s = new CPSolver();

//5- read the model and solve it
s.read(m);
s.solve();
if (s.isFeasible()) {
    do {
        for (int i = 0; i < n; i++) {
            System.out.print(s.getVar(vars[i]).getVal());
        }
        System.out.println("");
    } while (s.nextSolution());
}
//6- Print the number of solutions found
```

```
System.out.println("Nb_sol : " + s.getNbSolutions());
```

Which gives the following output :

```
133334 72
133335 82
233331 44
233334 74
233335 84
Nb_sol : 5
```

5 choco as a teaching and research tool

choco is used in many different places for teaching. For example, in France, the universities of Nantes, Montpellier, Rennes, Toulouse, Clermont-Ferrand; the engineering schools of École des Mines de Nancy, École des Mines de Nantes, École Nationale Supérieure des Sciences et Techniques Appliquées, etc. all use choco for teaching constraint programming. choco is not necessarily the only solver that is presented but one of its asset is that it is an open solver whose source code can be browsed and understood easily.

choco is also used in R&D divisions in several companies including Bouygues SA, Amadeus SA but also Dassault Aviation; research agencies such as ONERA and even NASA. It is worth noticing that a company has been created in France which exclusively uses choco as its optimization tool: KLS optim (http://klsoptim.com/).

6 Conclusion

choco is an open, user-oriented constraint solver which provides a clear separation between model and solver. It paves the way to provide a general problem solving library not necessarily dedicated to constraint programming. It is improving every day and eager to integrate user improvements, new constraints, new solvers, propositions, etc.

Visit choco.emn.fr for the latest news, the current version, teaching material, documentation, etc. about choco

Acknowledgments

choco would not exist without its founding fathers: François Laburthe (Amadeus, SA) and Narendra Jussien (École des Mines de Nantes), its core team: Hadrien Cambazard (4C, Cork) and Guillaume Rochart (Bouygues SA), its new generation of developers and contributors: Charles Prud'homme (EMN – project

management), Xavier Lorca (EMN – teaching, training), Guillaume Richaud (EMN – development), Julien Menana (EMN – development), Arnaud Malapert (EMN and University of Montreal – development) and its funding fathers: École des Mines de Nantes which hosts the choco web site, servers, project manager, Bouygues SA and Amadeus SA.

References

[AB93] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57–73, 1993.

[BC94] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Math. Comput. Modelling*, 20(12):97–123, 1994.

[BCDP07] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, 2007.

[BCP+07] Nicolas Beldiceanu, Mats Carlsson, Emmanuel Poder, R. Sadek, and Charlotte Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In *CP*, pages 180–194, 2007.

[BHH+05] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Filtering algorithms for the nvalue constraint. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR?05), volume 3524 of LNCS*, pages 79–93. Springer-Verlag, 2005.

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

[BLF08] N. Beldiceanu, X. Lorca, and P. Flener. Combining tree partitioning, precedence, and incomparability constraints. *Constraints*, 13(4), 2008.

[CJL02] Yves Caseau, François-Xavier Josset, and François Laburthe. Claire: combining sets, search and rules to better express algorithms. *Theory Pract. Log. Program.*, 2(6):769–805, 2002.

[DPR06] Sophie Demassey, Gilles Pesant, and Louis-Martin Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11(4):315–333, 2006.

[JB00] Narendra Jussien and Vincent Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.

[Pes04] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer, 2004.

[Ref04] P. Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming (CP'04*, pages 556–571, 2004.

[Rég94] J.-C. Régin. A filtering algorithm for constraints of difference in CSP. In *AAAI'94*, pages 362–367, 1994.

[Rég96] J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *AAAI'96*, pages 209–215, 1996.

Overview of the CaSPER* Constraint Solvers
(submitted to the CPAI08 solver competition)

Marco Correia and Pedro Barahona

Centro de Inteligência Artificial, Departamento de Informática,
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
{mvc,pb}@di.fct.unl.pt

Abstract. This paper describes the *casperzito* and *casperzao* constraint solvers submitted to the CPAI08 solver competition. These solvers are instances of the CaSPER library, an open C++ library for constraint solving that includes a set of specialized propagators for integer global constraints taken from recent literature. Additionally, we perform automatic symmetry detection and symmetry breaking, and implement original work on impact based search and search strategy sampling.

1 Introduction

CaSPER (Constraint Solving Programming Environment for Research) is an open C++ library for generic constraint solving. Its outstanding features are custom propagator scheduling, efficient domain delta information availability, and a user friendly modeling and searching interface available directly from the programming language. These features are implemented at the library core, with the additional functionality of any typical constraint solver, such as event-driven execution, callback scheduling, garbage collection, state handling, and trail-aware generic data structures. Domain specific reasoning extends the kernel in a modular fashion - currently there are modules for finite domain variables [18], finite set domain variables [3], graph domain variables [37], generic interval-based reasoning [5] and for 3d space reasoning [20].

Having being idealized to accommodate a quickly changing research environment, the library's design is utterly committed to flexibility and openness. The implementation is based on generic programming patterns, which have been proved successful for achieving this goal [25,1,36,9,35].

For the competition we created two instances of the library's finite domain constraint solver, named *casperzito* (light casper) and *casperzao* (heavy casper). Both solvers implement a typical finite domain framework with a set of specialized propagators for global constraints taken from recent literature. We followed the approach of [29,31] for breaking symmetries which extends naturally to the set of global constraints used in the competition. Additionally, we integrated original work on impact based search [10], and a new propagator for achieving GAC on negative table constraints.

In the following section we will briefly describe the propagation model used in the solvers. Section 3 focus on the symmetry breaking techniques used, and

in section 4 we describe the strategies applied to explore the search space. A preliminary analysis and discussion of the results is attempted in section 5, and we conclude in section 6 with some closing remarks and pointers to future work.

2 Propagation

2.1 Propagator scheduling

For many constraints CaSPER provides more than one propagator, usually attaining different consistency levels. When more than one propagator exists for the same constraint, then all of them will eventually participate in the fixpoint calculation. The procedure is governed by a method known as staged propagation [34] which basically sorts propagators for execution based on an individual (hard coded) propagation cost. While there exists other methods for scheduling propagators with better performance for some class of problems [12], we settled with this one for its robustness.

2.2 Predicates

Except for global constraints (see below), arithmetic predicates used in the competition are enforced in CaSPER using bounds consistency. In many instances used in the competition, predicates are grouped together in larger predicates (using conjunctions), and we found that enforcing bounds consistency in the decomposition was sometimes penalizing performance. To solve this problem, we translated these conjunctions of predicates to positive or negative table constraints (whichever is smaller) by solving the corresponding subproblems before search, and enforced GAC on these constraints during search. We only did this in *zao*, since we were not sure this was a good idea.

2.3 Global constraints

Table 1 describes the propagators used for the global constraints in the competition.

constraint	value	bounds	domain
positive table	no	no	[6,16] (a)
negative table	custom (b)	no	custom (b)
distinct	custom	[22]	[33]
element	no	custom	custom
linear	no	[38]	no
cumulative	no	[4,24]	no

Table 1. Global constraint propagators used in solvers.

For the domain consistency propagator for positive table constraints (a) we used the first algorithm [6] on *zito* and the new trie-based propagator of [16] in *zao*. While the latter exhibited better results in our tests, we still found the first more efficient for small arity constraints.

The most popular algorithm for propagating the negative table constraint (b) seems to be the two watched literal scheme introduced for SAT, which achieves value (node) consistency. In [6], a domain consistency algorithm for this constraint that makes heavy use of hashing for checking disallowed tuples was presented. We have extended the work of [16] which focus on positive table constraints to handle negative table constraints as well. While we also base our idea in the trie data structure, this is in fact a completely different propagator which has the advantage of performing much cheaper tests compared to the hashing proposal (details will be on a forthcoming paper). For the competition we used the value consistency propagator for negative table constraints in *zito* and our new trie-based propagator in *zao*.

3 Symmetry breaking

We mostly followed the ideas in [29] for automatic symmetry detection using computational group theory and [2,31] for symmetry breaking. The basic idea of the detection process is to translate the given CSP to a graph which expresses the symmetries associated with each constraint. The automorphism group of this graph defines the set of symmetries in the original CSP. In [29], Puget shows how to translate some common global constraints, e.g. the alldifferent constraint. Extending the idea for the predicates and global constraints used in the competition is not difficult. Additionally, both solvers perform a small amount of symbolic computation in order to circumvent some situations where the symmetries in the CSP would be hidden by the formulation. Although the detection process is able to identify both variable and value symmetries, we just focused on the first kind[1].

Since the number of detected variable symmetries is quite large for most problems, we followed the method of [2], that is we restrict to the variable symmetries present on the generators of the symmetry group (also referred as the GEN class in [30]). For actually breaking the symmetries we added a number of lexicographic ordering constraints [8] before starting search, which is a popular technique known as static symmetry breaking [28] (SSB). Moreover, both solvers do some effort to identify symmetries in sets of variables known to be all different, in which case we break symmetries by enforcing a stronger partial strict ordering (see also [31]).

It is known that SSB may potentially make the task of solving a satisfiable problem harder, since it can prune the solutions that would be found first by the search heuristics. In our preliminary tests we also tried breaking symmetries using the dynamic lex method [27] which does not have this drawback. Despite

[1] We adapted the code from saucy, a graph automorphism generator [13].

performing slightly better, this method requires a fixed value selection ordering, which we found too restrictive for our exploration strategies described in the next section.

4 Search

4.1 Heuristics

It is well known that variable and value selection heuristics play a crucial goal for guiding search towards a solution, or for proving that no solution exists. Recently, the *dom/wdeg* [7] heuristic has been given a lot of attention, although we have found that impact based heuristics [32] perform better for some problems [11]. For the competition we considered a portfolio composed of the *dom/wdeg* and impact variable heuristics for the *zito* solver, and also the *lookahead* variable heuristic (as described in [11]) for the *zao* solver.

While there has been recent work aiming at informed value selection heuristics [19], the most popular is probably the *min-conflicts* which selects the value having less conflicts with the values of other variables. Other common value selection heuristics select the values in increasing ordering (*min*), or just randomly (*rand*). Unfortunately, for the competition we didn't have time to implement anything more sophisticated than the *min* and *rand* value selection ordering.

4.2 Sampling

In order to choose which variable-value heuristic combination is finally used for solving a given instance, we introduced a sampling phase in the solving process (alg. 1). We evaluate each strategy based on the criteria of first-failness and best-promise [17,15]. Roughly, first-failness is the ability of the heuristic to easily find short refutations for large regions of the search tree that contain no solutions, while best-promise characterizes the potential to guide search quickly towards a solution. Typical search strategies combine these two components, usually by associating first-failness with the variable selection heuristic, and best-promise with the value selection heuristic.

Informally, our sampling phase works by performing several time bounded search runs (restarts) with each possible strategy while collecting information regarding its behavior. The time slice is increased geometrically from one run to the next in order to provide a basis for projecting the behavior of the strategy on a real (time unbounded) search run. After each run we compute an approximation of the ratio of the explored search space by analyzing the visited search tree, and store this information in F. After the sampling process, we compute from F an estimate of the first-failness and best-promise coefficients for each variable and value heuristic and select the best combination. Although the approximation makes a strong assumption that the search tree is uniformly balanced, our preliminary tests revealed that most of the times this method selects the best heuristics, specially when the choice of heuristic is crucial.

Algorithm 1: Search strategy sampling

Input: A set $\mathcal{S}$ of possible exploration strategies, initial and final time slice T_i,T_f, and geometric ratio r
Output: One of **SAT**,**UNSAT** or $\langle$**UNKNOWN**, $F\rangle$
$F \leftarrow \{\}$
foreach $s \in \mathcal{S}$ **do**
  $t \leftarrow T_i$
  while $t \leq T_f$ **do**
    $f \leftarrow$ `search`(s,t) `/* Search with strategy` s`, timeout at` t`. */`
    if f=**SAT or** f=**UNSAT then**
      return f
    $e \leftarrow$ `ratioOfExploredSearchSpace`()
    $F \leftarrow F \cup \langle s, t, e\rangle$
    $t \leftarrow t \times r$
return $\langle$**UNKNOWN**, $F\rangle$

4.3 SAC

Enforcing singleton arc consistency on a constraint network is a popular pruning technique [14], although its time complexity can be limiting. For the competition, both solvers enforce a time bounded SAC on the first propagation only. However, given that RSAC [26], a restricted form of SAC, is achieved while evaluating the lookahead heuristic (only on *zao*), then it may happen that RSAC is always enforced on some instances if it is selected by the method described in the previous section.

4.4 Restarts

For exploring the search tree we employed depth first search with time bounded restarts. Completeness is guaranteed by increasing the time allowed for each restart. We used 2.5 as the geometric ratio.

5 Experimental evaluation

The following discussion will be based on preliminary results which were made available to the contestants of the competition at the time of this writing.

Currently CaSPER does not implement any learning techniques, smart backjumping methods, constraint network analysis and (de)composition, or specialized data structures for CSPs given in extension (apart from table constraints). We think that these are required to be competitive in all categories except the GLB category, and perhaps on the set of instances from the INT and NINT categories that are too large to convert to extensional form. We will therefore

focus on the global constraints category only, despite the solvers are running in all of them[2].

Table 2 summarizes the distribution of the previously discussed features among both solvers.

	zito	*zao*
static symmetry breaking	yes	yes
predicate tabling	no	yes
lookahead var heuristic	no	yes
heuristic sampling	yes	yes
GAC for negative table	no	yes

Table 2. Summary of features in each solver.

The solver *zito* was able to solve 397 instances, while the solver *zao* solved 390 instances of a total of 556 instances. In order to assess their performance across the instance space, we plot in fig. 1 the percentage of instances solved by both solvers for each set of instances solved by a specific number of solvers. The rationale is that instances solved by less solvers should be harder than those solved by more solvers (meaning that the instance solving difficulty decreases along the x axis in the figure).

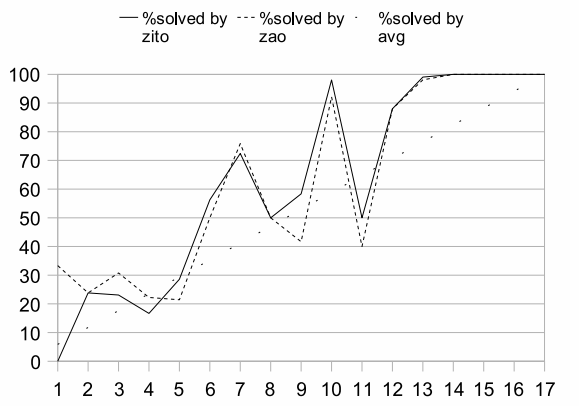


Fig. 1. Percentage of instances solved by *zito* and *zao* (yy) from the sets of instances solved by a number of solvers (xx). The dotted line shows the number of instances solved by an hypothetical average solver (assuming that solved instances distribute uniformly across all solvers).

[2] Additionally, there was a bug in the propagator achieving bounds consistency for the expression $mod(X, Y) = Z$ which caused both solvers to be disqualified from the INT category.

The performances of both solvers are quite similar, although as expected the solver *zao* is better on the hard instances, while *zito* is slightly better on the medium and easy instances. Comparing to other solvers, the performance of both solvers is almost always above the average, with *zao* performing significantly better than the average on the hard instances. The peaks in the performance chart suggest that there are sets of problems for which both solvers are not using the best techniques.

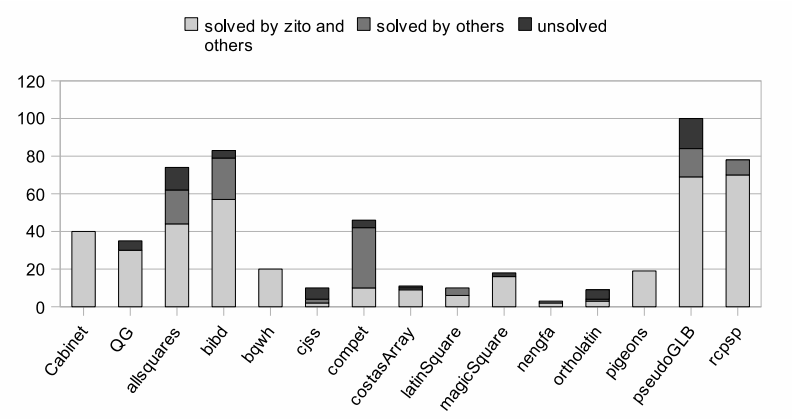


Fig. 2. Number of instances solved by *zito* in each problem family.

Figure 2 shows the number of instances solved by *zito* distributed across each family[3] of problems. Considering the instances that were solved by some solver, *zito* seems be fairly competitive except on the *allSquares*, *bibd* and specially in the *compet* family.

A final analysis on the strengths and weakness of the techniques used in our solvers will be made once we know the ranking and the techniques used by other solvers. Currently we can only point two known weak points of our approach that will eventually have a negative effect on our final position. The first is the lack of a smart value heuristic, and the second is the known conflicting issues between search heuristics and the method we used for breaking symmetries (SSB). Both problems can only affect the solving of SAT instances, and this may explain why the number of unsolved SAT instances is about six times the number of unsolved UNSAT instances (see fig. 3).

[3] We define a family as the set of all instances of a given problem, in contrast with the grouping used in the competition which often splits instances of the same problem into smaller groups (called series), e.g. allSquaresSAT and allSquaresUNSAT.

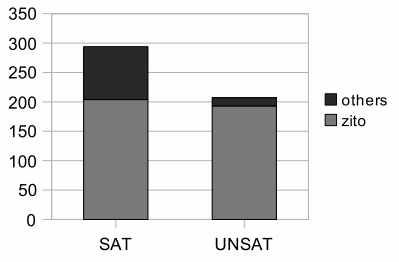


Fig. 3. Number of SAT and UNSAT instances solved by *zito*.

6 Conclusion

This paper describes the *casperzito* and *casperzao* constraint solvers as submitted to the CPAI08 solver competition. These solvers are small instantiations of the much larger CaSPER library which aims to provide a comprehensive environment for doing applied research in constraint programming.

After a brief description of the propagation model, we focused on the less standard features which were added specifically to the competition, such as automatic symmetry detection and symmetry breaking, or are product of our own research, such as search strategy sampling, and impact based search. Finally, we have made a preliminary analysis of the results from the competition, which suggest that the overall performance of both solvers is above the average.

There is much room for improvement, both in the black box solvers submitted to the competition and more extensively in the CaSPER library. At the propagation level a careful analysis on the best propagator to use for a given constraint when there is more than one choice could lead to significant speedups. The symmetry breaking framework could be made more dynamic and complete and extended to value symmetries as well. Search would certainly benefit from ideas such as learning from restarts [21], conflict-based static value ordering [23] and better integration of our own work on impact based search.

As short term goals we intend to formalize the search sampling procedure described in section 4, and perform consistent testing of trie-based data structures for propagating GAC on negative tables.

The CaSPER library is currently being developed at CENTRIA, and can be found at `http://proteina.di.fct.unl.pt/casper`.

References

1. Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
2. Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for boolean satisfiability. *IEEE Transactions on Computers*, 55(5):549–558, 2006.

3. Francisco Azevedo. Cardinal: A finite sets constraint solver. *Constraints journal*, 12(1):93–129, 2007.
4. Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In *CP*, pages 63–79, 2002.
5. Fréderic Benhamou. Interval constraint logic programming. In Andreas Podelski, editor, *Constraint programming: basics and trends*, volume 910 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 1995.
6. Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *IJCAI (1)*, pages 398–404, 1997.
7. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sa˙Boosting systematic search by weighting constraints.
8. M. Carlsson and N. Beldiceanu. Revisiting the lexicographic ordering constraint. Research Report T2002-17, Swedish Institute of Computer Science, 2002.
9. Marco Correia and Pedro Barahona. Overview of an open constraint library. In F. Fages Francisco Azevedo, Pedro Barahona and F. Rossi, editors, *Proceedings CSCLP 2006, Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming*, pages 159–168, Caparica, Portugal, 2006.
10. Marco Correia and Pedro Barahona. On the integration of singleton consistency and looh-ahead heuristics. In François Fages, Sylvain Soliman, and Francesca Rossi, editors, *Recent Advances in Constraints*, Rocquencourt, France, June 2007. Springer.
11. Marco Correia and Pedro Barahona. On the integration of singleton consistency and looh-ahead heuristics. In *Procs. of the annual ERCIM workshop on constraint solving and constraint logic programming*, Rocquencourt, France, June 2007.
12. Marco Correia, Pedro Barahona, and Francisco Azevedo. Casper: A programming environment for development and integration of constraint solvers. In Francisco Azevedo, editor, *Proceedings of the First International Workshop on Constraint Programming Beyond Finite Integer Domains (BeyondFD'05)*, 2005.
13. Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for cnf. In *DAC*, pages 530–534, 2004.
14. Romuald Debruyne and Christian Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Procs. of IJCAI'97*, pages 412–417, 1997.
15. Pieter Andreas Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Procs. of ECAI '92*, pages 31–35, New York, NY, USA, 1992. John Wiley & Sons, Inc.
16. Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *AAAI*, pages 191–197. AAAI Press, 2007.
17. R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
18. Pascal Van Henteryck. *Constraint satisfaction in logic programming.* MIT Press, 1989.
19. Eric I. Hsu, Matthew Kitching, Fahiem Bacchus, and Sheila A. McIlraith. Using expectation maximization to find likely assignments for solving csp's. In *AAAI*, pages 224–230, 2007.
20. Ludwig Krippahl and Pedro Barahona. Psico: Solving protein structures with constraint programming and optimization. *Constraints*, 7(3-4):317–331, 2002.
21. Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Recording and minimizing nogoods from restarts. *Journal on Satisfiability, Boolean Modeling and Computation(JSAT)*, 1:147–167, may 2007.

22. Ro Lopez-ortiz, Claude guy Quimper, John Tromp, and Peter Van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *In Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 245–250, 2003.
23. D. Mehta and M.R.C. van Dongen. Static value ordering heuristics for constraint satisfaction problems. In M.R.C. van Dongen, editor, *Proceedings of the Second International Workshop on Constraint Propagation And Implementation*, pages 49–62, 2005.
24. Luc Mercier and Pascal Van Hentenryck. Edge finding for cumulative scheduling. 20, 2008.
25. David R. Musser and Alexander A. Stepanov. Generic programming. In *ISSAC*, pages 13–25, 1988.
26. Patrick Prosser, Kostas Stergiou, and Toby Walsh. Singleton consistencies. In Rina Dechter, editor, *Proceeding of CP'00*, volume 1894 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2000.
27. Jean-François Puget. Dynamic lex constraints. In *CP*, pages 453–467, 2006.
28. Jean-Francois Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *ISMIS '93: Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, pages 350–361, London, UK, 1993. Springer-Verlag.
29. Jean-Francois Puget. Automatic detection of variable and value symmetries. In *CP*, pages 475–489, 2005.
30. Jean-Francois Puget. Breaking all value symmetries in surjection problems. In *CP*, pages 490–504, 2005.
31. Jean-Francois Puget. Breaking symmetries in all different problems. In *IJCAI*, pages 272–277, 2005.
32. Philippe Refalo. Impact-based search strategies for constraint programming. In Mark Wallace, editor, *Procs. of CP'07*, volume 3258 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2004.
33. Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI*, pages 362–367, 1994.
34. Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In *CP'04*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633. Springer, 2004.
35. Christian Schulte and Guido Tack. Views and iterators for generic constraint implementations. In *CSCLP*, volume 3978 of *Lecture Notes in Computer Science*, pages 118–132. Springer, 2005.
36. A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, 1994.
37. Ruben Viegas and Francisco Azevedo. GRASPER: A Framework for Graph CSPs. In Jimmy Lee and Peter Stuckey, editors, *Procs. of Sixth International Workshop on Constraint Modelling and Reformulation Procs. of the Sixth International Workshop on Constraint Modelling and Reformulation (ModRef'07)*, Providence, Rhode Island, USA, September 2007.
38. Zhang Yuanlin and Roland H. C. Yap. Arc consistency on n -ary monotonic and linear constraints. In *Principles and Practice of Constraint Programming*, pages 470–483, 2000.

A Comparison of Boosted versus Unboosted Weighted Degree Search.

Diarmuid Grimes

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

Abstract. This paper describes the two solvers $MDG\text{-}noprobe$ and $MDG\text{-}probe$ submitted to the 2008 CSP Solver competition. The two solvers differ in their approach to search, whereas one uses the weighted degree heuristic in its normal fashion, the other attempts to boost the power of the heuristic by performing a small amount of information gathering prior to search. We provide a detailed comparison of the performance of the two solvers and assess the advantages and disadvantages of each approach with regard to the results.

1 Introduction

The solvers $MDG\text{-}noprobe$ and $MDG\text{-}probe$ are based on a version of the solver mistral of Emmanuel Hebrard, which is a C++ based complete constraint solver. $MDG\text{-}noprobe$ combines complete binary branching search with the weighted-degree heuristic variant $dom/wdeg$ as introduced by Boussemart et al. [BHLS04]. $MDG\text{-}probe$ is an implementation of the random probing procedure of Grimes and Wallace [GW07], where a number of random probes are performed prior to complete search in order to gather information in the form of constraint weights, which are then used to improve the early decisions of $dom/wdeg$ on the run to completion.

The weighted degree heuristic was introduced as a means to identify contentious variables during search and move them up the variable ordering. This has the result of reducing thrashing (by identifying, through the constraint weights, the source of the thrashing and selecting it earlier upon backtracking), and improving the fail-firstness (handling contentious variables higher in the search tree should increase the likelihood of early mistake detection).

Refalo emphasised the importance of making good choices at the top of search, as these decisions generally have the largest impact on the following search effort required to solve the problem [Ref04]. However the weighted-degree heuristic has no weight information at the start of search, other than the degrees of the variables. Grimes and Wallace proposed a number of methods for dealing with this issue. One such method was random probing, which can be viewed as combining a form of iterative sampling with learning in the form of constraint weighting.

