Proc. SymCon'03

Symmetry and Propagation: Refining an AC algorithm

Ian P. Gent & Iain McDonald

School of Computer Science, University of St Andrews,
Fife, Scotland,
{iain,ipg}@dcs.st-and.ac.uk

Abstract. Research into symmetries in CSPs has shown how we can avoid redundant search to reduce the run-times of CSP solving. However, symmetries affect more than just search. For any information gathered about a CSP, the same is true of its symmetric equivalent. We should be able to use this fact to avoid more than just redundant search but redundant work. This paper proposes ways in which we can use symmetries to improve propagation techniques resulting in a modified version of the AC-2001 algorithm with empirical results.

1 Introduction

Symmetries occur in many problems where identical or indistinguishable objects take place. Whereas they may improve the structure of the problem making them easier to deal with in some sense, they manifest in CSPs increasing the amount of effort needed to solve. This is especially true when trying to find all solutions or an optimised solution.

Much of the previous work into symmetry in CSPs generally tries to lead the search routine away from redundant search (and duplicate solutions) by adding constraints to the problem at some point.

Some of these techniques include enforcing a lexicographic ordering [1] i.e. posting *lex* constraints, adding constraints dynamically during search [2,3] e.g. symmetry breaking during search - SBDS, or backtracking from the current node when it can be shown to be equivalent to a previous nogood [4,5] e.g. symmetry breaking via dominance detection - SBDD.

The above techniques (and most of the other symmetry breaking techniques not mentioned here) affect the traversal of the search tree by adding constraints that will force certain subtrees to be avoided. This was the thinking behind their creation: altering the search routine to avoid redundant search.

Constraint solving however is a balance between search and inference. There are various levels of consistency that can be maintained while searching for a solution e.g. bounds consistency, forward checking, arc consistency, path consistency. There are many algorithms for enforcing these levels of consistency, most notable are the arc consistency (AC) algorithms (e.g. AC-3 [6], AC-6 [7], AC-7 [8], AC-2000, AC-2001 [9]) that enforce AC on binary CSPs. There are also algorithms for generalised arc consistency (GAC) for non binary constraints e.g.

GAC-Schema [10]. Popular global constraints (such as *all different* [11], or *lex* constraints [12]) have specialised algorithms to enforce GAC efficiently.

While the area of inference is an important one to constraint programming, this is the first research to look at using symmetry to improve propagation algorithms. Although we cannot present changes to all the propagation algorithms that exist, at the heart of the thinking behind this research is the simple fact that any time we learn something about a CSP, the same is true of its symmetric equivalents. Thus we can try to re-use information gathered by an inference algorithm.

Since achieving arc consistency is the strongest level of consistency possible for binary CSPs, we suggest ways in which symmetry in CSPs can be used specifically to improve AC algorithms. We then present pseudo-code for a modified version of the AC-2001 algorithm developed by Bessière and Régin and give empirical results for maintaining AC on symmetric problems. Finally we discuss some of the possible directions for this exciting new research area.

2 Improving Arc consistency

There are two possible paths for improving arc consistency algorithms - exploiting symmetry to reduce run-time, or exploiting symmetry to reduce run-time and discard duplicate solutions i.e. enforce a *higher* level of consistency. If the constraint solver is using some symmetry breaking search routine, then the latter method may not be significantly effective when combined with such a symmetry breaking search routine that already discards non-unique solutions. However it may be a cheaper (more tractable) alternative to the symmetry breaking search method. We will initially look at enforcing AC and use symmetries just to reduce run-times.

Definition 1. CSP - *A CSP is a set of constraints $\mathcal{C}$ acting on a finite set of variables $\mathcal{X} : X_1..X_n$, each of which has a finite domain of possible values $D(X_i)$. A solution to a CSP L, is an instantiation of all the variables in $\mathcal{X}$ where $\forall i \; \exists j \; X_i = j, j \in D(X_i)$ such that all the constraints in $\mathcal{C}$ are satisfied.*

Definition 2. Symmetry - *Given a CSP L, with a set of constraints $\mathcal{C}$, a symmetry of L is a bijective function $f : A \rightarrow A$ where A is some representation of a state in search e.g. a list of assigned variables, a set of current domains etc., such that the following holds:*

1. *Given A, a partial or full assignment of L, if A satisfies the constraints $\mathcal{C}$, then so does $f(A)$.*
2. *Similarly, if A is a nogood, then so too is $f(A)$.*

Definition 3. Orbit - *If we have an assignment A, and a* group[1] *G representing the symmetries of a CSP, we define the* orbit *$\mathcal{O}$ of A to be a set of all the distinct assignments that can be derived by applying the elements of G to A*

[1] See [13] for a detailed explanation of group theory concepts.

