Generating Special-purpose Stateless Propagators for Arbitrary Constraints

Ian P. Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale
{ipg,caj,ianm,pn}@cs.st-andrews.ac.uk

School of Computer Science, University of St Andrews, St Andrews, Scotland, UK.

Abstract. Given an arbitrary constraint c on n variables with domain size d, we show how to generate a custom propagator that establishes GAC in time $O(nd)$ by precomputing the propagation that would be performed on every reachable subdomain of *scope*(c). Our propagators are *stateless*: they store no state between calls, and so incur no overhead in storing and backtracking state during search. The preprocessing step can take exponential time and the custom propagator is potentially exponential in size. However, for small constraints used repeatedly, in one problem or many, this technique can provide substantial practical gains. Our experimental results show that, compared with optimised implementations of the table constraint, this technique can lead to an order of magnitude speedup, while doing identical search on realistic problems. The technique can also be many times faster than a decomposition into primitive constraints in the Minion solver. Propagation is so fast that, for constraints available in our solver, the generated propagator compares well with a human-optimised propagator for the same constraint.

1 Introduction

Constraint models of structured problems often contain many copies of a constraint, which differ only in their scope. English Peg Solitaire, for example, is naturally modelled with a *move* constraint for each of 76 moves, at each of 31 time steps, giving 2,356 copies of the constraint [14]. Efficient implementation of such a constraint is vital to solving efficiency, but choosing an implementation is often difficult. The solver may provide a hand-optimized propagator, otherwise there are two choices: decompose the constraint, or use a table propagator. Decompositions typically introduce extra variables (and so overhead) and/or reduce propagation. In the worst case table propagators take time exponential in the size of their scope. Even hand-optimized propagators the solver provides may not be optimal if they are designed for a more general class of constraints.

The algorithms we give herein generate GAC propagators for arbitrary constraints that run in time $O(nd)$, in extreme cases an exponential factor faster than any table constraint propagator [4, 12, 7, 6, 17, 15]. As our experiments show, generated propagators can even outperform hand-optimized propagators when performing the same propagation. Our approach is general but in practice does not scale to large constraints as it precomputes domain deletions for all reachable subdomains. It scales easily to 10 Boolean variables, as a case study shows.

2 Theoretical Background

We summarise relevant definitions. For further discussion of propagators see [2].

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 $\forall v \in V : D(v) \subsetneq \mathbb{Z}$; 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 for X is a function from variables to sets which satifies: $\forall 1 \leq i \leq n : \; S(x_i) \subseteq D(x_i)$ We abuse notation, in a natural way, to write $R \subseteq S$ for subdomain lists R and S iff $\forall 1 \leq i \leq n : R(x_i) \subseteq S(x_i)$. Given a CSP instance $P = \langle V, D, C \rangle$, a **search state** for P is a subdomain list for V.*

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 Generalized Arc Consistency (GAC):

Definition 3. *Given a constraint c, a subdomain list S of scope(c) is **Generalized Arc Consistent (GAC)** if, for every $d \in S(v)$, the literal $\langle v, d \rangle$ is in some assignment which satisfies c and is contained in S.*

Any literal that does not satisfy the test in Definition 3 may be removed.

Definition 4. *Given a CSP $P = \langle V, D, C \rangle$, a search state S for P 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 in no solution to c, and is otherwise identical to S.*

3 Propagator Generation

GAC propagation is NP-hard for 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 [9](§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 $scope(c)$. The 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 sub-domain list with a bit, and concatenate the bits to form an integer.

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 $(2^d-1)^n.nd$.

This algorithm has obvious disadvantages. This preprocessing step can take substantial time and T requires substantial space. However, for small constraints which are used repeatedly, in either one problem or many problems, we shall show how a refinement of this technique can provide substantial practical gains.

4 Generating Tree Propagators

The above approach uses a large data structure, containing all possible subdomain lists. Also, the generated propagator tests the presence of every value in each domain before propagating. 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 fails or is entailed. This improves the average- but not the worst-case complexity. In this section we describe an algorithm that generates a tree-propagator, given any propagator and entailment checker for the constraint in question. First we define tree-propagator.