In its original form, iterative sampling [Lan92] involves repeatedly selecting a variable and value randomly until either a solution is found or a deadend is reached. In the case of a deadend, search is restarted. Note that no information is stored regarding

previous exploration so search may revisit the same tree. Crawford and Baker point out that this approach is only suited for problems with many solutions [CB94].

In the approach of Grimes and Wallace, only the variable ordering is random. Search runs to a fixed cutoff t_{init} for a fixed number of restarts R. On the final run, the cutoff is removed and search runs to completion using the $dom/wdeg$ heuristic with the information from the 'random probes' guiding the heuristic's early decisions. Weights continue to be updated on the final run.

The purpose of performing these random probes of the search space is to generate a global weight profile of the problem, i.e. a weighted degree ranking of the variables which is reflective of their constraints' participation in failures across the search space. It can be viewed as an attempt to find the best starting point for search.

Combining the weighted degree heuristic with restarting is quite a popular search approach, indeed at least three of the solvers in the previous CSP solver competition employed such an approach to good effect.Two of those (Abscon and Tramontaine) combined the heuristic directly with a geometric restarting strategy and produced very impressive results. The probing strategy studied here is somewhat different to the normal use of restarting (as a means to escape a bad subtree), where the probes are mainly intended as an information gathering phase, albeit with the capability of solving the problem prior to the final run.

2 Probing Parameters

The parameters used for probing were 100 runs with a cutoff of 30 failures per run. A small cutoff is generally best as it gives an overview of contention within the specific search tree explored without overweighting local points of contention. Clearly we would like a diverse sample from which to assess the globality of the contention associated with a variable. 100 probes should provide a sufficient sample. Defining the cutoff in terms of number of failures guarantees that learning will occur on each run and is problem independent compared to a cutoff defined in terms of nodes or time.

Binary branching also allows for increased diversification within each probe, compared to d-way branching. In fact if search backtracks to level 1, then it the subsequent search is similar to performing a new probe. An overall time-limit of 10 minutes was imposed on the probing phase, which was checked after each probe.

There are a number of modeling options available in mistral. The options which were used by both solvers in the competition involved the addition of auxiliary variables in certain situations (e.g. problems with large predicate constraints), and the inference of all-different constraints.

3 Experimental Comparison

Table 1 gives a summary of the results of each solver in the 5 different categories. It should be noted that $MDG\text{-}probe$ was disqualified from the N-EXT category because it gave a wrong answer to a problem. This was due to a small bug in the version of mistral (which was independent of the probing procedure) which depended on a number of

factors happening simultaneously to occur. I reran on this problem with various random seeds and the only seed which produced the error was the seed used in the competition. Thus I believe that the rest of the results for $MDG\text{-}probe$ in this category are correct.

Table 1. Summary of results

Category	Solver	# inst	# solved	% solved	Avg time per solved instance
B-EXT	$MDG\text{-}noprobe$	635	558	88%	76.91
	$MDG\text{-}probe$	635	561	88%	68.19
B-INT	$MDG\text{-}noprobe$	696	499	72%	73.04
	$MDG\text{-}probe$	696	515	74%	102.34
GLOBAL	$MDG\text{-}noprobe$	556	353	63%	46.85
	$MDG\text{-}probe$	556	337	61%	52.43
N-EXT	$MDG\text{-}noprobe$	704	570	81%	85.89
	$MDG\text{-}probe$	704	564	80%	-
N-INT	$MDG\text{-}noprobe$	716	530	74%	34.18
	$MDG\text{-}probe$	716	560	78%	35.98

Notes: $MDG\text{-}probe$ was disqualified from the category N-EXT because it gave a wrong answer for a problem.

Overall $MDG\text{-}probe$ solved the most problems, although there were only clear differences in the binary intensional and n-ary intensional categories. However probing proved detrimental in the global category, and to a lesser degree in the n-ary extensional category. One would expect that the cost of probing would result in poorer average times. However, the average time per solved instance for $MDG\text{-}probe$ was less than $MDG\text{-}noprobe$ for the binary extensional category, and only 1 second worse for the n-ary intensional category. The lack of a handicap due to cost of probing can be seen most clearly by the graphical comparisons available at the competition website, where there is no discernible difference between the times for both on the top 50% fastest solved problems for both approaches. Figure 1 below shows a sample graph comparing the two approaches on the binary intensional category.

3.1 In Depth Comparison

We now look at problem types where one approach was clearly better than the other, either in number of problems solved or in time to solve the problems, and provide possible explanations why one approach is suited/unsuited. This should provide indications as to when one approach should be used over the other.

In the binary extensional category there were very few problem types where one approach was clearly better than the other. $MDG\text{-}probe$ solved 5 more QCP problems. It was also the only solver to solve one of the $rand$-2-50-23 problems, and was the quickest solver on two other of these random problems.

The main reason for the better performance of $MDG\text{-}probe$ in the binary intensional category (it solved 16 more problems) was down to the scheduling problems: job

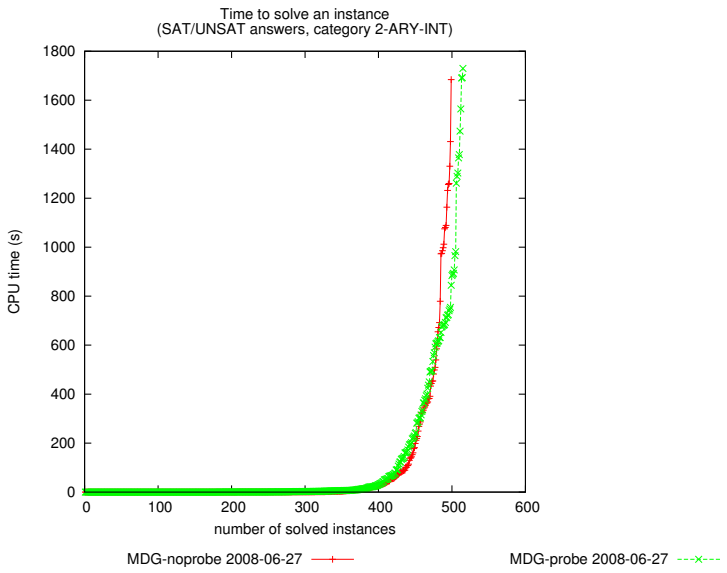


Fig. 1. Binary intensional category, time taken on solved instances..

shop, open shop and super-open shop. It solved 13 more of these types of problem, and was generally faster on the harder instances. Most of the easier instances (e.g. jobshop-e*ddr) were solved during the probing phase, while most of the harder instances were solved on the run to completion. The problems were often solved in less than 1 second on the run to completion (and virtually backtrack free).

These problems generally have loose constraints and large domains. Thus a poor choice at the top of the search tree can result in the exploration of an extremely large insoluble subtree, from which search may never recover in a feasible amount of time. This is known as the early mistake problem [CB94]. The weights learnt during probing clearly identify certain variables as bottleneck variables, and by selecting these variables first on the final run, the problems can be solved quite quickly.

Figure 2 shows the weight profile generated for the problem $os\text{-}taillard$-20-100-0 using the same random seed as was used in the competition. The variables are ranked in order of largest weighted degree increase (i.e. weighted degrees after probing minus their original degrees). As one can see, most of the weight is collected by a tenth of the variables, with roughly half the variables receiving no weight increase at all. The slope of the line indicates the discrimination between successively ranked variables, the greater the slope the clearer the difference between the weighted degrees of the variables. This shows that some variables are clearly more contentious than others.

$MDG\text{-}probe$ also performed well on the radio link frequency allocation problems (rlfaps), solving 1 more and was much quicker on the harder instances. These problems are all unsatisfiable and contain an insoluble core which is identified by the probing approach. Although $MDG\text{-}noprobe$ was quicker in general on the frequency assignment

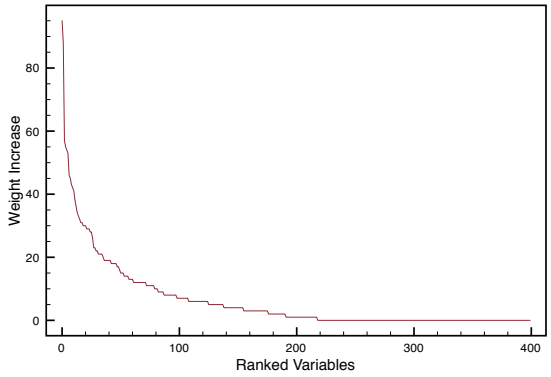


Fig. 2. Weight profile after probing, sample os-taillard-20 problem.

problems with polarization constraints (fapps), probing actually solved 3 more problems. On closer inspection the cost of probing was extremely high for these problems, it hit the 10 minute cutoff on a number of problems. However many of the problems were solved extremely quickly on the run to completion, in some cases they were solved virtually backtrack free.

Interestingly, one problem type that probing was expected to perform well on turned out to the contrary. On the $queens$-$knights$ problems, weights learnt by probing actually proved detrimental to the subsequent search compared to MDG-$noprobe$. The reason for this is that mistral generated auxiliary variables to handle the queens diagonal constraint, which consisted of a large number of predicates. Probing branched on these variables and subsequently failed with the result that there was more weight on queens variables than on the knights after probing.

In the global category MDG-$noprobe$ was vastly superior to probing on the balanced incomplete block design problems (BIBDs), in particular on the BIBDVariousK problems. Overall it solved 17 more of these problems than probing, 15 of which were BIBDVariousK problems (often solving problems, that probing couldn't solve, in less than 1 second). These problems contain a large amount of symmetry, it is only when the first variable is assigned that the symmetry begins to break and so local weight information is much more important than global weight information. Probing in this case would result in jumping around in disparate parts of the search space diminishing the buildup of contention. MDG-$noprobe$ was also better on the magic square problems, solving 3 more problems.

Problems which probing outperformed no probing in this category were the timetabling from the 08 competition (probing solved 3 more and in most cases the vast majority of the time was spent in probing), and the costas array problems where probing solved 1 more and was generally much quicker on the larger instances.

In the n-ary extensional category MDG-$noprobe$ was consistently quicker on the large BDD problems, as the probing phase took 100s and then solved the problem in

roughly equivalent time on the run to completion. $MDG\text{-}noprobe$ was also consistently quicker on the Dimacs dubois boolean problems. In this case weights learnt during probing were detrimental to the subsequent search. Probing was also ineffective on the blank grid crossword problems with large dictionaries ($OgdVg$ and $UkVg$), where probing solved 6 problems less and was generally slower (although this may in part be because probing took nearly 300 seconds and 100 seconds respectively for these problem sets).

Finally the category n-ary intensional, where the largest difference in the number of problems solved by one approach over the other occurred (probing solved 30 more problems than no probing). In general this was down to solving problems in the random probing phase, as opposed to solving problems on the run to completion with the weights learnt. For example, $MDG\text{-}probe$ solved 9 more of the pseudo boolean problems ii (which encode inductive inference problems), all of which were solved during the probing phase. Similarly it solved 14/15 of the all-interval series problems, again all of which were solved during probing. This was 4 more problems than $MDG\text{-}noprobe$ solved.

4 Conclusions

In this paper we have described the two solvers $MDG\text{-}noprobe$ and $MDG\text{-}probe$ which were submitted to this years CSP solver competition. We have provided an analysis of the experimental results of each approach and provided insight as to why one approach was better than the other on some problem types. Surprisingly the cost of probing was generally much less of a handicap than expected.

References

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

[CB94] J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI*, pages 1092–1097, 1994.

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

[Lan92] P. Langley. Systematic and nonsystematic search strategies. In *Proceedings of the first international conference on Artificial intelligence planning systems*, pages 145–152, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[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 an open source constraint library written in C++. In this short paper we describe with some level of detail two constraint solvers based on *Mistral* that entered the 2008 CSP Solver Competition.

1 Introduction

In *Mistral*, as in most constraint programming libraries, three main types of objects are defined:

1. A set of *data structures* to represent and manipulate variables and other backtrackable objects such as primitive types and containers.
2. A set of *constraint propagators* as well as methods to handle the communication between these propagators and the variables. For instance, around 30 constraints, and for each at least one propagators are predefined in *Mistral*. They include the usual arithmetical $(+, -, *, /, \%)$ relational $(\leq, <, \geq, >, =, \neq)$ and logical $(\wedge, \vee, \neg)$ constraints and predicates, as well as some commonly used global constraints, such as `AllDifferent` and `Global Cardinality Constraint` (Bounds consistency, algorithm and code from Quimper et al. [10, 13] for both), `Element` [5], `WeightedSum`, `LexOrder` [6], `Occurrence`, `Slide` [1], `Tree` [12] or `Distance` [3].
3. A set of *search heuristics and algorithms* such as variable ordering (`dom/wdeg` [2], `impact` [14], `impact/wdeg`,...), restart policies (`Geometric` [15], `Luby` [11]), tree search algorithms (depth first search and limited discrepancy search) and branching strategies.

During the 2008 CSP Solver Competition, two solvers, written on top of *Mistral* were submitted. These solvers use Roussel's XML parser to read the input file, and make a number of decisions in order to model the problem instance according to the XML specification and using the best suited *Mistral*'s objects and classes. This short paper is organized as follows. For each one of the three aspects above, we describe the objects and methods that were used during the competition and show how the modelling choices were made.

2 Data Structures

2.1 Primitive types and Containers

Reversible Primitive Types: In *Mistral*, all primitive numerical types have their reversible, or backtrackable, version. The code is extremely simple, and compact due to the use of genericity. All reversible objects inherit from the class

`ReversibleObj`, and they all access to a global stack of pointers to reversible objects. When a reversible object saves its current state, it also puts a pointer to itself on this stack. Then, when withdrawing a decision, all reversible objects saved after this decision was taken are notified, triggering their restoration. The class `ReversibleNum<T>`, where `T` is type in {`char`,`int`,`double`,`float`,..} simply keeps a vector of values for each decision along a branch, as well as a vector of integers to index decisions that triggered a saving operation. The space complexity is thus $O(n)$ where n is the maximal length of a branch in the search tree. Moreover, two useful container types, bitsets and lists also have their reversible counterparts in *Mistral*.

Bitset: To represent a set S, the bitset itself requires $max(S) - min(S) + 1$ bits. Then for each 32-bits word, and for each bit set to one in this word, a "trailed" 32-bits word is allocated. Whenever a 32-bits word changes for the first time for a given level in the search tree, it is first copied in the next available trailed word, and this new one is used instead in that subtree. The space complexity for a set S implemented in this way is therefore: $O(max(S) - min(S))$. That is, $max(S) - min(S) + 1$ bits for the original bitset, and $32 * |S|$ bits for the trailed memory.

List: Finally, reversible lists of integer are implemented using simply two integer arrays, one to keep the current list of values, and one to keep the index of these values in the list, and a reversible integer standing for the size of the list. Insertions and deletions can be done in constant time by swapping with the last element and changing the size. Upon backtrack the list is restored to its previous state in constant time as well, by restoring the size attribute to its previous value. Notice that in this data structure, the list does not stay in any particular order. The space complexity for a set S implemented in this way is therefore: $O(min(n, (max(S) - min(S))))$ Where n is the maximum depth of the search tree. That is, at most $64 * min(n, (max(S) - min(S)))$ bits to keep the size of the list and its trail, $32 * (max(S) - min(S) + 1)$ bits for the indexing array, and $32 * |S|$ bits for the list array.

2.2 Variables

Five different types of variables are implemented in *Mistral*, all subclasses of `VariableInt`. All are finite domain integer variables and specialisations thereof.

Specialised Finite Domain Variables: Three such types are used for usual specialisations of integer variables. The simplest one (`VariableConstant`) encodes constants, that is constrained objects with a single immutable value. When possible, constant values are absorbed into the constraint definition before the search objects are actually created. For instance if one post an `AllDifferent` constraint where one of the variable is a constant k, then k will be removed from the domain of all other constrained variables and this variable will be ignored. However, it

is not always possible nor easy to do this, and the `Constant` type acts a safety net for such cases.

Next, there is specialised type for variables with Boolean domains (`VariableBool`). The domain is stored on an `Int`, the value 0 stands for an empty domain, 1 stands for $\{0\}$, 2 stands for $\{1\}$ and 3 stands for $\{0, 1\}$. Notice that the binary encoding corresponds to the bitset representation of the domain (respectively 00, 01, 10 and 11). This makes the operations involving Boolean variables and general finite domain variables represented with bitsets easier and faster. When the domain of a Boolean variable need to be restored, we know it can only be restored to $\{0, 1\}$, hence upon backtrack, the Boolean variable whose domains have changed are simply assigned the value 3 (11 in binary).

Finally, the classical `Range` variables (`VariableRange`) are available for bound reasoning. They are implemented using two reversible integers, for lower and upper bounds. During the modeling phase, any variable constrained by a relation that can remove non-extremal values are flagged so that this representation is not available for them.

Integer Variables as bit-vectors (`VariableBit`): In this implementation, a domain $D(x)$ is represented as a reversible bitset such as the one described above. 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.

Integer Variables as lists (`VariableList`): In this implementation, a domain $D(x)$ is represented as a reversible list (see description above), coupled with a non-reversible bitset. The value of the bitset is synchronized to the reversible list on domain modification and on backtrack. This bitset is used mainly for the AC3bitset algorithm.

2.3 Competition Setting

During the competition, a variable x is represented using the "best suited" data structure using the relatively simple following rules:

1. If $|D(x)| = 1$: use `VariableConstant`
2. Else if $D(x) = \{0, 1\}$: use `VariableBool`
3. Else if $|D(x)| > 64$, and the constraints on x are all convex: use `VariableRange`
4. Else if the constraints on x only use the methods `remove()` and `setDomain()` during propagation: use `VariableList`
5. Else: use `VariableBit`

3 Constraint Propagation

3.1 Variables and Constraints Queue

The closure algorithm is a simple AC3-like *first in, first out* queue. With two slight modifications. The first modification is that, like in Ilog Solver for instance, three types of domain modification event are distinguished.

1. A *Value event* is thrown when variable is assigned a value.
2. A *Range event* is thrown when at least one variable's bound changes.
3. A *Domain event* is thrown whenever a domain changes.

Notice that these events go from more specific to more general. In other word, a value event is necessarily a range event, which is necessarily a domain event. Therefore, for each variable, the list of constraints has a very specific order. First come the constraints that should be triggered on any event, then those that should be triggered on range or value events, and last those that should be triggered only on value events. Three lists (Domain, Range and Value) are thus defined, where the last elements of a list are those of the next. When an event is caught, we traverse the corresponding list until the end. It means that a constraint that should be triggered only on value events, such as $\neq$, is never called nor induces any computation or test unless a variable is assigned. The second modification is that particularly expensive propagators are delayed. That is, they are put on a second queue (of constraints rather than variables). A constraint on this queue is propagated only when the variable queue is empty (that is, a fixed point has been reached on the non-delayed constraints).

3.2 Extensional Constraints

Three algorithm for extensional constraints are implemented in *Mistral*.

AC3bitset: The first algorithm, restricted to binary relations is an implementation of the AC3 algorithm using the "&" operation on bitsets to check the existence of a support as described in [8].

GAC3rValid: The second algorithm can handle non-binary constraints and uses the notion of residual supports ([9, 7]). The relations are stored as bitset standing for a flattened matrix, or can be given as an intentional `check()` method. When looking for a support, we loop through the Cartesian product of the domain.

GAC2001Allowed: The third algorithm keeps, for each pair (variable, value), a list of allowed tuples as well as a pointer to the last element of the list that was used as support earlier during search. The pointer is implemented simply with a reversible integer. When looking for a support, we loop through the list of supports, starting from the pointed one, until we reach a valid one (that is, a tuple whose elements are currently in their respective domains), or we fail. This algorithm is optimal, each list of supports is traversed at most once along a branch of the search tree, hence the complexity is in $O(rd^r)$ where r stands for the arity of the constraint and d for the domain size. Another optimisation is to order the domain membership checks by increasing domain size of the variables, so that the likelihood to abort a validity check early is greater.

3.3 Intentional Predicates

Two different representations of predicates were used.

Decomposition: 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:

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

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 $Y_1 = X_3$.

Generalised Arc Consistency: 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 `GAC3rValid`, 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.

3.4 Global Constraints

Four global constraints were used during the competition: `AllDifferent`, `Cumulative`, `Element` and `WeightedSum`. Since the constraint `Cumulative` is not implemented in *Mistral*, a decomposition [4] using Boolean variables and `WeightedSum` constraint was used instead.

3.5 Competition Setting

Given an extensional constraint with arity r and tightness t, we use the following rules to select what algorithm should be used:

1. If $r = 2$ and $t < .999$: `AC3bitset` is used.
2. Else if $t < .98$: `GAC3rValid` is used.
3. Else: `GAC2001Allowed` is used.

Given an intentional constraint specified as a tree of predicates, we need to decide whether we are going to use a generic GAC algorithm (`GAC3rValid`), or decompose it into small arity predicates using additional variables. We compute a complex and heuristic value based on the following criteria:

1. Constraint arity (higher arity is more favourable to a decomposition)
2. Individual predicate types (some predicates have better propagators than others)
3. Ratio nodes/leaves in the predicate tree (Bushier trees are more favourable to GAC)
4. Domain continuity (Holes in the domain are more favourable to GAC)
5. Size of the Cartesian product of the domains (The decomposition can better adjust the complexity)
6. Boolean domains (When domains are all Boolean, BC is equivalent to AC, hence is favours the decomposition)
7. Total number of constraints in the problem (The GAC approach is much more space efficient)

For each one of the aspects some values are reported and weighted in a completely heuristic and empirical way, the final number tells which representation should be used.

4 Search Heuristics

4.1 Variable Ordering

Mistral-prime used a slightly modified version of *domain over weighted degree* [2] which we call *domain over weighted-by-level degree*. 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 $maxlevel - level + 1$ instead of 1, where *maxlevel* is the deepest level in the search tree explored so far. 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.

Mistral-option used *domain over weighted degree* for extensional constraints. On intentional benchmarks, a variation of the *Impact* heuristic [14] was used instead. Notice that on most implementations of the Impact heuristics, complements-to-one of the impact are stored instead of the direct impact value. By doing so it is possible to select the variable with minimum sum of these complement-to-one, hence simulating Minimum domain when all impacts are equal. Therefore it is natural to use impact along with degree or weighted degree information. We used *impact over weighted degree* which as its name suggests choose the variable with minimum sum of impact complement over the sum of its neighboring constraints weighted degree:

$$\text{choose } x \text{ such that } \frac{\sum_{j \in D(x)} 1 - I(x = j)}{\sum_{x \in V} w(C(V))} \text{ is minimum}$$

In this formula, $I(x = j)$ is the impact of the decision $x = j$ as defined in [14]. Surprisingly this approach seems far inferior *domain over weighted degree* on most benchmarks of the competition.

4.2 Competition Setting

In *Mistral*-prime, variables are ordered by increasing *domain over weighted-by-level degree* and values are explored in lexicographical order. In *Mistral*-option, on the other hand, when the domain was large enough and the constraints intentional, domain-splitting was used instead. That is, given a variable x we branched on the left side with the constraint $x \leq (\frac{max(x)-min(x)}{2})$ and on the right side with the constraint $x > (\frac{max(x)-min(x)}{2})$. Moreover, *Mistral*-option chooses its Variable ordering using the following rules:

1. If the total number of values is greater than 16000 or if all constraints are extensional, then *domain over weighted degree* [2] is used.
2. Else: *impact over weighted degree* is used.

Both versions of *Mistral* used a geometric restart policy for problems involving intentional constraints. The initial cutoff on the number of backtracks was set to $min(n, 1000)$ where n is the number of variables in the problem definition. Then it is multiplied by $1 + \frac{1}{3}$ upon every restart.

5 Conclusion

We described the two *Mistral*-based solvers that entered the 2008 CSP Solver Competition. At the time we write this paper, only *Mistral*'s, *Choco*'s, *Abscon*'s and *cpHydra*'s results are known to the authors, we give a short summary (using the best overall version of the above solvers) in Table 1.

Table 1. Results summary: Percentage of instances solved among those solved by at least one solver, and average CPU-time on solved instances for each category.

Category (#instances)	*cpHydra*		*Abscon*		*Choco*		*Mistral*	
	Solved	CPU-time	Solved	CPU-time	Solved	CPU-time	Solved	CPU-time
Binary Extensional (622)	**92%**	**62.44 s**	88%	93.26 s	89%	95.78 s	89%	70.21 s
Binary Intentional (634)	**94%**	71.37 s	81%	**43.40 s**	82%	55.89 s	82%	58.23 s
Global (501)	**84%**	80.83 s	37%	170.62 s	69%	69.69 s	80%	**56.59 s**
N-ary Extensional (607)	**97%**	**78.20 s**	90%	80.09 s	73%	189.10 s	94%	83.67 s
N-ary Intentional (660)	**86%**	54.70 s	74%	53.21 s	78%	49.70 s	80%	**32.02 s**

References

1. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. Slide: A Useful Special Case of the Cardpath Constraint. In Malik Ghallab, editor, *Proceedings of*

the 18th European Conference on Artificial Intelligence (ECAI-08), pages 475–479, Patras, Greece, July 2008. IOS Press.

2. F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. 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.
3. E. Hebrard, B. O'Sullivan, and T. Walsh. The Distance Constraints in Constraint Satisfaction. In Manuala Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 106–111, Hyderabad, India, January 2007. Professional Book Center.
4. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.
5. P. Van Hentenryck and J.-P. Carillon. Generality versus specificity: An experience with ai and or techniques. In Tom Mitchell and Reid Smith, editors, *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI-88)*, pages 660–664, St Paul, MN, USA, 1988. AAAI Press / The MIT Press.
