Artificial Intelligence 211 (2014) 1–33

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Generating custom propagators for arbitrary constraints

Ian P. Gent, Christopher Jefferson *, Steve Linton, Ian Miguel, Peter Nightingale *

School of Computer Science, University of St Andrews, St Andrews, Fife KY16 9SX, UK

ARTICLE INFO

Article history:
Received 29 November 2012
Received in revised form 27 February 2014
Accepted 2 March 2014
Available online 12 March 2014

Keywords:
Constraint programming
Constraint satisfaction problem
Propagation algorithms
Combinatorial search

ABSTRACT

Constraint Programming (CP) is a proven set of techniques for solving complex combinatorial problems from a range of disciplines. The problem is specified as a set of decision variables (with finite domains) and constraints linking the variables. Local reasoning (*propagation*) on the constraints is central to CP. Many constraints have efficient constraint-specific propagation algorithms. In this work, we generate custom propagators for constraints. These custom propagators can be very efficient, even approaching (and in some cases exceeding) the efficiency of hand-optimised propagators.

Given an arbitrary constraint, we show how to generate a custom propagator that establishes GAC in small polynomial time. This is done by precomputing the propagation that would be performed on every relevant subdomain. The number of relevant subdomains, and therefore the size of the generated propagator, is potentially exponential in the number and domain size of the constrained variables.

The limiting factor of our approach is the size of the generated propagators. We investigate symmetry as a means of reducing that size. We exploit the symmetries of the constraint to merge symmetric parts of the generated propagator. This extends the reach of our approach to somewhat larger constraints, with a small run-time penalty.

Our experimental results show that, compared with optimised implementations of the table constraint, our techniques can lead to an order of magnitude speedup. Propagation is so fast that the generated propagators compare well with hand-written carefully optimised propagators for the same constraints, and the time taken to generate a propagator is more than repaid.

1. Introduction

Constraint Programming is a proven technology for solving complex combinatorial problems from a range of disciplines, including scheduling (nurse rostering, resource allocation for data centres), planning (contingency planning for air traffic control, route finding for international container shipping, assigning service professionals to tasks) and design (of cryptographic S-boxes, carpet cutting to minimise waste). Constraint solving of a combinatorial problem proceeds in two phases. First, the problem is modelled as a set of decision variables with a set of constraints on those variables that a solution must satisfy. A decision variable represents a choice that must be made in order to solve the problem. Consider Sudoku as a simple example. Each cell in the 9×9 square must be filled in such a way that each row, column and 3×3 sub-square contain all distinct non-zero digits. In a constraint model of Sudoku, each cell is a decision variable with the domain $\{1 \ldots 9\}$. The

* Corresponding authors.
E-mail addresses: ian.gent@st-andrews.ac.uk (I.P. Gent), caj21@st-andrews.ac.uk (C. Jefferson), sl4@st-andrews.ac.uk (S. Linton), ijm@st-andrews.ac.uk (I. Miguel), pwn1@st-andrews.ac.uk (P. Nightingale).

http://dx.doi.org/10.1016/j.artint.2014.03.001

constraints require that subsets of the decision variables corresponding to the rows, columns and sub-squares of the Sudoku grid are assigned distinct values.

The second phase is solving the modelled problem using a constraint solver. A solution is an assignment to decision variables satisfying all constraints, e.g. a valid solution to a Sudoku puzzle. A constraint solver typically works by performing a systematic search through a space of possible solutions. This space is usually vast, so search is combined with constraint *propagation*, a form of inference that allows the solver to narrow down the search space considerably. A constraint propagator is an algorithm that captures a particular pattern of such inference, for example requiring each of a collection of variables to take distinct values. A state-of-the-art constraint solver has a suite of such propagators to apply as appropriate to an input problem. In this paper we will consider propagators that establish a property called Generalised Arc Consistency (GAC) [1], which requires that every value in the domains of the variables in the scope of a particular constraint participates in at least one assignment that satisfies that constraint.

Constraint models of structured problems often contain many copies of a constraint, which differ only in their scope. English Peg Solitaire,[1] for example, is naturally modelled with a *move* constraint for each of 76 moves, at each of 31 time steps, giving 2356 copies of the constraint [2]. Efficient implementation of such a constraint is vital to solving efficiency, but choosing an implementation is often difficult.

The solver may provide a hand-optimised propagator matching the constraint. If it does not, the modeller can use a variety of algorithms which achieve GAC propagation for arbitrary constraints, for example GAC2001 [3], GAC-Schema [4], MDDC [5], STR2 [6], the Trie table constraint [7], or Regular [8]. Typically these propagators behave well when the data structure they use (whether it is a trie, multi-valued decision diagram (MDD), finite automaton, or list of tuples) is small. They all run in exponential time in the worst case, but run in polynomial time when the data structure is of polynomial size.

The algorithms we give herein generate GAC propagators for arbitrary constraints that run in time $O(nd)$ (where n is the number of variables and d is the maximum domain size), in extreme cases an exponential factor faster than any table constraint propagator [3,7,9,5,6,10–13]. As our experiments show, generated propagators can even outperform hand-optimised propagators when performing the same propagation. It can take substantial time to generate a GAC propagator, however the generation time is more than repaid on the most difficult problem instances in our experiments.

Our approach is general but in practice does not scale to large constraints as it precomputes domain deletions for all possible inputs of the propagator (i.e. all reachable subsets of the initial domains). However, it remains widely applicable — like the aforementioned Peg Solitaire model, many other constraint models contain a large number of copies of one or more small constraints.

Propagator trees

Our first approach is to generate a binary tree to store domain deletions for all reachable subdomains. The tree branches on whether a particular literal (variable, value pair) is in domain or not, and each node of the tree is labelled with a set of domain deletions. After some background in Section 2, the basic approach is described in Section 3.

We have two methods of executing the propagator trees. The first is to transform the tree into a program, compile it and link it to the constraint solver. The second is a simple virtual machine: the propagator tree is encoded as a sequence of instructions, and the constraint solver has a generic propagator that executes it. Both these methods are described in Section 3.5.

The generated trees can be very large, but this approach is made feasible for small constraints (both to generate the tree, and to transform, compile and execute it) by refinements and heuristics described in Section 4. The binary tree approach is experimentally evaluated in Section 5, demonstrating a clear speed-up on three different problem classes.

Exploiting symmetry

The second part of the paper is about exploiting symmetry. We define the symmetry of a constraint as a permutation group on the literals, such that any permutation in the group maintains the semantics of the constraint. This allows us to compress the propagator trees: any two subtrees that are symmetric are compressed into one. In some cases this replaces an exponential sized tree with a polynomially sized symmetry-reduced tree. Section 6 gives the necessary theoretical background. In that section we develop a novel algorithm for finding the canonical image of a sequence of sets under a group that acts pointwise on the sets. We believe this is a small contribution to computational group theory.

Section 7 describes how the symmetry-reduced trees are generated, and gives some bounds on their size under some symmetry groups. Executing the symmetry-reduced trees is not as simple as for the standard trees. Both the code generation and VM approaches are adapted in Section 7.3.

In Section 8 we evaluate symmetry-reduced trees compared to standard propagator trees. We show that exploiting symmetry allows propagator trees to scale to larger constraints.

[1] Problem 37 at www.csplib.org.

2. Theoretical background

We briefly give the most relevant definitions, and refer the reader elsewhere for more detailed discussion [1].

Definition 1. A **CSP instance**, P, is a triple $\langle V, D, C\rangle$, where: V is a finite set of **variables**; D is a function from variables to their **domains**, where $\forall v \in V : D(v) \subset \mathbb{Z}$ and $D(v)$ is finite; and C is a set of **constraints**. A **literal** of P is a pair $\langle v, d\rangle$, where $v \in V$ and $d \in D(v)$. An **assignment** to any subset $X \subseteq V$ is a set consisting of exactly one literal for each variable in X. Each **constraint** c is defined over a list of variables, denoted $scope(c)$. A constraint either forbids or allows each assignment to the variables in its scope. An assignment S to V **satisfies** a constraint c if S contains an assignment allowed by c. A **solution** to P is any assignment to V that satisfies all the constraints of P.

Constraint propagators work with subdomain lists, as defined below.

Definition 2. For a set of variables $X = \{x_1 \dots x_n\}$ with original domains $D(x_1), \dots, D(x_n)$, a **subdomain list** S for X is a function from variables to sets of domain values that satisfies: $\forall i \in \{1 \dots n\}$: $S(x_i) \subseteq D(x_i)$. We extend the $\subseteq$ notation to write $R \subseteq S$ for subdomain lists R and S iff $\forall i \in \{1 \dots n\}$: $R(x_i) \subseteq S(x_i)$. Given a CSP instance $P = \langle V, D, C\rangle$, a **search state** for P is a subdomain list for V. An assignment A is contained in a subdomain list S iff $\forall \langle v, d\rangle \in A$: $d \in S(v)$ (and if $S(v)$ is not defined then $d \in S(v)$ is false).

Backtracking search operates on search states to solve CSPs. During solving, the search state is changed in two ways: branching and propagation. Propagation removes literals from the current search state without removing solutions. Herein, we consider only propagators that establish Generalised Arc Consistency (GAC), which we define below. Branching is the operation that creates a search tree. For a particular search state S, branching splits S into two states S_1 and S_2, typically by splitting the domain of a variable into two disjoint sets. For example, in S_1 branching might make an assignment $x \mapsto a$ (by excluding all other literals of x), and in S_2 remove only the literal $x \mapsto a$. S_1 and S_2 are recursively solved in turn.

Definition 3. Given a constraint c and a subdomain list S of $scope(c)$, a literal $\langle v, d\rangle$ is *supported* iff there exists an assignment that satisfies c and is contained in S and contains $\langle v, d\rangle$. S is **Generalised Arc Consistent (GAC)** with respect to c iff, for every $d \in S(v)$, the literal $\langle v, d\rangle$ is supported.

Any literal that does not satisfy the test in Definition 3 may be removed. In practice, CP solvers fail and backtrack if any domain is empty. Therefore propagators can assume that every domain has at least one value in it when they are called. Therefore we give a definition of GAC propagator that has as a precondition that all domains contain at least one value. This precondition allows us to generate smaller and more efficient propagators in some cases.

Definition 4. Given a CSP $P = \langle V, D, C\rangle$, a search state S for P where each variable $x \in V$ has a non-empty domain: $|S(x)| > 0$, and a constraint $c \in C$, the *GAC propagator* for c returns a new search state S' which:

1. For all variables not in $scope(c)$: is identical to S.
2. For all variables in $scope(c)$: omits all (and only) literals in S that are not supported in c, and is otherwise identical to S.

3. Propagator generation

We introduce this section by giving a naïve method that illustrates our overall approach. Then we present a more sophisticated method that forms the basis for the rest of this paper.

3.1. A naïve method

GAC propagation is NP-hard for some families of constraints defined intensionally. For example, establishing GAC on the constraint $\sum_i x_i = 0$ is NP-hard, as it is equivalent to the subset-sum problem [14] (§35.5). However, given a constraint c on n variables, each with domain size d, it is possible to generate a GAC propagator that runs in time $O(nd)$. The approach is to precompute the deletions performed by a GAC algorithm for every subdomain list for $scope(c)$. Thus, much of the computational cost is moved from the propagator (where it may be incurred many times during search) to the preprocessing step (which only occurs once).

The precomputed deletions are stored in an array T mapping subdomain lists to sets of literals. The generated propagator reads the domains (in $O(nd)$ time), looks up the appropriate subdomain list in T and performs the required deletions. T can be indexed as follows: for each literal in the initial domains, represent its presence or absence in the subdomain list with a bit, and concatenate the bits to form an integer.

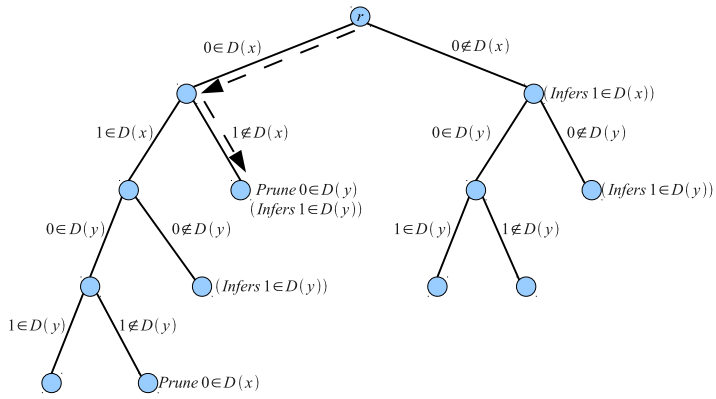


Fig. 1. Example of propagator tree for constraint $x \vee y$ with initial domains of $\{0, 1\}$.

T can be generated in $O((2^d - 1)^n.n.d^n)$ time. There are $2^d - 1$ non-empty subdomains of a size d domain, and so $(2^d - 1)^n$ non-empty subdomain lists on n variables. For each, GAC is enforced in $O(n.d^n)$ time and the set of deletions is recorded. As there are at most nd deletions, T is size at most $(2^d - 1)^n.nd$.

3.2. Propagator trees

The main disadvantage of the naïve method is that it computes and stores deletions for many subdomain lists that cannot be reached during search. A second disadvantage is that it must read the entire search state (for variables in scope) before looking up the deletions. We address both problems by using a tree to represent the generated propagator. The tree represents only the subdomain lists that are *reachable*: no larger subdomain list fails or is entailed. This improves the average- but not the worst-case complexity.

In this section we introduce the concept of a *propagator tree*. This is a rooted binary tree with labels on each node representing actions such as querying domains and pruning domain values. A propagator tree can straightforwardly be translated into a program or an executable bytecode. We will describe an algorithm that generates a propagator tree, given any propagator and entailment checker for the constraint in question. First we define propagator tree.

Definition 5. A *propagator tree node* is a tuple $T = \langle Left, Right, Prune, Test\rangle$, where *Left* and *Right* are propagator tree nodes (or *Nil*), *Prune* is a set of literals to be deleted at this node, and *Test* is a single literal. Any of the items in the tuple may be *Nil*. A *propagator tree* is a rooted tree of nodes of type T. The root node is named r. We use dot to access members of a tree node v, so for example the left subtree is $v.Left$.

Example 1. Suppose we have the constraint $x \vee y$ with initial domains of $\{0, 1\}$. An example propagator tree for this constraint is shown in Fig. 1. The tree first branches to test whether $0 \in D(x)$. In the branch where $0 \notin D(x)$, it infers that $1 \in D(x)$ because otherwise $D(x)$ would be empty. Both subtrees continue to branch until the domains $D(x)$ and $D(y)$ are completely known. In two cases, pruning is required (when $D(x) = \{0\}$ and when $D(y) = \{0\}$).

An execution of a propagator tree follows a path in the tree starting at the root r. At each vertex v, the propagator prunes the set of literals specified by $v.Prune$. If $v.Test$ is *Nil*, then the propagator is finished. Otherwise, the propagator tests if the literal $v.Test = (x_i, a)$ is in the current subdomain list S. If $a \in S(x_i)$, then the next vertex in the path is the left child $v.Left$, otherwise it is the right child $v.Right$. If the relevant child is *Nil*, then the propagator is finished.

Example 2. Continuing from Example 1, suppose we have $D(x) = \{0\}$, $D(y) = \{0, 1\}$. The dashed arrows in Fig. 1 show the execution of the propagator tree, starting at r. First the value 0 of $D(x)$ is tested, and found to be in the domain. Second, the value 1 of $D(x)$ is tested and found to be not in the domain. This leads to a leaf node where 0 is pruned from $D(y)$. The other value of y is assumed to be in the domain (otherwise the domain is empty and the solver will fail and backtrack).

3.3. Comparing propagator trees to handwritten propagators

Handwritten propagators make use of many techniques for efficiency. For example they often have state variables that are incrementally updated and stored between calls to the propagator. They also make extensive use of *triggers* — notifications from the solver about how domains have changed since the last call (for example, literal $\langle x, a\rangle$ has been pruned).

In contrast, propagator trees are stateless. They also do not use triggers. It is not clear how triggers could be used with a single tree because the order that trigger events arrive has no relation to the order of branching in the tree. In future work

Algorithm 1 SimpleGenTree(c, SD, $ValsIn$)

```
1: Deletions ← Propagate(c, SD)
2: SD′ ← SD \ Deletions
3: if all domains in SD′ are empty then
4:     return T = ⟨Prune = Deletions, Test = Nil, Left = Nil, Right = Nil⟩
5: ValsIn* ← ValsIn \ Deletions
6: ValsIn′ ← ValsIn* ∪ {(x, a) | (x, a) ∈ SD′, |SD′(x)| = 1}
7: if SD′ = ValsIn′ then
8:     return T = ⟨Prune = Deletions, Test = Nil, Left = Nil, Right = Nil⟩
   {Pick a variable and value, and branch}
9: (y, l) ← heuristic(SD′ \ ValsIn′)
10: LeftT ← SimpleGenTree(c, SD′, ValsIn′ ∪ (y, l))
11: RightT ← SimpleGenTree(c, SD′ \ {(y, l)}, ValsIn′)
12: return T = ⟨Prune = Deletions, Test = (y, l), Left = LeftT, Right = RightT⟩
```

we plan to create multiple propagator trees which will be executed for different trigger events, dividing responsibility for achieving GAC among the trees.