i.e. $\forall A_o \in \mathcal{O}$, $\exists g \in G$ *s.t.* $g(A) = A_o$. *The definition of an orbit also extends to tuples of assignments.*

Definition 4. Stabilizer - *If we have a group* G*, we can create* H*, a* subgroup *of* G *by calculating the* stabilizer *of an assignment* A*. The subgroup* H *is the stabilizing subgroup of* A *w.r.t. to* G *if* $H \subseteq G$ *and* $\forall h \in H$, $h(A) = A$*. The definition of a stabilizing subgroup also extends to tuples of assignments.*

We assume that the constraint programmer produces a group representing the symmetries of the problem prior to search. This group can then be used by the modified AC algorithm. In order to maintain AC during search, we need to note that the symmetries of the problem change as assignments are made. Anytime a search decision is made e.g. variable $X_i = j$, or variable $X_y \neq z$, we must take the *stabilizer* of these decisions. In this paper we are taking the *pointwise* stabilizer rather than the *setwise* stabilizer of decisions. This will result in the group tending toward the identity element sooner but allows for cheaper group theory computations.

There are two main facts that we learn while performing arc consistency:

1. Inconsistent domain elements
2. Support for domain elements

Once we find an inconsistent domain value, we can remove it since it is guaranteed to violate some constraint if instantiated. The same is true of its symmetric equivalents, therefore we can remove all the elements (assignments) of the *orbit* of this domain value. This should reduce the number of constraint checks needed.

When searching for *support* for a domain value $v \in D(X_i)$, we are looking for a potential assignment that does not violate a specific constraint when instantiated alongside $X_i = v$. In order to find support, we perform many constraint checks. AC-2001 searches the domain of a variable for support lexicographically and bookmarks the first value it finds that provides support. If this value is deleted, search for a new support continues from the point in the domain *after* the previous support.

If the symmetries of a CSP act on the variables and not the assignments, the lexicographic ordering of the domains are respected by the symmetries. Thus if we find support s, for a particular assignment a, then if f is a symmetry acting on variables, $f(s)$ is support for $f(a)$. In addition, any domain element less than $f(s)$ **cannot** be support for $f(a)$. Thus we can avoid further constraint checks by re-using support.

2.1 Refining AC-2001

Algorithm 2.1, 2.2 and 2.3 contain pseudo-code based on the AC-2001 algorithm developed by Bessière and Régin. The time and space complexity for this algorithm is optimal: $\mathcal{O}(ed^2)$ and $\mathcal{O}(ed)$ respectively.

Algorithm 2.1: MAIN($\mathcal{X}$)

$Q \leftarrow \emptyset$;
for each $X_i \in \mathcal{X}$
do { **for each** X_j *such that* $C_{ij} \in \mathcal{C}$
do { $R \leftarrow$ SYMMETRICREVISE2001(X_i, X_j)
for each $X_k \in R$
do { **if** $D(X_k) = \emptyset$
then return $(false)$;
$Q \leftarrow Q \bigcup \{X_k\}$; }}}
return (SYMMETRICPROPAGATION2001(Q));

Algorithm 2.2: SYMMETRICPROPAGATION2001(Q)

while $Q \neq \emptyset$
do { `pick` X_j `from` Q;
for each X_i *such that* $C_{ij} \in \mathcal{C}$
do { $R \leftarrow$ SYMMETRICREVISE2001(X_i, X_j)
for each $X_k \in R$
do { **if** $D(X_k) = \emptyset$
then return $(false)$;
$Q \leftarrow Q \bigcup \{X_k\}$; }}}
return $(true)$;

Algorithm 2.3: SYMMETRICREVISE2001(X_i, X_j)

$CHANGE \leftarrow \emptyset$;
$G \leftarrow$ `group acting on CSP`;
for each $v_i \in D(X_i)$
do { **if** LAST$(X_i, v_i, X_j) \notin D(X_j)$
then { **if** $\exists v_j \in D(X_j) / v_j >_d$ LAST$(X_i, v_i, X_j) \bigwedge C_{ij}(v_i, v_j)$
then { LAST$(X_i, v_i, X_j) \leftarrow v_j$;
if G `acts on` **variables**
then { $\mathcal{S} \leftarrow$ ORBIT$(G, [(X_i, v_i), (X_j, v_j)])$;
for each $[(X_y, v_y), (X_z, v_z)] \in \mathcal{S}$
do LAST$(X_y, v_y, X_z) \leftarrow v_z$; }}
else { $\mathcal{O} \leftarrow$ ORBIT$(G, (X_i, v_i))$
for each $(X_k, v_k) \in \mathcal{O}$
do { `remove` v_k `from` $D(X_k)$;
$CHANGE \leftarrow CHANGE \bigcup \{X_k\}$; }}}
return $(CHANGE)$;

The main changes involve taking the orbit of inconsistent domain elements and support domain elements and re-using them. As a consequence, the Algorithm 2.3 doesn't return a boolean indicating whether or not the domain has been reduced, but rather a set of variables whose domain *have* been reduced. Notice how in Algorithm 2.3, whether we find support or not, information is re-used. For those readers not familiar with the original AC-2001 algorithm, the value $Last(X_i, v_i, X_j)$ is the last recorded support domain element $v_j \in X_j$, for the assignment $X_i = v_i$.