6. Z. Kiziltan. *Symmetry Breaking Ordering Constraints.* PhD thesis, Uppsala University, 2004.
7. C. Lecoutre and F. Hemery. A Study of Residual Supports in Arc Consistency. In Manuala Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 125–130, Hyderabad, India, January 2007. Professional Book Center.
8. C. Lecoutre and J. Vion. Enforcing Arc Consistency using Bitwise Operations. *Information Processing Letters*, 2:21–35, 2008.
9. 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.
10. 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.
11. M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.
12. P. Prosser and C. Unsworth. Rooted Tree and Spanning Tree Constraints. In *Workshop on Modelling and Solving Problems with Constraints, held at the 17th European Conference on Artificial Intelligence (ECAI 06)*, Riva del Garda, Italy, 2006.
13. 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.
14. P. Refalo. Impact-Based Search Strategies for Constraint Programming. 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 557–571, Toronto, Canada, October 2004. Springer-Verlag.

15. T. Walsh. Search in a Small World. In Thomas Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, Stockholm, Sweden, January 1999. Morgan Kaufmann.

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,

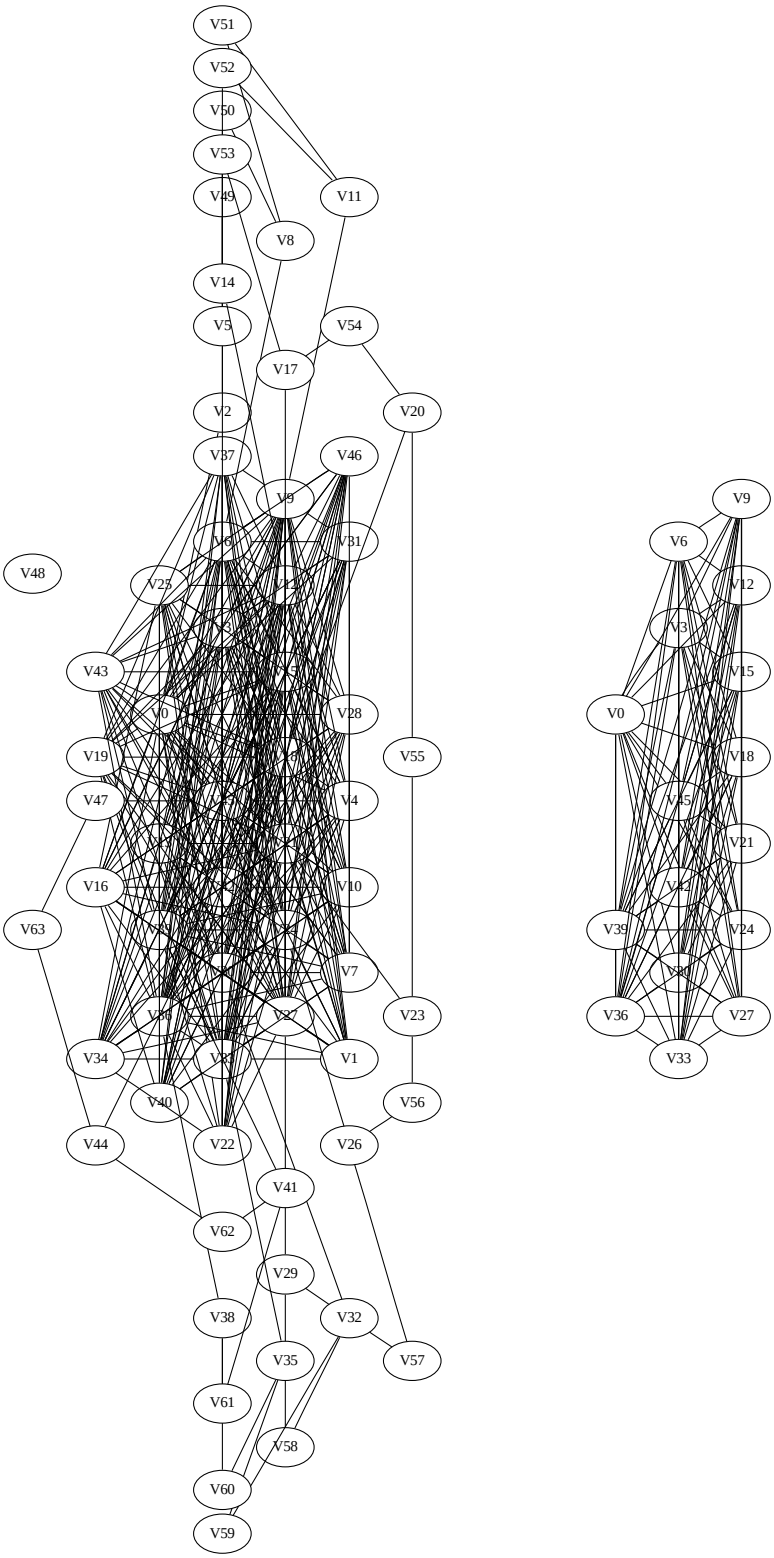


Fig. 1. The constraint graph of the instance *blackHole*-4-4-*e*-0 contains a 16-clique. A constraint $allDifferent$ generated from this clique can be shown to be disentailed.

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.

SPIDER: A Basic CSP Solver

Chavalit Likitvivatanavong

School of Computing, National University of Singapore
Republic of Singapore
chavalit@comp.nus.edu.sg

Abstract. This document provides a short description of a MAC-based solver written by the author for the Third International CSP Solver Competition (2008).

1 Features

SPIDER ("SPider Is a Dull solvER") is a complete CSP solver written in C for solving problems with binary and non-binary extensional constraints, using standard binary-branching MAC [2–4]. The solver is based on the CSP solver code from CPlan [1]. It uses the revision-queue structure for propagating the effect of AC [5]. The solver that ran in the competition has the following features.

- *variable ordering.* The heuristic dom/wdeg [6] was used.
- *binary branching.* After the second branch, the effect of the refutation is propagated. That is, when value a of variable X is tried and the search failed, a is removed from the domain of X and arc consistency is re-enforced.

  Interestingly, a binary-branching MAC with no propagation after the refutation branch can be just as good as the binary-branching MAC with propagation. It was found that for some benchmark instances, one version can be several times better than the other while the performance reverses on some other benchmarks.

  It was shown in [2, 3] that binary-branching is strictly more powerful than d-way branching (d is the domain size). However, propagation is not essential to the proof: with or without propagation, binary-branching is still more powerful than d-way branching. More studies need to be done to see under what conditions is the propagation more effective.
- *value ordering.* Values are statically ordered. Each value has an associated integer weight. The value with the largest weight will be picked first for instantiation. Before the search starts, the solver goes through each tuple in each extensional constraint and increases a value's weight by 1 if that value appears in the tuple and the constraint is of type "support", or decreases the weight by 1 if the constraint is of type "conflict".

- *residue*. For each constraint C, variable X in the scope of C, and value a in the domain of X, two caches were used to record previous supports for a in C. One is for direct *residue* [7]. The other is for multi-directional residue [8]
- *restart*. Search is restarted if the number of nodes encountered during backtracking search exceeds some limit. The limit is initially set to $2n$ where n is the number of variables, and increased by two-fold every time the limit is reached. Weights for dom/wdeg are retained from previous runs.

  Rather than using the number of nodes visited during search as the limit, counting the number of backtracks is another possible choice. However, it seems to make little difference.
- *propagation-queue ordering*. Queue of variables is used to record which variables should be revised next. Variables are picked using the same dom/wdeg heuristic as that for variable instantiation.

  It is worth noting that because dom/wdeg is used for variable ordering heuristic, the heuristic for queue revision will have an effect on the weights for wdeg. For dom/wdeg, the main reason we order the queue is not to make the wipeout occur as soon as possible (if the network has become arc inconsistent), but rather to make sure the constraint found to cause the wipeout is the one that matters most when its weight is increased. In other words, for the usual dom/future-degree variable ordering heuristic, queue ordering would affect the number of constraint checks but not the number of nodes visited, while for dom/wdeg (or simply wdeg) it would affect both.

  Using dom/wdeg to pick variables for both propagation and instantiation could accelerate the increase in weight for a constraint with already high weight as well as cutting down on the propagation. But in general it is not clear whether a good heuristic for queue revision will always perform well for variable selection and vice versa, but for dom/wdeg there may be a possible synergy between the two.
- *constraint subdivision*. For each extensional constraint C, tables $C(X, a)$ for each X in the scope of C and a in the domain of X are created. The contents of the table are index to location of the corresponding tuples in C. This can be considered a very crude form of tries [9]. For example, if *scope*(C)= (X,Y,Z) and $C = \{(3,6,7), (0,1,4), (3,0,4)\}$. Then $C(X, 0) = \{2\}$, $C(X, 3) = \{1,3\}$, $C(Y, 1) = \{2\}$, $C(Y, 0) = \{3\}$, $C(Y, 6) = \{1\}$, $C(Z, 4) = \{2,3\}$, $C(Z, 7) = \{1\}$.
- *preprocessing*. No preprocessing was performed.

1.1 Specific features for solving non-binary instances

- *GAC*. The GAC algorithm by Lecoutre and Szymanek was implemented [10].
- *trigger for GAC*. For normal GAC, once a variable is instantiated any constraint involved must be revised. For this solver however, GAC is called only

when the number of instantiated variables in a constraint is greater than a fixed number k. For the competition, k is set to 5.

Strictly speaking, SPIDER does not perform full MGAC since there is no GAC preprocessing and GAC during search is carried out selectively.

2 Current Status and Future Work

It was found out in the preliminary round that SPIDER reported an incorrect answer in one non-binary instances. This was traced back to some bug in the implementation of GAC algorithm. The solver still remains in the competition for the binary-instance category.

As of this writing, GAC based on multiple-valued decision diagrams is one the fastest algorithms [11]. Future work on SPIDER will focus on incorporating this algorithm.

References

1. van Beek, P., Chen, X.: Cplan: A constraint programming approach to planning. In: Proceedings of AAAI-99, Orlando, Florida (1999)
2. Mitchell, D.G.: Resolution and constraint satisfaction. In: Proceedings of CP-03, Kinsale, Ireland (2003) 555–569
3. Hwang, J., Mitchell, D.G.: 2-way vs. d-way branching for csp. In: Proceedings of CP-05, Sitges, Spain (2005) 343–357
4. Sabin, D., Freuder, E.C.: Contradicting conventional wisdom in constraint satisfaction. In: Proceedings of ECAI-94, Amsterdam, The Netherlands (1994) 125–129
5. Mackworth, A.K.: Consistency in networks of relations. Artificial Intelligence **8** (1977) 99–118
6. Boussemart Frederic, Hemery Fred, L.C., Lakhdar, S.: Boosting systematic search by weighting constraints. In: Proceedings of ECAI-04, Valencia, Spain (2004) 146–150
7. Lecoutre, C., Likitvivatanavong, C., Shannon, S., Yap, R.H.C., Zhang, Y.: Maintaining arc consistency with multiple residues. Constraint Programming Letters **2** (2007) 3–19
8. Lecoutre, C., Boussemart, F., Hemery, F.: Exploiting multidirectionality in coarse-grained arc consistency algorithms. In: Proceedings of CP-03, Kinsale, Ireland (2003) 480–494
9. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: Proceedings of AAAI-07, Vancouver, Canada (2007) 191–197
10. Lecoutre, C., Szymanek, R.: Generalized arc consistency for positive table constraints. In: Proceedings of CP-06, Nantes, France (2006) 284–298
11. Cheng, K.C.K., Yap, R.H.C.: Maintaining generalized arc consistency on ad hoc r-ary constraints. In: Proceedings of CP-08, Sydney, Australia (2008)

Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving*

Eoin O'Mahony, Emmanuel Hebrard,
Alan Holland, Conor Nugent, and Barry O'Sullivan

Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{e.omahony|e.hebrard|a.holland|c.nugent|b.osullivan}@4c.ucc.ie

Abstract. It has been shown in areas such as satisfiability testing and integer linear programming that a carefully chosen combination of solvers can outperform the best individual solver for a given set of problems. This selection process is usually performed using a machine learning technique based on feature data extracted from constraint satisfaction problems. In this paper we present CPHYDRA, an algorithm portfolio for constraint satisfaction that uses case-based reasoning to determine how to solve an unseen problem instance by exploiting a case base of problem solving experience. We used data from the 2006 CSP Solver Competition to set up the case base and to assess the superiority of our portfolio approach over each of its constituent solvers.

1 Introduction

It is recognised within the field of constraint programming that different solvers are better at solving different problem instances, even within the same problem class [3]. It has been shown in other areas, such as satisfiability testing [13] and integer linear programming [5], that the best on-average solver can be out-performed by carefully exploiting a portfolio of possibly poorer on-average solvers. Selecting from a portfolio usually relies on a machine learning technique based on feature data extracted from constraint satisfaction problems.

Several related pieces of work have been reported in the literature. The SATZILLA[1] system builds runtime prediction models using linear regression techniques based on structural features computed from instances of the Boolean satisfiability problem. Given an unseen instance of the satisfiability problem, SATZILLA selects the solver from its portfolio that it predicts to have the fastest running time on the instance. In the International SAT Competition 2007, SATZILLA won two of the categories, and came second and third in two others. The AQME system is a portfolio approach to solving quantified Boolean formulae, i.e. SAT instances with some universally quantified variables [9]. AQME is built on the Weka data-mining library[2]. Three versions of AQME have

* This work was supported by Science Foundation Ireland (Grant No. 05/IN/I886).

[1] http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/

[2] http://www.cs.waikato.ac.nz/ml/weka/

competed in the International Competitive Quantified Boolean Formula Evaluation[3]: a version using decision trees to select which solver to use, a version using logistic regression, and another using 1-nearest neighbour. Like SATZILLA, AQME selects one solver to run for a given unseen formula. Our approach contrasts with both of these in that we select a set of solvers to run on the given instance rather than a single solver. Streeter et al. [12] build upon the work of Sayag et al. [11], by using optimisation techniques to produce a schedule of solvers that should be tried in a specific order, for specific amounts of time, in order to maximise the probability of solving the given instance. This work is similar to ours, except we use a much more "knowledge-light" approach by relying on case-based reasoning (CBR) to advise on the composition of our schedule of solvers. A related work to our own is by Gebruers et al. [2], who use case-based reasoning to select solution strategies for constraint satisfaction. Our approach is quite different since we do not tune a particular solver, but make a more high-level decision about which solvers to run.

The motivation for this research is two-fold. Firstly, constraint programming systems are often quite difficult for non-expert users to apply in practice. We address this concern by developing a system that uses artificial intelligence techniques to automatically select an appropriate solver for a given unseen problem instance. Secondly, we aim to show that given the current state of CSP Solver development, one could win the International CSP Solver Competition by not implementing any new solvers, but by using machine learning to select between a small set of common solvers that have already been developed. Our approach uses case-based reasoning to inform the selection process. We build a case base of problem solving experience by solving a variety of typical problem instances with each solver in our algorithm portfolio. Then we employ case retrieval methods in a number of increasingly sophisticated ways, giving better performance in each case.

2 Portfolios of Solvers

It often occurs that there is a low correlation between the performance of different search techniques. This may, fortuitously, offer the prospect of combining various search mechanisms so that we form a portfolio of algorithms amongst which computing resources are shared so that the combined effectiveness of multiple solvers outperforms the single most effective solver. There is a balance between risk (the variability in search performance) and reward (the expected search performance) that must be achieved. In previous work, Huberman *et al.* [4] developed a theory of algorithm portfolio design that employed an economics-based approach in an effort to balance risk and reward. Theirs was a general method for combining existing programs in a *static* portfolio so that the combinations were unequivocally superior to any of the individual algorithms. They employed Modern Portfolio Theory, as described by Markowitz [8], to model the *efficient frontier*. An efficient portfolio is one that has the highest possible reward for a given level of risk, or the lowest risk for a given reward.

[3] `http://www.qbflib.org/index_eval.php`

2.1 Case-Based Reasoning

Case-Based Reasoning is a machine learning methodology that adopts a lazy learning approach and contains no explicit model of the problem domain. Instead a set of past examples called *cases* are retained. Each case is made up of a description of a past example or experience and its respective solution. The full set of past experiences encapsulated in individual cases is called the case base.

In CBR problems are solved "by using or adapting solutions to old problems" [10]. When a new problem is presented, the case base is searched, similar past examples are found and these are used to solve the presented problem. The need to detect and model general patterns over the entire problem space is avoided. This approach to problem solving in CBR has a number of advantages. In particular, CBR has proven to be successful in solving *weak-theory* problems, in which little insight into the problem exists and the problem domain may be complex. This characteristic makes CBR a good candidate for the tasks of solver selection and portfolio generation in CSPs. While many different solving techniques exist, it is a difficult task even for domain experts to predict which techniques, or combination thereof, will be effective on a given problem instance.

Given its apparent suitability, it is not surprising that CBR has previously been considered for the task of strategy selection in Constraint Programming. CBR has been outlined as part of a framework for capturing expert knowledge in terms of CP problem modelling [6]. CBR has also successfully outperformed other CP strategy selection techniques when tested on two different CSPs [2]. Within CPHYDRA, the CBR methodology is used to inform a portfolio approach to problem solving.

2.2 CPHYDRA

The most fundamental element of any CBR system is the case base. In CPHYDRA, cases contain the feature values describing a particular CSP problem instance. Each problem instance in the CSP Solver Competition is described in XML and we can use this to help generate the feature values for a particular problem instance. We do this in two distinctly different ways. Firstly, we extract a set of *syntactic* features such as maximum constraint arity, average and maximum domain size, number of variables and of constraints, domain continuity, ratio of different types of constraints (extensional, intentional or global) and ratio of subclasses of constraints within these main categories. Then we complement these static features with *solver specific* features by running a constraint solver, `Mistral`, for a limited amount of time (typically two seconds) and recording its modeling choices and search statistics, such as number of variables whose domains are represented as a range/boolean/bitset, number of extra variables created to represent constraint reification, number of nodes explored, number of constraint propagation calls and average constraint weighting.

We used 36 features in total to describe the problem instances. Of these features, 14 were generated using `Mistral`. It is also important to note that quantified features, such as maximum domain size, were log-scaled whereas the ratios were all percentages. As well as the set of feature-values, each case contains a list of how long each solver being used by CPHYDRA took to solve that problem instance. This formed the 'experience' element of each case.

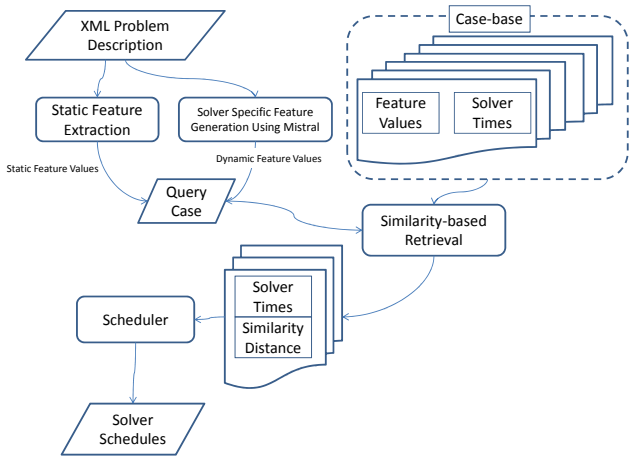


Fig. 1. Overview of CPHYDRA.

The CBR methodology can be broken down into four distinct phases: Retrieval, Reuse, Revision and Retention. These phases are often referred to as the 'Four REs'-cycle [1]. The first two are of particular importance to CPHYDRA, as can be seen in Figure 1. However, the final two phases are relevant too. We will now discuss each of these four phases relates to CPHYDRA in turn.

Retrieval. To begin with, a query case is first produced. The XML presentation of the problem instance is used to generate both the static and the dynamic feature-values as described previously. Using the query case the case base is searched and the most similar cases to the present problem description are retrieved. A simple k-nearest neighbour (K-NN) algorithm is used for this task; we set $k = 10$. In situations where similarity ties occur all cases with similarity equal to the k^{th}-ranked case are also returned. Since all features are real valued the Euclidean similarity measure is used.

Reuse. The objective of CPHYDRA is not simply to supply a prediction but a schedule describing how long each solver should run. This makes the Reuse phase of CPHYDRA more complex than many other CBR systems where the objective is a classification. As can be seen in Figure 1 the retrieval process returns the set of solver times for each of the k most similar problem instances found in the case base along with their similarities to the Query Case. This information is then used to generate a solver schedule. This process will be explained in detail in Section 2.3.

Revision. During this phase the proposed solution is evaluated and validated. This process involves running each of the solvers for the proportion of time allocated by the scheduler and determining if the problem is successfully solved by one of the solvers within its allocated time slot. If at least one solver solves the problem instance within

its time slot then the schedule is deemed a success. In competition conditions there is no opportunity to revise a solution in light of its performance or to update the case base. However, in non-competition conditions this would be desirable.

Retention. Normally once a satisfactory solution has been determined the problem description and solution are added to the case base. However, in CPHYDRA the 'solution' or experience attached to each case, the solver times, are only indirectly used to produce a solution. In order to create a complete case that could be retained each solver would have to run until it solved the problem instance or timed-out. This phase is not possible in competition conditions but could form part of a continuous online learning system.

As we have already stated, the most involved action in this cycle is the generation of a schedule given the solver performances on similar past examples. We will now discuss this process in greater detail.

2.3 Solver Scheduling

Typically, in previous CSP Solver Competitions runtime distributions for each solver display a very fast rate of decay. That is, the number of problems solved for a given amount of CPU-time decreases rapidly. Moreover, when analysing the competition results, it turns out that no solver is completely dominated. For example, sixteen of the twenty-two solvers are the fastest on at least one instance, and nine of them are the fastest on at least one hundred instances. The conjunction of these two conditions entails that partitioning CPU-time between solvers is inherently a good strategy. We show, however, that one can improve a naive partitioning by taking into account the information given by the case base reasoner. We define a method for computing good CPU-time partitions, i.e., *solver schedules*.

Our goal is to compute a solver schedule, that is a function $f : \mathcal{S} \mapsto \mathbb{R}$ mapping an amount of CPU-time to each element of a set $\mathcal{S}$ of solvers. Given a query instance, the case base reasoner returns a set $\mathcal{C}$ of similar cases. Informally, we compute the schedule so that the number of cases in $\mathcal{C}$ that would be solved using this schedule is maximised. More formally, consider a set $\mathcal{C}$ of similar cases. For a given solver $s \in \mathcal{S}$ and a time point $t \in [0..1800]$ we define $C(s,t)$ as the subset of $\mathcal{C}$ solved by s if given at least time t. The schedule f can be computed using the following constraint program:

$$\text{maximise } |\bigcup_{s\in\mathcal{S}} C(s, f(s))| \tag{1}$$

$$\text{subject to } \sum_{s\in\mathcal{S}} f(s) \leq 1800 \tag{2}$$

Notice that this problem is NP-hard as a generalisation of the knapsack problem. However, because the number of solvers is small, solving this problem to optimality is easy in practice. We refined the objective function (1) by weighting the elements of the sets (cases) according to their similarity to query case. Let $d(c)$ be the distance of case $c \in \mathcal{C}$ to the query case, we can modify the objective function in the following way:

$$\text{maximise } \sum_{c\in\bigcup_{s\in\mathcal{S}} C(s,f(s))} \frac{1}{d(c)+1} \tag{3}$$

We solved this problem using a very simple complete search procedure. The low number of solvers in CPHYDRA (5) and number of similar cases (10 to 50) makes the problem tractable. However, for a large number of solvers a more sophisticated approach would be necessary.

Often, the constraint program above can be trivially solved by allocating to each solver s an amount of CPU-time t such that $|C(s,t)|$ is maximised. For instance consider the situation where we have five solvers, and none of them can solve any more instances after $t = 100$. In this case the objective function is trivially maximised, without violating the constraints, by allocating 100 seconds to each solver. In other words, in this type of situation the schedule, as defined previously, is useless. We, therefore, distinguish these cases and apply a simple alternative procedure. We first disregard solvers that are dominated by some others. That is, let t be a time point such that no case can be solved by any solver in less than t seconds. We say that s_1 is dominated iff (1) $\exists s_2 \in \mathcal{S}$ such that $C(s_1,t) \subset C(s_2,t)$, or (2) there exists $s_2 \in \mathcal{S}$ such that $C(s_1,t) = C(s_2,t)$ and there exists t' such that $C(s_2,t') = C(s_2,t)$ and $C(s_1,t') \subset C(s_2,t')$, or (3) $C(s_1,t) = C(s_2,t)$ for all t and s_2 comes before s_1 for some arbitrary static order (tie breaking). Since we are focusing on maximising the probability of solving the query instance within the time limit, the order does not matter. However, it is important to notice that if one wants to minimise the expected solving time, the chosen order can be significant. Once dominated solvers are eliminated, the remaining CPU-time is distributed amongst non-dominated solvers proportionally to the amount of CPU-time already assigned. This corresponds to the maximally risk-aggressive algorithm portfolio that seeks to maximise the probability of finding a solution. However, this does not necessarily minimise the expected time for finding a solution.

3 Experimental Results

Experimental Aims. Given that our aim is to produce a portfolio of solvers for the CSP Competition, to test different strategies we use the percentage of problems solved by each strategy as a metric, where the total number of problems is the number of problems that were solved by at least one solver within the corresponding time cutoff. Notice that in Figure 2 the solver `Buggy` has a higher percentage of solved problems after five minutes than when it is able to use the full thirty minutes, i.e., it solved more instances in five minutes relative to other solvers, but not in absolute value.

Experimental Data and Methodology. The following tests ran on data from the 2006 CSP Solver Competition. We used this data as it is complete and has a diverse range of solvers. The complete case base therefore contains 3013 cases, one for each instance solved by at least one entrant of the competition within a 30 minutes cutoff. All the results were obtained by running a randomised ten-fold cross validation ten times and averaging the results. We compared the five best solvers of the previous competition (`Buggy`, `Bprolog`, `Sugar`, `Mistral` and `Abscon`) against CPHYDRA with the same solvers in the portfolio, and using the three following strategies:

Split schedule: A schedule giving each solver an equal portion of the total time.
Static schedule: A schedule generated as in Section 2.3 using the entire case base.

Dynamic schedule: A schedule generated as in Section 2.3 using the k nearest neighbours of the target case ($k = 10$).

Experimental Results. It is clear from Figure 2 that even a naive schedule such as dividing the time evenly between the solvers performs very well compared to the best individual solver with an 8-10% greater success rate. In this graph, we plot the percentage of instances solved, within either 30 minutes, 5 minutes of 1 minutes, for each individual solver and each of the time splitting strategies described above. Notice that the percentages are over the number of instances that were solved by at least one solver in the respective duration. However, the best scheduling approach only marginally outperforms the others (See Table 1). The performance of the split schedule degrades as the available time is decreased. There is a 1% difference between it and a dynamic schedule when 30 minutes is available whereas there is a 3% gap when the time available is decreased to 5 minutes and a 10% gap when there is only 1 minute available.

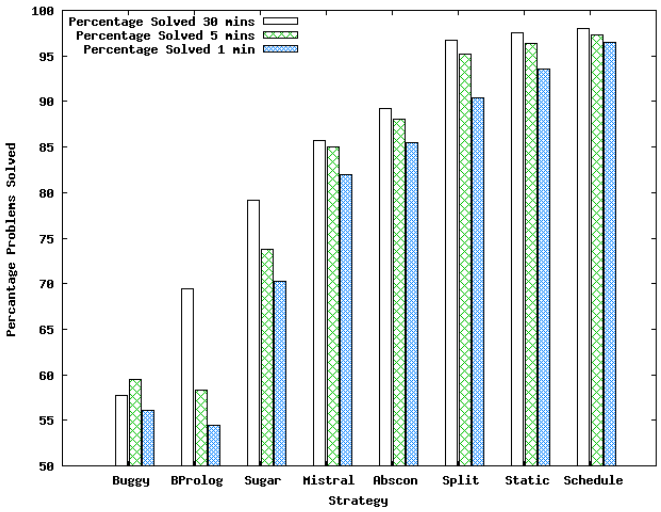


Fig. 2. A graph comparing the percentage of problems solved by different strategies when there are different amounts of time available. Running each solver in the portfolio exclusively, splitting the time evenly between the solvers, and two different schedules.

This is illustrated in Figure 3. The static and dynamic schedules begin to outperform a split schedule rapidly as the time available decreases. The static and dynamic schedules are roughly equivalent in terms of problems solved when the time allowed is more than 5 minutes. We believe the reason for this is that most of the problems in the CSP Solver Competition are solved in very short time period. Thus, given generous time constraints, any distributed schedule should do well. The efficiency of these approaches becomes apparent when the time available decreases and algorithm selection becomes more critical. The dynamic schedule clearly outperforms the static schedule when the time allowed is shorter.

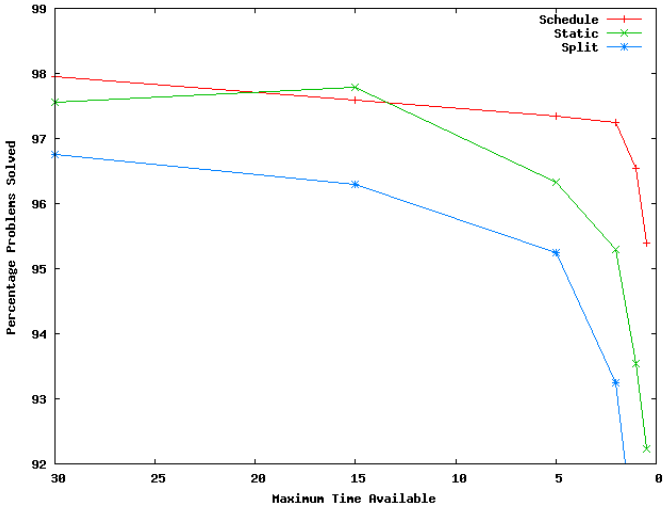


Fig. 3. Percentage (avg) of solved instances for each approach for different time-limits (minutes).

From Table 1 it is clear that the relative performance gap increases as the time available decreases. There is a clearly discernable trend as the time available approaches 30 seconds. The static and dynamic schedules go from being within 0.2% of each other at 30 minutes and at 15 minutes to being 1% apart at 5 minutes, 1.8% apart at 2 minutes to 3% apart at 1 minute. This is evidence that the case-based reasoning system is having a beneficial effect and the schedules generated on local neighbours are more robust than schedules generated from all known data.

Table 1. Table detailing the performance of each approach.

	Split	**Static**				**Dynamic**			