3.4. Generating propagator trees

SimpleGenTree (Algorithm 1) is our simplest algorithm to create a propagator tree given a constraint c and the initial domains D. The algorithm is recursive and builds the tree in depth-first left-first order. When constructed, each node in a propagator tree will test values to obtain more information about S, the current subdomain list (Definition 2). At a given tree node, each literal from the initial domains D may be in S, or out, or unknown (not yet tested). SimpleGenTree has a subdomain list SD for each tree node, representing values that are in S or unknown. It also has a second subdomain list $ValsIn$, representing values that are known to be in S. Algorithm 1 is called as SimpleGenTree(c, D, $\emptyset$), where c is the parameter of the Propagate function (called on line 1) and D is the initial domains. For all our experiments, Propagate is a positive GAC table propagator and thus c is a list of satisfying tuples.

SimpleGenTree proceeds in two stages. First, it runs a propagation algorithm on SD to compute the prunings required given current knowledge of S. This set of prunings is conservative in the sense that they can be performed whatever the true value of S because $S \subseteq SD$. The prunings are stored in the current tree node, and each pruned value is removed from SD to form SD'. If a domain is empty in SD', the algorithm returns. Pruned values are also removed from $ValsIn$ to form $ValsIn'$ — these values are known to be in S, but the propagator tree will remove them from S. Furthermore, if only one value remains for some variable in SD', the value is added to $ValsIn'$ (otherwise the domain would be empty).

Propagate is assumed to empty all variable domains if the constraint is not satisfiable with the subdomain list SD. A GAC propagator (according to Definition 4) will do this, however Propagate does not necessarily enforce GAC. The proof of correctness below is simplified by assuming Propagate always enforces GAC.

Throughout this paper we will only consider GAC propagators according to Definition 4. If the Propagate function does not enforce GAC then the propagator tree that is generated does not necessarily enforce the same degree of consistency as Propagate. Characterising non-GAC propagator trees is not straightforward and we leave an investigation of this to future work.

The second stage is to choose a literal and branch. This literal is unknown, i.e. in SD' but not $ValsIn'$. SimpleGenTree recurses for both left and right branches. On the left branch, the chosen literal is added to $ValsIn$, because it is known to be present in S. On the right, the chosen literal is removed from SD. There are two conditions that terminate the recursion. In both cases the algorithm attaches the deletions to the current node and returns. The first condition is that all domains have been emptied by propagation. The second condition is $SD' = ValsIn'$. At this point, we have complete knowledge of the current search state S: $SD' = ValsIn' = S$.

3.5. Executing a propagator tree

We compare two approaches to executing propagator trees. The first is to translate the tree into program code and compile it into the solver. This results in a very fast propagator but places limitations on the size of the tree. The second approach is to encode the propagator tree into a stream of instructions, and execute them using a simple virtual machine.

3.5.1. Code generation

Algorithm 2 (GenCode) generates a program from a propagator tree via a depth-first, left-first tree traversal. It is called initially with the root r. GenCode creates the body of the propagator function, the remainder is solver specific. In the case of Minion solver specific code is very short and the same for all propagator trees.

3.5.2. Virtual machine

The propagator tree is encoded into an array of integers. Each instruction is encoded as a unique integer followed by some operands. The virtual machine has only three instructions, as follows.

Algorithm 2 GenCode(Propagator tree T, vertex v)

```
1: if v = Nil then
2:     WriteToCode("NoOperation;")
3: else
4:     WriteToCode("RemoveValuesFromDomains("+v.Prune+");")
5:     if v.Test ≠ Nil then
6:         (x_i, a) ← v.Test
7:         WriteToCode("if IsInDomain("+a+","+x_i+") then")
8:         GenCode(T, v.Left)
9:         WriteToCode("else")
10:        GenCode(T, v.Right)
11:        WriteToCode("endif;")
```

Branch : $\langle var, val, pos\rangle$ — If the value val is *not* in the domain of the variable var then jump to position pos. Otherwise, execution continues with the next instruction in the sequence. A jump to -1 ends execution of the virtual machine.

Prune : $\langle var1, val1, var2, val2, \ldots, -1\rangle$ — Prune a set of literals from the variable domains. The operands are a list of variable–value pairs terminated by -1.

Return : $\langle\rangle$ — End execution of the virtual machine.

Tree nodes are encoded in depth-first left-first order, and execution of the instruction stream starts at location 0. Any node that has a left child is immediately followed by its left child. The Branch instruction will either continue at the next instruction (the left child) or jump to the location of the right child. When an internal node is encoded, the position of its right child is not yet known. We insert placeholders for pos in the branch instruction and fill them in during a second pass.

The VM clearly has the advantage that no compilation is required, however it is somewhat slower than the code generation approach in our experiments below.

3.6. Correctness

In order to prove the SimpleGenTree algorithm correct, we assume that the Propagate function called on line 1 enforces GAC exactly as in Definition 3. In particular, if Propagate produces a domain wipe-out, it must delete all values of all variables in the scope. This is not necessarily the case for GAC propagators commonly used in solvers. We also assume that the target constraint solver removes all values of all variables in a constraint if our propagator tree empties any individual domain. In practice, constraint solvers often have some shortcut method, such as a special function *Fail* for these situations, but our proofs are slightly cleaner for assuming domains are emptied. Finally we implicitly match up nodes in the generated trees with corresponding points in the generated code for the propagator. Given these assumptions, we will prove that the code we generate does indeed establish GAC.

Lemma 1. *Assuming that the Propagate function in line 1 establishes GAC, then: given inputs* $(c, SD, ValsIn)$*, if Algorithm 1 returns at line 4 or line 8, the resulting set of prunings achieve GAC for the constraint c on any search state S such that* $ValsIn \subseteq S \subseteq SD$.

Proof. If Algorithm 1 returns on either line 4 or line 8, the set of deletions returned are those generated on line 1. These deletions achieve GAC propagation for the search state SD.

If the GAC propagator for c would remove a literal from SD, then that literal is in no assignment which satisfies c and is contained in SD. As S is contained in SD, that literal must also be in no assignment that satisfies c and is contained in S. Therefore any literals in S that are removed by a GAC propagator for SD would also be removed by a GAC propagator for S.

We now show no extra literals would be removed by a GAC propagator for S. This is separated into two cases. The first case is if Algorithm 1 returns on line 4. Then GAC propagation on SD has removed all values from all domains. There are therefore no further values which can be removed, so the result follows trivially.

The second case is if Algorithm 1 returns on line 8. Then $SD' = ValsIn'$ on line 7. Any literals added to $ValsIn'$ on line 6 are also in S, as literals are added when exactly one value exists in the domain of a variable in SD, and so this value must also be in S, otherwise there would be an empty domain in S. Thus we have $ValsIn' \subseteq (S \setminus \text{Deletions}) \subseteq SD'$. But since $ValsIn' = SD'$, we also have $SD' = S \setminus \text{Deletions}$. Since we know SD' is GAC by the assumed correctness of the Propagate function, so is $S \setminus \text{Deletions}$. □

Theorem 1. *Assuming that the Propagate function in line 1 establishes GAC, then: given inputs* $(c, SD, ValsIn)$*, then the code generator Algorithm 2 applied to the result of Algorithm 1 returns a correct GAC propagator for search states S such that* $ValsIn \subseteq S \subseteq SD$.

Proof. We shall proceed by induction on the size of the tree generated by Algorithm 1. The base is that the tree contains just a single leaf node, and this case is implied by Lemma 1. The rest of the proof is therefore the induction step that a tree node is correct given both its left and right children (if present) are correct. For this proof, we implicitly match up nodes generated by Algorithm 1 with points in the code generated by Algorithm 2.

By the same argument used in Lemma 1, the Deletions generated on line 1 can also be removed from S. If applying these deletions to S leads to a domain wipe-out, then the constraint solver sets $S(x) = \emptyset$ for all $x \in scope(c)$, and the propagator has established GAC, no matter what happens in the rest of the tree.

If no domain wipe-out occurs, we progress to line 9. At this point we know that $ValsIn' \subseteq S \setminus \text{Deletions} \subseteq SD'$. Also, since we passed line 7, we know that $ValsIn' \neq SD'$, and therefore there is at least one literal for the heuristic to choose. There are now two cases. The literal (y, l) chosen by the heuristic is in S, or not.

If $l \in S(y)$, then the generated propagator will branch left. The propagator generated after this branch is generated from the tree produced by SimpleGenTree$(c, SD', ValsIn' \cup (y, l))$. Since $l \in S(y)$, we have $ValsIn' \cup (y, l) \subseteq S \setminus \text{Deletions} \subseteq SD'$. Since the tree on the left is strictly smaller, we can appeal to the induction hypothesis that we have generated a correct GAC propagator for $S \setminus \text{Deletions}$. Since we know that Deletions were correctly deleted from S, we have a correct GAC propagator at this node for S.

If $l \notin S(y)$, the generated propagator branches right. The propagator on the right is generated from the tree given by SimpleGenTree$(c, SD' \setminus (y, l), ValsIn')$ on $S \setminus \text{Deletions}$. Here we have $ValsIn' \subseteq S \setminus \text{Deletions} \subseteq SD' \setminus (y, l)$. As in the previous case, the requirements of the induction hypothesis are met and we have a correct GAC propagator for S.

Finally we note that the set $SD \setminus ValsIn$ is always reduced by at least one literal on each recursive call to Algorithm 1. Therefore we know the algorithm will eventually terminate. □

Corollary 1. *Assuming the Propagate function correctly establishes GAC for any constraint c, then the code generator Algorithm 2 applied to the result of Algorithm 1 with inputs $(c, D, \emptyset)$, where D are the initial domains of the variables in c, generates a correct GAC propagator for all search states.*

Lemma 2. *If r is the time a solver needs to remove a value from a domain, and s the time to check whether or not a value is in the domain of a variable, the code generated by Algorithm 2 runs in time $O(nd \max(r, s))$.*

Proof. The execution of the algorithm is to go through a single branch of an if/then/else tree. The tree cannot be of depth greater than nd since one literal is chosen at each depth and there are at most nd literals in total. Furthermore, on one branch any given literal can either be removed from a domain or checked, but not both. This is because Algorithm 1 never chooses a test from a removed value. Therefore the worst case is nd occurrences of whichever is more expensive out of testing domain membership and removing a value from a domain. □

In some solvers both r and s are $O(1)$, e.g. where domains are stored only in bitarrays. In such solvers our generated GAC propagator is $O(nd)$.

4. Generating smaller trees

Algorithm 3 shows the GenTree algorithm. This is a refinement of SimpleGenTree. We present this without proof of correctness, but a proof would be straightforward since the effect is only to remove nodes in the tree for which no propagation can occur in the node and the subtree beneath it.

The first efficiency measure is that GenTree always returns *Nil* when no pruning is performed at the current node and both children are *Nil*, thus every leaf node of the generated propagator tree performs some pruning. The second measure is to use an entailment checker. A constraint is *entailed* with respect to a subdomain list SD if every tuple allowed on SD is allowed by the constraint. When a constraint is entailed there is no possibility of further pruning. We assume we have a function entailed(c, SD) to check this. The function is called at the start of GenTree, and also after the subdomain list is updated by pruning (line 9). In both cases, entailment leads to the function returning before making the recursive calls.

To illustrate the difference between SimpleGenTree and GenTree, consider Fig. 2. The constraint is very small ($x \vee y$ on boolean variables, the same constraint as used in Fig. 1) but even so SimpleGenTree generates 7 more nodes than GenTree. The figure illustrates the effectiveness and limitations of entailment checking. Subtree C contains no prunings, therefore it would be removed by GenTree with or without entailment checking. However, the entailment check is performed at the topmost node in subtree C, and GenTree immediately returns (line 2) without exploring the four nodes beneath. Subtree B is entailed, but the entailment check does not reduce the number of nodes explored by GenTree compared to SimpleGenTree. Subtree A is not entailed, however GAC does no prunings here so GenTree will explore this subtree but not output it.

4.1. Bounds on tree size

At each internal node, the tree branches for some literal in SD' that is not in $ValsIn'$. Each unique literal may be branched on at most once down any path from the root to a leaf node. This means the number of bifurcations is at most nd down any path. Therefore the size of the tree is at most $2 \times (2^{nd}) - 1 = 2^{nd+1} - 1$ which is $O(2^{nd})$.

The dominating cost of GenTree for each node is calling the constraint propagator on line 3. We use GAC2001, and its time complexity is $O(n^2 d^n)$ [3]. Detecting entailment is less expensive. To implement entailment and the heuristic, we maintain a list of all tuples within SD that do not satisfy the constraint. It takes $O(nd^n)$ to filter this list at each node, and the constraint is entailed when the list is empty. Overall the time complexity of GenTree is $O(n^2 d^n \times 2^{nd})$.

Algorithm 3 Generate propagator tree: GenTree(c, SD, $ValsIn$)

```
1: if entailed(c, SD) then
2:     return Nil
3: Deletions ← Propagate(c, SD)
4: SD' = SD \ Deletions
5: if all domains in SD' are empty then
6:     return T = ⟨Prune = Deletions, Test = Nil, Left = Nil, Right = Nil⟩
7: ValsIn* ← ValsIn \ Deletions
8: ValsIn' ← ValsIn* ∪ {(x, a)|(x, a) ∈ SD', |SD'(x)| = 1}
9: if SD' = ValsIn' or entailed(c, SD') then
10:    if Deletions = ∅ then
11:        return Nil
12:    else
13:        return T = ⟨Prune = Deletions, Test = Nil, Left = Nil, Right = Nil⟩
   {Pick a variable and value, and branch}
14: (y, l) ← heuristic(SD' \ ValsIn')
15: LeftT ← GenTree(c, SD', ValsIn' ∪ (y, l))
16: RightT ← GenTree(c, SD' \ {(y, l)}, ValsIn')
17: if LeftT = Nil And RightT = Nil And Deletions = ∅ then
18:    return Nil
19: else
20:    return T = ⟨Prune = Deletions, Test = (y, l), Left = LeftT, Right = RightT⟩
```

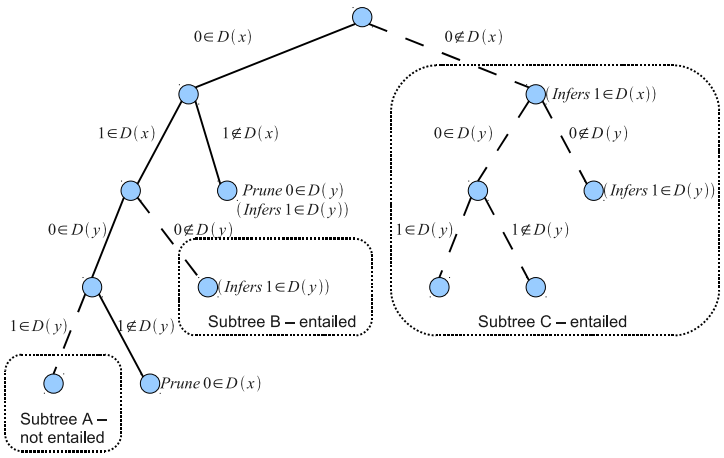


Fig. 2. Example of propagator tree for constraint $x \vee y$ with initial domains of $\{0, 1\}$. The entire tree is generated by SimpleGenTree (Algorithm 1). The more sophisticated algorithm GenTree (Algorithm 3) does not generate the subtrees A, B and C.

For many constraints GenTree is very efficient and does not approach its upper bound. The lemma below gives an example of a constraint where GenTree does generate a tree of exponential size.

Lemma 3. *Consider the parity constraint on a list of variables $\langle x_1, \ldots, x_n \rangle$ with domain $\{0, 1\}$. The constraint is satisfied when the sum of the variables is even. Any propagator tree for this constraint must have at least 2^{n-1} nodes.*

Proof. The parity constraint propagates in exactly one case. When all but one variable is assigned, the remaining variable must be assigned such that the parity constraint is true. If there are two or more unassigned variables, then no propagation can be performed.

Suppose we select the first $n-1$ variables and assign them in any way (naming the assignment A), leaving x_n unassigned. x_n must then be assigned either 0 or 1 by pruning, and the value depends on every other variable (and on every other variable being known to be assigned). The tree node that performs the pruning for A cannot be reached for any other assignment $B \neq A$ to the first $n-1$ variables, as the node for A requires knowing the whole of A to be able to prune x_n. Therefore there must be a distinct node in the propagator tree for each of the 2^{n-1} assignments to the first $n-1$ variables. □

4.2. Heuristic

The choice of literal to branch on is very important, and can make a huge difference in the size of the propagator tree. In this section we propose some dynamic heuristics and compare them.

Table 1
Size of propagator tree for proposed heuristics and anti-heuristics. Figures for the Random heuristic are a mean of ten trees, each other tree was generated once. Where it took longer than 24 hours to generate a single tree, the entry reads >24 h. SR denotes symmetry-reduced trees.

