A Symmetry Breaking Constraint for Indistinguishable Values *

Ian P. Gent**

School of Computer Science, University of St. Andrews, Fife KY16 9SS, United Kingdom. ipg@dcs.st-and.ac.uk

Abstract. It is very common in constraint programming that the values a set of variables can take are *indistinguishable*: for example in graph colouring the names of colours can be interchanged freely. Breaking such symmetry is important for efficient search. In this paper I present a constraint to break this kind of symmetry. I present two variants, the second of which is remarkably elegant but less generally useful. I prove that both constraints have the intended theoretical property of uniquely forcing assignments and thereby breaking all symmetry. These are the first published constraints I am aware of for this task. I outline some theoretical and practical questions that are raised.

1 Introduction

Very often in constraint programming, we have to break some symmetry inherent in the problem we are modelling. If we don't then we get a multitude of solutions, and search many equivalent non-solutions. A very common condition is that of *indistinguishable values*, where the values a set of variables can take doesn't matter [2].

Typical examples are graph-colouring and bin packing. In graph colouring, we might have to label each edge of a graph with a colour, with the actual colours being irrelevant. If there are k colours, there are $k!$ different solutions for each essentially different solution, with the colours being permuted. In bin packing, we might have to put each number into one of k bins. If all bins have the same capacity, there are again $k!$ different versions of each solution, this time the bins being permuted.

Sometimes it is quite easy to see how to break symmetry. Other times, it is not quite so obvious. For example, suppose we are solving number partitioning, which is essentially bin packing with only two bins. An easy way to break symmetry is to insist that the first number must go into the first bin. This does break the key symmetry, but it is not so obvious how to apply this idea when we have 3 or

* The author thanks past and present members of the APES research group, of which he is a member, for helpful discussions: www.dcs.st-and.ac.uk/~apes. He also thanks reviewers for encouraging me to do the proofs.

** I dedicate this paper to my wife, daughter, and the citizens of the USA — *September 12, 2001.*

more bins. If we insist the first number goes into the first bin, we can't demand that the second must go into the second bin, as the second number might also have to go into the first. While we can insist that the i^{th} number cannot go into a bin larger than number i, this leaves number 3 with two symmetrical possibilities if numbers 1 and 2 both go into bin 1: it can go into either bin 2 or bin 3. This difficulty disallows the normal solution used of imposing monotonicity constraints when we have indistinguishable *variables.*

In this paper I present a solution to this problem. One solution in a language like Ilog Solver would be to write demons to fire whenever bin assignments were made, ensuring that symmetry was not broken. Alternatively, constraints can be added during search for any set of symmetries, including those arising from indistinguishable values [1, 4]. However, my solution is extensional, i.e. works through addition of constraints, rather than the definition of code to run when certain events happen or constraints added dynamically during search.

2 A general symmetry breaking constraint for indistinguishable values

Suppose that we have a set of numbered variables I[1] , a set of numbered values V, and an assignment of values to objects, $A : I \to V$. Presumably the assignment A is constrained in other ways by the rest of the constraint program, for example in colouring a difference constraint would be imposed on $A(i)$ and $A(j)$ if i and j represented adjacent nodes.

I will assume that the set of values is indistinguishable, just as the set of colours in a graph colouring problem is, or a set of bins with identical capacities is in a bin packing problem. So we wish to break the symmetry by demanding that only one assignment A is considered for each set of symmetrically equivalent assignments. I will do this by using the ordering inherent in the numbering of I and V. Typically I will be integers from 0 to $n-1$ and V integers from 0 to $k-1$. I will assume for simplicity that this is the case.[2]

We work on equivalence classes of variables taking the same value. Each class has a least numbered variable, i.e. a least member of I. We can take advantage of this arbitrary ordering to break symmetry. I first introduce a new function R, for "representative." For any $i \in I$, $R(i)$ will be the least numbered variable with the same value as variable i under the assignment A. We now have a unique label for each equivalence class, so we can use this to impose monotonicity constraints. Specifically, the values will be assigned in order of representative: the class with least numbered representative will be given value 0; the class with the next least numbered representative will be given value 1; and so on.