Time (Mins)	Solved	Best	Worst	Avg	Dev	Best	Worst	Avg	Dev
30	96.76	97.63	97.42	97.56	0.075	98.16	97.13	**97.86**	0.28
15	96.30	97.83	97.76	**97.79**	0.02	97.76	97.39	97.59	0.12
5	95.24	96.39	96.18	96.33	0.07	97.50	97.22	**97.34**	0.08
2	93.25	95.60	95.31	95.49	0.1	97.95	97.02	**97.25**	0.31
1	90.45	93.77	93.26	93.54	0.14	96.79	96.32	**96.54**	0.15
0.5	85.38	92.24	92.12	92.22	0.04	95.63	95.02	**95.39**	0.18

Preformance at the 2008 CSP Solver Competition. At the time of writing the final version of this paper, preliminary results from the 2008 CSP Solver Competition were available, showing that we achieved our goal to obtain better performance than each of the constituent solvers of the portfolio. Our competition entry of CPHYDRA comprised three solvers: `Abscon`, `Choco` and `Mistral`. Unfortunately the results of other entrants are not available yet. The results are summarized in Table 2. For each solver, we give the percentage of instances solved by each solver, as well as the average CPU-time

Table 2. Results summary for CPHYDRA and its constituent solvers at the 2008 CSP Solver Competition.

Category (#instances)	CPHYDRA		Abscon		Choco		Mistral	
	Solved	CPU-time	Solved	CPU-time	Solved	CPU-time	Solved	CPU-time
Binary Extensional (622)	**92%**	**62.44 s**	88%	93.26 s	89%	95.78 s	89%	70.21 s
Binary Intentional (634)	**94%**	71.37 s	81%	**43.40 s**	82%	55.89 s	82%	58.23 s
Global (501)	**84%**	80.83 s	37%	170.62 s	69%	69.69 s	80%	**56.59 s**
N-ary Extensional (607)	**97%**	**78.20 s**	90%	80.09 s	73%	189.10 s	94%	83.67 s
N-ary Intentional (660)	**86%**	54.70 s	74%	53.21 s	78%	49.70 s	80%	**32.02 s**

spent on solved instances. CPHYDRA dominates its constituent solvers in every category for the percentage of instances solved (the criterion used to rank solvers during the competition). More surprisingly, it is also competitive in average CPU-time. Therefore, CPHYDRA would win a competition against its constituent solvers.

4 Conclusion

We introduced CPHYDRA, a portfolio of constraint solvers exploiting a case base of problem solving experience. We detailed the novelties of our approach with respect to related work. In particular, CPHYDRA combines machine learning (CBR) with the idea of partitioning CPU-time between components of the portfolio in order to maximise the expected number of solved problem instances within a fixed time limit. Finally, we assessed the effectiveness of our portfolio over each of its constituent solvers using data from the most recent CSP Solver Competition showing the dominance of CPHYDRA.

References

1. Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Commun.*, 7(1):39–59, 1994.
2. Cormac Gebruers, Brahim Hnich, Derek Bridge, and Eugene Freuder. Using cbr to select solution strategies in constraint programming. In *Proc. of ICCBR*, pages 222–236, 2005.
3. Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artif. Intell.*, 126(1-2):43–62, 2001.
4. Bernardo E. Huberman, Rajan M. Lukose, and Tadd Hogg. An Economics Approach to Hard Computational Problems. *Science*, 275:51–54, 1997.
5. Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *CP*, pages 556–572, 2002.
6. James Little, Cormac Gebruers, Derek Bridge, and Eugene Freuder. Capturing constraint programming experience: A case-based approach. In A. M. Frisch, editor, *Workshop on Reformulating Constraint Satisfaction Problems*, 2002.
7. Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977.
8. Harry M. Markowitz. Portfolio selection. *Journal of Finance*, 7(1):77–91, 1952.
9. Luca Pulina and Armando Tacchella. A multi-engine solver for quantified boolean formulas. In *CP*, pages 574–589, 2007.
10. C. Riesbeck and R. Schank. *Inside Case-Based Reasoning*. Erlbaum, 1989.
11. Tzur Sayag, Shai Fine, and Yishay Mansour. Combining multiple heuristics. In *STACS*, pages 242–253, 2006.

12. Matthew J. Streeter, Daniel Golovin, and Stephen F. Smith. Combining multiple heuristics online. In *AAAI*, pages 1197–1203. AAAI Press, 2007.
13. Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. : The design and analysis of an algorithm portfolio for sat. In *CP*, pages 712–727, 2007.

Max-CSP competition 2008: `toulbar2` solver description

M. Sanchez[1], S. Bouveret[3], S. de Givry[1], F. Heras[2], P. Jégou[4], J. Larrosa[2], S. Ndiaye[4], E. Rollon[2], T. Schiex[1], C. Terrioux[4], G. Verfaillie[3], and M. Zytnicki[1]

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

Abstract. This document presents the key techniques used in `toulbar2` solver submitted to the Max-CSP competition 2008. `toulbar2` solves Weighted Constraint Satisfaction Problems (WCSPs), a generalisation of Max-CSP. Two complete solving methods that have been used for the competition are presented in this paper: Depth-First Branch and Bound (DFBB) and a new algorithm, Russian Doll Search with tree decomposition (RDS-BTD), which exploits the problem structure.
DFBB is commonly used to solve constraint optimization problems such as WCSPs. The worst-case time complexity of this algorithm can be improved by exploiting the constraint graph structure, identifying independent subproblems and caching their optima. However, the exploitation of the structure is done *a posteriori*: each time a new subproblem occurs, it has to be solved before its optimum can be used. RDS-BTD solves a relaxation of every subproblem before solving the whole problem, in the spirit of the Russian Doll Search algorithm. This relaxation allows to exploit subproblem lower bounds in a more proactive way.

1 Weighted Constraint Satisfaction Problem

A Weighted CSP (WCSP) is a quadruplet (X, D, W, m). X and D are sets of n variables and finite domains, as in a standard CSP. The domain of variable i is denoted D_i. The maximum domain size is d. For a set of variables $S \subset X$, we note $\ell(S)$ the set of tuples over S. W is a set of cost functions. Each cost function (or soft constraint) w_S in W is defined on a set of variables S called its scope and assumed to be different for each cost function. A cost function w_S assigns costs to assignments of the variables in S i.e. $w_S : \ell(S) \rightarrow [0, m]$. The set of possible costs is $[0, m]$ and $m \in \{1, \ldots, +\infty\}$ represents an intolerable cost. Costs are combined by the bounded addition $\oplus$, such as $a \oplus b = \min\{m, a + b\}$ and compared using $\geq$. The operation $\ominus$ subtracts a cost b from a larger cost a where $a \ominus b = (a - b)$ if $a \neq m$ and m otherwise.

For unary/binary cost functions, we use simplified notations: a unary (resp. binary) cost function on variable(s) i (resp. i and j) is denoted w_i (resp. w_{ij}). If they do not exist, we add to W a unary cost function w_i for every variable i, and a nullary cost function, noted $w_\varnothing$ (a constant cost payed by any assignment). All these additional cost functions have initial cost 0, leaving the semantics of the problem unchanged.

The cost of a complete assignment $t \in \ell(X)$ in a problem $P = (X, D, W, m)$ is $Val_P(t) = \bigoplus_{w_S \in W} w_S(t[S])$ where $t[S]$ denotes the usual projection of a tuple on the set of variables S. The problem of minimizing $Val_P(t)$ is an optimization problem with an associated NP-complete decision problem.

Enforcing a given local consistency property on a problem P consists in transforming $P = (X, D, W, m)$ in a problem $P' = (X, D, W', m)$ which is equivalent to P ($Val_P = Val_{P'}$) and which satisfies the considered local consistency property. This enforcing may increase $w_\varnothing$ and provide an improved lower bound on the optimal cost. Enforcing is achieved using Equivalence Preserving Transformations (EPTs) moving costs between different scopes [12, 8, 4, 6, 1, 3, 2].

A classical complete solving method is Depth-First Branch and Bound (DFBB). We give its pseudo-code in Algorithm 1. It enforces at each search node a given local consistency property Lc (line 1). The pruning condition is applied if the resulting $w_\varnothing \geq m$ (line 2). m is updated to the cost of the last solution found (line 3). The initial call is DFBB(P, X, $\emptyset$). It assumes an already local consistent problem P and returns its optimum. P/A denotes the subproblem P under assignment A. The operator . is used to get an element of P. Function pop(S) returns an element of S and remove it from S.

DFBB worst-case time complexity is $O(d^n)$ and it uses linear space. In the next section, we briefly present how DFBB can be extended to exploit the problem structure.

2 Depth-First Branch and Bound with tree decomposition

Assuming connected problems, a tree decomposition of a WCSP is defined by a tree (C, T). The set of nodes of the tree is $C = \{C_1, \ldots, C_k\}$ where each C_e is a set of variables ($C_e \subset X$) called a cluster. T is a set of edges connecting clusters and forming a tree (a connected acyclic graph). The set of clusters C must cover all the variables ($\bigcup_{C_e \in C} C_e = X$) and all the cost functions ($\forall w_S \in W, \exists C_e \in C \text{ s.t. } S \subset C_e$). Furthermore, if a variable i appears in two clusters C_e and C_g, i must also appear in all the clusters C_f on the unique path from C_e to C_g in T.

For a given WCSP, we consider a rooted tree decomposition (C, T) with an arbitrary root C_1. We denote by $Father(C_e)$ (resp. $Sons(C_e)$) the parent (resp. set of sons) of C_e in T. The separator of C_e is the set $S_e = C_e \cap Father(C_e)$. The set of proper variables of C_e is $V_e = C_e \setminus S_e$.

The essential property of tree decompositions is that assigning S_e separates the initial problem in two subproblems which can then be solved independently. The first subproblem, denoted P_e, is defined by the variables of C_e and all its descendant clusters in T and by all the cost functions involving *at least* one proper variable of these clusters. The remaining cost functions, together with the variables they involve, define the remaining subproblem.

Example 1. Consider the MaxCSP problem depicted in Figure 1. It has eleven variables with two values (a, b) in their domains. Binary cost functions of difference ($w_{ij}(a,a) = w_{ij}(b,b) = 1, w_{ij}(a,b) = w_{ij}(b,a) = 0$) are represented by edges connecting the corresponding variables. In this problem, the optimal cost is 5 and it is attained with e.g. the assignment (a,b,b,a,b,b,a,b,b,a,b) in lexicographic order. A C_1-rooted tree decomposition with clusters $C_1 = \{1,2,3,4\}, C_2 = \{4,5,6\}, C_3 = \{5,6,7\}, C_4 = \{4,8,9,10\}$,

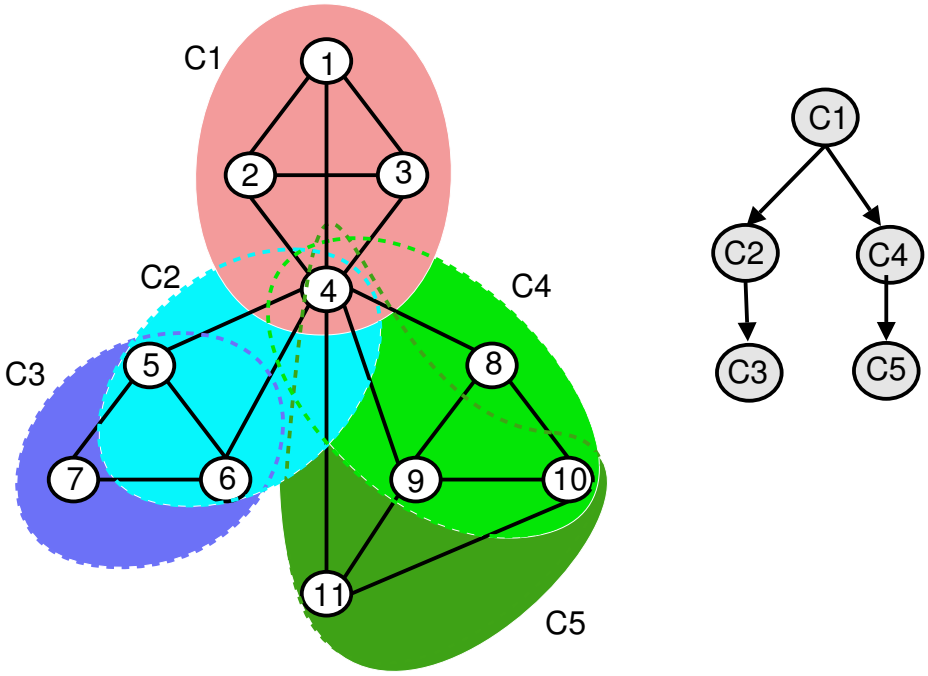


Fig. 1. The constraint graph of Example 1 and its associated tree decomposition.

and $C_5 = \{4,9,10,11\}$, is given on the right hand-side in Figure 1. For instance, C_1 has sons $\{C_2, C_4\}$, the separator of C_3 with its father C_2 is $S_3 = \{5,6\}$, and the set of proper variables of C_3 is $V_3 = \{7\}$. The subproblem P_3 has variables $\{5,6,7\}$ and cost functions $\{w_{5,7}, w_{6,7}, w_7\}$ (w_7 initially empty). P_1 corresponds to the whole problem.

Depth-First Branch and Bound with Tree Decomposition (BTD) [7, 5] exploits this property by restricting the variable ordering. Imagine all the variables of a cluster C_e are assigned before any of the remaining variables in its son clusters and consider a current assignment A. Then, for any cluster $C_f \in Sons(C_e)$, and for the current assignment A_f of the separator S_f, the subproblem P_f under assignment A_f (denoted P_f/A_f) can be solved independently from the rest of the problem. If memory allows, the *optimal cost* of P_f/A_f may be recorded which means it will never be solved again for the same assignment of S_f.

In [5], we show how to exploit a better initial upper bound for solving P_f. However this has the side-effect that the optimum of P_f may be not computed but only a lower bound. The lower bound and the fact it is optimal can be recorded in LB_{P_f/A_f} and Opt_{P_f/A_f} respectively, initially set to 0 and *false*.

As in DFBB, BTD enforces local consistency during search. However, local consistency may move costs between clusters, thereby invalidating previously recorded information. We store these cost moves in a specific *backtrackable* data structure ΔW as defined in [5]. During the search, we can obtain the total cost that has been moved

out of the subproblem P_f/A_f by summing up all the $\Delta W_i^f(a)$ for all values (i,a) in the separator assignment A_f and correct any recorded information: $LB'_{P_f/A_f} = LB_{P_f/A_f} \ominus \bigoplus_{i \in S_f} \Delta W_i^f(A_f[i])$.

Moreover, we keep the nullary cost function local to each cluster: $w_\varnothing = \bigoplus_{C_e \in C} w_\varnothing^e$.

For pruning the search, BTD uses the maximum between local consistency and recorded lower bounds as soon as their separator is completely assigned by the current assignment A. We denote by $lb(P_e/A)$ this lower bound:

$$lb(P_e/A) = w_\varnothing^e \oplus \bigoplus_{C_f \in Sons(C_e)} \max(lb(P_f/A), LB'_{P_f/A_f}) \quad (1)$$

Example 2. In the problem of Example 1, variables $\{1,2,3,4\}$ of C_1 are assigned first, e.g. using a dynamic variable ordering *min domain / max degree* inside each cluster.

Let assume $A = \{(4,a),(1,a),(2,b),(3,b)\}$ be the current assignment[5]. Enforcing EDAC local consistency [6] on P_1/A produces $w_\varnothing^1 = 2, w_\varnothing^2 = w_\varnothing^4 = 1, w_\varnothing^3 = w_\varnothing^5 = 0$, resulting in $lb(P_1/A) = \bigoplus_{C_e \in C} w_\varnothing^e = 4$ (no lower bound recorded yet).

Then, subproblems $P_2/\{(4,a)\}$ and $P_4/\{(4,a)\}$ are solved independently, resulting in $LB_{P_2/\{(4,a)\}} = 1$, $LB_{P_4/\{(4,a)\}} = 2$, $Opt_{P_2/\{(4,a)\}} = Opt_{P_4/\{(4,a)\}} = true$ (no initial upper bound) which are recorded. A first complete assignment of cost $w_\varnothing^1 \oplus LB_{P_2/\{(4,a)\}} \oplus LB_{P_4/\{(4,a)\}} = 5$ (all ΔW costs are zero in this case) is found.

In Algorithm 1, we present the pseudo-code of the BTD algorithm combining tree decomposition and a given level of local consistency Lc. This algorithm uses our initial enhanced upper bound (line 4), value removal based on local cuts [5] and lower bound recording (lines 6 and 7). The initial call is BTD(P_1, V_1, $\emptyset$, 0), with $P_1 = P$, an already local consistent problem, returning its optimum.

The lower bound $lb(P_e/A)$ of Equation 1 does not take into account a possible recorded lower bound LB_{P_e/A_e}, which may exist if $Opt_{P_e/A_e} = false$ and the same subproblem is solved again. We therefore ensure a monotonically increasing lower bound during the search by passing the best lower bound found recursively (line 5 and 9), resulting in a stronger pruning condition (line 8).

BTD time complexity is $O(md^{w+1})$ with $w = \max_{C_e \in C} |C_e| - 1$, the maximum cluster size minus one, called the tree-width of the tree decomposition. Its memory complexity is bounded by $O(d^s)$ with $s = \max_{C_e \in C} |S_e|$, the maximum separator size [5].

3 Russian Doll Search with tree decomposition

The original Russian Doll Search (RDS) algorithm [13] consists in solving n nested subproblems of an initial problem P with n variables. Given a fixed variable order, it starts by solving the subproblem with only the last variable. Next, it adds the preceding variable in the order and solves this subproblem with two variables, and repeats this process until the complete problem is solved. Each subproblem is solved by a DFBB

[5] Variable 4 has been selected first as it has the highest degree in C_1.

Algorithm 1: DFBB, BTD, and RDS-BTD algorithms.

Function DFBB(P, V, A) : $[0,+\infty]$
  if $(V = \emptyset)$ **then**
    return $P.w_\varnothing$ /* *A new solution is found for P* */ ;
  else
    i :=pop(V) /* *Choose an unassigned variable of P* */ ;
    $d := P.D_i$;
    /* *Enumerate every value in the domain of i* */ ;
    while $(d \neq \emptyset$ and $P.w_\varnothing < P.m)$ **do**
      a :=pop(d) /* *Choose a value* */ ;
1   P' :=Lc($P/A \cup \{(i,a)\}$) /* *Enforce local consistency on* $P/A \cup \{(i,a)\}$ */ ;
2   **if** $(P'.w_\varnothing < P.m)$ **then**
3       $P.m$:=DFBB(P', V, $A \cup \{(i,a)\}$) ;
    return $P.m$;

Function BTD(P_e, V, A, blb) : $[0,+\infty]$
  if $(V = \emptyset)$ **then**
    $S := Sons(C_e)$;
    /* *Solve all cluster sons whose optima are unknown* */ ;
    while $(S \neq \emptyset$ and $lb(P_e/A) < P_e.m)$ **do**
      C_f :=pop(S) /* *Choose a cluster son* */ ;
      if $(not(Opt_{P_f/A_f}))$ **then**
4       $P_f.m := P_e.m \ominus lb(P_e/A) \oplus lb(P_f/A_f)$;
5   res :=BTD(P_f, V_f, A, $lb(P_f/A_f)$) ;
6       $LB_{P_f/A_f} := res \oplus \bigoplus_{i \in S_f} \Delta W_i^f(A[i])$;
7       $Opt_{P_f/A_f} := (res < P_f.m)$;
    return $lb(P_e/A)$ /* *A new solution is found for* P_e */ ;
  else
    i :=pop(V) /* *Choose an unassigned variable in* C_e */ ;
    $d := P_e.D_i$;
    /* *Enumerate every value in the domain of i* */ ;
    while $(d \neq \emptyset$ and $\max(blb, lb(P_e/A)) < P_e.m)$ **do**
      a :=pop(d) /* *Choose a value* */ ;
      P'_e :=Lc($P_e/A \cup \{(i,a)\}$) /* *Enforce local consistency on* $P_e/A \cup \{(i,a)\}$ */ ;
8   **if** $(\max(blb, lb(P'_e/A \cup \{(i,a)\})) < P_e.m)$ **then**
9       $P_e.m$:=BTD(P'_e, V, $A \cup \{(i,a)\}$, $\max(blb, lb(P'_e/A \cup \{(i,a)\}))$) ;
    return $P_e.m$;

Function RDS-BTD(P, P_e^{RDS}) : $[0,+\infty]$
  foreach $C_f \in Sons(C_e)$ **do**
    RDS-BTD(P, P_f^{RDS}) ;
10 $P_e^{RDS}.m := P.m \ominus lb(P/\emptyset) \oplus lb(P_e^{RDS}/\emptyset)$;
11 $LB_{P_e^{RDS}}$:=BTD(P_e^{RDS}, V_e, $\{(i,$EAC(i)$)|i \in S_e\}$, $lb(P_e^{RDS}/\emptyset)$) ;
12 Set to *false* all recorded $Opt_{P_f/A}$ such that C_f is a descendant of C_e, $S_f \cap S_e \neq \emptyset$, $A \in \ell(S_f)$;
  return $LB_{P_e^{RDS}}$;

algorithm with a static variable ordering heuristic following the nested subproblem decomposition order. The lower bound combines the optimum of the previously solved subproblems with the lower bound produced by enforcing soft local consistency.

RDS-BTD, recently proposed in [10], applies the RDS principle to a tree decomposition. The main difference with RDS is that the set of subproblems to solve is defined by a rooted tree decomposition (C,T).

We define P_e^{RDS} as the subproblem defined by the proper variables of C_e and all its descendant clusters in T and by all the cost functions involving *only* proper variables of these clusters. P_e^{RDS} has no cost function involving a variable in S_e, the separator with its father, and thus its optimum is a lower bound of P_e for any assignment of S_e.

RDS-BTD solves $|C|$ subproblems ordered by a depth-first traversal of T, starting from the leaves to the root $P_1^{RDS} = P_1$.

Each subproblem P_e^{RDS} is solved by BTD instead of DFBB. This allows to exploit decomposition and caching done by BTD. Because caching is only performed on completely assigned separators, and considering all possible assignments of S_e would be too costly in memory and time, we assign S_e before solving P_e^{RDS}. This is needed since otherwise, caching on P_f, a descendant of C_e, with $S_f \cap S_e \neq \emptyset$, would use a partially assigned A_f. To assign S_e, we use the fully supported value of each domain[6] (maintained by EDAC [6]) as temporary values used for caching purposes only.

The advantage of using BTD is that recorded lower bounds can be reused during the next iterations of RDS-BTD. However, the optimum found by BTD for a given subproblem P_f when solving P_e^{RDS} is no more valid in $P_{Father(e)}^{RDS}$ due to possible cost functions between variables in $C_{Father(e)}$ and in P_f. At each iteration of RDS-BTD, after P_e^{RDS} is solved, we reset all Opt_{P_f/A_f} such that $S_f \cap S_e \neq \emptyset$ (line 12).

During search, RDS-BTD exploits the maximum between local consistency, recorded, and RDS lower bounds. Let $LB_{P_e^{RDS}}$ denote the optimum of P_e^{RDS} found by one iteration of RDS-BTD. Because costs can be moved between clusters, this information has to be corrected in order to be valid in the next iterations of RDS-BTD. For that, we use the maximum of ΔW on each current domain of the (possibly unassigned) separator variables. The lower bound corresponding to the current assignment A is then:

$$lb(P_e/A) = w_\varnothing^e \oplus \bigoplus_{C_f \in Sons(C_e)} \max(lb(P_f/A), LB'_{P_f/A_f}, LB_{P_f^{RDS}} \ominus \bigoplus_{i \in S_f} \max_{a \in D_i} \Delta W_i^f(a)) \quad (2)$$

Example 3. Applied on the problem of Example 1, RDS-BTD solves five subproblems $(P_3^{RDS}, P_2^{RDS}, P_5^{RDS}, P_4^{RDS}, P_1)$ successively. For instance, P_3^{RDS} has variable $\{7\}$ and cost function $\{w_7\}$. Before solving P_3^{RDS}, RDS-BTD assigns variables $\{5,6\}$ of the separator S_3 to their fully supported value ($\{(5,a),(6,a)\}$ in this example). In solving P_2^{RDS}, it can record e.g. the optimum of $P_3/\{(5,a),(6,a)\}$, equal to zero (recall that $w_{5,6}$ does not belong to P_3), that can be reused when solving P_1. In solving P_4^{RDS}, it can record e.g. the optimum of $P_5/\{(4,a),(9,a),(10,a)\}$, also equal to zero. However, due to the fact that variable 4 belongs to $S_5 \cap S_4$ and P_4^{RDS} does not contain $w_{4,11}$, this recorded information is only a lower bound for subsequent iterations of RDS-BTD. So, we set

[6] Fully supported value $a \in D_i$ such that $w_i(a) = 0$ and $\forall w_S \in W$ *with* $i \in S, \exists t \in \ell(S)$ *with* $t[i] = a$ such that $w_S(t) = 0$.

$Opt_{P_5/\{(4,a),(9,a),(10,a)\}}$= *false* before solving P_1. The resulting optima are: $LB_{P_3^{RDS}} = LB_{P_5^{RDS}} = 0, LB_{P_2^{RDS}} = LB_{P_4^{RDS}} = 1$ and $LB_{P_1^{RDS}} = 5$, the optimum of P_1.

In this simple example, for $A = \{(4,a),(1,a),(2,b),(3,b)\}$, $lb(P_1/A)$ using Equation 1 or 2 is the same because EDAC propagation provides lower bounds equal to RDS lower bounds. In the contrary, for $A = \emptyset$, $lb(P_1/\emptyset) = LB_{P_2^{RDS}} \oplus LB_{P_4^{RDS}} = 2$ using Equation 2 and $lb(P_1/\emptyset) = 0$ using Equation 1 (assuming EDAC local consistency in preprocessing and no initial upper bound).

We present the pseudo-code of the RDS-BTD algorithm in Algorithm 1. RDS-BTD call BTD to solve each subproblem P_e^{RDS} (line 11), using Equation 2 instead of Equation 1 to compute lower bounds. An initial upper bound for P_e^{RDS} is deduced from the global problem upper bound and the already computed RDS lower bounds (line 10). It initially assigns variables in S_e to their fully supported value (given by EAC function at line 11) as discussed above. The initial call is RDS-BTD(P, P_1^{RDS}). It assumes an already local consistent problem $P_1^{RDS} = P$ and returns its optimum.

Notice that as soon as a solution of P_e^{RDS} is found having the same optimal cost as $lb(P_e^{RDS}/\emptyset) = \bigoplus_{C_f \in Sons(C_e)} LB_{P_f^{RDS}}$, then the search ends thanks to the initial lower lound given at line 11.

The time and space complexity of RDS-BTD is the same as BTD.

4 Implementation details

We implemented DFBB and RDS-BTD in an open-source C++ solver named toulbar2[7]. DFBB uses default parameter values of toulbar2.

Dynamic variable ordering (*min domain / max degree*, breaking ties with maximum unary cost) is used inside clusters (RDS-BTD) and by DFBB. EDAC local consistency is enforced on binary [6] and ternary [11] cost functions during search. Larger arity cost functions are delayed from propagation until they become ternary or less.

We use the Maximum Cardinality Search heuristic to build a tree decomposition and choose the largest cluster as the root. In order to relax the restriction imposed by RDS-BTD on the dynamic variable ordering heuristic, we propose to merge clusters with their parent if their separator is too large. Starting from the leaves of a given tree decomposition, we merge a cluster with its parent if the separator size is strictly greater than $r = 4$ (parameter *B2r4* in toulbar2).

Recorded (and if available RDS) lower bounds are exploited by local consistency enforcing as soon as their separator variables are fully assigned. If the recorded lower bound is optimal (Opt_{P_e/A_e} =*true*) or strictly greater than the one produced by local consistency, i.e. $\max(LB'_{P_e/A_e}, LB_{P_e^{RDS}} \ominus \bigoplus_{i \in S_e} \Delta W_i^e(A[i]) > \bigoplus_{P_f \subseteq P_e} w_\emptyset^f$, then the corresponding subproblem (P_e/A_e) is deconnected from local consistency enforcing and the positive difference in lower bounds is added to its parent cluster lower bound ($w_\emptyset^{Father(C_e)}$), allowing possible new value removals by node consistency enforcing on the remaining problem.

[7] Version 0.7 available at http://mulcyber.toulouse.inra.fr/gf/project/toulbar2

All the solving methods exploit a binary branching scheme depending on the domain size d of the branching variable. If $d > 10$ then it splits the *ordered* domain into two parts (by taking the middle value), else the variable is assigned to its EDAC fully supported value or this value is removed from the domain. In both cases, it selects the branch which contains the fully supported value first, except for RDS-BTD where it selects the branch which contains the value corresponding to the last solution(s) found first if available.

At each search node, before branching, DFBB and RDS-BTD eliminate all variables (except variables occuring in a separator for RDS-BTD) with a degree less than or equal to two, possibly creating new binary cost functions on the fly. They apply successively EDAC propagation (which may assign some variables and reduce current degrees) and 2-degree variable elimination until there is no more elimination nor propagation.

The dynamic variable ordering heuristic is modified by a conflict back-jumping heuristic as suggested in [9]. It branches on the same variable again if the first branch in the binary branching scheme was directly pruned by propagation.

No initial upper bound is provided.

Acknowledgments `toulbar2` solver has been partly funded by the French *Agence Nationale de la Recherche* (`STALDECOPT` project).

References

1. M. Cooper. High-order consistency in valued constraint satisfaction. *Constraints*, 10(3):283–305, 2005.
2. M. Cooper, S. de Givry, M. Sanchez, T. Schiex, and M. Zytnicki. Virtual arc consistency for weighted csp. In *Proc. of AAAI-08*, Chicago, IL, 2008.
3. M. Cooper, S. de Givry, and T. Schiex. Optimal soft arc consistency. In *Proc. of IJCAI-07*, pages 68–73, Hyderabad, India, 2007.
4. M. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154:199–227, 2004.
5. 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.
6. 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.
7. P. Jégou and C. Terrioux. Decomposition and good recording. In *Proc. of ECAI-2004*, pages 196–200, Valencia, Spain, 2004.
8. J. Larrosa and T. Schiex. Solving Weighted CSP by Maintaining Arc-consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.
9. C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Last conflict based reasoning. In *Proc. of ECAI-2006*, pages 133–137, Trento, Italy, 2006.
10. M. Sanchez, D. Allouche, S. de Givry, and T. Schiex. Russian doll search with tree decomposition. In *Workshop on Preferences and Soft Constraints*, Sydney,Australia, 2008.
11. M. Sanchez, S. de Givry, and T. Schiex. Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. *Constraints*, 13(1):130–154, 2008.
12. T. Schiex. Arc consistency for soft constraints. In *Proc. of CP-2000*, pages 411–424, Singapore, 2000.
13. G. Verfaillie, M. Lemaître, and T. Schiex. Russian Doll Search for Solving Constraint Optimization Problems. In *Proc. of AAAI-96*, pages 181–187, Portland, OR, 1996.

System Description of a SAT-based CSP Solver Sugar

Naoyuki Tamura[1], Tomoya Tanjo[2], and Mutsunori Banbara[1]

[1] Information Science and Technology Center, Kobe University, JAPAN
{tamura,banbara}@kobe-u.ac.jp
[2] Graduate School of Engineering, Kobe University, JAPAN

Abstract. This paper gives the system description of a SAT-based CSP solver Sugar submitted to the Third International CSP Solver Competition. The Sugar solver solves a finite linear CSP and MAX-CSP by translating it into a SAT problem using the order encoding method and then solving the translated SAT problem with an external SAT solver (e.g. MiniSat). In the order encoding method, a comparison $x \leq a$ is encoded by a different Boolean variable for each integer variable x and integer value a.

1 Introduction

This paper gives the system description of a SAT-based CSP solver *Sugar*[3] submitted to the Third International CSP Solver Competition.

The Sugar solver solves a finite linear CSP and MAX-CSP by translating it into a SAT problem by using *order encoding* method [1, 2] and then solving the translated SAT problem by a SAT solver, such as MiniSat [3] and PicoSAT [4].

The method of the order encoding is basically the same with the one used for job-shop scheduling problems by Crawford and Baker in [5] and studied by Soh, Inoue, and Nabeshima in [6–8]. It encodes a comparison $x \leq a$ 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 compared with the other encoding methods, such as direct encoding and support encoding, used by other SAT-based CSP solvers [9–13].

Axiom clauses with two literals, such as $\{\neg(x \leq a), x \leq a+1\}$ for each integer a, represent the order relation of an integer variable x. Clauses, for example $\{x \leq a, \neg(y \leq a)\}$ for each integer a, can be used to represent the constraint among integer variables, i.e. $x \leq y$.

2 Implementation

This section describes some implementation details of the Sugar solver.

[3] http://bach.istc.kobe-u.ac.jp/sugar/