	Entail	AntiEnt	Static	LMF	SMF	SM+DF	Random
LABS 2	396	473	**372**	469	**372**	**372**	488
LABS 2 SR	62	155	**60**	161	**60**	**60**	265
LABS 3	4728	8284	**4316**	7207	**4316**	**4316**	7780
LABS 3 SR	171	764	**166**	828	**166**	**166**	2658
LABS 4	52,004	154,619	**47,092**	114,665	**47,092**	**47,092**	124,381
LABS 4 SR	398	4139	**390**	3697	**390**	**390**	25,550
LABS5 SR	747	16,613	**736**	14,373	**736**	**736**	209,970
LABS 6 SR	**1287**	62,172	1336	49,767	1336	1336	>24 h
Life	28,351	**11,057**	26,524	12,061	26,524	26,524	24,904
Life SR	740	683	**410**	476	**410**	**410**	7682
Brian SR	185,252	111,443	132,668	**106,267**	135,575	135,575	>24 h
Immig SR	121,070	39,977	34,717	59,839	**34,712**	**34,712**	>24 h
PegSol	316	191	315	**161**	315	315	222
PegSol SR	95	83	94	**66**	94	94	160

Entailment heuristic

To minimise the size of the tree, the aim of this heuristic is to cause Algorithm 3 to return before branching. There are a number of conditions that cause this: entailment (lines 1 and 9); domain wipe-out (line 6); and complete domain information (line 9).

The proposed heuristic greedily attempts to make the constraint entailed. This is done by selecting the literal contained in the greatest number of disallowed tuples of c that are valid with respect to SD'. If this literal is invalid (as in the right subtree beneath the current node), then the greatest possible number of disallowed tuples will be removed from the set.

Smallest Domain heuristics

Smallest Domain First (SDF) is a popular variable ordering heuristic for CP search. We investigate two ways of adapting SDF. The first, Smallest Maybe First (SMF) selects a variable with the smallest non-zero number of literals in $SD' \setminus ValsIn'$. SMF will tend to prefer variables with small initial domains, then prefer to obtain complete domain information for one variable before moving on to the next. Preferring small domains could be a good choice because on average each deleted value from a small domain will be in a large number of satisfying tuples. Ties are broken by the static order of the variables in the scope. Once a variable is chosen, the smallest literal for that variable is chosen from $SD' \setminus ValsIn'$.

The second adaptation is Smallest Maybe+Domain First (SM+DF). This is similar to SMF with two changes: when selecting the variable SD' is used in place of $SD' \setminus ValsIn'$, and variables are chosen from the set of variables that have at least one literal in $SD' \setminus ValsIn'$ (otherwise SM+DF could choose a variable with no remaining literals to branch on).

Comparison

We compare the three proposed heuristics Entail, SMF and SM+DF against corresponding anti-heuristics AntiEntail and LMF (Largest Maybe First), one static ordering, and a dynamic random ordering (at each node a literal is chosen at random with uniform probability). We used all the constraints from both sets of experiments (in Sections 5 and 8).

The static ordering for Peg Solitaire and LABS is the order the constraints are written in Sections 5.2 and 5.3 respectively. For Life, Immigration and Brian's Brain, the neighbour variables are branched first, then the variable representing the current time-step, then the next time-step.

Table 1 shows the size of propagator trees for each of the heuristics. Static, SMF and SM+DF performed well overall. SMF and SM+DF produced trees of identical size. In two cases (Brian Sym and Immigration Sym) the tree generated with the static ordering is slightly larger than SMF. In most cases SMF performed better than its anti-heuristic LMF. SMF also has the advantage that the user need not provide an ordering.

Comparing the Entailment heuristic to Random shows that Entailment does have some value, but Entailment proved to be worse than SMF and Static in most cases. Also, Entailment is beaten by its anti-heuristic in 6 cases as opposed to 4 for SMF.

We use the SMF heuristic for all experiments in Sections 5 and 8.

4.3. Implementation of GenTree

The implementation of Algorithm 3 is recursive and very closely follows the structure of the pseudocode. It is instantiated with the GAC2001 table propagator [3]. The implementation maintains a list of disallowed tuples of c that are valid with respect to SD (or SD' after line 4). This list is used by the entailment checker: when the list becomes empty, the constraint

Table 2
Time taken to generate the propagator trees in Python and the C++ compiler.

	GenTree	Compiler
LABS	0.32	20.89
Life	8.26	4054.17
Peg Solitaire	0.37	21.58

is entailed. It is also used to calculate the entailment heuristic described above. It is implemented in Python and is not highly optimised. It is executed using the PyPy JIT compiler[2] version 1.9.0.

5. Experimental evaluation of propagator trees

In all the case studies below, we use the solver Minion [16] 0.15. We experiment with 3 propagator trees, in each case comparing against hand-optimised propagators provided in Minion, and also against generic GAC propagators (as described in the subsection below). All instances were run 5 times and the mean was taken. In all cases times are given for an 8-core Intel Xeon E5520 at 2.27 GHz with 12 GB RAM. Minion was compiled with g++ 4.7.3, optimisation level −O3. For all experiments 6 Minion processes were executed in parallel. We ran all experiments with a 24 hour timeout, except where otherwise stated.

Table 2 reports the time taken to run GenTree, and separately to compile each propagator and link it to Minion. The propagator trees are compiled exactly as every other constraint in Minion is compiled. Specifically they are compiled once for each variable type, 7 times in total. In the case of Life, in our previous work [15] we compiled the propagator tree once (for Boolean variables), taking 217 s, whereas here it takes 4054.17 s. In each experiment in this section, we build exactly one propagator tree, which is then used for all instances in that experiment, and on multiple scopes for each instance.

5.1. Generic GAC propagators

In some cases a generic GAC propagator can enforce GAC in polynomial time. Typically this occurs if the size of the data structure representing the constraint is bounded by a polynomial. Generic propagators can also perform well when there is no polynomial time bound simply because they have been the focus of much research effort.

We compare propagator trees to three table constraints: Table, Lighttable, and STR2+. Table uses a trie data structure with watched literals [7]. Lighttable employs the same trie data structure but is stateless and uses static triggers. Lighttable searches for support for each value of every variable each time it is called. Finally STR2+ is the optimised simple tabular reduction propagator by Lecoutre [6].

We also compare against MDDC, the MDD propagator of Cheng and Yap [5]. The MDD is constructed from the set of satisfying tuples. The MDDC propagator is implemented exactly as described by Cheng and Yap, and we used the sparse set variant. To construct the MDD, we used a simpler algorithm than Cheng and Yap. Our implementation first builds a complete trie representing the positive tuples, then converts the trie to an MDD by compressing identical subtrees.

Many of our benchmark constraints can be represented compactly using a Regular constraint [8]. We manually created deterministic finite automata for these constraints. These automata are given elsewhere [17] for space reasons. In the experiments we use the Regular decomposition of Beldiceanu et al. [18] which has a sequence of auxiliary variables representing the state of the automaton at each step, and a set of ternary table constraints each representing the transition table. We enforce GAC on the table constraints and this obtains GAC on the original Regular constraint.

5.2. Case study: English Peg Solitaire

English Peg Solitaire is a one-player game played with pegs on a board. It is Problem 37 at www.csplib.org. The game and a model are described by Jefferson et al. [2]. The game has 33 board positions (*fields*), and begins with 32 pegs and one hole. The aim is to reduce the number of pegs to 1. At each move, a peg (A) is jumped over another peg (B) and into a hole, and B is removed. As each move removes one peg, we fix the number of time steps in our model to 32.

The model we use is as follows. The board is represented by a Boolean array $b[32, 33]$ where the first index is the time step $\{0 \ldots 31\}$ and the second index is the field $\{1 \ldots 33\}$. The moves are represented by Boolean variables $moves[31, 76]$, where the first index is the time step $\{0 \ldots 30\}$ (where move 0 connects board states 0 and 1), and the second index is the move number, where there are 76 possible moves. The third set of Boolean variables are $equal[31, 33]$, where the first index is the time step $\{0 \ldots 30\}$ and the second is the field. The following constraint is posted for each *equal* variable: $equal[x, y] \Leftrightarrow (b[x, y] = b[x+1, y])$. The board state for the first and last time step are filled in, with one hole at the starting position, and one peg at the same position in the final time step. We consider only starting positions 1, 2, 4, 5, 9, 10, or 17, because all other positions can be reached by symmetry from one of these seven.

[2] In our previous paper [15] we used the standard Python interpreter therefore timings are different.

Table 3
Results on peg solitaire problems.

Starting position	Node rate (per s)				Nodes
	Propagator tree		Min	Reified Sumgeq	
	Compiled	VM			
1	**9046**	6663	7445	3953	–
2	**5624**	4423	4714	3329	10,268
4	**8634**	6556	7064	4007	–
5	**8684**	6834	7565	4318	–
9	**8827**	6536	6990	4114	–
10	**10,076**	7727	7921	4694	1,486,641
17	**6470**	4797	4702	3502	10,269

Starting position	Node rate (per s)				
	Lighttable	Table	MDDC	Regular	STR2+
1	755	1992	1902	326	1373
2	715	1657	1697	301	1281
4	672	2067	1597	304	1269
5	754	2206	1749	345	1385
9	735	1738	1682	297	1290
10	719	1931	1848	312	1437
17	701	1827	1686	303	1206

For each time step $t \in \{0 \dots 30\}$, exactly one move must be made, therefore constraints are posted to enforce $\sum_i moves[t, i] = 1$. Also for each time step t, the number of pegs on the board is $32 - t$, therefore constraints are posted to enforce $\sum_{i=1}^{33} b[t, i] = 32 - t$.

The bulk of the constraints model the moves. At each time step $t \in \{0 \dots 30\}$, for each possible move $m \in \{0 \dots 75\}$, the effects of move m are represented by an arity 7 Boolean constraint. Move m jumps a piece from field f_1 to f_3 over field f_2. The constraint is as follows.

$$\big(b[t, f_1] \wedge \neg b[t+1, f_1] \wedge b[t, f_2] \wedge \neg b[t+1, f_2] \wedge \neg b[t, f_3] \wedge b[t+1, f_3]\big) \Leftrightarrow moves[t, m]$$

Also, a frame constraint is posted to ensure that all fields other than f_1, f_2 and f_3 remain the same. The constraint states (for all relevant fields f_4) that $equal[t, f_4] = 1$ when $moves[t, m] = 1$.

In this experiment, the arity 7 move constraint is implemented in nine different ways, and all other constraints are invariant. First the move constraint is implemented as a propagator tree (compiled or using the VM). As shown in Table 2, the propagator tree was generated by GenTree in 0.37 s and compiled in 21.58 s. The tree has 315 nodes, and GenTree explored 509 nodes.

The Reified Sumgeq implementation uses a sum to represent the conjunction. The negation of some b variables is achieved with views [19], therefore no auxiliary variables are introduced. The sum constraint is reified to the $moves[t, m]$ variable, as follows: $[(b[t, f_1] + \dots + b[t+1, f_3]) \geqslant 6] \Leftrightarrow moves[t, m]$.

The Min implementation uses a single `min` constraint. Again views are used for negation and no auxiliary variables are introduced. The constraint is as follows: $min(b[t, f_1], \dots, b[t+1, f_3]) = moves[t, m]$.

The move constraint is also implemented using the Lighttable, Table, MDDC and STR2+ propagators. The table has 64 satisfying tuples. The Regular implementation [17] has 9 states and uses a ternary table constraint (representing the transition table) with 17 satisfying tuples.

Table 3 shows our results for peg solitaire. All nine methods enforce GAC, therefore they search exactly the same space. When one or more methods completed the search within the 24 hour timeout, we give the node count. The compiled propagator tree outperforms Min by a substantial margin, which is perhaps remarkable given that Min is a hand-optimised propagator. The compiled propagator tree outperforms Reified Sumgeq by an even wider margin. None of the generic GAC methods Lighttable, Table, MDDC, Regular or STR2+ come close to the handwritten propagators or the propagator tree. For the harder instances, the compiled propagator tree more than repays the overhead of its generation and compilation compared to Min. For example instance 10 was solved in 187 s by the Min implementation and 147 s (169 s when including the cost of building the propagator tree) with propagator trees.

5.3. Case study: low autocorrelation binary sequences

The Low Autocorrelation Binary Sequence (LABS) problem is described by Gent and Smith [20]. The problem is to find a sequence s of length n of symbols $\{-1, 1\}$. For each interval $k \in \{1 \dots n-1\}$, the correlation C_k is the sum of the products $s[i] \times s[i+k]$ for all $i \in \{0 \dots n-k-1\}$. The overall correlation C_{min} is the sum of the squares of all C_k: $C_{min} = \sum_{k=1}^{n-1} (C_k)^2$. C_{min} must be minimised.

The sequence is modelled directly, using variables $s[n] \in \{-1, 1\}$. For each $k \in \{1 \dots n-1\}$, and each $i \in \{0 \dots n-k-1\}$, we have a variable $p_k^i \in \{-1, 1\}$ and the `product` constraint $p_k^i = s[i] \times s[i+k]$. For each $k \in \{1 \dots n-1\}$ we have a

Table 4
Results on LABS problems of size 25–30. All times are a mean of 5 runs.

n	Time (s)							
	Propagator tree		Product	Light-table	Table	MDDC	Reglr	STR2+
	Compiled	VM						
25	**9.22**	10.03	11.57	22.42	20.06	18.49	47.13	14.00
26	**18.85**	20.46	21.95	42.00	44.45	41.88	103.49	28.88
27	42.21	**40.88**	49.72	87.68	84.53	92.12	233.94	59.81
28	**81.44**	82.29	95.62	196.64	173.16	167.16	437.02	114.85
29	**131.40**	136.42	170.72	317.48	283.93	291.70	701.04	220.60
30	**237.06**	239.63	276.64	539.49	502.08	535.10	1199.77	348.53

n	Search nodes		Node rate (per s)		
	All GAC methods	Product	Propagator tree		Product
			Compiled	VM	
25	206,010	350,119	22,344	20,541	30,260
26	404,879	709,228	21,481	19,788	32,309
27	790,497	1,343,545	18,726	19,335	27,022
28	1,574,100	2,684,883	19,328	19,129	28,077
29	2,553,956	4,441,023	19,437	18,722	26,014
30	4,120,335	7,118,749	17,381	17,194	25,733

variable $C_k \in \{-n \ldots n\}$. C_k is constrained to be the sum of p_k^i for all i. There are also variables $C_k^2 \in \{0 \ldots n^2\}$, and a binary `lighttable` constraint is used to link C_k and C_k^2. Finally we have $C_{min} = \sum_{k=1}^{n-1} C_k^2$, and C_{min} is minimised. Gent and Smith identified 7 symmetric images of the sequence [20]. For each symmetric image we post one `lexleq` (lexicographic ordering) constraint to break the symmetry. Gent and Smith also proposed a variable and value ordering that we use here.

There are more ternary product constraints than any other constraint in LABS. C_k is a sum of products: $C_k = (s[0] \times s[k]) + (s[1] \times s[k+1]) + \cdots$. To test propagator trees on this problem, we combine pairs of product constraints into a single arity 5 constraint: $(s[i] \times s[k+i]) + (s[i+1] \times s[k+i+1]) = p_k^i$. This allows almost half of the p_k^i variables to be removed. When there are an odd number of products, one of the original product constraints is retained for the largest value of i.

We compare eight models of LABS: *Product*, the model with ternary product constraints; *Propagator tree*, where the new 5-ary constraint has a propagator tree, and this is either compiled or executed in the VM; *Table*, *Lighttable*, *MDDC* and *STR2+* where the 5-ary constraint is implemented with a generic propagator using a table with 16 satisfying tuples; and *Regular* [17] which has 10 states and uses a ternary table constraint (representing the transition table) with 17 satisfying tuples. All models except *Product* enforce GAC on the 5-ary constraint. All other constraints are the same for all eight models.

As shown in Table 2, the propagator tree was generated by GenTree in 0.32 s. The algorithm explored 621 nodes and the tree has 372 nodes. It was compiled in 20.89 s.

Table 4 shows our results for LABS sizes 25 to 30. The instances were solved to optimality. The Propagator Tree, Table, Lighttable, MDDC, Regular and STR2+ models search the same number of nodes as each other, and exhibit stronger propagation than Product, but their node rate is lower than Product in all cases. The generic GAC propagator (and Regular decomposition) models are slower than Product. However, both propagator tree variants are faster than Product, and for the larger instances it more than repays the overhead of compiling the specialised constraint.

The virtual machine also performs better than might be expected, almost matching the speed of the compiled propagator tree while saving the compilation time.

5.4. Case study: maximum density oscillating life

Conway's Game of Life was invented by John Horton Conway. The game is played on a square grid. Each cell in the grid is in one of two states (*alive* or *dead*). The state of the board evolves over time: for each cell, its new state is determined by its previous state and the previous state of its eight neighbours (including diagonal neighbours). *Oscillators* are patterns that return to their original state after a number of steps (referred to as the *period*). A period 1 oscillator is named a *still life*.

Various problems in Life have been modelled in constraints. Bosch and Trick considered period 2 oscillators and still lifes [21]. Smith [22] and Chu et al. [23] considered the maximum-density still life problem. Here we consider the problem of finding oscillators of various periods. We use simple models for the purpose of evaluating the propagator generation technique rather than competing with the sophisticated still-life models in the literature. However, to our knowledge we present the first model of oscillators of period greater than 2.