3 Experimental Results

For the experiments, a simple backtracking binary constraint solver was implemented[2] in Java. This solver takes instances from the model B, random binary CSP generator [14].

To give an idea of how it measures against the original implementation of the AC-2001 algorithm, the experiments from [9] were re-created (see Table 1, which records the number of constraint checks taken, the runtime and the number of deletions by the AC algorithm). As in [9], 50 instances were generated and the mean values were calculated. All experiments in this paper were run on an Athlon XP 2200 1.8GHz processor with 512Mb of RAM.

	Original AC-2001		Java AC-2001		
	#ccks	time	#ccks	AC del.	time
<150, 50, 500, 1250>	100,010	0.05	99,968	0	1.38
<150, 50, 500, 2350>	487,029	0.16	478,062	3,224	7.78
<150, 50, 500, 2296>	688,606	0.34	677,886	3,038	11.32
<50, 50, 1225, 2188>	1,147,084	0.61	1,114,781	1,255	18.05

Table 1. Results of comparing the original implementation by Bessière and Régin with the new Java implementation.

The main problem for the applicability of using propagation in AC algorithms is that the most symmetric problems that interest the symmetry in constraint programming community contain n-ary constraints. Such constraints cannot be dealt with by a binary AC algorithm such as is presented here.

The ideal problem for these experiments is a highly symmetric problem with a direct binary CSP model where the symmetries act on variables as this would allow us to re-use support. Finding *latin squares* is such a problem. A latin square is an $n \times n$ grid of numbers from 1 to n such that each number can only appear once in each row and column. In this problem, we can freely permute the rows and columns as well as inverting around a diagonal thus giving a total of $2n!^2$ symmetries.

[2] Thanks to Christian Bessière for helping to verify its correctness.

	AC-2001		Modified AC		
n	#con. checks	runtime	size of group	#con. checks	runtime
15	100,800	0.29	3.4×10^{24}	16	0.33
16	130,560	0.35	8.8×10^{26}	17	0.45
17	166,464	0.44	2.5×10^{29}	18	0.56
18	209,304	0.64	8.2×10^{31}	19	0.66
19	259,920	0.79	2.6×10^{34}	20	0.83
20	319,200	0.94	1.2×10^{37}	21	1.02
21	388,080	1.13	5.2×10^{39}	22	1.27
22	467,544	1.59	2.5×10^{42}	23	1.52
23	558,624	1.87	1.3×10^{45}	24	1.89
24	662,400	2.10	7.7×10^{47}	25	2.29
25	780,000	2.78	4.8×10^{50}	26	2.82
26	912,600	3.22	3.3×10^{53}	27	3.60

Table 2. AC on uninstantiated latin squares. The predicted number of constraint checks is produced experimentally.

	AC-2001				Modified AC			
n	fails	#ccks	AC del.	time	fails	#ccks	AC del.	time
3	0	623	5	0.02	0	324	5	0.07
4	0	3,371	9	0.06	0	1,550	9	0.13
5	4	13,432	24	0.07	5	5,743	23	0.17
6	8	41,003	40	0.13	8	14,696	40	0.29
7	55	140,454	110	0.38	55	54,141	110	0.86
8	0	198,073	63	0.62	0	54,128	63	1.28
9	95	601,669	309	2.75	101	203,451	303	4.65
10	408	2,097,243	734	12.11	409	720,123	733	16.93
11	1,277	6,785,424	3,602	48.45	1,290	2,723,297	3,607	64.49
12	5,208	49,502,231	8,654	255.21	5,208	10,412,996	8,654	348.03
13	38,209	416,371,008	72,967	2465.15	38,232	100,507,570	72,942	3315.85

Table 3. Maintaining AC while searching for a solution. Though the number of fails is sometimes slightly different, the solutions found were identical.

