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

finite-domain CSP: variables $v_1 \ldots v_n$ with domains $\mathrm{dom}(v_i) = \{a_1, \ldots, a_m\}$ of values, plus constraints

the problem: *find an assignment of values to all variables such that no constraint is violated*

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 BIBDs...

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: 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

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

1011100001
0011011010
1101000110
0000101111
0110010101
1100111000

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

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}

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 SB and computational group theory

several distinct methods have been reported for SB in CSPs...

SB 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 SB*, eg for BIBDs we can break row and column symmetries (**double-lex**) but not row-column symmetries

adding constraints also does not respect the search heuristics, but dynamic SB 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 CGT system (GAP-SBDS) which allows symmetries to be specified compactly via group generators

it can handle billions of symmetries but no more

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

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*), 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

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

our method

we describe and test a new approach to partial SB

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

this opens up SB to metaheuristics, which often scale better than backtrack-based algorithms

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

(informal proof in paper)

dominance as optimisation

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

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

we can treat G as a LS space with each $g \in G$ being a search state

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

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

easy to show: 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 $G \setminus H$: we do this for heuristic reasons

objective function value of a state g: lex-ranking of A^g (which can be considered as a number)

LS: from each state g try to find a local move h that reduces the objective function ($A^{g \circ h} \prec_{\text{lex}} A^g$)

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

even smaller memory requirement than SBDD, as we need store only the current partial assignment A and current group element g (plus whatever data structures are needed by the underlying constraint solver)

dominance as nonstationary optimisation

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

if LS 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 LS

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

LS 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 LS 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 LS

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*

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

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

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

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

more heuristics

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

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

if none then randomly exchange either v's row or column with the next one (no justification for this heuristic, and no doubt a better one can be found)

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

these heuristics do not affect the correctness of symmetry breaking, only its efficiency

SB overhead

runtime profiling shows that SBNO consumes over 90% of the total execution time: 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

time complexity of our constraint checking algorithm at each search node is only $O(v)$ whereas that of SBNO is $O(vb)$

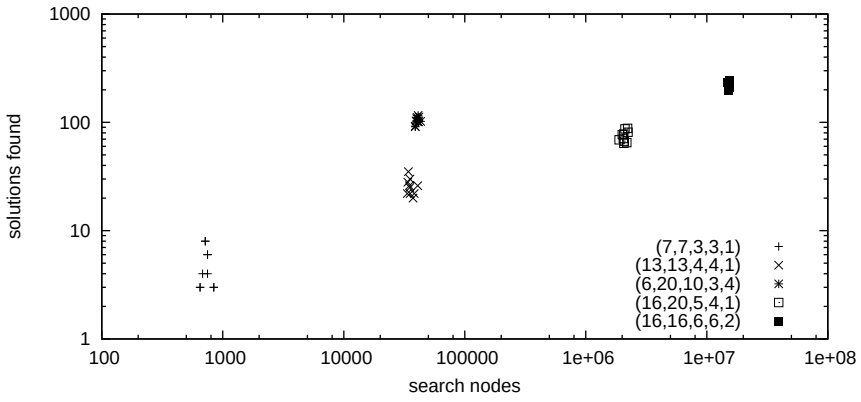
but propagation algorithms are typically at least linear in the number of problem variables, which is vb in this application

so we expect SBNO overhead to be negligible when applied to a real constraint solver (to be tested)

performance variation

use of LS for SB makes the DFS runtime and number of solutions found nondeterministic

10 runs of five different instances:



little variation in the # search nodes for complete tree search with SB

more variation in # solutions found but this reduces as the problem hardness increases

harder problems are most interesting so we can use 1 run per instance

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 SB constraints:

v	b	r	k	λ	GACLexLeq adj pairs	GACLexLeq 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 not dominated 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 faster than double-lex but breaks fewer symmetries

SBNO beats GAP-SBDD in time but does not break all symmetries, and GAP-SBDD automates some of the implementation

The best known results for many instances are those of [Puget] and we can't match them (see paper for results)

but for a trivial DFS algorithm (without propagation) it does surprisingly well!

next step: implement SBNO in a real CP system

further results: we improved the results a lot by replacing TABU with a memetic algorithm (to be presented at the CP'08 symmetry workshop)

conclusion

SBNO is a new partial symmetry breaking method for CP

related to SBDD but

- using different dominance detection
- solving it by resource-bounded LS instead of by CP or CGT
- smaller memory requirement
- very low time complexity (should be almost negligible)

good first results even without propagation

will apply it to other symmetrical problems