Symmetry Breaking by Nonstationary Optimisation

S. Prestwich, B. Hnich, R. Rossi, S. A. Tarim

4C/UCC, Cork
Izmir University of Economics, Turkey
Hacettepe University, Turkey

(AICS'08)

introduction

many CSPs contain *symmetries*: transformations of solutions that yield other solutions

eg N-queens has 8 symmetries: each solution may be rotated through 90 degrees and reflected to obtain other solutions

other problems may have many symmetries, eg Balanced Incomplete Block Designs (BIBDs): *an arrangement of v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks*

or more simply...

BIBD: a binary matrix with v rows, b columns, r ones per row, k ones per column, and scalar product λ between any pair of distinct rows

instances are specified by parameters (v, b, r, k, λ), eg (6,10,5,3,2):

```
1011100001
0011011010
1101000110
0000101111
0110010101
1100111000
```

very challenging problem with quite small open instances, eg (22,33,12,8,4)

the hardness is partly due to the many symmetries: given any solution, any two rows or columns may be exchanged to obtain another solution

the symmetry group is the direct product $S_v \times S_b$ so there are $v!b!$ symmetries, eg (9,120,40,3,6) has more than 10^{200} symmetries

symmetry breaking

symmetry implies that search effort is being wasted by exploring equivalent regions of the search space more than once

by *symmetry breaking* we may speed up search significantly

symmetries form groups, and there are close connections between symmetry breaking and computational group theory

several distinct methods have been reported for symmetry breaking in CSPs...

symmetry breaking methods

reformulating a problem to eliminate symmetries is excellent when possible, but often difficult or impossible

adding constraints is probably the most common method

all symmetries can in principle be broken by this method, which was developed into the **lex-leader** method

too many constraints might be needed, but a subset can be used for *partial symmetry breaking*, eg for BIBDs we might break row and column symmetries (**double-lex**) but not combined row-column symmetries a drawback of adding symmetry breaking constraints: adding constraints *does not respect the search heuristics*

that is, the excluded solutions might be the easiest to find under the variable/value ordering

but *dynamic* symmetry breaking methods have been devised that *do* respect search heuristics...

SBDS adds constraints during search so that, after backtracking from a decision, future symmetrically equivalent decisions are disallowed

can be implemented by combining a constraint solver with the GAP system (GAP-SBDS) which allows symmetries to be specified compactly via group generators

it can handle *billions* of symmetries

STAB is a related method that only adds constraints that do not affect the current partial variable assignment

does not break all symmetries but has given good results on problems with up to 10^{91} symmetries: even more scalable than SBDS

SBDD detects when the current search state is symmetrical to a previously-explored "dominating" state, thus respecting search heuristics: *dominance detection*

no need to compare the current search state with *all* previous states: only those corresponding to fully-explored subtrees (*nogoods*) — the number is at worst linear in the number of variables

GAP-SBDD exists, but better results are found by treating dominance detection (which is NP-hard: *subgraph isomorphism*) as an auxiliary CSP and solved by CP methods [Puget 2005]

SBDD solves the space problem (it doesn't add any symmetry breaking constraints) and is the most scalable method

our method

we describe and test a new approach to partial symmetry breaking

related to SBDD but uses a different dominance detection technique, expressed as a non-stationary optimisation problem and solved by metaheuristics

this opens up symmetry breaking to metaheuristic algorithms, which often scale better than backtrackers

it has lower time and memory requirements than SBDD and, unlike other partial symmetry breaking methods, the symmetries it fails to break are *likely to be those with little effect on runtime*

a new dominance test

we use a different dominance test than SBDD:

if we can apply a group element $g \in G$ to the current partial assignment A s.t. $A^g \prec_{\text{lex}} A$, then (under some assumptions such as DFS) A^g dominates A in the SBDD sense and we can backtrack from A

(proof in paper relies on DFS and static value ordering)

dominance as optimisation

we can express the dominance test as an optimisation problem, suitable for solution by local search instead of CP methods

the problem at each search node A is to find a $g \in G$ such that $A^g \prec_{\text{lex}} A$

we can treat G as a local search space

to define a local search algorithm we need several things...