[1] You can think of I as the indices of variables, for example in Ilog Solver the indices in an array of variables.

[2] It is also necessary to assume this in my Ilog Solver implementation as when I did not assume this, I came across what looks like an obscure bug in Solver 4.3.

To make this concrete, I introduce a further new assignment M. This represents the *maximum* value of all variables from 0 to i. The following constraints tie in A, M, and R, and break the symmetry on A.

$$\begin{aligned}
A(i) &= A(R(i)) && (1)\\
R(i) &= R(R(i)) && (2)\\
R(i) &\leq i && (3)\\
M(0) &= A(0) \quad = \quad R(0) \quad = \quad 0 && (4)\\
M(i) &= \max(M(i-1), A(i)) \qquad [i \neq 0] && (5)\\
R(i) = i \Leftrightarrow A(i) &= M(i-1)+1 \qquad [i \neq 0] && (6)\\
R(i) < i \Leftrightarrow A(i) &\leq M(i-1) \qquad [i \neq 0] && (7)\\
A(R(i)) &= M(R(i)) && (8)
\end{aligned}$$

1. For any $i \in I$, the value of its representative is the same as its value.
2. The representative of any $i \in I$ is its own representative. For example, if element 2 is element 7's representative, the representative of element 2 must itself be 2.
3. For any $i \in I$, its representative is lower or equally numbered than itself.
4. Since I am assuming that values and indices start from 0, the representative of 0 must be 0, and indeed this is redundant as it is implied by (3). Further, variable 0 must take the first value, 0, and thus the maximum of values up to index 0 must be 0.
5. The constraint (5) simply makes each $M(i)$ the maximum of all $A(j)$ for $j \leq i$.
6. The constraint (6) says that where we need a new value for $A(i)$, then i must be its own representative: no $j < i$ can have $A(j) = A(i)$ because $A(i) > M(i-1) \geq M(j) \geq A(j)$.

The final two constraints are redundant. I include them as they might help understanding, or in practice might make implementation more efficient through better propagation. The proof below does not use the last two constraints.

7. The constraint (7) means that if $R(i) < i$, then we cannot use a new value for $A(i)$, and propagation should ensure that $M(i) = M(i-1)$. It is redundant because constraint (6) is a double implication, and if $R(i) \neq i$ then $R(i) < i$ from (2), and because if $A(i) \neq M(i-1)+1$ then $A(i) \leq M(i-1)$ from (5).
8. The final constraint states that the value of $R(i)$ must always be a point where the counter M 'ticks'. We can't have that $A(R(i)) < M(R(i))$ because that means that $A(R(i))$ takes the same value as some $A(j)$ for $j < i$, and we do not want this to happen. The constraint is also redundant. We know that $R(R(i)) = R(i)$ from (2). So the index $R(i)$ satisfies the left hand side of (6), so we have that $A(R(i)) = M(R(i)-1)+1$. So from (5) we have that $A(R(i)) = M(R(i))$ if $i \neq 0$, and this is trivially true from (4) otherwise.

Taken together the above constraints break all symmetry based on indistinguishable values. This statement can be formalised into a theorem, depending on the assumption that we have determined all

Theorem 1 *If the above set of constraints are added to a consistent problem in which equivalence classes of values of A are given by statements $A(i) = A(j)$ and $A(i) \neq A(j)$, then there are unique satisfying values of A, R, and M. For each i, $R(i)$ is the least numbered variable with the same value as i and $M(i)$ is the maximum value taken by variables $0 \dots i$.*

Proof. We work by induction on i, assuming that for $k \leq i$, $A(k)$ is uniquely determined and the claimed properties for R and M.

For the induction base, constraint (4) trivially gives the values $M(0) = A(0) = R(0) = 0$ satisfying all the properties claimed

We now consider the variable $i+1$ for the induction step. There are two cases, depending on whether $A(i+1)$ is equal to some earlier value.