Definition 5. *A **tree-propagator** is a rooted tree $T=\langle V,L,R,r,Prune,Test\rangle$ with vertices V, where $r\in V$ is the root, L is a function mapping vertices to their left child, and R maps vertices to their right child. Two other functions map vertices to prunings and to a literal: $Prune: V\rightarrow 2^{\{(x_i,a)|a\in D_i\}}$, and $Test: V\rightarrow\{(x_i,a)|a\in D_i\}$.*

An execution of a tree-propagator follows a path in T starting at the root r. At each vertex v, the propagator prunes the values specified by $Prune(v)$, and tests if the literal $Test(v)=(x_i,a)$ is in the current domain. If $a\in D(x_i)$, then the next vertex in the path is the left child $L(a)$, otherwise it is the right child $R(a)$. If the relevant child is not present, then the propagator is finished.

SimpleGenTree (Algorithm 1) is a naive 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. Let D_{cur} be the current domain. As a tree-propagator is executed, it tests values to obtain more information about D_{cur}. At a given tree node, each value from the initial domain D may be in D_{cur}, or out, or unknown (not yet tested). SimpleGenTree constructs a sub-domain list SD for each tree node, representing values that are in D_{cur} or unknown. It also constructs *ValsIn*, representing values that are known to be in D_{cur}.

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

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

SimpleGenTree proceeds in two stages. First, it runs a propagation algorithm on SD to compute the prunings required given current domain knowledge. The prunings are stored in the current tree node, and each pruned value is removed from SD and $ValsIn$ to form SD' and $ValsIn^*$. If a domain is empty in SD', the algorithm returns. If only one value remains for some variable in SD', the value is added to $ValsIn^*$ to form $ValsIn'$ (because otherwise the domain is empty).

The second stage is to choose a literal and branch. This literal is unknown, ie 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 D_{cur}. On the right, the chosen literal is removed from SD. The main terminating condition for the recursion is when $SD' = ValsIn'$. At this point, we have complete knowledge of the current domains: $SD' = ValsIn' = D_{cur}$. The recursion also terminates when a domain is emptied by propagation.

4.1 Generating Code

Algorithm 2 (GenCode) generates a program from a tree-propagator 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 this code is very short and the same for all generated propagators. As an alternative to generating code, it is possible to execute a tree-propagator by traversing the tree at run time. However, in preliminary experiments we found this approach to be roughly 25% slower.

4.2 Correctness

In order to prove the SimpleGenTree algorithm correct, we assume that the Propagate function called on line 1 enforces GAC. We need to be careful about what GAC propagators do. Note that, if a GAC propagator produces a domain wipeout, it should also delete all values of all other variables in the constraint.

Algorithm 2 GenCode(Tree-propagator T, Vertex v)

```
1: if $v$=Nil then
2:     WriteToCode("NoOperation;")
3: else
4:     WriteToCode("RemoveValuesFromDomains("+$Prune(v)$+");")
5:     if $Test(v) \neq$ Nil then
6:         $(x_i, a) \leftarrow Test(v)$
7:         WriteToCode("if IsInDomain("+$a$+","+$x_i$+") then")
8:         GenCode($T$,$L(v)$)
9:         WriteToCode("else")
10:        GenCode($T$,$R(v)$)
11:        WriteToCode("endif;")
```

We assume that the Propagate function does this. We also assume that the target constraint solver removes all values of all variables in a constraint if our generated propagator 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 propagations 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 which satisfies c and is contained in S. Therefore any literals in S which 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. This can be reached in one of two cases:

1. $S \setminus$ Deletions has at least one empty domain. In this case, the returned tree node correctly leads to S having a domain wipeout.
2. $S \setminus$Deletions has no empty domains. In this case, 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, else there would be an empty domain in S. Thus we have $\mathit{ValsIn}' \subseteq (S \setminus \text{Deletions}) \subseteq SD'$. But since $\mathit{ValsIn}' = SD'$, 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.