search states: each $g \in G$ is a search state

neighbourhood structure: choose some $H \subset G$, then from any state g the local moves are the elements of H leading to neighbouring states $g \circ H$

(so all G elements are local search states, and some of them (*H*) are also local moves)

objective function: the value of a state g is the lex-ranking of A^g (which can be considered as a number) then we can apply **hill climbing:** from each state g try to find a local move h that reduces the objective function ($A^{g \circ h} \prec_{\mathsf{lex}} A^g$)

if a series of moves $(h_1, h_2, \ldots)$ reduces the lex-ranking sufficiently then we hope to find $A^{g \circ h_1 \circ h_2 \circ \cdots} \prec_{\mathsf{lex}} A$, establishing dominance

this has even smaller memory requirement than SBDD, as we need store only the current group element g

it's easy to show that *if H is a generator set for G ($\langle H\rangle = G$) then the search space is connected*

using a generator also gives small neighbourhoods: any group G has a generator of size $\log_2(|G|)$ or smaller

but we can also use a non-generator H and allow some random moves from $G \setminus H$: in fact we do this, for purely heuristic reasons

dominance as nonstationary optimisation

how much effort should we devote to solving these dominance detection problems?

if local search fails to find a dominating state, this might be because there is no such state...

...but it could also be because the algorithm has not searched hard enough

too little search might miss important symmetries, while too much will slow down DFS

this is a drawback of using an incomplete approach such as local search our answer is to devote very little effort indeed at each search node: we apply only *one* local move $h \in H$ per search tree node

local search is now being used to solve an optimisation problem whose objective function *changes in time*: as DFS changes variable assignments A, the objective value of any given g changes because it depends on A^g

this is called *nonstationary optimisation* so we call our method *Symmetry Breaking by Nonstationary Optimisation* (SBNO) if a dominance is not detected by local search then it might detect it after a few extra local moves and search tree nodes

DFS can then backtrack, possibly jumping many levels in the search tree

a nice feature:

- a symmetry that would only save a small amount of DFS effort is unlikely to be detected by SBNO, because DFS might backtrack past A before an appropriate g can be discovered

- one that would save a great deal of DFS effort has a great deal of time in which to be detected by local search so we hope that SBNO will detect and break all *important* symmetries: those that make a significant difference to the size of the search tree and hence the execution time

this is unlike partial symmetry breaking methods such as double-lex and STAB, which choose symmetries to break for space reasons

experiments

we test SBNO on BIBDs and compare with published results for other methods

we use the most direct CSP model for BIBDs: represent each matrix element by a binary variable, add 3 types of constraint:

(i) v b-ary constraints for the r ones per row

(ii) b v-ary constraints for the k ones per column

(iii) $v(v-1)/2$ $2b$-ary constraints for the λ matching ones in each pair of rows

a simple BIBD solver

we implemented a simple BIBD solver: DFS with static variable ordering ordered by rows then columns, and a static value ordering trying 1 then 0

no constraint propagation at all is used in this prototype: at each search node we simply check that no constraint has been violated

no constraint programmer would use such a feeble algorithm!

but this is a prototype SBNO implementation

SBNO implementation

local search states: elements of $G = S_v \times S_b$

local moves: elements h of the group generator H consisting of arbitrary row or column swaps, *restricted to the subset of swaps involving the matrix entry corresponding to the binary variable v at which the last $\prec_{\text{lex}}$ test failed*

in experiments this gave better performance than a more obvious use of the smaller generator of *adjacent* (or first-last) row/column swaps time complexity:

- the restriction makes the neighbourhood sizes either v or b depending on whether we swap a row or a column

- time to compare rows and columns takes $O(b)$ or $O(v)$ time respectively

- therefore the time to find an improving move if one exists is $O(vb)$ (linear in # variables)

this heuristic was also inspired by *conflict-directed* heuristics used in many successful local search algorithms — these focus the search effort on the source of failure

more heuristics

compensate for incompleteness by randomising g at each local move with probability $1/vb$

from each local search state, the possible local moves h are tested in random order until finding one that satisfies $A^{g \circ h} \prec_{\text{lex}} A^g$

if there are none then randomly exchange either *v*'s row or column with the next one

TABU tenure: no improving move is allowed if it reverses a move made within the last 10 moves

we call this algorithm <u>SBNO-TABU</u>

symmetry breaking overhead

runtime profiling shows that SBNO-TABU consumes over 90% of the total execution time: this seems to contradict our claim that it is a low-overhead method!