If no value $k \leq i$ has $A(k) = A(i+1)$ then $R(i+1) = i+1$ from (1) and (3). Clearly the claimed property of R holds for $i+1$. From (6), we have that $A(i+1) = M(i)+1$. By the induction hypothesis, $M(i)$ is the maximum value of $A(k)$ for $k \leq i$, so in particular $A(i+1)$ is uniquely determined. Constraint (5) gives that $M(i+1) = A(i+1)$, establishing that $M(i+1)$ is the claimed maximum value.

If we have that $A(i+1) = A(k)$ for some $k \leq i$, we first note that $A(k)$ is uniquely determined and thus that $A(i+1)$ is set to the same value. The induction hypothesis gives $M(i) \geq A(k) = A(i+1)$, so constraint (5) then gives $M(i+1) = M(i)$, establishing the claimed property of M. Since the right hand side of (6) is false, we have that $R(i+1) \neq i+1$, and so (3) gives $R(i+1) < i+1$. The only consistent values k for $R(i+1)$ must satisfy $A(i+1) = A(k)$ from (1). Consider any k which is not the least numbered variable with the same value. We have from induction that $R(k) < k$. But then k cannot be the value of $R(i+1)$ as it would not satisfy constraint (2). The only remaining value of $R(i+1)$ is the least index j taking the same value as $A(i+1)$, completing the induction step.

I believe that the new constraint satisfies one desirable property. That is, all values of A, M, and R can be determined by a constraint solver by propagation alone without any backtracking, once the equivalence classes of variables taking the same values have been determined. I was careful in the above presentation to ensure that every deduction was either on simple implication given values of variables, or on removal of values not satisfying constraints. Both these steps are taken by standard arc consistency algorithms. The inductive nature of the proof would be mirrored by propagation iteratively setting the values for $i = 0, 1, 2$, etc. However, to prove my belief formally we would have to consider the way that array constraints such as $A(R(i)) = A(i)$ are dealt with in a given solver (or formal abstraction of a constraint solver) and I'm not currently sure how this is done.[3] So it remains possible that a given solver might need to backtrack if it does not make all the immediate deductions I have assumed in the above proof.

[3] Pointers appreciated

3 Implementation in Ilog Solver

The key feature of my constraints are the use of function values as indices, as for example the use of $R(i)$ in $A(R(i)) = A(i)$. The constraint solving language Ilog Solver allows this through the use of the *array* constraint [5]. I have written a Ilog Solver (Version 4.3) constraint IanSymmetry to encapsulate these constraints. The result to the user is that symmetry breaking can be achieved by a one line expression of a single constraint. With M, R and A as arrays, Solver has defined array indexing in a general way so that we can write `A[R[i]] == A[i]` for example, even though we don't know the value of `R[i]` With this syntax, the above only needs $O(n+k)$ constraints, though I don't know if the array index constraints use more space internally.

As a trivial example of its use, in a number partitioning problem with 5 numbers to go into two bins, the declaration `IlcIntVarArray A(m,5,0,1);` declares the array `A` to be the assignment of numbers to bins. Symmetry on this is broken by the one liner

```
IanSymmetry(m,A);
```

The argument `m` is Ilog's search manager, and I wrote IanSymmetry to post the symmetry breaking constraints to this manager when it is created.

To date, I have not programmed more than toy examples using this constraint, so cannot demonstrate its practical utility or otherwise. [4]

4 A special case

The constraint I have proposed may not seem particularly elegant. In a special case, everything becomes much more beautiful.

Suppose that we don't need to make the set of values V be the integers $0 \dots k-1$. In fact, suppose that we can identify the sets V and I. That is, the possible values for integers $i \in I$ is the set I itself. Assuming that two different integers i and j take the same value, of course there will be gaps in the set of used values, but that need not be a problem. Remember that we are dealing with situations where the actual values are arbitrary, and are just names of colours or bins. In practical terms, we may not care at all about the values V, but perhaps just want to know how many are set. For example, in a bin packing problem we might just want to know how many bins are used, i.e. the number of different values $A(i)$. Such information can be derived in a constraint language from the values of A: the number of different values can be expressed as the number of indices i satisfying $A(i) = i$.