The problem of size $n \times n$ (i.e. live cells are contained within an $n \times n$ bounding box at each time step) and period p is represented by a 3-dimensional array of Boolean variables $b[n+4, n+4, p]$ indexed (from 0) by position i, j and time step t. To enforce the bounding box, for each t, the rows 0, 1, $n+2$ and $n+3$ are set to 0. Similarly, columns 0, 1, $n+2$

Table 5
Time to solve to optimality for propagator tree, sum and MDDC implementations of the Life constraint.

n	p	Time (s)				Nodes
		Propagator tree		Sum	MDDC	
		Compiled	VM			
5	2	0.02	0.04	0.08	0.18	1166
5	3	0.11	0.17	0.39	0.66	5489
5	4	0.53	0.71	2.36	3.43	21,906
5	5	1.47	2.38	6.80	12.99	49,704
5	6	3.08	4.46	13.79	17.82	71,809
6	2	0.17	0.28	0.68	1.09	13,564
6	3	1.20	1.85	5.76	10.46	88,655
6	4	14.90	23.66	78.30	119.11	837,541
6	5	189.48	266.26	934.89	1413.19	6,172,319
6	6	618.86	1139.67	3269.44	5334.13	16,538,570
7	2	2.46	3.68	11.43	17.01	260,787
7	3	22.14	39.90	128.77	202.23	1,843,049
7	4	454.26	679.37	2175.51	4416.74	28,194,835
7	5	13,376.00	21,314.90	70,910.76	timeout	564,092,290
7	6	timeout	timeout	timeout	timeout	–

and $n+3$ are set to 0. For a cell $b[i,j,t]$ at time step t, the liveness of its successor $b[i,j,(t+1) \bmod p]$ is determined as follows. The 8 adjacent cells are summed: $s = \sum \text{adjacent}(b[i,j,t])$, and the transition rules are as follows:

- $(s > 3 \vee s < 2) \Rightarrow b[i,j,(t+1) \bmod p] = 0$;
- $(s = 3) \Rightarrow b[i,j,(t+1) \bmod p] = 1$; and
- $(s = 2) \Rightarrow b[i,j,(t+1) \bmod p] = b[i,j,t]$.

We refer to the grid at a particular time step as a *layer*. For each pair of layers, a `watchvecneq` (vector not-equal) constraint is used to constrain them to be distinct. To break some symmetries, the first layer is constrained to be lex less than all subsequent layers. Also, the first layer may be reflected horizontally and vertically, and rotated 90 degrees, so it is constrained to be lex less or equal than each of its 7 symmetric images. To find oscillators of maximum density, the number of dead cells in the first layer is summed to a variable m which is then minimised.

The liveness constraint involves 10 Boolean variables. As shown in Table 2, GenTree takes 8.26 s. The algorithm explored 86,685 nodes, and the resulting propagator tree has 26,524 nodes. Compilation took 4054.17 s.

The propagator tree is compared to six other implementations. The *Sum* implementation adds an auxiliary variable $s[i,j,t] \in 0 \ldots 8$ for each $b[i,j,t]$, and the sum constraint $s[i,j,t] = \sum \text{adjacent}(b[i,j,t-1])$. $s[i,j,t]$, $b[i,j,t-1]$ and $b[i,j,t]$ are linked by a ternary table (`lighttable`) constraint encoding the liveness rules. The *Table*, *Lighttable*, *MDDC* and *STR2+* implementations simply encode the arity 10 constraint using a table with 512 satisfying tuples. The *Regular* implementation [17] has 18 states and uses a ternary table constraint (representing the transition table) with 35 satisfying tuples.

We used instances with parameters $n \in \{5,6,7\}$ and period $p \in \{2,3,4,5,6\}$. Results are shown in Tables 5 and 6. All five generic GAC methods are shown in Tables 6 and 5 includes only the best generic GAC method (MDDC). In 13 cases, the instances timed out after 24 hours, but otherwise they were solved to optimality. All models explored the same number of nodes in all cases (node counts are slightly different to those we reported previously [15] because a different optimisation function was used).

The propagator tree is substantially faster than the sum implementation. For instance $n = 7$ $p = 5$, Compiled is 5.3 times faster than Sum. Also, Sum is consistently faster than MDDC. For the six hardest instances that were solved ($n = 6$, $p \in \{4,5,6\}$, and $n = 7$, $p \in \{3,4,5\}$), the VM more than paid back its 8.26s overhead compared to Sum. For the most difficult solved instance ($n = 7$, $p = 5$) the compiled propagator tree more than paid back its overhead of 4062 s (GenTree plus compilation). Furthermore, note that the propagator tree is identical in each case: that is the arity 10 constraint is independent of n and p since it depends only on the rules of the game. Therefore the overhead can be amortised over this entire set of runs, as well as any future problems needing this constraint. We can conclude that the propagator tree is the best choice for this set of instances, and by a very wide margin.

6. Symmetry in propagator trees

We have described a technique for generating a propagator which runs in polynomial time for any constraint, at the cost of exponential pre-processing time, and exponential space complexity. The pre-processing cost can be amortised over all uses of the constraint, but the space complexity is relevant whenever the constraint is used. If this grows larger than the

Table 6
Time to solve the Life problem to optimality with each generic propagator and the Regular decomposition.

n	p	Time (s)				
		Lighttable	Table	MDDC	Regular	STR2+
5	2	0.18	0.21	0.18	1.42	0.15
5	3	1.17	1.12	0.66	8.82	0.68
5	4	7.03	6.12	3.43	56.87	3.28
5	5	24.28	18.91	12.99	192.83	11.44
5	6	47.18	27.99	17.82	356.05	19.86
6	2	1.99	1.95	1.09	16.07	0.99
6	3	19.70	18.82	10.46	155.44	9.72
6	4	256.49	200.19	119.11	2018.75	128.54
6	5	2835.33	2075.39	1413.19	22,051.68	1415.23
6	6	10,668.74	7500.97	5334.13	timeout	5273.69
7	2	34.00	40.08	17.01	264.80	16.23
7	3	448.30	364.75	202.23	2714.52	217.45
7	4	7695.24	6443.26	4416.74	59,845.68	4845.00
7	5	timeout	timeout	timeout	timeout	timeout
7	6	timeout	timeout	timeout	timeout	timeout

physical memory of the computer being used the speed of the propagator drops dramatically, so this is often the limiting factor.

In all three of the case studies above, the constraint has symmetry. For example, in Maximum Density Oscillating Life, the eight variables representing the neighbours of the cell may be permuted freely without changing the semantics of the constraint. There is potential to save both pre-processing time and reduce the space complexity by merging symmetric subtrees of the propagator trees.

While the technique of merging identical subtrees to compress a tree is well known, merging symmetric subtrees is novel to the best of our knowledge, and requires an extension of an existing group-theoretic algorithm [24]. This extended algorithm is implemented in the GAP computational algebra system [25].

The use of symmetry can reduce an exponential size propagator tree to polynomial size when the constraint is highly symmetric. In this section we present the necessary group theory background and algorithms to be able to identify symmetric subtrees. In the section that follows we adapt the GenTree algorithm to generate symmetry-reduced trees.

6.1. Group theory background

Generating symmetry-reduced trees requires a number of concepts from group theory. These are given in brief below. For a more in-depth discussion of group theory, see [26].

Definition 6. Given a set S, a *permutation* of S is a bijective function on the members of S. Given two permutations f and g, $(f.g)(x) = g(f(x))$. A *group* G on S is a set of permutations of S which contains the identity function e and satisfies the conditions $f, g \in G \rightarrow f.g \in G$ and $f \in G \rightarrow f^{-1} \in G$. Following traditional group theory notation, we denote the image of $s \in S$ under a permutation g as s^g.

For convenience, given a permutation g of S and a set $T \subseteq S$, we define $T^g = \{t^g \mid t \in T\}$. Also we define $\langle A_1, \ldots, A_n \rangle^g = \langle {A_1}^g, \ldots, {A_n}^g \rangle$.

The h *conjugate* of a group G, denoted G^h, is the group consisting of the elements $\{h^{-1}.g.h \mid g \in G\}$. The *stabiliser* of a set S in a group G, denoted $\mathrm{stab}(G, S)$, is the subgroup of G consisting of the members $\{g \in G \mid S^g = S\}$. Stabilisers for other objects are defined in the same way. Stabilisers are always themselves groups [26].

To generate symmetry-reduced trees, we need a way of finding if there exists a permutation which maps one subtree to another. This could be done by comparing all possible pairs of subtrees, but it is more efficient to use a canonicalising function, defined in Definition 7.

Definition 7. Given a group G on a set S, a *canonicalising function* $f : T \rightarrow G$ is a function which satisfies the property that for all t_1, t_2 in T, if there exists g in G such that ${t_1}^g = t_2$, then the permutations $g_1 = f(t_1)$ and $g_2 = f(t_2)$ have the property that ${t_1}^{g_1} = {t_2}^{g_2}$. The *canonical image* of $t \in T$ is $t^{f(t)}$.

We use the letter T in this definition to represent any set where the appropriate operation is defined: permutations $g \in G$ can be applied to members of T. Note that our canonicalising function returns a group element rather than the image. It is trivial to obtain the image given the group element, but not vice versa.

Example 3. Consider the group G of all permutations on $S = \{1, 2, 3, 4, 5\}$. Suppose we need a canonicalising function for subsets of S. One such canonicalising function f maps a set of size n to the set $\{1 \dots n\}$. Suppose we have sets $S_1 = \{1, 3, 5\}$ and $S_2 = \{1, 4, 5\}$. $f(S_1)$ must map the values $\{1, 3, 5\}$ to $\{1, 2, 3\}$ in some order, and $\{2, 4\}$ to $\{4, 5\}$ in some order. One suitable $f(S_1)$ is $\{1 \mapsto 3, 2 \mapsto 5, 3 \mapsto 1, 4 \mapsto 4, 5 \mapsto 2\}$. Similarly, one suitable $f(S_2)$ is $\{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 5, 4 \mapsto 3, 5 \mapsto 2\}$. The important fact is that ${S_1}^{f(S_1)} = {S_2}^{f(S_2)} = \{1, 2, 3\}$.

The reason to use a canonicalising function is that we can store the canonical image of every subtree, and know that there exists a permutation from one subtree to another within G iff they have the same canonical image. The canonicalisation function we use is not specific to propagator trees, it acts on a sequence of objects. It is developed in Appendix A.

6.2. Symmetries of constraints

The propagator trees created by the algorithm GenTree (Algorithm 3) can be executed in $O(nd)$ time, where n is the arity of the constraint, and d is the domain size. However they have the disadvantage that they can have $O(2^{nd})$ nodes. In this section we show how to generate symmetry-reduced trees, and that they can be much more compact than standard propagator trees. In particular, for some constraints (and associated symmetry groups) the space required is polynomial in n and d rather than exponential. First we must define symmetry of both assignments and constraints.

Definition 8. Consider a total assignment A to a set of variables X, and a permutation g of the literals of X. The image of A under g (denoted A^g) is defined iff applying g pointwise to A (i.e. applying g to each literal in A separately) produces another total assignment of X. In this case A^g is defined as the total assignment generated by the pointwise image of A under g.

This definition ensures that a total assignment is mapped to another total assignment, thus for any two literals from A, their images in A^g may not refer to the same variable.

Definition 9. Consider a constraint c and a permutation g of the literals of variables in $scope(c)$. c^g is defined iff A^g is defined for each assignment A that satisfies c. In this case, c^g is defined as the constraint with the same scope as c that is satisfied by the set $\{A^g \mid A \text{ satisfies } c\}$. g is a *symmetry* of c iff $c^g = c$. G is a *symmetry group* of c iff $\forall g \in G . c^g = c$.

Cohen et al. [27] surveyed definitions of symmetry for CSP, and gave two precise definitions, *solution symmetry* and *constraint symmetry*. If we define a CSP containing only one constraint and only the variables in its scope, then our Definition 9 is identical to solution symmetry, but not identical to constraint symmetry.

In some cases, our tree generation algorithm will not work correctly with the whole group G as defined above. To avoid this problem, we allow permutations $g \in G$ that permute variables, and permute values within the domains, but not that map two literals of the same variable onto two different variables. More precisely, each $g \in G$ must have the following property.

Definition 10. Given constraint c, a permutation g is *variable-stable* iff, given two literals $\langle x, d_1 \rangle$, $\langle x, d_2 \rangle$ from the same variable, then $g(\langle x, d_1 \rangle)$ and $g(\langle x, d_2 \rangle)$ are also literals from the same variable.

6.3. Symmetries of propagator trees

Examining Algorithm 3, we see that each node of the tree is generated from 3 pieces of information. The constraint being propagated (which is fixed), the set of literals which are known to be present, called *ValsIn*, and those literals that are not known to be deleted, called *SD* (for subdomain list). Note that $ValsIn \subseteq SD$ at all times.

Definition 11. The *node-state* of a tree node S comprises S_{ValsIn} and S_{SD}. The constraint being propagated is implicit. The image of S under permutation g is S^g where $S^g_{ValsIn} = \{\langle x, a \rangle^g \mid \langle x, a \rangle \in S_{ValsIn}\}$, and $S^g_{SD} = \{\langle x, a \rangle^g \mid \langle x, a \rangle \in S_{SD}\}$.

To apply a symmetry $g \in G$ to a propagator tree we define an image function in Definition 12.

Definition 12. Given a propagator tree T defined on constraint c and a literal permutation $g \in G$, then T^g is defined recursively as follows:

$$(Nil)^g = Nil$$

$$T^g = \left\langle (T.Prune)^g, (T.Test)^g, (T.Left)^g, (T.Right)^g \right\rangle$$

The group element g is applied pointwise to *Prune* and *Test*, and the image function is applied recursively to the *Left* and *Right* subtrees.

Theorem 2 shows an important, but very simple, result relating the images of trees under a permutation. This theorem does *not* require that the permutation is a symmetry of the constraint, as it applies the permutation to both the constraint and the propagator tree. This result is almost self-evident, as it performs a simple relabelling. However, it is the basis for all the symmetric tree results we will build.

Theorem 2. *Given a propagator tree T generated for a constraint c and node-state S, and given any variable-stable permutation g, T^g is a propagator tree for constraint c^g and node-state S^g.*

Proof. The proof of this theorem follows simply from the definition of these concepts. A variable-stable permutation can be seen as a simple relabelling of the variable names, and the values in the domain of each variable. As these labels are unimportant, this simple relabelling has no effect on the correctness of T^g for c^g and S^g. □

Corollary 2. *Given a propagator tree T generated for a constraint c and node-state S, and given any variable-stable permutation g which is a symmetry of c, T^g is a propagator tree for constraint c and node-state S^g.*

Proof. Follows trivially from Theorem 2, and the fact that $c^g = c$ as g is a symmetry of c. □

Corollary 2 is the basis of our approach. When generating a propagator tree, if the current node-state S' is symmetric to some previously seen node-state S, then instead of generating a propagator tree for S', we can re-use the propagator tree built for S.

6.4. Constraint symmetries and variable-stability

All constraints we use in our experiments have only variable-stable symmetries. However constraint symmetries that are not variable-stable do occur, particularly in problems involving allDifferent constraints. Consider the following example.

Example 4. Let x_1, x_2, x_3 be variables with domain $\{1, 2, 3\}$ and let g be the permutation that maps $x_i \mapsto j$ to $x_j \mapsto i$ for all $i, j \in \{1, 2, 3\}$. The constraint $c = \text{allDifferent}(x_1, x_2, x_3)$ has the symmetry g.

Theorem 2 (the critical proof of this paper) relies on the permutation g being variable-stable. This raises the question of whether variable stability is required. Example 5 demonstrates that applying permutations that are not variable-stable can lead to invalid propagators.

Example 5. Consider the symmetry in Example 4. We will create a GAC propagator tree for constraint c. Recall that propagator trees are never invoked on a search state with an empty domain (Definition 4). We construct a propagator tree that first branches for each value of x_1. In the case where the domain of x_1 is empty, the tree performs no deletions and returns (this case will never be reached). In all other cases the propagator performs GAC.

However, if we applied the symmetry in Example 4 to it, it would branch on literals $x_1 \mapsto 1$, $x_2 \mapsto 1$ and $x_3 \mapsto 1$ first. Suppose x_1, x_2 and x_3 were all assigned the value 3, the propagator would perform no deletions and return. This is clearly incorrect.

To avoid this problem, throughout the rest of this paper we consider only variable-stable permutations.

7. Generating and executing symmetry-reduced propagator trees

We can adapt GenTree (Algorithm 3) to generate symmetry-reduced trees using the canonicalisation function. Suppose we are part-way through generating a propagator tree, and we reach a node-state S. Suppose also that S will be an internal node in the completed tree. We compute the canonical image of S, and check if any other node-state with an identical canonical image has already been seen. If not, then we carry on as before. If so, we generate a new type of node that performs a jump to the previously seen symmetric node-state. Each jump has a permutation of the literals associated with it.

The other key ingredient is that when a symmetry-reduced tree is executed a permutation of the literals is maintained. The domains are viewed and pruned through the lens of this permutation, and it is updated when a jump is performed.

First we give the algorithm for generating the symmetry-reduced trees, then discuss the symmetry groups that may be used and bounds on the size of the trees. Following that we discuss efficient execution of symmetry-reduced trees in Section 7.3.