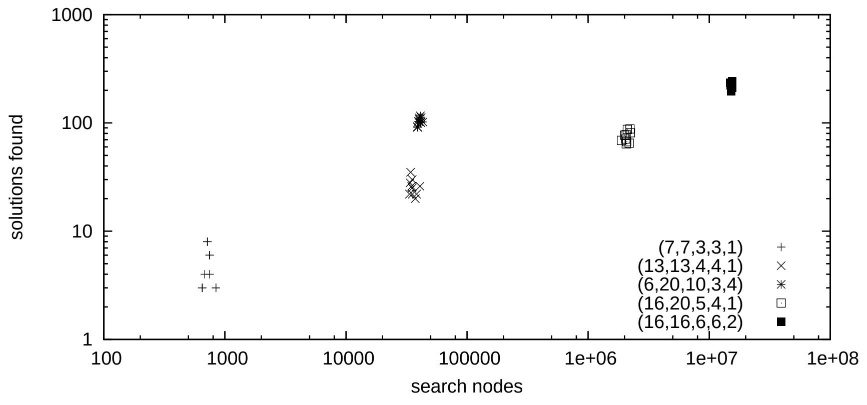
but our algorithm currently performs no constraint propagation, so the time spent at each node is very small — in fact the time complexity of our constraint checker at each search node is only $O(v)$ whereas that of SBNO-TABU is $O(vb)$

but propagation algorithms are typically *at least linear* in the number of problem variables, which is vb in this application

so we expect SBNO-TABU overhead to be **negligible** when applied to a real constraint solver

performance variation

use of local search for symmetry breaking makes the DFS runtime and number of solutions found *nondeterministic* — 10 runs of five different instances (complete tree search):



there's little variation in the # search nodes

there's more variation in # solutions found, but this reduces as the problem hardness increases

harder problems are most interesting so nondeterminism isn't serious, and we can use 1 run per instance in experiments

comparison with other methods

different researchers use different BIBD instances to test their algorithms, and we use the same instances

[Frisch, Hnich, Kiziltan, Miguel, Walsh] for 1-solution runs using global symmetry breaking constraints:

					GACLexLeq	GACLexLeq		
v	b	r	k	λ	adj pairs	all pairs	Decomp	SBNO
6	50	25	3	10	1.7	1.8	11	1.6
6	60	30	3	12	4.6	4.9	45	6.0
10	90	27	3	6	111	120	742	104
9	108	36	3	9	8.4	7.6	73	248
15	70	14	3	2	6.2	8.4	21	0.02
12	88	22	3	4	249	317	1154	1333
9	120	40	3	10	8.0	7.2	82	648
10	120	36	3	8	1316	1132	—	1227
13	104	24	3	4	398	448	1667	328

SBNO-TABU undominated by any of the other methods on these instances, and is roughly comparable in execution time to the Decomposition method double-lex [Flener, Frisch, Hnich, Kiziltan, Miguel, Pearson, Walsh] and GAP-SBDD [Gent, Harvey, Kelsey, Linton], all-solution runs:

					distinct	double-lex		GAP-SBDD	SBNO	
v	b	r	k	λ	solns	solns	time	time	solns	time
7	7	3	3	1	1	1	1.1	0.2	6	0.004
6	10	5	3	2	1	1	1.0	0.6	4	0.008
7	14	6	3	2	4	24	11	5.0	55	0.05
9	12	4	3	1	1	8	28	1.9	10	0.02
8	14	7	4	3	4	92	171	66	162	0.3
6	20	10	3	4	4	21	10	56	107	0.2
11	11	5	5	2	1			19	12	0.08
13	13	4	4	1	1			42	25	0.2
7	21	6	2	1	1			11	32	0.05
16	20	5	4	1	1			6078	67	18
13	26	6	3	1	2			59344	5694	186