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 wipeout, then (as we have assumed) the constraint solver sets $S = \emptyset$, and the propagator has established GAC, no matter what happens in the rest of the tree.

If no domain wipeout occurs, we can progress to Line 9. Again using the same arguments as in Lemma 1, assuming that the Deletions do not cause a domain wipeout in S, then once we get to line 9, we know that $ValsIn' \subseteq S \setminus \text{Deletions} \subseteq SD'$. Since we passed Line 7, we know that $ValsIn' \neq SD'$, and therefore there is at least one value for the heuristic to choose.

There are now two cases. The heuristic value (y, l) is in S, or not.

If $(y, l) \in S$, 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 $(y, l) \in S$, 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 $(y, l) \notin S$, 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 $SD \setminus \mathit{ValsIn}$ is always reduced by at least one literal on each recursive call to Algorithm 1, and can never grow. Therefore we know the algorithm will eventually terminate. With this theorem proved the main result we want is an immediate corollary. □

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* $(c, \emptyset, D)$*, 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 $O(nd)$.

5 Generating Smaller Trees

Algorithm 3 shows the GenTree algorithm. This is an improvement of SimpleGenTree. We present this without proof of correctness, but a proof would be easy since the effect is only to remove nodes in the tree for which no propagation can occur at any subtree.

The first efficiency measure is that GenTree always returns Nil when no pruning is performed at the current node or any of its children. This means that the generated tree will have pruning at all of its leaf nodes. The second efficiency 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' to check this. The function entailed(c, SD) is called at the start of GenTree, and also after domains are updated by pruning (line 9). If the constraint is entailed under SD, then no pruning is possible for SD or any sub-domain of it. The value returned is either Nil (if values were pruned at the current node) or a tree node with no children.

To illustrate the difference between SimpleGenTree and GenTree, consider Figure 1. The constraint is very small ($x \vee y$ on Boolean domains) 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.

Heuristic The choice of literal to branch on is very important, and can make a huge difference in the size of the propagator-tree. To minimize the size of

Algorithm 3 Generate Tree-Propagator: GenTree(c, SD, *ValsIn*)

```
1: if entailed($c, SD$) then
2:     return Nil
3: Deletions ← Propagate($c$, $SD$)
4: $SD' = SD \setminus$ Deletions
5: if all domains in $SD'$ are empty then
6:     return Treenode(Prune=Deletions, Test=Nil, Left=Nil, Right=Nil)
7: $\mathit{ValsIn}^* \leftarrow \mathit{ValsIn} \setminus$ Deletions
8: $\mathit{ValsIn}' \leftarrow \mathit{ValsIn}^* \cup \{(x,a) | (x,a) \in SD', |SD'(x)| = 1\}$
9: if $SD' = \mathit{ValsIn}'$ or entailed($c, SD$) then
10:     if Deletions=Nil then
11:         return Nil
12:     else
13:         return Treenode(Prune=Deletions, Test=Nil, Left=Nil, Right=Nil)
   {Pick a variable and value, and branch}
14: $(y, l) \leftarrow$ heuristic($SD' \setminus \mathit{ValsIn}'$)
15: LeftT←GenTree($c$, $SD'$, $\mathit{ValsIn}' \cup (y, l)$)
16: if $SD'(y) \setminus \{l\} = \emptyset$ then
17:     RightT←Nil
18: else
19:     RightT←GenTree($c$, $SD' \setminus \{(y, l)\}$, $\mathit{ValsIn}'$)
20: if LeftT=Nil And RightT=Nil And Deletions=$\emptyset$ then
21:     return Nil
22: else
23:     return Treenode(Prune=Deletions, Test=$(y, l)$, Left=LeftT, Right=RightT)
```

the tree, the aim of a heuristic must be to cause Algorithm 3 to return before branching. There are a number of conditions that cause this: entailment (lines 2 and 15); domain wipe-out (line 8); and complete domain information (line 15).

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.