In this situation, the somewhat confusing set of constraints in the general case collapses elegantly. The key is that we can identify the two functions A, and R, making them one and the same function. If we do not care that the values should

[4] I did break some symmetry in matrix models this way, but I omit it as it was less interesting than the other work on matrix models at to this workshop [3, 6].

be consecutive, we discard the function M and its associated constraints. We are left only with the constraints

$$A(A(i)) = A(i)$$
$$A(i) \leq i$$

These extremely simple and elegant constraints do indeed break all symmetry, where any values may be used as long as we preserve equivalence classes. However, we still want to know that the constraints do break the symmetry correctly: there is much less to prove so the basic correctness result is much simpler. For example, there is no need to prove that variables with the same value have the same representative: since values and representatives are the same thing, there is nothing to prove.

Theorem 2 *If the above set of constraints are added to a consistent problem in which equivalence classes of values of A are given by statements $A(i) = A(j)$ and $A(i) \neq A(j)$, then there are unique satisfying values of A. For each i, the value $A(i)$ is the least numbered variable with the same value as i.*

Proof. We work by induction on i, assuming that for $k \leq i$, $A(k)$ is uniquely determined with the claimed property.

We now consider the variable $i+1$ for the induction step. If no value $j \leq i$ has $A(j) = A(i+1)$ then the first constraint disallows any value $j \leq i$ for $A(i+1)$ – otherwise we would have $A(j) = A(A(i+1)) = A(i+1)$. The second constraint forces $A(i+1)$ to be the only remaining value, $i+1$.

If we have that $A(i+1) = A(j)$ for some $j \leq i$, then by the induction hypothesis the value $A(j)$ is the least numbered variable with the same value as $A(j)$, and this is also the least numbered variable with the same value as $A(i+1)$, as required.

Again I have implemented this as a special purpose constraint `IanSymmetrySelf`, the Self indicating that domain of the array is the array itself.

4.1 An unexpected application

In many cases one can get rid of symmetry just by insisting that the values of the relevant variables are monotonic. For example, in chapter 24 of the User Manual [5], Ilog present a solution to the n-queens problem. They start off assuming that the variables will be pairs of x-y locations for the queens. This has symmetry in (say) the y locations, as these will turn out just to be permutations of $0 \ldots n-1$. Ilog then say that they will impose an ordering constraint $y_0 < y_1 < \ldots < y_{n-1}$. But actually, that is not what they do in the implementation: they take the next step and do not even bother to construct the y variables at all.

It occurred to me to try to apply my new constraint to this situation. Since we have identified a symmetry, in the y variables, it would be nice if we could just express this, rather than have to work out the reformulation which gets rid of the symmetry. While the reformulation may not be too hard to see in this case, the general issue of reformulating to avoid symmetry is very difficult.

This is an interesting case for my constraint. First, it is natural to apply the special case, because there are 100 y variables which take 100 possible values. Second, my constraint was not designed with any application such as this in mind. My constraint was designed for cases where it is not so obvious how to avoid the symmetry. In this case we can easily avoid the symmetry by posting constraints $y(i-1) < y(i)$.