Algorithm 4 GenTreeSym(c, SD, $ValsIn$)

```
1: if entailed(c, SD) then
2:     return Nil
3: Deletions ← Propagate(c, SD)
4: SD′ = SD \ Deletions
5: if all domains in SD′ are empty then
6:     return T = ⟨Prune = Deletions, Test = Nil, Left = Nil, Right = Nil⟩
7: ValsIn* ← ValsIn \ Deletions
8: ValsIn′ ← ValsIn* ∪ {(x, a)|(x, a) ∈ SD′, |SD′(x)| = 1}
9: if SD′ = ValsIn′ or entailed(c, SD′) then
10:    if Deletions = ∅ then
11:        return Nil
12:    else
13:        return T = ⟨Prune = Deletions, Test = Nil, Left = Nil, Right = Nil⟩
   {Check if a symmetric node-state has already been generated}
14: ⟨g, CanImage⟩ ← CanSym([SD′, ValsIn′])
15: if CanonicalLookup contains key CanImage then
16:    if DeletedLookup contains CanImage then
17:        if Deletions ≠ ∅ then
18:            return T = ⟨Prune = Deletions, Test = Nil, Left = Nil, Right = Nil⟩
19:        else
20:            return Nil
21:    ⟨h, id⟩ ← CanonicalLookup[CanImage]
22:    return T = ⟨Prune = Deletions, Perm = g.h^−1, Node = id⟩
       {g.h^−1 maps from id to the current node.}
23: else
24:    CanonicalLookup[CanImage] ← ⟨g, getNewUniqueId()⟩
   {Pick a variable and value, and branch}
25: (y, l) ← heuristic(SD′ \ ValsIn′)
26: LeftT ← GenTreeSym(c, SD′, ValsIn′ ∪ (y, l))
27: RightT ← GenTreeSym(c, SD′ \ {(y, l)}, ValsIn′)
28: if LeftT = Nil And RightT=Nil And Deletions = ∅ then
29:    Add CanImage to DeletedLookup
30:    return Nil
31: else
32:    return T = ⟨Prune = Deletions, Test = (y, l), Left = LeftT, Right = RightT⟩
```

7.1. Generating symmetry-reduced trees in detail

Recall that the node-state consists of *ValsIn* and *SD* (Definition 11). In the new algorithm, we maintain the following two data structures to track node-states seen so far.

CanonicalLookup[c] — Hash table indexed by canonical image c, containing a pair $\langle g, id\rangle$ where $g \in G$ is a permutation mapping a node-state S to c, and id is a number that identifies the node where S was seen.

DeletedLookup — Set (implemented as a hash table) containing canonical images. When a tree node is deleted because it (and the subtree beneath it) contains no prunings, the canonical image of it is stored in *DeletedLookup*.

CanonicalLookup contains the canonical image of every node-state seen so far. Thus it allows us to efficiently check if the current node-state is symmetrically equivalent to any previous node-state. Also, it allows us to compute a permutation from the current node-state to the previous node-state via their shared canonical image.

The reason for *DeletedLookup* is more subtle. When a tree node is deleted because it contains no prunings (lines 17 and 18 of GenTree) it could be removed from *CanonicalLookup*, and this would prevent a jump being inserted to the deleted node. However, a tree node can only be deleted in this way after the subtree beneath it has been explored (potentially a time consuming process) and this work would be repeated if we reached a symmetric node-state in the future. Therefore we retain the deleted node in *CanonicalLookup*, and also insert it in *DeletedLookup*.

Algorithm 4 (GenTreeSym) is similar in structure to GenTree. Lines 14–24 and 29 have been added, and the function name on lines 26 and 27 has been changed. Other lines in GenTreeSym are identical to GenTree.

In the new section the first task is to compute the canonical image of the current node-state. This is done by calling CanSym which encodes the node state as a sequence of sets of integers, calls CanonicalSetList (Algorithm 7 in Appendix A) and returns both the canonicalising permutation g and the canonical image *CanImage*. *CanImage* is then looked up in *CanonicalLookup*. If it is not present, it is added (line 24) and the algorithm continues as GenTree would. If the canonical image is in *CanonicalLookup*, the algorithm branches for three cases. It makes at most one new node, either containing a jump or some deletions.

One important point is that we calculate the canonical image *after* pruning domains. This means that a node found in *CanonicalLookup* is only symmetric after deletions have been applied. Hence, on line 18, the algorithm discovers that the current node-state is symmetric to a previously deleted node, but the current node must perform the pruning so it cannot be deleted. Also, on line 22, the deletions are stored with (and performed before) the permutation and jump.

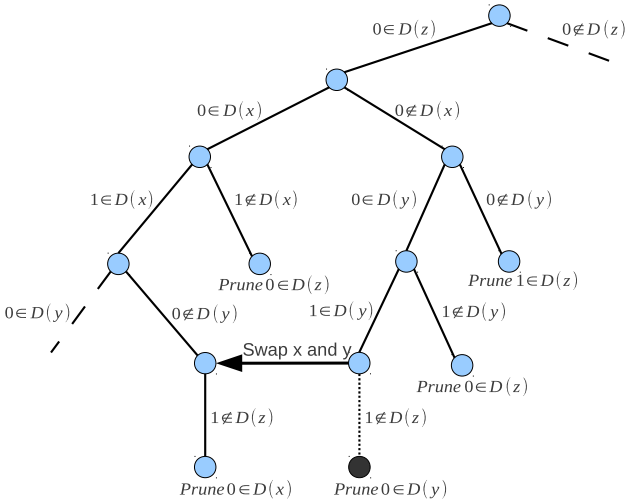


Fig. 3. Example of part of a propagator tree for the constraint $x \wedge y \Leftrightarrow \neg z$. The arrow indicates a jump with the permutation that swaps variables x and y. The black node would not be included in the symmetry-reduced tree. Prior to the jump, we have $ValsIn' = \{x \mapsto 1, y \mapsto 0, y \mapsto 1, z \mapsto 0\}$ and $SD' = ValsIn' \cup \{z \mapsto 1\}$. After the jump, we have $ValsIn' = \{x \mapsto 0, x \mapsto 1, y \mapsto 1, z \mapsto 0\}$ and $SD' = ValsIn' \cup \{z \mapsto 1\}$.

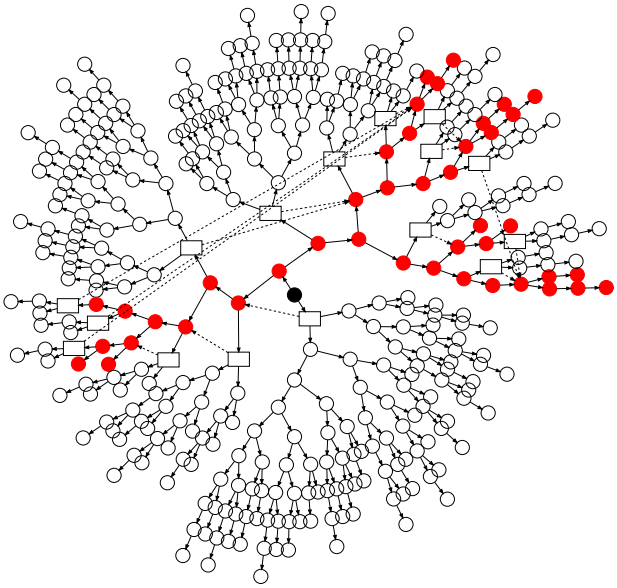


Fig. 4. Propagator tree for LABS 2 constraint (arity 5). The solid black node is the root. Red nodes are shared between the standard and symmetry-reduced trees. Rectangular nodes show where a normal node was replaced with a symmetric jump. Dashed arrows are symmetric jumps. Unfilled circular nodes are included in the standard propagator tree only. The standard propagator tree has 372 nodes and the symmetry-reduced tree has 60. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Fig. 3 shows an example of a symmetry-reduced propagator tree created by GenTreeSym with the entailment heuristic. Fig. 4 illustrates the difference made by symmetry reduction on the LABS 2 constraint.

7.2. Bounds on tree size

The sole reason for exploiting symmetry is to reduce the size of the generated trees. In this section we will show that for a wide range of constraints, symmetry-reduced trees achieve a polynomial bound in tree size, where standard propagator trees are exponential. We will consider one class of symmetric constraints, given in Definition 13.

Definition 13. A partially symmetric constraint defined by the parameters $\langle n_1, d_1, n_2, d_2 \rangle$ is a constraint with $n_1 + n_2$ variables, where the first n_1 variables have domain size d_1 and the last n_2 variables have domain size d_2. Further, the constraint has, for each $i, j \in \{1 \ldots n_1\}$, the symmetry that swaps the assignment to variables i and j, leaving all other variables unchanged.

Of the constraints we consider in our experiments below, all three variants of Life fit Definition 13, with $n_1 = 8$ and $n_2 = 2$. All the symmetry of Life and Brian's Brain is captured in that definition. Peg Solitaire also fits the definition with $n_1 = 3$. In the experiments we exploit more symmetries, such as permuting values, that further reduce the tree size. Lemma 4 gives a simple bound on the number of equivalence classes of node-states a partially symmetric constraint can have.

Lemma 4. *Given a partially symmetric constraint defined by the parameters $\langle n_1, d_1, n_2, d_2\rangle$, there are $O((3^{d_2} - d_2 - 1)^{n_2}.(n_1+1)^{(3^{d_1}-d_1-1)})$ equivalence classes of node-states (Definition 11).*

Proof. A variable with d domain values has 3^d states, because there are 3 values a literal can have, either known present, known not present, or unknown. We discount the state where all values are not present, because we assume the propagator is never invoked for such domains. Also we discount the d states where all but one literal are known not present, and the remaining literal is unknown, because we know that at least one literal must be present. Therefore a variable of domain size d has $3^d - d - 1$ possible states.

Consider the n_1 symmetric variables. As the order of these variables is unimportant, we can fully characterise each equivalence class by the number of symmetric variables it contains of each $3^{d_1} - d_1 - 1$ possible state, giving a bound of $(n_1+1)^{(3^{d_1}-d_1-1)}$. This bound is a loose approximation but is sufficient to show that the number of equivalence classes is polynomial in n_1 when d_1 is fixed.

The number of states of any one of the n_2 asymmetric variables can take is $3^{d_2} - d_2 - 1$. Therefore the number of states of the asymmetric variables is simply $(3^{d_2} - d_2 - 1)^{n_2}$. Therefore the total number of equivalence classes of node-states is $O((3^{d_2} - d_2 - 1)^{n_2}.(n_1+1)^{(3^{d_1}-d_1-1)})$. □

Lemma 4 does not directly give a bound on the size of the symmetry-reduced tree, because a tree can contain multiple nodes belonging to one equivalence class. The first of these nodes has a subtree beneath it, and the rest of them have a jump to the first.

Lemma 5. *Suppose a constraint c (with symmetry group G) has e equivalence classes of node states. The number of nodes of a symmetry-reduced tree for c is $O(e)$.*

Proof. Given the symmetry-reduced tree T for c and G, remove all symmetric jumps from the tree to form the labelled binary tree T'. In T', the nodes corresponding to jump nodes in T are now leaf nodes. For each equivalence class, there can be at most one interior node belonging to the class because any other node in the class must be a leaf node in T' (and a jump node in T). Therefore there are at most e interior nodes, and at most $2e + 1$ nodes in total. □

The lemma above gives us a bound on the symmetry-reduced tree size which is polynomial in n_1 and exponential in n_2. This can be compared to the bound of $O(2^{nd})$ derived in Section 4.1.

7.2.1. A tighter bound given branching restrictions

While we have shown that using symmetry-reduced trees can, in highly symmetric constraints, produce a polynomial bound in tree size, these polynomials can be extremely large. For example, for a constraint with total variable symmetry and variables of domain size 3 the upper bound is $O(n^{23})$. In this section we will substantially tighten this bound.

In order to find a tighter bound, we restrict the branching order. We choose a variable x, and branch only on literals of x until we have complete knowledge of the domain of x. This is similar to enumeration branching (also known as d-way branching) in CP search [1] (4.2), however we are still performing 2-way branches.

In order to prove this result, we first derive a bound with true enumeration branching. This is performed by selecting a variable, and branching for each variable state. For a variable with domain size d, there will be $2^d - 1$ non-empty subdomains therefore at most $2^d - 1$ branches.

Lemma 6. *Given enumeration branching, there are $O((2^{d_2})^{n_2}(n_1+1)^{2^{d_1}})$ equivalence classes of node-states of a partially symmetric constraint with parameters $\langle n_1, d_1, n_2, d_2\rangle$.*

Proof. There are clearly $2^d - 1$ non-empty subdomains for a variable of domain size d. While we may deduce that some literals in variables not yet branched on are either in or out by GAC propagation, two node-states which are equivalent before GAC will be equivalent after GAC, therefore we can treat the domains of variables we have not branched on as completely unknown for the purpose of counting equivalence classes.

Including the completely unknown state, each variable has 2^d states. We can apply the same reasoning as Lemma 4 to show that there are

$$O\left(\left(2^{d_2}\right)^{n_2}(n_1+1)^{2^{d_1}}\right)$$

equivalence classes of node-states. □

Suppose the number of equivalence classes is e. Using a similar argument to Lemma 5, we can show that the number of interior (non-jump, non-leaf) nodes is e, therefore the total number of nodes is $(2^d - 1)e + 1$ (where d is the maximum of d_1 and d_2).

Now we must convert the result to binary trees. For each node with t children, we convert it to $t - 1$ nodes by branching on each value in the domain in turn. We call this *whole-variable branching*. For an enumeration tree with $(2^d - 1)e + 1$ nodes and a branching factor of $2^d - 1$ we have $(2^d - 1) \times ((2^d - 1)e + 1) - 1$ nodes in the binary tree. Combining this with Lemma 6 leads to the following theorem.

Theorem 3. *Given a partially symmetric constraint c defined by parameters $\langle n_1, d_1, n_2, d_2 \rangle$, the size of a symmetry-reduced tree for c that performs whole-variable branching is as follows, where $d = \max(d_1, d_2)$.*

$$O\left(2^{2d+d_2 n_2}(n_1+1)^{2^{d_1}}\right)$$

To take our example of a totally symmetric constraint with domain size 3, the bound from the previous section is $O(n^{23})$, and we have improved it to $O(n^8)$.

7.3. Execution of symmetry-reduced trees

We extend both methods of executing standard propagator trees to work with symmetry-reduced trees in the sections below.

7.3.1. Virtual machine

We extend the virtual machine described in Section 3.5.2 with two more instructions:

Perm : $\langle l_1, l_2, \ldots, l_n \rangle$ — Apply the given permutation of the literals. The number of operands is the sum of the sizes of the initial domains.
Jump : $\langle pos \rangle$ — Jump to the position given.

To perform a jump to a symmetrically-equivalent state, the instruction stream must have a Perm followed by a Jump.

When execution starts, the variable domains may be queried and pruned directly. However, after the execution jumps to a symmetric state, the instructions no longer directly relate to the variable domains. Each literal queried or pruned must be mapped through a permutation. Suppose the execution makes a second jump to a symmetric state. Now each literal queried or pruned must be mapped through two permutations (or the composition of them). We need some mechanism for storing and composing permutations as the propagator is executed. In Algorithm 5 we give the (almost trivial) algorithm to compose two permutations. It takes three references p, q and r to blocks of memory, and composes p (the currently stored permutation) with q and stores the result in r.

Algorithm 5 Permutation composition compose(p, q, r)

Require: p: Current permutation.
Require: q: New Permutation from Perm instruction.
Require: r: Storage for composed permutation.
  for $i = 1$ to length(p) **do**
    $r(i) = p(q(i))$

The most straightforward method of composing permutations begins with the identity $p(i) = i$ and a spare buffer r. Each time a new permutation q must be composed with p, we call compose(p, q, r) then copy r into p. This has a number of inefficiencies. Repeatedly copying r into q is expensive. Also, it is necessary to initialise p at the start of the algorithm. Further, all domain queries and prunings must be done through the permutation, incurring a cost even for propagator trees that do not contain any permutations.

To solve these problems, we introduce a four state finite state machine which removes many of these costs. This finite state machine is shown in Algorithm 6. This machine provides two functions. *Apply* takes an integer i and returns the image of i under the current permutation. *Update* takes a permutation reference q and updates the state accordingly.

Algorithm 6 minimises the costs of storing and applying permutation as far as possible, avoiding all copying.

The state machine above could be implemented as *Apply* and *Update* functions, each containing a switch statement. However, this would introduce a substantial inefficiency, particularly for *Apply* which is very heavily used. Instead we compile the whole virtual machine once for each of the four states. The *Apply* function for each state is now very simple and efficient, and is readily inlined. The *Update* function for each state performs the composition then jumps into a different specialisation of the virtual machine.

One particular advantage of specialising the whole VM for each of the four states is that in State 1 the *Apply* function is the identity, and the compiler is able to optimise it away. This removes all cost when a propagator tree contains no Perm instructions, therefore we use the same virtual machine for our experiments with both symmetry-reduced and standard propagator trees.

Algorithm 6 Efficient permutation storage

Local Variable: *ptr*: A pointer to a permutation.
Local Variable: P_3, P_4: Two permutations.

State 1 (Initial State)
$Apply(i) = i$
$Update(q)$: Stores a reference to q in *ptr*. Moves to state 2

State 2 (Pointer State)
$Apply(i) = ptr[i]$
$Update(q)$: Calls compose(ptr, q, P_3). Moves to state 3

State 3 (Stored State A)
$Apply(i) = P_3[i]$
$Update(q)$: Calls compose(P_3, q, P_4). Moves to state 4

State 4 (Stored State B)
$Apply(i) = P_4[i]$
$Update(q)$: Calls compose(P_4, q, P_3). Moves to state 3

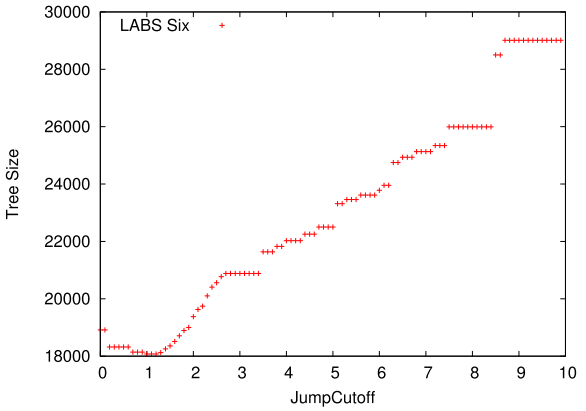


Fig. 5. Tree size of LABS Six constraint.