Implementation The implementation is recursive and very closely follows the structure of Algorithm 3. It is instantiated with the GAC2001 table propagator [4]. The implementation maintains a list of disallowed tuples of c that are valid with respect to SD. This list is used by the entailment checker: when the list becomes empty, the constraint is entailed. It is also used to calculate the heuristic described above. It is implemented in Python and is not highly optimized.

In all the case studies below, we use the solver Minion [11] 0.10. We experiment with 3 generated propagators, in each case comparing against hand-optimized propagators provided in Minion, and also against table constraints where appropriate. All case studies were run with a time out of 24 hours. Instances that took less than 1 hour were run 5 times and the median was taken. In all cases times are given for an 8-core Intel Xeon E5520 at 2.27GHz with 12GB RAM. Minion was compiled with g++ 4.4.1, optimisation level -O3.

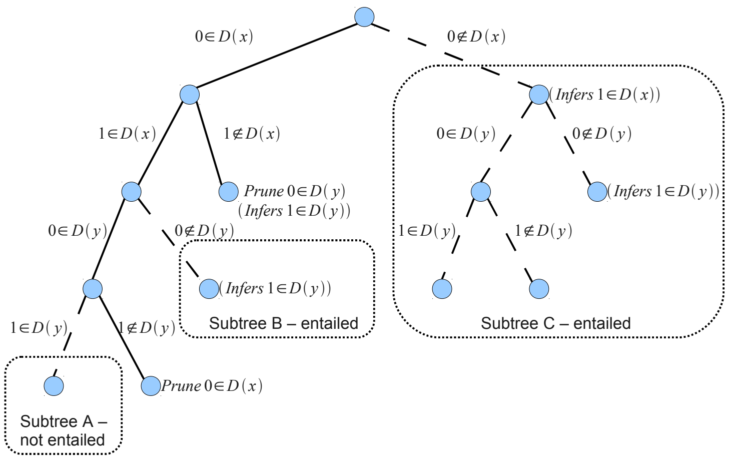

Fig. 1. Example of propagator tree for constraint $x \vee y$ initial domains of {0,1}. The entire tree is generated by SimpleGenTree (Algorithm 1). The more sophisticated algorithm GenTree (Algorithm 3) does not generate the subtrees A, B and C.

Two table constraints were used: Table, which uses a trie data structure with watched literals (as described in [12]), and Lighttable, which uses the same trie data structure but is stateless and uses static triggers. It searches for support for each value of every variable each time it is called.

6 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 [14]. 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 step, 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 moves in our model to 31.

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. 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 constraints are posted: $equal[a, b] \Leftrightarrow (b[a, b] = b[a + 1, b])$. The board state for the first and last time step are filled in, with one hole at the position we are starting at and one peg at the same position we are finishing at.

Starting position	Time (s)			Node rate (per s)			Nodes
	Generated	Min	Reified Sumgeq	Generated	Min	Reified Sumgeq	
1	>86389	>86394	>86400	11249	7088	3303	—
2	1.62	2.48	3.10	6338	4140	3312	10,268
4	>86393	>86381	>86369	10986	7514	3926	—
5	879.25	1351.88	3120.26	12964	8431	3652	11,398,210
9	>86400	>86385	>86380	11135	7531	3544	—
10	110.48	167.30	379.22	13456	8886	3920	1,486,641
17	1.49	2.38	3.97	6892	4315	2587	10,269

Table 1. Results on peg solitaire problems.

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 $\sum_i b[t, i] = 32 - t$.

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.

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

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 $moves[t, m] = 1$.

The arity 7 move constraint is implemented in three ways. The Reified Sumgeq implementation uses a sum to represent the conjunction. The negation of some b variables is achieved with mappers, therefore no auxiliary variables are introduced. The sum constraint is reified to the moves$[t, m]$ variable, as follows: $[(\sum b[t, f_1], \dots, b[t+1, f_3]) \geq 6] \Leftrightarrow moves[t, m]$.

The Min implementation uses a single `min` constraint, as follows. Again mappers are used for negation. $min(b[t, f_1], \dots, b[t+1, f_3]) = moves[t, m]$

The Generated propagator was generated by GenTree in 0.14s. The tree has 316 nodes, and the algorithm explored 521 nodes. The propagator was compiled and Minion linked in 16.5s. (For all case studies, we give the time to compile the generated propagator only, plus the time to link Minion, excluding compilation of the rest of Minion.)