I coded up the special case in Solver, via the class IanSymmetrySelf. I took Ilog's nqueens code from the distribution, and changed it only by adding the y variables and expressing the symmetry and all different constraints

```
 IlcIntVarArray y(m, nqueen, 0, nqueen-1);
 IanSymmetrySelf breaksymmetry(m,y);
 m.add(IlcAllDiff(y));
```

and by changing the lines which create variables representing diagonals

```
for (i = 0; i < nqueen; i++) {
    x1[i] = x[i]+i;
    x2[i] = x[i]-i;}
```

to allow for the fact that we don't know the value of `y[i]`

```
for (i = 0; i < nqueen; i++) {
    x1[i] = x[i]+y[i];
    x2[i] = x[i]-y[i];}
```

I tested this out and for $n = 100$ it took roughly twice as long to solve the problem with one more fail, finding a different solution. The reason it works so well is that it quickly works out that actually $y(i) = i$ for all i. That's because $y(0) \leq 0$ so $y(0) = 0$; and $y(1) \leq 1$ and $y(1) \neq 0$, so $y(1) = 1$, and so on. [5] Given that twice as long was only 0.44 seconds, that's not bad. I think avoiding finding a problem reformulation is well worth 0.24 cpu seconds.

What I thought was a general symmetry constraint works well in a case I did not design it for. It's always nice when something works outside the scope you originally planned it for. My symmetry constraint does this.

5 Discussion & Further Work

While my solution is general for indistinguishable values, it may have some drawbacks. First, it only captures one kind of symmetry, while others may be present in the problem: for example if at some later stage two bins both have the same reduced capacity, they have an undesirable symmetry which this constraint does not address. Second, I do not yet how this constraint propagates. It may be that propagation takes place too late: it may be that the symmetry constraints

[5] Given that, I don't understand why it found a different solution.

don't fail until after considerable search has been done in an invalid part of the search space. There are subtle interactions between search strategy and symmetry propagation. For example, if our search carefully constructed bin 3, it might have to entirely reconstruct it if symmetry demanded it had to be bin 2 instead. The third drawback is that it might well be less efficient than a special purpose constraint wired into solver by demons.

Despite these potential drawbacks, I feel the new constraint is valuable in providing a simple way for breaking symmetry in the common case of indistinguishable values. It is particularly easy if somebody has implemented it in the constraint language that you have used.

There are a number of directions for future research. I would be happy to discuss these with anyone wishing to help.

First, there is the theoretical validation of the constraints I have introduced. I have established basic correctness, but it remains to be shown whether the values of the associated arrays can be established only through propagation. It is also interesting to establish the interaction of these constraints with the search process, since they presuppose a unique preferred assignment.

Second, empirical work is needed to establish the practical usefulness of the constraints. It is possible that they meet theoretical desiderata while being impractical for one reason or another.

Finally, I suggest that one of the main issues facing this and many other pieces of work on symmetry is the interaction between symmetries. Constraint problems often have much symmetry in several different parts of the problem. These interact in complicated ways. The natural way to address one set of symmetries might be through a symmetry breaking constraint; a second might be addressed through reformulating part of a problem; and a third set of symmetries might be addressed by a technique such as symmetry breaking during search. To ensure that such techniques can be combined correctly in theory and effectively in practice is one of the big questions facing those researching symmetry breaking in constraint programming.

References

1. R. Backofen and S. Will. Excluding symmetries in constraint-based search. In *Proceedings, CP-99*. Springer, 1999. LNCS 1713.
2. B. Benhamou. Study of symmetry in Constraint Satisfaction Problems. In *Proceedings PPCP'94*, pages 246–254, May 1994.
3. P. Fleiner, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Symmetry in matrix models. Technical Report APES-30-2001, APES group, 2001. Available from http://www.dcs.st-and.ac.uk/ apes/reports/apes-30-2001.ps.gz. Submitted to SymCon'01 (Symmetry in Constraints), CP2001 post-conference workshop.
4. I.P. Gent and B.M. Smith. Symmetry breaking in constraint programming. In W. Horn, editor, *Proceedings of ECAI-2000*, pages 599–603. IOS Press, 2000.
5. ILOG S.A. *ILOG Solver 4.3 User's Manual*. ILOG, 1998.
6. B.M. Smith and I.P. Gent. Reducing symmetry in matrix models: Sbds v. constraints. Technical Report APES-31-2001, APES Research Group, September 2001. Available from http://www.dcs.st-and.ac.uk/~apes/apesreports.html.