7.3.2. Code generation

The use of jumps in symmetry-reduced trees means we cannot use the simple nested if/then/else structure used in Section 3.5.1. Instead, we produce code that closely follows the virtual machine instructions. Each instruction becomes a block of code with a label, and Branch and Jump instructions use goto to jump to the appropriate label.

Code generation produces a very large function, therefore we compile it once and it is not specialised for the four states of the permutation state machine. The *Apply* and *Update* functions used here contain switch statements with one branch for each of the four states. This means *Apply* and *Update* are likely to be less efficient than in the VM.

7.4. Refining GenTreeSym by limiting jumping

We will see below that eliminating symmetries can greatly reduce the size of a propagator tree. However, there are situations near the leaves where the space taken to insert a jump is greater than the size of the subtree that it replaces, therefore inserting a jump will increase the size of the propagator tree. Furthermore, when the propagator tree is executed, additional jumps will slow down propagation.

To address this problem, we first assume that the representation is the virtual machine instructions given in Sections 3.5.2 and 7.3.1. This means we can calculate the size s_t of the destination subtree in terms of the number of integers in the VM instructions. We can also calculate the size s_j of the proposed jump in the same way. If $s_t < s_j$, then to insert the jump would increase the overall tree size.

We introduce a new parameter *JumpCutoff* that controls when to insert a jump. If $s_t > JumpCutoff \times s_j$ then a jump is inserted, otherwise GenTreeSym continues as GenTree would. Prior to line 22 of GenSymTree s_t and s_j are calculated, and line 22 is only executed if the condition holds, otherwise the algorithm continues at line 25.

Note that s_t is the size of the destination subtree T_1. Suppose we do not insert a jump, and instead generate a new subtree T_2. T_1 and T_2 are generated from symmetric states, so we might expect them to be the same size. However, the state of *CanonicalLookup* may have changed, therefore T_2 may be smaller. In some rare cases this means that changing *JumpCutoff* does not have the expected effect.

For values between 0 and 1 of *JumpCutoff*, we should see the size of the tree decreasing and propagation speed increasing. As *JumpCutoff* is increased above 1, the size of the tree will probably increase, and we expect that larger trees will also have faster propagation speed. When $JumpCutoff = \infty$, GenTreeSym generates exactly the same tree as GenTree. For the LABS Six

Table 7
Time taken to generate the standard and symmetry-reduced propagator trees, in Python, GAP and the C++ compiler.

	Standard Tree		Symmetry-Reduced Tree		
	Python	Compiler	Python	GAP	Compiler
LABS 2	0.32	20.89	0.80	4.54	22.48
LABS 3	1.92	98.06	1.84	8.74	25.03
LABS 4	10.32	6256.25	3.80	17.14	25.12
LABS 5	451.19		8.35	44.91	41.23
LABS 6			31.15	116.94	60.98
Brian			507.10	2241.89	
Immig			279.90	1014.00	5605.33
Life	8.26	4054.17	3.61	14.82	31.72
PegSol	0.37	21.58	0.93	5.40	24.26

constraint, and symmetry group given in Section 8.3, Fig. 5 shows the tree size for values of *JumpCutoff* from 0 to 10. This graph shows a minimum at 1.0 as expected.

For all our experiments we use $JumpCutoff = 1$ to obtain the smallest (in the VM representation) symmetry-reduced trees.

7.5. Complexity of execution of symmetry-reduced trees

To find the complexity we need the set $ValsMaybe = SD \setminus ValsIn$. This set has the property that its size is monotonically reduced as the tree is executed. Each branch reduces *ValsMaybe* by one literal, whether the literal is in or out of domain. Deletions may reduce the size of *ValsMaybe*. Jumps potentially change the literals in *ValsMaybe* but not its size. We also need to observe that a jump cannot take us to a node with another jump instruction, because jump nodes are not entered in the *CanonicalLookup* table in Algorithm 4, and jump destinations are always taken from *CanonicalLookup*.

We use the size of *ValsMaybe* as our measure of progress. At the root node the size is at most nd, therefore in an execution path we have at most nd nodes where we branch, plus one leaf node. We also have up to nd jump nodes, because there are at most nd destinations.

To perform $O(nd)$ branches has a cost of $O(nds)$, where s is the cost of testing whether a value is in the domain. Performing $O(nd)$ permutation applications and jumps has a cost of $O(n^2d^2)$. The cost of deleting literals is less straightforward. We use r for the cost of deleting a single literal. When we perform a jump, the destination node may delete literals that have already been deleted. Since we have at most $2nd + 1$ nodes and trivially $O(nd)$ deletions at each node, the cost of deleting literals is $O(n^2d^2r)$. Combining the three gives us a total cost of $O(nds + n^2d^2 + n^2d^2r)$.

Theorem 4. *Given a solver where querying and deleting literals is $O(1)$ (such as Minion) the complexity of executing a symmetry-reduced tree is $O(n^2d^2)$.*

8. Experimental evaluation of symmetry-reduced trees

In this section we compare the scalability of symmetry-reduced trees to that of propagator trees, and also measure the overhead of exploiting symmetry when the propagator is executed. We use the same three problems as in Section 5, and also add two variants of Life, Life Immigration and Brian's Brain, both of which have three colours.

For each constraint, we have a group of permutations of the literals. To describe the group compactly we only give the group generators, therefore to obtain the full group all possible products of the generators must be added.

8.1. Time taken to generate propagators

In this section we compare the time taken to run GenTree and GenTreeSym. This is relevant for both the VM and code generation. For code generation, we report the time to compile the propagator tree and link it to Minion. These figures are shown in Table 7, and empty cells denote the computer running out of memory (>12 GiB). For GenTreeSym we have an additional column in Table 7 for group computation performed in GAP.

8.2. Case study: English Peg Solitaire

The English Peg Solitaire problem is described in Section 5.2. We generate propagators for the following constraint on boolean variables.

$$(x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4 \wedge \neg x_5 \wedge x_6) \Leftrightarrow x_7$$

The symmetry group we use is as follows: x_1, x_3 and x_6 are interchangeable, and so are x_2, x_4 and x_5. The following pairs of literals may be swapped simultaneously: $(x_1 \mapsto 0, x_2 \mapsto 1)$ and $(x_1 \mapsto 1, x_2 \mapsto 0)$ (i.e. the two variables are exchanged and the values 0, 1 are exchanged). The size of the group is 720.

Table 8
Results on peg solitaire problems.

Starting position	Node rate (per s)				
	Propagator tree				Min
	Standard		Sym-reduced		
	Compiled	VM	Compiled	VM	
1	**9046**	6663	6823	5726	7445
2	**5624**	4423	5518	4695	4714
4	**8634**	6556	6947	5547	7064
5	**8684**	6834	8139	6361	7565
9	**8827**	6536	6841	5837	6990
10	**10,076**	7727	8924	6513	7921
17	**6470**	4797	4820	4808	4702

Table 9
Results on LABS problem size 30. All times are a mean of 5 runs. For the VM, 'mem' indicates that the GenTree exceeded 12 GB memory. For the compiled variant, 'mem' indicates that either GenTree or the compiler exceeded 12 GB.

		Two	Three	Four	Five	Six
Standard	Compiled	**237.06**	257.41	275.96	mem	mem
	VM	239.63	263.81	304.77	323.51	mem
Sym-reduced	Compiled	271.55	317.21	401.15	440.06	534.38
	VM	293.64	361.15	452.21	488.00	553.80
Lighttable		539.49	940.82	1246.17	1864.01	2543.93
Table		502.08	796.60	1135.31	1549.06	2103.86
MDDC		535.10	662.38	863.96	1006.41	1173.48
Regular		1199.77				
STR2+		348.53	398.06	508.56	645.57	1086.97
Product				276.64		
Tree Size	Standard	372	4316	47,092	495,196	mem
	Sym-reduced	60	166	390	736	1336
Group size		64	768	12,288	245,760	5,898,240

The standard propagator tree has 315 nodes, and the algorithm explores 509 nodes when generating it. The symmetry-reduced tree has 94 nodes and GenTreeSym explored 121 nodes.

Table 8 shows our results for peg solitaire. We omit run times and just give node rates because all methods explore the same tree. Of the two hand-written propagators (Min and Reified Sumgeq), Min is always superior (Table 3) so we omit Reified Sumgeq from this table. We also omit Lighttable, Table, MDDC, STR2+ and Regular.

Table 8 shows very little overhead from exploiting symmetry when using the VM. However when using code generation, the overhead can be more than 25%. As we noted in Section 7.3.2, code generation has the disadvantage that the *Apply* and *Update* functions are less efficient than in the VM. Even so, code generation outperforms the VM whether or not we apply symmetry reduction.

8.3. Case study: low autocorrelation binary sequences

The Low Autocorrelation Binary Sequence problem is described in Section 5.3. In the previous experiment, we grouped pairs of product constraints to form a 5-ary constraint and reduce the number of auxiliary variables. In this experiment we combine sets of 2, 3, 4, 5 and 6 product constraints to form constraints of arity 5, 7, 9, 11 and 13. Take for example the constraint of arity 7, where the domains of $x_1 \ldots x_6$ are $\{-1, 1\}$ and the domain of x_7 is $\{-3, -1, 1, 3\}$:

$$(x_1 \times x_2) + (x_3 \times x_4) + (x_5 \times x_6) = x_7$$

The generators of the symmetry group for the arity 7 constraint are as follows. x_1 and x_2 are interchangeable, and pairs (x_1, x_2), (x_3, x_4) and (x_5, x_6) are interchangeable. x_1 and x_2 may be negated simultaneously (i.e. for both variables, swap the values -1 and 1). Finally, x_1, x_3, x_5 and x_7 may be negated simultaneously. This final generator states that if each term in the sum is negated, then the total is also negated. The symmetry group is adapted in the straightforward way to other arities.

For Lighttable, Table, MDDC and STR2+, the size of the table when grouping 2, 3, 4, 5, and 6 product constraints is 16, 64, 256, 1024 and 4096. The Regular decomposition was consistently the slowest method when grouping 2 product constraints, and so we did not extend it to 3, 4, 5 and 6.

Table 9 shows run times for the largest instance of LABS ($n = 30$), and the sizes of the propagator trees (number of nodes) for each arity. From the tree sizes we can see that exploiting symmetry allows propagator trees to scale much better.

Table 10
Results on LABS problem size 25. All times are a mean of 5 runs.

		Two	Three	Four	Five	Six
Standard	Compiled	**9.22**	9.39	11.71	mem	mem
	VM	10.03	10.35	11.47	12.38	mem
Sym-reduced	Compiled	11.97	12.41	14.64	18.04	22.85
	VM	11.02	14.59	18.74	19.65	22.25
Lighttable		22.42	38.14	53.38	80.88	114.20
Table		20.06	29.74	47.17	58.48	91.20
MDDC		18.49	26.35	30.72	35.86	43.77
Regular		47.13				
STR2+		14.00	16.05	20.45	28.12	31.11
Product				11.57		

The tree for six pairs (arity 13) with symmetry is smaller than the tree for three pairs (arity 7) without symmetry. Exploiting symmetry can reduce the tree size by orders of magnitude.

However, as the constraints are scaled up, we find that the solver becomes less efficient. This is explained by two factors. First, increasing the length of the constraints does not strengthen propagation, because the sum of products is a tree. Second, propagator trees have no incremental state and cannot exploit triggers (as described in Section 3.3). Each time they are called they start from scratch, with a bound of $O(n^2d^2)$ (when using symmetry), therefore the cost of executing a propagator tree is likely to increase as the arity increases. In contrast, the cost of the product propagator is $O(1)$, and the sum is $O(n)$.

The same pattern can be seen on the $n = 25$ instance (Table 10). For both $n = 25$ and $n = 30$, the fastest configuration is the compiled standard propagator tree, group two. Longer constraints slow the solver down substantially. The other instances $n \in \{26, 27, 28, 29\}$ also exhibit the same pattern.

Tables 9 and 10 also show that propagator trees compare well to the generic GAC propagators as the arity is increased. STR2+ is the fastest of the generic GAC propagators and it is consistently slower than all propagator tree methods.

This experiment has demonstrated that symmetry is very helpful in extending the scalability of propagator trees. However, on this particular problem, increasing the arity does not allow more powerful propagation.

8.4. Case study: maximum density oscillating life & variants

Life, and the problem of finding maximum density oscillators, is described in Section 5.4. In addition to Life, we sought related automata where the cells have three states. This allows us to scale up the number of literals in the generated constraints, and demonstrate the value of symmetry reduction.

Immigration [28] and Brian's Brain [29] are both variants of Life where the cells have three states. For both Immigration and Brian's Brain, it is not possible to generate the standard propagator tree within 12 GB memory, however it is possible to generate symmetry-reduced trees.

The Life, Immigration and Brian's Brain constraints all have the symmetry that the first eight variables (representing the neighbours) are interchangeable. In Immigration it is also possible to swap the two *alive* states for all variables simultaneously.

Life

Of the three problems, only Life can be used to compare propagator trees with symmetry-reduced trees. The Life constraint has $8! = 40{,}320$ symmetries, the standard propagator tree has 26,524 nodes and the symmetry-reduced tree has 410 nodes. Table 11 shows that the symmetry-reduced tree is less efficient than the standard tree on this problem, taking up to 3 times longer to solve to optimality. Code generation proved to be somewhat more efficient than the VM for the symmetry-reduced tree.

In the previous Life experiment we found Sum to be more efficient than any of the generic propagators and the Regular decomposition (as shown in Tables 5 and 6). The symmetry-reduced tree compares well to Sum, being approximately twice as fast for all instances.

As Table 7 shows, the overhead of generating the compiled, symmetry-reduced Life propagator is 50.15 s in total, therefore on five instances ($n = 6$, $p \in \{5, 6\}$ and $n = 7$, $p \in \{3, 4, 5\}$) that propagator tree more than pays back its overhead.

Immigration

Immigration is similar to Life, but there are two *alive* states (usually represented as two colours). When a cell becomes alive, it takes the state of the majority of the 3 neighbouring live cells that caused it to become alive. Otherwise the rules of Immigration are the same as those of Life. The Immigration constraint has the same scope as the Life constraint, but each variable has three values.

Table 11
Time to solve to optimality for standard and symmetry-reduced propagator trees on the Life problem.

n	p	Time (s)				
		Propagator tree				Sum
		Standard		Sym-reduced		
		Compiled	VM	Compiled	VM	
5	2	0.02	0.04	0.04	0.05	0.08
5	3	0.11	0.17	0.19	0.23	0.39
5	4	0.53	0.71	1.01	1.15	2.36
5	5	1.47	2.38	3.62	4.55	6.80
5	6	3.08	4.46	6.66	8.49	13.79
6	2	0.17	0.28	0.34	0.35	0.68
6	3	1.20	1.85	2.76	2.89	5.76
6	4	14.90	23.66	32.42	37.30	78.30
6	5	189.48	266.26	480.09	500.43	934.89
6	6	618.86	1139.67	1715.76	1947.89	3269.44
7	2	2.46	3.68	6.08	7.34	11.43
7	3	22.14	39.90	65.50	70.16	128.77
7	4	454.26	679.37	1195.79	1236.32	2175.51
7	5	13,376.00	21,314.90	32,022.86	38,031.06	70,910.76
7	6	timeout	timeout	timeout	timeout	timeout

Table 12
Time to solve to optimality, for each implementation of the Immigration constraint, for various values of board size n and period p.

n	p	Time (s)					
		Symmetry-reduced tree		Sum	Table	MDDC	Light-table
		Compiled	VM				
5	2	5.41	**4.27**	12.38	16.79	11.51	18.02
5	3	32.15	**25.83**	106.96	88.38	77.49	128.13
5	4	377.39	**330.28**	1781.38	1057.20	833.23	1582.97
5	5	3664.06	**3087.38**	15,940.08	7242.53	6879.83	15,373.76
5	6	12,561.54	**11,161.40**	50,838.98	22,767.08	25,032.70	56,345.80
6	2	1434.13	**1294.49**	3214.36	3909.51	2264.14	5456.36
6	3	5074.60	**4104.27**	15,084.32	13,956.02	9752.76	19,364.86
6	4	60,636.74	**50,209.10**	timeout	timeout	timeout	timeout

n	p	Time (s)	
		Regular	STR2+
5	2	68.81	46.26
5	3	483.82	419.95
5	4	5953.59	4930.51
5	5	56,861.56	56,461.66
5	6	timeout	timeout
6	2	16,048.28	5321.00
6	3	62,645.52	46,649.66
6	4	timeout	timeout

n	p	Nodes	
		GAC Methods	Sum
5	2	90,745	193,684
5	3	347,115	851,602
5	4	2,743,923	8,923,604
5	5	17,216,657	57,187,571
5	6	48,273,400	130,935,764
6	2	26,735,448	53,300,293
6	3	53,878,608	133,274,167
6	4	469,264,819	timeout

The Immigration constraint has $8! \times 2 = 80{,}640$ symmetries. It is not possible to generate the standard propagator tree within 12 GB of memory. The symmetry-reduced tree has 34,712 nodes.

