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).

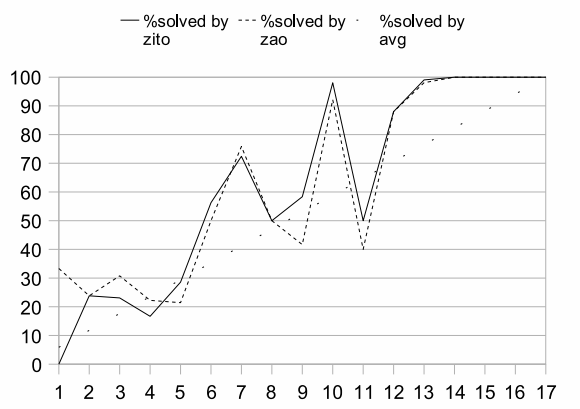

Fig. 1. Percentage of instances solved by *zito* and *zao* (yy) from the sets of instances solved by a number of solvers (xx). The dotted line shows the number of instances solved by an hypothetical average solver (assuming that solved instances distribute uniformly across all solvers).

[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.

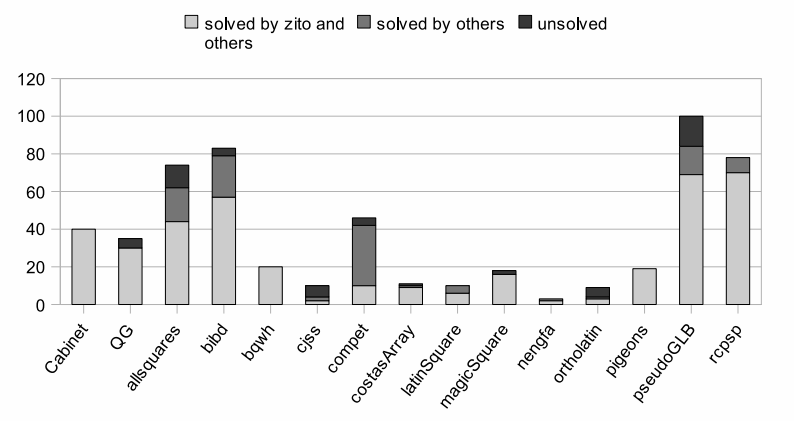

Fig. 2. Number of instances solved by *zito* in each problem family.

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.

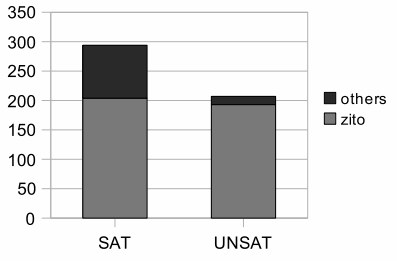

Fig. 3. Number of SAT and UNSAT instances solved by *zito*.

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 Sais. Boosting systematic search by weighting constraints. In *ECAI*, pages 146–150, Spain, 2004. IOS Press.
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 SAIS, SÃ©bastien 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.