```
(domain D0 0 2)
(int V0 D0)
(int V1 D0)
(predicate (P0 X0 X1) (ne X0 X1))
(relation R0 2 (conflicts (0 0) (1 1) (2 2)))
(P0 V0 V1)
(R0 V0 V1)
(alldifferent (V0 V1))
```

Fig. 1. Example of Sugar CSP description

2.1 Preprocessing

At a preprocessing stage, XCSP 2.1 file in the abridged notation is translated into a Sugar CSP file written in a Lisp-like list format.

Fig. 1 shows an example of the Sugar CSP file.

2.2 Conversion to a Clausal Form CSP

Before encoding CSP to SAT, Sugar converts the CSP into its clausal form (that is, Conjunctive Normal Form).

A clausal form CSP consists of a set of CSP literals, and a CSP literal is one of the followings:[4]

- A Boolean literal: p or $\neg p$ where p is a Boolean variable.
- A linear comparison literal: $\sum a_i x_i \leq b$ where a_i's and b are integer constants and x_i's are integer variables.
- A relation literal: a set of conflict tuples or support tuples with integer variable arguments (for example, `(R0 V0 V1)` in the Fig. 1 is represented as a relation literal).

Expressions other than $\sum a_i x_i \leq b$ are translated by using the conversion rules 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 respectively.

Basically, Sugar is not able to handle non-linear expressions in the current implementation except some special cases. For example, $xy < 0$ is translated into $((x < 0) \wedge (y > 0)) \vee ((x > 0) \wedge (y < 0))$, and xy is translated into if$(x = a_1, a_1 y, a_2 y)$ when the domain of x is $\{a_1, a_2\}$.

The alldifferent$(x_1, x_2, \ldots, x_n)$ constraint is translated into $\bigwedge_{i<j}(x_i \neq x_j)$ with extra pigeon hole constraints $\neg \bigwedge(x_i < lb + n - 1)$ and $\neg \bigwedge(x_i > ub - n + 1)$ where lb and ub are the lower and upper bounds of $\{x_1, x_2, \ldots, x_n\}$. Other global constraints are translated in a straightforward way.

Finally, constraints are converted into clausal form by using the Tseitin transformation of introducing new Boolean variables.

[4] Literals for multiplications and power functions will be used in a future implementation.

Expression	Replacement	Extra condition
$E < F$	$E + 1 \leq F$	
$E = F$	$(E \leq F) \wedge (E \geq F)$	
$E \neq F$	$(E < F) \vee (E > F)$	
$\max(E, F)$	x	$(x \geq E) \wedge (x \geq F) \wedge ((x \leq E) \vee (x \leq F))$
$\min(E, F)$	x	$(x \leq E) \wedge (x \leq F) \wedge ((x \geq E) \vee (x \geq F))$
$\mathrm{abs}(E)$	x	$(x \geq E) \wedge (x \geq -E) \wedge ((x \leq E) \vee (x \leq -E))$
E div c	q	$(E = c\,q + r) \wedge (0 \leq r) \wedge (r < c)$
$E \bmod c$	r	$(E = c\,q + r) \wedge (0 \leq r) \wedge (r < c)$
$\mathrm{if}(C, E, F)$	x	$(C \supset x = E) \wedge (\neg C \supset x = F)$

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

2.3 Constraint Propagation

Constraint propagation is done for the clausal form CSP to reduce the redundant integer variable values, CSP clauses, and CSP literals. AC-3 algorithm is used in the current implementation.

For example, more than half billion values were removed in the FISCHER11-6-fair instance.

2.4 SAT encoding

The clausal form CSP is encoded into a SAT problem by using the order encoding method [1].

Compared with the previous version submitted to the Second CSP solver competition [2], conflict tuples in extensional constraints are combined into conflict regions. For example, conflict tuples $((0,1),(0,2))$ for variables x and y was encoded into two clauses $\neg((x \geq 0) \wedge (x \leq 0) \wedge (y \geq 1) \wedge (y \leq 1))$ and $\neg((x \geq 0) \wedge (x \leq 0) \wedge (y \geq 2) \wedge (y \leq 2))$ in the previous version. They are now encoded into one clause $\neg((x \geq 0) \wedge (x \leq 0) \wedge (y \geq 1) \wedge (y \leq 2))$ by combining into coflict regions.

Support tuples are encoded by considering their complement space.

2.5 SAT solver

Sugar can use MiniSat [3] or PicoSAT [4] as an external SAT solver.

MiniSat is well known to be very efficient especially for unsatisfiable problems and widely used in many application areas. PicoSAT is a solver based on MiniSat with rapid restarts and reusing phases of assigned variables.

In our experiments, MiniSat is slightly better for many problems, PicoSAT, however, uses less memory and can solve some problems which are not solved by MiniSat under the CSP solver competition environment (that is, 900MiB memory limitation).

3 Conclusion

In this paper, we have described some implementation details of Sugar constraint solver submitted to the Third International CSP Solver Competition. The Sugar solver solves a finite linear CSP and MAX-CSP by translating it into a SAT problem using the order encoding method and then solving the translated SAT problem with an external SAT solver (e.g. MiniSat).

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. Tamura, N., Banbara, M.: Sugar: A CSP to SAT translator based on order encoding. In: Proceedings of the Second International CSP Solver Competition. (2008) 65–69
3. 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
4. Biere, A.: Adaptive restart strategies for conflict driven SAT solvers. In: Proceedings of the Theory and Applications of Satisfiability Testing (SAT 2008). (2008) 28–33
5. 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
6. 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)
7. Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M., Tamura, N.: A competitive and cooperative approach to propositional satisfiability. Discrete Applied Mathematics **154** (2006) 2291–2306
8. 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
9. Biere, A.: Translating CSP to SMV and then to SAT for the CSP'05 competition. In: Proceedings of the Second International Workshop on Constraint Propagation and Implementation, Volume II Solver Competition. (2005) 49–52
10. Berre, D.L.: CSP2SAT4J: A simple CSP to SAT translator. In: Proceedings of the Second International Workshop on Constraint Propagation and Implementation, Volume II Solver Competition. (2005) 59–66

11. Roussel, O.: Some notes on the implementation of csp2sat+zchaff, a simple translator from CSP to SAT. In: Proceedings of the Second International Workshop on Constraint Propagation and Implementation, Volume II Solver Competition. (2005) 83–88
12. van Dongen, M.R.C., Lecoutre, C., Roussel, O.: Results of the second CSP solver competition. In: Proceedings of the Second International CSP Solver Competition. (2008) 1–10
13. Berre, D.L., Lynce, I.: CSP2SAT4J: A simple CSP to SAT translator. In: Proceedings of the Second International CSP Solver Competition. (2008) 43–54

Sugar++: A SAT-Based MAX-CSP/COP Solver

Tomoya Tanjo[1], Naoyuki Tamura[2], and Mutsunori Banbara[2]

[1] Graduate School of Engineering, Kobe University, JAPAN
tanjo@stu.kobe-u.ac.jp
[2] Information Science and Technology Center, Kobe University, JAPAN
{tamura,banbara}@kobe-u.ac.jp

Abstract. This paper briefly describes some features of Sugar++, a SAT-based MAX-CSP/COP solver entering the Third International CSP Solver Competition. In our approach, a MAX-CSP is translated into a Constraint Optimization Problem (COP), and then it is encoded into a SAT problem by the order encoding method. SAT-encoded COP can be efficiently solved by using the incremental search feature of MiniSat solver.

1 Introduction

This paper briefly describes some features of `Sugar++`, a SAT-based MAX-CSP/COP solver entering the Third International CSP Solver Competition.

In our approach, a MAX-CSP is translated into a Constraint Optimization Problem (COP), and then it is encoded into a SAT problem by the *order encoding* method [1]. SAT-encoded COP can be efficiently solved by repeatedly using the MiniSat solver [2].

The order encoding method encodes a comparison $x \leq a$ by a different boolean variable for each integer variable x and integer value a. `Sugar` [3] is a SAT-based CSP solver based on order encoding, and its effectiveness has been shown through application to Open-Shop Scheduling in [1].

A solution to COP can be obtained by repeatedly solving a certain number of CSPs. In `Sugar`, each CSP is solved by a different MiniSat process independently. Therefore the learnt clauses generated in each process disappear and can not be reused. This slows down the execution speed.

To solve this problem, we have developed a SAT-based MAX-CSP/COP solver `Sugar++` that is an enhancement of `Sugar` by using the incremental search feature of MiniSat. `Sugar++` invokes only one MiniSat process to solve MAX-CSP/COP and can reuse the learnt clauses generated during the search.

2 Translating MAX-CSP into COP

Definition 1 (CSP). A Constraint Satisfaction Problem (CSP) is defined as a tuple $(V, d, \mathcal{C})$ where

- $V = \{x_1, \ldots, x_m\}$ is a set of variables,

- d is a mapping from x_i to a finite domain D_i containing possible values x_i may take, and
- $\mathcal{C} = \{C_1, \ldots, C_n\}$ is a set of constraints.

A *solution* to a CSP is an assignment α, a mapping from every variable $x_i \in V$ to an element of D_i so that every constraint in $\mathcal{C}$ is satisfied.

Definition 2 (COP). A Constraint Optimization Problem (COP) is defined as a tuple $(V, d, \mathcal{C}, v)$ where

- $(V, d, \mathcal{C})$ is a CSP, and
- $v \in V$ is an objective variable to be minimized [3].

A *solution* to a COP$(V, d, \mathcal{C}, v)$ is an assignment α which is a solution of CSP$(V, d, \mathcal{C})$ taking the minimum $\alpha(v)$ value.

Given a CSP, the Max-CSP problem is to find an assignment that minimizes the number of violated constraints. Therefore, the Max-CSP problem for a CSP$(V, d, \mathcal{C})$ can be translated into a COP$(V^*, d^*, \mathcal{C}^*, \mathit{cost})$ where

- $V^* = V \cup \{c_1, \ldots, c_n, \mathit{cost}\}$
  Each c_i represents the penalty of the constraint C_i, and *cost* is the objective variable to be minimized.
- $d^*(x) = \begin{cases} \{0, 1\} & \text{if } x = c_i \quad (1 \leq i \leq n) \\ \{0, \ldots, n\} & \text{if } x = \mathit{cost} \\ d(x) & \text{otherwise} \end{cases}$
- $\mathcal{C}^* = \{(c_i \geq 1) \vee C_i \mid 1 \leq i \leq n\} \cup \{\mathit{cost} \geq \sum c_i\}$

It is also possible to translate Weighted-CSP into COP in the same way by replacing $d^*(\mathit{cost})$ with $\{0, \ldots, \sum w_i\}$ and $\mathit{cost} \geq \sum c_i$ with $\mathit{cost} \geq \sum w_i c_i$ where w_i is the given positive weight for the constraint C_i.

3 Solving COP by using the incremental search of SAT solver

The `Sugar` constraint solver [3] can solve a finite linear COP by encoding it into a SAT problem based on the *order encoding* method [1], and then a SAT-encoded COP is solved by using the MiniSat solver [2]. The main feature of order encoding can be the natural representation of order relation on integers by encoding a comparison $x \leq a$ into a different boolean variable for each integer variable x and integer value a. In the followings, "$p(x, a)$" is used to represent the boolean variable for the comparison $x \leq a$.

The optimal value of COP$(V, d, \mathcal{C}, v)$ can be obtained by repeatedly solving CSPs.

$$\min \{a \in d(v) \mid \ CSP(V, d, \mathcal{C} \cup \{v \leq a\}) \ \mathit{has\ a\ solution}\}$$

[3] Without the loss of generality, we assume COPs as minimization problems.

A solution to COP can be efficiently found by bisection search with varying a as proposed in previous works [4–6].

Let P be a COP whose objective variable is v. Fig. 1 shows the minimization procedure of `Sugar` to find the optimal value of P. The outline of `minimize(`P`,` v`)` is as follows:

(1) Encodes P into a SAT problem and sets `S` to it. The `S` represents a SAT file in the actual implementation.
(2) Sets `found` to `false`. The `found` is a flag indicating whether a solution is found or not.
(3) Sets `lb` and `ub` to v's lower bound and v's upper bound + 1 respectively.
(4) If `lb` $<$ `ub` does not hold, goes to the step (10).
(5) Sets `a` to the value of $\lfloor$(`lb`+`ub`)$/2\rfloor$.
(6) Sets `c` to a unit clause $\{p(v,$ `a`$)\}$ where $p(v,$ `a`$)$ represents the boolean variable for the comparison $v \leq$ `a`.
(7) Executes the MiniSat with `S` $\cup \{$`c`$\}$ as an input SAT problem.
(8) Updates `ub` to the value of `a` and also sets `found` to `true` if the result is satisfiable. Otherwise updates `lb` to the value of `a`+1.
(9) Goes back to the step (4).
(10) If `found` is `true`, this procedure succeeds to find the optimal value of P. Otherwise fails.

`Sugar` encodes a COP into a SAT once at first, and repeatedly modifies only the clause corresponding $v \leq a$. However there has still remained some points to be improved.

- A number of MiniSat processes are invoked until the optimal value is found.
- The learnt clauses generated in each MiniSat process are not reused.

Reusing learnt clauses is effective to significantly reduce the search space. It is therefore very important to reuse learnt clauses. To solve this problem, we take advantage of the incremental search of MiniSat [2].

First, we modify the MiniSat so that it can deal with the following three commands from the standard input, where L_i means a literal:

- `add` $L_1\ L_2\ \cdots\ L_n$
  This command adds a clause $\{L_1, L_2, \ldots, L_n\}$ to the SAT clause database. The clause is never removed, and the conflict analysis in the further search uses this clause as well as those in the initial database.
- `solve` $L_1\ L_2\ \cdots\ L_m$
  This command makes the MiniSat to solve the SAT problem with an assumption $\{L_1\ L_2\ \cdots\ L_m\}$. This assumption is passed as an argument to the solve method of MiniSat and temporarily assumed to be true during the search. After solving the problem, this assumption is undone. It is noted that the assumption does not affect learnt clauses to be generated since the MiniSat's conflict detecting mechanism is independent of it.
- `exit`
  This command makes the MiniSat terminate.