For the Sum model each Immigration constraint is represented as follows. For each $b[i, j, t]$, we introduce two auxiliary variables $s_{dead}[i, j, t]$ and $s_1[i, j, t]$ both with domain $\{0 \dots 8\}$. s_{dead} is the number of dead adjacent cells, and s_1 is the number in live state 1 adjacent cells. Both are linked to the adjacent cells using an occurrence constraint. $s_{dead}[i, j, t]$, $s_1[i, j, t]$, $b[i, j, t]$ and $b[i, j, t + 1]$ are linked with a lighttable constraint encoding the liveness rules. This encoding does not enforce GAC on the original constraint.

As in previous experiments we have five generic GAC methods: Lighttable, Table, MDDC and STR2+ with a table containing 19,683 satisfying tuples, and the Regular decomposition [17] with 25 states and ternary table constraints (for the transition table) with 67 satisfying tuples.

Table 12 shows that the symmetry-reduced tree methods outperform all five generic GAC methods while exploring the same search tree. Table and MDDC are the most efficient among the five generic GAC methods, and VM outperforms both Table and MDDC by approximately two times. VM is somewhat faster than code generation on this problem. Finally, the

Table 13
Time to solve to optimality, for each implementation of the Brian's Brain constraint, for various values of board size n and period p.

n	p	Time (s)				
		Symreduced tree, VM	Sum	Table	MDDC	Light-table
6	2	0.18	**0.03**	0.20	4.37	0.22
6	3	**0.99**	1.64	3.74	7.88	5.56
6	4	**0.93**	4.09	3.88	10.22	4.72
6	5	**1.25**	8.14	5.22	13.31	7.90
6	6	**23.62**	4973.44	91.28	57.38	133.09
7	2	0.19	**0.04**	0.20	5.78	0.20
7	3	**9.97**	20.47	42.23	30.43	53.67
7	4	**4.83**	43.94	21.44	24.48	31.27
7	5	**8.04**	117.42	29.37	33.44	42.32
7	6	**635.54**	timeout	2746.48	1584.82	3885.54
8	2	0.19	**0.05**	0.20	7.55	0.21
8	3	**163.86**	445.54	697.13	334.98	926.29
8	4	**30.20**	394.76	137.12	81.85	151.18
8	5	**49.93**	2223.32	239.38	150.47	378.54
8	6	**16,698.16**	timeout	67,789.40	41,338.70	timeout

n	p	Time (s)	
		Regular	STR2+
6	2	0.08	3.61
6	3	5.15	51.68
6	4	3.74	56.98
6	5	5.94	94.99
6	6	108.44	878.82
7	2	0.10	4.58
7	3	50.42	539.35
7	4	26.64	390.13
7	5	38.58	555.07
7	6	3125.78	31,813.20
8	2	0.11	5.60
8	3	813.51	7979.88
8	4	141.52	2513.07
8	5	273.36	4728.69
8	6	82,353	timeout

n	p	Nodes	
		GAC Methods	Sum
6	2	30	30
6	3	6658	31,978
6	4	4451	68,193
6	5	5155	95,601
6	6	80,501	53,499,585
7	2	42	42
7	3	74,367	473,036
7	4	28,722	690,201
7	5	35,085	1,646,109
7	6	2,415,289	timeout
8	2	56	56
8	3	1,228,908	8,938,209
8	4	168,530	6,585,497
8	5	252,274	28,950,186
8	6	64,063,724	timeout

symmetry-reduced tree methods are substantially more efficient than the Sum model. Sum is slower per node and explores many more nodes than VM.

The total overhead of generating the VM symmetry-reduced propagator is 1293.9 s. Therefore, for instances $n = 5$, $p \in \{5, 6\}$ and $n = 6$, $p \in \{2, 3, 4\}$ it repays its overhead (even if the propagator were generated once for each instance) and remains substantially faster than the other methods. Because the constraint is the same for all instances, the cost can actually be amortised over all instances.

Brian's Brain

Brian's Brain is another variant of Life with three values: *dead*, *alive* and *dying*. If a cell is dead and has exactly two *alive* (not dying) neighbours, it will become alive, otherwise it remains dead. If a cell is alive, it is always *dying* after one time step. If a cell is *dying*, it becomes *dead* after one time step.

The Brian's Brain constraint has $8! = 40{,}320$ symmetries. It is not possible to generate the standard propagator tree for this constraint within 12 GiB of memory. The symmetry-reduced propagator tree has 135,575 nodes. This can be executed using the VM, but not by code generation (Section 7.3.2) because the compiler exceeds 12 GiB of memory.

For the Sum model each Brian's Brain constraint is represented as follows. For each $b[i, j, t]$, we introduce one auxiliary variable $s_{alive}[i, j, t]$ with domain $\{0 \ldots 8\}$. This is linked to the adjacent cells using an occurrence constraint. $s_{alive}[i, j, t]$, $b[i, j, t]$ and $b[i, j, t + 1]$ are linked with a lighttable constraint encoding the liveness rules. This encoding does not enforce GAC on the original constraint.

As for Immigration we have five generic GAC methods: Lighttable, Table, MDDC and STR2+ with a table containing 19,683 satisfying tuples, and the Regular decomposition [17] with 11 states and ternary table constraints (the transition table) with 27 satisfying tuples.

Table 13 shows our results. In the case of Brian's Brain, the Sum encoding performs particularly badly. For example when $n = 6$, $p = 6$, Sum takes over 600 times more search nodes than the other methods.

Once again the symmetry-reduced tree outperforms all types of table constraint and the Regular decomposition. The total overhead of generating the symmetry-reduced tree (from Table 7) is 2749 s. If the tree were generated once for each instance, it would repay its overhead only on the hardest instance $n = 8$, $p = 6$. However in general we amortise the cost of generating the tree over all instances.

8.5. XCSP benchmarks

Our final experiment is on the XCSP benchmarks compiled by Christophe Lecoutre.[3] We used CSP and MaxCSP benchmarks and discarded WCSP. MaxCSP instances are treated as CSP. Benchmarks containing only intensional constraints were discarded. All remaining benchmarks were translated to Minion file format.

In this section we say a *relation* is a semantic description of a constraint, and a *scope* is the application of a relation to a particular set of variables in a particular benchmark. XCSP benchmarks contain both positive and negative extensional relations. We represent an extensional relation by a set of initial domains, a table of (satisfying or unsatisfying) tuples of domain values, and a single boolean value indicating whether the table is positive or negative. Two relations are distinct iff this representation is distinct.

The table below summarises the occurrences of extensional relations and scopes in the benchmark set. The first line indicates that (of the 6.5 million scopes) in 10.61% of cases the same relation has no other scope, and in 85.60% of cases the same relation has at least 99 other scopes (the 100+ column). The second line indicates that most of the relations have only one scope.

	Number	Percentage of occurrences			
		1	2–9	10–99	100+
Extensional Scopes	6,534,116	10.61	2.21	1.58	85.60
Extensional Relations	750,346	92.37	6.94	0.58	0.11

We focus on relations with 100 or more scopes. This means we consider only 827 relations, but over 85% of scopes.

The largest constraints for which we have successfully generated symmetry-reduced trees are Brian's Brain and Immigration (both of which have 30 literals) and LABS Six (which has 31 literals). All three took over two minutes to generate (Table 7). To avoid long generation times we filtered out the 113 relations that have more than 30 literals.

For the remaining 714 relations we found the symmetry group of each relation using a graph automorphism algorithm implemented in GAP. We ran GenTree and GenTreeSym on these 714 relations. GenTree was limited to exploring 3 million nodes, and GenTreeSym was limited to exploring 400,000 nodes. Within these limits, both algorithms generated trees for the same set of 683 relations. GenTree took a total of 184,291 s, and GenTreeSym took 147,863 s (including both Python and GAP) when executed in parallel on a 32-core AMD Opteron 6272 at 2.1 GHz.

The symmetry-reduced trees algorithm performed only 8% as much search while generating propagator trees, and the symmetry-reduced trees took 13% as much space as the standard trees. However both approaches generated trees for the same set of relations within the node limits. There are two reasons for this: firstly the library (named SCSCP) we used to link Python and GAP is quite slow therefore we have a much lower node limit on GenTreeSym than GenTree. Secondly, the symmetry groups were in the main quite small, with most having between 1 and 1024 symmetries.

The VM instructions for these 1366 propagator trees were stored on disk using an SHA-1 hash of the relation as part of the filename. For this experiment Minion was extended with a special table constraint that computes the hash of the relation and attempts to load a matching propagator tree. If there is no propagator tree it uses a generic GAC propagator.

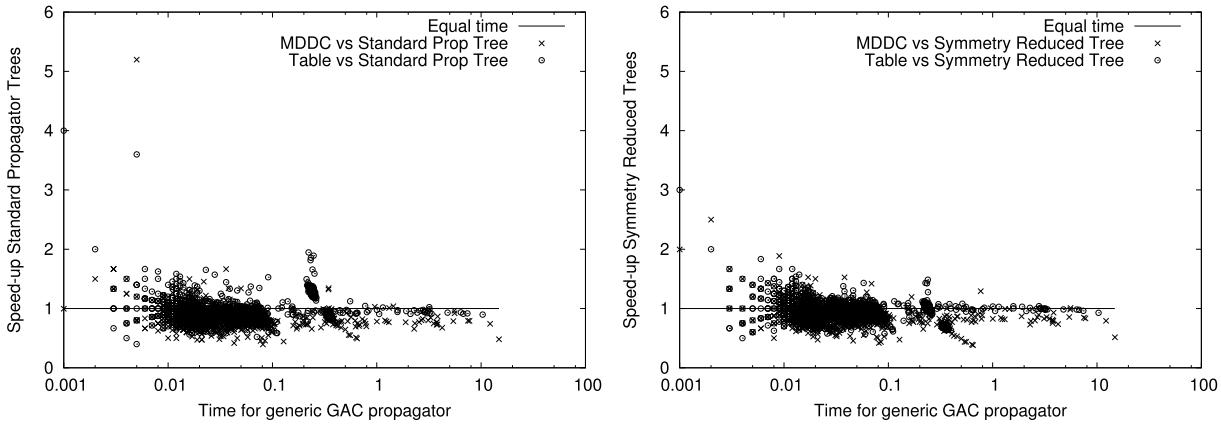
We filtered the benchmark set to remove any benchmarks containing no scopes of the set of 683 relations. We also filtered out benchmarks that take more than 12 GiB memory.[4] 1930 benchmarks remained from 34 series.

On the Life, LABS, Peg Solitaire, Immigration and Brian's Brain problem classes, no one generic GAC propagator clearly dominates the others. Minion's Table propagator, MDDC and STR2+ are each most efficient for different subsets of the instances. For this experiment we need both positive and negative table propagators, and we do not have a negative STR2+ propagator. Therefore we compare propagator trees to Minion's Table propagator and its negative counterpart (both using a trie datastructure), and to MDDC (the Sparse variant, as in previous experiments) using an MDD generated from either a positive or negative table.

When comparing MDDC to propagator trees, each benchmark is executed three times. First it is executed with all extensional relations implemented by MDDC. Second, each of the 683 relations with a standard propagator tree are implemented by the propagator tree and the other relations by MDDC. Third, each of the relations with a symmetry-reduced propagator tree are implemented by that propagator tree and the others by MDDC. Similarly, to compare to Table each benchmark was

[3] The entire set of XCSP benchmarks was downloaded from http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html on 26th June 2013.

[4] Minion's Discrete variable type was used for all variables. Discrete is the only variable type that allows GAC to be enforced on table constraints. Memory use is proportional to the number of domain values.



	Generic Propagator	Standard Tree	Sym-reduced Tree
Table	0.034	0.033	0.034
MDDC	0.041	0.034	0.034

Fig. 6. XCSP experiment comparing MDDC and Table to both standard and symmetry-reduced propagator trees for all benchmarks with 100 or fewer nodes of search. The x-axis is the time without propagator trees for the generic GAC propagator. The y-axis is the speed-up factor obtained when propagator trees are used. The table gives the geometric mean of the total time for each configuration.

executed three times. Each run had a time limit of 30 minutes and they were performed 32 in parallel on an AMD Opteron 6272 at 2.1 GHz.

Fig. 6 plots the results for benchmarks where there was 100 or fewer nodes of search (1470 benchmarks). These plots compare total time. On these benchmarks, on average propagator trees provide very little benefit compared to either MDDC or Table.

Fig. 7 shows the results for all benchmarks with more than 100 search nodes (460 benchmarks). Many benchmarks timed out so we use node rate in these plots. The plots for standard and symmetry-reduced trees are broadly similar, and for both we find most points lie between a factor of 3 speed-up and equal speed. Comparing MDDC to Table, the results are also broadly similar. For both MDDC and Table, most points lie between 1 and 3 times speed-up.

Comparing Table to standard trees using geometric means, the speed-up factor is 1.61. 184,291 s was spent generating the standard trees, which is on average 401 s per benchmark. On average, after 657 s of search the standard tree configuration has paid off the initial cost of GenTree. Of the 460 benchmarks, 303 searched for more than 1000 s and so more than paid off the cost of generating the trees.

When generating the standard trees, we observed that in almost all cases GenTree takes less than 5 s, and the total time is inflated by a small number that take thousands of seconds. Setting a limit of 5 s would dramatically reduce the total time (to less than 3570 s) while generating 633 propagator trees as opposed to 683, and we expect it would reduce the pay-off point dramatically too.

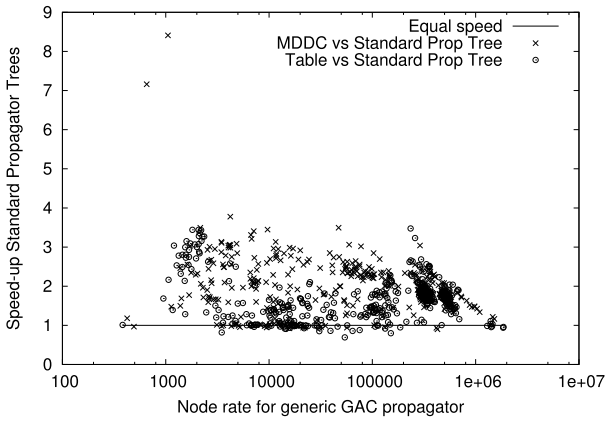
Finally, our experiments underestimate the effect of propagator trees because they include propagating all other extensional and intensional constraints and the search algorithm.

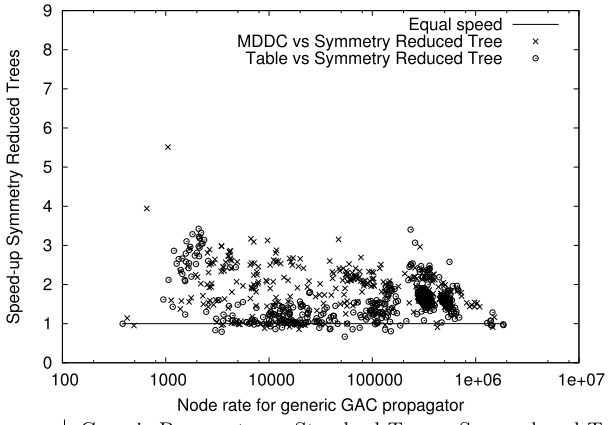
8.6. Experimental conclusions

These experiments have demonstrated that symmetry is useful in extending the scalability of propagator trees. On LABS, we found that the symmetry-reduced trees were orders of magnitude smaller than standard propagator trees. For Life, we found the symmetry-reduced tree was 64 times smaller. Also, we were able to scale up to Immigration and Brain's Brain (with 30 literals, compared to 20 for Life).

The efficiency of symmetry-reduced trees during execution (compared to standard propagator trees) is good for LABS and Peg Solitaire, but for Life we found them to be approximately two times slower. Even so, symmetry-reduced trees outperformed table constraints in all our experiments except XCSP, where symmetry-reduced trees still performed better on average than table constraints. For each problem, the best symmetry-reduced tree outperforms all other methods except standard propagator trees.

Finally we compared standard and symmetry-reduced trees to generic GAC propagators using a large set of XCSP benchmarks. This experiment showed that propagator trees can be of benefit on a wide range of problems, with a few conditions: that the problems should be sufficiently difficult that they cause the solver to do a non-trivial amount of search, that there are relations small enough to apply GenTree or GenTreeSym, and that some of those relations have multiple scopes in the set of problems.





	Generic Propagator	Standard Tree	Sym-reduced Tree
Table	87,041	139,822	131,126
MDDC	75,038	141,686	131,456

Fig. 7. XCSP experiment comparing MDDC and Table to both standard and symmetry-reduced propagator trees for all benchmarks with more than 100 nodes of search. The x-axis represents the node rate without propagator trees for the generic GAC propagator. The y-axis is the speed-up factor obtained when propagator trees are used. The table gives the geometric mean of the node rate for each configuration.

9. Related work

GAC table propagators

There are a variety of algorithms which achieve GAC propagation for arbitrary constraints, for example GAC2001 [3], GAC-Schema [4], MDDC [5], STR2 [6] and Regular [8]. These approaches can typically enforce GAC in polynomial time when their data structure is of polynomial size (whether it is a list of tuples, a trie, an MDD or a finite automaton). In the worst case they have exponential time complexity. Our approach differs in that it guarantees polynomial time propagation after an exponential preprocessing step.