SBNO-TABU faster than double-lex but breaks fewer symmetries

SBNO-TABU beats GAP-SBDD in time but does not break all symmetries

memetic symmetry breaking

SBNO can be used with other metaheuristics, and evolutionary algorithms have been successfully applied to nonstationary optimisation

we use a *memetic algorithm*: a genetic algorithm (GA) in which local search is used to improve chromosomes before insertion into the population

we use a *steady-state* GA with 3 populations of group elements, each with 1000 organisms

each organism has 2 chromosomes: a row and a column permutation why 3 populations?

- population 1: row permutation fixed to identity permutation
- population 2: column permutation fixed to identity permutation
- population 3: neither is fixed

we could use just population 3, because row/column symmetries subsume row and column symmetries, but better results were obtained by treating them separately at each DFS node we generate 1 *offspring* from 2 random parents in a random population $p \in \{1,2,3\}$: we apply *cycle crossover* separately to row & column chromosomes, then 1 *exchange mutation* to each chromosome (these are standard genetic operators for permutation problems)

we compare the new offspring to a random organism in the population: if it's fitter then replace it in the population

the organism used for dominance detection at each node is the fitter of (i) the most recent offspring and (ii) the random organism tested

to avoid stagnation: if the parents are identical then randomise one of them before applying genetic operators

GAs can be enhanced by applying local search to new chromosomes: a *memetic algorithm*, and we apply a simple form of local search similar to SBNO-TABU

we call this algorithm SBNO-MEME

comparison (1st-solution):

v	b	r	k	λ	SBNO-TABU	SBNO-MEME
6	50	25	3	10	2.1	1.0
6	60	30	3	12	7.3	1.9
10	90	27	3	6	158	16
9	108	36	3	9	416	16
15	70	14	3	2	0.02	0.02
12	88	22	3	4	1781	129
9	120	40	3	10	1007	29
10	120	36	3	8	1722	118
13	104	24	3	4	510	25

SBNO-MEME is better

on harder all-solution problems (see the paper) SBNO-MEME again beats SBNO-TABU

in fact SBNO-MEME is better on harder problems, in terms of broken symmetries *and* execution time

the memetic approach seems to be more scalable than the local search approach: evolutionary algorithms often do well on permutation problems

related work

there are few connections between metaheuristics and symmetry:

- Petcu & Faltings (2003) used a form of symmetry (*interchangeability*) to guide a distributed local search algorithm

- a negative result of Prestwich (2003), Prestwich & Roli (2005) is that some forms of symmetry breaking have a detrimental effect on local search performance

- symmetry breaking is often applied to GAs, but there it has a different meaning, and refers to clustering of the population within a symmetric region of the search space

as far as we know, SBNO is the first use of metaheuristic search to break symmetry with respect to the area of *hybrid search algorithms* SBNO-TABU is an example of an integration of local and tree search

in CP tree search there is often a trade-off between (i) performing expensive reasoning at each node to potentially eliminate large subtrees, and (ii) processing nodes cheaply to reduce overheads

when this reasoning is used to solve another NP-hard problem, incomplete reasoning can be applied in the hope of finding something useful in a short time

eg Sellmann & Harvey (2002) use local search within backtrack search to generate tight redundant constraints (*heuristic propagation*)

SBNO-TABU is another example of this type of integration, but we do not know of any similar use of evolutionary methods

conclusion

SBNO is a new partial symmetry breaking method for CP

it's related to SBDD but:

- uses different dominance detection
- solves it by resource-bounded local/evolutionary search instead of by CP or computational group theory
- smaller memory requirement
- smaller time complexity (should be almost negligible)

we found good first results even without propagation

we will soon implement SBNO in a real CP solver, and apply it to other symmetrical problems such as the Social Golfer

SBNO is very suitable for large problems with many symmetries, but other methods are better for problems with fewer symmetries