```
procedure minimize(P, v);
begin
  S := SAT encoding of P;
  found := false;
  lb := lower bound of v;
  ub := upper bound of v + 1;
  while lb < ub do
    a := ⌊(lb+ub)/2⌋;
    c := {p(v, a)};
    result := execute MiniSat for S ∪ {c};
    if result is satisfiable then
      found := true;
      ub := a;
    else
      lb := a + 1;
    end if
  end while
  if found then
    OPTIMUM FOUND;
  else
    UNSATISFIABLE;
  end if
end
```

Fig. 1. The minimization procedure of `Sugar`

We have developed the `Sugar++` system that is an enhancement of `Sugar` to use the incremental version of MiniSat described above. The main features of `Sugar++` are as follows:

- Bi-directional IO is used to communicate between `Sugar++` and the incremental version of MiniSat.
- MiniSat is invoked only once, therefore, SAT file is parsed only once.
- Learnt clauses are reused during the search.

Fig. 2 shows the new minimization procedure of `Sugar++`. The outline of `minimize(`P`, `v`)` is as follows:

(1) Encodes P into a SAT problem and sets `S` to it. The `S` represents a SAT file in the actual implementation.
(2) Sets `found` to `false`. The `found` is a flag indicating whether a solution is found or not.
(3) Sets `lb` and `ub` to v's lower bound and v's upper bound + 1 respectively.
(4) Starts the incremental version of MiniSat process with `S`, and the process waits for the commands: `add`, `solve`, and `exit`.
(5) If `lb` $<$ `ub` does not hold, goes to the step (10).
(6) Sets `a` to the value of $\lfloor$(`lb`+`ub`)$/2\rfloor$.
(7) Sends the command '`solve` $p(v,$ `a`$)$' to the MiniSat process.