In GAC2001 and GAC-Schema, constraints presented as a set of allowed tuples have the allowed tuples stored as a simple list. There have been a number of attempts to improve upon these algorithms by using different data structures to store the allowed tuples. Notable examples are tries [7], Binary Decision Diagrams [9], Multi-valued Decision Diagrams [5] and c-tuples (compressed tuples) [11]. In all cases the worst case complexity is polynomial in the size of the data structure. In some cases the data structure can be much smaller than an explicit list of all allowed tuples, but the worst case time remains exponential. That is, establishing GAC during search can take time d^n, compared to our worst case of $O(nd)$, or $O(n^2d^2)$ with symmetry reduction (assuming the solver can query and remove domain values in $O(1)$ time).

Other improvements to GAC table propagators, such as caching and reusing results [30], have also improved average-case performance, but have not removed the worst-case exponential behaviour.

Constraint handling rules

Constraint Handling Rules is a framework for representing constraints and propagation. Apt and Monfroy [31] have shown how to generate rules to enforce GAC for any constraint, although they state that the rules will have an exponential running time in the worst case. ARM [32] will automatically generate sets of constraint handling rules for a constraint, but may not

achieve GAC. Further, how completely and efficiently the rules will be executed is dependent on the CHR system the rules are used in.

The major difference therefore between these techniques and the algorithms in this paper is that our algorithms provide guaranteed polynomial-time execution during search, at the cost of much higher space requirements and preprocessing time than any previous technique. Work in CHR is closest in spirit to our work, but does not guarantee to achieve GAC in polynomial time.

It is possible that techniques from knowledge compilation [33] (in particular prime implicants) could be usefully applied to propagator generation. However, the rules encoded in a propagator tree are not prime implicants — the set of known domain deletions is not necessarily minimal. We do not at present know of a data structure which exploits prime implicants and allows $O(nd)$ traversal.

Symmetry

There is a large body of work on symmetry breaking in constraint programming. The research focuses on reducing search effort by avoiding search states that are symmetric to previously-seen states, using a number of different techniques. For example, Symmetry Breaking During Search [20] posts constraints during search to forbid visiting symmetric states in the future. Symmetry Breaking by Dominance Detection (SBDD) [34] checks each state for symmetry to previously-seen states. Also, there are many approaches to breaking symmetry by adding constraints prior to search, for example lexicographic ordering constraints [35].

Of these approaches, our algorithm is most similar to SBDD. However, unlike SBDD we are not merely checking if the current state is dominated, we need a reference to the previous (symmetric) state and a permutation mapping one to the other. Therefore we store all previous states, whereas in SBDD sibling states are merged in the database. Also, our algorithm runs in polynomial time during search, whereas SBDD solves an NP-complete problem at every node.

Our definition of symmetry is based on Cohen et al. [27].

10. Conclusion

We have presented a novel and general approach to propagating small constraints. The approach is to generate a custom stateless propagator that enforces GAC in $O(nd)$. This is a spectacular improvement over other general techniques, which are exponential in the worst case, but comes with an equally spectacular tradeoff. This is that the stored propagator can be very large — it scales exponentially in the size of the constraint — therefore generating and storing it is only feasible in general at very small sizes.

We have presented two methods for storing and then executing the generated constraints. One is to construct special purpose code (in our case in C++) and then compile it before use. The second is that we use a simple virtual machine with a tiny special purpose instruction set in which propagator trees can be executed. The second method has the advantage of not requiring compilation — apart from the convenience of not needing a compiler sometimes the propagator code becomes too big to compile.

We demonstrated that the propagator generation approach can be highly efficient compared to table constraints. For example, on Life $n = 7$, $p = 4$, the standard propagator tree is 9.7 times faster than MDDC, and 7.2 times faster than an encoding using a sum constraint. Remarkably, propagator trees can even be faster than hand-optimised propagators. For example, we achieved a 27% speedup on a min constraint in peg solitaire instance 10.

We significantly extended the scalability of our approach by exploiting symmetry within the constraint. To do this we introduced symmetry-reduced trees and algorithms for dealing with them. This allowed us to scale up from the Life constraint (with 20 literals) to extended variants of Life with 30 literals. While this may seem a small step, it enabled us to solve variants of Life for which we could not previously build trees. On the LABS problem we observed three orders of magnitude reduction in the size of the generated propagator tree. Again we provided both compiled and virtual machine implementations. However run time worsens to $O(n^2d^2)$ in the worst case from $O(nd)$ in the non-symmetric case. This did cause a slowdown in our experiments compared to the non-symmetric version where available, but we still achieved very good performance.

Our analysis of the XCSP benchmark set showed that while there were 750,346 different constraint relations applied to over 6.5 million scopes, the most common 827 constraint relations covered over 85% of the constraint scopes. This demonstrates how a small number of specialised propagators can cover a large proportion of the constraint scopes in a large set of benchmarks.

We believe that our approach of building special purpose generated constraint propagators has considerable promise for the future. While surprisingly fast, the propagator trees are entirely stateless — there is no state stored between calls, and no local variables. They also do not make use of trigger events, which are often essential to the efficiency of propagators. Therefore we believe there is scope to scale the approach further and to improve efficiency. Additionally, we believe that symmetry-reduced trees are worthy of further study. They are a general construction and further study may show them to have other important applications beyond constructing efficient propagators.

Acknowledgements

We would like to thank anonymous reviewers for their helpful comments. This research was supported by EPSRC grants with numbers EP/H004092/1 and EP/E030394/1.

Appendix A. Canonicalisation of sequences of objects

In order to generate symmetry-reduced trees, we need to identify symmetric node-states. To do this, we use a canonicalisation function. A node-state is represented by a sequence of sets. We develop a canonicalisation function which operates on sequences of objects (including sets). The function is novel to the best of our knowledge, and is an extension of an existing group-theoretic algorithm [24]. The algorithm requires that the objects in the sequence can be stabilised and have a canonicalising function.

Definition 14. Given the following:

- a list $L = [l_1, \ldots, l_n]$;
- a *canonicalising function* $f(l_i, H_c)$ for the l_i and any group H_c; and
- a stabilising function $s(l_i, H_s)$ which returns (for any group H_s) the subgroup of H_s which stabilises l_i,

then the function $Can(L, G)$ is defined as follows:

If L is the empty list return the identity element of G, otherwise,

1. Find $GCan = f(L[1], G)$.
2. Find $GStab = s(L[1]^{GCan}, G)$.
3. Generate the list L' where $\forall i \in \{2..n\}.L'[i-1] = L[i]^{GCan}$, which is one element shorter than L.
4. Return the permutation $GCan.Can(L', GStab)$.

The following theorem proves the correctness of the key definition above.

Theorem 5. *The function $Can(L, G)$, given in Definition 14, is a canonicalisation function.*

Proof. The permutation returned by $Can(L, G)$ in Definition 14 is always a member of G, as it is constructed by composing elements of G. Therefore it suffices to prove for any sequences L and M of equal length, if there exists $g \in G$ such that $L^g = M$ then $L^{Can(L,G)} = M^{Can(M,G)}$. We proceed by induction on the length of L and M. If they are empty, then the result is trivially true.

We shall refer to $f(L[1], G)$ as c, and $f(M[1], G)$ as d. As f is a canonicalising function, and $L[1]^g = M[1]$, then $L[1]^c$ and $M[1]^d$ are equal. Therefore both $s(L[1]^c, G)$ and $s(M[1]^d, G)$ are the same group. Call this group $GStab$.

Now we consider the recursive call to Can. For L, this involves applying c to $L[2], \ldots, L[n]$. For M, this involves applying d to $M[2], \ldots, M[n]$, which is the same as applying $g.d$ to $L[2], \ldots, L[n]$.

We will now prove that there exists a group element h in $GStab$ that maps $L[2..n]^c$ to $M[2..n]^d$. h is the equivalent of g in the inductive step. As discussed earlier, $L[1]^c = M[1]^d$ and $M[1]^d = L[1]^{g.d}$. Let h be defined such that $c.h = g.d$. It is trivially true that $L[1]^{c.h} = L[1]^{g.d}$ and therefore $L[1]^c = M[1]^d = L[1]^{g.d} = L[1]^{c.h}$, so h is in the stabiliser of $L[1]^c$, which is $GStab$.

Let $a = Can(L[2..n]^c, GStab)$ and $b = Can(M[2..n]^d, GStab)$. As the group element h which maps $L[2..n]^c$ to $M[2..n]^d$ is in $GStab$, by the inductive hypothesis, $L[2..n]^{c.a} = M[2..n]^{d.b}$. As a and b are in $GStab$, $L[1]^{c.a} = L[1]^c$ and $M[1]^{d.b} = M[1]^d$. Therefore $L^{c.a} = M^{d.b}$, so $L^{Can(L,G)} = M^{Can(M,G)}$. □

We now provide a concrete implementation of Can (Definition 14) for a list of sets of points (represented using integers) in Algorithm 7. This algorithm assumes the existence of two pre-existing group theory algorithms:

1. `SetStabiliser`(S, G) : Generates the subset of G which stabilises S.
2. `MinimalImagePerm`$(S, [Stab,]G)$: Generates the element h of G such that $\forall g \in G.h(S) \leqslant g(S)$. The function may optionally be given $Stab =$ `SetStabiliser`(S, G) to provide a performance improvement. This is the canonicalising function for sets that we use in Algorithm 7.

`SetStabiliser` is provided by any computational group theory package. The algorithm `MinimalImagePerm` is built from the `SmallestImage` algorithm of Linton [24]. The original algorithm of Linton provides the canonical image of a set, and we modified it to return the permutation which generates the canonical image. It is simple to augment the algorithm to produce this as it progresses.

Calculating set stabilisers and minimal images are both expensive operations, while calculating the conjugate of a group is very cheap. In [24], the algorithm `SmallestImage`(S, G) may be given the result of `SetStabiliser`(S, G), which in

Algorithm 7 CanonicalSetList(G, $\langle S_1, \ldots, S_n \rangle$)

```
1: ModPerm ← e {The identity permutation}
2: CurrentG ← G
3: i ← 1
4: while i ⩽ n do
5:     Stab ← SetStabiliser(ModPerm.S_i, CurrentG)
6:     MinPerm ← MinimalImagePerm(ModPerm.S_i, Stab, CurrentG)
7:     CurrentG ← Stab^ModPerm {Take the ModPerm conjugate of Stab}
8:     ModPerm ← MinPerm.ModPerm
9:     if |CurrentG| = 1 then
10:        return ModPerm
11:    i ← i + 1
12: return ModPerm
```

some cases leads to a substantial speed improvement. As we have to calculate at least one set stabiliser during each step of our algorithm anyway, we generate one early so we can pass it to `MinimalImagePerm`, and then conjugate it for the next step of the algorithm.

Theorem 6. *Given a list of sets $L = \langle S_1, \ldots, S_n \rangle$ and a group G, then Algorithm 7 is a canonicalising function.*

Proof. Theorem 5 proves the abstract algorithm correct. Algorithm 7 optimises the basic algorithm shown in Definition 14 by not transforming the whole list at every step, but by constructing the permutation *ModPerm* which must be applied to the rest of the list at each step. The final value of variable *ModPerm* is the canonicalising permutation. Also, we use the basic group theory result that for all $g \in G$, $s(x, G)^g = s(x^g, G)$, which allows us to calculate just one stabiliser and use it in two places. Finally, if the group becomes trivial we are able to terminate the algorithm early. □

References

[1] C. Bessiere, Handbook of Constraint Programming, Elsevier Science Inc., New York, NY, USA, 2006, pp. 29–83, Ch. Constraint Propagation.
[2] C. Jefferson, A. Miguel, I. Miguel, A. Tarim, Modelling and solving english peg solitaire, Comput. Oper. Res. 33 (10) (2006) 2935–2959.
[3] C. Bessière, J.-C. Régin, R. Yap, Y. Zhang, An optimal coarse-grained arc consistency algorithm, Artif. Intell. 165 (2005) 165–185.
[4] C. Bessière, J.-C. Régin, Arc consistency for general constraint networks: Preliminary results, in: IJCAI(1), 1997, pp. 398–404.
[5] K.C. Cheng, R.H. Yap, An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints, Constraints 15 (2) (2010) 265–304.
[6] C. Lecoutre, STR2: optimized simple tabular reduction for table constraints, Constraints 16 (4) (2011) 341–371.
[7] I.P. Gent, C. Jefferson, I. Miguel, P. Nightingale, Data structures for generalised arc consistency for extensional constraints, in: AAAI'07: Proceedings of the 22nd National Conference on Artificial Intelligence, AAAI Press, 2007, pp. 191–197.
[8] G. Pesant, A regular language membership constraint for finite sequences of variables, in: Proceedings of the 10th International Conference on the Principles and Practice of Constraint Programming (CP 2004), 2004, pp. 482–495.
[9] K.C.K. Cheng, R.H.C. Yap, Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints, in: Proceeding of the 2006 Conference on ECAI 2006, IOS Press, Amsterdam, The Netherlands, 2006, pp. 78–82.
[10] C. Lecoutre, R. Szymanek, Generalized arc consistency for positive table constraints, in: Principles and Practice of Constraint Programming – CP 2006, 2006, pp. 284–298.
[11] G. Katsirelos, T. Walsh, A compression algorithm for large arity extensional constraints, in: Principles and Practice of Constraint Programming (CP 2007), 2007, pp. 379–393.
[12] C. Lecoutre, C. Likitvivatanavong, R.H.C. Yap, A path-optimal GAC algorithm for table constraints, in: ECAI 2012 – 20th European Conference on Artificial Intelligence, 2012, pp. 510–515.
[13] J.-B. Mairy, P. Van Hentenryck, Y. Deville, An optimal filtering algorithm for table constraints, in: CP 2012 – 18th International Conference on Principles and Practice of Constraint Programming, 2012, pp. 496–511.
[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd ed., MIT Press/McGraw-Hill, 2001.
[15] I.P. Gent, C. Jefferson, I. Miguel, P. Nightingale, Generating special-purpose stateless propagators for arbitrary constraints, in: Proceedings of 16th International Conference on Principles and Practice of Constraint Programming (CP 2010), 2010, pp. 206–220.
[16] I.P. Gent, C. Jefferson, I. Miguel Minion, A fast, scalable, constraint solver, in: Proceedings 17th European Conference on Artificial Intelligence (ECAI 2006), 2006, pp. 98–102.
[17] I.P. Gent, C. Jefferson, S. Linton, I. Miguel, P. Nightingale, Finite state automata for the paper Generating Custom Propagators for Arbitrary Constraints, Tech. Rep. CIRCA Preprint 2013/7, University of St Andrews, 2013.
[18] N. Beldiceanu, M. Carlsson, R. Debruyne, T. Petit, Reformulation of global constraints based on constraints checkers, Constraints 10 (4) (2005) 339–362.
[19] C. Schulte, G. Tack, View-based propagator derivation, Constraints 18 (1) (2013) 75–107.
[20] I.P. Gent, B.M. Smith, Symmetry breaking in constraint programming, in: W. Horn (Ed.), Proceedings of ECAI-2000, IOS Press, 2000, pp. 599–603.
[21] R. Bosch, M. Trick, Constraint programming and hybrid formulations for three life designs, Ann. Oper. Res. 130 (2004) 41–56.
[22] B.M. Smith, A dual graph translation of a problem in 'Life', in: Principles and Practice of Constraint Programming (CP 2002), 2002, pp. 402–414.
[23] G. Chu, P.J. Stuckey, M.G. de la Banda, Using relaxations in maximum density still life, in: Principles and Practice of Constraint Programming (CP 2009), 2009, pp. 258–273.
[24] S. Linton, Finding the smallest image of a set, in: Proceedings of ISSAC 04, ACM Press, 2004, pp. 229–234.
[25] The GAP Group, GAP – Groups, Algorithms, and Programming, Version 4.5.6; 2012 (http://www.gap-system.org).
[26] D. Wallace, Groups, Rings and Fields, Springer-Verlag, 1998.
[27] D. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, B.M. Smith, Symmetry definitions for constraint programming, Constraints 11 (2–3) (2006) 115–137.
[28] E.W. Weisstein, Immigration, http://www.ericweisstein.com/encyclopedias/life/Immigration.html.
[29] Brian's brain, http://en.wikipedia.org/wiki/Brian's_Brain.

[30] C. Lecoutre, F. Hemery, A study of residual supports in arc consistency, in: IJCAI'07: Proceedings of the 20th International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007, pp. 125–130.
[31] K.R. Apt, E. Monfroy, Constraint programming viewed as rule-based programming, Theory Pract. Log. Program. 1 (6) (2001) 713–750.
[32] S. Abdennadher, A. Olama, N. Salem, A. Thabet, ARM: Automatic rule miner, in: Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, 2006, pp. 17–25.
[33] A. Darwiche, P. Marquis, A knowledge compilation map, J. Artif. Intell. Res. 17 (2002) 229–264.
[34] T. Fahle, S. Schamberger, M. Sellmann, Symmetry breaking, in: Proceedings of Principles and Practice of Constraint Programming (CP 2001), 2001, pp. 93–107.
[35] A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh, Propagation algorithms for lexicographic ordering constraints, Artif. Intell. 170 (10) (2006) 803–834.