Table 1 shows our results for peg solitaire. In all cases the generated propagator outperforms Min by a substantial margin (54% on instance 5), which is perhaps remarkable given that Min is a hand-optimized propagator. For the harder instances, Generated more than repays the overhead of compiling the specialized constraint. The generated propagator outperforms Reified Sumgeq by an even wider margin.

n	Time (s)				Search nodes		Nodes per second	
	Generated	Product	Lighttable	Table	Generated, (Light)Table	Product	Generated	Product
25	8.86	11.92	20.54	20.71	206,010	365,470	23252	30660
26	18.20	24.86	51.21	43.03	404,879	731,886	22246	29440
27	41.37	53.27	91.58	90.58	790,497	1,383,351	19108	25969
28	80.67	110.66	182.55	184.72	1,574,100	2,755,212	19513	24898
29	131.91	184.64	326.88	360.36	2,553,956	4,550,121	19361	24643
30	258.58	325.63	711.18	697.31	4,120,335	7,345,259	15934	22557

Table 2. Results on LABS problems of size 25-30. All times are a median of 5 runs.

7 Case Study: Low Autocorrelation Binary Sequences

The Low Autocorrelation Binary Sequence (LABS) problem is described by Gent and Smith [13]. The problem is to find a sequence s of length n of symbols $\{-1, 1\}$. For each $k \in \{1 \ldots n-1\}$, the correlation C_k is the sum of the products $s[i] \times s[i+k]$ for all $i \in \{0 \ldots n-k-1\}$. The overall correlation is the sum of the squares of all C_k: $\sum_{k=1}^{n-1} (C_k)^2$. This quantity must be minimized.

The sequence is modelled directly, using variables $s[n] \in \{-1, 1\}$. For each $k \in \{1 \ldots n-1\}$, and each $i \in \{0 \ldots n-k-1\}$, 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 $k \in \{1 \ldots n-1\}$ we have a 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 $minvar = \sum_{k=1}^{n-1} C_k^2$, and $minvar$ is minimized. Gent and Smith identified 7 symmetric images of the sequence [13]. We use these to post 7 symmetry-breaking constraints on s. 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 constraint generation on this problem, we combine pairs of product constraints into a single 5-ary constraint: $(s[i] \times s[k]) + (s[i+1] \times s[k+i+1]) = p_k^i$. This allows almost half of the p_k^i variables to be removed.

We compare four models of LABS: *Product*, the model with ternary product constraints; *Generated*, where the new 5-ary constraint has a generated propagator; *Table* and *Lighttable* where the 5-ary constraint is implemented with a table propagator. The Product model does not enforce GAC on the 5-ary constraint. The Generated propagator was generated by GenTree in 0.007s. The algorithm explored 621 nodes and the resulting propagator has 396 nodes. It was compiled and Minion linked in 15.69s.

Table 2 shows our results for LABS sizes 25 to 30. The instances were solved to optimality. The Generated, Table and Lighttable 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 table models are substantially slower than Product. However, Generated is faster than Product, and for the larger instances it more than repays the overhead of compiling the specialized constraint. This is perhaps remarkable when comparing against hand-optimized product and sum constraints.

8 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 [5]. Smith [18] and Chu et al [8] 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, $n+2$ and $n+3$ are set to 0. For a cell $b[i, j, t]$ at time step t, its liveness is determined as follows. The 8 adjacent cells at the previous step are summed: $s = \sum \text{adjacent}(b[i, j, t-1])$, and $(s > 3 \vee s < 2) \Rightarrow b[i, j, t] = 0$, $(s = 3) \Rightarrow b[i, j, t] = 1$, and $(s = 2) \Rightarrow b[i, j, t] = b[i, j, t-1]$. If t is the first time step, then $p-1$ is the previous step, to complete the loop.