```
procedure minimize(P, v);
begin
  S := SAT encoding of P;
  found := false;
  lb := lower bound of v;
  ub := upper bound of v + 1;
  Start a MiniSat process with S;
  while lb < ub do
    a := ⌊(lb+ub)/2⌋;
    Send 'solve p(v, a)' to MiniSat;
    result := receive the result from MiniSat;
    if result is satisfiable then
      found := true;
      ub := a;
      Send 'add p(v, a)' to MiniSat;
    else
      lb := a + 1;
      Send 'add -p(v, a)' to MiniSat;
    end if
  end while
  Send 'exit' to MiniSat;
  if found then
    OPTIMUM FOUND;
  else
    UNSATISFIABLE;
  end if
end
```

Fig. 2. The new minimization procedure of Sugar++

(8) Sends the command 'add $p(v, \mathtt{a})$' to the MiniSat process after updating ub to the value of a and setting found to true if the result is satisfiable. Otherwise updates lb to the value of a+1 and sends the command 'add $-p(v, \mathtt{a})$'.
(9) Goes back to the step (5).
(10) Sends the command 'exit' to the MiniSat process.
(11) If found is true, this procedure succeeds to find the optimal value of P. Otherwise fails.

Compared with Sugar, Sugar++ succeeds not only in reducing the overhead of parsing SAT problems but also in reusing learnt clauses during the search.

4 Conclusion

In this paper, we have described some features of Sugar++, a SAT-based MAX-CSP/COP solver entering the Third International CSP Solver Competition. Sugar++ solves a MAX-CSP by translating it into a COP, and encoding it into a SAT problem by the *order encoding* method. SAT-encoded COP is solved by using the incremental version of MiniSat solver.

Acknowledgments

We would like to give thanks to the competition organizers for their efforts.

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. Tamura, N., Banbara, M.: Sugar: A CSP to SAT translator based on order encoding. In: Proceedings of the Second International CSP Solver Competition. (2008) 65–69
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 **154** (2006) 2291–2306
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

BPSOLVER 2008

Neng-Fa Zhou

CUNY Brooklyn College & Graduate Center

Abstract. This paper provides details of the B-Prolog CSP solver (bpsolver) submitted to the Third International CSP Solver Competition. This version has several improvements over the version submitted to the previous competition, including new propagators for extensional constraints and more efficient propagators for non-linear constraints. This paper also describes the implementation of the global constraints for the competition.

1 Introduction

This paper provides details of bpsolver submitted to the Third International CSP Solver Competition. The constraint propagators used in the solver are implemented in AR (action rules) [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 new version has several improvements over the version submitted to the previous competition [6]. First, unlike in the previous version where extensional constraints are represented as hashtables, extensional constraints in the new version are encoded as finite-domain variables. This encoding, which has been employed by other solvers (e.g., Mistral [2]), is much more space efficient than hashtables especially for dense relations. Second, improved propagators are used for non-linear constraints. The `dom` event is intensively used in the implementation of the propagators to speed up propagation. Third, new global constraints including `element` and `cumulative` are considered for the third competition in addition to `all_distict`. The implementation of `all_distinct` remains the same. It is based on a weak version of the hall-set finding algorithm [4], which is weaker in terms of pruning power than Régin's filtering algorithm [3]. Channeling constraints, which are facilitated by the `dom_any` event in AR, are used to remedy the weakness of the algorithm. The implementation of `element` also relies on the `dom` event. A weak-form edge-finding algorithm [1] is implemented for `cumulative` when jobs are required to be mutually disjunctive.

The B-Prolog system is available at *www.probp.com*, and the submitted solver which accepts the XCSP format will be made available at *www.probp.com/solvers*.

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 [5]. 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 [5] 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.

[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.

– `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:

```
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

An extensional constraint, whether conflict or support, is encoded as a finite domain variable. Since a conflict constraint can be easily converted to a support constraint by replacing the relation with its complement, we only consider how to encode support constraints.

Let $(V_1, \ldots, V_n)$ *in* R be a support constraint, where V_1, ..., V_n are component variables and R be a relation, i.e., a set of tuples. Let C_i is the set of values

[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`.

obtained by projecting R to the ith column. The domain constraint V_i *in* C_i is posted first so that no value in the domain of V_i can remain unless it is also in C_i. Let D_i be the domain of V_i after this filtering. A tuple $(e_1, \ldots, e_n)$ in the relation is said to be *valid* if for any $i = 1, \ldots, n$, $e_i \in D_i$. We assume that R contains valid tuples only. A value e is said to be *supported* in the ith column if a tuple exists in the relation whose ith argument is e.

Let l_i be the lower bound and u_i be the upper bound of D_i, and $s_i = u_i - l_i + 1$. The code for a tuple $(e_1, \ldots, e_n)$ is defined as follows:

$\text{code}(e_n) = e_n - l_n$.
$\text{code}(e_i,e_{i+1},\ldots,e_n) = e_i - l_i + s_i \times \text{code}(e_{i+1},\ldots,e_n)$.

For a support constraint $(V_1, \ldots, V_n)$ *in* R, a new finite domain variable V is created whose domain contains the codes of the tuples in R. A bit vector of $s_1 \times \ldots \times s_n$ bits is needed to represent the finite domain. This representation is very compact when the relation is dense. For sparse relations, however, this representation can be very space consuming. The relationship between V and the component variables is maintained. Whenever the domain of a component variable is updated, the domains of the other component variables are filtered to contain only supported values. The time complexity is not good because it is in the order of $s_1 \times \ldots \times s_n$, not the size of the relation.

4 Propagators for Linear 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 [5].

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 Propagators for Nonlinear Constraints

The `dom` event also plays a big role in the implementation of non-linear constraints. Consider the nonlinear constraint $abs(X) = Y$. For each value in the domain of X, there should exist a unique support in the domain of Y. Whenever a value y has been excluded from the domain of Y, both y and $-y$ must be excluded from the domain of X to make the constraint arc consistent. For each value in the domain of Y, there may exist two supports in the domain of X. Whenever a value x has been excluded from the domain of X, if $-x$ no longer exists in the domain of X, then $abs(x)$ must be excluded from the domain of Y to make the constraint arc consistent.

6 The `all_distinct` Constraint

Many algorithms have been proposed for maintaining different levels of consistency for the `all_distinct` constraint [4]. The filtering algorithm by Régin [3] 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 [5] 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.

7 The `element` Constraint

The constraint `element`(I,L,X) means that the Ith element of L is X, where I and X are domain variables, and L a list of domain variables.

Let L be a list `[`e_1, ...,e_n`]`. Then I must be in the range of $1..n$. On one hand, each value in the domain of I must be supported by X. As long as X is known not to be equal to some element e_i, i can be excluded from the domain of I. This relationship can be expressed by the following entailment constraint X `#\=` e_i `#=>` I `#\=` i.

When L is a list of integers, we can have more efficient propagators. We use a counter C_{e_i} for e_i that tells the number of occurrences of e_i in L. Each time a value i is excluded from the domain of I, the counter C_{e_i} is decremented. If it becomes zero, then we can post the constraint X `#\=` e_i.

The entailment constraints X `#\=` e_i `#=>` I `#\=` i $(1 \leq i \leq n)$ can be encoded using only one propagator thanks to the availability of the `dom_any` event. To achieve this, we represent L as an association map so that for each e_i its indexes and counter can be returned in constant time. The following shows the propagator:

```
element_X_to_I(X,I,Map),var(X),
   {dom_any(X,E)}
```

```
   =>
   map_get_indexes(Map,E,Indexes),
   I notin Indexes.
element_X_to_I(X,I,Map) => true.
```

Whenever a value `E` is excluded from `X`'s domain, the constraint `I notin Indexes` ensures that `I` cannot take the index of any occurrence of `E`. When `X` is instantiated to be a non-variable term, the propagator vanishes.

On the other hand, each value in the domain of X must be supported by values in the domain of I as well. The propagation from I to X can be done using only one propagator as well. Let `Vect` be a vector representation of L, with which the element of a given index can be returned in constant time. The following defines the propagator:

```
element_I_to_X(I,X,Vect,Map),var(I),
   {dom_any(I,Ii)}
   =>
   arg(Ii,Vect,Ei),
   decrement_counter(Map,X,Ei).
element_I_to_X(I,X,Vect,Counters) => true.
```

The call `decrement_counter(Map,X,Ei)` decrements the counter of `Ei` and executes the primitivet `exclude(X,Ei)` if `Ei`'s counter becomes zero.[3]

In addition to the two propagators `element_X_to_I` and `element_I_to_X`, we need two extra propagators to handle `ins`(I) and `ins`(X) events. When I is instantiated to an integer i, the constraint X `#=` e_i is generated, and when X is instantiated, the domain of I is reduced to contain only the indexes of the occurrences of X in L.

8 The `cumulative` Constraint

The constraint `cumulative(Starts,Durations,Resources,Limit)` is useful for describing and solving scheduling problems. The arguments `Starts`, `Durations`, and `Resources` are lists of integer domain variables of the same length and `Limit` is an integer domain variable. Let `Starts` be [S_1,...,S_n], `Durations` be [D_1,...,D_n] and `Resources` be [R_1,...,R_n]. For each job `i`, S_i represents the start time, D_i the duration, and R_i the units of resources needed. `Limit` is the units of resources available at any time. The constraint ensures that the amount of resources used at any time does not exceed the limit. When evey job consumes only a unary resource (i.e., `Resources` is a list of 1's) and `Limit` is 1, then the constraint ensures that the jobs are mutually disjunctive.

For each job and each point over the time span, a Boolean variable is generated. The cumulative constraint entails that the cumulative sum of the resources used at each time point is not greater than the limit. Assume the time span is

[3] To be more efficient, the primitive `exclude(X,Ei)` is executed when `Ei`'s counter is equal to 1 before it is decremented.

from 1 to m, and the Boolean variables generated for each job i are $B_{i1}, \ldots, B_{im}$ $(i = 1, ..., n)$. The cumulative constraint means that $\Sigma_{i=1}^{n}(B_{ij} \times R_i) \leq Limit$ holds for every time point j.

A weak-form edge-finding algorithm [1] is implemented for `cumulative` when the jobs are required to be mutually disjunctive. The idea of edge finding is to examine a job with respect to a set of other jobs and to find out if the job must be processed before or after the set of jobs. Let A be a job and Ω be a set of jobs that does not contain A. Let $p(\Omega)$ denotes the total duration time, $min(start(\Omega))$ the earliest possible start time and $max(end(\Omega))$ the latest possible end time of the jobs in Ω. If

$$min(start(\Omega)) + p(\Omega \cup \{A\}) > max(end(\Omega \cup \{A\}))$$

then the job A must be processed *before* any job in Ω. Similarly, if

$$min(start(\Omega \cup \{A\})) + p(\Omega \cup \{A\}) > max(end(\Omega))$$

then the job A must be processed *after* any job in Ω.

Since there are an exponential number of subsets of jobs to consider for a given job, it is inviable to examine all subsets. In our implementation, three random sequences are maintained and for each job the subsets of jobs that occur before the job in the sequences are examined. Of course, this is very ad hoc and a systematic and advanced algorithm developed in the scheduling community needs to be employed.

References

1. R. Barták. Practical constraints: A tutorial on modelling with constraints. In *Proceedings of the 5th Workshop on Constraint Programming in Decision and Control*, 2003.
2. Emmanuel Hebrard. Mistral, a constraint satisfaction library. In M.R.C. van Dongen, Christophe Lecoutre, and Olivier Roussel, editors, *Proceedings of the Second International CSP Solver Competition*, pages 35–42, 2008.
3. 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.
4. W. J. van Hoeve. The alldifferent constraint: A survey. Technical report, 2001.
5. Neng-Fa Zhou. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming (TPLP)*, 6(5):483–508, 2006.
6. Neng-Fa Zhou. A report on the B-Prolog CSP solver. In M.R.C. van Dongen, Christophe Lecoutre, and Olivier Roussel, editors, *Proceedings of the Second International CSP Solver Competition*, pages 89–95, 2008.