The results for enforcing AC on an uninstantiated instance of a latin square problem are presented in Table 2. Since the problem is uninstantiated (unlike when we are searching for a solution), it is trivial to calculate the size of the orbit before computing the orbit itself. This allows us to implement the orbit finding algorithm very efficiently. The latin squares problem is underconstrained and as such no domain removals are made. Ensuring the problem is arc consistent means just finding support for each assignment. For each variable (of which there are n^2), there are $2(n-1)$ arcs i.e. variables they are constrained with. For each arc, $n+1$ checks are required to find support for all domain elements. Thus you can see how for an $n \times n$ latin square, enforcing arc consistency takes $2n^2(n^2-1)$ constraint checks. However, for the modified algorithm, once it has been shown that one arc is arc consistent, we can infer via the symmetry of the problem that *all* arcs are consistent. So for an $n \times n$ latin square, enforcing arc consistency takes $n+1$ constraint checks. The results for maintaining arc consistency while searching for a solution (MAC) are shown in Table 3.

Disappointingly, the runtimes have not improved by re-using information. This is because the runtime of the algorithm for finding the orbit of a tuples of points of size two, outweighs the benefit of a reduced number of constraint checks. The "big-oh" complexity of an efficient orbit finding algorithm is $\mathcal{O}(|orbit| \times g)$ where g is the number of generators of the group. In retrospect, it is hard to improve an algorithm that has a low quadratic complexity.

Though the runtimes are not promising, a more detailed look at the complexity of this algorithm would be interesting to show whether or not it could be worth using other cases. It is hoped that more constrained problems or problems with more expensive constraint checks would be improved with inference algorithms that take symmetry into account.

4 Conclusions and Future Work

In this paper we proposed ways in which symmetries in CSPs can be used to make the most of gathered information. We presented a modified version of the AC-2001 algorithm which was shown to drastically reduce the number of constraint checks needed to enforce AC on a highly symmetric problem.

This is a first step into a research area with huge potential. Though the runtimes were disappointing, the large reduction in the number of constraint checks demands further research (especially into symmetric problems with expensive constraint checks). There are many paths this research could take from here, most notably:

- Information re-use in specialised constraint propagation algorithms e.g. sum, all different.
- Higher levels of consistency that can remove duplicate solutions.
- Concise representations for constraints produced by enforcing k-consistency. This could reduce the expensive time *and* space complexity of such algorithms.

– Modifying support re-use for symmetries on variables *and* values.

Whatever the outcome of this paper, we hope we have convinced the reader that the effect of symmetries in combinatorial search problems stretches further than just search and we can use this fact to avoid redundant work wherever it occurs.

Acknowledgements

We would like to thank all the members of the APES research group, notably Patrick Prosser and Ian Miguel for showing enthusiasm in this research and also to Christian Bessière, Steven Prestwich and the reviewers. This work is partly supported by EPSRC grant GR/R29666, by a Royal Society of Edinburgh SEELLD Support Fellowship and by an EPSRC PhD studentship.

References

1. Pierre Flener, Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In P. van Hentenryck, editor, *Principles and Practice of Constraint Programming*, pages 462–476. Springer-Verlag, 2002.
2. Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In Alex Brodsky, editor, *Principles and Practice of Constraint Programming*, pages 73–87. Springer-Verlag, 1999.
3. Ian Gent and Barbara Smith. Symmetry breaking in constraint programming. In W. Horn, editor, *Proceedings of ECAI-2000*, pages 599–603. IOS Press, 2000.
4. Torsten Fahle, Stefan Schamberger, and Meinolf Sellman. Symmetry breaking. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP2001*, pages 93–107. Springer-Verlag, 2001.
5. Filippo Focacci and Michaela Milano. Global cut framework for removing symmetries. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP2001*, pages 77–92. Springer-Verlag, 2001.
6. Alan K. Macworth. Consistency in networks of relations. In *Artificial Intelligence, 8*, pages 99–118. 1977.
7. Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *AAAI-93: Eleventh National conference on Artificial Intelligence*, pages 108–113, Washington, DC, 1993.
8. Christian Bessière, Eugene Freuder, and Jean-Charles Régin. Using inference to reduce arc-consistency computation. In *Fourteenth International Joint Conference of Artificial Intelligence*, pages 592–598, Montreal, Canada, 1995.
9. Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *International Joint Conference of Artificial Intelligence*, pages 309–315, Seattle, WA, 2001.
10. Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: preliminary results. In *International Joint Conference of Artificial Intelligence*, pages 398–404, Nagoya, Japan, 1997.

11. Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI-94: Twelfth National conference on Artificial Intelligence*, pages 362–367, Seattle, WA, 1994.
12. Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Global constraints for lexicographic orderings. In P. van Hentenryck, editor, *Principles and Practice of Constraint Programming*, pages 93–108. Springer-Verlag, 2002.
13. Gregory Butler. *Fundamental Algorithms for Permutation Groups*. Springer-Verlag, 1991.
14. Daniel Frost, Christian Bessière, Rina Dechter, and Jean-Charles Régin. *Random Uniform CSP Generator*, 1996. Available from http://www.lirmm.fr/~bessiere/generator.html.