We refer to the grid at a particular time step as a *layer*. For each pair of layers, a `watchvecneq` constraint is used to constrain them to be distinct. To break some symmetries, the first layer is 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. Finally, all cells in all layers are summed to a variable m which is maximized.

The liveness constraint involves 10 Boolean variables. We generated a propagator using the GenTree algorithm. The algorithm explored 87041 nodes in 45s. The resulting propagator tree has 28351 nodes. The constraint is compiled and Minion linked in 217s, so the total overhead is 262s[1].

The generated propagator is compared to two other implementations. The *Sum* implementation adds an auxiliary variable $s[i, j, t] \in 0 \dots 8$ 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

[1] In this case the generated constraint was compiled once for Boolean variables only, rather than multiple times for different variable types as is standard in Minion.

n	period p	Time (s)				Nodes	Nodes per s,
		Generated	Sum	Lighttable	Table		Generated
5	2	0.04	0.09	0.20	0.22	1,169	29,225
5	3	0.08	0.42	1.34	1.26	5,489	68,613
5	4	0.42	2.38	7.42	6.05	21,906	52,157
5	5	1.09	6.35	21.55	16.66	49,704	45,600
5	6	2.34	11.18	40.00	38.15	71,809	30,688
6	2	0.13	0.67	2.03	2.17	13,631	104,853
6	3	0.93	7.02	19.18	24.59	88,655	95,328
6	4	11.98	75.29	350.19	225.29	886,371	73,988
6	5	124.75	896.97	2779.78	1999.82	6,172,319	49,478
6	6	446.44	3108.18	13929.2	6231.22	16,538,570	37,045
7	2	2.34	13.63	44.57	66.58	316,612	135,304
7	3	18.84	122.13	585.48	377.50	1,905,288	101,130
7	4	366.59	2517.26	12163.6	6706.33	29,194,918	79,639
7	5	9822.84	67014.9	>86393	>86397	564,092,290	50,664
7	6	>86395	>86398	>86398	>86359	—	32,922

Table 3. Time to solve to optimality, for each implementation of the life constraint

liveness rules. The *Table* implementation simply encodes the arity-10 constraint as a `table` or `lighttable` constraint.

We used instances with parameters $n \in \{5, 6, 7\}$ and period $p \in \{2, 3, 4, 5, 6\}$. Results are shown in Table 3. In 6 cases, the instances timed out after 24 hours, but otherwise they were solved to optimality. The three models explored the same number of nodes in all cases.

The generated propagator is substantially faster than the sum implementation. For instance $n = 7$ $p = 5$, Generated is 6.8 times faster than Sum. Also, Sum is faster than Table by a factor of 2 or more. For the four hardest instances that were solved ($n = 6$, $p \in \{5, 6\}$, and $n = 7$, $p \in \{4, 5\}$), the generated propagator more than paid back its 262s overhead. Furthermore, note that the generated propagator 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 generated propagator is the best choice for this set of instances, and by a very wide margin.

9 Related Work

There are a variety of algorithms which achieve GAC propagation for arbitrary constraints, for example GAC2001 [4] and GAC-Schema [3]. The major weakness of these and similar algorithms is that their time complexity for propagation is exponential, with a worst case of (at least) d^n. In GAC2001 and GAC-Schema, constraints presented as allowed tuples have the allowed tuples stored as a simple list. There have been a number of attempts to improve these algorithms by using a more suitable data structure to store the allowed tuples. Many have been used, including tries [12], Binary Decision Diagrams [7], Multi-valued Decision Diagrams [6], skip lists [17] and decision trees [15]. 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 $O(dn)$.

Other improvements to GAC table propagators, such as caching and reusing results [16], have also improved average-case performance, but have not removed the worst-case exponential behaviour.

Constraint Handling Rules is a framework for representing constraints and propagation. Apt and Monfroy [1] 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. However, such systems can produce very compact sets of propagation rules for some constraints.

The major difference therefore between these techniques and the algorithm in this paper is that our algorithm provides 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 algorithm, but does not guarantee to achieve GAC in polynomial time.

It is possible that techniques from knowledge compilation [10] (in particular prime implicates) could be usefully applied to propagator compilation. However, the rules encoded in a propagator-tree are not prime implicates — the set of known domain deletions is not necessarily minimal. We do not at present know of a data structure which exploits prime implicates and allows $O(nd)$ traversal.

10 Conclusion

We have presented a novel approach to propagating small constraints. The approach is to generate a custom stateless propagator that enforces GAC in $O(nd)$ time. The tradeoff is that the propagator program can be very large — it scales exponentially in the size of the constraint — therefore generating and compiling it is only feasible up to a certain size.

In three case studies, we demonstrated that the propagator generation approach can be highly efficient, compared to table constraints and decompositions. For example, on Life $n = 7$ $p = 4$, the generated constraint is 18 times faster than a table propagator, and 6.9 times faster than a decomposition. Remarkably, generated propagators can even be faster than hand-optimized propagators. For example, 54% faster than a min constraint on peg solitaire 5.

While surprisingly fast, the generated propagators 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 much scope to improve the scalability of this approach.

Acknowledgements This research is supported by UK EPSRC Grants no.'s EP/H004092/1 and EP/E030394/1.

References

1. Apt, K.R., Monfroy, E.: Constraint programming viewed as rule-based programming. Theory and Practice of Logic Programming 1(6), 713–750 (2001)
2. Bessiere, C.: Handbook of Constraint Programming, chap. Constraint Propagation, pp. 29–83. Elsevier Science Inc., New York, NY, USA (2006)
3. Bessière, C., Régin, J.C.: Arc consistency for general constraint networks: Preliminary results. In: IJCAI(1). pp. 398–404 (1997)
4. Bessière, C., Régin, J.C., Yap, R., Zhang, Y.: An optimal coarse-grained arc consistency algorithm. Artificial Intelligence 165, 165–185 (2005)
5. Bosch, R., Trick, M.: Constraint programming and hybrid formulations for three life designs. Annals of Operations Research 130, 4156 (2004)
6. Cheng, K.C., Yap, R.H.: An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. Constraints 15(2), 265–304 (2010)
7. Cheng, K.C.K., Yap, R.H.C.: Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints. In: Proceeding of the 2006 conference on ECAI 2006. pp. 78–82. IOS Press, Amsterdam, The Netherlands, The Netherlands (2006)
8. Chu, G., Stuckey, P.J., de la Banda, M.G.: Using relaxations in maximum density still life. In: Principles and Practice of Constraint Programming (CP 2009). pp. 258–273 (2009)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (2nd ed.). MIT Press/McGraw-Hill (2001)
10. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of Artificial Intelligence Research 17, 229–264 (2002)
11. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast, scalable, constraint solver. In: Proceedings 17th European Conference on Artificial Intelligence (ECAI 2006). pp. 98–102 (2006)
12. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: AAAI'07: Proceedings of the 22nd national conference on Artificial intelligence. pp. 191–197. AAAI Press (2007)
13. Gent, I.P., Smith, B.M.: Symmetry breaking in constraint programming. In: Horn, W. (ed.) Proceedings of ECAI-2000. pp. 599–603. IOS Press (2000)
14. Jefferson, C., Miguel, A., Miguel, I., Tarim, A.: Modelling and solving english peg solitaire. Computers and Operations Research 33(10), 2935–2959 (2006)
15. Katsirelos, G., Walsh, T.: A compression algorithm for large arity extensional constraints. In: Principles and Practice of Constraint Programming (CP 2007). pp. 379–393 (2007)
16. Lecoutre, C., Hemery, F.: A study of residual supports in arc consistency. In: IJCAI'07: Proceedings of the 20th international joint conference on Artifical intelligence. pp. 125–130. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)
17. Lecoutre, C., Szymanek, R.: Generalized arc consistency for positive table constraints. In: Principles and Practice of Constraint Programming - CP 2006,. pp. 284–298 (2006)
18. Smith, B.M.: A dual graph translation of a problem in 'Life'. In: Principles and Practice of Constraint Programming (CP 2002),. pp. 402–414 (2002)